23/11/10

OOPs

1. Datahiding
2. Abstraction.
3. Encapsulation.
4. Tightly Encapsulated class.
5. Is-A Relationship.
6. Has-A Relationship.
7. Method Signature.
8. Overloading
9. Overriding
10. Methodhiding
11. Static-Control-flow.
12. Instance-Control-flow.
13. Constructors.
14. Coupling.
15. Cohesion.
16. Type-casting.

① Datahiding:

Outside person can't access our data directly. This is called "Datahiding". By using private modifier we can achieve "Datahiding". The main advantage of Datahiding is security.

Ex:
```
class Account
{
  private double balance;
  ---
}
```

Note: Recommended modifier for Datamembers is "private".

② Abstraction:

Highlight the set of services by hiding internal implementation details is called abstraction.

ie; we have to highlight the set of services what we are offering and we have to hide internal implementation details.

By using abstract classes and interfaces we can implement abstraction.

The main advantages of abstraction are

① we can achieve security as we are not highlighting our internal implementation.

② Enhancement will become very easy as without effecting outside person we can able to change our internal implementation.

(3) It improves maintainability of the application.

Ex: ATM.

(3) Encapsulation:

Def 1: Grouping functions and corresponding data into a single capsule is called Encapsulation.

Ex: (1) Every Java class is an Encapsulated component.

(2) Every package is an Encapsulation Mechanism.

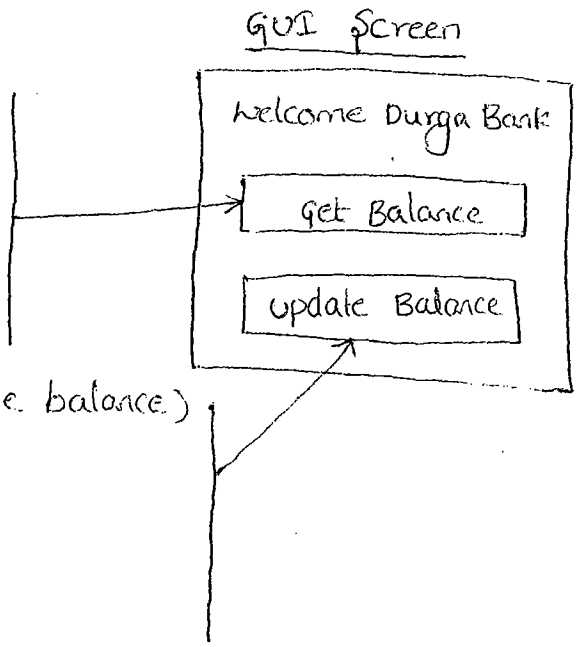
Def 2: If any component follows Datahiding and abstraction is called Encapsulation.

∴ Encapsulation = Datahiding + abstraction

Ex:

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

Encapsulated component



Def-3: Hiding data behind methods is the central concept Encapsulation.

The main advantages of Encapsulation are

1. we can achieve Security.
2. Enhancement will become very easy.
3. Maintainability and modularity will be improved.

The main Limitation of Encapsulation is it increases length of the code and ~~shows~~ slows down execution.

—

4. Tightly Encapsulated class:

A class is said to be Tightly Encapsulated iff every variable declared as the private.

Ex:

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

Q) which of the following are tightly Encapsulated classes.

Tightly Encapsulated class

```
class A
{
  private int x=0;
}
```

Not Tightly Encapsulated class

```
class B extends A
{
  int y=20;
}
```

Tightly Encapsulated class

```
class C extends A
{
  private int z=30;
}
```

Q) which of the following classes are Tightly Encapsulated.

X

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

X

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

X

```
class C extends B
{
  private int z=30;
}
```

Note: If the parent is not tightly encapsulated, then No child class is tightly Encapsulated.

(5) Is-A-Relationship:

(1) Is-A-Relationship also known as Inheritance.

(2) By using extends keyword we can implement Is-A-Relationship.

(3) The main advantage of Is-A-Relationship is Reusuability of the code.

Ex:
```
class P
{
  m1()
  {
  }
}
class C extends P
{
  m2()
  {
  }
}
class Test
{
public static void main (String[] args)
{
```

(1) `P p=new P();`

`p.m1();` ✓

`p.m2();` X → CE: cannot find symbol.
symbol: method m2()
location: class P.

(2) `C c=new C();`

`c.m1();` ✓

`c.m2();` ✓

③ P p = new C();

p.m1(); ✓

p.m2(); X → C.E: cannot find symbol.

symbol: method m2()

location: class P

④ C c = new P(); → C.E: incompatible types.

found: P

required: C

}

}

Conclusions:

① whatever the parent has by default available to the child. Hence parent class methods we can call on the child class objects.

② whatever the child has by default not available to the parent Hence child specific methods we can't call on the parent referen

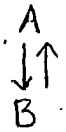
③ parent reference can be used to hold child class object. But by using that reference we have to call only parent specific methods. i.e; By using parent reference we can't call child Specific methods.

④ child reference can't be used to hold parent class object.

Multiple inheritance is not allowed in Java.

Ex:
```
class A
{
}
class B
{
}
class C extends A,B
{
}
```
C.E: ✓

Cyclic inheritance is not allowed in Java.

Ex:
```
class B extends A
{
}
class A extends B
{
}
```


C.E: Cyclic inheritance involving A

Ex:
```
class A extends A
{
}
```
C.E: Cyclic inheritance involving A

Note: Multiple inheritance is not possible in Java, but through interfaces we can implement.

A class can't extend more than one class at a time. whereas an interface can extend any no. of interfaces simultaneously.

24/11

(6) Has-A-Relationship:

→ Has-A-Relationship also known as comparision (or) agaregation.

→ There is no specific keyword to implement this, but mostly we are using new keyword.

→ The main advantage of Has-A-Relationship is Reusuability.

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

"Class Car has Engine reference"

→ The main advantage of Has-A-Relationship is Reusuability. whereas its main limitation is, It increases depency between the componets and creates maintainance problems.

(7) Method Signature:

→ Method signature consists of name of the method and argument types.

Ex: public void m1(float f, char ch)

Its Method signature is **m1(float, char)**

→ In java returntype is not part of method signature. Compiler will always use method signature while resolving method calls.

→ within the same class two methods with the same signature not allowed. Otherwise, we will get compiletime Error.

Ex:
```
class Test
{
 public void m1(int i)
{
}
 public int m1(int k)
{
 return 10;
}
 public static void main(String[] args)
{
 Test t=new Test();
 t.m1(10);
}
}
```

→ CE: m1(int) is already defined in Test.

—

(8) Overloading:

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

→ In 'C' language we can't take two methods with same name but different arguments.

Ex: abs() for int type → abs();

labs() for long type → labs(10L);

sabs() for short type → sabs(10);

⋮

→ Hence Lack of overloading in 'C', increases complexity of the programming. But in Java two methods with sam name and different arguments is allowed and these method are considered as overloaded methods.

Ex: we can use abs() method for int type, long type, float and double type.

abs() → abs(10); ✓

abs(10.5); ✓

abs(10L); ✓

Having overloading concept in Java makes programming simple.

Ex: class Test

overloading methods

```
{
 public void m1()
 {
  S.O.P("no-arg");
 }
 public void m1(int i)
 {
  S.O.P("int-arg");
 }
 public void m1(double d)
 {
  S.O.P("double-arg");
 }
 public static void main(String[] args)
 {
  Test t=new Test();
  t.m1(10); int-arg.
  t.m1(); no-arg.
  t.m1(10.5); double-arg.
 }
}
```

In overloading method resolution always takes care by compiler based on reference type. Hence overloading is also considered as static polymorphism (or) compiletime polymorphism (or) earlybinding.

Case(1): Automatic promotion in Overloading:

While performing overloading method resolution if there is no method with specified argument type compiler won't raise any error immediately.

First compiler promotes that argument to the next level and checked for matched method.

If the match is available, then it will be considered. Otherwise, once again promotes that argument to the next level until all possible promotion, still matched method is not available then only compiler raises error.

This promotion of argument type is called automatic promotion in overloading.

The following are various possible automatic promotion in java.

byte → short → int → long → float → double

char ↗

Ex: class Test

```
{
  public void m1(int i)
  {
   s.o.pln("int-arg");
  }
  public void m1(float f)
  {
   s.o.pln("float-arg");
  }
  public static void main(String[] args)
  {
   Test t=new Test();
   t.m1(10); int-arg.
   t.m1(10.5f); float-arg.
   t.m1('a'); int-arg.
   t.m1(101); float-arg.
   t.m1(10.5); → CE: cannot find symbol.
                  symbol: method m1(double)
                  location: class Test.
  }
}
```

overloaded methods

=

Case (ii):

Ex:

```
class Test
{
    public void m1(Object o)
    {
        s.o.pln("Object version");
    }
    public void m1(String s)
    {
        s.o.pln("String version");
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1(new Object()); → Object version.
        t.m1("durga"); → String version.
        t.m1(null); → String version.
    }
}
```

Overloaded methods

Object
↑
String

In overloading method resolution child will get high priority when compared with parent.

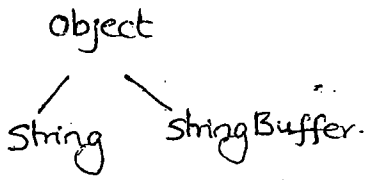
25/11/10

Case (iii):

Ex:

```
class Test
{
 public void m1(String s)
 {
  S.O.pln("String version");
 }
 public void m1(StringBuffer sb)
 {
  S.O.pln("StringBuffer version");
 }
 public static void main(String[] args)
 {
  Test t=new Test();
  t.m1("durga");  → String version.
  t.m1(new StringBuffer("durga"));  → StringBuffer version.
  t.m1(null);  → C.E: reference to m1() is ambiguous.
  String s=null;
  t.m1(s);  → String version.
  StringBuffer sb=null;
  t.m1(sb);  → StringBuffer version.
 }
}
```

overloaded methods



Ex: class Test

```
{
 public void m1(int i, float f)
 {
  S.o.pln("int-float version");
 }
 public void m1(float f, int i)
 {
  S.o.pln("float-int version");
 }
 public static void main(String[] args)
 {
  Test t=new Test();
  t.m1(10,10.5f); → int-float version.
  t.m1(10.5f,10); → float-int version.
  t.m1(10,10); ⟶ C.E: "reference to m1() is ambiguous"
  t.m1(10.5f,10.5f); → C.E: cannot find symbol.
                       Symbol: method m1(f,f)
                       Location: class Test
 }
}
```

Overloaded methods

Ex:

```
class Animal
{
}
class Monkey extends Animal
{
}
class Test
{
  public void m1(Animal a)
  {
    S.o.pln("Animal version");
  }
  public void m1(Monkey m)
  {
    S.o.pln("Monkey version");
  }
  public static void main(String[] args)
  {
    Test t=new Test();
```

Overloaded methods

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 the reference type.

In overloading runtime object never play any role.

(9) Overriding:

Whatever the parent has by default available to the child. If the child is not satisfied with parent implementation. Then child class has flexibility to redefine based on its specific required way. This process is called overriding and the parent class method which is overriden is called "overriden method" and child class method which is overriding is called "overriding method".

Ex:

```
class P
{
  public void marry()
  {
    S.o.pln("Subbalaxmi");
  }
}

class C extends P
{
  public void marry()
  {
    S.o.pln("kajal / Tapsy / samantha");
  }
}

class Test
{
  public static void main(String[] args)
  {
    P p=new P();
    p.marry(); → Parent method

    C c=new C();
    c.marry(); → child method
```

(overriden method: `public void marry() { S.o.pln("Subbalaxmi"); }` — overriding method: `public void marry() { S.o.pln("kajal / Tapsy / samantha"); }`)

```
P p = new C();
        ↳ Runtime Object

p.marry(); → child Method.
  }
}
```

In overriding, method resolution always takes care by JVM based on Runtime object. Hence overriding is also considered as Runtime-polymorphism (or) Dynamic polymorphism (or) Latebinding.

Overriding method resolution also known as "Dynamic method-dispatch".

Note: In overloading reference place very important role whereas In overriding runtime object place the role.

Rules to follow while overriding:

1. In overriding method names and arguments must be matched i.e; Method signatures should be same.
2. While overriding the return types must be same. This rule is applicable until 1.4 version. But from 1.5 version onwards Co-varient return types are allowed. According to this child method return type need not be same as parent method return type its child is also allowed.

Ex: ① object ✓

↓

Object | String | StringBuffer | Integer ---

✓ (2) Number → Integer | x (3) String → Object | x (4) double → int

Note: Co-varient returntype concept is not applicable for primitive types.

Ex:
```
class P
{
 public Object m1()
 {
  return null;
 }
}
 class C extends P
{
 public String m1()
{
 return null;
}
}
```

> javac P.java ✓

> javac -source 1.4 P.java x

C.E: m1() in C cannot override m1() in P; attempting to use incompatible return type.

found: java.lang.String

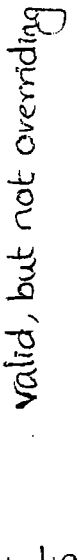
required: java.lang.Object

—

(3) private methods are not visible in child classes, Hence overriding concept is not applicable for private methods.

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

Ex:
```
class P
{
  private void m1()
  {
  }
}
class C extends P
{
  private void m1()
  {
  }
}
```



26/11/10

(4) Parent class final methods cannot be overriden in child classes.

But a non-final methods can be overriden as final.

Ex:
```
class P
{
  public final void m1()
  {
  }
}
class C extends P
{
  public void m1()
  {
  }
}
```

C.E: m1() in 'C' cannot override m1() in P; overriden method is final.

Ex:
```
class P
{
 public void m1()
 {
 }
}
 class C extends P
 {
  public final void m1()
  {
  }
 }
```
valid

(5) We should override Parent class abstract methods in child classe to provide implementation.

Ex:
```
abstract class P
 {
  public abstract void m1();
 }
 class C extends P
 {
  public final void m1(){ }
 }
```

(6) A non-abstract method can be overriden as abstract to stop availability of parent class method implementation to the child classes.

Ex: class P

```
{
 public void m1()
{
}
}
abstract class C extends P
{
 public abstract void m1();
}
```

The following modifiers won't keep any restrictions in overriding.

① Synchronized

② native

③ strictfp

Summary:

final

X ↓

non-final

non-final

✓ ↓

final

abstract

✓ ↓ ↑

non-abstract

synchronized

✓ ↓ ↑

non-synchronized

native

✓ ↓ ↑

non-native

strictfp

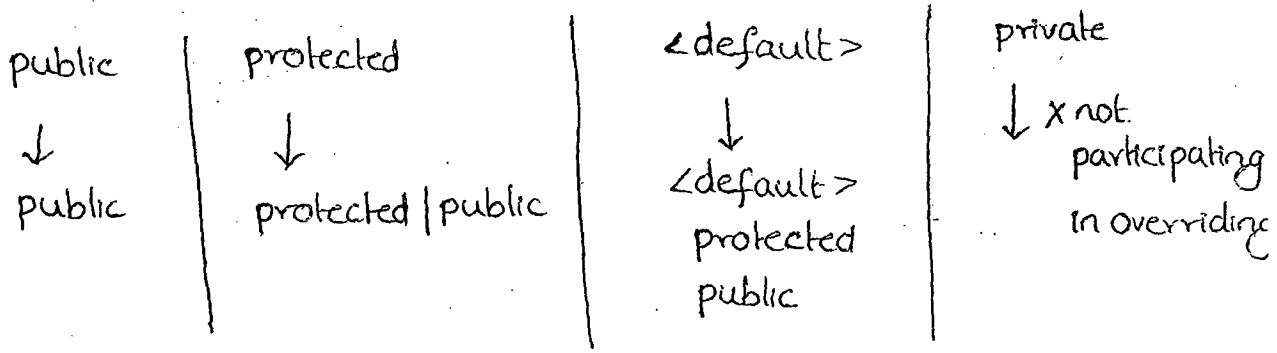
✓ ↓ ↑

non-strictfp

while overriding weakering accen modifiers are not allowed. But we can increase.

Ex:
```
class P
{
 public void m1()
 {
 }
}
class C extends P
{
 void m1()
 {
 }
}
```

C.E: m1() in C cannot override m1() in P; attempting to assign weaker access privileges; was public.



private < default < protected < public.

This rule is applicable even while implementing interface method also. whenever we are implementing interface methods compulsary it should be declared as public. Because every interface method is always public by default.

Ex:
```
interface X
{
 void m1();
}
class P implements X
{
  void m1()
  {
  }
}
```

C.E: m1() in P cannot implement in X; attempting to assign weaker access privileges; was public.

If m1() in P declared as public then we won't get any Compiletime Error.

while overriding the size and Level of checked Exceptions we can't increase. But decreasing is allowed.

There are no restrictions for Unchecked Exceptions. Increasing and decreasing and both are allowed.

✓ (1) Parent: public void m1() throws Exception.
Child : public void m1() throws IOException.

✗ (2) Parent: public void m1() throws IOException.
child : public void m1() throws Exception.

✓ (3) Parent: Public void m1() throws Exception.
Child : Public void m1()

✗ (4) Parent: Public void m1()

child : Public void m1() throws Exception.

✓ (5) Parent: Public void m1() throws IOException.

child : Public void m1() throws FileNotFoundException, EOFExceptio

✗ (6) Parent: Public void m1() throws IOException.

child : Public void m1() throws InterruptedException.

✓ (7) Parent: Public void m1()

child : Public void m1() throws ArithmeticException, NullPointerExcep

Note: while overriding decreasing acces modifiers is not allowed But increasing is allowed. while overriding increasing the size and level of checked exceptions is not allowed but decreasing is allowed.

Overriding w.r.to. static methods:

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

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

CE: m1() in C cannot override m1() in P; overriden method is stati

(2) Similarly we can't override a non-static method as static.

(3) If both Parent and child class methods are static then we won't get any compiletime Error. It seems to be overriding is possible but it is not overriding, it is method hiding.

Method hiding:

It is exactly same as overriding. Except the following differences.

Overriding	Method hiding
(1) Both Parent & child methods are non-static	(1) Both Parent & child methods are static.
(2) Method resolution is always takes care by JVM based on Runtime object.	(2) Method resolution always takes care by Compiler based on reference type.
(3) It is considered as Runtime polymorphism (or) dynamic polymorphism (or) late binding.	(3) It is considered as static polymorphism (or) compiletime polymorphism (or) (or) early binding.

Ex: class P

```
{
 public static void m1()
 {
  S.o.pln("parent");
 }
 class C extends P
{
 public static void m1()
 {
  S.o.p("child");
 }
}
 class Test
{
 public static void main(String[] args)
 {
    P p = new P();
    p.m1(); → Parent
    C c = new C();
    c.m1(); → child
    P p1 = new C();
    p1.m1(); → Parent
 }
}
```

If both methods are not static then, it will become overriding and method resolution should be based on the Runtime Object. Hence in this case the output is: parent
child
child.

27/11/10

Overriding w.r.to. var-arg methods:

We can override a var-arg method with var-arg method only. If we are trying to override a var-arg method with normal method. Then it will become overloading but not overriding.

Ex:

```
class P
{
public void m1(int... i)
{
 S.o.pln("parent");
}
}

class C extends P
{
 public void m1(int i)
{
 S.o.pln("child");
}
}

class Test
{
public static void main(String[] args)
{
 P p = new P();
 p.m1(10); → Parent
 C c = new C();
 c.m1(10); → child.
 P p1 = new C();
 p1.m1(10); → Parent.
}
```

If we are declaring child class method also as var-arg. Then it will become overriding. Hence in this case the output is

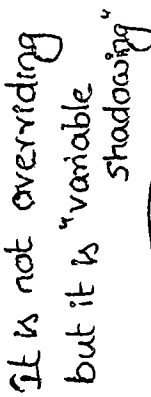
parent
child
child

---

overriding w.r.to. variables:

→ overriding concept is not applicable for variables.

→ variable resolution should be done by the compiler based on the reference type.



Ex:
```
class P
{
  int x=888;
}
class C extends P
{
  int x=999;
}
class Test
{
  public static void main(String[] args)
  {
    P p=new P();
    S.O.P(p.x); ⟶ 888
    C c=new C();
    S.O.P(c.x); ⟶ 999
    P p1=new C();
    S.O.P(p1.x); ⟶ 888
  }
}
```

→ If both variables declared as the static (Eventhough) there is no change in the output.

Difference between Overloading & Overriding:

property	overloading	overriding
1) Method Names	must be same	must be same
2) Arguments	must be different (atleast order)	must be same (including order)
3) Return types	No restrictions.	must be same until 1.4 version. But from 1.5 version onwards Co-varient returntypes also allowed.
4) Access Modifiers	No restrictions	Weakering is not allowed.
5) Throws clause	No restrictions	The size and level of checked Exceptions are not allowed to increase. But we can decrease. No restrictions for unchecked Exceptions.
6) private, static, final methods.	can be overloaded.	cannot be overriden.
7) Method resolution	always takes care by compiler based on reference type.	always takes care by JVM based on Runtime object.
8) also known as	compiletime polymorphism (or) static polymorphism (or) early binding.	Runtime polymorphism (or) Dynamic polymorphism (or) late binding.

Q) Consider the method declaration

public int m1(int i) throws IOException.

In child classes which methods we are allowed to declare.

X (i) public int m1(int i) throws Exception

→ Increasing the level of checked Exceptions.

✓ (ii) public void m1(long i) throws Exception.

→ overloading.

X (iii) public static int m1(int i) throws IOException.

→ non-static not possible to static

✓ (iv) public static final void m1() throws InterruptedException

→ overloading

X (v) private abstract int m1(int i)

→ weakering

X (vi) public abstract synchronized int m1(int i) throws IOException.

→ synchronized abstract illegal combination.

✓ (vii) public native int m1(int i) throws ArithmeticException, NullPointerException, ClassCastException.

→ by overriding unchecked Exceptions.

Note: If any method call executed by polymorphism (overloading (or) overriding (or) method hiding) such type of method calls are called polymorphic method calls.

polymorphism:

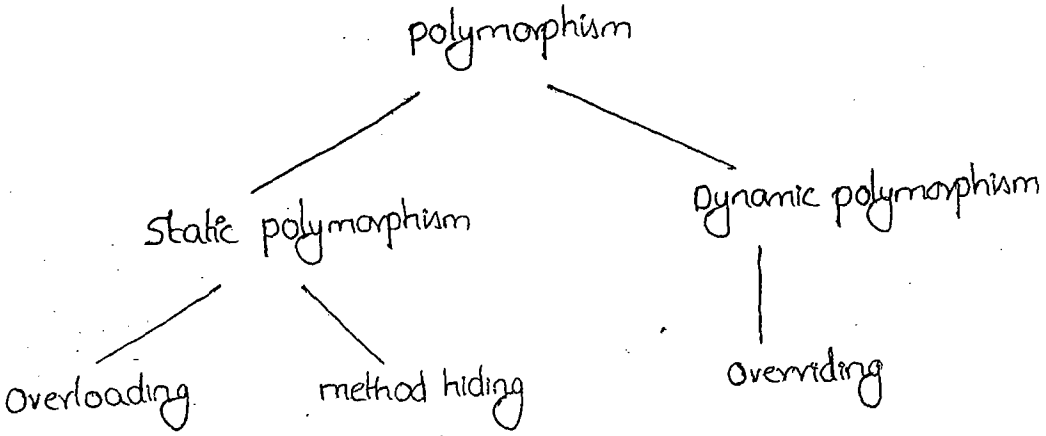
One name with multiple forms.

Ex: abs(int)

abs(long)

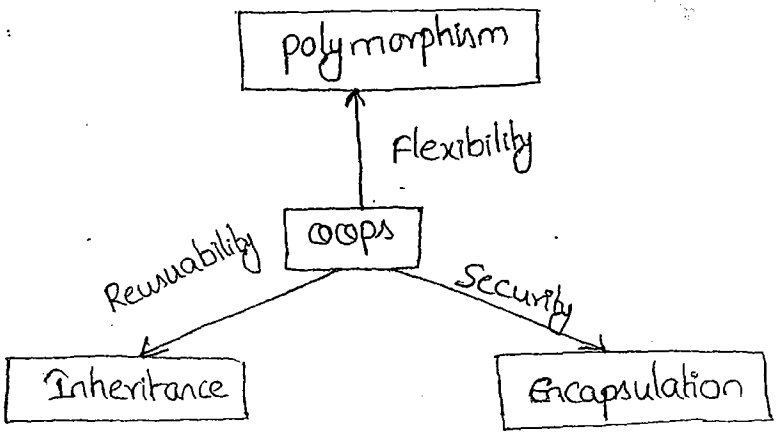
abs(double)

There are two types of polymorphism:



Three pillers of oops:

1. Inheritance
2. polymorphism
3. Encapsulation



Note: In overriding we have to consider several things like modifier return type, signature, throws clause --- e.t.c.

whereas in overloading we have to consider ~~several things~~ like only method name & arguments. Method name should be same, whereas arguments should be different. All the remaining things are not required to check in overloading.

---

(11) Static-control-flow:

```
class Base
{
(1) static int i=10; (7)
(2) static
    {
    m1(); (8)
    S.o.pln("FSB"); (10)
    }
(3) public static void main(String[] args)
    {
    m1(); (13)
    S.o.pln("main method"); (15)
    }
(4) public static void m1()
    {
    S.o.pln(j); (9),(14)
    }
```

⑤ static

{

so.println("SSB"); ⑪

}

⑥ static int j=20; ⑫

}

Process:

Whenever we are executing a Java class the following sequence of actions will be performed.

① Identification of static members from top to bottom [1 to 6].

② Execution of static variable assignments & static blocks from top to bottom: [7 to 12].

③ Execution of main method. [13 to 15].

Java Base ↵

Output: 0
FSB
SSB
20
main method

[i=0 (RIWO) → Read indirectly
j=0 (RIWO) write only.
i=10 (RKW)
j=20 (RKW)]

Read Indirectly write only (RIWO):

If a variable in ReadIndirectlywriteonly state then we are not allowed to perform Read operation directly. Otherwise, we will get compiletime Error saying "Illegal forward reference."

Ex: class Test

```
{
① static int i =10; ③
② static
  {
④ S.O.ph(i);
  }
}
```

i=0 (RIWO)
i=10 (R&W)

Output: 10

RuntimeException: NoSuchMethodError: main.

Ex: class Test

```
{
① static
  {
   S.O.p(i); ——→ i=0 (RIWO)
  }
② static int i=10;
}
```

C.E: Illegal Forward reference.

Static block:

At the time of class loading if we want to perform any activity then we have to define that activity within the static block.

Because static blocks will be executed at the time of class loading.

Within a class we can take any no.of static blocks but all these blocks will be executed from top to bottom.

Ex:①

Native libraries should be loaded at the time of class loading, hence we have to define this activity within the static block.

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

```
Ex:  public class object
     {
      private static native void registerNatives();
      static
      {
      registerNatives();
      }
     }
```

Ex:2:

In every JDBC Driver class there is a static block to register Driver with DriverManager. Hence while loading Driver class automatically Registering with DriverManager will be performed. Because of this we are not required to register Driver class explicitly with DriverManager.

```
Ex:  class Driver
     {
      static
      {
       Register this Driver with DriverManager
      }
     }
```

Q) without using main() method, Is it possible to print some stmt to the console.

Ans Yes, by using static block.

Ex:
```
class Test
{
 static
 {
  S.O.P("Hello we can print");
  System.exit(0);
 }
}
```

Output: Hello we can print.

Q) without using main method & static block, Is it possible to print some statements to the console?

Ans Yes.

Ex:
```
class Test
{
 static int i=m1();
 public static int m1()
 {
  S.O.P("Hello we can print");
  System.exit(0);
  return 10;
 }
}
```

Output: Hello we can print.

Ex: class Test

```
{
 static Test t=new Test();
 Test()
 {
  S.O.p("Hello we can print");
  System.exit(0);
 }
}
```

Ex: class Test

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

Q) Without using S.O.pln, Is it possible to print some statements to the console.

Ans Yes, By simulating functionality of S.O.p with our own classes.

~~Static~~ control

Static control flow in parent and child classes:

Ex: class Base

```
{
(1) static int i = 10; (12)   i=0 (RIWO)
(2) static
    {
     m1(); (13)
     S.O.pln("BSB"); (15)
    }
(3) public static void main(String[] args)
    {
     m1();
     S.O.pln("Base main method");
    }
(4) public static void m1()
    {
     S.O.P(j); (14)
    }
(5) static int j = 20; (16)   j=0 (RIWO)
}
```

class Derived extends Base

```
{
(6) static int x=100; (17)   x=0 (RIWO)
(7) static
    {
     m2(); (18)
     S.O.p("DFSB"); (20)
    }
(8) public static void main (String[] args)
    {
     m2(); (23)
     S.O.p("Derived main"); (25)
    }
```

```
(9) public static void m2()
{
    S.o.p(y); (19), (24)
}
(10) static
{
    S.o.pln("DSSB"); (21)
}
(11) static int y = 200; (22)   y=0 (RIWO)
}
```

```
i=0 (RIWO)
j=0 (RIWO)
x=0 (RIWO)
y=0 (RIWO)
i=10 (RKW) (12)
j=20 (RKW) (16)
x=100 (RKW) (17)
y=200 (RKW) (22)
```

> javac Derived.java

Base.class Derived.class.

> java Derived ↵

Process:

(1) Identification of static members from parent to child. [1 to 11].

(2) Execution of static variable assignments & static blocks from parent to child [12 to 22]

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

> Java Derived ↵

output: 0
BSB
0
DFSB
DSSB
200
Derived main.

> java Base

output: 0
BSB
20
Base main method

Note: ① whenever we are loading child class then automatically parent classes will be loaded.

② whereas whenever we are loading parent class, child classes won't be loaded.

---

(12) Instance-control-flow:

Ex:

```
class Parent
{
(3) int i=10; (9)
(4) {
    m1(); (10)
    S.O.P ("FIB"); (12)
}
(5) Parent()
{
    S.O.P ("constructor"); (15)
}
(1) public static void main(String[] args)
{
(2) Parent p=new Parent(); 
    S.O.P ("main method"); (16)
}
(6) public void m1()
{
    S.O.P(j); (11)
}
(7) {
    S.O.Pln ("SIB"); (13)
}
(8) int j=20; (14)
}
```

i=0 (RIWO)
j=0 (RIWO)
i=10 (RKW) (9)
j=20 (RKW) (14)

process:

whenever we are creating an object the following sequence of events will be executed automatically.

① Identification of instance members from top to bottom.

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

③ Execution of constructor.

Output: 0
FIB
SIB
Constructor
Main method.

Note: ① static-control-flow is one-time activity which should be executed at the time of class loading.

② whereas instance-control flow is not one-time activity. It will be executed for every object creation.

---

instance control flow in Parent to child:

```
class Parent
{
 int i=10;
 {
  m1();
  S.O.P("IIB");
 }
  Parent()
  {
   S.O.P("parent constructor");
  }
```

```
public static void main(String[] args)
{
 Parent p=new Parent();
 S.O.pln("Parent main");
}
public void m1()
{
 S.O.p(j);
}
int j=20;
}
class child extends Parent
{
 int x=100;
 {
  m2();
  S.O.p("CFIB");
 }
 child()
 {
  S.O.p("child constructor");
 }
 public static void main(String[] args)
 {
  child c=new child();
  S.O.p("child main");
 }
  public void m2()
  {
   S.O.p(y);
  }
  {
   S.O.p("CSIB");
  }
   int y=200;
 }
>java child↵
```

Process:

whenever we are creating child class objects the following sequence of events will be executed automatically.

(1) Identification of instance members from Parent to child.

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

(3) Execution of Parent class constructor.

(4) Execution of instance variable assignments & instance blocks in child class.

(5) Execution of child class constructor.

Output:
```
0
PIB
Parent Constructor.
0
CFIB
CSIB
Child Constructor
Child main.
```

Note: (1) The most costly operation in java is object creation. Hence if there is no specific requirement, then it is never recommended to create object.

(2) If any method implementation not related to any object. Then that method compulsary should be declared as static.

(3) Over instance methods static methods are recommended to use.

Note: We can't access non-static members directly from static area because while executing that static area JVM may not identify that instance member.

Ex:
```
class Test
{
    int i=10;
    public static void main(String[] args)
    {
        S.o.pln(i);
    }
}
```
→ CE: Non-static variable 'i' cannot be referenced from static context.

Ex:
```
public class Initialization
{
    private static String m1(String msg)
    {
        S.o.p(msg);
        return msg;
    }
    public Initialization()
    {
        m=m1("1");
    }
    {
        m=m1("2");
    }
    String m=m1("3");
    public static void main(String[] args)
    {
        Object obj=new Initialization();
    }
}
```

m=null (RIWO) ⑥
m=2 ⑦
m=3 ⑧
m=1 ⑨

Output: 2
3
1

Ex: public class Initialization2

```
{
 private static String m1(String msg)
 {
  S.o.ph(msg);
  return msg;
 }
  static String m=m1("1");
  {
   m=m1("2");
  }
  static
  {
   m=m1("3");
  }
  public static void main(String[] args)
  {
   Object obj=new Initialization2();
  }
}.
```

Output: 1
3
2

(13) Constructors:

Object creation is not enough compulsary we should perform initialization. Then only that object is in a position to provide response properly.

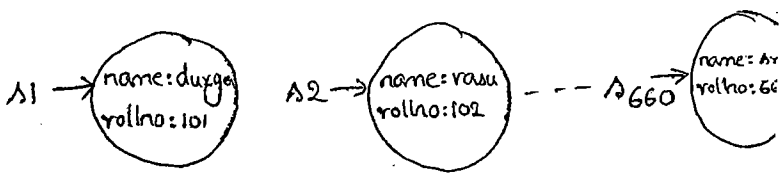
whenever we are creating an object some piece of code will be executed automatically to perform initialization. This piece of code is nothing but constructor. Hence the main objective of constructor is to perform initialization for the object.

Ex:

```
class Student
{
 String name;
 int rollno;

 Student (String name, int rollno)
 {
  this.name=name;
  this.rollno=rollno;
 }
 public static void main(String[] args)
 {
  Student s1= new student ("durga", 101);
  Student s2= new student ("vasu", 102);
  :
  Student s660= new Student ("sri", 660);
 }
}
```

Constructor



Constructor vs instance block:

The main purpose of constructor is to perform initialization for the object.

Otherthan initialization if we want to perform any activity for every object, then we should go for instance block.

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

Ex:

```
class Test
{
 static int count=0;
 Test()
 {
  count++;
 }
 public static void main(String[] args)
 {
 S.O.p("The no.of objects created:"+count); 0
 Test t1 = new Test();
 Test t2 = new Test();
 S.O.p("The no.of objects created:"+count); 2
 }
}
```

Not recommended: we are misusing constructor

Ex: class Test

```
{
 static int count=0;
{
 count++;  → It is the best use of instance block.
}
 public static void main(String[] args)
{
 S.o.p("The no.of objects created:"+count);
 Test t1=new Test();
 Test t2=new Test();
 S.o.p("The no.of objects created:"+count.);
}
}
```

Rules for defining constructor:

① The name of the class, name of the constructor must be same.

② Returntype concept is not applicable for constructor, even void also. By mistake if we are declaring returntype for the constructor then we won't get any Compiletime (or) RuntimeError, it is simply treated as a method.

Ex: class Test

```
{
  void Test()
  {
   S.o.p("constructor");
  }
}
```

→ treated as a method, but not constructor.

Hence it is legal (but stupid) to have a method whose name is exactly same as class name.

③ The only applicable modifiers for constructors are

public
default
protected
private

If we are using any other modifier, we will get CE saying modifier XXX not allowed here.

Ex:
```
class Test
{
  private Test()
  {
  }
}
```

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

Singleton class: (By private constructor):

For any java class if we are allowed to create only one object. such type of classes are called Singleton classes.

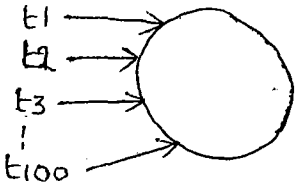
Ex: ① Runtime class
② ActionServlet (Structs 1.X)
③ BusinessDelegate } (EJB)
④ ServiceLocator }

Creation of our own Singleton classes:

We can create our own singleton classes also. By using private constructor, static variable, static method we can implement Singleton classes.

Ex:
```
public class Test implements Cloneable
{
  private static Test t;
  private Test()
  {
  }
  public static Test getInstance()
  {
   if(t==null)
   t=new Test();
   return t;
  }
  public Object clone()
  {
   return this;
  }
}
```

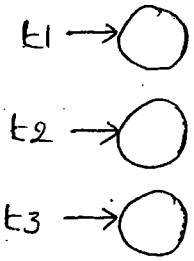
```
Test t1 = Test.getInstance();
Test t2 = Test.getInstance();
Test t3 = Test.getInstance();
  :
Test t100 = Test.getInstance();
```



case(I): If constructor is not private:

Then outside person can create object directly by calling the constructor. In that case he can create multiple objects also and we will miss Singleton nature.

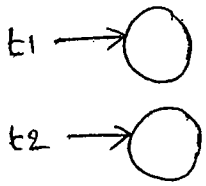
```
Test t1 = new Test();
Test t2 = new Test();
Test t3 = new Test();
```



Case(II): If 't' is not private:

Then after creation of first object outside person can reassign 't' with null. In that case a second new object will be created, whenever we call getInstance() method again.

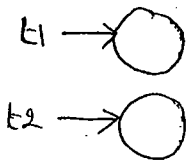
```
Test t1 = Test.getInstance();
Test.t = null;
Test t2 = Test.getInstance();
```



Case(III): If we are not overriding clone() method & Test class implements Cloneable:

Then object class clone() method will be executed which provides always a seperate new object.

```
Test t1 = Test.getInstance();
Test t2 = (Test) t1.clone();
```



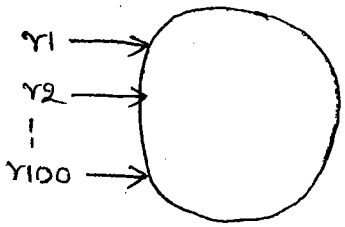
Ex: Runtime r1 = Runtime.getRuntime();

↳ Factory method.

Runtime r2 = Runtime.getRuntime();

⋮

Runtime r100 = Runtime.getRuntime();



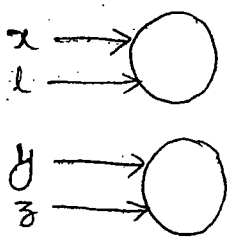
S.O.P (r1 == r2); true

→ How to create our own doubleton classes:

Ex:
```
public class Test implements cloneable
{
 private static Test t1;
 private static Test t2;
 private Test()
{
}
 public static Test getInstance()
{
 if(t1==null)
 {
  t1=new Test();
  return t1;
 }
 elseif (t2==null)
{
 t2=new Test();
 return t2;
}
 return t1/t2;
}
```

```
    else
    {
      int c = (int) (Math.random()+0.5);
      if (c==0)
        return t1;
      else
        return t2;
    }
    public Object clone()
    {
     return this;
    }
}
```

```
Test x = Test.getInstance();
Test y = Test.getInstance();
Test z = Test.getInstance();
Test t = Test.getInstance();
```



Note: We can create tripleton, ---tenton classes also (we can create any Xxxton classes also).

Advantage of Singleton classes:

Instead of creating multiple objects we can run entire show with only one object. but with multiple references. Hence the main advantage of Singleton class is performance will be improved.

Default Constructor:

Every java class including abstract class contains constructor concept. If we are not writing any constructor then compiler will generate default constructor.

If we are writing atleast one constructor then compiler won't generate default constructor.

Hence every java class can contain either programmer written constructor (or) compiler generated constructor but not both simultaneously.

Prototype of default constructor: (Structure):

① It is always no-arg constructor.

② The access modifier of the default constructor is same as class modifier. (but it is applicable only for public & default).

③ It contains only one line i.e; `super();`

It is a no-arg call to super class constructor.

Summary:

programmer code

```
① class Test
  {
  }
```

Compiler generated code

```
① class Test
  {
    Test()
    {
      super();
    }
  }
```

programmer code	compiler generated code

② public final class Test
{
}
 	
	② public final class Test
	{
	public Test()
	{
	super();
	}
	}

③ class Test
{
void Test()
{
}
}
	③ class Test
	{
	Test()
	{
	super();
	}
	void Test()
	{
	}
	}

④ class Test
{
Test(int i)
{
}
}
	④ class Test
	{
	Test(int i)
	{
	super();
	}
	}

⑤ class Test
{
Test()
{
this(10);
}
Test(int i)
{
}
}
	⑤ class Test
	{
	Test()
	{
	this(10);
	}
	Test(int i)
	{
	super();
	}
	}

Programmer code	Compiler generated code

⑥ class Test
{
Test()
{
super();
}
}
	⑥ class Test
	{
	Test()
	{super();
	}
	}

The first line inside every constructor should be super() (or) this(), if we are not writing anything then compiler will always place super() keyword.

Case(1): within the constructor we have to use super() (or) this() in the first line only. If we are using anywhere else we will get compiletime Error.

Ex: class Test
```
{
 Test()
 {
  S.O.ph("constructor");
  super();
 }
}
```
C.E: "call to super() must be first statement in constructor".

Ex: class Test
```
{Test()
 {
  S.O.P("Hello");
  this(10);
 }
 Test(int i)
 {
 }
}
```
C.E: "call to this() must be first Statement in constructor".

Case (ii):

Ex: class Test
{
Test()
{
super();
this();
S.O.p("constructor");
}
}

C.E: call to this() must be first statement in constructor.

Conclusion: We can use either super() (or) this(), but not both simultaneously.

Case (iii):

We can call constructors directly by super() and this() only inside constructor. i.e; We can't use these direct constructor calls from outside of the constructor.

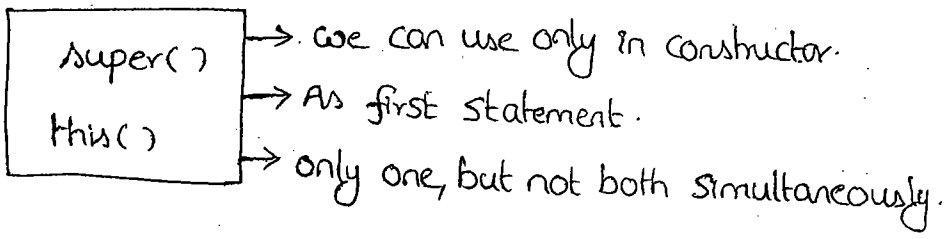
Ex: class Test
{
Test()
{
super();
}
public void m1()
{
super();
}
}

C.E: call to super() must be first statement in constructor.

Note:

Only inside constructors { super( ) → A call to super class constructor.
this ( ) → A call to current class constructor.

We can use anywhere except in static area. { super → reference to parent
this → reference to current class object.



Constructor overloading:

Within a class we can take multiple constructors and all these constructors are considered as overloaded constructors. Hence Construct overloading is possible.

Ex:

```
class Test
{
    Test()
    {
        this(10);
        S.o.pln("no-arg");
    }
    Test(int i)
    {
        this(10.5);
        S.o.pln("int-arg");
    }
    Test(double d)
    {
        S.o.pln("double-arg");
    }
```

overloaded constructors.

```
public static void main(String[] args)
{
  Test t1 = new Test();  ⟶ Double-arg.
                             Int-arg
                             no-arg.

  Test t2 = new Test(10); ⟶ Double-arg
                             Int-arg

  Test t3 = new Test(10.5); ⟶ Double-arg.
 }
}
```

---

* ⇒ Inheritance & overriding concepts are not applicable to the Constructors. But overloading concept is applicable.

* ⇒ Every class in java including abstract class also contain ~~constru~~ constructor concept. But interfaces cannot have constructors.

Ex: class Test

```
{
 Test()
 {
 }
}
```

valid

Ex: abstract class Test

```
{
 Test()
 {
 }
}
```

valid

Ex: interface Test

```
{
 Test()
 {
 }
}
```

Invalid

Ex: enum Test

```
{
 Test()
 {
 }
}
```

valid

Q) We can't create an object for abstract class, but abstract class can contain constructor. What is the need?

Ans To perform initialization for the parent class (abstract class) instance members at parent level only. For the child class object i.e; abstract class constructors will be executed to perform initialization of child class object.

Ex:

```
abstract class Person
{
 name;
 age;
 height;
}
 person (name, age, height)
 {
  this.name=name;
  this.age =age;
 this.height = height;
}
```

```
class SoftwareEngineer extends Person
{
 super(name, age, height)
}
```

```
class Student extends Person
{
 super (name, age, height)
}
```

Ex:
```
class P
{
 public static void m1()
 {
 }
}
class C extends P
{
 public static void m2()
 {
 }
// public static void main(String[] args)
 {
   C c=new C();
     c.m1();
     c.m2();
 }
}
```

Ex:
```
class P
{
  P()
  {
  }
}
class C extends P
{
 C(int i)
 {
 super();
 }
public static void main(String[] args)
{
 C c=new C(10);
 C c=new C(); → C.E:
```
C.E: cannot find symbol.
Symbol: constructor C().
Location: class C:

Case (1):

"Recursive method call" is always a "RuntimeException". whereas "Recursive constructor" invocation is a compiletime Error.

Recursive method call

```
class Test
{public static void m1()
 { m2();
 }
 public static void m2()
{ m1();
 }
 public static void main(String[] args)
{ m1();
 S.O.pln("Hello");
}
```

m2()
m1()
m2()
m1()
main()

Recursive constructor invocation.

```
class Test
{Test()
 {this(10);
 }
 Test(int i)
 {this();
 }
public static void main (String[] args)
{S.O.pln("Hello dont get shock");
}
```
C.E: recursive constructor invocation

Case (ii):

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

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

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

→ compiler generated code

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

Compiletime Error:

cannot find symbol

Symbol: constructor P()

Location: class P.

Conclusions:

*① If the parent class contains any constructor, then while writing child class constructor we have to take special care.

*② Whenever we are writing any argument constructor, then it is highly recommended to write no-arg constructor also.

Case (iii):

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

→ `C() { super(); }` → compiler generated code. → CE: Unreported Exception java.io.IOException in default constructor.

If we are taking constructor in child class as follows, we won't get any compiletime Error.

```
C() throws IOException/Exception
{
 super();
}
```

Conclusions:

(1) If parent class constructor throws some checked Exception then child class constructor should throws the same checked Exception or its parent.

(2) within the constructor if there is any chance of raising checked Exception, then highly recommended to handle that Exception within the constructor only by using try, catch.

Q) which of the following is true.

X (1) Then name of the constructor need not be same as name of the class.

X (2) Return type concept is applicable for constructors.

X (3) we can use any modifier for the constructor.

X (4) we can't declare a constructor explicitly as private.

X (5) we can develop a Singleton class without using private constructor.

X (6) within a class we can take atmost one constructor.

✓ (7) compiler will always generate default constructor.

X (8) If we are not writing no-arg constructor then only compiler will generate default constructor.

X (9) A class can contain both programmer return constructor and compiler generated constructor simultaneously.

✗ (10) Overloading concept is not applicable for constructors.

✗ (11) Inheritance concept is applicable for constructors but cannot be overriden.

✗ (12) Overriding concept is applicable for constructors but not overloading.

✗ (13) The first line in every constructor should be super() always.

✗ (14) The first line in every constructor should be either super() (or) this() and if we are not writing anything then compiler will alway place this().

✗ (15) We can use super() and this() anywhere.

✗ (16) Only concrete classes can contain constructors but not abstract classes.

✗ (17) Interface can contain constructors.

✗ (18) Recursive constructor invocation is always RuntimeException.

✗ (19) If the Parent class constructor throws some Unchecked Exception then compulsary every child class constructor should through the same unchecked Exceptions.

✓ (20) None of the above.

---

(16) Type-casting:

Parent class reference can be used to hold child class object.

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

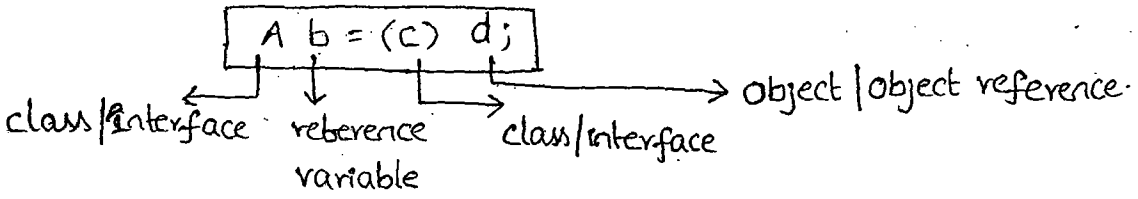
Interface reference can be used to hold implemented class objects.

```
Runnable r = new Thread ();
List     l = new ArrayList ();
```

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

StringBuffer sb = (StringBuffer) o;

Prototype of Type Casting:



compiler's checking-1:

The type of 'd' and 'c' must have some relationship. (either parent to child (or) child to parent (or) same type.). Otherwise we will get compiletime Error saying

> C.E: Inconvertible types.
> found: d type
> required: c type.

Ex: class Test

```
{
  public static void main (String [] args)
  {
    Object o = new String ("durga");
    StringBuffer sb = (StringBuffer) o;
  }
```

Ex: class Test

```
{
  public static void main(String[] args)
  {
    String s = new String("Durga");
    StringBuffer sb = (StringBuffer)s; → C.E: inconvertible types
  }                                      found: java.lang.String
}                                        required: java.lang.StringBuffer
```

=

Compiler's checking-2:

'C' should be either same (or) derived type of 'A'. Otherwise we will get compiletime Error saying

> C.E: Incompatable types
> found: C
> required: A

Ex: Object o = new String("durga");
StringBuffer sb = (StringBuffer)o;
valid.

Ex: String s = new String("durga");
StringBuffer sb = (Object)s; → C.E: incompatible types.
found: java.lang.Object
required: java.lang.StringBuffer

=

Runtime checking by JVM:

The underlying object type of 'd' must be same (or) Derived type of 'C'. otherwise we will get RuntimeException saying ClassCastExceptio

R.E: ClassCastException

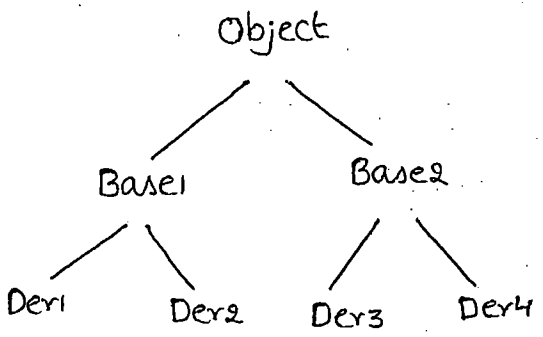
Ex: Object o = new String ("durga");

StringBuffer sb = (StringBuffer) o; → R.E.

↓

R.E: ClassCastException: String cannot be cast to StringBuffer.

Ex:




```
Der4 d = new Der4();
Object o = (Base2)d;
Der3 d3 = (Der3)o; → R.E: ClassCastException.
Der4 d4 = (Base2)d; → C.E:1
Der4 d5 = (Base1)d; → C.E:2
Object o = (Base1)((Object)(new Derived()));
```

C.E:1 : Incompatible types
found: Base2
required: Der4

C.E:2: Inconvertible types
found: Der4
required: Base1

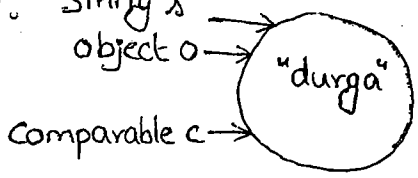
→ In typecasting we are not creating completly seperate independent object just we are creating another type of reference for the existing object.

Ex: String s = new String ("durga");

Object o = (object) s;

S.O.pln (s == o); true

Comparable c = (Comparable) s;



Ex: C2 c = new C2();

c.m1(); → C2:m1

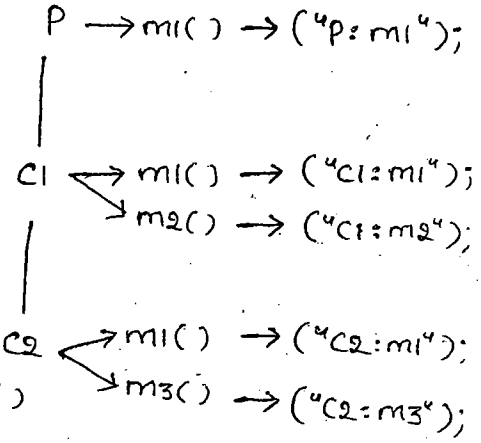
((C1)(c)).m1(); → C2:m1

((C1)(c)).m3(); → C.E:

Cannot find symbol

Symbol: method m3()

Location: class c.



(P((C1)(c))).m1(); → C2:m1

(P(C1)(c)).m2(); X

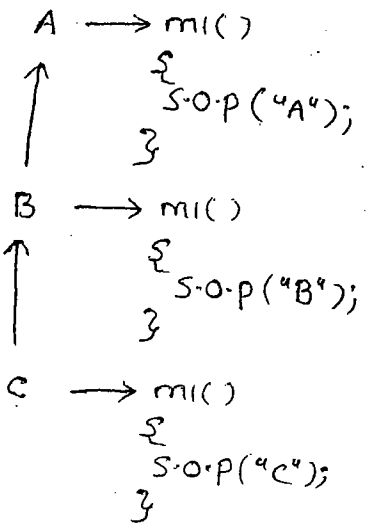
(P(C1)(c)).m3(); X

Ex: C c = new C();

c.m1(); → c

((B)c).m1(); → c

((A)((B)c)).m1(); → c



→ If every method is the static then

```
C c = new C();
c.m1(); ⟶ C
((B)c).m1(); ⟶ B
((A)((B)c)).m1(); ⟶ A
```

=

```
C c = new C();
S.o.p(c.i); ⟶ 888
S.o.p(((B)c).i); ⟶ 777
S.o.p(((A)((B)c)).i); ⟶ 666
```

```
A ⟶ int i=666;
↑
B ⟶ i=777;
↑
C ⟶ i=888;
```

→ If all variables as static then no change in output.

=

(14) Coupling:

The degree of dependency between the components is called coupling.

Ex:

```
class A
{
static int i=B.j;
}
```

```
class B
{
static int j=C.m1();
}
```

```
class C
{
public static int m1()
{
return D.k;
}
```

```
class D
{
static int k=10;
}
```

The dependency between the above components is high. Hence these components are said to be tightly coupled with each other.

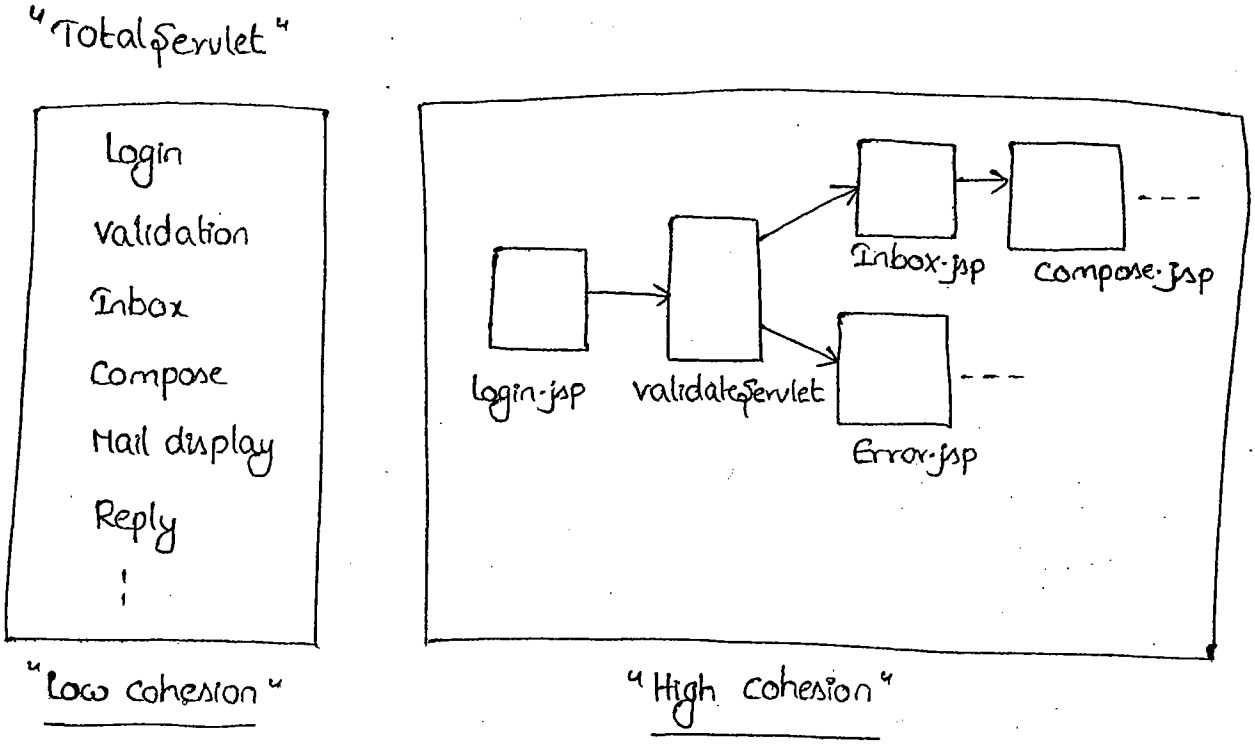
Tightly coupling is never recommended. Because it has the following serious disadvantages:

1. It reduces maintainability of the appln.
2. Without effecting remaining components we can't modify any component.
3. Hence enhancement will become very ~~different~~ difficult.
4. It doesn't promote reusuability of the code.

Note: loosly coupling is always good programming practice.

(15) Cohesion:

For every component we have to define a clear well defined functionality. Such type of component is said to be follow high cohesion.



"Low cohesion"

"High cohesion"

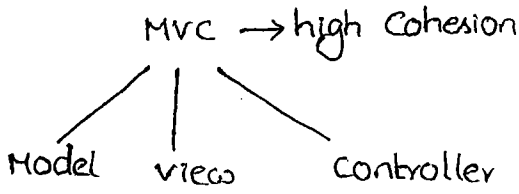
→ High cohesion is always recommended. Because it has several advantages.

(1) Without effecting remaining components we can modify any component. Hence enhancement will become very easy.

(2) It improves maintainability of the application.

(3) It promotes reusuability of the code. i.e; whenever validation is required we can reuse the same validate servlet without rewriting.

Ex: MVC framework follows high cohesion.



Model: meant for Business logic.

View: meant for Presentation logic.

Controller: meant for co-ordination activity.

Hence for every component a clear well defined functionality is defined. Hence it is said to be follow "high cohesion".