OOPS concepts:

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

Data Hiding:

Outside person cannot access our internal data directly or our internal data should not go out directly.
This OOP feature is nothing but data hiding.
The data should be available only after validation/Authentication.

Example:

1. After providing proper username and password we can access our Gmail inbox information.
2. Even though we are valid customer of the Bank, we can access only our account information, we cannot access other account's information.

How to achieve data hiding:

- By declaring datamember(variable) as private.
- Using Getter setter methods.

```
public class Account
{
        private double Balance;
        .
        .
        .
        public double getBalance()
        {
                //Validation
                return balance;
        }
        .
        .
        .
```

}

Advantage:
Security

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

Abstraction:

Hiding internal implementation and just highlight the set of services that we(application) are offering is the concept of abstraction.

Example:
Trough ATM GUI Screen, Bank highlights the set of services that bank offers, without highlighting internal implementation.

How to achieve:

- By using Abstract classes – partial Abstraction
- By using Interfaces- full Abstraction

Advantage:

1. Outside person is not aware of our implementation.→ security.
2. Without affecting end user internal implementation can be changed.→easy Enhancement .
3. Improves Easiness to use the system.
4. Maintainability.

Encapsulation:

The process of binding data members and corresponding methods into a single unit is nothing but encapsulation.

Example:

- Every Java class in example of encapsulation

```
public class Account
{
    private double balance;

    public double getBalance()
    {
            //validation
            Return balance;
    }
```

```
        public void setBalance(double balance)
        {
                //validation
                this.balance=balance;
        }
  }
```

A GUI screen to use this functionality.

How to achieve:

```
Class Student
{
        data members
                +
        Methods(behavior)
}
```

Note: If any component follows data hiding and abstraction then such type of component is said to be encapsulated component

Encapsulation=data hiding + abstraction

Advantages:

- We can achieve security.
- Enhancement will become easy
- Improves maintainability of the application

Tightly encapsulated class:

If all the variables of the class are declared private, such class is called Tightly Encapsulated class.

Note: If parent class is not tightly encapsulated then no child class is tightly encapsulated.

IS-A Relationship:

- It is also known as Inheritence.
- The main advantage of IS-A Relationship is code reusability.
- By using extends keyword we can implement IS-A relationship.

Example:

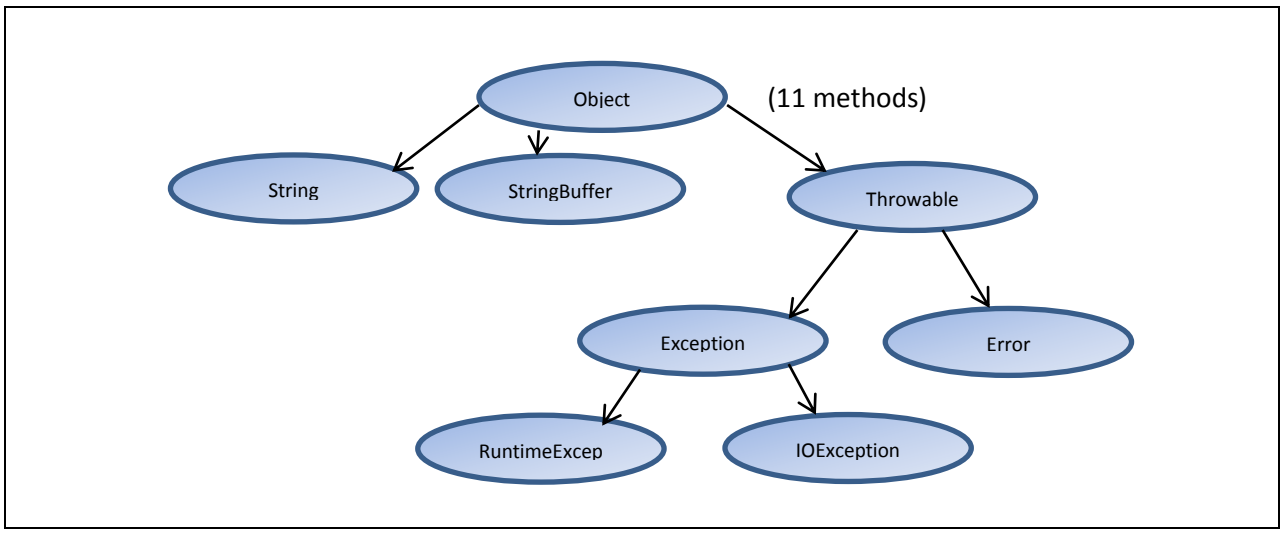
```
class P
{
		public void m1()
		{
			sopln("parent");
		}
}

class C extends P
{
	public void m2()
	{
		sopln("child");
	}
}

class Test
{
	psvm(String args[])
	{
		1. P p=new P();
		p.m1();                    // correct
		p.m2();                    //Compile error
		2. C c=new P();
		c.m1();                    //correct
		c.m2();                    //correct
		3. P p1=new C();
		p1.m1();                   //correct
		p1.m2();                   //compile error
		4. C c1=new P();           //compile error
	}
}
```

- Parent methods are by default available to the child and hence on the child reference we can call both parent and child class methods.
- Child methods are not available to the parent and hence parent reference we can't call child specific methods.
- parent reference can be used to hold child object but using that reference we can't call child specific methods, but we can call the methods present in parent class.
- Parent reference can be used to hold child object, but child reference cannot be used to hold parent object.

Advantage:
Code reusability

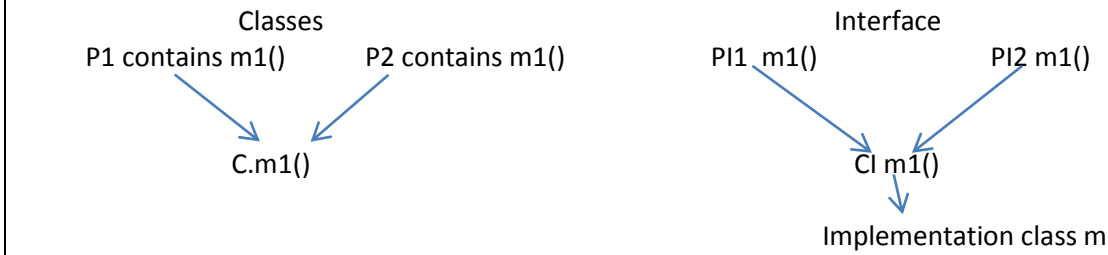
Note:

1. The most common methods which are applicable for any type of child, we have to define in parent class.
2. The specific methods applicable for particular child, we have to define in child class.



3. Entire Java API is implemented based on Inheritance concept.
4. Most common methods which are applicable for any java object are defined in Object class, and hence every class in Java is subclass(child) of object either directly or indirectly so that object class methods are by default available to every java class without re writing, due to this Object class acts as root for all Java class.
5. Throwable class defines the most common methods which are required for every exception and error classes, hence this class act as root for Java Exception hierarchy.

Why Multiple Inheritance is not supported:



There may be chance of Ambiguity problem.

6. Multiple Inheritance: A Java class cannot extend more than one class at a time, hence java won't provide support for multiple inheritance in classes. But it provides multiple inheritance for interfaces.
   If our class does not extend any other class then only our class is direct child of object.
   Eg.
   A is direct child of object.

Class A{

}
If our class extends any other class then our class is indirect child of Object.
Eg A is not direct child of Object, A is child of B, B is child of Object.(**MultiLevel Ineritance**).
Class A extends B
{

}

7. Cyclic inheritance is not allowed in java.
   Eg class A extends A or class A extends B | class B extends A

Has-A Relationship:

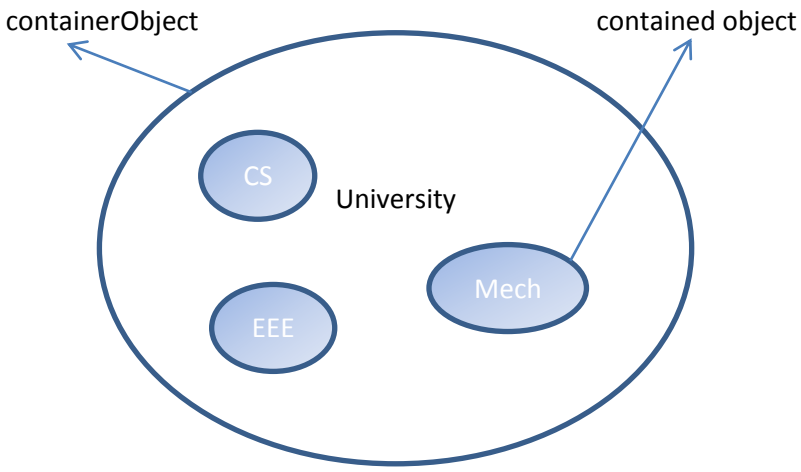
- Has-A Relationship is also known as Composition or Aggregation.
- There is no specific keyword to implement Has-A Relation, but most of the times we are depending on "new" keyword.
- The Main advantage of Has-A relationship is reusability of the code.

Example:
Class Car{

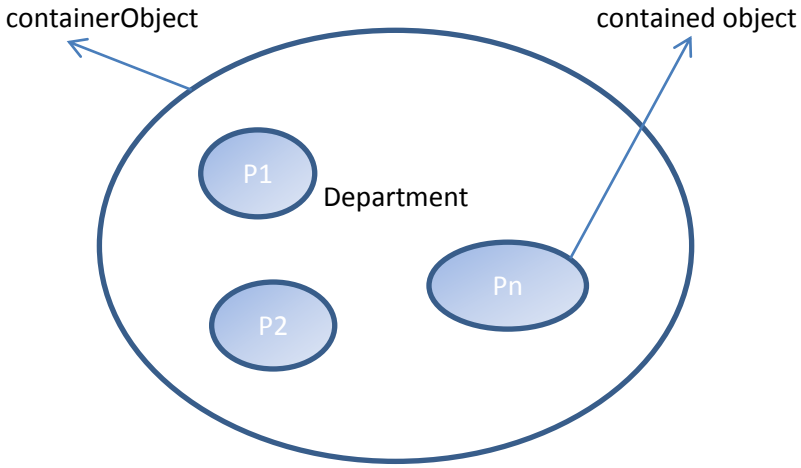
    Engine e=new Engine();
}

Car Has-A engine reference.

Difference between composition and aggregation
Composition: Without existing container object if there is no chance of existing contained objects, then container and contained objects are strongly associated. Such a strong association is known as Composition.
Eg: University consists of several depts, without existing University there is no chance of existing dept, hence University and dept are strongly associated and this strong association is called Composition.



Aggregation: Without existing container object if there is a chance of existing contained object, then such container and contained objects are weakly associated. Such a weak association is aggregation.
Eg: Department consists of several professors, without existing dept. there may be a chance of existing professor objects. Hence dept and professor objects are weakly associated and this weak association is called aggregation.



Note:

1. In composition objects are strongly associated whereas in aggregation objects are weakly associated.
2. In composition container object holds directly contained objects whereas in aggregation container object holds just references of contained objects.

IS-A vs HAS-A
If we want total functionality of a class automatically then we should go for IS-A relationship.
If we want part of the functionality then we should go for HAS-A relationship.

Method Signature:

- In java method signature consists of method name followed by argument types.
  Eg: public static int m1(int i,float f);
  Method signature – m1(int,float)

- Compiler will use method signature to resolve method calls.

Eg:

```
Class Test
{
Public void m1(int i){}
Public void m2(String s){}
}
```

```
Test t= new Test();
t.m1(10);
t.m2("XYZ");
t.m3(10.5);→ CE: cannot find symbol
                m3(double)
```

m1(int)
m2(String)

- Within a class two methods with same signature are not allowed.

OverLoading:

Two methods are said to be overloaded if and only if both the methods have same name but different argument types.

Example:

```
Class Test{
        Public void m1(){
                Sopln("no-args);
        }
        Public void m1(int i){
                Sopln("int-args");
        }
        Public void m1(double d){
                Sopln("double-args");
        }
}
```

```
Test t=new Test();
t.m1(); //no-args
t.m1(10); //int-args
t.m1(10.5); //double-args
```

In overloading method resolution is always taken care by compiler based on reference type, hence overloading is also considered as compile time polymorphism or static polymorphism or early binding.

Case1: Automatic promotion in overloading

a. While resolving overloaded methods if exact match method is not available then we won't get any compile time error immediately.
b. First the argument is promoted to the next level and check whether matched method is available or not.
c. If matched method is available then it will be considered, else the compiler promotes argument once again to the next level. This process will be continued until all possible promotions.
d. Still if matched method is not available then we will get compile time error.
e. Following are all possible promotion in overloading:
   Byte→short→int→long→float→double.
   Char→int→long→float→double.
   This process is called automatic promotion in overloading.
   Eg:
   Overloaded methods

```
Class Test{
Public void m1(int i){
        Sop("int-args");
}
Public void m1(float f){
        Sop("float-args");
}
}
```

```
Test t = new Test();
t.m1(10);       //int-args
t.m1(10.5f);    //float-args
t.m1('a');      //char to int int-args
t.m1(10l);      //long to float float-args
t.m1(10.5)      //double Compile error
```

Case2: child vs parent type

While resolving Overloaded methods compiler will always give the precedence for child type argument when compared with parent type argument.

```
Overloaded methods
Class Test{
Public void m1(String s){
        Sop("String ver");
}
Public void m1(Object o){
        Sop("Obejct ver");
}
}
```

```
Test t = new Test();
t.m1(new Object());      //Object ver
t.m1("Vibhav");          //String ver
t.m1(null);       //String ver
```

Case3: Ambiguity

While resolving Overloaded methods if argument is accepted for both methods and methods are of same precedence then it results in compile time error

```
Overloaded methods
Class Test{
Public void m1(String s){
        Sop("String ver");
}
Public void m1(SringBuffer sb){
        Sop("StringBuffer ver");
}
}
```

```
Test t = new Test();
t.m1(new StringBuffer("a");//buff ver
t.m1("Vibhav");          //String ver
t.m1(null); //compile error:reference to
            m1 is ambiguous
```

Case4: multiple arguments

```
        Overloaded methods
Class Test{
Public void m1(int I,float f){
        Sop("i-f ver");
}
Public void m1(float f,int i){
        Sop("f-I ver");
}
}
```

```
Test t = new Test();
t.m1(10,10.5f);    //i-f ver
t.m1(10.5f,10);    //f-i ver
t.m1(10,10);//compile error ambiguous
t.m1(10.5f,10.5f);//CE cannot find sym
```

Case5: Old version vs New version

The old version always gets the preference.

```
Overloaded methods
Class Test{
Public void m1(int i){
        Sop("General");
}
Public void m1(int… x){
        Sop("var args");
}
}
```

```
Test t = new Test();
t.m1();    //var args
t.m1(10,20);       //var args
t.m1(10);          //General
```

In general var-args method will get least priority i.e. if no other method matched then only var-arg method will get chance. It is exactly same as default case inside switch.

Case6: Objects

```
Class Animal{                          Test t= new Test();
}                                      Animal a=new Animal();
Overloaded methods                     t.m1(a);  //animal ver
Class monkey extends Animal{           Monkey m= new Monkey();
}                                      t.m1(m);   //monkey ver
Class test {                           Animal a=new Monkey();
Public void m1(Animal a){              t.m1(a);   //animal ver since method resolution is based on ref
        Sop("animal ver");
}
Public void m1(Monkey m){
        Sop("monkey ver");
}
}
```

Note: In overloading runtime object won't play any role in method resolution

Overriding:

- Whatever methods parent has are by default available to the child through inheritance.

If child class is not satisfied with parent class implementation then child is allowed to re-define those methods based on its requirement. This processes is called overriding.

- The parent class method is called overridden method and the child class method is called overriding method.

Example:

```
Class P{
    Public void property(){
            Sop("cash,land,gold");
    }
    Public void mary(){                    //overridden
            Sop("ABC");
    }
}
Class c extends P{                                      overriding
    Public void mary(){                    //overriding
            Sop("XYZ");
    }
}
P p=new P();→parent method
C c=new C();→child method
P p=new C();→child method
```

- In Overriding method resolution is always taken care by JVM based on run-time object and hence overriding is also considered as run-time polymorphism or dynamic polymorphism or late binding.

Rules for overriding:

1. In overriding method names and argument types must be matched. i.e. method signature must be same.
2. In Overriding return types must be same but this rule is applicable until 1.4 ver only, from 1.5 ver onwards we can use co-variant return types. According to this child class method return need not be same as parent method return type, its child type also allowed.
   Eg:

```
Class P{                                    class C extends P{
        Public object m1(){                         public String m1(){          //String is
                return null;                                return null;         co-variant
        }                                           }
}                                           }
```

3. Co-variant return type concept is applicable only for object types but not for primitive types.

Parent class method Return type	Object	Number	String	double
Child class method Return type	String,StringBuffer… valid	int,float… valid	Object **invalid**	int **invalid**

4. Parent class private methods not available to the child and hence overriding concept does not apply for private methods

```
Eg: class P{                                class C extends P{
        Private void m1(){                          private void m1(){      //not overriding
        }                                           }
    }                                       }
```

   Based on our requirement we can define exactly same method in child class it is valid but not overriding.
5. If parent class method is declared final, then we cannot override that method.
6. Parent class Abstract methods should be overridden in child class, if not then child should be abstract too.

```
Eg: abstract class P{                       class c extends P{
        Public abstract void m1();                  public void m1(){
    }                                                       }
                                            }
```

7. Parent class non-abstract method can be overridden as abstract in child class.

```
Eg: class P{                                abstract class c{
        Public void m1(){}                          public abstract void main(){}
    }                                       }
```

   Advantage: stop availability of parent class methods to next level.
8. In overriding the following modifier won't keep any restriction.
   a. Synchronized b. native c. StrictFP

parent	**Final**	abstract	Synchronized	Non-final	native	strictfp
Child	**Non-final/final**	Non-abstract	Non-synchronized	final	Non-native	Npn-strict fp
validity	**Not valid**	Valid both	Valid both	valid	Valid both	Valid both

9. While overriding we cannot reduce scope of access modifier, but we can increase the scope. Public→protected→default→private //highest to lowest
10. Exceptions: If the child class method throws any checkedException then parent class method should compulsory throw the **same checkedEception** or **its parent**, else we get compile time error.
    There are no restrictions for UnCheckedExceptions

Interview Questions:

P:public void m1() throws Exception C:public void m1()throws	**Valid**
P:public void m1() throws C:public void m1()throws Exception	**Not valid**
P:public void m1() throws Exception C:public void m1()throws IOException	**Valid**
P:public void m1() throws IOException C:public void m1()throws Exception	**Not Valid**
P:public void m1() throws IOException C:public void m1()throws FileNotFoundException,EOFException	**valid**
P:public void m1() throws IOException C:public void m1()throws EOFException,InterupptedException	**Not valid**
P:public void m1() throws IOException C:public void m1()throws AE,NP,CCE	**valid**

Overriding w.r.t Static methods:

1. We cannot override a static method as non-static, otherwise we will get compile time error.
2. We cannot override a non-static method as static, otherwise we will get compile time error.
3. If parent and child class both methods are static, then we won't get any compile time error it seems overriding concept applicable for static methods but it's not actual overriding, its method hiding.
4. Method Hiding: All rules are same as method overriding Except following:

Method hiding	Method
Both parent and child class methods static	Both parent and child class methods non static
Compiler reference	JVM runtime object

```
Class P{
        Public static void m1()
        {
                Sop("parent");
        }
  }
Class c extends P{
        Public static void m1()
        {
                Sop("child");
        }
}
```

```
P p=new P();
        p.m1();→parent
        C c=new C();
        c.m1();→child
        P p1=new C();
        p1.m1();→parent
```

Overriding w.r.t var-args:

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

Example:

Overloading

```
Class P{
        Public void m1(int… x){
                Sop("parent");
        }
}
Class C extends P{
        Public void m1(int x){
                Sop("child");
        }
}
```

overriding

```
class P{
        public void m1(int… x){
                sop("parent");
        }
}
class C extends P{
        public void m1(int… x){
                sop("child");
        }
}
```

Overriding w.r.t variables

Variable resolution is always taken care by compiler based on reference type, irrespective of whether variable is static or non-static(overriding concept is applicable only for methods but not for variables).

Example:

```
Class P{
        Int x=888;
}
Class C extends P{
        Int x=999;
}
```

P p=new P() ; sop(p.x);→888
C c=new C(); sop(c.x)→999
P p1=new C();sop(p1.x)→888

Interview Question:

Overloading vs overriding

- Method name
- Argument type
- Method signature
- Return type
- Private, static, final methods
- Access modifiers
- Throws clause
- Method resolution
- Also known as

Interview:

Consider following method in parent class

Public void m1(int x) throws IOException

In the child class which of the following methods can be written

a. Public void m1(int i) → valid through overriding
b. Public static int m1(long l)→valid through over loading
c. Public static void m1(int i)→ invalid through overriding
d. Public void (int i) throws Exception→invalid through overriding

e. Public static abstract void m1(doube d)→invalid static and abstract cannot be used together

Note: In overloading we have to check only method names(must be same) and argument types(must be different). We are not required to check remaining like, return type, access modifiers etc.
In overriding all these needs to be checked.

Polymorphism:

- One name but multiple forms is the concept of polymorphism.
  It can be achieved through overloading or overriding.
- Usage of parent reference to hold child object is considered as polymorphism.
- Parent class reference can be used to hold child class object but by using that reference we can call on the methods available in parent class.
  P p=new C(); p→m1()
  p.m1();//valid C→m2()
  p.m2();//Compile time error
- By using child reference we can call both parent and child class methods.
  C c=new C();
  c.m1();
  c.m2();
- If we don't know exact runtime type of object then we should go for parent reference.
  Eg: the first element in arraylist can be any type,student,customer,string etc. hence return type of get method is Object, which can hold any object.

C c=new C(); eg:AL l=new AL();	**P p=new C();eg: List l=new List();**
We can use this approach if we know exact runtime type of object	We can use this approach when exact runtime type of object is not known
By using child reference we can call both parent and child class methods.	By using parent reference we can call only parent class methods.
We can use child reference to hold only particular child class object.	We can use parent reference to hold both parent and any child class objects.

Coupling:

The degree of dependency between the components is called coupling.
If the dependency is more, then it is considered as tight coupling and if dependency is less, then it is considered as loose coupling.

Tightly coupled:

Class A{
Static int i=B.j;
}
	Class B{
	Static int j=C.k;
	}
		Class C{
		Static int k=D.m1();
		}
			Class D{
			Public static int m1(){
			Return 10;
			}
			}

- The above components are said to be tightly coupled with each other because dependency between the components is more.
- Tightly coupling is not a good programing practice because it has several disadvantages:

1. Without affecting remaining components, we can't modify any component and hence enhancement will become difficult.
2. It suppresses reusability.
3. It reduces maintainability of the application.

- Hence we have to maintain dependency between the components as less as possible i.e. loosely coupling is a good programing practice.

Cohesion:

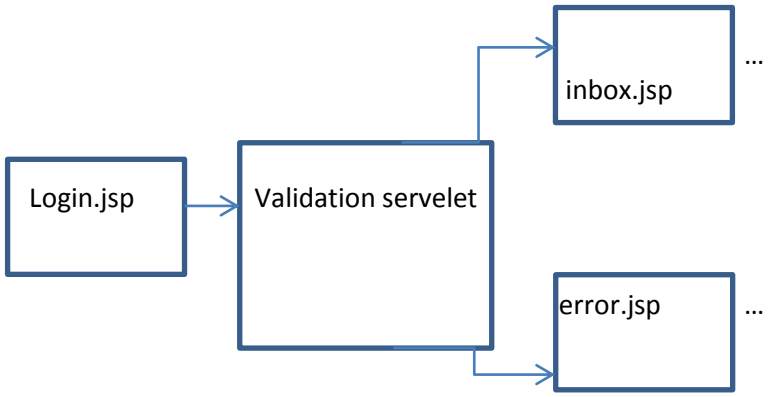
If for every component a clear well defined functionality is defined then that component is said to be follow high cohesion.

Eg:

Total servelet

Loginpage display
Validation
Inbox page
Reply page
Compose page
.
.
.

Low cohesion



High Cohesion

- High cohesion is always a good programing practice because it has several advantages:
  a. Without affecting remaining components we can modify any component. Hence enhancement becomes easy.
  b. Code reusability is improved
  c. Easy to maintain.

Note: loosely coupling and high cohesion are good programming practices.

Object Type-casting:

We can use parent reference to hold child object, similarly we can use interface reference to hold implemented class object.

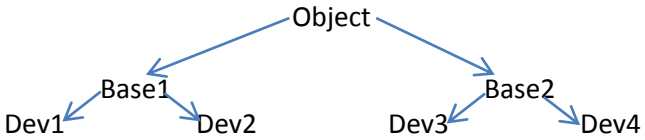
```
A b= (C) d;
```

A→class|interface name
b→name of reference variable
C→class|interface name
d→reference variable name

Rules:

1. compile time check 1- the type of 'd' and 'C' must have some relation, either child to parent or parent to child or same type. Otherwise we will get Compile time Error saying "incovetible types found: d type required:C"
   eg: Object o=new String("vibhav");
   StringBuffer sb=(StringBuffer)o;
   String s=new String("Vibhav");
   StringBuffer sb=(StringBuffer) s;//error
2. compile time check 2- the type of 'C' must be same as type of 'A' or derived(child) type of 'A'. Otherwise we will get compile time error saying "incompatible types found:C required:A"
   eg: Object o=new String("vibhav");
   StringBuffer sb=(StringBuffer)o;
   String s=new String("Vibhav");
   StringBuffer sb=(String) s;//error
3. Run time check by JVM- Runtime object type of 'd' must be either same or derived type of 'C'. Otherwise we will get runtime exception saying "classCastException".
   Eg:Object o=new String("vibhav");
   StringBuffer sb=(StringBuffer)o;//error
   Object o=new String("Vibhav");
   Object o1=(String)o;//correct
4. Strictly speaking through type casting we are not creating any new object, for the existing object we are providing another type of reference variable i.e. we are perfoming type-casting but not object casting.
   Note:A→B→C (A)((B)C)→A a=new C();

Interview:



Base2 b=new Dev4();

Object o=(Base2)b;	Valid
Object o=(Base1)b;	inconvertible
Object o=(Dev3)b;	CCE
Base2 b1=(Base1)b;	inconvertible
Base1 b1=(Dev4)b;	incompatible
Base1 b1=(Dev1)b;	Inconvertible

1. P→m1() ,c extends P,c→m2();
   C c=new C();
   c.m1(); valid
   c.m2(); valid
   P p=new C();
   p.m1(); valid
   p.m2(); invalid- parent cannot called child specific methods

2. A→B→C A contains m1() sop("A") B contain m1() Sop("B") C contains m1() Sop ("C");\
   C c=new C();
   c.m1();→ C
   ((B)c).m1();→C
   ((A)((B)c))m1;→C //since overriding resolution is based on runtime object.

3. A→B→C A contains static m1() sop("A") B contains static m1() Sop("B") C contains static m1() Sop("C")
   C c=new C();
   c.m1();→ C
   ((B)c).m1();→B
   ((A)((B)c))m1;→A //since Static resolution is based on compile reference.

4. A→B→C A contains x=777 B contains x=888 C x=999;
   C c=new C();
   Sop(c.x);→999
   Sop(((B).c).x);→888
   Sop((A((B).c)).x);→777

Static control flow:

Whenever we are executing a java class, the following sequence of activities/steps will be executed as a part of static control flow

1. Identification of static members from top to bottom.[1-6]
2. Execution of static variable assignments and static blocks from top to bottom.[7-12]
3. Execution of main method.[13-15]

```
Example:  default values of i=0[RIWO] j=0[RIWO] when assigned i=10[R&W] j=20[R&W]
Class Base
{
(1)     Static int i=10;                        (7)
(2)     Static
        {
                m1();                           (8)
                sop("First static block");      (10)
        }
(3)     Public static void main(String args[])
        {
                m1();                           (13)
                sop("main method");             (15)
        }
(4)     Public static void m1()
        {
                Sop(j);                         (9) //j=0  ,(14)//j=20
        }
(5)     Static
        {
                Sop("second static block");     (11)
        }
(6)     Static int j=20;                        (12)
}
```

Output:
0
First Static Block
Second Static Block
20
Main method

Direct/Indirect Read

Inside static block if we are trying to read a variable that read operation is called direct read.
If we are calling a method and within that method if we are trying to read a variable that read operation is called indirect read.

If a variable is just identified by the JVM and original value not yet assigned then the variable is said to be in ReadIndirectlyWriteOnly[RIWO] state.
If a variable is in RIWO state and we try to read it directly we get compile time error saying "illegal forward reference".

Interview:

Class Test{
Static int x=10;
Static{
Sop(x);
}
}
	Class Test{
	Static{
	Sop(x);
	}
	Static int x=10;
	}
		Class Test{
		Static{
		m1();
		}
		P S V m1(){
		Sop(x);
		}
		Static int x=10;
		}

o/p:10 RE: NoSuchMethodError:main	CE:illegal forward reference	o/p: 0 RE:NoSuchMethodError :main

Static Block:

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

Example1: at the time of java class loading the corresponding native libraries should be loaded hence we have to define this activity inside static block.

```
Class test{
        Static{
                System.loadLibraries("native library path");
        }
}
```

Example2: after loading every Database driver class we have to register driver class with driver manager, but inside database driver class there is a static block to perform this activity.

```
Class DbDriver{
        Static{
                Register this driver with driver manager
        }
}
```

Within a class we can declare any number of static blocks but all these static blocks will be executed from top to bottom.

Interview :

- Without writing main method is it possible to print some statement to console?
  -yes by using static block and system.exit(0);

```
Class test{
        Static{
                Sop("hello");
                System.exit(0);
        }
}
```

- Without writing main method and static block is it possible to print some statement to the console??
  -yes using static variable and static method or Instance block or constructor

```
Class Test{
        Static int x=m1();
        Public static int m1(){
                Sop("hello");
                System.exit(0);
                return 10;
        }
}
```

- From 1.7 version onwards main method is mandatory to start a program execution. Hence from 1.7 ver onwards without writing main method it is impossible to print some statements to the console.

Static control flow in parent to child relationship:

Whenever we are executing child class the following sequence of events will be executed automatically as a part of static control flow.

1. Identification of static members from parent to child.[1-11]
2. Execution of static variable assignments and static blocks from parent to child.[12-22]
3. Execution of child class main method only.[23-25]

Example:

```
class Base
{
(1)     static int i=10;                        (12)
(2)     static{
                m1();                           (13)
                sop("Base static block");       (15)
        }
(3)     p s v main(String args[]){
                m1();
                sop("Base main");
        }
(4)     p s v m1(){
                sop(j);                         (14)j=0
```

```
        }
(5)     static int j=20;                        (16)j=20
}
class Derived extends Base{
(6)     static int x=100;                       (17)x=100
(7)     static{
                m2();                           (18)
                sop("Derived 1st static block");  (20)
        }
(8)     p s v main(String args[]){
                m2();                           (23)
                sop("Derived main");            (25)
        }
(9)     p s v m2(){
                sop(y);                         (19)y=0(24)
        }
(10)    static{
                sop("Derived 2nd static block"); (21)
        }
(11)    static int y=200;                       (22)y=200
}
child class execution
```

O/P:0
Base static block
0
Derived 1st static block
Derived 2nd static block
Derived main

Note: whenever we are loading child class automatically parent class will be loaded. But whenever we are loading parent class child class will not be loaded.(because parent class member are by default available to the child class whereas child class member are not available for parent class.)

Instance control flow:

Whenever we are executing a java class first static control flow will be executed.

In the static control flow if we are creating an object the following sequence of events will be executed as a part of Instance control flow:

1. Identification of Instance members from top to bottom.[3-8]
2. Execution of instance variable assignments and instance block from top to bottom.[9-14]
3. Execution of constructor.[15]
   [1,2,16]→main method

Example:

```
class Test
{
(3)     int i=10;                                   (9)
(4)     {
                m1();                               (10)
                sop("First Instance Block");        (12)
        }
(5)     Test()
        {
                sop("constructor");                 (15)
        }
(1)     P S V main(String args[])
        {
(2)             Test t=new Test();                  Line1
                sop("main");                        (16)
        }
(6)     public void m1()
        {
                sop(j);                             (11)j=0
        }
(7)     {
                sop("second instance block");       (13)
        }
(8)     int j=20;                                   (14)j=20
}
```

o/p: 0
First Instance Block
second instance block
constructor
main

if we comment Line1 then output is
o/p: main

Note: static control flow is one time activity which will be performed at the time of class loading, but instance control flow is not one time activity and it will be performed for every object creation.
Object Creation is most costly operation if there is no specific requirement then it's not recommended to create object.

Instance control flow in parent to child relationship:
Whenever we are creating child class object the following sequence of events will be performed automatically as a part of instance control flow:

1. Identification instance members from parent to child class.[4-14]
2. Execution of instance variable assignment and instance block only in parent class.[15-19]
3. Execution of parent constructor.[20]
4. Execution of instance variable assignment and instance block in child class.[21-26]
5. Execution of child constructor.[27]
   1,2,3,28→main

Example:

```
class parent{
(4)     int i=10;                                   (15)i=10
(5)     {
            m1();                                   (16)
            sop("parent Instance block");           (18)
        }
(6)     parent(){
            sop("parent constructor");              (20)
        }
(1)     P S V main(String args[]){
            Parent p=new Parent();
            sop("parent main");
        }
(7)     public void m1(){
            sop(j);                                 (17)j=0
        }
(8)     j=20;                                       (19)j=20
}
class child extends Parent{
(9)     int x=100;                                  (21)x=100
(10)    {
            m2();                                   (22)
            sop("CFIB");                            (24)
        }
(11)    child(){
            sop("child constructor");               (27)
        }
(2)     P S V main(Sting args[]){
(3)         Child c=new Child();
            sop("child main");                      (28)
        }
(12)    public void m2(){
            sop(y);                                 (23)y=0
        }
(13)    {
            Sop(“CSIB”);                            (25)
        }
(14)    y=200;                                      (26)y=200
}
```

o/p:
```
0
parent Instance block
parent constructor
0
CFIB
CSIB
child constructor
child main
```

Note: From static area we cannot access instance members directly because while executing static are JVM may not identify instance memebers.
Example:

```
Class Test{
        Int x=10;
        P s v main(String args[]){
                Sop(x);
        }
}
```

Compile time error- non static variable x cannot be referenced from static context

Interview:

Q:How many ways we can create an object in java? Or In how many ways we can get object in Java

A: Following

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;

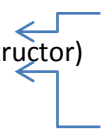
5. By Using Deserialization
   FileInputStream fis=new FileInputStream("abc.scr");
   ObjectInputStream ois=new ObjectInputStream(fis);
   Dog d=(Dog)ois.readObject();

Constructors:

- Once we create an object compulsory we should perform initialization then only the object is a position to respond properly.
  Example: s1→name:null s2→name:null
  Rollno:0 rollno:0
- Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of the object() this piece of code is nothing but constructor. Hence the main purpose of constructor is to perform initialization of object().
  Example:

```
Class student{
        String name;
        Int  rollno;
```

```
                Student(String name,int rollno){
(Constructor)           this.name=name;
                        T his.rollno=rollno;
                }
                P s v main(String args[]){
                        Student s1=new Student("Vibhav",1);
                        Student s2=new Student("Xyz",2);
                }
        }
```



Difference between constructor and Instance block:

The main purpose of constructor is to perform Initialization of object, but other than initialization if we want to perform any activity for every object creation then we should go for instance block.(like updating one entry in DB for every object creation or incrementing count value for every object creation etc.)

Constructor and Instance block have their own different purposes and replacing once concept with another concept may not work always.

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

Example:

```
Class Test{
        Static int count=0;
        {
                Count++;
        }
        Test(){}
        Test(int i){}
        Test(double d){}
        P s v main(String args[]){
                Test t= new Test();
                Test t1=new Test(10);
                Test t2=new Test(10.5);
                Sop("no. of objects created:"+count);
        }
}
```

Rules of writing constructors:

1. Name of the class and name of the constructor must be matched.
2. Return type concept not applicable for constructors. Even void also. By mistake if we are trying to declare return type for the constructor then we won't get any compile time error because compiler treats it as a method.

   Eg:
   ```
   class Test{
           Void Test(){
                   Sop("its method but not constructor");
           }};
   ```

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

3. The only applicable modifiers for constructors are **public, private, protected, default.** If we are trying to use any other modifier we will get compile time error.

Default Constructor:

1. Compiler is responsible to generate default constructor (but not JVM).
2. If we are not writing any constructor then only compiler will generate default constructor i.e. if we are writing at least one constructor then compiler won't generate default constructor.
   Hence every class in java can contain constructor, it may be default generated by compiler or customized constructor explicitly provided by programmer but not both simultaneously.
3. Prototype:
   - It is always no args constructor.
   - The access modifier of default constructor is exactly same as access modifier of class. This rule is applicable only for public and default.
   - It contains only one line- super(); It is no argument call to super class constructor.

Interview Default Construtor:

Programmer's code	Constructor's code

Class Test{
}
	Class Test{
	Test(){
	Super();
	}
	}

Public class Test{
}
	Public Class Test{
	Public Test(){
	Super();
	}
	}

Public class Test{
Void Test(){
}
}
	Public class Test{
	Public Test(){
	Super();
	}
	Void Test(){
	}
	}

Class Test{
Test(){
}
}
	Class Test{
	Test(){
	Super();
	}
	}

Class Test{
Test(){
Super();
}
}
	Class Test{
	Test(){
	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(). If we are not writing anything then compiler will always place super(); in the constructor written.

Case1:

We can take super() or this() only in first line of constructor, if we are trying to take anywhere else we get compile time error.

```
Eg: class Test{
        Test(){                                  //compile time error : call to super must be first
                Sop("constructor");                                  statement in constructor
                Super();
        }
   }
```

Case2:

Within the constructor we can take either super() or this() but not both simultaneously

```
Eg: class Test{
        Test(){                          //CE: call to this must be first statement in constructor
                Super();
                This();
        }
    }
```

Case3:

We can use super() or this() only inside constructor, if we are trying to use outside constructor we will get compile time error. i.e. we can call a constructor directly from another constructor only.

```
Eg: class Test{
        Public void m1(){                //CE: call to this must be first statement in constructor
                Super();
                Sop("hello");
        }
    }
```

Super() and this():

a. We can use only inside constructor
b. Only in first line
c. Only one but not both simultaneously

Super(), this()	Super,this
These are constructor calls to call super class	These are keywords to refer super class and

and current class constructors.	current class instance members
We can use only in constructors at first line	We can use anywhere except static area
We can use only once in constructors	We can use any number of times

Overloaded Constructors:

- Within a class we can declare multiple constructors and all these constructors having same name but different type of arguments.
  Hence all this constructors are considered as over loaded constructors. Hence overloading concept applicable for constuctors.

```
Eg: class Test{
        Test(){
                This(10);
                Sop("no args");
        }
        Test (int i){
                This(10.5);
                Sop("int args");
        }

        Test(double d){
                Sop("double arg");
        }
    }
```

```
Test t=new Test();→double arg
                        int args
                        no args
Test t1=new Test(10);→double arg
                        int args
Test t2=new  Test(10.5);→double arg
Test t3=new Test(10l);→double arg
        //here long promoted to double
```

- For constructors inheritance and overriding concepts are not applicable, but overloading is applicable.
- Every class in java including abstract class can contain constructor, but interface cannot contain constructor.

Case1:

Recursive method call is a runtime exception saying stack overflow error
But in our program if there is a chance of recursive constructor invocation, then the code won't compile and we will get compile time error.

```
Eg: class Test{
        Test(){
                This(10);
        }
        Test(int i){
                This();
        }
    }
//CE:Recurrsive constructor
        Invocation
```

```
class Test{
        P S V m1(){
                m2();
        }
        P S V m2(){
                m1();
        }
        P s v main(String args[]){
                                        m1();
                sop("hello");
        }
}
//RE: Stackoverflow
```

Case2:

```
Class P{
}
Class C extends P{
}
```
(1)

```
class P{
        p(){
        }
}
Class C extends P{
        C(){
        }
}
```
(2)

```
class P{
        P(int i){
        }
}
class C extends P{
        C(){
        }
}
```
(3)

1. If parent class contains any argument constructor then while writing child classes, we have to take special care with respect to constructors.
2. Whenever we are writing any argument constructor it is highly recommended to write no-arg constructor also.

Case3:

```
Class P{
 P() throws IOException
{
}
}
Class C extends P{
C(){
Super();
}
}
```

```
class C extends P{
C() throws IOEXception|Exception|thrpwable
{
Super();
}
}
```

CE: UnReportedException java.IO.IOException in default constructor

If parent class constructor throws any checked exception compulsory child class constructor should throw same checked Exception or its parent otherwise code won't compile.

Interview:

Which of the following is valid

1. The main purpose of constructor is to create an object
   -Invalid
2. The main purpose of constructor is to perform initialization of object.
   -Valid
3. The name of the constructor need not be same as class name.
   -invalid
4. Return type concept applicable for constructors but only void
   -invalid
5. We can apply any modifier for constructor
   -invalid only public private protected default
6. Default constructor generated byt JVM
   -invalid
7. Compiler is responsible to generate default constructor.
   -valid

8. Compiler will always generate default constructor.
   -invalid, only if class does not provide constructor
9. If we are not writing no-arg constructor then compiler will generate default constructor.
   -invalid
10. Every no-arg constructor is always default constructor.
   -invalid
11. Default constructor is always no-arg constructor
   -valid
12. First line inside every constructor must be either super() or this(0, if we are not writing anything then compiler will generate this().
   - Invalid super()
13. For constructors both overloading and overriding concept applicable.
   -invalid only overloading
14. Inheritance concept is applicable for constructors but not overriding
   - invalid
15. Only concrete classes can constructor but abstract class cannot
   -Invalid
16. Interface can contain constructor
   -Invalid
17 recursive constructor invocation is a runtime exception
   -invalid
18. If parent class constructor throws some checked exception then compulsory child class constructor should throw the same checked exception or its child
   -invalid

Singleton Class:

- For any java class if we are allowed to create only one object such type of class is called singleton class
  Eg:Runtime, Service locator, Business delegate(Design pattern)
  Advantages:
  1. If several people have same requirement then it is not recommended to create a separate object for every requirement. We have to create only one object and we can reuse same object for every similar requirement so that performance and memory utilizations will be improved.
     This is the central idea of singleton classes
     Runtime r1=Runtime.getRuntime();
     Runtime r2=Runtime.getRuntime();
     Runtime rn=Runtime.getRuntime();
     There are n reference but only one object.

- How to create our own singleton classes:
  We can create our own singleton classes for this we have to use private Constructor and private static variable and public factory method.
  Approach1:
  Class Test
  {

```
        Private static Test t=new Test();                    Test t1=Test.getTest();
        Private Test();                                      Test t2=Test.getTest();
        {                                                    Test tn=Test.getTest();
        }
        Public static Test getTest(){
                Return t;
        }
}
```

Note: Runtime class is internally implemented b using this approach.

Approach2:

```
Class Test
{
        Private static Test t=null;                          Test t1=Test.getTest();
        Private Test();                                      Test t2=Test.getTest();
        {                                                    Test tn=Test.getTest();
        }
        Public static Test getTest(){
                If(t==null){
                        t=new Test();
                }
                return t;
        }
}
```

At any point of time for test class we can create only one object hence test class is singleton class.

Interview:

Class is not final but we are not allowed to create child classes how it is possible?

-by declaring every constructor as private we can restrict child class creation

```
 Class P{                          class C extends P{
        Private P(){}                       }                          //due to super();
}
```

For above class it is impossible to create child class.