* Inheritance :-

→ The process of getting Features [Data members & Methods] from one class to another class is called 'Inheritance'.

* Parent / super / Base class :-

→ The class which is giving features to some other class is called 'Parent class'.

* child / sub / ~~delivered~~ Derived class :-

→ The class which is taking features from some other class is called 'child class'.

* Note :-

→ Inheritance is also called as 'Reusability' (or) 'Derivation' (or) 'Sub classing' (or) 'Extendable classes'.

* Note :-

→ The 'Data members' & method from parent class are available 'logically' in child class. But it won't occupy any amount of memory space.

* Derived class :-

→ It contains features of parent class as well as its own class features

→ It contains more features compared to Base class

* Need of Inheritance :-

→ Wherever we want to 'reuse' the components then we go for Inheritance.

* Advantages of Inheritance :-

① Application development time is less

② Memory space is also taking less.

③ Execution time is also less.

④ Performance is improved.

⑤ Write once Run Anywhere [WORA].

* Types of Inheritance :-

① Single Inheritance.

② Multilevel Inheritance.

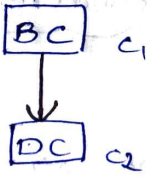
③ Hiearchical Inheritance.

④ Multiple Inheritance.

⑤ Hybrid Inheritance.

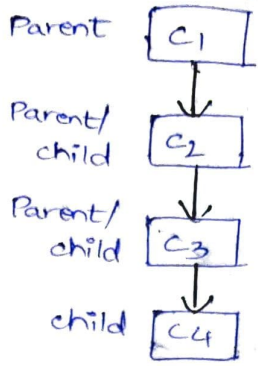
① Single Inheritance :-

→ There exists only one Base class & one Derived class.



② Multiple level Inheritance :-

→ There exists 'one Base class', 'one Derived class' and Multiple Intermediate Base classes.

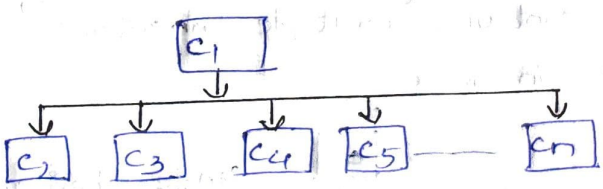


$C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4$

→ The classes which are act as sometimes Base/parent classes and sometimes child classes are called "Intermediate Base classes".

③ Hiearchical Inheritance :-

→ There exists 'single Base class' and 'Multiple Derived classes'.

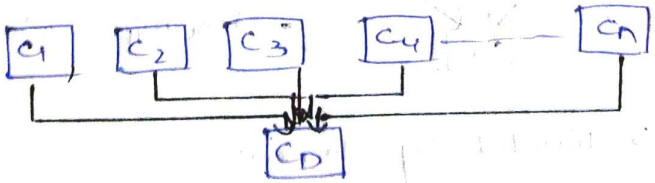


$C_1 \rightarrow C_2$, $C_1 \rightarrow C_3$, $C_1 \rightarrow C_4$, ---- $C_1 \rightarrow C_n$

④ Multiple Inheritance :-

→ There exists 'multiple Base classes' but 'only one Derived class'.

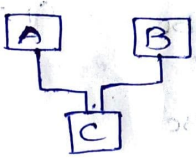
→ It supports in 'c++' but in 'java' it supports through Interfaces only.



Example :-

```
class A // Base class
{
  int a=2;
}

class B // Base class
{
  int a=3;
}
```




```
class C // Derived class
{
  //a=2  //a=3
  S.O.P(a); // Ambiguity [2 or 3].
}
```

↓
Compile Time Error.

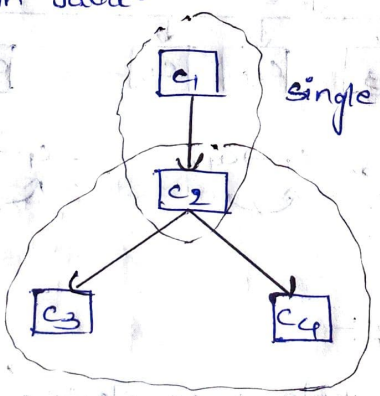
→ To avoid ambiguity we avoid multiple inheritance through classes.

⑤ Hybrid Inheritance :-

→ It is a combination of 'all available Inheritances'

→ If it contains 'multiple Inheritance' then it doesn't supports in Java.

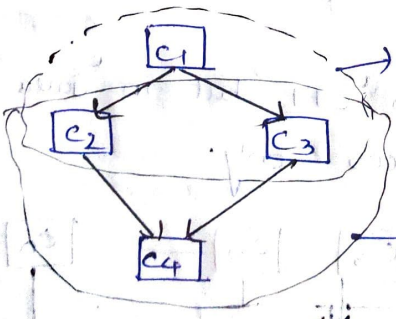
Ex :- ①



single Inheritance

Hierarchical Inheritance.

②



→ Hierarchical Inheritance

→ Multiple Inheritance

→ It is Invalid by classes.

→ But valid by Interfaces [User Defined Datatype].

* Syntax of Inheritance :

→ We have to use 'extends' keyword.

Ex : structure :

Doc :

```
class <clsname> extends <Baseclsname>
{
    variable, method( );
}
```

Ex: Single Inheritance

```
class C2 extends C1
{
}
```

```
class C1
{
}
```

Multilevel :

```
class C1
{
}
```

```
class C2 extends C1
{
}
```

```
class C3 extends C2
{
}
```

Hiearchieal :

```
class C1
{
}
```

```
class C2 extends C1
{
}
```

```
class C3 extends C1
{
}
```

Multiple :

```
class C0 extends C1, C2
{
}
// Compile Time Error
```

```
class C1
{
}
```

```
class C2
{
}
```

* Note :-

→ final int a=5; // constant
  static int b=5; // common to all
  int c=6; // Normal variable.

→ For 'final', value cannot be change but memory allocates many times.

→ For 'static', value & memory will not change.

* constant in java :-

→ final static int d=7; // constant

* Note :-

① 'constructors' are not allowed in Inheritance.

② 'static methods' are not allowed in Inheritance.

③ All 'final classes', all 'private methods/data members' won't participate in inheritance.

* Final keyword :-

→ We can use 'final keyword' before variables, methods & class levels.

→ All 'final data members' can't be 'initialized' & 'final methods' can't be 'override'.

* object class :-

→ It is responsible to perform Garbage collection.

→ By default in java, every class participates in Inheritance by object class. [java.lang.Object] superclass.

Ex :-

```
class Object
{
    ========
}
class Bc extends Object
{
    ========
}
```

* Relationships :-

(1) is-A :- Inheritance, Base class → child class
→ All inheritance classes and object class.

(2) has-A / Kind of :-

→ object of one class in another class.
Ex :- system.out.print();

(3) uses-A :-

→ object of one class in method of another class whenever we use Execution & Business classes.

Ex :- In main method, we creates object for Business class

* Program :- Ex :-

```
class BC
{
   int a;
   void display() { s.o.p("Display BC"); }
}
class DC extends BC
{
   int a;
   DC(int s, int y) { }
   void display()
   {
      s.o.p("Display DC");
   }
}
```

```
class EC
{
   P.S.vm(S a[])
   {
      DC d = new DC(5,6);
      d.display();
   }
}
```

* Super class (or) 'super' keyword :- Not supports for Multilevel

```
class BC
{
   int a;
   void display() {
      s.o.p("display BC");
   }
}
class DC extends BC {
   int a;
   DC(int x, int y)
   {
      this.a = x (or) a = x;
      super.a = y;
   }
   void display() {
      s.o.p("Display DC");
      super.display();
   }
}
class EC { }
{
   Psum(S a[])
   {
      DC d = new DC(5,6);
      d.display();
   }
}
```

* Note:-

```
class C1
{
    int a;
    void display();
    {
        ======
    }
}
class C2 extends C1
{
    int a;bi
    void display();
    {
        ======
    }
}
```

Data members + Methods

// Logically available in C_2

// override.

→ Whenever we 'override' logically thing then it will give priority for 'local variables' & 'Methods'.

→ If Local 'D.M' & 'Methods' are not present then only it goes for 'Base class' [Logically available].

→ 'constructors' doesnot participate in Inheritance because it is violating the 'Naming principle'. then compiler will treats as 'Normal method'.

→ By default, For 'every class' the topmost 'super class is 'object class' then every class is participates in Inheritance.

→ The 'object class' is responsible for removing unused space.

→ Because it contains 'finalize method'.

* Polymorphism:-

→ Many forms

Ex:- Method

overloaded
sum( );
sum(int);
sum(float);
sum(int, float).

overridden
sum( )
{
}

→ If any particular method going to refer during run time & same method at compile time then it is static polymorphism.

Ex:-
```
class A
{
    void dc()
    {
        s.o.p("Ad");
    }
}
class B extends A
{
    void dc()
    {
        s.o.p("Bd");
    }
}
B
```

```
B b=new B();
b.dc(); // Bd
b.jk(); // Error
```

→ During runtime refers to one method & execution time refers to another method is "Dynamic polymorphism"

→ It is for Memory ↓, Performance ↑.

→ It is based on Dynamic Binding → "Object for Base class."

→ Static Binding :- 'Private', 'Final', 'static methods' - Compile time.

* Note :-

```
class Fig
{abstract void areaC);
abstract Void area ( ) { sopC"Area of fig"); }

}
class Rect extends Fig
{
  void area ( ) { s.o.p (" Rec Area"); }

}
class square extends Fig
{
  void area ( ) { s.o.p("square Area"); }

}
Rect r = new Rect ( );
r. Area ( );

Fig f = new Rect ( );
f. area ( );
```

✗

✓

* Abstract class :-

→ Normal classes are concrete classes;

→ whenever we want a method without Body and to overcome the error we use 'Abstract method'.

→ Abstract method – common to all & overridden. & having their own body.

→ If a particular class contains atleast one abstract method then it is 'Abstract class'

→ For abstract classes we can't create 'objects'.

Note :-

→ If a particular class contains no Abstract methods we can also make it abstract class.

→ All the abstract classes in Java contain common features.

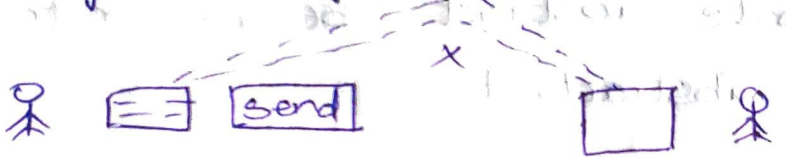
→ Every abstract class must have to participate in Inheritance. & should not belong to final class & static.

* Abstraction:

→ It is a process of "Hiding Implementation Details" (or) "unnecessary Details" (or) "Internal Details".

Ex:

① sending a Message



② Driving a car

break
gear
acceleration → Acceleration . Internal process
Accelator works/ gears are changing

Advantages of Abstract classes:

① It will reduce the complexity.

② It Avoids code Duplicates & Reusability.

③ Helps to improve Security of Apps.

④ Modularity (methods).

Interfaces :-

→ It is a 'user defined datatype' and to get 100% of abstraction' [security].

Need of Interfaces :-

① 'Multiple Inheritence' [Diamond problem].

② '100% Abstraction' (or) 'Total Abstraction'.

Abstract classes :- (0 to 100%) abstraction.

```
abstract class Hello
{
    abstract void itc();
    abstract void csec();
    abstract void Ecec();
    abstract void mecc();
}
```
100%.

```
abstract class Hai
{
    abstract void itc(){};
    abstract void csec(){};
    void Ecec(){};
    void mecc(){};
}
```
50%.

```
abstract class Hi
{
    void itc(){};
    void csec(){};
    void Ecec(){};
    void mecc(){};
}
```
0%.

→ To define an interface, we use a 'keyword' "interface".

→ Like class, 'interface' also a collection of 'data-members' and 'purely abstract methods'.

→ Need not to write 'abstract' before method by default it is abstract.ed.

Ex :-

```
interface Hello
{
    int age=100;   → Must Initialized [common to all] & final & static.
    void csec();
    void Ecec();
    void Mecc();
}
```
These methods are 'common to all' so 'public' and 'need not create memory' everytime then 'static' for variables.

→ It is 'not possible to create object' for interfaces as it contains all abstract methods. but we create 'References' [Indirectly]

definition :-

→ 'Interface' is an user defined Data type and having 100% security [abstraction].

→ It is a collection of "Public static final initialized data members" and "Public abstract methods"

Note :-

→ By default the 'variables' in 'interface' are 'public static final' and 'methods' are 'public abstract' so no need to mention every time.

structure :-

```
interface <interfacename>
{
   public static final Initialized variables;
   public abstract methods;
}
```

* Relation b/w Abstract class & concrete class :-

① AC ↓ ✓ C

② AC ↓ ✓ AC

③ C ↓ ✓ AC

④ C ↓ ✓ C

* Relation b/w Interface & classes :-

① Interface ↓ ✓ class

② Interface ↓ ✓ Interface

③ class ↓ ✓ Interface

[Defined things] ↓ ✓ [Undefined things]

class ✗ Interface