Abstract Modifier
==========
  It is the modifier applicable only for methods and class but not for variables.

Abstract Methods::
   Even though we don't have implementation still we can declare a method with abstract modifier.
   That is abstract methods have only declaration but not implementation.
   Hence abstract method declaration should compulsorily ends with semicolon.

Example
=======
```
   public abstract void methodOne();  => valid
   public abstract void methodOne(){} => Invalid
```

Child classes are responsible to provide implementation for parent class abstract methods.

Example
=======
Vehicle.java
============
```
abstract class Vehicle{
   public abstract int getNoOfWheels();
}
```

Bus.java
========
```
class Bus extends Vehicle{
     public int getNoOfWheels(){return 7;}
}
```

Auto.java
=========
```
class Auto extends Vehicle{
     public int getNoOfWheels(){return 3;}
}
```

Advantages of abstract methods
  a. By declaring abstract methods in parent class we can provide guide lines to the child class
     such that which methods they should compulsorily implement.

Note:: Abstract methods never speaks about implementation whereas if any modifier talks about
       implementation then the modifier will be enemy to abstract and that is always illegal
       combination for methods.

abstract => final
abstract => private
abstract => native
abstract => strictp
abstract => static
abstract => synchronized

Abstract class
==========
For any java class,if we dont want to create any object then such type of classes are refered as
abstract class.
Instantitation of abstract class is not possible.

example::
```
abstract class Test{
     public static void main(String... args){
         Test t=new Test();//CE:instantion not possible
      }
}
```

Rules of Abstract class
==========
a. If a class contains atleast one abstract method make that class also as abstract.
b. Even if the class does not contain any abstract method still the class can be made as abstract.
eg:: HttpServlet is an abstract class,but it does not contain abstract method.

eg1::
```
class Parent{
   public void methodOne();
}
```

eg2::
```
class Parent{
     public abstract void methodOne(){}
}
```
eg3::
```
abstract class Parent{
     public abstract void methodOne();
}
```

Inheritance over abstract class
==========
eg1::
```
abstract class Parent{
     public abstract void methodOne();
     public abstract void methodTwo();
}
class Child extends Parent{
     public void methodOne(){...}
}
```
output::CompileTimeError.

What is the difference b/w final and abstract?
For abstract methods compulsorily,we should override in the child class to provide implementation,whereas for final methods we cannot override hence abstract final comibnation is illegal for methods.

For abstract classes we should compulsorily create a child class to provide implementation

whereas for final class we cannot create child class. Hence final abstract implementation
is illegal for classes.

final classes cannot contain abstract methods whereas abstract class can contain final methods.

Example

```
final class A{
  public abstract void methodOne();//Invalid
}

abstract class A{
  public final void methodOne(){...}//Valid
}
```

What is the difference b/w strictfp and abstract?
strictfp:: It is a modifier applicable for classes and method, it is not applicable
for variables.
=> It is introduced in jdk1.2 version.
=> Result of floating point of arithmetic is varying from platform to platform.
to overcome this problem we use strictfp modifier.
=> If a method is declared as strictfp, then all floating point calculations in that method has
to follow IEEE754 standard, so that we will get platform independant result.

```
eg:: class Sample{
        public strictfp static void main(String... args){
              System.out.println(10.0/3); // 3.333
           }
     }
```

if the class is declared as strictfp, then all the concrete methods under that class
if they perform floating point arithmetic opertion will give platform independant results.

strictfp:: It speaks about implementation.
abstract:: It never speaks about implementation.
so combination of strictfp and abstract is illegal for methods, but it is legal for class.

Agenda
1. Introduction.
2. Normal or Regular inner classes
   - o Accessing inner class code from static area of outer class
   - o Accessing inner class code from instance area of outer class
   - o Accessing inner class code from outside of outer class
   - o The applicable modifiers for outer & inner classes
   - o Nesting of Inner classes
3. Method Local inner classes
4. Anonymous inner classes
   - o Anonymous inner class that extends a class
   - o Anonymous Inner Class that implements an interface
   - o Anonymous Inner Class that define inside method arguments
   - o Difference between general class and anonymous inner classes
   - o Explain the application areas of anonymous inner classes ?
5. Static nested classes
   - o Compression between normal or regular class and static nested class ?
6. Various possible combinations of nested class & interfaces
   - o class inside a class
   - o interface inside a class
   - o interface inside a interface
   - o class inside a interface
7. Conclusions

=> Sometimes we can declare a class inside another class such type of classes are called inner classes.

SUNMS came up with JDK1.0 in 1991 with features like
1. Platform Independent
2. Robust
3. OOPs
4. Secured
5. Simple
..
..
..

But java was not upto the mark in 2 areas
a. Performance was not upto the mark
b. GUI concepts(AWT)

Performance was improved through "JIT compiler".
=> Sun people introduced inner classes in 1.1 version as part of "EventHandling" to resolve GUI bugs.
=> But because of powerful features and benefits of inner classes slowly the programmers starts using in regular coding also.
=> Without existing one type of object if there is no chance of existing another type of object then we should go for inner classes.

Eg1
Without existing University object there is no chance of existing Department object
hence we have to define Department class inside University class.

```
class University{//Outer class
      class Department{//Inner class

      }
}
```

Eg2
Without existing Bank object there is no chance of existing Account object hence we
have to define Account class inside Bank class.

```
class Bank{//Outer class
      class Account{//Inner class

      }
}
```

Eg3:
Without existing Car Object, there is no chance of existing Engine Object hence we have to
define Engine class Inside Car class.

```
class Car{//Outer class
      class Engine{//Inner class

      }
}
```

Eg4:
Without existing Map object there is no chance of existing Entry object hence Entry interface is define inside Map interface.
Map is a collection of key-value pairs, each key-value pair is called an Entry.

```
interface Map{//Outer interface
      interface Entry<K,V>{//Inner interface

      }
   }
```

Note:
The relationship between outer class and inner class is not IS-A relationship and it is Has-A relationship.
It promotes Composition/Aggregation.

Based on the purpose and position of declaration all inner classes are divided into 4 types.
They are:
1. Normal or Regular inner classes.
2. Method Local inner classes.
3. Anonymous inner class.
4. static nested classes.

Regular inner class
==========
If we are declaring any named class inside another class directly without static modifier such type of inner classes are called normal or regular inner classes.
Example:
Outer.java

```
class Outer{
      class Inner{
      }
}
```

Output:

```
javac Outer.java
   Outer.class
```

```
   Outer$Inner.class

java Outer
      NoSuchMethodError/main method not found
java Outer$Inner
      NoSuchMethodError/main method not found

Example:
class Outer{
      class Inner{
      }
      public static void main(String[] args){
            System.out.println("outer class main method");
      }
}
Output:
 javac Outer
 java Outer
      outer class main method
 java Outer$Inner
      NoSuchMethodError/main method not found

=> Inside inner class we can't declare static members. Hence it is not
possible to declare main() method and we can't invoke
     inner class directly from the command prompt.
Example
class Outer {
      class Inner{
            public static void main(String[] args) {
                  System.out.println("Inner class main method");
               }
      }
}
Output: CE: can't declare static inside Inner class.


Accessing inner class code from static area of outer class:
============================================================
class Outer {
      class Inner{
            public void m1(){
                  System.out.println("inner class instance m1()");
            }
      }
      public static void main(String[] args) {
            Outer o= new Outer();
            Outer.Inner i=o.new Inner();
            i.m1();
      }
}
Output
=====
 javac Outer
 java Outer
   inner class instance m1()

Note:
Scenario1:
 Outer.Inner i= new Outer().new Inner();
    i.m1();
```

Scenario2:

```
   new Outer().new Inner().m1();
```

Accessing inner class code from instance area of outer class

==========

```
class Outer {
      class Inner{
            public void m1(){
                  System.out.println("inner class instance m1()");
            }
      }
      public void m2(){
            Inner i= new Inner();
            i.m1();
      }
      public static void main(String[] args) {
            Outer o= new Outer();
            o.m2();

      }
}
```

```
javac Outer
java Outer
  inner class instance m1()
```

Accessing inner class code from outside of outer class

==========

```
class Outer {
      class Inner{
            public void m1(){
                  System.out.println("inner class instance m1()");
            }
      }
}
class Test{
    public static void main(String[] args) {
            Outer o= new Outer();
            Outer.Inner i=o.new Inner();
            i.m1();

      }
}
```

Output

=====

```
javac Test
java Test
  inner class instance m1()
```

=>From inner class we can access all members of outer class (both static and non-static, private and non private methods and variables) directly.

example

```
class Outer {
      int x=10;
      static int y=20;
      class Inner{
            public void m1(){
                  System.out.println(x);
```

```
                System.out.println(y);
            }
      }
      public static void main(String[] args) {
            new Outer().new Inner().m1();
      }
}
javac Outer
java Outer
10
20
```

=> Within the inner class "this" always refers current inner class object. To refer current outer class object we have to use "outer class name.this".

```
class Outer {
      int x=10;
      class Inner{
            int x=100;
            public void m1(){
                  int x=1000;
                  System.out.println(x);
                  System.out.println(this.x);
                  System.out.println(Outer.this.x);

            }
         }
      public static void main(String[] args) {
            new Outer().new Inner().m1();
      }
}
```

Output

```
======
javac Outer
java Outer
1000
100
10
```

The applicable modifiers for outer classes are:

1. public
2. default
3. final
4. abstract
5. strictfp

But for the inner classes in addition to this the following modifiers also allowed.

1.private
2.protected
3.static

Nesting of Inner classes :

We can declare an inner class inside another inner class

```
class A {
      class B{
            class C{
                  public void m1(){
                        System.out.println("C class method");
```

```
                }
            }
      }
}
class Test{
      public static void main(String[] args) {
            A a = new A();
            A.B b = a.new B();
            A.B.C c = b.new C();
                  c.m1();
        }
}
javac Test
java Test
C class method

Note:
 new A().new B().new C().m1();

Method local inner classes:
=> Sometimes we can declare a class inside a method such type of inner
classes are called method  local inner classes.
=> The main objective of method local inner class is to define method
specific repeatedly required   functionality.
=> Method Local inner classes are best suitable to meet nested method
requirement.
=> We can access method local inner class only within the method where we
declared it.
     That is from outside of the method we can't access. As the scope of
method local inner classes is very less,
     this type of inner classes are most rarely used type of inner
classes.

eg::
class Outer {
            public void m1(){
                  class Inner{
                        public void sum(int x,int y){
                              System.out.println("The sum is ::"+(x+y));
                        }
                  }
                  new Inner().sum(10,20);
                        ;;;;
                        ;;;;
                        ;;;;
                  new Inner().sum(100,200);
                        ;;;;
                        ;;;;
                        ;;;;
                        ;;;;
                  new Inner().sum(1,2);

      }
      public static void main(String[] args) {
            new Outer().m1();
      }
}
javac Outer
java  Outer
 The sum is :: 30
```

```
 The sum is :: 300
 The sum is :: 3
```

=> If we are declaring inner class inside instance method then we can access both static and non static members of outer class directly.
=> But if we are declaring inner class inside static method then we can access only static members of outer class directly and
     we can't access instance members directly.

eg#1.

```
class Outer {
      int x=10;
      static int y=20;
      public  void m1(){
                  class Inner{
                        public void m2(){
                              System.out.println(x);
                              System.out.println(y);
                        }
                  }
                  Inner i=new Inner();
                  i.m2();
      }
      public static void main(String[] args) {
            new Outer().m1();

      }
}
```

Output

```
10
20
```

case2:
if we declare inner class inside static method then it would result in CE.

=> If we declare methodOne() method as static then we will get compile time error saying "non-static variable x cannot be referenced from a static context".
=> From method local inner class we can't access local variables of the method in which we declared it.
      But if that local variable is declared as final then we won't get any compile time   error.

eg#1.

```
class Outer {
      public  void m1(){
            final int x=10;
            class Inner{
                  public void m2(){
                              System.out.println(x);
                  }
            }
            Inner i=new Inner();
            i.m2();
      }
      public static void main(String[] args) {
            new Outer().m1();

      }
```

```
}

CaseStudy
=========
class Outer {
      int i=10;
      static int j=20;
      public void m1(){
            int k=30;
            final int l=40;

            class Inner{
                  public void m2(){
                        //line-1
                  }
            }
            Inner i=new Inner();
            i.m2();
      }
      public static void main(String[] args) {
            new Outer().m1();

      }
}
At line-1 how many variables we can access?
  i,j,l
if we make m1() as static how many variables we can access?
  j,l
if we make m2() as static how many variables we can access
 compile time error.

The only applicable modifiers for method local inner classes are:
1. final
2. abstract
3. strictfp


Anonymous inner classes:
=> Sometimes we can declare inner class without name such type of inner
classes are called  anonymous inner classes.

=> The main objective of anonymous inner classes is "just for instant
use".
=> There are 3 types of anonymous inner classes
      1. Anonymous inner class that extends a class.
      2. Anonymous inner class that implements an interface.
      3. Anonymous inner class that defined inside method arguments.

1. Anonymous inner class that extends a class.
class PopCorn{
      public void taste(){
            System.out.println("spicy");
       }
}
class Test {
      public static void main(String[] args) {
            PopCorn p=new PopCorn()
            {
                  public void taste(){
                        System.out.println("salty");
```

```
                }
            };
            p.taste();//salty
            PopCorn p1=new PopCorn()
            p1.taste();//spicy
      }
}
```

Analysis:

1. PopCorn p=new PopCorn();
   We are just creating a PopCorn object.

2. PopCorn p=new PopCorn(){
   };
   We are creating child class without name for the PopCorn class and for that child class we are creating an object with Parent PopCorn reference.

```
2. PopCorn p=new PopCorn()
   {
      @Override
      public void taste(){
         System.out.println("salty");
      }
   };
```

1. We are creating child class for PopCorn without name.
2. We are overriding taste() method.
3. We are creating object for that child class with parent reference.

Note:
Inside Anonymous inner classes we can take or declare new methods but outside of anonymous inner classes we can't call these methods directly because we are depending on parent reference.[parent reference can be used to hold child class object but by using that reference we can't call child specific methods]. These methods just for internal purpose only.

eg#1.
Example 1:

```
class PopCorn{
      public void taste(){
            System.out.println("spicy");
      }
}
class Test {
      public static void main(String[] args) {
            PopCorn p=new PopCorn(){
                  public void taste(){
                        methodOne();//valid call(internal purpose)
                  System.out.println("salty");
                    }
                  public void methodOne(){
                        System.out.println("child specific method");
                     }
              };
      //p.methodOne();//here we can not call(outside inner class)
      p.taste();//salty
      PopCorn p1=new PopCorn();
      p1.taste();//spicy
   }
}
```

Output:
Child specific method
Salty
Spicy

Example 2:

```
class Test {
      public static void main(String[] args){
            Thread t=new Thread(){
                  public void run(){
                        for(int i=0;i<10;i++){
                              System.out.println("child thread");
                        }
                  }
            };
            t.start();
            for(int i=0;i<10;i++){
                  System.out.println("main thread");
            }
      }
}
```

Anonymous Inner Class that implements an interface
==========

Example:

```
class InnerClassesDemo{
      public static void main(String[] args) {
            Runnable r=new Runnable(){ //here we are not creating for
Runnable interface are creating implements class object.

            public void run(){
                  for(int i=0;i<10;i++){
                        System.out.println("Child thread");
                  }
            }
             };
            Thread t=new Thread(r);
            t.start();
            for(int i=0;i<10;i++){
                  System.out.println("Main thread");
            }
      }
}
```

Anonymous Inner Class that define inside method arguments
==========

Example:

```
class Test {
      public static void main(String[] args) {
            new Thread(new Runnable(){
                  public void run(){
                        for(int i=0;i<10;i++){
                              System.out.println("child thread");
                        }
                  }
                  }).start();
            for(int i=0;i<10;i++){
                  System.out.println("main thread")}
            }
      }
```

```
}
Output:
=> This output belongs to example 2, anonymous inner class that
implements an interface example       and anonymous inner class that
define inside method arguments example.
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread

Difference between general class and anonymous inner classes:
General Class
1) A general class can extends only one class at a time.
2) A general class can implement any no. Of interfaces at a time.
3) A general class can extends a class and can implement an interface
simultaneously.
4) In normal Java class we can write constructor because we know name of
the class.

Anonmyous Inner class
1. Of course anonymous inner class also can extends only one class at a
time.
2) But anonymous inner class can implement only one interface at a time.
3) But anonymous inner class can extends a class or can implements an
interface but not
   both simultaneously.
4) But in anonymous inner class we can't write constructor because
anonymous inner class not
   having any name of the class.

Static nested classes
=====================
=> Sometimes we can declare inner classes with static modifier such type
of inner classes are called static nested classes.
=> In the case of normal or regular inner classes without existing outer
class object  there is no chance of existing inner class object.
      i.e., inner class object is always strongly associated with outer
class object.
=> But in the case of static nested class without existing outer class
object there may be a chance of existing static nested class object.
        i.e., static nested class object is not strongly associated with
outer class object.

eg#1.
```

Example:

```
class Test {
   static class Nested{
      public void methodOne(){
            System.out.println("nested class method");
      }
   }
   public static void main(String[] args){
      //Test.Nested t=new Test.Nested();
      //t.methodOne();

      Nested n=new Nested();
      n.methodOne();
   }
}
```

=> Inside static nested classes we can declare static members including main() method also.

Hence it is possible to invoke static nested class directly from the command prompt.

Example:

```
class Test {
      static class Nested{
            public static void main(String[] args){
                  System.out.println("nested class main method");
            }
      }
      public static void main(String[] args){
            System.out.println("outer class main method");
      }
}
```

```
java Test
outer class main method
java Test$Nested
nested class main method
```

=> From the normal inner class we can access both static and non static members of outer class but from static nested class we can access only static members of outer class.

Example:

```
class Test {
      int x=10;
      static int y=20;
      static class Nested{
            public void methodOne(){
                  System.out.println(x);//C.E:non-static variable x cannot be reference from a static context
                  System.out.println(y);
            }
      }
}
```

```
javac Test.java (Test.class, Test$Nested.class)
```

Compression between normal or regular class and static nested class ?

Normal /regular inner class

1) Without existing outer class object there is no chance of existing inner class object.

That is inner class object is always associated with outer class object.
2) Inside normal or regular inner class we can't declare static members.
3) Inside normal inner class we can't declare main() method and hence we can't invoke regular inner class directly from the command prompt.
4) From the normal or regular inner class we can access both static and non static members of outer class directly.

Static nested class
1) Without existing outer class object there may be a chance of existing static nested class object.
That is static nested class object is not associated with outer class object.
2) Inside static nested class we can declare static members.
3) Inside static nested class we can declare main() method and hence we can invoke static nested class directly from the command prompt.
4) From static nested class we can access only static members of outer class directly

Various possible combinations of nested class &interfaces :
1. class inside a class :
We can declare a class inside another class
Without existing one type of object, if there is no chance of existing another type
of object, then we should go for clas inside a class.

```
class University {
      class Department {
      }
}
```

Without existing University object, there is no chance of existing Department object.
i.e., Department object is always associated with University.

2. interface inside a class :
We can declare interface inside a class

```
  class X {
            interface Y {
                   }
}
```

Inside class if we required multiple implementation of an interface and these implementations of relevant to a particular class, then we should declare interface inside a class.

```
class VehicleType {
      interface Vehicle {
            public int getNoOfWheels();
        }
  class Bus implements Vehicle {
      public int getNoOfWheels() {
            return 6;
        }
  }
  class Auto implements Vehicle {
   public int getNoOfWheels() {
            return 3;
   }
  }
```

```
}
```

3. interface inside a interface :
We can declare an interface inside another interface.

```
      interface Map {
            interface Entry {
                  public Object getKey();
                  public Object getValue();
                  public Object setValue(Object newValue);
            }
      }
```

Nested interfaces are always public,static whether we are declaring or not. Hence we can implements inner inteface directly with out implementing outer interface.

```
interface Outer {
      public void methodOne();
      interface Inner {
            public void methodTwo();
      }
}
class Test implements Outer.Inner {
      public void methodTwo() {
            System.out.println("Inner interface method");
      }
      public static void main(String args[]) {
            Test t=new Test();
            t.methodTwo();
      }
}
```

Whenever we are implementing Outer interface , it is not required to implement Inner interfaces.

```
class Test implements Outer {
      public void methodOne() {
            System.out.println("Outer interface method ");
      }
      public static void main(String args[]) {
            Test t=new Test();
            t.methodOne();
         }
}
```

i.e., Both Outer and Inner interfaces we can implement independently.

4. class inside a interface :
We can declare a class inside interface. If a class functionality is closely associated with
the user interface then it is highly recommended to declare class inside interface.

Example:

```
interface EmailServer {
      public void sendEmail(EmailDetails e);
            class EmailDetails {
                  String from;
                  String to;
                  String subject;
            }
}
```

In the above example EmailDetails functionality is required for EmailService and we are
not using anywhereelse . Hence we can declare EmailDetails class inside EmailServiceinterface .
We can also declare a class inside interface to provide default implementation for that
interface.

Example :

```
interface Vehicle {
      public int getNoOfWheels();
      class DefaultVehicle implements Vehicle {
            public int getNoOfWheels() {
                  return 2;
            }
      }
}
class Bus implements Vehicle {
      public int getNoOfWheels() {
          return 6;
      }
}
class Test {
      public static void main(String args[]) {
            Bus b=new Bus();
            System.out.println(b.getNoOfWheels());//2
            Vehicle.DefaultVehicle d=new Vehicle.DefaultVehicle();
            System.out.println(d.getNoOfWheels());//3
      }
}
```

In the above example DefaultVehicle in the default implementation of Vehicle interface where as Bus customized implementation of Vehicle interface.
The class which is declared inside interface is always static ,hence we can create object directly without having outer interface type object.

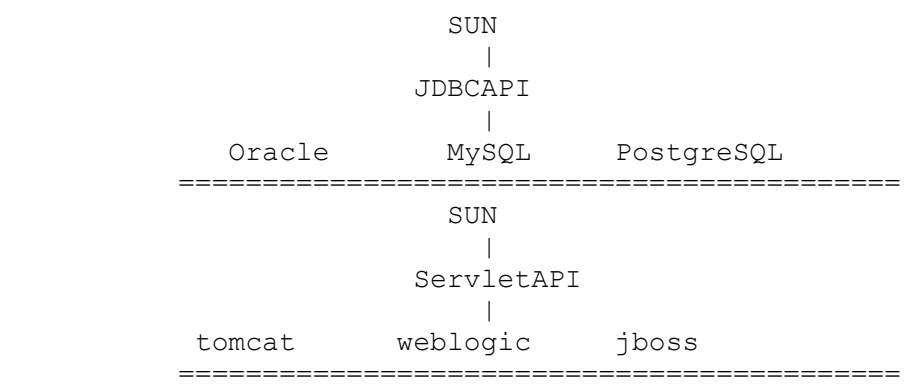
Conclusions :
1. We can declare anything inside any thing with respect to classes and interfaces.

```
      class A{
            class B{

            }
      }
      class A{
            interface B{//static

            }
      }
      interface A{
            interface B{//public static

            }
      }
      interface A{
            class B{//public static

            }
      }
```

2. Nesting interfaces are always public, static whether we are declaring or not.
3. class which is declared inside interface is always public,static whether we are declaring or not
4. interface declared inside a class is always static,but need not be public.

Def1::Any service requirement specification is called interface.

eg:: SUN people responsible to define JDBC API and database vendor will provide implementation for that



Def2::From client point of view an interface define the set of services what is expecting.
From service provider point of view an interface define the set of services what is "offering".
So interface acts a contract b/w client and serviceprovider.

eg:: GUI screen of ATM defines the set of services what the customer is expecting,
Bank people offered the same set of services what the customer is expecting.
Cusomter => GUI => Bank

Def3:: Inside interface every method is always abstract whether we are delcaring or not hence interface is considered as
100% pure abstract class.

Summary::
Interface corresponds to service requirement specification or contract b/w client and service provider or 100% pure abstract class.

Declaration and implementation of Interface
==========
a. Whenever we are implementing an interface compulsorily for every method of that interface
we should provide implentation otherwise we have to declare class as abstract class in that case child class is responsible to provide implementation for remaining methods.

b. Whenever we are implementing an interface method compulsorily it should be declared as public otherwise it would result in
CompileTime Error.

```
eg:: interface ISample{
        void methodOne();
        void methodTwo();
     }

     abstract class Sample implements ISample{
      public void methodOne(){...}
     }
```

```
class SubServiceProvider extends Sample{
   @Override
 public void methodTwo(){...}
}
```

Difference b/w extends vs implements

extends :: One class can extends only class at a time

eg::
```
class One{}
        class Two extends One{}
```

implements:: One class can implements any no of interface at a time.

eg::
```
interface One{
          public void methodOne();
         }
        interface Two{
         public void methodTwo();
        }
      class Demo implements One,Two{
           public void methodOne(){}
           public void methodTwo(){}
      }
```

2. A class can extend a class and can implements any no of interfaces simultaneously.

eg:
```
interface One{
          public void methodOne();
         }
        class Two{
         public void methodTwo(){}
        }
        class Three extends Two,implements One{
        @Override
          public void methodOne(){
              ....
          }

          @Override
          public void methodTwo(){
         ..
          }
       }
```

3. An interface can extend any no of interfaces at at time.

eg::
```
interface One{
                public void methodOne();
            }
            interface Two{
                 public void methodTwo();
            }
            interface Three extends One,Two{
               public void methodOne();
          public void methodTwo();
                 public void methodThree();
            }
```

Interface Methods
==========

Every method present inside the interface is public.

```
 Every method present inside the method is abstract.
   valid declarations
       a. void methodOne();
       b. public void methodOne();
       c. abstract void methodOne();
       d. public abstract void methodOne();

  public   => To make the method available for every implementation
class.
  abstract => Implementation class is responsible for providing the
implementation.

Since the methods present inside the interface is
   => public,abstract we can't we use the modfiers like
static,private,protected,strictfp,synchronized,native,final.

Interface variables
===================
  => Inside the interface we can define variables.
  => Inside the interface variables define requirement level constants.
  => Every variable present inside the interface is by default public
static final.

eg:: interface ISample{
           int x=10;
        }
     public :: To make it available for implementation class Object.
     static  :: To access it without using implementation class Name.
     final    :: Implementation class can access the value without any
modification.

variable declaration inside interface
  a. int x=10;
  b. public int x=10;
  c. static int x=10;
  d. final  int x=10;
  e. public static int x=10;
  f. public final int x=10;
  g. static final int x=10;
  h. public static final int x=10;

since the variable defined in interface is public static final,we cannot
use modifiers like private,protected,transient,volatile.
since the variable is static and final,compulsorily it should be
initialized at the time of declaration otherwise it would result in
compile time error.

eg:: interace IRemote{ int x;}// compile time error.

=> interface variables can be accessed from implementation class,but
cannot modify if we try to modify it would result in compile time error.

eg::  interface Remote{
         int VOLUME = 100;
      }
      class Lg implements Remote{
         public static void main(String... args){
            VOLUME=0;//CE:: cannot assign a value to final variable VOLUME
            System.out.println("value of volume is ::"+VOLUME);
            }
```

```
      }

eg:: interface Remote{
         int VOLUME = 100;
      }
      class Lg implements Remote{
         public static void main(String... args){
            int VOLUME=0;
            System.out.println("value of volume is ::"+VOLUME);//0
            }
      }
```

Interface Naming Conflicts
==========

Case 1::

If 2 interfaces contain a method with same signature and same return type in the implementation class only one method implementation is enough.

```
eg::
   interface Left{ public void methodOne();}
   interface Right{public void methodOne();}
   class Test implements Left,Right{
      @Override
        public void methodOne(){
            ...
        }
   }
```

Case2:

If 2 interfaces contain a method with same name but different arguments in the implementation class we have to provide implementation for both methods and these methods acts as a Overload methods.

```
eg::
   interface Left{ public void methodOne();}
   interface Right{public void methodOne(int i);}
   class Test implements Left,Right{
        @Override
        public void methodOne(){
             ...
        }

      @Override
        public void methodOne(int i){
            ...
        }
   }
```

case3::

If two interfaces contains a method with same signature but different return types then it is not possible to implement both interface simultaneously.

```
eg:: interface Left { public void methodOne(); }
        interface Right{ public int methodOne(); }
       class Test implements Left,Right{
        @Override
```

```
        public void methodOne(){
             ...
        }

      @Override
        public int methodOne(){
            ...
        }
    }
```

Variable naming conflicts::
   Two variables can contain a variable with same name and there may be a chance variable naming
   conflicts but we can resolve variable naming conflicts by using interface names.

example1:

```
  interface Left{ int x=888;}
  interface Right{ int x=999;}
  public class Test implements Left,Right{
      public static void main(String... args){
            System.out.println(Left.x);
            System.out.println(Right.x);
      }
  }
```

MarkerInterface
==========
   => If an interface does not contain any methods and by implementing that interface if our Object will get some ability such type of
      interface are called "Marker Interface"/"Tag Interface"/"Ability Interface".

   => example
           Serializable,Cloneable,SingleThreadModel.

example1
   By implementing Serializable interface we can send that object across the network and we can save state of an object into the file.

example2
   By implementing SingleThreadModel interfaace servlet can process only one client request at a time so that we can get "Thread Safety".

example3
   By implementing Cloneable Interface our object is in a position to provide exactly duplicate cloned object.

Without having any methods in marker interface how objects will get ability?
  Ans.JVM is responsible to provide requried ability.

Why JVM is providing the required ability to Marker Interfaces?
 Ans. To reduce the complexity of the programming.

Can we create our own marker interface?
  Yes, it is possible but we need to cusomtize JVM.

==========

Adapter class

==========

It is a simple java class that implements an interface only with empty implememtation for every method.
If we implement an interface compulsorily we should give the body for all the methods whether it
is required or not. This approach increases the length of the code and reduces readability.

```
eg:: interface X{
      void m1();
         void m2();
         void m3();
      void m4();
         void m5();
     }
    class Test implements X{
      public void m3(){
           System.out.println("I am from m3()");
         }
         public void m2(){}
         public void m3(){}
      public void m4(){}
         public void m5(){}
    }
```

In the above approach, even though we want only m3(), still we need to give body for all the abstract methods, which increase the
length of the code, to reduce this we need to use "Adapater class".
Instead of implementing the interface directly we opt for "Adapter class".
Adapter class are such classes which implements the interface and gives dummy implementation for all the abstract methods of interface.
So if we extends Adapter classes then we can easily give body only for those methods which are interested in giving the body.

```
eg::
interface X{
      void m1();
         void m2();
         void m3();
      void m4();
         void m5();
}
abstract class AdapaterX implements X{
      public void m1(){}
         public void m2(){}
         public void m3(){}
      public void m4(){}
         public void m5(){}
}
class TestApp extends AdapterX{
      public void m3(){
            System.out.println("I am from m3()");
      }
}
```

```
eg:: interface Servlet{....}
```

```
abstract class GenericServlet{}
abstract class HttpServlet{}
class MyServlet extends HttpServlet{}
```

Note:: Adapter class and Marker interface are big utilites to programmer to simplify programming.

When to go interface, abstract class and concrete class?
  interface:: It is prefered when we speak only about specification.
  abstract class:: It is prefered when we speak about partial implementation.
  concreate class:: It is prefered when we speak about complete implementation and ready to
                    provide service then we go for concrete class.

Difference b/w Abstract class and Interface?
  Interface:: If we dont know anything about implementation just we have requirement specification then we should go for interface.
  Abstract class: If we are talking about implementation but not completely then we should go for abstract class.

  Interface:: Every method present inside the interface is always public and abstract whether we are declaring or not.
  Abstract :: Every method present inside abstract class need not be public and abstract.

  Interface:: We can't declare interface methods with the modifiers like private,protected,final,static,synchronized,native,strictfp.
  Abstract :: There are not restrictions on abstract class method modifiers.

  Interface:: Every interface variable is always public static final whether we are declaring or not.
  Abstract:: Every abstract class variable need not be public static final.

  Interface:: Every interface variable is always public static final we can't declare with the
                    following modifiers like
private,protected,transient,volatile.
  Abstract:: No restriction on access modifiers

  Interface:: For every interface variable compulsorily we should perform initialisation at the time of declaration,
                otherwise we get compile time error.
  Abstract:: Not required to perform initialisation for abstract class variables at the time of declaration.

  Interface:: Inside interface we can't write static and instance block.
  Abstract :: Inside abstract class we can write static and instance block.

  Interface:: Inside interface we can't write constructor.
  Abstract :: Inside abstract class we can write constructor.

Note::

What is the need of abstract class? can we create an object of abstract class, Does it contains constructor?
   => abstract class object cannot be created becoz it is abstract.
   => But constructor is need for constructor to initialize the object.

```
eg::
class Parent {
      Parent() {
            System.out.println(this.hashCode());
      }
}
class Child extends Parent {
      public Child() {
            System.out.println(this.hashCode());
      }
}
public class TestApp {
      public static void main(String[] args) {
            Child c = new Child();
            System.out.println(c.hashCode());
      }
}
```

Note::Every method present inside the interface is abstract, but in abstract class also we take only abstract methods then
          what is the need of interface concept?
     => we can replace interface concept with abstract class, but it is not a good programming practise.

```
interface X{
      ...
      ...
}
class Test implements X{
      ...
        ...
}
  Test t=new Test();
```

i. performance is high.
ii.While implementing X we can extends
   one more class,through which we can bring reusablity.

```
eg#2.
  abstract X{
      ...
      ...
  }
  class Test extends X{
        ...
        ...
  }
    Test t=new Test();
```

i.performance is high.
ii.While extending X we can't extends
   any other classes so reusablity is not brought.

If everything is abstract then it is recommended to go for interface.

constructor:: It is used to perform initialisation of object.
              It is used to provide values for instance variables.
            abstract class can contain instance variables, which are required for child
              class object to perform initialisation of these instance variables so constructor
              is required inside abstract class.

Why abstract class can contain constructor where as interface doesnot contain constructor?
   abstract class => it is used to perform initialization of the object.
                              it is used to provide the value for the instance variable.
                              it is used to contain instance variable which are requried for child
                              object to perform intialisation for those instance variables.

    interface => every variable is always static,public and final their is no chance of existing instance variable inside the class.
                               so we should perform initialisation at the time of declaration.
                               so constructor is not required for interface.

eg#1.
```
abstract class Person{
      String name;
      int age;
      int height;
      int weight;

      Person(String name,int age,int height,int weight){
            super();
            this.name=name;
            this.age=age;
            this.height=height;
            this.weight weight;
      }
}

class Student extends Person{

      int rollno;
      int marks;

        Student(String name,int age,int height,int weight,int rollno,int marks){
            super(name,age,height,weight,rollno);
            this.rollno=rollno;
            this.marks=marks;
      }
}
```

Method Overriding
==========
Whatever the parent has by default available to child through inheritance,if the child is not satisifed with parent class method implementation then child is allow to redefine that parent class method in child class in its own way this process is called as "Overriding".

The parent class method which is overriden is called "Overridden method".
The child class method which is overriding is called "Overriding method".

```
eg1::
class Parent{
      public void property(){System.out.pritnln("Gold+cash+land");}
        public void marry(){System.out.pritnln("Relative girl only u
should marry!!!");}
}
class Child extends Parent{
       public void property(){System.out.pritnln("Gold+cash+land");}

       @Override
       public void marry(){System.out.println("I will marry
Deepika!!!");}

}
class Test{
      pubilc static void main(String[] args){
            Parent p=new Parent();
            p.marry();//Relative girl only u should marry!!!

            Child c=new Child();
            c.marry();//I will marry Deepika!!!

            Parent p =new Child();
            p.marry();//I will marry Deepika!!!
      }
}
```

In method overriding, method resolution is always based on the runtime object created by JVM, so we say method overriding as "RunTime Polymorphism/Dynamic dispatch/Late Binding".
Here reference type is just like a dummy template.

Rules of Overriding
==========
To apply the rules of Overriding, the method signature should be same only then the methods is participating in Overriding.

Note::
1. In overriding method names and arguments must be same. That is method signature must be same.
2. Until 1.4 version the return types must be same but from 1.5 version onwards covariant return types are allowed.
3. According to this Child class method return type need not be same as Parent class method return type its Child types also allowed.

```
eg#1.
class Parent{
      public Object methodOne(){ return null;}
}
```

```
class Child extends Parent{
      public String methodOne(){return null;}
}
```
It compiles succesfully.

```
case studies::
    Object(P) => String(C)  //valid
    Number(P) => Integer(P) //valid
    String(P) => Object(C) //invalid
   Integer(P) => int(c)    //invalid
```

Rule::Co-Variant type is applicable only on objects not on primitive types.

Scenario2::
Private methods are not visible in the Child classes hence overriding concept is not applicable for private methods. Based on own requirement we can declare the same Parent class private method
in child class also. It is valid but not overriding.

eg#1.
```
class Parent{ private void methodOne(){}}
class Child extends Parent{private void methodOne(){}}
```

2. If the method is declared as final in parent class then those methods we cannot override in
   child class,it would result in compile time error.

```
    eg1:: class Parent{public final void methodOne(){}}
          class Child extends Parent{public void methodOne(){}}//Compile
```
time error

3. Parent class non final methods can be made as final in child class
```
    eg1:: class Parent{public  void methodOne(){}}
          class Child extends Parent{public final void methodOne(){}}
```

4. In the parent class, if the method is abstract then in the child class we should compulsory override to give the implementation
```
       eg1:: abstract class Parent{public abstract void methodOne();}
             class Child extends Parent{public void methodOne(){}}
```

5. We can override NonAbstract method as abstract, this approach is helpful to stop the availablity of parent method implementation to next level child classes.
```
      eg1:: class Parent{public  void methodOne(){}}
            abstract class Child extends Parent{public abstract void
```
methodOne();}

```
Note:: final => Nonfinal (invalid)
       Nonfinal => final (valid)
       abstract => nonabstract(valid)
       native   => nonnative(valid)
       synchronized => nonsychronized(valid)
       strictfp     => nonstrictfp(valid)
```

Rules with respect to scope
==========
While Overriding we cannot reduce the scope of accessmodifier

```
   eg1:: class Parent{public  void methodOne(){}}
         class Child extends Parent{protected void
methodOne(){}}//Compile time error
Note
public => public
protected => protected/public
default   => default/protected/public
private   => Overriding concept is not applicable

Overrding w.r.t static methods
==============================
1. we can't override static method as instance method, it would result in
compile time error.

eg#1.
class Parent{
      public static void methodOne(){ }
}
class Child extends Parent{
      public void methodOne(){}
}

2. we can't override instance method as static,it would result in compile
time error.

eg#2.
class Parent{
      public void methodOne(){ }
}
class Child extends Parent{
      public static void methodOne(){}
}

3.
class Parent{
      public static void methodOne(){ }
}
class Child extends Parent{
      public static void methodOne(){}
}
It seems to be overriding,but it is not overriding it is method hiding.
In case of static method, compiler will take the resolution of method
call.

Difference b/w method Overriding and method hiding?
Overriding =>In both parent and child class method should be instance
           Method resolution is based on runtime object.
           JVM will take the call so it is called as
runtimepolymorphsim/dynamic dispatch.

Hiding     =>In both parent and child class method should be static.
           Method resolution is based on reference type.
           Compiler will take the call so it is called as Compile time
polymorphism.

eg#1.
class Parent{
      public static void methodOne(){System.out.println("I am from
parent"); }
}
```

```
class Child extends Parent{
      public static void methodOne(){System.out.println("I am from
child");}
}
public class Test {
      public static void main(String[] args) {
                  Parent p= new Parent();
                  p.methodOne();//I am from parent

                  Child c=new Child();
                  c.methodOne();//I am from child

                  Parent p1=new Child();
                  p1.methodOne();//I am from parent
      }
}
```

Overriding w.r.t var arg method
==========
var arg method should be overriden with var arg only,if we try to override w.r.t normal method
then it would become Overloading nor overriding.

```
eg#1.
class Parent{
      public  void methodOne(int... i){System.out.println("I am from
parent"); }
}
class Child extends Parent{
      public  void methodOne(int i){System.out.println("I am from
child");}
}
public class Test {
      public static void main(String[] args) {
                  Parent p= new Parent();
                  p.methodOne(10);// I am from Parent

                  Child c=new Child();
                  c.methodOne(10);.// I am from Child

                  Parent p1=new Child();
                  p1.methodOne(10);// I am from Parent
      }
}
```

If we replace child class method with var arg method then it will become Overriding.

```
eg#2.
class Parent{
      public  void methodOne(int... i){System.out.println("I am from
parent"); }
}
class Child extends Parent{
      public  void methodOne(int... i){System.out.println("I am from
child");}
}
public class Test {
      public static void main(String[] args) {
                  Parent p= new Parent();
```

```
                p.methodOne(10);//I am from parent

                Child c=new Child();
                c.methodOne(10);//I am from child

                Parent p1=new Child();
                p1.methodOne(10);//I am from child
      }
}
```

Overridng w.r.t variables
==========
1. Overriding concept is not applicable for variables.
2. In case of Overriding compiler will resolve the call based on the reference type.

eg#1.

```
class Parent{int x= 888;}
class Child extends Parent{int x=999;}
public class Test {
      public static void main(String[] args) {
            Parent p= new Parent();
            System.out.println("X value is ::"+p.x);//X value is :: 888

            Child c=new Child();
            System.out.println("X value is :: "+c.x);//X value is :: 999

            Parent p1=new Child();
            System.out.println("X value is :: "+p1.x);//X value is :: 888
      }
}
```

In the above pgm,if the variable is static then also there is no change in the output,becoz compiler will resolve the call based on the reference type.

Difference b/w Overloading vs Overriding?

```
     Overloading
MethodName     : same
ArgumentType   : must be different
MethodSignature: same
  return type  : no restrictions
 access modifer: no restrictions
private,final  : no restrictions
   static
  throws       : no restrictions
 binding       : compiler
alternativename: CompileTimebinding/early binding/eager binding/static
binding.

     Overriding
MethodName     : same
ArgumentType   : must be same
MethodSignature: same
  return type  : same or covariant type
 access modifer: same or increase the
scope(private,default,protected,public)
private,final  : can't be Overriden
   static
```

throws : if child class throws some exception then parent should throw the same or its
Parent type(rule applicable only for CheckedException not for UnCheckedException)
binding : JVM(based on run time object)
alternativename: RunTimePolymorphism/lazy binding/dynamic dispatch.

Note:
1. In overloading we have to check only method names (must be same) and arguments
(must be different) the remaining things like return type extra not required to check.
2. But In overriding we should compulsory check everything like method names, arguments, return types, throws keyword, modifiers etc.

Consider the method in parent class
Parent: public void methodOne(int i)throws IOException

In the child class which of the following methods we can take..
1. public void methodOne(int i) //valid(overriding)
2. private void methodOne()throws Exception//valid(overloading)
3. public native void methodOne(int i);//valid(overriding)
4. public static void methodOne(double d)//valid(overloading)
5. public static void methodOne(int i)
Compile time error :methodOne(int) in Child cannot override methodOne(int) in Parent; overriding method is static
6. public static abstract void methodOne(float f) Compile time error :
1. illegal combination of modifiers: abstract and static
2. Child is not abstract and does not override abstract method methodOne(float) in Child