interface :
----

interface **[Loose coupling + Runtime Polymorphism]**

Upto JDK 1.7

From JDK 1.8 onwards

interface JDK 1.7V :
----

* An interface is a keyword in java which is used to define an interface. [Naming convention -> Adjectives]

* An interface defines the **"Working functionality of a class"**

* It is contract between the client and developer where developer must need to fulfill all the clinet requirements.

Example :

```
interface Client
{
   void onlineTransaction();
   void dbSecurity();
   void onlineSecurity();
}
class TCS implements Client
{
     //Implement all the clinet requirements.
}
```

* An interface contains only abstract method that means inside an interface there is a gurantee that only abstract methods are available.

* Interface methods are bydefault **public and abstract** so at the time of overriding, We cannot reduce the visibility.

* In order to implement the abstract methods of an interface, We should use "implements" keyword.

* Actually an interface defines "**WHAT TO DO** ?" whereas it's implementer classes define "**HOW TO DO**?"

* An inetraface does not contain **non static field, constructor, static and non static block.**

* Interface variables are by default **public, static and final.**

* It does not support constructor so, We can achieve multiple Inheritance.
[Never say that interafce came in java to support multiple inheritance, Only we can achieve it]

* THE MAIN PURPOSE OF INTERFACE TO ACHIEVE LOOSE COUPLING AND DYNAMIC POLYMORPHISM [SERIOUS POLYMORPHISM]

* We can achieve 100% abstraction by using interface.

* We can't create an object for interface. [Interanlly It is a pure abstract class]

* All the abstract methods of an interface must be overridden in the implementer classes otherwise implementer class will become as an abstract class.

```
package com.ravi.interface_demo;

public sealed interface Moveable permits Car
{
   int MAX_SPEED = 120; //public + static + final

   void move(); //public + abstract
}

package com.ravi.interface_demo;

public non-sealed class Car implements Moveable
{
    @Override
    public void move()
    {
        System.out.println("Car is moving with :"+MAX_SPEED+ " KM/HR");
    }
}

package com.ravi.interface_demo;

public class InterfaceDemo1
{
    public static void main(String[] args)
    {
        Moveable m = new Car() ;
        m.move();
        System.out.println("Speed of the Car is :"+Moveable.MAX_SPEED);

    }

}
```

```
package com.ravi.interface_demo;

interface Banking
{
    void deposit(double amount);
    void withdraw(double amount);
    double getBalance();
}
class Customer implements Banking
{
    private double balance;

    public Customer(double balance)
    {
        super();
        this.balance = balance;
    }

    @Override
    public void deposit(double amount)
    {
        if(amount <=0)
        {
            System.err.println("Amount is Invalid");
        }
        this.balance = this.balance + amount;

    }

    @Override
    public void withdraw(double amount)
    {
        if(amount > this.balance)
        {
            System.err.println("Insufficient Balance!!!");
        }
        this.balance = this.balance - amount;
    }

    @Override
    public double getBalance()
    {
        return this.balance;
    }
}

public class InterfaceDemo2
{
  public static void main(String[] args)
  {
    Customer scott = new Customer(10000);
    System.out.println("Current Balance is :"+scott.getBalance());

    scott.deposit(25000);
    System.out.println("Current Balance is :"+scott.getBalance());

    scott.withdraw(5000);
    System.out.println("Current Balance is :"+scott.getBalance());

   }
}
```

//Program on Loose coupling and Dynamic Polymorphism :

Program on loose coupling : (Industry Standard Program)

Loose Coupling :- If the degree of dependency from one class object to another class is very low then it is called loose coupling. [interface is reqd]

Tightly coupled :- If the degree of dependency of one class to another class is very high then it is called Tightly coupled.

According to IT industry standard we should always prefer loose coupling so the maintenance of the project will become easy.

High Cohesion [Encapsulation]:

Our private data must be accessible via public methods (setter and getters) so, in between data and method we must have high cohesion.
(tight coupling) so, validation of outer data is possible.

//Program :

```
package com.ravi.loose_coupling_demo;

public interface HotDrink
{
    void prepare();
}

package com.ravi.loose_coupling_demo;

public class Tea implements HotDrink
{
    @Override
    public void prepare()
    {
        System.out.println("Preparing Hyderabad Irani Tea!!!!");
    }
}

package com.ravi.loose_coupling_demo;

public class Coffee implements HotDrink
{
    @Override
    public void prepare()
    {
        System.out.println("Preparing Filter Coffee!!");
    }
}

package com.ravi.loose_coupling_demo;

public class Boost implements HotDrink
{
    @Override
    public void prepare()
    {
        System.out.println("Preparing Boost!!!");
    }

}

package com.ravi.loose_coupling_demo;

public class Restaurant
{
  public static void prepareHotDrink(HotDrink hd) //hd = new Tea();
  {
     hd.prepare();
  }
}

package com.ravi.loose_coupling_demo;

public class LooseCouplingDemo
{
    public static void main(String[] args)
    {
        Restaurant.prepareHotDrink(new Tea());
        Restaurant.prepareHotDrink(new Coffee());
        Restaurant.prepareHotDrink(new Boost());
    }
}
```