interface programs :
```
package com.ravi.interface_demo_ex;

abstract interface Movable
{
    int SPEED = 90;  //public + static + final
    void move();     //public + abstract
}

class Car implements Movable
{
    @Override
    public void move()
    {
        //SPEED = 120;  //field is final so Re-assignment is not possible
        IO.println("Car speed is :"+SPEED+" KM/HR");
    }
}

public class InterfaceDemo1
{
    public static void main(String[] args)
    {
        Movable moveable = new Car();
        moveable.move();

        IO.println("Moveable object Speed is :"+Movable.SPEED+" KM/HR");
    }

}
```
Note : From the above program, We can say : An interface defines "WHAT TO DO" on the other hand it's implementor classes defines "HOW TO DO"

```
package com.ravi.interface_demo_ex;

interface Bank
{
    void deposit(double amount);
    void withDraw(double amount);
    double getBalance();
    String getName();
}

class Customer implements Bank
{

    private String name;
    private double balance;


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

    @Override
    public void deposit(double amount)
    {
        if(amount <=0)
        {
            System.err.println("Deposit amount cannot be zero OR Negative");
            System.exit(0);
        }

        this.balance += amount;
        IO.println(amount +" amount deposited successfully in the "+this.getName()+" Account");
    }

    @Override
    public void withDraw(double amount)
    {
        if(amount > this.balance)
        {
            System.err.println("Sorry!!! Insufficient Balance");
            System.exit(0);
        }

        this.balance -= amount;
        IO.println(amount +" amount has withdrawn successfully from "+this.getName()+" Account");

    }

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

    @Override
    public String getName()
    {
        return this.name;
    }


}

public class InterfaceDemo2 {

    public static void main(String[] args)
    {
        Bank bank = new Customer("Mr Alen", 50000);
        IO.println(bank.getName()+" has "+bank.getBalance()+ " amount in the account");
        bank.deposit(10000);
        IO.println(bank.getName()+" has "+bank.getBalance()+ " amount in the account");
        bank.withDraw(30000);
        IO.println(bank.getName()+" has "+bank.getBalance()+ " amount in the account");

    }

}
```

Assignement :
```
interface Calculate
{
    void sum(double x, double y)  {4 methods}
}

class ArithmeticOperation implements Calculate
{
}
```

[Interface Program on Loose coupling :
OR
Serious Polymorphism
OR
LCRP [Loose Coupling Runtime Polymorphism) Architecture

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 required]

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.
```

//Programs :
```
package com.ravi.loose_coupling_arch;

public interface HotDrink
{
    void prepare();
}
```
```
package com.ravi.loose_coupling_arch;

public class Tea implements HotDrink
{
    @Override
    public void prepare()
    {
        IO.println("Preparing Iran1 Tea!!! ");
    }
}
```
```
package com.ravi.loose_coupling_arch;

public class Coffee implements HotDrink
{
    @Override
    public void prepare()
    {
        IO.println("Preparing Java beans coffee");
    }

}
```
```
package com.ravi.loose_coupling_arch;

public class Boost implements HotDrink {

    @Override
    public void prepare()
    {
        IO.println("Preparing Boost");
    }

}
```
```
package com.ravi.loose_coupling_arch;

public class Restaurant
{
    public static void prepareHotDrink(HotDrink hd)  //hd = new Tea();
    {
        hd.prepare();
    }
}
```
```
package com.ravi.loose_coupling_arch;

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

}
```
Factory Design Pattern :

* It is possible to take interface as a return type of the method which is known as Factory Design Pattern.

* Here We have a facility to return multiple objects i.e all the implementer classes objects as shown in the example:
```
public HotDrink accept()
{
  return new Tea() OR new Coffee() OR new Boost(); [In future We can add more classes]
}
```

Interface Java 1.8V