Chapter 9 : Interfaces

1. Abstract class :

➔ We know that sometime we do need a class which gives methods that can be expressed differently through its subtypes. Also its absurd to create objects of these classes so compiler gives an option so that objects of them can't be created. This can be done through 'Abstract' methods and a class with one or more abstract methods is qualified to be called as 'Abstract' class.

➔ (KnS)Instance of the abstract class cannot be created, it should be extended. Similarly abstract method is the one which should be overridden.

➔ (KnS)Abstract method is the one which are not been implemented. The abstract methods ends up with semicolon rather than braces.

➔ (KnS)Abstract class can have both abstract and non-abstract methods.

➔ (KnS)A class cannot be both abstract and final because they are exactly opposite of each other and code wont compile.

➔ A class with abstract method must be marked abstract.

➔ If one inherit from an abstract class and if one want to create an object of the new derived class then first he has to implement all the abstract methods of abstract base class. If one don't then the new class will also be qualified as an Abstract class.

➔ When a concrete class implement abstract methods of abstract class then same rules of overridding applies to it but abstract methods of interfaces must be marked public.

➔ (KnS) An abstract class can implement an interface. And an concrete class extending such abstract class should implement all the abstract methods.

Example : abstract class Ball implements Bounceable {}

```
package ch9Interfaces;

interface InterfaceClass1
{
	void IntfMethod();
}

abstract class AbClass1 implements InterfaceClass1
{
	abstract void absMethod();
    //void printint();//Error
}
public class InterfaceClass extends AbClass1 implements InterfaceClass1
{
	public static void main(String args[])
	{
		InterfaceClass ic = new InterfaceClass();
		ic.absMethod();
```

```
              ic.IntfMethod();
        }
            public void absMethod()
            {

                    System.out.println("This is a abstract method");

            }
              public void IntfMethod()//overridden method
              {
                    System.out.println("this is interface method");
              }
}
Output:
This is a abstract method
this is interface method
```

➔ One can have an Abstract class without any abstract methods if one want to have a class but don't want to create an object of that class. Example is shown below.

```
package ch9Interfaces;

abstract class AbsWOAbsMethod1
{
      void method1()//methods is implementd hence not marked abstract
      {
            System.out.println("method1");
      }
      void method2()
      {
            System.out.println("method2");
      }
}

class AbsWOAbsMethod2 extends AbsWOAbsMethod1
{
      void methodAbs()
      {
            method1();
      }
}

public class AbsWOAbsMethod {

      public static void main(String[] args)
      {

            //ClassAbs ab1 = new ClassAbs();//error:cant instanstiate
                                            //the type ClassAbs
            AbsWOAbsMethod2 abs = new AbsWOAbsMethod2();
            abs.methodAbs();
            abs.method2();
      }
}
```

```
Output:
method1
method2
```

Example : 2

```
package ch9Interfaces;

abstract class NoAbsMethClass1
{
	void method1()
	{
		System.out.println("method1");
	}

	void method2()
	{
		System.out.println("method2");
	}
}

abstract class NoAbsMethClass2 extends NoAbsMethClass1
{
	void method3()
	{
		System.out.println("method3");
	}
}

public class NoAbsMethClass extends NoAbsMethClass2
{
	public static void main(String[] args)
	{
		NoAbsMethClass na = new NoAbsMethClass();
		na.method1();
		na.method2();
		na.method3();
	}
}
Output:
method1
method2
method3
```

Note : Above we can see an abstract class can be extended by both an another abstract class and concrete class even if it don't have abstract method. Also it can be noted that a class with an abstract method has to be marked abstract but not vice versa i.e. an abstract class don't need to have an abstract method.

2. Interfaces :

➔ Interfaces are nothing but fully abstract classes as none of its methods have any bodies.

- (KnS) Interfaces have all the methods as abstract method i.e. if an abstract class have all the methods as abstract then it should be marked as interface rather.
- An Object for interface or abstract class cannot be created.
- Interface is created by using 'interface' keyword.
- Class needs to implement an interface and a class can implement any number of interfaces separated by commas. Example : class A implement interface1, interface2, interface3.
- (KnS) Interface has(implicitly)
    - Interface: public and abstract
    - Method : public and abstract
    - Variable : public static final
- During implementation in a concrete class all methods of interface should be marked as 'public' else compiler will give error.
- For using an interface, a class should impelement it first.

Example : `Class A implement interface1`

- (KnS) An interface can extend one or more other interfaces.
- (KnS) Interface can't extend anything but other interfaces.
- (KnS) public abstract interface Rollable {}

  Public interface Rollable {}

Both are legal, but no need to mention 'abstract' in first statement as interfaces are implicitly abstract.

- (KnS) As variables in interface are public static final implicitly, so classes which implement them directly inherit these constant variable too.
- (KnS) An abstract class can implement an interface. And an concrete class extending such abstract class should implement all the abstract methods.

Ex. `abstract class Ball implements Bounceable {}`

- An interface can be marked abstract and its method too can be marked abstract but its not needed as interfaces and its methods are implicitly abstract.
- When a class(abstract or concrete) and an interface have the same method(concrete method inside class) name, return type and access specifier(public in this case) and if a subclass extends and implement them then its not mandatory to implement common method in subclass.

```
package ch9Interfaces;

interface InterfaceFight
{
	void fight();
}

interface InterfaceBright
{
      void bright();
}

interface InterfaceTight
{
      void tight();
}

/*abstract*/ class InterfaceClassShareMethod1
{
      public void fight()//This must be public as all abstract methods of interface
                         //has to be marked public else it needs to be overridden
                         //inside interface implementing class
      {
            System.out.println("InterfaceClassShareMethod1 class");

      }

      //abstract void abstmethod();
}

public class InterfaceClassShareMethod extends InterfaceClassShareMethod1
implements InterfaceFight, InterfaceBright, InterfaceTight
{
/**
       * a class extends another class and an interface both
       * of whom have the same method
       */
      public void bright()
      {
            System.out.println("this is bright");
      }
      public void tight()
      {
            System.out.println("this is tight");
      }
      /*void abstmethod()
      {
            System.out.println("abstract method");
      }*/

      public static void main(String[] args) {
            InterfaceClassShareMethod icm=new InterfaceClassShareMethod();
                  icm.fight();
                  icm.bright();
                  icm.tight();
            }
}
Output:
InterfaceClassShareMethod1 class
this is bright
this is tight
```

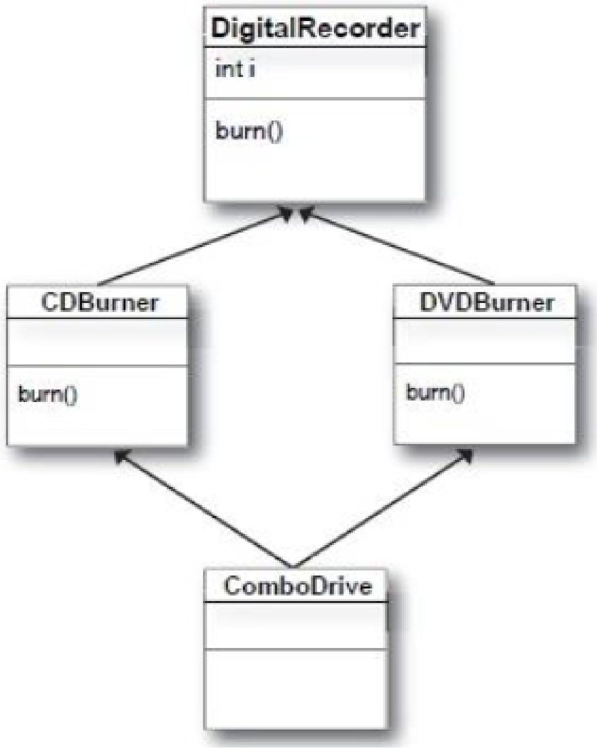
Note : Even comments for ‘abstract’ is removed it will work fine .

➔ If in the previous case, if common method will be implemented in the subclass then it will **overloads** the method of superclass.

3. <u>Multiple inheritance in java</u> :

➔ Multiple inheritance in java will be done through interfaces. i.e. in java there no multiple inheritance through classes but through interfaces.

➔ Why the multiple inheritance is done through interfaces in java bcos of DDD (deadly diamond of death) problem in java. Example of DDD is shown below.



In the above diagram we can see that when an object of class ComboDrive is created and burn() is called on it then confusion will occur and it is called as DDD.

➔ But how if a class A want three different methods each from one of three class ?? How one will going to implement that since multiple inheritance is not available. Answer for this question is interface is not here for 'code reuse' , its only made for a particular type or context and upon that only everything works in java. And in almost all of the circumstances there are few chances that such scenario take place.

➔ What happens when a class extends a abstract class and implements an interface both having the exactly same abstract method.

```
package ch9Interfaces;

abstract class AbstrInterfaceClass1
{
	abstract void method1();
}

interface AbstrInterfaceClass2
{
	void method1();
}

class AbstrInterfaceClass3 extends AbstrInterfaceClass1 implements
AbstrInterfaceClass2

{
	public void method1()
	{
		System.out.println("inside method");
	}
}

public class AbstrInterfaceClass
{

	public static void main(String[] args)
	{
		AbstrInterfaceClass1 aic1 = new AbstrInterfaceClass3();
		AbstrInterfaceClass2 aic2 = new AbstrInterfaceClass3();

		aic1.method1();
		aic2.method1();
	}
}
Output:
inside method
inside method
```

➔ One of the advantages for interfaces(also for abstract classes) that although an interface object cannot be created but surely a reference can be created which can be used polymorphically. A simple example shown below.

```
package ch9Interfaces;

interface intfcone
	{
	void infmet1();
	}

class InterfacePolymorphicNature1 implements intfcone
	{
		public void infmet1()
		{
		System.out.println("inside class 1");
		}
	}
class InterfacePolymorphicNature2 implements intfcone
```

```
	{
		public void infmet1()
		{
			System.out.println("inside class 2");
		}
	}

class InterfacePolymorphicNature3 implements intfcone
	{
		public void infmet1()
		{
			System.out.println("inside class 3");
		}
	}

public class InterfacePolymorphicNature
	{
		/**
		 * polymorphic nature of interfaces
		 */
		public static void main(String[] args) {
			// TODO Auto-generated method stub
			intfcone inf1 = new InterfacePolymorphicNature1();
			inf1.infmet1();
			intfcone inf2 = new InterfacePolymorphicNature2();
			inf2.infmet1();
			intfcone inf3 = new InterfacePolymorphicNature3();
			inf3.infmet1();
		}
}
Output:
inside class 1
inside class 2
inside class 3
```

➔ Same above program can be rewritten using Strategy pattern as shown below.

```
package ch9Interfaces;

interface intfcone1
{
	void infmet1();
}

class InterPolStrategyPattern1 implements intfcone1
{
	public void infmet1()
	{
		System.out.println("inside class 1");
	}
}
class InterPolStrategyPattern2 implements intfcone1
{
	public void infmet1()
	{
		System.out.println("inside class 2");
	}
}

class InterPolStrategyPattern3 implements intfcone1
{
	public void infmet1()
	{
		System.out.println("inside class 3");
	}
}

class StrategyClass
{
	void strategyMethod(intfcone1 inf)
	{
		inf.infmet1();
	}
}

public class InterPolStrategyPattern
{
	/**
	 * polymorphic nature of interfaces using strategy pattern
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		InterPolStrategyPattern1 inf1 = new InterPolStrategyPattern1();
		InterPolStrategyPattern2 inf2 = new InterPolStrategyPattern2();
		InterPolStrategyPattern3 inf3 = new InterPolStrategyPattern3();
		StrategyClass strg = new StrategyClass();
		strg.strategyMethod(inf1);
		strg.strategyMethod(inf2);
		strg.strategyMethod(inf3);

	}
}
```
Output:
```
inside class 1
inside class 2
inside class 3
```