Week – 10

ABSTRACTION

ABSTRACT METHODS AND CLASSES

Abstraction is a process of hiding the implementation details from the user, only the functionality will be visible to the user.

Here are two ways to achieve abstraction in java

1. Abstract class (partial abstraction)
2. Interface (full abstraction)

Abstract class in Java

A class that is declared as abstract is known as abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body example like, *public void get();*
- But, if a class contain at least one abstract method, then the class must be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To utilize an abstract class, you have to inherit it from another class, provide implementations for the abstract methods in it.
- It needs to be extended and its method implemented.
- If you inherit an abstract class, you have to provide implementations for all the abstract methods in it.

Syntax:

```
abstract class class_name
{
}
```

Abstract method

Method that are declared without any body within an abstract class are called abstract method. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

- **The abstract** keyword is used to declare the method as abstract.
- You have to mention the **abstract** keyword before the method name in the method declaration.
- An abstract method has a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semicolon (;) at the end.

Syntax:

```
abstract return_type function_name ();    // No definition
```

```
abstract class A
{
          abstract void callme();
}
class B extends A
{
          void callme()
          {
          System.out.println("this is call me inside child.");
           }
          public static void main(String[] args)
           {
            B b = new B();
           b.callme();  // this is call me inside child
          }
}
```

INTERFACE IN JAVA

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

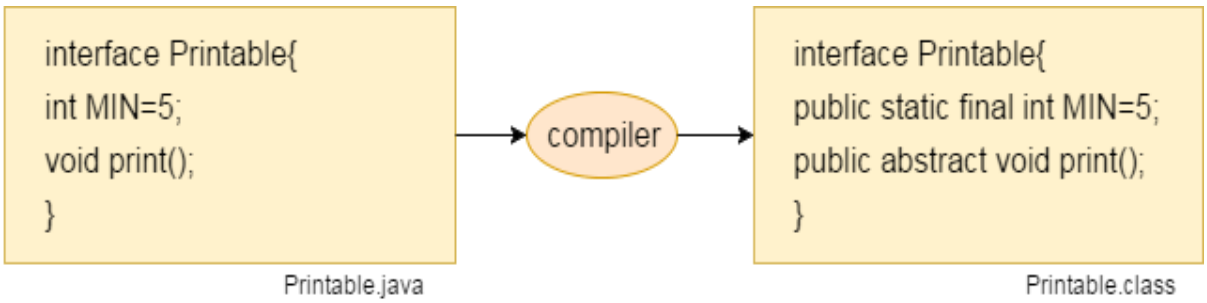
Syntax:

```
interface <interface_name>
{

   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

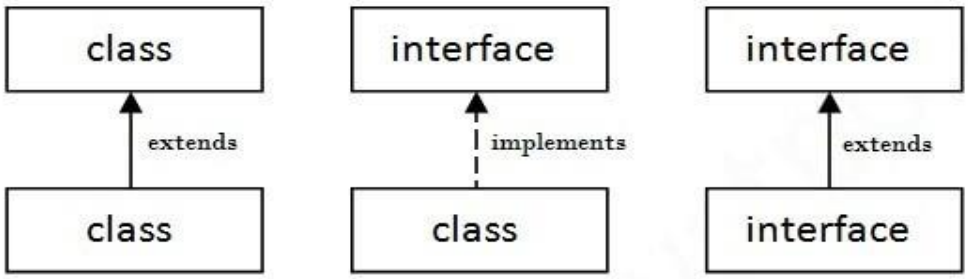
The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



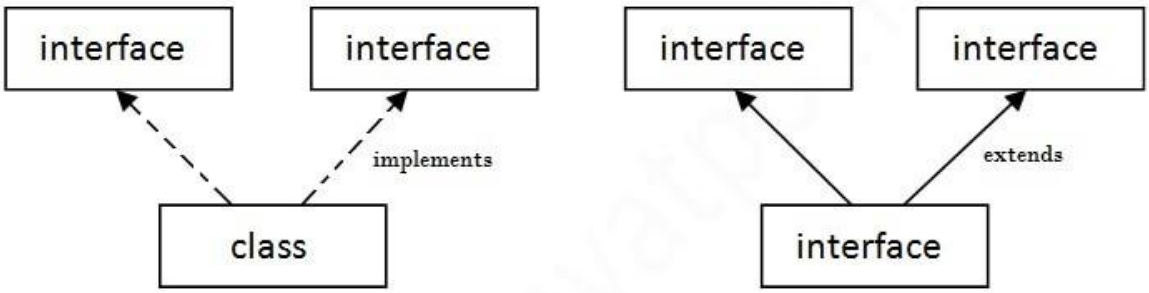
The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

EXTENDING AND IMPLEMENTING INTERFACES

Like classes, interfaces can also be extended. i.e interfaces can be sub interfaced from other interfaces.

In the below syntax name1 and name2 are interfaces so we use extends keyword.

interface name2 **extends** name1

{

Body of name2

}

Interfaces are used as "superclasses" whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interfaces.

In the below syntax class cname implements the interface name1.

class cname **implements** name1

{

Body implementing interface name1

}

Or

In the below syntax class cname implements many interfaces (name1,name2 and so on...)

class cname **implements** name1, name2,name3 ….

{

Body of classname

}

Example to demonstrate

- **use of implementing interfaces**
- **use of extending interfaces.**

```
interface Area
{
  final static float pi=3.14f;
  double compute(double x, double y);
}

interface Display extends Area
{
   void display_result(double result);
}

class Rectangle implements Display
{
  public double compute(double x, double y)
  {
   return(pi*x*y);
  }

  public void display_result(double result)
  {
   System.out.println("The area is:"+result);
  }
}

class InterfaceDemo
{
  public static void main(String args[])
  {
  Rectangle r1=new Rectangle();
  double res=r1.compute(10.2,20.4);
  r1.display_result(res);
  }
}
```

```
C:\javaprg>javac prog13demo.java

C:\javaprg>java prog13demo
The area is:653.3712218284606
```