04. Inheritance, Interfaces, Packages and Enum.

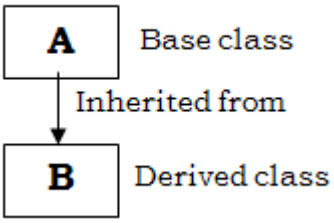
Introduction:

- We know that, inheritance is one of the most important object oriented concept and Java language supports it.
- The main concept behind inheritance is reusability of code (Software) is possible by adding some extra feature into existing software without modifying it.
- In java language, '*Object'* is super class of all classes even your own defined class also.

Definition:

- "The mechanism of creating new class from existing class is called as Inheritance".
- The existing or old class is called as 'Base class' or 'Parent class' or 'Super class' and new class is called as 'Derived class' or 'Child class' or 'Sub class'.
- While deriving the new class from exiting one then, the derived class will have some or all the features of existing class and also it can add its own features i.e sub class will acquires features of base class without modifying it.
- When a new class is derived from exiting class then all public, protected and default data of base class can easily accessed into its derived class where as private data of base class cannot inherited i.e. this private data cannot accessed in derived class.

Consider following example:



In above figure class B is inherited from class A. Therefore class A is called as *Super class* or or *parent class or base class* of class B where as class B is called as *Sub class or child class or derived class*.

Need or Features of Inheritance:

1) It is always nice to use existing software instead of creating new one by adding some extra features without modifying it. This can be done by only inheritance.
2) Due to inheritance software cost is reduces.
3) Due to inheritance software developing time is also reduces.
4) Inheritance allows reusability of code.
5) Due to inheritance we can add some enhancement to the base class without modifying it this is due to creating new class from exiting one.

Syntax to derive new class from existing class:

```
class       derived_class_name    extends    base_class_name
{
        //Declaration of Variables and definition of methods.
}
```

Here;

c*lass*	is keyword to define class.
extends	is keyword used to create or derive or extends new class.
derived_class_name	is name of newly created class which is an identifier.
base_class_name	is name of existing class which is also an identifier.

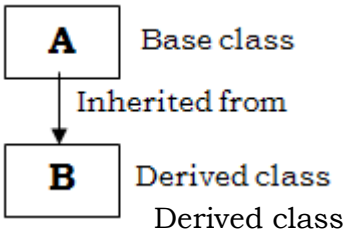
Types (forms) of Inheritance:

Depending upon number of base classes and number of derived classes and their arrangement, inheritance has following types.

1) Single Inheritance:

- In case of Single inheritance there is only *one base class from which we derive only one new class*.

e.g.



From above figure we can write single inheritance as:

```
class       A
{
   //Declaration of Variables and definition of methods.
}

class       B   extends       A
{
      //Declaration of Variables and definition of methods.
}
```

Following program shows implementation of single inheritance:

```
import      java.util.Scanner;
class       A
{
          int    x,y;
          Scanner   sc=new   Scanner(System.in);
          void get( )
          {
          System.out.println("Enter two numbers ");
                    x=sc.nextInt( );
                    y=sc.nextInt( );
          }
}
```

```
class      add    extends      A
{
          int z;
          void       doadd( )
          {
                    super.get( );
                    z=super.x+super.y;
                    System.out.println("Addition="+z);
          }
}
public  class   single
{
          public    static   void   main(String [ ] args)
          {
                    add   p=new  add( );
                    p.doadd( );
          }
}
```

'super' keyword:

- If we create an object of super class then we can access only super class members, i.e. we are not able to access the sub class members using object of super class.
- But, *if we create an object of sub class, then all the members of super class as well as sub class are easily accessible by object of sub class*. And that's why we always create an object of sub class instead of super class.
- Some time, both super class and sub class may have members (variable or methods) with same name in such situation, *only members of sub class is accessible with its object*. Here, super class members are not considered. And hence, in such condition if we want to access super class members into its sub class then 'super' keyword is used along with member of base class.

Use of 'super' keyword:

- 'super' keyword is used to access members of base class into its derived class.
- We can access, instance variables of base class into its derived class using syntax:

```
super.variable;
```

- We can access, methods of base class into its derived class using syntax:

```
super.method(arg1,arg2, ...);
```

- Note that, *we need not call the default constructor of super class because it is defaultly available to its sub class*. And hence, We can made call to parameterized constructor of base class into its derived class using syntax:

```
super( arg1,arg2,.....);
```

Consider following example that demonstrate use of 'super' keyword:

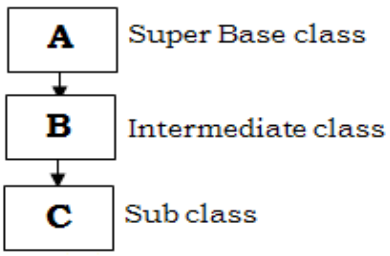
```
class    A
{
         int    x=55;

}
class    B    extends    A
{
         int      x=70;
         void    get( )
         {
          System.out.println("Super Class variable="+super.x);   //access 'x' of base class using 'super' keyword
          System.out.println("Sub Class variable="+x);
         }
         public   static   void    main(String [ ]   args)
         {
                  B   p=new  B( );
                  p.get( );
         }
}
OUTPUT:
                  Super Class variable= 55
                  Sub Class variable= 70
```

2) Multilevel Inheritance:

- In case of multilevel inheritance we can derive *new class from another derived classes, further from derived classes we again derive new class and so on*.
- Also, there are some levels of inheritances therefore it is called as "Multilevel inheritance"

E.g.

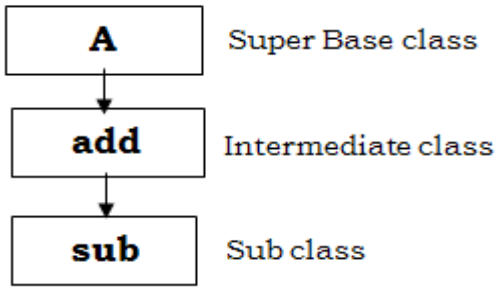


From above figure we can write multilevel inheritance as fallow:

```
class       A
{
   //Declaration of Variables and definition of methods.
}
class       B    extends     A
{
      //Declaration of Variables and definition of methods.
}
class       C      extends    B
{
   //Declaration of Variables and definition of methods.
}
```

Implementation of Multilevel Inheritance is as fallow:

Figure:



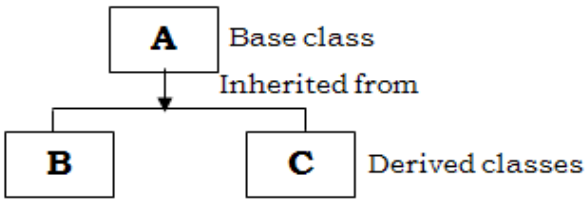

```
import  java.util.Scanner;
class A
{
        int   a,b;
        void get()
        {
          Scanner sc=new Scanner(System.in);
          System.out.println("Enter two no=");
          a=sc.nextInt();
          b=sc.nextInt();
        }
}
class   add  extends   A
{
        int   c;
        void doadd()
        {
           super.get();
           c=super.a+super.b;
           System.out.println("Addition="+c);
        }
}
```

```
class  sub  extends  add
{
        int  c;
        void  dosub()
        {
          super.doadd();
          super.get();
          c=super.a-super.b;
          System.out.println("Subraction="+c);
        }
}
public  class  Multilevel
{
   public static void main(String []args)
   {
        sub  t=new sub();
        t.dosub();
   }
}
```

3) Hierarchical Inheritance:

- In case of hierarchical inheritance there is *only one base class from that class we can derive multiple new classes*.
- The base class provides all its features to all derived classes which are common to all sub classes.
- The hierarchical inheritance comes into picture when certain feature of one level is shared by many other derived classes.

e.g.

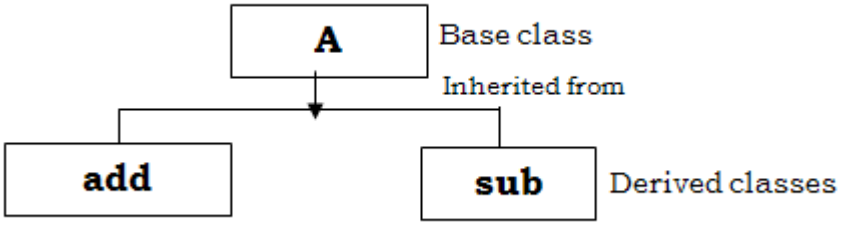


From above figure we can write hierarchical inheritance as fallow:

```
class     A
{
   //Declaration of Variables and definition of methods.
}
class     B     extends     A
{
      //Declaration of Variables and definition of methods.
}
class     C      extends     A
{
    //Declaration of Variables and definition of methods.
}
```

Implementation of Hierarchical Inheritance is as fallow:

Figure:




```
import    java.util.Scanner;
class  A
{
        int     a,b;
        void   get()
        {
          Scanner sc=new Scanner(System.in);
          System.out.println("Enter two No=");
          a=sc.nextInt();
          b=sc.nextInt();
        }
}
class add extends A
{
        int    c;
        void    doadd()
        {
          super.get();
          c=super.a+super.b;
          System.out.println("Addition="+c);
        }
}
```

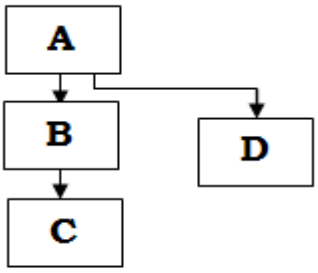
```
class  sub  extends  A
{
        int   c;
        void   dosub()
        {
          super.get();
          c=super.a-super.b;
          System.out.println("Subtraction="+c);
        }
}

public class Hierarchical
{
        public static void main(String []args)
        {
          add  p=new add();
          p.doadd();
          sub q=new sub();
          q.dosub();
        }
}
```

4) Hybrid Inheritance:

- Hybrid inheritance is the *combination of two or more forms of inheritances*.
- To implement hybrid inheritance, we have to combine two or more forms of inheritances and such a combination of different forms of inheritances is called “Hybrid inheritance”.

E.g.

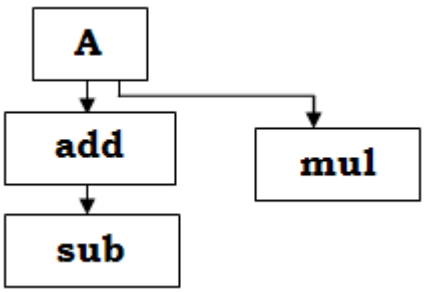


The above inheritance is a combination of multilevel and hierarchical inheritances therefore it is “Hybrid Inheritance”.

From above figure we can write hybrid inheritance as fallow:

```
class     A
{
   //Declaration of Variables and definition of methods.
}
class     B     extends     A
{
      //Declaration of Variables and definition of methods.
}
class     C      extends     B
{
    //Declaration of Variables and definition of methods.
}
class     D     extends     A
{
      //Declaration of Variables and definition of methods.
}
```

Implementation of Hybrid Inheritance is as fallow:

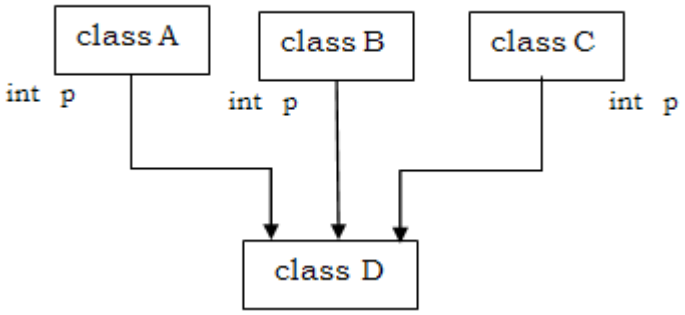



```
import java.util.Scanner;
class   A
{
        int a,b;
        void get()
        {
          Scanner sc=new Scanner(System.in);
          System.out.println("Enter two No=");
          a=sc.nextInt();
          b=sc.nextInt();
        }
}
class  add  extends  A
{
        int  c;
        void  doadd()
        {
          super.get();
          c=super.a+super.b;
          System.out.println("Addition="+c);
        }
}
class  sub  extends  add
{
        int   c;
        void  dosub()
        {
          super.doadd();
          super.get();
          c=super.a-super.b;
          System.out.println("Subtraction="+c);
        }
}
```

```
class  multi  extends  A
{
        int c;
        void  domulti()
        {
          super.get();
          c=super.a*super.b;
          System.out.println("Multiplication="+c);
        }
}
public  class  Hybrid
{
        public static void main(String []args)
        {
          sub  p=new sub();
          p.dosub();
          multi   q=new multi();
          q.domulti();
        }
}
```

5) Multiple Inheritance:

- Multiple inheritance means deriving or creating *only one sub class from multiple super classes*.
- But, Java does not support for multiple inheritance. And that multiple inheritance leads confusion for programmer, which is shown in following figure:



- In above fig, Super classes i.e. *Class A, Class B* and *Class C* has same member 'int p' and from these classes *Class D* is derived or extended. In this case, *Class D* has confusion regarding with copy of 'int p'. That is *Class D* does not know whose copy of 'int p' is accessed.
- Also, *class D extends A, B, C* such syntax is invalid in Java.
- If Java does not support for 'Multiple inheritance' then it is not considered as pure OOP language. In fact Java is pure OOP language.
- But, In Java, *multiple inheritance can be implemented using 'interface'* concept. So let's see the 'Interface' concept in details.

Interface

Introduction:

We know that, Java language does not supports for multiple inheritance but we can implement multiple inheritance using 'interface'.

Interface:

- An 'interface' is collection of only abstract methods.
- All the methods of interface by default *public & abstract*.
- Also, 'interface' has variables but by default all interface variables are *static, final and public*.
- An interface contains only abstract methods which are all incomplete therefore it is not possible to create object of interface.
- But, we can create separate classes from interface where we can implement all the methods of interface. These classes are called as 'implementation' classes.
- Since, 'implementation classes' contains method definition therefore we can create object of implementation classes.
- Every implementation class can have its own implementation of abstract methods of the interface.

Syntax to define interface:

```
interface    Interface_name
{
      datatype     var1=value;
                   ⋮
      return_type  Method1( );
                   ⋮
}
```

here,

'interface' is keyword used to define interface.
'Interface_name' is name of interface which is an identifier.

Syntax to define new class (implementation class) from interface:

```
class     implement_class_name     implements    Interface_name
{
      datatype     var1, varN;

      public  return_type     Method1( )
      {
            --------
            --------
      }

      public  return_type     MethodN( )
      {
            --------
            --------
      }
}
```

here,

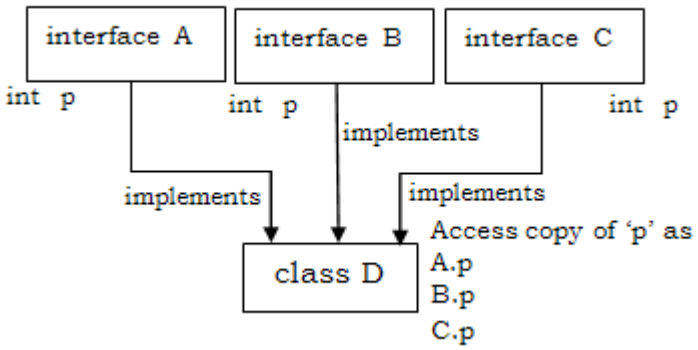
'implements' is keyword used to define or create new class from interface.
'Interface_name' is name of interface from which *implement_class_name* is created.

Characteristic or properties of Interface:

1) An 'interface' is a *specification of method prototype* only i.e. methods does not have any body
2) An interface will have zero or more abstract methods which are all public & abstract by default.
3) An interface can have variables which are public, static and final by default. This means all variables of interface are constant therefore we have to assign them value.
4) None of the methods in interface can be private, protected or static.
5) We cannot create an object of interface but we can create reference to interface.
6) All methods of interface should be defined in its implementation classes. If any method is not implemented then that implementation class should be declared as 'abstract'
7) An interface can extend another interface.
8) An interface cannot implement another interface.
9) A class can implement (not extends) multiple interfaces.
   For Ex: class MyClass implements interface1, interface2, interface3 . . . .

Multiple Inheritance using Interfaces:

- Multiple inheritance means deriving or creating *only one sub class from multiple super classes*.
- But, Java does not support for multiple inheritance. And that multiple inheritance leads confusion for programmer, which is shown in following figure:



- In above fig, *class D* is implementation class which is implemented from three interfaces viz. *interface A*, *interface B* and *interface C*. In this case, *Class D* can access individual copy of *'int p'* from all interfaces using *A.p, B.p and C.p*. Because, by default all variables of interface are static therefore they are accessed by interface name. Thus, confusion regarding copy of *'int p'* in *Class D* is solved.
- Also, *class D implements A, B, C* such syntax is valid in Java.
- Thus, In Java, *multiple inheritance is implemented using 'interface'* concept.

Following program shows multiple inheritance using interface.

```
interface  A
{
            int  p=10;
          void  calculate( );
}
interface  B
{
            int  p=20;
          void  calculate( );
}
interface  C
{
          int  p=30;
          void  calculate( );
}
```

```
class  D  implements  A,B,C
{
    public  void  calculate( )
    {
       int z;
       z=A.p+B.p+C.p;
       System.out.println("Addition="+z);
    }
}
class   multiple
{
     public  static  void  main(String s[ ])
     {
        D   obj=new   D( );
        obj.calculate();
      }
}
```

In above program:

class D is implementation class which is implemented from interfaces A,B and C.

The *calculate( )* method in implementation *'class D'* must be defined with **'public'** modifier because it overrides its abstract methods of interfaces A,B,C. And by default all interface methods are **public**.

If do not define it with public modifier then compiler gives compilation error.

Method Overriding:

- When both classes i.e. super class & sub class have same methods with same signatures i.e. both super class and sub class have same method prototype then sub class method overrides (replaces) the super class method such a concept is called as 'Method overriding'
- We know that, super class methods are also accessed by object of sub class. But, in method overriding *both classes has same methods with same signatures in such situation; JVM only executes methods of sub class i.e. here sub class method replaces (overrides) the super class method and hence super class method not executes*.
- That is we can say that, super class method is overridden by sub class method. And hence super class method not executed by sub class object.
- Consider following example which shows Method overriding concept:

```
class     first
{
          void    get(int     p)
          {
                  ----------
                  ----------
                  ----------
          }
}
class     second   extends    first
{
          void   get(int     x)
          {
                  ----------
                  ----------
                  ----------
          }
}
```

- In above example, 'first' is super class whereas 'second' is sub class of 'first' class.
- Also, both classes has *get( )* method with same signatures. And if we made call to *get( )* method *using object of sub class* then only sub class *get( )* method is invoked i.e. sub class method overrides the base class method & *get( )* method of 'first' class never executed. Hence, here method overriding is done.

Following program demonstrate the Method overriding concept using instance method:

```
class      one
{
       void    add( )
        {
          System.out.println("I am in Base class");
        }
}
class  two    extends    one
{
     void   add( )
      {
        System.out.println("I am in Derived Class");
      }
 }
```

```
class      all
{
     public    static   void   main( String  [ ]ar)
    {
          two      t=new    two( );
          t.add( );
     }
}

OUTPUT:
                I am in Derived Class
```

Dynamic Method Dispatch or Runtime Polymorphism in Java:

- We know that, Runtime polymorphism is achieved in Java by 'method overriding'.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- When an overridden method is called through a superclass reference, Java determines which version (superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as "upcasting". Java uses this fact to resolve calls to overridden methods at run time.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.
- Following program shows Dynamic method dispatch concept:

```
class A
{
   void   m1()
   {
      System.out.println("Inside A's m1 method");
   }
}

class B extends A
{
                        // overriding m1()
   void   m1()
   {
      System.out.println("Inside B's m1 method");
   }
}

class C extends A
{
                        // overriding m1()
   void   m1()
   {
      System.out.println("Inside C's m1 method");
   }
}
```

```
class     Dispatch
{
   public static void main(String    args[])
   {
      A   x = new A();          // object of type A
      B   y = new B();          // object of type B
      C   z = new C();          // object of type C
      A    ref;      // reference of type A
      ref = x;      // ref refers to an A object
      ref.m1();  // calling A's version of m1()
      ref = y;   // now ref refers to a B object
      ref.m1();  // calling B's version of m1()
      ref = z;   // now ref refers to a C object
     ref.m1();   // calling C's version of m1()
   }
}
```

OUTPUT:

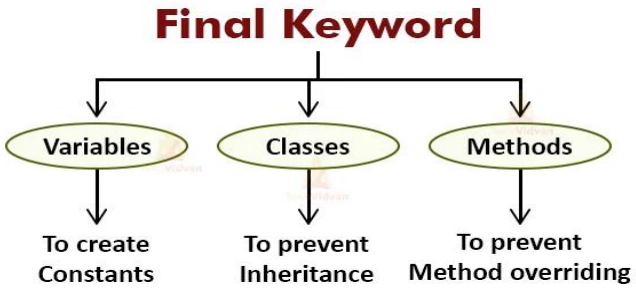
```
Inside A's m1 method
Inside B's m1 method
Inside C's m1 method
```

Difference between method overloading and method overriding:

Method Overloading	Method Overriding
1) Defining multiple method with same name with different signature is called 'Method overloading'	1) Defining multiple method with same name with same signature is called 'Method overriding'
2) Method overloading is done within same class.	2) Method overriding is done within different classes i.e. in super class & sub class.
3) In method overloading, return type of methods can be same or different.	3) In method overriding, return type of methods must be same.
4) JVM decides which method has to be called depending upon parameters of methods.	4) JVM decides which method has to be called depending upon object of sub class or super class.
5) Method overloading is code development. Same methods are developed to perform same task depending on parameters.	5) Method overriding is code replacement. The sub class method overrides (replaces) the super class method.

'final' Keyword:

- The final keyword is a non-access modifier used for classes, class attributes and class methods, which makes them non-changeable (impossible to inherit or override).
- We can use 'final' keyword *before variable declaration or before method definition or before class definition*. We explain all these as follows:



1) final Variable:

- If we use 'final' keyword before variable declaration then that variable becomes 'final' variable.
- And final variables are treated as constant in Java.
- That is, to declare constant in Java, 'final' keyword is used.
- *final variable* cannot be modified because they treated as constant.

Syntax:

```
final     datatype     name=value;
```

here,

'final' is keyword.
'datatype' is valid data type in java.
'name' is an identifier which is constant name .
'value' is any constant value assigned to final variable.

Ex: final double PI= 3.14;

Here, Incrementation or Decrementation or initialization other value to *'PI'* is invalid because it is 'final' and not modified.

2) final Method:

- If we use 'final' keyword before method definition then that method becomes 'final' method.
- And *final methods* cannot override by its derived or sub class. That is sub class method *cannot replace* code of its base class *final method*.
- *Note that: 'final' methods of base class* ***can be*** *accessible into its derived class.*

Syntax:

```
final     return_type     Method_name( )
{
        ----------
        ----------
        ----------
}
```

Note:

- *Final methods of base class cannot override into its derived class therefore method overriding using final methods are not possible. That is final methods are not overridden.*
- *But, 'final' methods of base class can be accessible into its derived class.*

Program:

Program:- 1)

```
class    A
{
          int   a;
 final   void   get()
  {
          a=10;
  }
}
class  B  extends   A
{
          int   b;
          void   get()
          {
                    b=50;
          }
}
public  class   FinalMethod
{
  public static void main(String []args)
  {
                    B     obj=new   B();
                    obj.get();
  }
}
```

Above program gives compile error as:
get() in B cannot override get() in A
Since, final method cannot override.

Program:- 2)

```
class    A
{
          int a;
          final  void  get()
          {
                    a=10;
          }
}
class B extends A
{

          void   show()
          {
          super.get();    //access final method in sub class
          System.out.println("Value="+super.a);
          }
}
public  class  FinalMethod
{
  public static void main(String []args)
  {
                    B     obj=new   B();
                    obj.show();
  }
}
```

Above program successfully executes. Since, here is no method overriding concept.

3) final Class:

- If we use 'final' keyword before class definition then that class becomes 'final' class.
- And *final class* prevents inheritance. That is, *we are not able to create or extends new class from 'final class'*
- If we don't want to allow sub class to access features of its base class then base class has to made as 'final'

Syntax:

```
final     class      class_name
{
          ----------
          ----------
          ----------
}
```

Ex.

```
final     class      A
{
          --------
          --------
}
class    B   extends    A          // Is invalid.....
```

Program:- 1)

```
final   class    A
{
          void get()
          {
                    System.out.println("Base class");
          }
}
class  B  extends  A
{
```

Program:- 2)

```
class    A
{
          void get()
          {
                    System.out.println("Base class");
          }
}
class  B  extends  A
{
```

void show()
{
super.get();
System.out.println("Derived class");
}
}
public class FinalClassDemo
{
public static void main(String []args)
{
B obj=new B();
obj.show();
}
}
	void show()
	{
	super.get();
	System.out.println("Derived class");
	}
	}
	public class FinalClass
	{
	public static void main(String []args)
	{
	B obj=new B();
	obj.show();
	}
	}

Above program gives compile error as: **cannot inherit from final A** Since, final class cannot inherited.	Above program successfully executes. Since, here no final class.

Abstract Method and Abstract class

Abstract Method:

- Abstract method is such method that does not have any definition i.e. it has no body.
- Abstract method contains only method header i.e. method prototype.
- Abstract method has no body therefore they are also called 'incomplete method'
- Abstract method should be declared with keyword '*abstract'*
- When abstract method is declared into super class then *it is compulsory to define it into its sub classes*.
- An abstract method is written when *same method has to perform different tasks depending upon object requirements*.

Ex:

```
abstract class ABC
{
    abstract void get( );
}
```

- In above example, get( ) method is declared as abstract using keyword 'abstract'. And that method does not have any definition.

Abstract class:

- Abstract class is such class that declared with keyword 'abstract' and that *contains zero or more abstract methods*.
- Abstract class may also contain instance variable & concrete methods.
- When abstract method is declared into super class then *it is compulsory to define it into its sub classes*.
- We *cannot create object of Abstract class* i.e. Abstract class cannot instantiated. Because abstract class contains incomplete abstract methods.
- But *we can create reference of Abstract class*. And that *reference* may refer to *object of its sub classes*.
- Abstract class may have constructors. Then, question is that who invoke constructor of abstract class? Since, object of abstract class cannot be created.
- Answer is that: constructors of abstract class is invoked by object of its sub classes.
- We cannot declare a class as *final* and *abstract* at a time. Because, *final* keyword prevents to create sub classes and *abstract* keyword allows super class to have sub classes.

Ex:

```
abstract class ABC
{   abstract void get( );
    void show( )
    {
        ----------
        ----------
    }
}
```

- In above example, class *ABC* is declared as abstract using keyword 'abstract'. And that contains get( ) abstract method and show( ) concrete method.

Following program demonstrate the use of abstract class and abstract methods.

```
abstract class MyBase
{
        abstract void calculate(double x);
}
class Square extends MyBase
{
        void calculate(double a)
        {
          System.out.println("Square="+(a*a));
        }
}
class squareRoot extends MyBase
{
   void calculate(double a)
    {
   System.out.println("SquareRoot="+Math.sqrt(a));
    }
}
```

```
class Cube extends MyBase
{
        void calculate(double a)
        {
          System.out.println("Cube="+(a*a*a));
        }
}
class abstractDemo
{
        public static void main(String as[ ])
        {
                Square t1=new Square ();
                squareRoot t2=new squareRoot ();
                Cube t3=new Cube ();
                t1.calculate(5);
                t2.calculate(36);
                t3.calculate(4);
        }
}
```

Difference between Abstract class and Interface:

Abstract Class	Interface
1) An abstract class contains some abstract methods and also some concrete methods.	1) An interface only contains abstract methods.
2) An abstract class contains instance variable	2) An interface cannot contain instance variables. It contains only constants.
3) By default abstract methods of abstract class is not public.	3) By default abstract methods of interface is public.
4) All abstract methods of abstract class should be implemented into its sub classes.	4) All abstract methods of interface should be implemented into its implementation classes.
5) Abstract class is declared using 'abstract' keyword.	5) Interface is declared using 'interface' keyword.
6) Methods in abstract class can be private, protected or static.	6) Methods in interface cannot be private, protected or static.
7) Abstract class may have 'constructors'	7) Interface does not have 'constructors'

Packages:

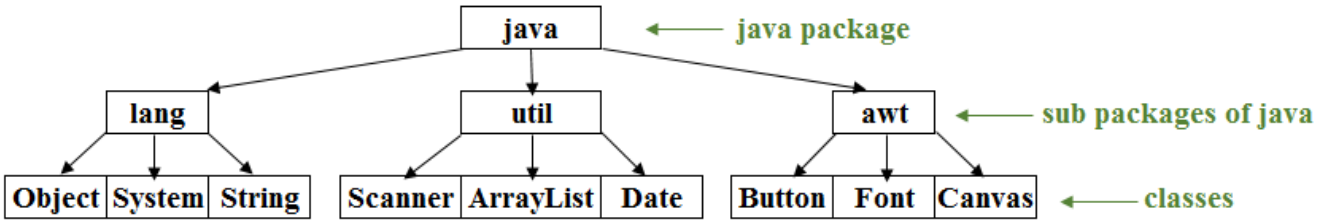
- A java package is a directory or folder which is group of similar types of classes, interfaces and sub-packages.
- We use packages to avoid name conflicts, and to write a better maintainable code.
- **Advantage of Java Package:**
  1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
  2) Java package provides access protection.
  3) Java package removes naming collision.
- Packages are divided into two categories:
  1) Built-in Packages (packages from the Java API)
  2) User-defined Packages (create your own packages)

Let's see these types in details-

1) Built-in Packages:

- Built-in packages are those packages which are predefined in Java Library (in JDK).
- This library contains methods which are useful for managing input, database programming, and much more.
- This library is divided into packages and classes. Means we can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

- Following diagram shows built-in packages:



- To use a class or a package from the library, we use the **import** keyword as follow:

```
import    package.name.Class;     // Import a single class
import    package.name.*;         // Import the whole package
```

Import a single Class:

- We can import single class as follow:

```
import    java.util.Scanner;      // Import a single class
```

In the example above, java.util is a package, while Scanner is a class of the java.util package.

Import a whole package:

We can import whole java.util package as follow:

```
import    java.util.*;      // Import whole util package
```

2) User defined Packages:

- The packages which are created or defined by user is called "User defined packages".
- To create user defined package, **package** keyword is used as follow:

```
package       packageName;
```

- We create user defined package **mypack** as follow:

```
package    mypack;
public   class        MyPackageClass
 {
   public static void main(String[] args)
  {
    System.out.println("This is my package!");
  }
}
```

Compiling user defined package:

Syntax:

```
D:\2101>javac  -d   directoryWithPath    JavaSourceProgram.java
```

The **-d** keyword specifies the destination for where to save package and the class file.
We can use any directory name, like d:\2101 etc.
or, if we want to keep the package within the same directory then use the dot sign "**.**" at directory.

Example: Above MyPackageClass.java package source file can be compiled as-

```
D:\2101>javac  -d    .   MyPackageClass.java
```

- This forces the compiler to create the "mypack" package within same directory.
- After successful compilation of given program, a new folder was created, called "mypack" that contains MyPackageClass.

Note: The package name should be written in lower case to avoid conflict with class names.

Running user defined package:

Syntax:

```
D:\2101>java   packageName.ByteCodeFileName
```

Example: Above package can be run as follow:

```
D:\2101>java   mypack.MyPackageClass
```

Creating and using (importing) User defined package:

- **Step 1:** Create java source program with following code:

```
package newpackage;
public class all
{
    public double add(double a, double b)
    {
        return(a+b);
    }
    public double sub(double a, double b)
    {
        return(a-b);
    }
    public double multi(double a, double b)
    {
        return(a*b);
    }

    public double div(double a, double b)
    {
        return(a/b);
    }
}
```

- **Step 2:** Save above program with **all.java** in your local directory.
- **Step 3:** Compile program as: d:\2201>javac -d . Multiplication.java
  After compiling above program it would create **newpackage** package in your local directory with **Multiplication.class** byte code.
- **Step 4:** Now to use this package in our program. Create new java source program and import **newpackage** package as-

```
import newpackage.all;
class UseDemo
{
    public static void main(String []arg)
    {
        all m=new all();
        double p=m.add(15,4);
        double q=m.sub(55,7);
        double r=m.multi(5,4);
        double s=m.div(65,7);
        System.out.println("Add="+p);
        System.out.println("Sub="+q);
        System.out.println("Multi="+r);
        System.out.println("Division="+s);
    }
}
```

- **Step 5:** Now compile above program as- d:\2201>javac UseDemo.java
- **Step 6:** Run above program as- d:\2201>java UseDemo

Wrapper classes:

- The wrapper class in Java provides the mechanism to convert primitive data types into object and object into primitive data types.
- Wrapper class uses autoboxing and unboxing feature to convert primitives into objects and objects into primitives automatically.
- The automatic conversion of primitive data types into an object is known as autoboxing whereas conversion of object into primitive type is called unboxing.

Use of Wrapper classes in Java:

- Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. where we need to use the wrapper classes.

- Change the value in Method: Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- Serialization: We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- Synchronization: Java synchronization works with objects in Multithreading.
- java.util package: The java.util package provides the utility classes to deal with objects.
- Collection Framework: Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

- Following eight classes of the java.lang package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing:

- The automatic conversion of primitive data type into its corresponding wrapper class is known as "Autoboxing". For example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short referred as Autoboxing.
- Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive data types into objects
public class AutoBox
{
public static void main(String args[])
{
int          a=20;
boolean    b=false;
Integer      j=a;       //autoboxing, compiler will write Integer.valueOf(a) internally
Boolean    k=b;   //autoboxing
System.out.println("j="+j);
System.out.println("k="+k);
}
}
```

Unboxing:

- The automatic conversion of wrapper type into its corresponding primitive type is known as "unboxing".
- It is the reverse process of Autoboxing. Wrapper class Example: Wrapper to Primitive

```
//Java program that converts object to primitive data types
public  class  UnBox
{
   public static void main(String    args[])
{
   Integer   a=new     Integer(3);
   int     j=a;           //unboxing, compiler will write a.intValue() internally
   Boolean   b=new  Boolean(false);
   boolean    c=b;
   System.out.println("int="+j);
   System.out.println("boolean="+c);
   }
}
```

Enumerations:

- The Enumerations or Enum in Java is a data type which contains a fixed set of constants.
- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.
- Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change).
- The Java enum constants are static and final implicitly. It is available since JDK 1.5.
- Enums are used to create our own data type like classes. The enum data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more powerful. Here, we can define an enum either inside the class or outside the class.

Points to remember for Java Enum:

- Enum improves type safety.
- Enum can be easily used in switch.
- Enum can be traversed.
- Enum can have fields, constructors and methods.
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Example of Java Enum:

```
class      EnumDemo
{
   enum  direction {NORTH,SOUTH,EAST,WEST };
   public  static  void   main(String[] args)
  {
     for (direction   s : direction.values())
     System.out.println(s);
   }
}
```

Note: The enum type has a values() method, which returns an array of all enum constants. This method is useful when you want to loop through the constants of an enum

Theory Assignment No: 04

1) What is Inheritance? Write need or features of inheritance.
2) Explain Single inheritance with example.
3) Explain Multilevel inheritance with example.
4) Explain Hierarchical inheritance with example.
5) Explain Hybrid inheritance with example.
6) How multiple inheritance creates confusion or problem in Java?
7) What is Interface? List out characteristics or properties of interface.
8) How multiple inheritance is implemented using interface in Java?
9) Explain Method overriding with example.
10) What is Dynamic method dispatch? Explain with example.
11) Write difference between method overloading and method overriding.
12) Write difference between Abstract class and interface.
13) What is wrapper class? Write its use.
14) What is Autoboxing and Unboxing? Explain with example.
15) Write short note on:
    a) 'final' keyword
    b) 'super' keyword
    c) Abstract method
    d) Abstract class
    e) Packages
    f) Enum

Practical Assignment No: 06

1) Write a program to implement single inheritance.
2) Write a program to implement multilevel inheritance.
3) Write a program to implement hierarchical inheritance.
4) Write a program to implement hybrid inheritance.
5) Write a program to implement multiple inheritance using interface.
6) Write a program that demonstrate use of method overriding.
7) Write a program that demonstrate dynamic method dispatch.
8) Write a program that demonstrate use ‘final’ keyword.
9) Write a program that demonstrate Autoboxing concept.
10) Write a program that demonstrate Unboxing concept.
11) Write a program that demonstrate use of **enum** data type.
12) Write a program that creates user defined package and use it in new java source program.
13) Write a program demonstrate the use of abstract class and abstract methods.