Unit-4

4.1 Basics of inheritance

- Types of inheritance:
  - single
  - Multiple
  - Multilevel
  - Hierarchical
  - Hybrid inheritance
- concept of method overriding
- extending class, super class, subclass
- dynamic method dispatch
- Object class

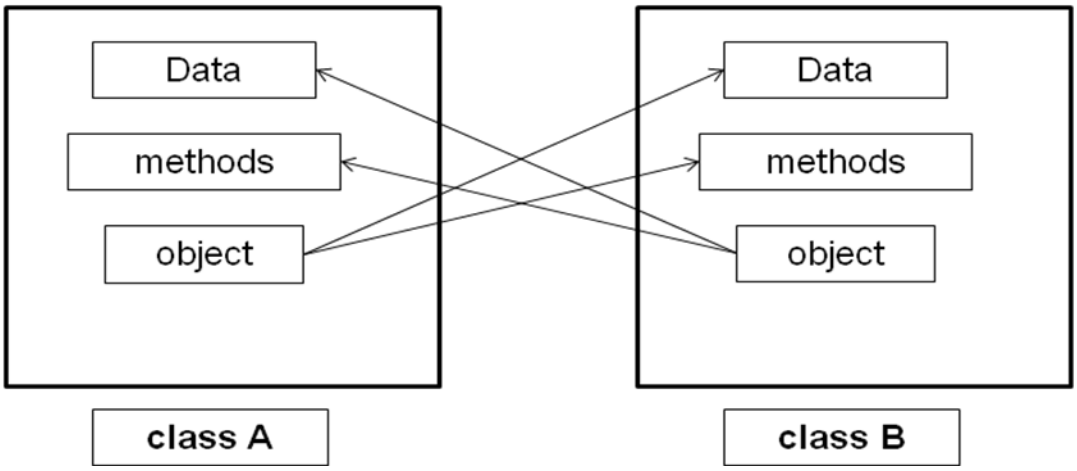
4.2 creating package, importing package, access rules for packages, class hiding rules in package

4.3 defining interface, inheritance on interface, implementing interface, multiple inheritance using interface

4.4 abstract class and final class

➢ Inheritance

- It is process by which objects of one class can use properties of objects of another class.
- The mechanism of deriving a new class from old class is called inheritance.



- Inheritance provides reusability of code.
- Java class can be reused in several ways by creating new class.
- Reusing the properties of existing class called inheritance.

Old class	New class
▫ Base class ▫ Super class ▫ Parent class	▫ Derived class ▫ Sub class ▫ Child class

- Inheritance allows subclasses to inherit all variable and methods of their parent class.
- Object have no super class.

Types of inheritance :

1) **Single inheritance** (only one super class)
2) **Hierarchical inheritance**(1 super class, many subclasses)
3) **Multilevel inheritance** (derived from a derived class)
4) **Multiple inheritance**(several super class)
5) **Hybrid Inheritance**

[Java does not directly support multiple inheritance and hierarchical inheritance which is actually supported in c ++, but it is implemented using interface.]

Syntax for defining a subclass:

```
class subclassname extends superclassname
{
        Variable section;
        method section;
}
```

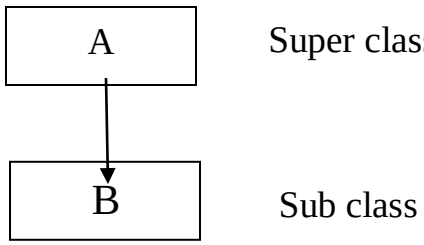
extends :it is keyword.it signifies that properties of ***superclassname are extended to the subclassname***

- So, subclass will now contain its own variables and methods as well as those of super class.
- It occurs when we want to add some more properties to an existing class without actually modifying it.
- Sub class can use the members of super class but super class can not access subclass's members.

1. Single inheritance

- Only one super class
- Every class has one and only one direct super class.
- If class inherits only a single class, is called single inheritance.

```
class A
{
   ---------
   ---------
}
class B extends A
{
   ---------
   ---------
}
```



Single inheritance

Example:

```
class A                          // super class
{
        int i;
}
```

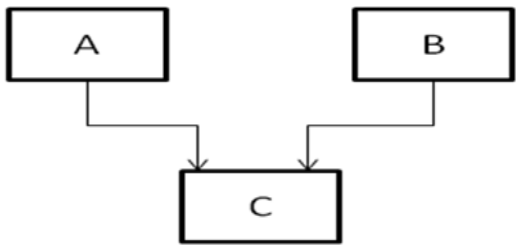
```
class B extends A                    //sub class
{
        void display()
        {
        System.out.println("i="+i);
        }
}
class Single
{
public static void main(String args[])
{
        B b1=new B();
        b1.i=7;
        b1.display();
}
}
```

Output:

```
i=7
```

2. Multiple inheritance

- One class extends properties of Several super classes.
- Java does not support this directly.



multiple inheritance

Here class C uses properties of class A and class B at same level.

Ex:

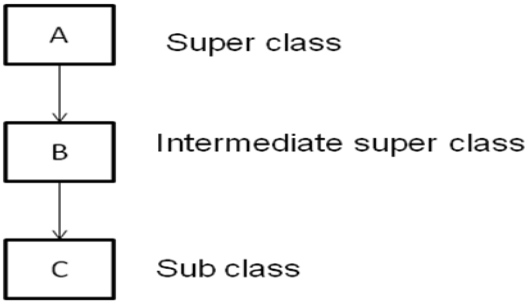
```
class A
        {
                ---------
                ---------
```

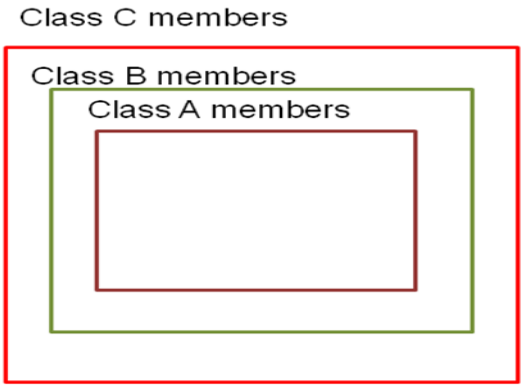
```
}
class   B

{
        ---------
        ---------
}
class C extends A ,B     //invalid not supported in java
{
        ---------
        ---------
}
```

3. Multilevel inheritance

- New class is derived from derived class.
- Java uses mainly in building its class library.





Multilevel inheritance

Ex:
```
class   A
{
        ---------
        ---------
}
class B extends A
{
        ---------
        ---------
}
class C extends B
{
        ---------
```

```
        ---------
}
```

Here class C can use properties of class B and class B uses the properties of class A. So class C can use properties of A and B.

Example of Multilevel inheritance

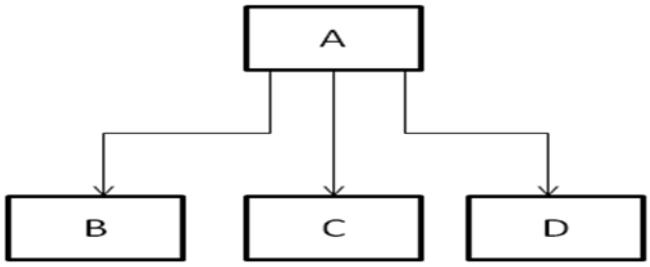
```
class A                         //super class
{
        int i;
}
class B extends A  //intermediate super class which extends A
{
        int j;
        void display ()
        {
        System.out.println ("i in class B="+i);
        }
}
class C extends B  //sub class C which extends B
{
        void sum ()
        {
        System.out.println ("sum of i and j in C ="+ (i + j));
        }
}
class Multilevel
{
public static void main (String args[])
{
        B b1=new B ( );
        b1.j=5;
        System.out.println ("j in B="+b1.j);
        C c1=new C (  );
        c1.i=3;
        c1.j=4;
        c1.sum();
}
}
```

Output:

J in B=5
sum of i and j in C=7

4. Hierarchical inheritance

- one super class, many subclasses.
- Many classes extends one sub class.
- Many programming problems can be featured on one level shared by other below level.



Multilevel inheritance

Ex:

```
class   A
{
        ---------
        ---------
}
class B extends A
{
        ---------
        ---------
}
class C extends A
{
        ---------
        ---------
}
Class D extends A
{
        ---------
        ---------
}
```

Here, class A extended by class B, C, and D.

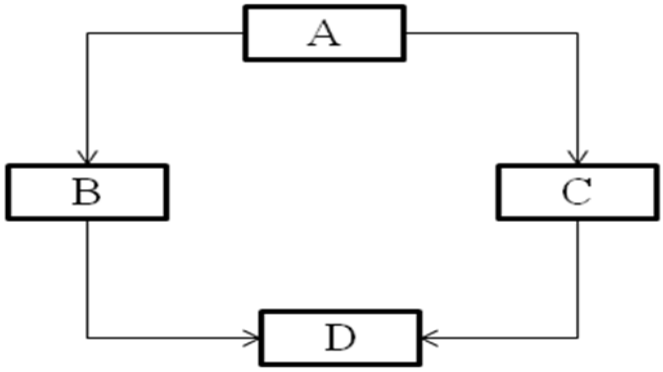
Example of Hierarchical inheritance

```
class A                          //super class
{
        int i;
}
class B extends A          // sub class of A
{
        int j;
        void display()
        {
        System.out.println("sum of i and j in B="+ (i +j));
        }
}
class C extends A          //subclass of B
{
        int k;
        void sum()
        {
        System.out.println("sum of i and k in C ="+ (i+k));
        }
}
class Hierarchical
{
        public static void main(String args[])
        {
        B b1=new B();
        b1.i=4;
        b1.j=5;
        b1.display();

        C c1=new C();
        c1.i=3;
        c1.k=5;
        c1.sum();
        }
}
```

5. Hybrid inheritance

- We can inherit properties of several classes at same level. If you are using only classes then this is not allowed in java, but using interfaces it is possible to have implement hybrid inheritance in java.



Hybrid inheritance

- In above diagram, it is a combine form of single and multiple inheritances. Since java doesn't support multiple inheritances, the hybrid inheritance is also not possible.
- Case 1: Using classes: If in above figure B and C are classes then this inheritance is not allowed as a single class cannot extend more than one class (Class D is extending both B and C).
- Case 2: Using Interfaces: If B and C are interfaces then the above hybrid inheritance is allowed as a single class can implement any number of interfaces in java.

- **Hierarchical inheritance is different than hybrid inheritance.** Hierarchical inheritance is possible to have in java even using the classes alone itself as in this type of inheritance two or more classes have the same parent class or in other words a single parent class has two or more child classes, which is quite possible to have in java.

Below program of hybrid inheritance will not execute.

```
class A
{
    public void methodA()
    {
        System.out.println("Class A methodA");
    }
}
class B extends A
{
    public void methodA()
    {
        System.out.println("Child class B is overriding inherited method A");
    }
```

```
    public void methodB()
    {
        System.out.println("Class B methodB");
     }
}
public class C extends A
{
    public void methodA()
    {
        System.out.println("Child class C is overriding the methodA");
     }
    public void methodC()
    {
        System.out.println("Class C methodC");
     }
}
class D extends B,C                          // it gives errorwe cans
   {
       public void methodD()
       {
           System.out.println("Class D methodD");
        }
   }
   }
   class multiple
   {
        public static void main(String args[])
        {
             D obj1= new D();
             obj1.methodD();
             obj1.methodA();
        }
   }
```

In the above program class B and C both are extending class A and they both have overridden the methodA(), which they can do as they have extended the class A. But since both have different version of methodA(),**compiler is confused** which one to call when there has been a call made to methodA() in child class D (child of both B and C, it's object is allowed to call their methods), this is a ambiguous situation and to avoid it, such kind of scenarios are not allowed in java. In C++ it's allowed.

We can implement hybrid inheritance using interfaces.

```
interface A
{
    public void methodA();
}
interface B extends A
{
    public void methodB();
}
interface C extends A
{
    public void methodC();
}

//hybrid inheritance, class D uses both B and C interface
class D implements B,C
{
   public void methodA()
   {
       System.out.println("MethodA");
   }
   public void methodB()
   {
       System.out.println("MethodB");
   }
   public void methodC()
   {
       System.out.println("MethodC");
   }
}
class hybrid
{
   public static void main(String args[])
   {
       D obj1= new D();
       obj1.methodA();
       obj1.methodB();
       obj1.methodC();
    }
}
```

Output

MethodA
MethodB
MethodC

➢ Nesting of methods

A method can be called by using only its name by another method of same class is nesting of methods.

Example

```
class Nesting
{           int  m,n;
            Nesting(int x , int y)
            {           m=x;
                        n=y;
            }
            int largest ( )                 //defining a method
            {
                  if( m >= n)
                        return(m);
                  else
                        return(n);
}
void display ( )
{
            int large=largest ( );          //calling a method
            System .out. println ("large value"+large);
            }
}
class Nest_test
{
      public static void main(String args[ ])
      {
      Nesting  n=new Nesting ( 50 , 40 );
      n .display ( );
      }
}
```

output : 50

➢ Sub class constructor and super keyword

- Constructor which is defined in subclass is called sub class constructor.
- It is used to create instance variable of both sub class and super class.
- **super** keyword is used.
- We can use super in two ways:
  - **super** is used to invoke constructor method of super class.
  - **super** is used to access variable of super class which have same name as in sub class.
  - We can call method of super class from subclass.

Condition to use super keyword

1. **super** may only be used with in sub class constructor method.
2. Call to **super class constructor must appear as first statement sub class constructor.**
3. The parameter in **super** call must match the **order and type of instance variable** declared in super class.

First use of super:

Constructor in sub (derived) class uses **super** to pass values that are required by super (base) class constructor.

Example

```
class Room                          //super class
{
        int length;
        int width;
        Room ( int x , int y)     //super class constructor
        {
        length=x;
        width=y;
        }
        int area ( )
        {
        return  ( length * width ) ;
        }
}
class BedRoom extends Room          //sub class
{
        int height;
```

```
        BedRoom( int x ,int y, int z)    //sub class constructor
        {
        super( x , y);                   // pass values to super class
        height=z;
        }
        int volume ( )
        {
        return ( length * width * height );
        }
}
class Inheritance
{
        public static void main ( String args[ ])
        {
        BedRoom  room1=new BedRoom ( 10 , 10 , 12);

        int area1=room1.area ( );                    //super class method
        int vol =room1.volume( );                    //sub class method

        System .out .println ("area1=" + area1);
        System .out. println ("volume=" + vol);
        }
}
```

Output :

area1= 100
volume = 120

Second use of super:

We can access variable of super class if sub class and super class have variables of same name.

```
class A
{
        int i ;
}

class B extends A
{       int i;
        B ( int a , int b)
        {
```

```
            super . i =a;
            i =b;
        }
        void show( )
        {
            System . out .println ( “i in super =“ +super .i ) ;
            System . out .println( “i in sub =“ +i);
        }
}

class  SuperDemo
{
        public static void main ( String args[ ])
        {
        B subob= new B( 1, 2);
        subob .  show();
        }
}
```

Output:

```
i in super=1
i in sub=2
```

Third use of super:

```
class A
{
            void show1()
            {
            System . out .println( "from  super class method " );
            }
}
class B extends A
{
            void show( )
            {
            super.show1 ();
            System . out .println( "from sub class method ");
            }
}
class  SuperDemo1
```

```
{
public static void main ( String args[ ])
{
        B obj=new B();
        obj.show();
}
}
```

Output:

```
from  super class method
from sub class method
```

➢ Method overriding (Function Overriding)

- Methods defined in super class is inherited and used by objects of the subclass.
- If we want an object to respond to same method but having different behavior when method is called.
- It is possible by defining a method in sub class that has same name, same arguments, and same return type as in method in the super class.
- *When method in super class is called, the method define in subclass is invoked and executed instead of one in super class this is called overriding of method.*
- *When **same method name, same argument & same return type)** is used in sub class and super class, is called method overriding.*
- We can **overload** methods in same class.
- In this case, Method in sub class overrides the method in super class.

Note:

- Method overloading can be done in **same class.**
- Method overriding can be done in **different class.** When super class and subclas**s** are used (when inheritance is used).

Example of method overriding

```
class SuperClass
        {
                void display( )
                {
                System .out .println ("Super Class");
                }

                void show( int x)
```

```
                {
                System. out .println (" x in super class="+x);
                }
        }
class SubClass extends SuperClass
{
        void display( )                 //method same as in super class
        {
        System .out .println ( " Sub Class");
        }

        void show( int x)    //method same as in super class
        {
        System. out .println (" x in sub class ="+x);
        }
}
class MethodOverRide
{
        public static void main(String args[])
        {
                SubClass subo = new SubClass( );
                        subo . display();
                        subo . display();
                        subo .show(5);

                //SuperClass supero =new SuperClass();
                        //Supero . display();
                        //Supero . show(10);
        }
}
```

Output:
```
Sub class
Sub class
x in sub class= 5
```

Example of method overriding using subclass constructor, super and this

```
class  Super1
{
        int x;
        Super1 (int x)
        {
                this.x =x;
        }
        void display( )
        {
        System .out .println ("super x="+x);
        }
}
class  Sub extends Super1
{
        int y=10;
        Sub (int x, int y)
        {
        super( x);
        this. y =y;  // if put this statement in comment line, y has value 10.
        }
        void display( )
        {
        System .out .println ("super x="+x);
        System .out .println ("sub y="+y);
        }
}
class Override
{
        public static void main(String args[])
        {
        Sub s1=new Sub(100,200);
        s1.display();
        }
}
```

Output: super x=100
sub y=200

Runtime polymorphism or **Dynamic Method Dispatch**

- It is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a super class. The purpose of the method to be called is based on the object being referred to by the reference variable.

Up casting

- When reference variable of Parent class refers to the object of Child class, it is known as up casting.

For example

```
class A
        {
                ……..
        }
class B extends A
        {

                …….

        }
```

In main method class,we can do as follow:

```
B b1=new B();
A a=new B();            //upcasting

        Or
B b1=new B();
A a1;
A1=b1;                  //upcasting
```

Example:

```
class A
{
  void callme()
 {
    System.out.println("Inside A's callme method");
  }
}
class B extends A
{
  void callme()
{
    System.out.println("Inside B's callme method");
  }
}
class C extends A
 {
  void callme()
     {
    System.out.println("Inside C's callme method");
  }
}
class Dispatch
      {
  public static void main(String args[])
     {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r; // obtain a reference of type A
    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme

    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
  }
}
```

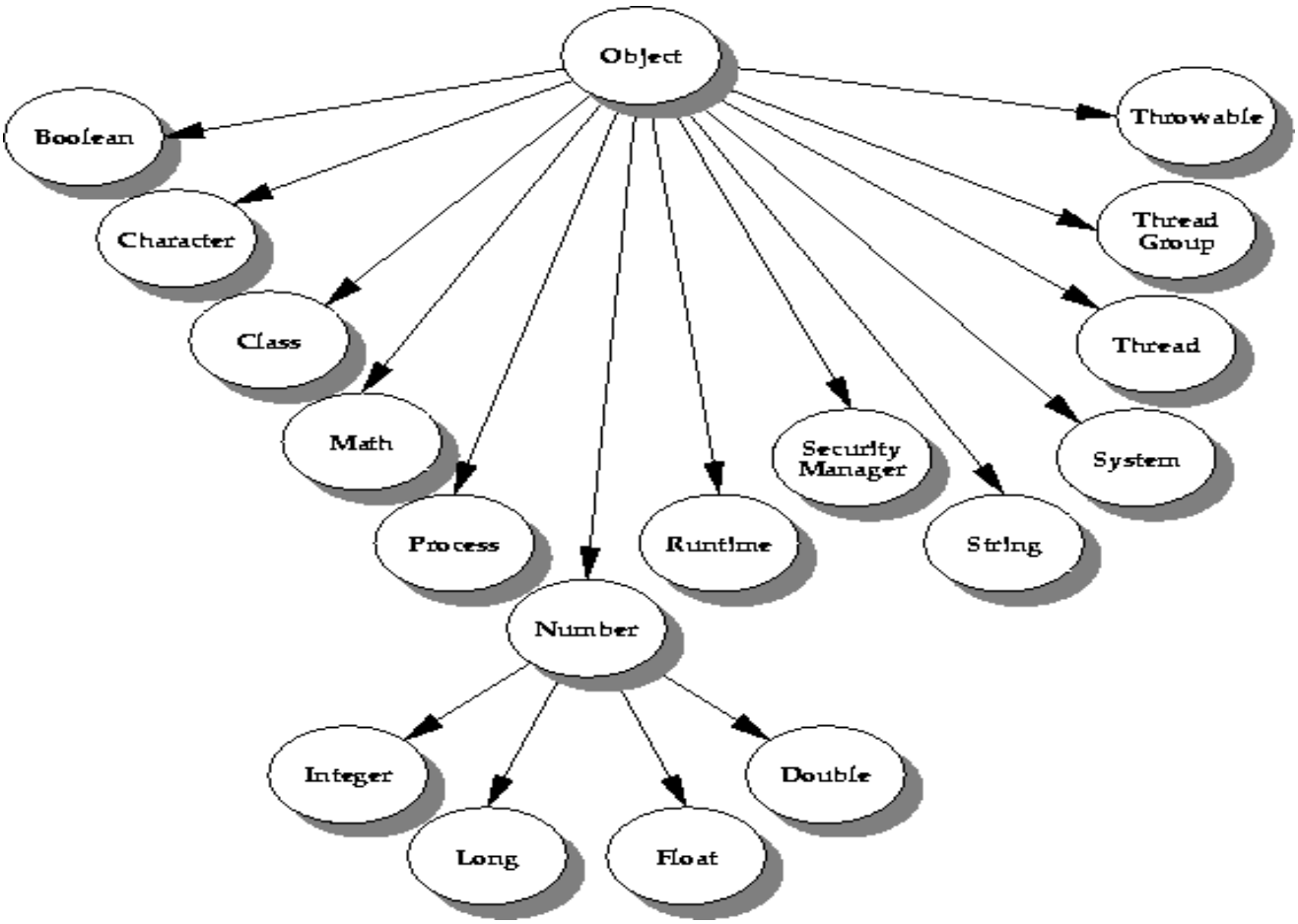
Output:

```
Inside A's callme method
Inside B's callme method
```

```
Inside C's callme method
```

Object class in Java

- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know.
- Note: parent class reference variable can refer the child class object, know as up casting.



Let's take an example, there is **getObject()** method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

Object obj=getObject();//we don't what object would be returned from this method

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

❖ Methods of Object class

No.	Method	Description
1	public final Class **getClass()**	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
2	public int **hashCode()**	returns the hashcode number for this object.
3	public boolean **equals**(Object obj)	compares the given object to this object.
4	protected Object **clone()** throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
5	public String **toString**()	returns the string representation of this object.
6	**public final void notify()**	wakes up single thread, waiting on this object's monitor.
7	**public final void notifyAll()**	wakes up all the threads, waiting on this object's monitor.
8	**public final void wait(long timeout) throws InterruptedException**	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
9	**public final void wait(long timeout,int nanos) throws InterruptedException**	causes the current thread to wait for the specified miliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
10	**public final void wait() throws InterruptedException**	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
11	**protected void finalize() throws Throwable**	is invoked by the garbage collector before object is being garbage collected.

Package

- OOP's main feature is its ability to reuse the code which is already created.
- If we want to use classes from other programs without physically copying them into program under development, then packages are used.
- **Packages** are grouping of variety of **classes and/or Interfaces** together.
- Packages are conceptually similar as "class libraries "of other languages.
- Packages act as "containers" for classes.
- **Java packages are classified into two types**
  - Java API packages **or** system defined package
  - User defined packages

Advantage of package

- The classes contained in the packages of other program can be easily reused.
- In packages, classes can be unique compared with classes in other packages. Means two classes in two different packages can have the same name.
- Packages provide a way to "hide " classes thus preventing other programs or packages from accessing
- Packages also provide a way of separating designing from coding. It is possible to change implementation of any method without affecting the rest of the design.

❖ Java API Packages

(System defined API packages or Standard Java packages)

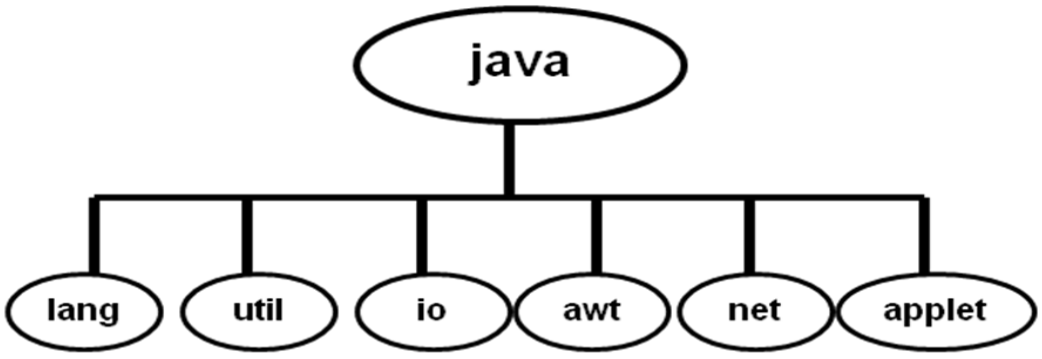


Fig. Java API Packages

No.	Package name	Description

1	java.lang	Language support classes • These are classes that java compiler itself uses and therefore they are automatically imported • They include classes for primitive types, math functions ,threads and exceptions.
2	java.util	• Language utility classes • It includes Vector , hash tables, random numbers.
3	java.io	• Input/output support classes • They provide facilities for the input and output of data
4	java.net	Network communication supporting classes • Classes for communicating with local computers as well as internet servers
5	java.awt	• (Abstract window toolkit) Set of classes for implementing graphical user interface. • They include classes for windows, buttons, lists, menus and many more
6	java. applet	Classes for creating and implementing applets.

Naming convention of package

- Package should begin with lower case letter.
- Every package name must be unique to make best use of packages. Duplicate name will cause errors on internet.
- Java suggests to use domain name as prefix to preferred package name.

  E.g. : cbe.psg.mypackage

  Where **cbe** denotes city name

  psg denotes organization name

Creating package

- To create a package first declares the name of package using package keyword.

  Syntax: **package packagename;**
- This must be first statement in source file.

Example:

```
package firstPackage;  //package declaration
public class First Class   //class declaration
{

        //body of class

}
```

- Save that file as **FirstClass.java** in directory **firstPackage**
- **.class** files must be located in directory **firstPackage.**

❖ Steps to create user-defined package

Question: How to create user-defined package?

Five steps:

1) Declare the package at the beginning of file using , Syntax :**package packagename ;**
2) Define the class that is to be put in the that package and declare it public.
3) Create a subdirectory under the directory where the all main source files are stored.
4) Store the listing file as **the classname.java** file in subdirectory you have created.
5) Compile that file. This will create .class file in subdirectory.

Java also support concept of package hierarchy .This allows to group related classes into a package and then group related packages into a larger package.

E.g. package **firstPackage**.**secondPackage**;

Note: store this package in sub directory

firstPackage\ secondpackage.

- A java package can have **more than one class definitions** in that case only one class may be declared as public and that class name with .java extension is the source file name.
- When a source file with more than one class definition is compiled .Java creates independent .class files for those classes.

Importing packages:

We can import any package (system package or user defined package) using **import keyword.**

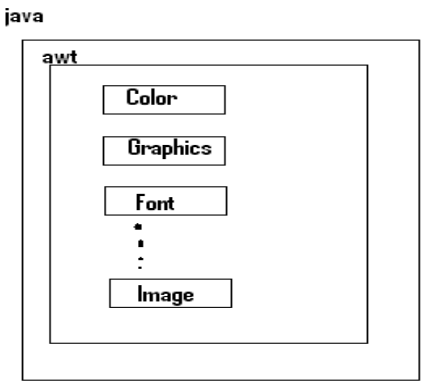
Two approaches:

1) Use fully class sname:

Syntax: import java.Packagename.classname; **ex**: import java.awt.Color;

2) Use short path by putting *:

Syntax : import java.Packagename.*; **ex**: import java.awt.*;

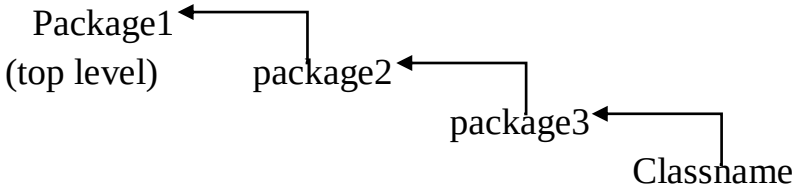


Accessing package:

import statement is used to search a list of packages for a particular class.

Syntax:

import package1 [.package2] [.package3].classname;



- Multiple import statements are allowed.
- * indicates that the compiler should search this entire package hierarchy when it encounters a class name.Drawback of that approach is that it is difficult to determine from which package a particular member came.

Example:

File1: saved in D:\javapro\package1\ClassA.java

```
package package1;
public class ClassA
{
        public void displayA ()
{
            System.out.println ("Class A");
}
}
```

Save this class in file with name ClassA.java in package1

File2: saved in D:\javapro\PackageTest.java

```
import package1.classA;
class PackageTest
{
    public static void main (String args [])
{
ClassA objA=new ClassA ();
objA. displayA ();
}
}
```

Output:

Class A

- This code should be saved as **Packagetest.java** and then compiled.
- In this ClassA imported from package1.The source file and compiled file would be saved in directory of which package1 was a sub directory.
- source file should be named ClassA.**java** and stored in subdirectory package1 and the resultant class ClassA. **Class** will be stored in same directory.
- When the Packagetest 1 is run, java looks for the file **Packagetest1.class** and loads it using class loader .now interpreter knows that it also needs code in the file **ClassA**. Class and loads it as well.

File3: saved in D:\javapro\package2\ClassB.java

```
package package2;
public class ClassB
{
        protected int m=10;
        public void displayB()
    {
        System.out.println("class B");
        System.out.println("m="+m);
    }
}
```

Importing classes from other packages

Here two classes are imported from two different packages.

ClassA from package1 & ClassB from package2

File4: saved in D:\javapro\Packagetest2.java

```
import package1.ClassA;    //imports ClassA from package package1
import package2.*;      //imports all public class of package package2
class Packagetest2
{
    public static void main(String args[])
    {
        ClassA objA=new ClassA ();
        ClassB objB=new ClassB ();
        objA. displayA ();
        objB. displayB ();
    }
}
```

Output:

```
Class B
Class A
M=10
```

Sub classing an imported class:

File5: saved in D:\javapro\Packagetest3.java

```
import package2.classB;
class ClassC extends ClassB              //extends ClassB from
package2
{
    int  n=20;
    void displayC()
    {
    System.out.println("Class c");
    System.out.println("m="+m);
        //protected variable from package2's class ClassB
    System.out.println("n="+n);
    }
}
class PackageTest3
{
        public static void main(String args[])
        {
        ClassC objc =new ClassC();
        objc. displayB ();
        objc. displayC ();
        }
}
```

Output:

```
Class B
m=10
Class C
m=10
n=10
```

Two packages contain Classes with identical names

package pack1;	**package pack2;**
public class Teacher {……}	public class Courses {……}
public class Student {……}	public class Student {……}

```
import   pack1.*;
import   pack2.*;

Student  stu1;                          // in valid
pack1.Student   student1;               //OK
pack2.Student   student2;               //OK

Teacher      t1;                        // OK
Courses      c1;                        //OK
```

Adding a class to a package

1. Place the package statement ------ package p1;
2. Define the class and make it public
3. Store this file as B.java file under the directory p1;
4. Compile B.java file. This will create B.class and place it in the directory p1.

Java source file can have only one class declared as public .This is because of the restriction that the file name of file should be same as name of public class with .java extension.

Create a package with multiple public class

1. Declare the name of package
2. Create a subdirectory with this name under the directory where main source files are stored.
3. Create a class that are to be placed in package in separate source file and declare the package statement : package packagename; at the top each source file
4. Switch to the sub directory created earlier and compile each source file .when completed, the package would contain .class files of all the source files.

❖ Hiding classes

- When we want to import a package using
  import packagename.*;
  then all public classes are imported.
- If we want to hide classes from accessing from outside of the package ,then such classes should be declared "not public".

Ex. :
```
package p1;
public class X
 {
    //body of class
 }
class Y
 {
    //body of class
 }
```

```
import p1.*;
X objx;        // valid, class X is available here
Y objy;        // in valid class y is not available here
```

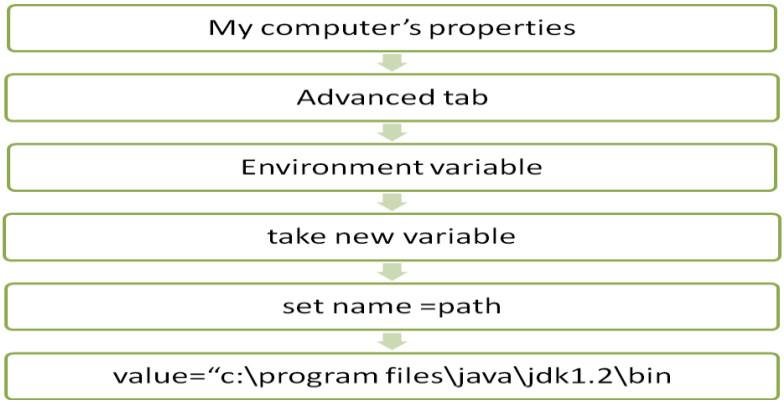
❖ Class path

- The compiler and the interpreter searches for the classes in the current directory and the directory, which consists JDK class files.
- All JDK class files and the current directory are automatically set in your class path.
- But when we want save our class file any other directory in place of JDK 's bin folders, then we have to set class path to run that class file.
- **A class path is a list of directories, which compiler and interpreter use to search for the respective class file.**
- We can set path in following three ways :

1. Javac –d classpath filename (in command prompt)
2. set path="c:\program files\java\jdk1.2\bin";

(type above line in notepad and save with name **autoexec.bat** in directory where you want save your java programs)

3. Globally path set:



Access protection for a classes of package using Visibility controller (Visibility modifier / Access modifiers /Access specifiers)

- It may be necessary in some situation to restrict the access certain variables and methods from outside the class
- We may not like to the objects of a class directly alter the value of a variable or access a method .We can do this by applying visibility modifiers.
- Modifiers provides different levels of protection.
- Java provides three types of modifiers:

  public , private and protected

Access Modifier / Access Location	**public**	**protected**	**Friendly (default)**	**private**
Same class	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	**No**
Other classes in same package	Yes	Yes	Yes	**No**
Subclass in other packages	Yes	Yes	**No**	**No**
Non-subclasses in other package	Yes	**No**	**No**	**No**

Note:

public → protected → friendly(default) → private

1) Public access

- Any variable or method is visible to entire class in which it is defined.
- A variable or method declared as **public** has the widest possible **visibility and accessible everywhere**.
- It can also used in different packages.

Example:

```
public int no;

public void sum( )

{

}
```

2) **friendly (default) Access**

- When no access modifier is specified, the member defaults to a limited version of public accessibility known as **friendly** level of access.
- A variable or method declared as friendly **(default)** makes them **visible to only in that package where it is defined.**
- It is not visible to the outside of package.
- **Difference between public and friendly:**

  Public : it makes field visible in all classes ,regardless of their package.

  friendly : it makes field visible only in the same package , but not in other package.

3) protected access

- The visibility level of protected lies between the public and friendly access.
- **protected** modifiers makes the fields visible **not only to all classes and subclasses in same package but also to subclasses in other package.**
- Non-subclasses in other package can not access "protected".

Ex: protected void finalize( )

{

}

4) private access

- private fields enjoy highest degree of protection.
- private fields are accessible only within their own class.
- They cannot be inherited by subclasses and therefore not accessible in subclasses.
- A method declared as private behaves like a method declared as final.

Summary :

Public: available everywhere

Friendly: available only in same package

Protected: available in same package and subclass in another package

Private: available in only that class in which it is defined

Example for access rules in package

Create below directory structure and learn to use access modifiers in package.

Directory Structure:

package program

```
        |
        |-------------------------------------same_package
        |                                       |-------Demo.java                 (1)
        |                                       |-------TestDemo.java             (2)
        |--- TestDemo_out_pack.java  (4)        |-------Test_extends_demo.java    (3)
        |--- TestDemo_extend_out_pack.java (5)
        .
```

1) Create Demo.java file and save it in following path :

E:\GPA\package program\same_package\Demo.java

```
//package same_package;   //use this line for file 4 and 5
public class Demo
{
        static protected int a=10;
        private int b=20;
        public int c=30;
                void show1()
                {
                System.out.println("----from Demo class show method----");
                System.out.println("a="+a);
                System.out.println("b="+b);
                System.out.println("c="+c);
                System.out.println("-----------------");
                }
    }
```

Compile : javac Demo.java

2) Create **TestDemo.java** file and save it in following path :

E:\GPA\package program\same_package\Test Demo.java

```
//no need to import class in same package
class TestDemo
```

```
{
        public static void main(String arg[])
        {
                Demo d=new Demo();
                d.show1();
        //show1 has default access,in other class in same package,accessible
                System.out.println("a="+d.a);
        //a has protected access,in other class of same package,accessible
                //System.out.println("b="+d.b);
        //b has private access, out side class, not accessible
                System.out.println("c="+d.c);
        //c has public access,in other class in any package,accessible
        }
}
```

Compile : javac TestDemo.java

Run : java TestDemo

Output :

```
----from Demo class show method-----
a=10
b=20
c=30
----------------------
a=10
c=30
```

3) Create **Test_extends_demo.java** file and save it in following path :

E:\GPA\package program\same_package\ Test_extends_demo.java

```
class Test_extends_demo extends Demo
{
        public static void main(String arg[])
        {
                Demo d=new Demo();
                d.show1();
        //show1 has default access,in any class in same package,accessible
                System.out.println("a="+d.a);
        //a has protected access,in sub class of same package ,accessible
                //System.out.println("b="+d.b);
        //b has private access, out side class, not accessible
```

```
            System.out.println("c="+d.c);
        //c has public access,in any class in any package,accessible
        }
}
```

Compile : javac Test_extends_demo.java

Run : java Test_extends_demo

Output :
```
----from Demo class show method-----
a=10
b=20
c=30
----------------------
a=10
c=30
```

4) Create **Test_extends_demo.java** file and save it in following path :

E:\GPA\package program\ TestDemo_out_pack.java

To access Demo.class in this program we have to uncomment following line from **Demo.java** file.

```
import same_package.*;
class TestDemo_out_pack
{
        public static void main(String arg[])
        {
        Demo d=new Demo();
            //d.show1();
        //show1 has default access,in class out side of package,not accessible
            //System.out.println("a="+d.a);
        //a has protected access,in class out side of package ,not accessible
            //System.out.println("a="+d.b);
        //b has private access, out side class, not accessible
            System.out.println("c="+d.c);
        //c has public access,in any class in any package,accessible
        }
}
```

Compile : javac TestDemo_out_pack.java

Run : java TestDemo_out_pack

Output : c=30

5) Create **Test_extends_demo.java** file and save it in following path :

E:\GPA\package program\ TestDemo_extend_out_pack.java

To access Demo.class in this program we have to uncomment following line from **Demo.java** file.

```
import same_package.Demo;
class P extends Demo
{
        void call()
        {
        System.out.println("a="+a);       // a has protected access ,Demo     is extended
                                here ,accessible in subclass out side package ,acessible
        }
}
class TestDemo_extend_out_pack
        {
        public static void main(String arg[])
        {
                P d=new P();
                d.call();
                //d.show1();
        //show1 has default access,in sub class out side of package,not accessible
                //System.out.println("a="+d.a);
//a has protected access,in sub class out side of package directly ,not accessible
                //System.out.println("a="+d.b);
        //b has private access, out side class, not accessible
                System.out.println("c="+d.c);
        //c has public access,in any class in any package,accessible
                }
        }
```

Compile : javac TestDemo_extend_out_pack.java

Run : java TestDemo_extend_out_pack

Output : a=10

c=30

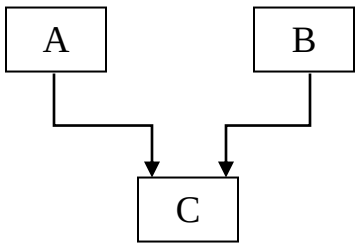
Interface: Multiple Inheritance

Why interface is needed?

- Java does not support multiple inheritances directly.
- Means, Java class cannot have more than super class.

Ex:

```
class  C extends B,A
{
        ---------
}
```



- But large number of real life applications requires the use of multiple inheritances where we can inherit methods and properties from several distinct classes.
- Normally for a method to be called from one class to another, both classes need to be present at compile time, javac checks to ensure that the method signatures are same.
- But when there is hierarchy of class functionality gets higher and higher complicated. Interfaces are designed to avoid this problem.
- Java provides an alternate approach interface to support the concept of multiple inheritance.
- They are designed to support dynamic method resolution at time
- **Java class cannot be a subclass of more than one super class.**
- But it can implement more than one interface, this enabling us to create classes that build upon other classes without problems created by multiple inheritance.

Interface

- Interface is basically a kind of class.
- Interface contains methods and variable but it defines only **abstract method** and **final variable.**
- **Interface do not specify any code to implement methods**(method does not have body) .
- **Variables are constant** (value can't be changed).
- Interface defines :

  What a class do , but not how it does it
- It is responsibility of class that implements an interface, to define code for implementation of these methods.

Syntax to define interface:

```
access interface interfaceName
{
        variable declaration;
        method declaration;
}
```

- interface is keyword**.**
- access: is access modifier, it must be either public or not used any other (friendly).

Declaration of variable :

static final *type* variablename = **value**;

Note:

- All variables are declared as constants.
- It is allowed to declared variable without **final** and **static** ,because all variables in interface are treated as constants.

Method declaration in interface:

returntype methodname **(** parameter-list **)**;

Note:

- Method declaration will contain only a list of methods (without body) like abstract methods.
- Code of that method will implements by class which use that method.
- Method declaration ends with semicolon.
- Methods that implement an interface must be declared **public**.

Example :
```
interface Item
{
static final int  code = 123;    // code is constant .
String  name= "Fan";             // name is constant.
void display ( ) ;               //method without body
}
```

- If no specifier is used then interface is only available to other members of package in which it is declared.

- If public access modifier used, then available to other package also.
- If no specifier is used then interface is only available to other members of package in which it is declared.
- If public access modifier used, then available to other package also.

Interface inheritance or How to extend interface?

- Interface can be extended, means it can be sub inherited from interface.
- New sub interface will inherit all member of super interface in same manner as class.

```
interface interface2 extends interface1
{
    body of interface2
}
```

Ex:

```
interface  ItemConstant
  {
    int code =1001;
    String name="Fan";
  }
interface  Item1 extends ItemConstant
  {
    void display( );
  }
interface Item extends ItemConstant , Item1
{
    ----------
    }
```

- While extending interfaces, sub interface can't define methods declared in super interface (because it's not class).
- When two or more interfaces are extended by one interface, they are separated by comma.
- **Interface can only extend interface but cannot extend classes.**
- If a class that implements an interface does not implement all methods of interface, then class becomes an abstract class, can't be instantiated.

Note:

- extends keyword : class extends class
- extends keyword : interface extends interface
- implements keyword : class implements interface

How to implement an interface to a class?

- Interfaces are act as super class whose properties are inherited by classes.
- **Syntax**:

```
class classname implements InterfaceName
{
        body of class name
}
```

Or

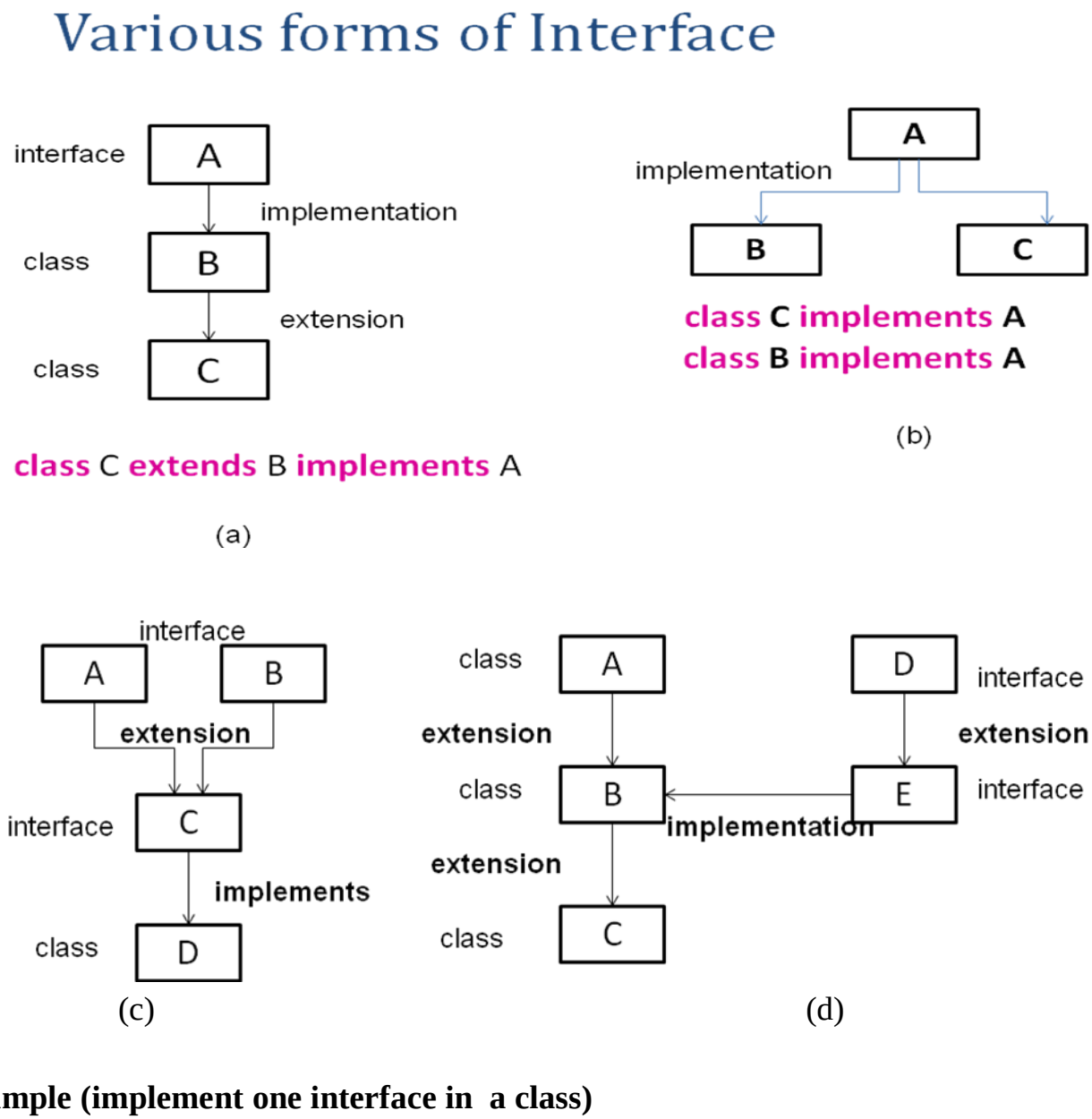
```
class classname extends SuperClass implements   interface1,interface2….
{
        body of classname
}
```

Note :

We can define method with body in interface but it works in jdk1.8 .Earlier jdk version's (like jdk 1.0 to jdk 1.7) don't support method with body in interface.

We can define method as follow

```
interface m1
{       default void show(){
                        //code
                }
 }
```



Example (implement one interface in a class)

```
interface Area          //      interface defined
{
    final static float PI=3.14F;
    float compute(float x, float y);
}

//  interface implemented
class Rectangle implements Area
{
```

```
    public float compute ( float x , float y)
    {
    return  (x * y);
    }
}
class Circle implements Area
{
    public float compute ( float x , float y)
    {
    return  (PI* x * x);
    }
}

class InterfaceTest
{
public  static void main(String args[ ])
    {
    // class object
            Rectangle rect = new Rectangle ( );
            Circle cir=new Circle( );
    // interface object
            Area a ;
            a = rect ;
            System.out.println ( "Area of rectangle =" + a. compute(10f,20f));

            a=cir;
            System.out.println ( "Area of circle =" +       a. compute(10f,0f));


    }
}
```

Output:

Area of rectangle=200.0

Area of circle=314.0

Accessing interface variable

- The constant values will be available to any class that implements the interface.

```
interface A
{
        int m=10;
}
class B  implements A
{
        int x=m;
        ----------
}
```

Implementing multiple inheritance

write a program which performs multiple inheritance

Example-1:

```
interface Sports
{
    float sportWt=6.0f;
    void putWt();
}
interface Display
{
    void display();
}
class Student
{
    int rollno;

    void getNo(int n)
    {
     rollno=n;
    }
    void putNo()
    {
      System.out.println("Roll No="+rollno);
    }
}
```

```
class Test extends Student
    {
        float part1,part2;
        void getmarks(float m1,float m2)
        {
                part1=m1;
                part2=m2;
        }
        void putmarks()
        {
                System.out.println("Marks obtained");
                System.out.println("Part1="+part1);
                System.out.println("part2="+part2);
        }
    }
    // multiple interface and class are accessed by one class

    class Results extends Test implements Sports,Display
    {
        float total;
        public void putWt()
        {
                System.out.println("Sports Wt="+sportWt);
        }
        void display()
        {
                total=part1+part2+sportWt;
                putNo();
                putmarks();
                System.out.println("Total score="+total);
        }
    }
    class Hybrid
    {
    public static void main(String args[])
    {
        Results stu1=new Results();
        stu1.getNo(1234);
        stu1.getmarks(2.1f,33.0f);
        stu1.display();
    }
    }
```

Example-2: multiple inheritance using interface

```
interface A1
    {
        int marks1=90;
    }
interface A2
    {
        int marks2=80;
    }
interface A3 extends A1,A2
    {
        int marks3=70;
        void sum( );
    }
class Result implements A3
    {
        int total;

        public void sum( )
        {
            total=marks1+marks2+marks3;
            System.out.println("Total marks="+ total );
        }
    }

class Multi_Inheri
{
    public static void main( String args[ ])
    {
    Result r=new Result ( );
    r.sum ( );

    }
}
```

Output :

Total marks=240

Example-3: Write a program that demonstrates interface inheritance. Interface P12 inherits from both P1 and P2. Each interface declares one constant and one method. The class Q implements P12. Instantiate Q and invoke each of its methods. Each method displays one of the constants

```
interface P1
{
        int i=20;
        void ishow();
}
interface P2
{
        int j=20;
        void jshow();
}
interface P12 extends P1,P2
{
        int k=40;
        void kshow();
}
class Q implements P12
{
        public void ishow()
        {
                System.out.println("from P1 :"+i);
        }
        public void jshow()
        {
                System.out.println("from P2 :"+j);
        }
        public void kshow()
        {
                System.out.println("from P12 :"+k);
        }
}
class Interfaceinheritance
{       public static void main ( String args[ ])
        {
        Q obj=new Q();
        obj.ishow();
        obj.jshow();
        obj.kshow();
        }                               }
```

Output

```
from P1 :20
from P2:30
from P12 :40
```

Write a program which uses 'date' package to display tomorrow's date.

```
package date;
import java.util.*;

public class A
{
    public void display()
    {

    String months[ ] = {"Jan", "Feb", "Mar", "Apr" ,"May", "Jun", "Jul", "Aug"
    ,"Sep", "Oct", "Nov", "Dec"};

            Calendar c1 = Calendar.getInstance();

            System.out.print(   "Today's Date: "     );
            System.out.print(   months[ c1.get(Calendar.MONTH)]  + "/" );
            System.out.print(          c1.get(Calendar.DATE )          );
            System.out.println( "/" +   c1.get(Calendar.YEAR)                );

            System.out.print(   "Today's Date: "     );
            System.out.print(   months[ c1.get(Calendar.MONTH)]  + "/" );
            System.out.print(          c1.get(Calendar.DATE )   + 1   );
            System.out.println( "/" +   c1.get(Calendar.YEAR)                );
    }
}
import date. A;
import java.util.Calendar;
class Date_disp
{
    public static void main(String args[])
    {
        A t1=new A();
        t1.display();
    }
}
```

Abstract class and methods

- **final keyword prevents method overriding**. final makes method protected means it can't be extended. Java also allows us to exactly opposite to above one.
- If we want method must always be redefined in subclass, to make overriding compulsory.
- **abstract (modifier) keyword is used to make overriding compulsory**.
- **abstract** keyword can be used with class and methods.

Ex:

```
abstract class Shape
    {
    ------------
    abstract void draw( );
    -----------
    -----------
    }
```

- Abstract methods are referred as sub classer responsibility because they have no implementation specified in the super class.Subclass must override them, it cannot simply use the version defined in super class.
- Abstract Method in class does not have body.

  Syntax: abstract type ***name*** (parameter-list);
- If a class contains one or more abstract methods, it should be declared as abstract class.
- Any subclass of an abstract class must either implement all of the abstract methods in the super class, or be itself declared abstract.
- Abstract class has both abstract (without body) and concrete methods (normal method which have body).

- Conditions to use abstract:

1. We cannot use abstract classes to initiate objects directly.
   Shape s=new **Shape**( );
2. The abstract methods of an abstract class must be redefined in its subclass.
3. We cannot declare abstract constructor or abstract static methods.

Example:

```
abstract class  A
{
        abstract void callme ( );   //abstract method(without  body)
                        // concrete method are still allowed in abstract class
        void callmeToo( )           //concrete method(with body)
        {
        System . out .println( "this is concrete method");
        }
}

class B extends A
{
        void callme( )
        {       System. out .println("B's implementation of call me");   }
}

class AbstractDemo
{
        public static void main(String args[])
        {
                B  b1=new B( );
        //      A  a1=new A( );
        // invalid, because  we cannot create object of abstract class.
                A a1;
                a1=b1;
                b1.callme( );
                b1.callmeToo( );
                a1.callme( ) );
        }
}
```

Output:

```
B's implementation of call me
this is concrete method
B's implementation of call me
```

Example:

```
abstract class  A
{
        abstract void sum (int a,int b);
        abstract void sub (int a ,int b);
        abstract void mul (int a,int b );
}

abstract class B extends A
{
B()
{
System.out.println("");
}
        void sum(int i,int j)
        {
         int sum=i+j;
         System.out.println("sum="+sum);
        }
        void sub(int i,int j)
        {
         int sub=i-j;
         System.out.println("sub="+sub);
        }
}
class C extends B
{
        void mul(int x,int y)
        {
         int ans=x*y;
         System.out.println("multiplication="+ans);
        }
}
class AbstractDemo1
{
public static void main(String args[])
```

```
{
        C  c1=new C( );
        c1.sum(10,20);
        c1.sub(10,5);
        //c1.mul(10,20);

        B b1;
        b1=c1;
        b1.mul(10,20);
}
}
```

Output :

Sum=30
Sub=5
Multiplication=200

Write a java program which declares one abstract class called Shape which has three subclasses say Triangle, Rectangle, and Circle. Define one method area( )in the abstract class and override this area( ) in these three subclasses to calculate for specific object i.e. area() of Triangle subclass should calculate area of triangle etc. Same for Rectangle and Circle.

```
abstract class Shape
{
        abstract void area (double val1,double val2 );          //abstract
method(without  body)
}
class Rectangle extends Shape
{
        public void area(double l,double b)
        {
                double ans1=l*b;
                System.out.println("area of rectangle ="+ans1);
        }
}
class Circle extends Shape
```

```
{
        public void area(double r,double b)
        {
                double ans1=Math.PI*r*r;
                System.out.println("area of circle ="+ans1);
        }
}
class Triangle extends Shape
{
        public void area(double b,double h)
        {
                double ans1=0.5*b*h;
                System.out.println("area of Triangle ="+ans1);
        }
}
class AbstractShape
{
        public static void main(String args[])
        {
                Rectangle  r=new Rectangle( );
                r.area(10,5);
                Circle c=new Circle();
                c.area(10,0);
                Triangle t=new Triangle();
                t.area(20,10);
        }
}
```

Output:

area of rectangle =50.0

area of circle=314.1592653589793

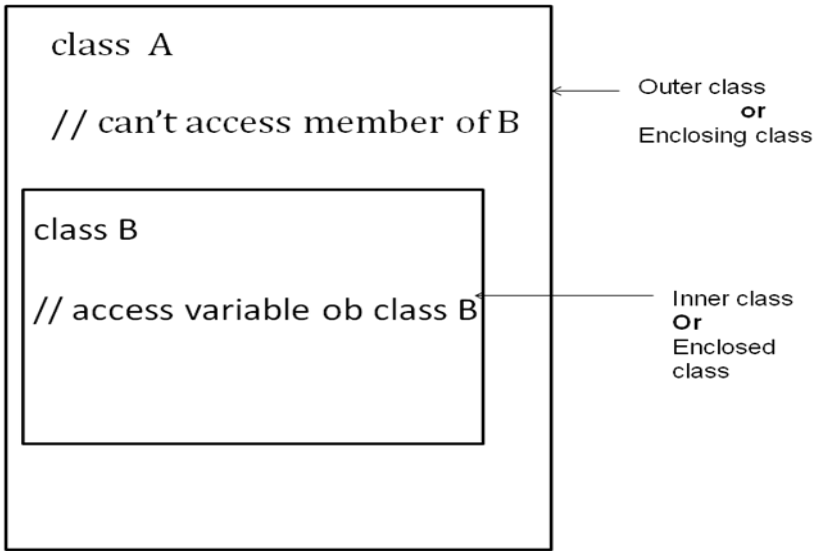
area of Triangle=100.0

Interface Vs Abstract class

	Interface	Abstract class
1	Method of interface are implicitly abstract (without body) and cannot have implementations.	Methods of abstract class can have abstract method (without body) and concrete methods (with body statement).
2	Interface contains only final variables. because Variable declared in interface is by default final.	Abstract class may contain non-final variables.
3	Members of interface are public by default.	A abstract class can have the different modifiers of class members like private, protected, etc.
4	interface should be implemented using keyword "implements";	abstract class should be extended using keyword "extends".
5	An **interface** cannot provide any code at all, much less default code.	An **abstract** class can provide complete code, default code, and/or just stubs that have to be overridden.
6	An interface can extend another Java interface only.	An abstract class can extend another Java class and implement multiple Java interfaces.
7	Interface cannot extend any type of class.	A Java class can extend only one abstract class.
8	Interface is absolutely abstract and cannot be instantiated	A Java abstract class also cannot be instantiated, but can be invoked if main ( ) exists.
9	If all the various implementations of code share the method signature then an interface works best.	If the various implementations of code are all of a kind and share a common status and behavior usually an abstract class work best.

Nested Inner class

- It is possible to define class within another class , that is known as nested class.
- Nested class has access to member including private member of class in which it is nested.but that class can't access to member of nested class.
- Two types of nested class,
  - static
  - Non-static
- Nested classes are particularly helpful in handling events in applets.



Static nested class:

- Class has **static** modifier specified.
- It access member of its enclosing class through an object.
- It cannot refer member of its enclosing class directly.

Non-static nested class:

Most important type of nested class is **inner class.**

- **Inner class is non static nested class.**
- It can access all variables and method of its outer class.
- Inner class is fully within the scope of its enclosing class.

Example:

```
class Outer
{
        int out_x =100;
        void test ( )
```

```
        {
                Inner inobj =new Inner( );
                inobj . display( );
                // display( );          gives error
        }
        class Inner
        {
        int y = 10;
                void display( )
        {
        System . out. println( "display : outer_x="+ out_x);
        }
        }
        void show( )
        {
        System . out. println( "y="+ y);//invalid
        }
}
class InnerClassDemo
{
public static void main (String args[ ])
{
        Outer outobj =new Outer ( );
        // Inner inobj =new Inner ( );
        outobj . test ( );
        //outobj . show ( ); //error, method can't use variable y of Inner class
        //outobj. display ( ); // error, it is method of Inner class
}
}
```

Output: out_x=100

Recursion

- Recursion is a function that allows method to call itself.
- When recursive method calls itself new local variables and parameter are removed from the stack and execution resumes.
- A recursive call does not make a new copy of method but arguments are new.
- Advantage: it is used to create simple and clearer version of several algorithm.
- We must have if statement that returns method without recursive call.
- If you don't use if statement for recursive call recursive method goes in the infinite loop.

```
class Sum
{
        int  sumn( int i)
        {
                int n;
                if ( i==1)
                        return 1;
                else
                        n=sumn (i-1)+ i ;
                        return n;
        }
}
class RecursionDemo

{
        public static void main(String args[])
        {
         Sum s=new Sum( );
        int total=s . sumn( 5) ;
        System . out .println( "sum of 1 to 5=" + total);
        }
}
```

Output: sum of 1 to 5= 15

Write a program to find factorial of given no. Using recursion.

```
class Factorial
{
int  fact( int i)
{
        int n;
        if ( i==1)
                return 1;
        else
                n=fact (i-1)* i ;
                return n;
}
}

class Fact1
{
public static void main(String args[])
{

        int no=Integer.parseInt(args[0]);
         Factorial s=new Factorial( );
        int total=s . fact( no) ;
        System . out .println( "factorial of=" + no + " is " + total);
}
}
```

```
javac Fact1.java
java Fact1            6
```

Output: factorial of 6 is 720