**Chapter Three**

**Inheritance and Polymorphism**

**3.1. Inheritance:**

To know the concept of inheritance clearly you must have the idea of class and its features like methods, data members, access controls, constructors, keywords this, super etc.

As the name suggests, inheritance means to take something that is already made. It is one of the most important features of Object Oriented Programming. It is the concept that is used for reusability purpose. Inheritance is the mechanism through which we can derive classes from other classes. The derived class is called as child class or the subclass or we can say the extended class and the class from which we are deriving the subclass is called the base class or the parent class. To derive a class in java the keyword extends is used.

*In inheritance, a new class is defined by means of an older, pre-existing class. This leads to a situation in which, the new class has all the functionality of the older, pre-existing class and, additionally, introduces its own specific functionality. We say the new class(child or subclass or derived class) inherits the functionality of another existing class (base or super class).*

One advantage of OOP is the *re-usage* of code. The capability to define custom data types using classes enables us to reuse the code that we develop. In real world, we may need an object that is almost similar to an already developed object but not exactly similar. Inheritance enables us to reuse an object more quickly; thus making slight adjustments where necessary.

To create a new derived class based on another pre-existing class, use the following syntax:

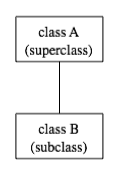
*modifier* class *SubClass* extends *BaseClass {*

// Body of the Subclass

}

***The following kinds of inheritance are there in java.***

**Simple Inheritance: One level inheritance**

When a subclass is derived simply from its parent class then this mechanism is known as simple inheritance. In case of simple inheritance there is only a sub class and its parent class. It is also called single inheritance or one level inheritance.

In Java, to create a class named "B" as a subclass of a class named "A", you would write:-

class B extends A {

// additions to, and modifications of,

// stuff inherited from class A

}

Single level inheritance Example:

public class baseclass {

public intx,y;

public baseclass(){

x=0; y=0;

System.out.println("Baseclass is executed");

}

public baseclass(int x1,int y1){

x=x1;

y=y1;

}}

public class subclass extends baseclass {

public int z;

public subclass(){

super();

this.z=0;

System.out.println("Subclass is executed");

}

public subclass(int x1,int y1,int z1){

super(x1,y1);

this.z=z1;

} }

public class Testsubclass {

public static void main(String args[]){

subclass sc=new subclass();

System.out.println(sc.x);// output 0

sc=new subclass(4,8,1);

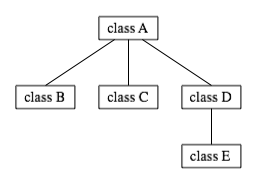
System.out.println(sc.x);// output 4

baseclassbc=new baseclass (2,5);

System.out.println(bc.x);

}}

**Multilevel inheritance:**

Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors -- namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram to the left, classes B, C, and D are sibling classes. Inheritance can also extend over several "generations" of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass. This whole set of classes form a small class hierarchy. Such type of inheritance is called multilevel inheritance.

Multilevel Inheritance Example:

class A {

System.out.println(“Class A is executed”);  
  }  
class B extends A{  
    System.out.println("Subclass B Derived from class A executed");  
}  
class C extends B{  
    System.out.println("Subclass C Derived from class B executed");  
  }  
  public static void main(String args[]){  
    C c  = new C();  
        }}

The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritances. Java does not support multiple inheritances but the multiple inheritances can be achieved by using the interface.

In Java Multiple Inheritances can be achieved through use of Interfaces by implementing more than one interface in a class.

**Using *super* and *this* keyword**

The super is java keyword. As the name suggest super is used to access the members of the super class. It is used for two purposes in java. The first use of keyword super is to access the hidden data variables of the super class hidden by the sub class.

Suppose class A is the super class that has two instance variables as int a and float b. class B is the subclass that also contains its own data members named a and b. then we can access the super class (class A) variables a and b inside the subclass class B just by calling the following command.

super.member;

Here member can either be an instance variable or a method. This form of super is most useful to handle situations where the local members of a subclass hide the members of a super class having the same name. The following examples clarify all the confusions.

class A{  
  int a;  
  float b;  
  void Show(){  
    System.out.println("b in super class:  " + b);  
  }}  
class B extends A{  
  int a;   
  float b;  
  B( int p, float q){  
    a = p;  
    super.b = q; //using variable b from super class

this.b=6;  
  }

  void Show(){  
    super.Show(); //calling super method show()  
    System.out.println("b in super class:  " + super.b);

//display value of super variable b  
    System.out.println("a in sub class:    " + a);

System.out.println(“b in subclass : ”+ b);

//or this.b,used to display values of subclass variable b  
  }}  
  class testsubclass{

public static void main(String[] args){  
    B subobj = new B(1, 5);  
    subobj.Show();  
  }}

The second use of super to call super class constructor: The second use of the keyword super in java is to call super class constructor in the subclass. This functionality can be achieved just by using the following command.

super (param-list);

Here parameter list is the list of the parameter requires by the constructor in the super class. super must be the first statement executed inside a super class constructor. If we want to call the default constructor then we pass the empty parameter list. The following program illustrates the use of the super keyword to call a super class constructor.

class A{  
  int a;  
  int b;  
  int c;  
  A(int p, int q, int r){  
    a=p;  
    b=q;  
    c=r;  
  }}  
    
  class B extends A{  
    int d;  
    B(int l, int m, int n, int o){  
      super(l,m,n);  
      d=o;  
      }  
    void Show(){  
      System.out.println("a = " + a);  
      System.out.println("b = " + b);  
      System.out.println("c = " + c);  
      System.out.println("d = " + d);  
    }}

Class testsupermethod{  
    public static void main(String args[]){  
      B b = new B(4,3,8,7);  
      b.Show();  
    }  }

In general, the keyword *super()* has two general forms. The *first* is *call of superclass constructor*. The *second* is used to access a *member (data and method) of the superclass* that has been *hidden* by a member of the subclass. Additional examples to see each use of the super keyword:

Example 1:

public class person {

private String name;

private int age;

public person(String name,int age)

{

this.name = name;

this.age = age;

}

public void printData1()

{

System.out.println("Name->"+name);

System.out.println("Age->"+age);

}

}//end class

public class student extends person {

String idno ;

String department;

public student(String studname,intstudage,String id, String dept)

{

//calling superclass constructor

super(studname,studage);

this.idno=id;

this.department=dept;

}

public void printData2()

{

System.out.println("Id No.->"+idno);

System.out.println("Department->"+department);

} }

public class Teststudent {

public static void main(String args[])

{

student stud = new student("Abreham", 24, "TER/123/01", "IT");

stud.printData1();

stud.printData2();

} }

Example 2:

public class ClassA{

inti;

}//end class

public class classB extends ClassA{

inti;//this i hides the i in class ClassA

public ClassB(inta,int b){

super.i = a;//i in ClassA

i = b;//i in ClassB

}

public void show(){

System.out.println(“i in superclass : ”+super.i);

System.out.println(“i in subclass : ”+i);

}}

public class TestSuper{

public static void main(String args[]){

ClassBclsB = new ClassB(10,20);

clsB.show();

}}

Although the instance variable i in ClassB hides the i in ClassA, super allows access to the i defined in the superclass. Super can also be used to call methods that are hidden by subclasses.

**Use of *final keyword* with Inheritance**

A class can be declared final if its definition is complete and no subclasses are desired or required. A compile-time error occurs if the name of a final class appears in the extended clause of another class declaration; this implies that a final class can’t have any subclasses.

Final keyword prevents *overriding*. To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final can’t be overridden. Have a look at the following example.

public class ClassA{

public final void firstMethod( ){

System.out.println(“This is a final method declared only once”);

}}

public class ClassB extends ClassA{

public void firstMethod( ){

System.out.println(“illegal to declare this method again”);

} }

Final prevents *inheritance*. Sometimes, you will want to prevent a class from being inherited. To do this, precede the class declaration with the keyword *final*. Declaring a class as final implicitly declares *all of its methods* as final too. Consider the following example.

public final classClassA{

//…

}//end class

public class ClassB extends ClassA{//illegal inheritance

}//end class

**Constructor order dependencies**

Instantiating a subclass object begins a chain of constructor calls in which the subclass constructor, before performing its own tasks, invokes its direct superclass’s constructor (calling the superclass’s default constructor or no-argument constructor). Similarly, if the superclass was derived from another class, the superclass constructor would be required to invoke the constructor of the next class up in the hierarchy, and so on. The last constructor called in the chain is always the constructor of class *Object*. The original subclass constructor’s body finishes executing *last*.

**Constructor example**

Ex1). The following three classes demonstrate how constructors are called for subclass and superclass.

public class ParentClass {

public ParentClass() {

System.out.println( "ParentClass constructor was called" );

}}

public class ChildClass extends ParentClass {

public ChildClass() {

System.out.println( "ChildClass constructor was called" );

}}

public class test {

public static void main(String[] args) {

ChildClass cc = new ChildClass();

}}

When you extend a class, the new class must choose one of its superclass's constructors to invoke.

Ex 2) When an object is created, it's necessary to call the constructors of all super classes to initialize their fields. Java does this automatically at the beginning *if you don't*.

Public class Base{

System.out.println(“Base class accessed first”);

}

public class Derived1 extends Base{

System.out.println(“ Derived1 class is accessed secondly”);

}

Public class Derived2 extends Derived1 {

System.out.println(“ Derived2 class is accessed Thirdly”);

}

public class test {

public static void main(String[] args)

Derived2 d2=new Derived2();//It automatically creates default constructors of //base and super constructors

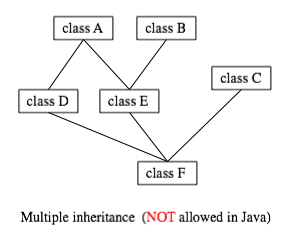
}}

**Private Methods Are Not Inherited**

As we noted in the previous topics, an instance variable (or method) that is private in a base class is not directly accessible in the definition of a method for *any other class, not even in a method definition for a derived class*. Note that private methods are just like private variables in terms of not being directly available. But in the case of methods, the restriction is more dramatic. A private variable can be accessed indirectly (through public methods of the class). A private method is simply not available. It is just as if the private method were not inherited.

This should not be a problem. Private methods should just be used as helping functions, and so their use should be limited to the class in which they are defined. If you want a method to be used as a helping method in a number of inherited classes, then it is not *just* a helping method, and you should make the method public.

**Interfaces: Multiple inheritances in java**

Some object-oriented programming languages, such as C++, allow a class to extend two or more super classes. This is called multiple inheritance. In the illustration below, for example, class E is shown as having both class A and class B as direct super classes, while class F has three direct super classes.

Such multiple inheritance is not allowed in Java. The designers of Java wanted to keep the language reasonably simple, and felt that the benefits of multiple inheritances were not worth the cost in increased complexity. However, Java does have a feature that can be used to accomplish many of the same goals as multiple inheritances called interface.

In Java, interface is a reserved word with an additional, technical meaning. An "interface" in this sense consists of a set of instance method interfaces, without any associated implementations. A class can implement an interface by providing an implementation for each of the methods specified by the interface.

Interfaces are similar to abstract classes but all methods are abstract and all properties are static final. Interfaces can be inherited (i.e. you can have a sub-interface). As with classes the extends keyword is used for inheritance.Java does not allow multiple inheritance for classes (i.e. a subclass being the extension of more than one superclass). An interface is used to tie elements of several classes together. Interfaces are also used to separate design from coding as class method headers are specified but not their bodies.

**3.2. Polymorphism in Java**

**Polymorphism in java** is a concept by which we can perform a single action by different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

**3.2.1. Method Overloading in Java**

If a class has multiple methods by same name but different parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs. So, we perform method overloading to figure out the program quickly.

### Different ways to overload the method

There are two ways to overload the method in java

1. By changing the data type
2. By changing number of arguments

#### In java, Method Overloading is not possible by changing the return type of the method.

Example:

class Maths{

  void sum(int a,int b){

System.out.println(a+b);

}

  void sum(int a,int b,int c){

System.out.println(a+b+c);

}

void add(int a,int b){

System.out.println(a+b);

}

void add(double a,double b){

System.out.println(a+b);

}

  public static void main(String args[]){

  Mathsm=new Maths ();

  m.sum(10,10,10);

  m.sum(20,20);

m.add(10.5,10.5);

 m.add(20,20);    }  }

**3.2.2. Method Overriding in Java**

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

In other words, if subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

### Usage of Java Method Overriding

* Method overriding is used to provide specific implementation of a method that is already provided by its super class.
* Method overriding is used for runtime polymorphism

#### Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

Example:

class Vehicle{

void run(){

System.out.println("Vehicle is running");

}  }

class Bike2 extends Vehicle{

void run(){

System.out.println("Bike is running safely");

}

public static void main(String args[]){

Bike2 obj = new Bike2();

obj.run();

}

# Difference between method overloading and method overriding in java

|  |  |  |
| --- | --- | --- |
| **No.** | **Method Overloading** | **Method Overriding** |
| 1) | used *to increase the readability* of the program. | used *to provide the specific implementation* of the method that is already provided by its super class. |
| 2) | performed *within class*. | occurs *in two classes* that have IS-A (inheritance) relationship. |
| 3) | *parameter must be different*. | *parameter must be same*. |
| 4) | example of *compile time polymorphism*. | example of *run time polymorphism*. |
| 5) | can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. | *Return type must be same or covariant* in method overriding. |

## 3.2.3. Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending SMS, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

### Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

## 3.2.3.1 Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

### Abstract class in Java

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

### Example abstract class

1. abstract class A{}

### Abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

### Example abstract method

1. abstract void printStatus();//no body in abstract

### Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

abstract class Bike{

  abstract void run();

}

  class Honda4 extends Bike{

void run(){System.out.println("running safely..");}

public static void main(String args[]){

 Bike obj = new Honda4();

 obj.run();  }  }

### Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the factory method.

A factory method is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

abstract class Shape{

abstract void draw();

}

//In real scenario, implementation is provided by others i.e. unknown by end user

class Rectangle extends Shape{

void draw(){System.out.println("drawing rectangle");}

}

class Circle1 extends Shape{

void draw(){System.out.println("drawing circle");}

}

//In real scenario, method is called by programmer or user

class TestAbstraction1{

public static void main(String args[]){

Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape() method s.draw();

}  }

### Another example of abstract class in java

 abstract class Bank{

abstract int getRateOfInterest();

}

class SBI extends Bank{

int getRateOfInterest(){return 7;}

}

class PNB extends Bank{

int getRateOfInterest(){return 7;}

}

class TestBank{

public static void main(String args[]){

Bank b=new SBI();//if object is PNB, method of PNB will be invoked

int interest=b.getRateOfInterest();

System.out.println("Rate of Interest is: "+interest+" %");

}}

### Abstract class having constructor, data member, methods etc.

An abstract class can have data member, abstract method, method body, constructor and even main() method.

//example of abstract class that have method body

 abstract class Bike{

   Bike(){System.out.println("bike is created");}

   abstract void run();

   void changeGear(){System.out.println("gear changed");}

 }

 class Honda extends Bike{

 void run(){System.out.println("running safely..");}

 }

 class TestAbstraction2{

 public static void main(String args[]){

  Bike obj = new Honda();

  obj.run();

  obj.changeGear();

 }  }

#### Rule: If there is any abstract method in a class, that class must be abstract.

class Bike12{

abstract void run();  //it is the abstract class b|c the method is abstract

}

#### Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

### Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

#### Note: If you are beginner to java, learn interface first and skip this example.

interface A{

void a();

void b();

void c();

void d();

}

abstract class B implements A{

public void c(){System.out.println("I am C");}

}

class M extends B{

public void a(){System.out.println("I am a");}

public void b(){System.out.println("I am b");}

public void d(){System.out.println("I am d");}

}

class Test5{

public static void main(String args[]){

A a=new M();

a.a();

a.b();

a.c();

a.d();

}}

## 3.2.3.2. Interface in Java

An interface in java is a blueprint of a class. It has static constants and abstract methods only. The interface in java is a mechanism to achieve fully abstraction. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

* Java Interface also represents IS-A relationship.
* It cannot be instantiated just like abstract class.

## Why use Java interface?

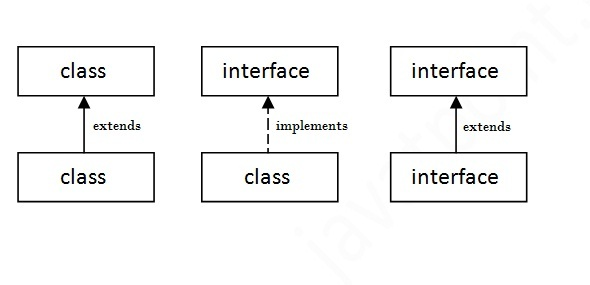
There are mainly three reasons to use interface. They are given below.

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

#### The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

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

#### Understanding relationship between classes and interfaces

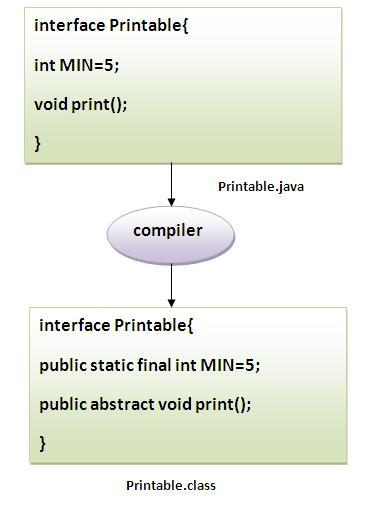
As shown in the figure given below, a class extends another class, an interface extends another interface but a class implements an interface.

## Simple example of Java interface

## In this example, Printable interface have only one method, its implementation is provided in the A class.

interface printable{

void print();

}

class A6 implements printable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

A6 obj = new A6();

obj.print();

 }  }

## 

**Multiple inheritance in Java by interface**

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

interface Printable{

void print();

}

interface Showable{

void show();

}

class A7 implements Printable,Showable{

public void print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");}

public static void main(String args[]){

A7 obj = new A7();

obj.print();

obj.show();

 } }

## Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

## As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface *because there is no ambiguity as implementation is provided by the implementation class.* For example:

interface Printable{

void print();

}

interface Showable{

void print();

}

class testinterface1 implements Printable,Showable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

testinterface1 obj = new testinterface1();

obj.print();

 }  }

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class A, so there is no ambiguity.

## Interface inheritance

A class implements interface but one interface extends another interface.

interface Printable{

void print();

}

interface Showable extends Printable{

void show();

}

class Testinterface2 implements Showable{

public void print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");}

public static void main(String args[]){

Testinterface2 obj = new Testinterface2();

obj.print();

obj.show();

 }  }

#### Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

interface printable{

 void print();

 interface MessagePrintable{

   void msg();

 } }

# Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| Abstract class | Interface |
| can have abstract and non-abstract methods. | can have only abstract methods. |
| doesn't support multiple inheritance. | supports multiple inheritance. |
| can have final, non-final, static and non-static variables. | has only static and final variables. |
| can have static methods, main method and constructor. | can't have static methods, main method or constructor. |
| can provide the implementation of interface. | can't provide the implementation of abstract class. |
| The abstract keyword is used to declare abstract class. | interface keyword is used to declare interface. |
| e.g.: public class Shape{ public abstract void draw();} | e.g.: public interface Drawable{ void draw();} |

* Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

### Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

//Creating interface that has 4 methods

interface A{

void a();//bydefault, public and abstract

void b();

void c();

void d();

}

  //Creating abstract class that provides the implementation of one method of A interface

abstract class B implements A{

public void c(){System.out.println("I am C");}

}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods class M extends B{

public void a(){System.out.println("I am a");}

public void b(){System.out.println("I am b");}

public void d(){System.out.println("I am d");}

}

//Creating a test class that calls the methods of A interface

class Test5{

public static void main(String args[]){

A a=new M();

a.a();

a.b();

a.c();

a.d();  }}

# 3.3. Java Package

A java package is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

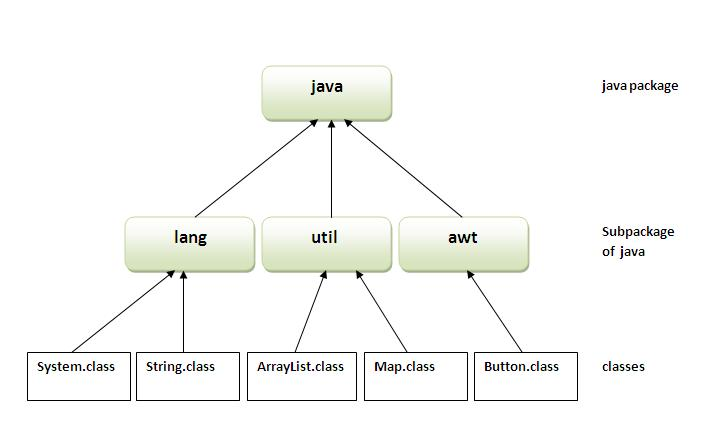
## Advantage of Java Package

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.

## Simple example of java package

****

The package keyword is used to create a package in java.

//save as Simple.java

package mypack;

public class Simple{

 public static void main(String args[]){

    System.out.println("Welcome to package");

}}

## How to compile java package

If you are not using any IDE, you need to follow the syntax given below:

1. javac -d directory javafilename

For example

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use**.**(dot).

## How to run java package program

You need to use fully qualified name e.g. mypack.Simpleetc to run the class.

|  |
| --- |
| To Compile:javac -d . Simple.java |
| To Run: java mypack.Simple |

Output:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

## How to access package from another package?

There are three ways to access the package from outside the package.

import package.\*;

import package.classname;

fully qualified name.

#### 1) Using packagename.\*

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.\*

//save by A.java

package pack;

public class A{

  public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.\*;

class B{

  public static void main(String args[]){

   A obj = new A();

   obj.msg();    }  }

#### 2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

## Example of package by import package.classname

//save by A.java

package pack;

public class A{

  public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.A;

class B{

  public static void main(String args[]){

   A obj = new A();

   obj.msg();

  }  }

#### 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

## Example of package by import fully qualified name

//save by A.java

  package pack;

public class A{

  public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

class B{

  public static void main(String args[]){

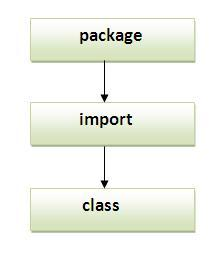
   pack.A obj = new pack.A();//using fully qualified name

   obj.msg();

  }  }

#### Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the

****

classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

#### Note: Sequence of the program must be package then import then class.

## Subpackage in java

Package inside the package is called the subpackage. It should be created to categorize the package further.

Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/output related classes in io package, Server and ServerSocket classes in net packages and so on.

#### The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

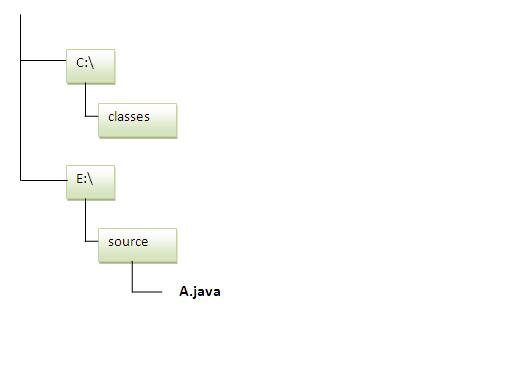
### Example of Subpackage

1. package com.javatpoint.core;
2. class Simple{
3. public static void main(String args[]){
4. System.out.println("Hello subpackage");
5. }
6. }

|  |
| --- |
| To Compile:javac -d . Simple.java |
| To Run: java com.javatpoint.core.Simple |

## How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:



//save as Simple.java

package mypack;

public class Simple{

 public static void main(String args[]){

    System.out.println("Welcome to package");

   }  }

### To Compile:

|  |
| --- |
| e:\sources>javac -d c:\classes Simple.java |

### To Run:

|  |
| --- |
| To run this program from e:\source directory, you need to set classpath of the directory where the class file resides. |
| e:\sources> set classpath=c:\classes;.; |
| e:\sources> java mypack.Simple |

### Another way to run this program by -classpath switch of java:

|  |
| --- |
| The -classpath switch can be used with javac and java tool. |
| To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example: |
| e:\sources> java -classpath c:\classes mypack.Simple |

### Ways to load the class files or jar files

|  |
| --- |
| There are two ways to load the class files temporary and permanent. |

* Temporary
  + By setting the classpath in the command prompt
  + By -classpath switch
* Permanent
  + By setting the classpath in the environment variables
  + By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

#### Rule: There can be only one public class in a java source file and it must be saved by the public class name.

//save as C.java otherwise Compilte Time Error

class A{}

class B{}

public class C{}

### How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

//save as A.java

package javatpoint;

public class A{}

//save as B.java

package javatpoint;

public class B{}