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.

If you overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

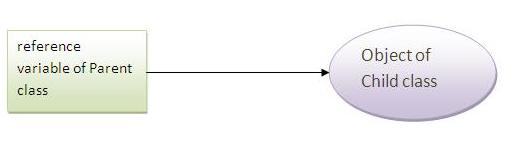
**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

**Upcasting**

When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



**class** A{}

**class** B **extends** A{}

A a=**new** B();//upcasting

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

**class** Bike{

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

}

**class** Splender **extends** Bike{

**void** run(){System.out.println("running safely with 60km");}

**public** **static** **void** main(String args[]){

    Bike b = **new** Splender();//upcasting

    b.run();

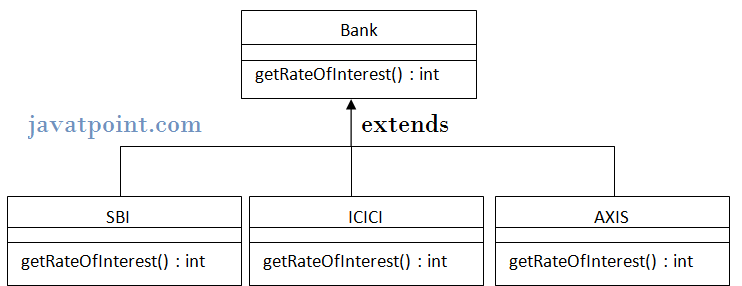
  }

}

Output:running safely with 60km.

Real example of Java Runtime Polymorphism

Consider a scenario, Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



Note: It is also given in method overriding but there was no upcasting.

**class** Bank{

**int** getRateOfInterest(){**return** 0;}

}

**class** SBI **extends** Bank{

**int** getRateOfInterest(){**return** 8;}

}

**class** ICICI **extends** Bank{

**int** getRateOfInterest(){**return** 7;}

}

**class** AXIS **extends** Bank{

**int** getRateOfInterest(){**return** 9;}

}

**class** Test3{

**public** **static** **void** main(String args[]){

Bank b1=**new** SBI();

Bank b2=**new** ICICI();

Bank b3=**new** AXIS();

System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest());

}

}

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

Java Runtime Polymorphism with data member

|  |
| --- |
| Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members. |
| In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always. |

***Rule: Runtime polymorphism can't be achieved by data members.***

**class** Bike{

**int** speedlimit=90;

}

**class** Honda3 **extends** Bike{

**int** speedlimit=150;

**public** **static** **void** main(String args[]){

  Bike obj=**new** Honda3();

  System.out.println(obj.speedlimit);//90

}

Output:90

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

**class** Animal{

**void** eat(){System.out.println("eating");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("eating fruits");}

}

**class** BabyDog **extends** Dog{

**void** eat(){System.out.println("drinking milk");}

**public** **static** **void** main(String args[]){

Animal a1,a2,a3;

a1=**new** Animal();

a2=**new** Dog();

a3=**new** BabyDog();

a1.eat();

a2.eat();

a3.eat();

}

}

Output: eating

eating fruits

drinking Milk

Try for Output

**class** Animal{

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("dog is eating...");}

}

**class** BabyDog1 **extends** Dog{

**public** **static** **void** main(String args[]){

Animal a=**new** BabyDog1();

a.eat();

}}

Output: Dog is eating

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. static binding (also known as early binding).
2. dynamic binding (also known as late binding).

Understanding Type

Let's understand the type of instance.

**1) variables have a type**

Each variable has a type, it may be primitive and non-primitive.

**int** data=30;

Here data variable is a type of int.

**2) References have a type**

**class** Dog{

**public** **static** **void** main(String args[]){

  Dog d1;//Here d1 is a type of Dog

 }

}

**3) Objects have a type**

|  |
| --- |
| An object is an instance of particular java class,but it is also an instance of its superclass. |

**class** Animal{}

**class** Dog **extends** Animal{

**public** **static** **void** main(String args[]){

  Dog d1=**new** Dog();

 }

}

|  |
| --- |
| Here d1 is an instance of Dog class, but it is also an instance of Animal. |

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

**class** Dog{

**private** **void** eat(){System.out.println("dog is eating...");}

**public** **static** **void** main(String args[]){

  Dog d1=**new** Dog();

  d1.eat();

 }

}

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

**class** Animal{

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("dog is eating...");}

**public** **static** **void** main(String args[]){

  Animal a=**new** Dog();

  a.eat();

 }

}

Output:dog is eating...

|  |
| --- |
| In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type. |

# Java instanceof

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

**class** Simple1{

**public** **static** **void** main(String args[]){

 Simple1 s=**new** Simple1();

 System.out.println(s **instanceof** Simple);//true

 }

}

Output:true

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

Another example of java instanceof operator

**class** Animal{}

**class** Dog1 **extends** Animal{//Dog inherits Animal

**public** **static** **void** main(String args[]){

 Dog1 d=**new** Dog1();

 System.out.println(d **instanceof** Animal);//true

 }

}

Output:true

instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

**class** Dog2{

**public** **static** **void** main(String args[]){

  Dog2 d=**null**;

  System.out.println(d **instanceof** Dog2);//false

 }

}

Output:false

Downcasting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

Dog d=**new** Animal();//Compilation error

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

Dog d=(Dog)**new** Animal();

//Compiles successfully but ClassCastException is thrown at runtime

Possibility of downcasting with instanceof

Let's see the example, where downcasting is possible by instanceof operator.

**class** Animal { }

**class** Dog3 **extends** Animal {

**static** **void** method(Animal a) {

**if**(a **instanceof** Dog3){

       Dog3 d=(Dog3)a;//downcasting

       System.out.println("ok downcasting performed");

    }

  }

**public** **static** **void** main (String [] args) {

    Animal a=**new** Dog3();

    Dog3.method(a);

  }

 }

Output:ok downcasting performed

Downcasting without the use of java instanceof

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

**class** Animal { }

**class** Dog4 **extends** Animal {

**static** **void** method(Animal a) {

       Dog4 d=(Dog4)a;//downcasting

       System.out.println("ok downcasting performed");

  }

**public** **static** **void** main (String [] args) {

    Animal a=**new** Dog4();

    Dog4.method(a);

  }

}

Output:ok downcasting performed

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

Animal a=**new** Animal();

Dog.method(a);

//Now ClassCastException but not in case of instanceof operator

Understanding Real use of instanceof in java

Let's see the real use of instanceof keyword by the example given below.

**interface** Printable{}

**class** A **implements** Printable{

**public** **void** a(){System.out.println("a method");}

}

**class** B **implements** Printable{

**public** **void** b(){System.out.println("b method");}

}

**class** Call{

**void** invoke(Printable p){//upcasting

**if**(p **instanceof** A){

A a=(A)p;//Downcasting

a.a();

}

**if**(p **instanceof** B){

B b=(B)p;//Downcasting

b.b();

}

}

}//end of Call class

**class** Test4{

**public** **static** **void** main(String args[]){

Printable p=**new** B();

Call c=**new** Call();

c.invoke(p);

}

}

Output: b method

# 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).

Before learning java abstract class, let's understand the abstraction in java first.

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 Abstaction**

There are two ways to achieve abstraction in java

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

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**

**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**

**abstract** **void** printStatus();//no body and abstract

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It 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();

}

}

running safely..

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.

*File: TestAbstraction1.java*

**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();

}

}

drawing circle

Another example of abstract class in java

*File: TestBank.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+" %");

}}

Rate of Interest is: 7 %

Abstract class having constructor, data member, methods etc.

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

*File: TestAbstraction2.java*

//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();

 }

}

bike is created

running safely..

gear changed

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

**class** Bike12{

**abstract** **void** run();

}

compile time error

***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();

}}

Output:I am a

I am b

I am c

I am d

# 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.

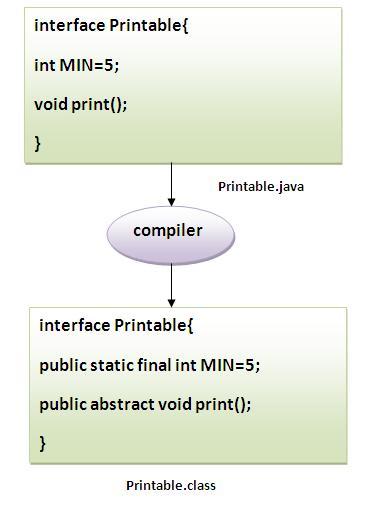
Why use Java interface?

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.

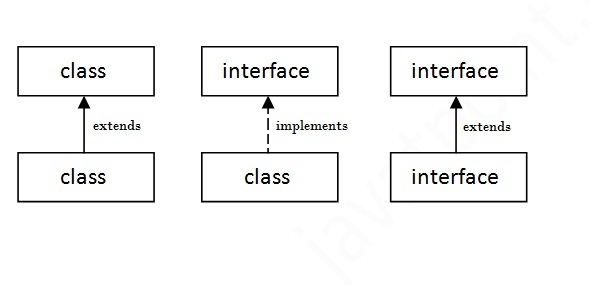
***The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.***

In other words, Interface fields are public, static and final bydefault, 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();

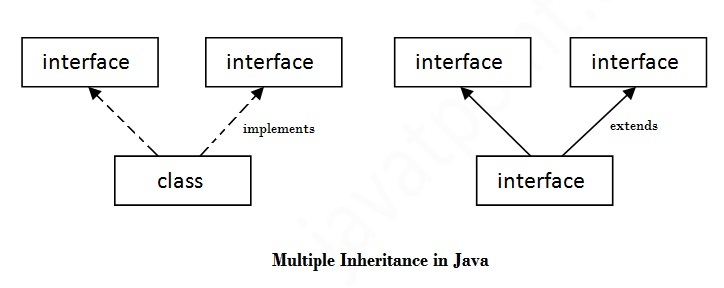
 }

}

Output:Hello

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();

 }

}

Output:Hello

Welcome

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();

 }

}

Hello

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();

 }

}

Hello

Welcome

Q) What is marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

//How Serializable interface is written?

**public** **interface** Serializable{

}

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 |
| 1) Abstract class can have abstract and non-abstractmethods. | Interface can have **only abstract** methods. |
| 2) Abstract class doesn't support multiple inheritance. | Interface **supports multiple inheritance**. |
| 3) Abstract class can have final, non-final, static and non-static variables. | Interface has **only static and final variables**. |
| 4) Abstract class can have static methods, main method and constructor. | Interface **can't have static methods, main method or constructor**. |
| 5) Abstract class can provide the implementation of interface. | Interface **can't provide the implementation of abstract class**. |
| 6) The abstract keyword is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 7) Example: public class Shape{ public abstract void draw(); } | **Example:** 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();

}}  Output:

I am a

I am b

I am c

I am d

# 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.

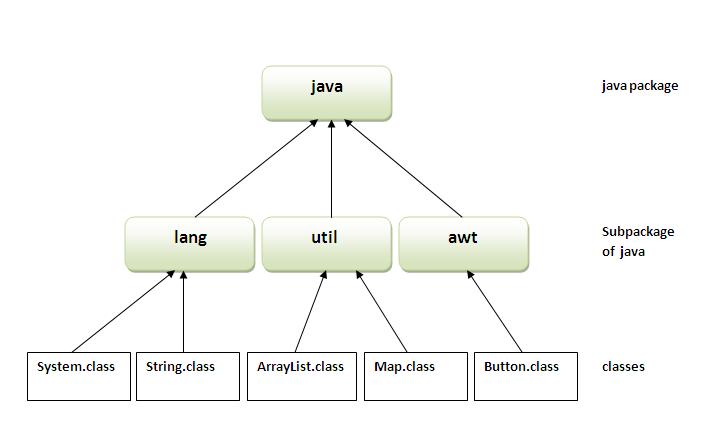
Here, we will have the detailed learning of creating and using user-defined packages.

**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:

javac -d directory javafilename

For **example**

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.Simple etc 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.   1. import package.\*; 2. import package.classname; 3. 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();

  }

}

Output:Hello

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();

  }

}

Output:Hello

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();

  }

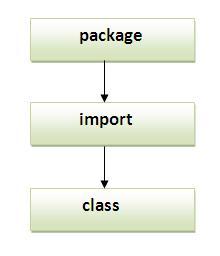
}

Output:Hello

***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 definded 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

**package** com.javatpoint.core;

**class** Simple{

**public** **static** **void** main(String args[]){

   System.out.println("Hello subpackage");

  }

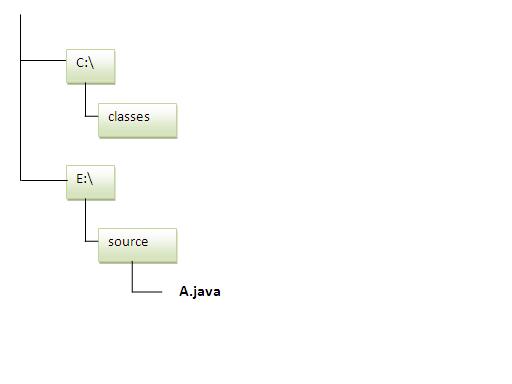
}

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

Output:Hello subpackage

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** |

Output:Welcome to package

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{}