***Inheritance:-***

1. *The process of acquiring properties (variables) & methods (behaviors) from one class to another class is called inheritance.*
2. *We are achieving inheritance concept by using* ***extends*** *keyword. Inheritance is also known as* ***is- a*** *relationship.*
3. *Extends keyword is providing relationship between two classes..*
4. *The main objective of inheritance is code extensibility whenever we are extending the class automatically code is reused.*
5. *In inheritance one class providing properties & another class is acquiring the properties.*
6. *In inheritance parent class is giving properties & Child is acquiring properties from Parent.*

***Notes:-***

1. *in java if we are extending the class then it will be parent class , if we are not extending the class then* ***Object*** *class will become parent class.*
2. *In above example A class is direct child class of object & B,C classes are indirect child classes of object.it represent in java every class is child of* ***Object*** *either directly(A) or indirectly(B,C).*
3. *The root class of all java classes is “****Object****” class.*
4. *Every java class contain parent class except* ***Object*** *class.*
5. *Object class present in* ***java.lang*** *package and it contains 11 methods & all java classes able to use these 11 methods because Object class is root class of all java classes*

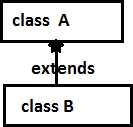
***C:\>javap java.lang.Object***

***Types of inheritance :-***

There are five types of inheritance in java,

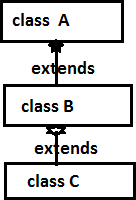
* 1. Single inheritance
  2. ***Multilevel inheritance***
  3. ***Hierarchical inheritance***
  4. ***Multiple inheritance***
  5. ***Hybrid Inheritance***

***Single inheritance:-***

* *One class has only one direct super class is called single inheritance.*
* *In the absence of any other explicit super class, every class is implicitly a subclass of* ***Object class.***

***Class B extends A ===>****class B acquiring properties of A class.*

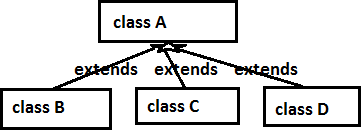
##### Multilevel inheritance:-

One Sub class is extending Parent class then that sub class will become Parent class of next extended class this flow is called multilevel inheritance.

Class B extends A ===> class B acquiring properties of A class Class C extends B ===> class C acquiring properties of B class

[indirectly class C using properties of A & B classes]

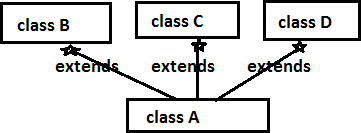
Hierarchical inheritance :-

More than one sub class is extending single Parent is called hierarchical inheritance.

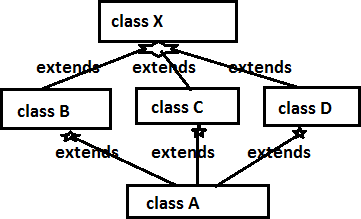
***Class B extends A*** *===> class B acquiring properties of A class* ***Class C extends A*** *===> class C acquiring properties of A class* ***Class D extends A*** *===> class D acquiring properties of A class*

Multiple inheritance:-

* *One sub class is extending more than one super class is called Multiple inheritance and java not supporting multiple inheritance because it is creating ambiguity problems (confusion state) .*
* *Java not supporting multiple inheritances hence in java one class able to extends only one class at a time but it is not possible to extends more than one class.*

Class A extends B ===>valid Class A extends B ,C ===>invalid

##### Hybrid inheritance:-

* *Hybrid is combination of hierarchical & multiple inheritance .*
* *Java is not supporting hybrid inheritance because multiple inheritance(not supported by java) is included in hybrid inheritance.*

***Super keyword:-***

“**this**” keyword is used to represent current class object& “**super**” keyword is used to represent super class object.

1. *Super class variables.*
2. *Super class methods.*
3. *Super class constructors.*
4. *Super class instance blocks.*
5. *Super class static blocks.*

**super class variables calling:-**

*class Parent*

*{*

*int a=10,b=20;*

*}*

*class Child extends Parent*

*{*

*int a=100; int b=200;*

*void m1(int a,int b) //local variables*

*{*

*System.out.println(a+b); //local variables addition*

*System.out.println(this.a+this.b); //current class variables addition*

*System.out.println(super.a+super.b); //super class variables addition*

*}*

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

*{*

*new Child().m1(1000,2000);*

*}*

*}*

***super class methods calling:-***

*class Parent*

*{*

*void m1()*

*{*

*System.out.println("parent m1() method");*

*}*

*}*

*class Child1 extends Parent*

*{ void m1()*

*{*

*System.out.println("child class m1() method");*

*}*

*void m3()*

*{*

*this.m1();*

*super.m1();*

*}*

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

*{*

*new Child1().m3();*

*}*

*}*

***Note:-*** *To call the super class methods super keyword is mandatory but to represent current class*

*methods this keyword is optional.*

***super class constructors calling:-***

*To call the current class constructors use this keyword but to call super class constructor use*

*super keyword.*

*super() ----> super class 0-arg constructor calling*

*super(10) ----> super class 1-arg constructor calling*

*super(10,20) ----> super class 2-arg constructor calling*

*super(10,'a',true) ----> super class 3-arg constructor calling*

*class Person*

*{*

*int id;*

*String name;*

*Person(int id,String name)*

*{*

*this.id=id;*

*this.name=name;*

*}*

*}*

*class Emp extends Person*

*{*

*float salary;*

*Emp(int id,String name,float salary)*

*{*

*super(id,name);//reusing parent constructor*

*this.salary=salary;*

*}*

*void display()*

*{*

*System.out.println(id+" "+name+" "+salary);}*

*}*

*class TestSuper5*

*{*

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

*{*

*Emp e1=new Emp(1,"ankit",45000f);*

*e1.display();*

*}*

*}*

***Important Notes:-***

* *Inside the constructor super keyword must be first statement otherwise compiler generates error message “call to super must be first line in constructor”.*
* *Inside the constructor it is possible to use either this keyword or super keyword but,*

*Two super keywords are not allowed.*

* *Two this keywords are not allowed.*
* *Both super & this keyword also not allowed.*

***Runtime polymorphism [Method Overriding]:-***

* *To achieve method overloading one java class sufficient but to achieve method overriding we required two java classes with parent and child relationship.*
* *The method implementations already present in parent class,*
  + - 1. *If the child class required that implementation then access those implementations.*
      2. *If the child class not required, parent class method implementations then override parent class method in child class to provide child specific implementations.*
* *The sub class overrides super class method to provide sub class method implementations.*
* *In overriding parent class method is called ===>* ***overridden method***

*Child class method is called ===>* ***overriding method***

**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

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

**class** Bike

{

**void** run()

{

System.out.println("running");

}

}

**class** Splendor **extends** Bike

{

**void** run()

{

System.out.println("running safely with 60km");

}

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

{

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

    b.run();

   }

}

**Example-2 :-**

**class** Bank

{

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

}

**class** SBI **extends** Bank{

**float** getRateOfInterest(){**return** 8.4f;}

}

**class** ICICI **extends** Bank{

**float** getRateOfInterest(){**return** 7.3f;}

}

**class** AXIS **extends** Bank{

**float** getRateOfInterest(){**return** 9.7f;}

}

**class** TestPolymorphism{

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

Bank b;

b=**new** SBI();

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

b=**new** ICICI();

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

b=**new** AXIS();

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

}

}

# 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).

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

1. **int** data=30;

#### References have a type

**class** Dog{

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

  Dog d1;//Here d1 is a type of Dog

 }

}

#### Objects have a type

|  |
| --- |
| An object is an instance of particular java class, but it is also an instance of its superclass.   1. **class** Animal 2. { 3. } 4. **class** Dog **extends** Animal 5. { 6. **public** **static** **void** main(String args[]) 7. { 8. Dog d1=**new** Dog(); 9. } 10. }   Notes:- 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. Dynamic binding When type of the object is determined at run-time, it is known as dynamic binding.   1. **class** Animal 2. { 3. **void** eat() 4. { 5. System.out.println("animal is eating..."); 6. } 7. } 9. **class** Dog **extends** Animal 10. { 11. **void** eat() 12. { 13. System.out.println("dog is eating..."); 14. } 16. **public** **static** **void** main(String args[]) 17. { 18. Animal a=**new** Dog(); 19. a.eat(); 20. } 21. }   **Note:-** 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. Final Keyword In Java :-  1. variable 2. method 3. class   The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. Java final variable If you make any variable as final, you cannot change the value of final variable(It will be constant).   1. **class** Bike9 2. { 3. **final** **int** speedlimit=90;//final variable 4. **void** run() 5. { 6. speedlimit=400; 7. } 8. **public** **static** **void** main(String args[]) 9. { 10. Bike9 obj=**new**  Bike9(); 11. obj.run(); 12. } 13. }//end of class   Output:Compile Time Error 2) Java final method If you make any method as final, you cannot override it.   1. **class** Bike 2. { 3. **final** **void** run() 4. { 5. System.out.println("running"); 6. } 7. } 9. **class** Honda **extends** Bike 10. { 11. **void** run() 12. { 13. System.out.println("running safely with 100kmph"); 14. } 16. **public** **static** **void** main(String args[]) 17. { 18. Honda honda= **new** Honda(); 19. honda.run(); 20. } 21. }  Java final class If you make any class as final, you cannot extend it.   1. **final** **class** Bike 2. { 3. } 4. **class** Honda1 **extends** Bike 5. { 6. **void** run() 7. { 8. System.out.println("running safely with 100kmph"); 9. } 11. **public** **static** **void** main(String args[]) 12. { 13. Honda1 honda= **new** Honda1(); 14. honda.run(); 15. } 16. }  Abstract class in Java A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated. Points to Remember  * An abstract class must be declared with an abstract keyword. * It can have abstract and non-abstract methods. * It cannot be instantiated. * It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also. * It can have final methods which will force the subclass not to change the body of the method.  Abstract Method in Java A method which is declared as abstract and does not have implementation is known as an abstract method.   1. **abstract** **void** printStatus();//no method body and abstract 2. **abstract** **class** Shape 3. { 4. **abstract** **void** draw(); 5. } 6. //In real scenario, implementation is provided by others i.e. unknown by end user 7. **class** Rectangle **extends** Shape 8. { 9. **void** draw() 10. { 11. System.out.println("drawing rectangle"); 12. } 13. } 14. **class** Circle1 **extends** Shape 15. { 16. **void** draw() 17. { 18. System.out.println("drawing circle"); 19. } 20. } 21. //In real scenario, method is called by programmer or user 22. **class** TestAbstraction1 23. { 24. **public** **static** **void** main(String args[]) 25. { 26. Shape s=**new** Circle1();//In a real scenario, object is provided through method, e.g., getShape() method 27. s.draw(); 28. } 29. }  Rule: If there is an abstract method in a class, that class must be abstract. |
|  |

***Interfaces***

1. *Interface is also one of the type of class it contains only abstract methods. And Interfaces not alternative for abstract class it is* ***extension*** *for abstract classes.*
2. *The interface allows to declare* ***only abstract methods*** *and these methods are by default public & abstract if we are declaring or not***.**
3. *The interface is highlighting set of functionalities but implementations are hiding.*
4. *For the interfaces also compiler will generates .class files after compilation.*
5. *Inside the source file it is possible to declare any number of interfaces. And we are declaring the interfaces by using* ***interface*** *keyword.*

**Syntax:- i*nterface interface-name***

interface It1

{

\*\*\*\*\*

}

* **if you don’t no the anything about implementation just we have the requirement specification then declare that requirements by using interface.**
* **If u know the implementation but not completely then we should go for abstract classes.**
* **if you know the implementation completely then we should go for concrete classes.**

Both examples are same

interface it1 abstract interface it1

{ {

void m1(); public abstract void m1();

void m2(); public abstract void m2();

void m3(); public abstract void m3();

} }

**Note: - If we are declaring or not each and every interface method by default public abstract. And the interfaces are by default abstract hence for the interfaces object creation is not possible.**

***Example-1 :-***

* *Interface contains abstract methods and by default these methods are “public abstract “.*
* *Interface contains abstract method for these methods provide the implementation in the implementation classes.*
* *Implementation class is nothing but the class which implements particular interface.*
* *While providing implementation of interface methods that implementation methods must be public methods otherwise compiler generate error message “****attempting to assign weaker access privileges”.***

interface it1

*{ Void m1(); /****/abstract method by default [public abstract]*** *Void m2(); /****/abstract method by default [public abstract]*** *Void m3(); /****/abstract method by default [public abstract]***

}

*Class Test implements it1* ***//Test is implementation class of It1 interface***

*{ Public void m1()* ***//implementation method must be public***

{

System.out.println(“m1-method implementation ”);

}

Public void m2()

{

System.out.println(“m2-method implementation”);

}

Public void m3()

{

System.out.println(“m3 –method implementation”);

}

Public static void main(String[] args)

{

Test t=new Test();

t.m1();

t.m2();

t.m3();

}

}

Example :-

*interface It1* ***//abstract***

*{ void m1();* ***//public abstract***

void m2();

void m3();

}

class Test implements It1

{

Publicvoidm1()

{

System.out.println("m1 method");

}

public void m2()

{

System.out.println("m2 method");

}

public void m3()

{

System.out.println("m3 method");

}

public static void main(String[] args)

{ Test t = new Test();

t.m1();

t.m2();

t.m3();

It1 i = new Test();

*i.m1();* ***//compile: It1 runtimeTest*** *i.m2();* ***//compile:It1 runtime test*** *i.m3();* ***//compile: It1 runtimeTest***

}

};

Example-2:-

* *Interface contains abstract method for these methods provide the implementation in the implementation class.*
* *If the implementation class is unable to provide the implementation of all abstract methods then declare implementation class with abstract modifier & complete the remaining abstract method implementation in next created child classes.*
* *If the child class also unable to provide implementation then declare the child class with abstract modifier & take one more child class to complete the implementations.*
* *In java it is possible to take any number of child classes but at final complete the implementation of all abstract methods.*

interface It1

*{ void m1();* ***//public abstract***

void m2();

void m3();

}

abstract class Test implements It1

{ public void m1()

{

System.out.println("m1 method");

}

}

abstract class Test1 extends Test

{ public void m2()

{

System.out.println("m2 method");

}

}

class Test2 extends Test1

{ public void m3()

{

System.out.println("m3 method");

}

public static void main(String[] args)

{

Test2 t = new Test2(); t.m1();

t.m2();

t.m3();

}

}

***Example:- The interface reference variables is able to hold child class objects.***

*interface It1* ***// interface declaration***

##### **{ void m1();** //abstract method by default [public abstract] **void m2();** //abstract method by default [public abstract] **void m3();** //abstract method by default [public abstract]

}

//Test1 is abstract class contains 2 abstract methods m2() m3()hence object creation not possible

*abstract class Test1 implements It1*

*{*

*public void m1()*

*{*

*System.out.println("m1 method");*

*}*

*}*

***//Test2 is abstract class contains 1 abstract method m3() hence object creation not possible***

*abstract class Test2 extends Test1*

*{*

*public void m2()*

*{*

*System.out.println("m2 method");*

*}*

*}*

***//Test3 is normal class because it contains only normal methods hence object creation possible***

*class Test3 extends Test2*

*{ public void m3()*

*{*

*System.out.println("m3 method");*

*}*

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

*{*

*It1 t = new Test3();*

*t.m1();*

*t.m2();*

*t.m3();*

*Test1 t1 = new Test3();*

*t1.m1();*

*t1.m2();*

*t1.m3();*

*Test2 t2 = new Test3();*

*t2.m1();*

*t2.m2();*

*t2.m3();*

*}*

*}*

## Java Interface Example: Bank

**interface** Bank

{

**float** rateOfInterest();

}

**class** SBI **implements** Bank

{

**public** **float** rateOfInterest()

{

**return** 9.15f;

}

}

**class** PNB **implements** Bank

{

**public** **float** rateOfInterest()

{

**return** 9.7f;

}

}

**class** TestInterface2

{

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

{

Bank b=**new** SBI();

System.out.println("ROI: "+b.rateOfInterest());

}

}

|  |  |
| --- | --- |
| **Difference between abstract classes & interfaces** | |
| **Abstract class** | ***Interface*** |
| The purpose of abstract class is to specify default functionality of an object and lest its sub classes explicitly implement that functionality. It stands it is providing abstraction layer that must be extended and implemented by the corresponding sub classes | It is providing complete abstraction layer and it contains only declarations of the project then write the implementations in implementation classes. |
| An abstract class is a class that declared with **abstract** modifier.  Ex: ***abstract class A***  ***{ abstract void m1(); }*** | Declare the interface by using  **interface** keyword. Ex:- ***interface It1*** { void m1(); } |
| The abstract allows declaring both abstract & concrete methods | The interface allows declaring only abstract methods. |
| Abstract class methods must declare with abstract modifier. | Interface methods are by default  **public abstract** |
| If the abstract class contains abstract methods then write the implementations in child classes. | The interface contains abstract methods writ the implementations in implementation classes. |
| In child class the implementation methods need not be public it means while overriding it is possible to declare any valid modifier. | In implementation class the implementation methods must be public. |
| The abstract class is able to provide implementations of interface methods. | The interface is unable to provide implementation of abstract class methods. |
| One java class is able to extends only one abstract class at a time. | One java class is able to implements multiple interfaces at a time. |
| Inside abstract class it is possible to declare main method &constructors. | Inside interface it is not possible to declare methods and constructors. |
| It is not possible to instantiate abstract class. | It is not possible to instantiate interfaces. |
| For the abstract classes compiler will generate **.class** files. | For the interfaces compiler will generate .class files. |
| The variables of abstract class need not be **public static final.** | The variables declared in interface by default **public static final.** |

# 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.

## 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.

# Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.