**Programme: Computer Engineering Semester: III**

**Course: Programming in JAVA Course code: PRJ228914**

# **Credits: 1.**

# **<https://www.javatpoint.com/inheritance-in-java>**

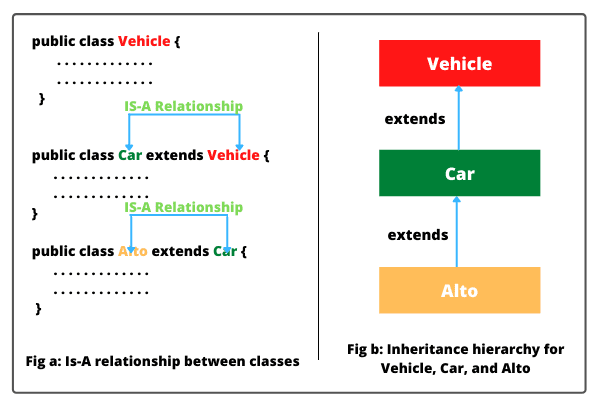
**2.** [**https://www.geeksforgeeks.org/packages-in-java/**](https://www.geeksforgeeks.org/packages-in-java/)

**JAVA INHERITANCE**

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of [OOPs](https://www.javatpoint.com/java-oops-concepts) (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java) that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.



### **Why use inheritance in java**

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.

### **Why Java Inheritance?**

There are several reasons why inheritance was introduced into Object-oriented languages. We will discuss some major reasons behind the introduction of inheritance.

* The capability to express inheritance relationships ensures the closeness with the **real-world models.**
* Another reason is the idea of **reusability.** One can derive a new class (sub-class) from an existing class and add new features to it without modifying its parent class. There is no need to rewrite the parent class in order to inherit it.
* One reason is the **transitive nature.** If class **A** inherits properties from another class **B**, then all subclasses of **A** will automatically inherit properties from **B**. This property is called the transitive nature of inheritance.

**Note:***A subclass defines only those features that are unique to it.*

For example, the class**Student** inherits from the class **Person**. Then although Student is a person, the reverse is not true. A Person need not be a Student. The class Student has properties that it does not share with class Person.

For instance, the **Student** has a marks-percentage, but the **Person** does not have.

### **Terms used in Inheritance**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

### **The syntax of Java Inheritance**

class Subclass-name extends Superclass-name

{

//methods and fields

}

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

class Employee

{

float salary=40000;

}

class Programmer extends Employee

{

int bonus=10000;

public static void main(String args[])

{

Programmer p=new Programmer();

System.out.println("Programmer salary is:"+p.salary);

System.out.println("Bonus of Programmer is:"+p.bonus);

}

}

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

## Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



#### **Note: Multiple inheritance is not supported in Java through class.**

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



## Single Inheritance Example

When a class inherits another class, it is known as a ***single inheritance*.** In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

class Animal

{

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

}

class Dog extends Animal

{

void bark(){System.out.println("barking...");}

}

class TestInheritance

{

public static void main(String args[])

{

Dog d=new Dog();

d.bark();

d.eat();

}

}

## Multilevel Inheritance Example

When there is a chain of inheritance, it is known as ***multilevel inheritance*.** As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

class Animal

{

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

}

class Dog extends Animal

{

void bark(){System.out.println("barking...");}

}

class BabyDog extends Dog

{

void weep(){System.out.println("weeping...");}

}

class TestInheritance2

{

public static void main(String args[])

{

BabyDog d=new BabyDog();

d.weep();

d.bark();

d.eat();

}}

## Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as ***hierarchical inheritance***. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

class Animal

{

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

}

class Dog extends Animal

{

void bark(){System.out.println("barking...");

}

}

class Cat extends Animal

{

void meow(){System.out.println("meowing...");}

}

class TestInheritance3

{

public static void main(String args[])

{

Cat c=new Cat();

c.meow();

c.eat();

//c.bark();//C.T.Error

}

}

## Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

class A

{

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

}

class B

{

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

}

class C extends A,B

{ //suppose if it were

public static void main(String args[]){

C obj=new C();

obj.msg();//Now which msg() method would be invoked?

}

}

**Questions**

1. Write a Java program to create a class called Shape with a method called getArea(). Create a subclass called Rectangle that overrides the getArea() method to calculate the area of a rectangle.
2. Write a Java program to create a class known as "BankAccount" with methods called deposit() and withdraw(). Create a subclass called SavingsAccount that overrides the withdraw() method to prevent withdrawals if the account balance falls below one hundred.
3. Write a Java program to create a class known as Person with methods called getFirstName() and getLastName(). Create a subclass called Employee that adds a new method named getEmployeeId() and overrides the getLastName() method to include the employee's job title
4. Write a Java program to create a vehicle class hierarchy. The base class should be Vehicle, with subclasses Truck, Car and Motorcycle. Each subclass should have properties such as make, model, year, and fuel type. Implement methods for calculating fuel efficiency, distance traveled, and maximum speed.
5. Write a Java program to create a class called Shape with methods called getPerimeter() and getArea(). Create a subclass called Circle that overrides the getPerimeter() and getArea() methods to calculate the area and perimeter of a circle.

# **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 a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

### **Usage of Java Method Overriding**

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

#### **Rules for Java Method Overriding**



### **Understanding the problem without method overriding**

Let's understand the problem that we may face in the program if we don't use method overriding.

class Vehicle

{

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

}

//Creating a child class

class Bike extends Vehicle{

public static void main(String args[])

{

//creating an instance of child class

Bike obj = new Bike();

//calling the method with child class instance

obj.run();

}

}

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

### **Example of method overriding**

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

class Vehicle

{

**//defining a method**

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

}

**//Creating a child class**

class Bike2 extends Vehicle

{

//defining the same method as in the parent class

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

public static void main(String args[])

{

Bike2 obj = new Bike2();//creating object

obj.run();//calling method

}

}

### **A real example of Java Method Overriding**

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



#### **Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.**

class Bank

{

int getRateOfInterest(){return 0;}

}

//Creating child classes.

class SBI extends Bank

{

int getRateOfInterest(){return 8;}

}

class ICICI extends Bank

{

int getRateOfInterest(){return 7;}

}

class AXIS extends Bank

{

int getRateOfInterest(){return 9;}

}

//Test class to create objects and call the methods

class Test2{

public static void main(String args[])

{

SBI s=new SBI();

ICICI i=new ICICI();

AXIS a=new AXIS();

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

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

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

}

}

### **Can we override static method?**

No, a static method cannot be overridden. It can be proved by runtime polymorphism

### **Why can we not override static method?**

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

**Can we override java main method?**

No, because the main is a static method.

# **Super Keyword in Java**

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

## Usage of Java super Keyword

## Usage of Java Super keyword

## 1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

class Animal

{

String color="white";

}

class Dog extends Animal

{

String color="black";

void printColor(){

System.out.println(color);//prints color of Dog class

System.out.println(super.color);//prints color of Animal class

}

}

class TestSuper1

{

public static void main(String args[]){

Dog d=new Dog();

d.printColor();

}

}

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

## 2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

class Animal{

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

}

class Dog extends Animal

{

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

void bark(){System.out.println("barking...");}

void work(){

super.eat();

bark();

}

}

class TestSuper2

{

public static void main(String args[]){

Dog d=new Dog();

d.work();

}

}

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

**3) super is used to invoke parent class constructor.**

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

class Animal

{

Animal(){System.out.println("animal is created");}

}

class Dog extends Animal

{

Dog(){

super();

System.out.println("dog is created");

}

}

class TestSuper3

{

public static void main(String args[]){

Dog d=new Dog();

}

}

#### **Note: super() is added in each class constructor automatically by compiler if there is no super() or this().**



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

## super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

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();

}

}

# **Final Keyword in Java**

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many contexts. Final can be:

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. We will have detailed learning of these. Let's first learn the basics of final keyword.



## 1) Java final variable

If you make any variable as final, you cannot change the value of final variable (It will be constant).

### **Example of final variable**

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

class Bike9

{

final int speedlimit=90;//final variable

void run(){

speedlimit=400;

}

public static void main(String args[]){

Bike9 obj=new Bike9();

obj.run();

}

}//end of class

**Output: Compile Time Error**

## 2) Java final method

If you make any method as final, you cannot override it.

### **Example of final method**

class Bike

{

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

}

class Honda extends Bike

{

void run(){System.out.println("running safely with 100kmph");}

public static void main(String args[]){

Honda honda= new Honda();

honda.run();

}

}

**Output: Compile Time Error**

## 3) Java final class

If you make any class as final, you cannot extend it.

### **Example of final class**

final class Bike

{

}

class Honda1 extends Bike

{

void run(){System.out.println("running safely with 100kmph");}

public static void main(String args[]){

Honda1 honda= new Honda1();

honda.run();

}

}

**Output: Compile Time Error**

### **Q) What is blank or uninitialized final variable?**

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

### **Q) Can we initialize blank final variable?**

Yes, but only in constructor.

### **Q) static blank final variable**

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

### **Q) What is final parameter?**

If you declare any parameter as final, you cannot change the value of it.

class Bike11

{

int cube(final int n)

{

n=n+2;//can't be changed as n is final

n\*n\*n;

}

public static void main(String args[])

{

Bike11 b=new Bike11();

b.cube(5);

}

}

### **Q) Can we declare a constructor final?**

No, because constructor is never inherited.

### **Q) Is final method inherited?**

Ans) Yes, final method is inherited but you cannot override it.

# **Polymorphism in Java**

**Polymorphism in Java** is a concept by which we can perform a *single action in 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 a 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**

If the 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

For upcasting, we can use the reference variable of class type or an interface type. For Example:

interface I{}

class A{}

class B extends A implements I{}

### **Example of Java Runtime Polymorphism**

In this example, we are creating two classes Bike and Splendor. Splendor 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, the 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 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();

}

}

## Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



#### **Note: This example is also given in method overriding but there was no upcasting.**

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());

}

}

## Java Runtime Polymorphism with Data Member

A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

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

}

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

## 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();

}

}

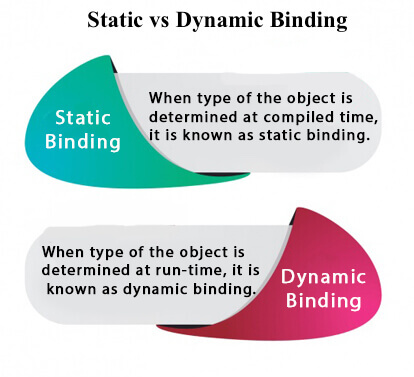
# **Static Binding and Dynamic Binding**



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

There are two types of binding

* Static Binding (also known as Early Binding)
* 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();

}

}

# 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.

# **Abstract class in Java**

A class which is declared with the abstract keyword is known as an abstract class in [Java](https://www.javatpoint.com/java-tutorial). It can have abstract and non-abstract methods (method with the body).

Before learning the 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 essential things to the user and hides the internal details, for example, sending SMS where you 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](https://www.javatpoint.com/object-and-class-in-java) 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%)

### **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**



**Example of abstract class**

1. **abstract** **class** A{}

### **Abstract Method in Java**

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

**Example of abstract method**

1. **abstract** **void** printStatus(); //no method body and abstract

### **Example of Abstract class that has an abstract method**

In this example, Bike is an 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, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a 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 a 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 8;}

}

class TestBank

{

public static void main(String args[]){

Bank b;

b=new SBI();

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

b=new PNB();

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

}

}

### **Abstract class having constructor, data member and methods**

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

abstract class Bike

{

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

abstract void run();

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

}

//Creating a Child class which inherits Abstract class

class Honda extends Bike

{

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

}

//Creating a Test class which calls abstract and non-abstract methods

class TestAbstraction2

{

public static void main(String args[]){

Bike obj = new Honda();

obj.run();

obj.changeGear();

}

}

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

class Bike12

{

abstract void run();

}

**OUTPUT: compile time error**

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

# **Interface in Java**

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve*[abstraction](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

## Why use Java interface?



## How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

interface <interface\_name>

{

// declare constant fields

// declare methods that abstract

// by default.

}

## Internal addition by the compiler

#### **The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.**

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



#### **The 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**.



## Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 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();

}

}

## Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

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());

}

}

## Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is 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 an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of [class](https://www.javatpoint.com/object-and-class-in-java) because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. **For example:**

interface Printable

{

void print();

}

interface Showable

{

void print();

}

class TestInterface3 implements Printable, Showable

{

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

public static void main(String args[]){

TestInterface3 obj = new TestInterface3();

obj.print();

}

}

**Output:**

**Hello**

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

## Interface inheritance

A class implements an interface, but one interface extends another interface.

interface Printable

{

void print();

}

interface Showable extends Printable

{

void show();

}

class TestInterface4 implements Showable

{

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

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

public static void main(String args[]){

TestInterface4 obj = new TestInterface4();

obj.print();

obj.show();

}

}

## Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

interface Drawable

{

void draw();

default void msg(){System.out.println("default method");}

}

class Rectangle implements Drawable

{

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

}

class TestInterfaceDefault

{

public static void main(String args[]){

Drawable d=new Rectangle();

d.draw();

d.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-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 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 provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

### **Example of abstract class and interface in Java**

### simple example where we are using interface and abstract class both.

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();

}

}

# **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.

Package names and directory structure are closely related.

For example if a package name is college.staff.cse, then there are three directories, college, staff and cse such that cse is present in staff and staff is present inside college.

**Adding a class to a Package :** We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it.

**Subpackages:** Packages that are inside another package are the **subpackages**. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

**Example :**

import java.util.\*;

**util** is a subpackage created inside **java** package.

**Accessing classes inside a package**

Consider following two statements:

**// import the Vector class from util package.**

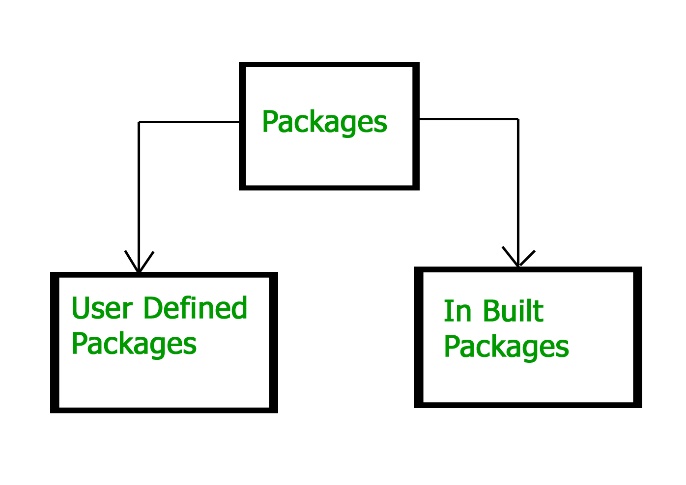
**import java.util.vector;**

**// import all the classes from util package**

**import java.util.\*;**

* First Statement is used to import **Vector** class from **util** package which is contained inside **java**.
* Second statement imports all the classes from **util** package.

**Types of packages:**



**Built-in Packages**  
These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:  
1) **java.lang:**Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.  
2) **java.io:**Contains classed for supporting input / output operations.  
3) **java.util:**Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.  
4) **java.applet:**Contains classes for creating Applets.  
5) **java.awt:**Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).  
6) **java.net:**Contain classes for supporting networking operations.

**User-defined packages**  
These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

// Name of the package must be same as the directory

// under which this file is saved

package myPackage;

public class MyClass

{

public void getNames(String s)

{

System.out.println(s);

}

}

Now we can use the **MyClass** class in our program.

/\* import 'MyClass' class from 'names' myPackage \*/

import myPackage.MyClass;

public class PrintName

{

public static void main(String args[])

{

String name = "GeeksforGeeks";

// Creating an instance of class MyClass in the package.

MyClass obj = new MyClass();

obj.getNames(name);

}

}

**Note :** **MyClass.java** must be saved inside the **myPackage** directory since it is a part of the package.

**Using Static Import**

Static import allows members ( fields and methods ) defined in a class as public **static** to be used in Java code without specifying the class in which the field is defined.  
Following program demonstrates **static import** :

// Note static keyword after import.

import static java.lang.System.\*;

class StaticImportDemo

{

public static void main(String args[])

{

// We don't need to use 'System.out' as imported using static.

out.println("GeeksforGeeks");

}

}

**Handling name conflicts**

The only time we need to pay attention to packages is when we have a name conflict . For example both, java.util and java.sql packages have a class named Date. So if we import both packages in program as follows:

import java.util.\*;

import java.sql.\*;

//And then use Date class, then we will get a compile-time error :

Date today ; //ERROR-- java.util.Date or java.sql.Date?

The compiler will not be able to figure out which Date class do we want. This problem can be solved by using a specific import statement:

import java.util.Date;

import java.sql.\*;

If we need both Date classes then, we need to use a full package name every time we declare a new object of that class.  
For Example:

java.util.Date deadLine = new java.util.Date();

java.sql.Date today = new java.sql.Date();

# **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.

### **Understanding Java Access Modifiers**

Let's understand the access modifiers in Java by a simple table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

### **1) Private**

The private access modifier is accessible only within the class.

**Simple example of private access modifier**

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

class A

{

private int data=40;

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

}

public class Simple

{

public static void main(String args[]){

A obj=new A();

System.out.println(obj.data);//Compile Time Error

obj.msg();//Compile Time Error

}

}

### **Role of Private Constructor**

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

class A

{

private A(){}//private constructor

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

}

public class Simple{

public static void main(String args[]){

A obj=new A();//Compile Time Error

}

}

#### **Note: A class cannot be private or protected except nested class.**

**2) Default**

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

**Example of default access modifier**

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

**//save by A.java**

package pack;

class A{

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();//Compile Time Error

obj.msg();//Compile Time Error

}

}

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

### **3) Protected**

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

**Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

**//save by A.java**

package pack;

public class A{

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

}

**//save by B.java**

package mypack;

import pack.\*;

class B extends A{

public static void main(String args[]){

B obj = new B();

obj.msg();

}

}

### **4) Public**

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

**Example of public access modifier**

**//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();

}

}

### **Java Access Modifiers with Method Overriding**

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

class A

{

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

}

public class Simple extends A

{

void msg(){System.out.println("Hello java");}//C.T.Error

public static void main(String args[]){

Simple obj=new Simple();

obj.msg();

}

}

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

# **Encapsulation in Java**

**Encapsulation in Java** is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.



We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

### **Advantage of Encapsulation in Java**

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

### **Simple Example of Encapsulation in Java**

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

**//A Java class which is a fully encapsulated class.**

**//It has a private data member and getter and setter methods.**

public class Student

{

private String name;

public String getName()

{

return name;

}

//setter method for name

public void setName(String name)

{

this.name=name

}

}

# **CLASSPATH in Java**

Package in Java is a mechanism to encapsulate a group of classes, sub-packages, and interfaces. Packages are used for:

* Preventing naming conflicts. For example, there can be two classes with the name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
* Making searching/locating and usage of classes, interfaces, enumerations, and annotations easier
* Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.

Packages can be considered as data encapsulation (or data-hiding). Here we will be discussing the responsibility of the CLASSPATH environment variable while programming in Java as we move forward we for sure short need usage of importing statements.

**Illustration:**

import org.company.Menu

What does this import mean? It makes the Menu class available in the package org.company to our current class. Such that when we call the below command as follows:

Menu menu = new Menu();

// java.io (input-output package)

import java.io.\*;

class GFG {

public static void main(String[] args)

{

System.out.println("I/O classes are imported from java.io package");

}

}

**Difference between PATH and CLASSPATH**

|  |  |
| --- | --- |
| **PATH** | **CLASSPATH** |
| PATH is an environment variable. | CLASSPATH is also an environment variable. |
| It is used by the operating system to find the executable files (.exe). | It is used by Application ClassLoader to locate the .class file. |
| You are required to include the directory which contains .exe files. | You are required to include all the directories which contain .class and JAR files. |
| PATH environment variable once set, cannot be overridden. | The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command. |