Objects and Classes in Java

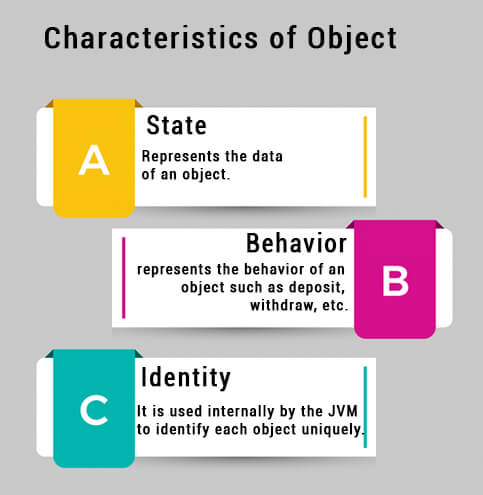
An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.



An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

* State: represents the data (value) of an object.
* Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.
* Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.



For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance (result) of a class.

Object Definitions:

* An object is *a real-world entity*.
* An object is *a runtime entity*.
* The object is *an entity which has state and behavior*.
* The object is *an instance of a class*.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

* Fields
* Methods
* Constructors
* Blocks
* Nested class and interface

Syntax to declare a class:

class <class\_name>{

    field;

    method;

}

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

* Code Reusability
* Code Optimization

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

*File: Student.java*

//Java Program to illustrate how to define a class and fields

//Defining a Student class.

class Student{

 //defining fields

 int id;//field or data member or instance variable

 String name;

 //creating main method inside the Student class

 public static void main(String args[]){

  //Creating an object or instance

  Student s1=new Student();//creating an object of Student

  //Printing values of the object

  System.out.println(s1.id);//accessing member through reference variable

  System.out.println(s1.name);

 }

}

Output:

0

null

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

*File: TestStudent1.java*

//Java Program to demonstrate having the main method in

//another class

//Creating Student class.

class Student{

 int id;

 String name;

}

//Creating another class TestStudent1 which contains the main method

class TestStudent1{

 public static void main(String args[]){

  Student s1=new Student();

  System.out.println(s1.id);

  System.out.println(s1.name);

 }

}

Output:

0

null

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

*File: TestStudent2.java*

class Student{

 int id;

 String name;

}

class TestStudent2{

 public static void main(String args[]){

  Student s1=new Student();

  s1.id=101;

  s1.name="Sonoo";

  System.out.println(s1.id+" "+s1.name);//printing members with a white space   }

}

Output:

101 Sonoo

We can also create multiple objects and store information in it through reference variable.

*File: TestStudent3.java*

class Student{

 int id;

 String name;

}

class TestStudent3{

 public static void main(String args[]){

  //Creating objects

  Student s1=new Student();

  Student s2=new Student();

  //Initializing objects

  s1.id=101;

  s1.name="Sonoo";

  s2.id=102;

  s2.name="Amit";

  //Printing data

  System.out.println(s1.id+" "+s1.name);

  System.out.println(s2.id+" "+s2.name);

 }

}

Output:

101 Sonoo

102 Amit

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

*File: TestStudent4.java*

class Student{

 int rollno;

 String name;

 void insertRecord(int r, String n){

  rollno=r;

  name=n;

 }

 void displayInformation(){System.out.println(rollno+" "+name);}

}

class TestStudent4{

 public static void main(String args[]){

  Student s1=new Student();

  Student s2=new Student();

  s1.insertRecord(111,"Karan");

  s2.insertRecord(222,"Aryan");

  s1.displayInformation();

  s2.displayInformation();

 }

}

Output:

111 Karan

222 Aryan



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through a constructor

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

*File: TestEmployee.java*

class Employee{

    int id;

    String name;

    float salary;

    void insert(int i, String n, float s) {

        id=i;

        name=n;

        salary=s;

    }

    void display(){System.out.println(id+" "+name+" "+salary);}

}

public class TestEmployee {

public static void main(String[] args) {

    Employee e1=new Employee();

    Employee e2=new Employee();

    Employee e3=new Employee();

    e1.insert(101,"ajeet",45000);

    e2.insert(102,"irfan",25000);

    e3.insert(103,"nakul",55000);

    e1.display();

    e2.display();

    e3.display();

}

}

Output:

101 ajeet 45000.0

102 irfan 25000.0

103 nakul 55000.0

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

*File: TestRectangle1.java*

class Rectangle{

 int length;

 int width;

 void insert(int l, int w){

  length=l;

  width=w;

 }

 void calculateArea(){System.out.println(length\*width);}

}

class TestRectangle1{

 public static void main(String args[]){

  Rectangle r1=new Rectangle();

  Rectangle r2=new Rectangle();

  r1.insert(11,5);

  r2.insert(3,15);

  r1.calculateArea();

  r2.calculateArea();

}

}

Output:

55

45

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

* By new keyword
* By newInstance() method
* By clone() method
* By deserialization
* By factory method etc.

We will learn these ways to create object later.

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

* new Calculation();//anonymous object

Calling method through a reference:

* Calculation c=new Calculation();
* c.fact(5);

Calling method through an anonymous object

* new Calculation().fact(5);

Let's see the full example of an anonymous object in Java.

class Calculation{

 void fact(int  n){

  int fact=1;

  for(int i=1;i<=n;i++){

   fact=fact\*i;

  }

 System.out.println("factorial is "+fact);

}

public static void main(String args[]){

 new Calculation().fact(5);//calling method with anonymous object

}

}

Output:

Factorial is 120

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

1. int a=10, b=20;

Initialization of refernce variables:

1. Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects

Let's see the example:

//Java Program to illustrate the use of Rectangle class which

//has length and width data members

class Rectangle{

 int length;

 int width;

 void insert(int l,int w){

  length=l;

  width=w;

 }

 void calculateArea(){System.out.println(length\*width);}

}

class TestRectangle2{

 public static void main(String args[]){

  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects

  r1.insert(11,5);

  r2.insert(3,15);

  r1.calculateArea();

  r2.calculateArea();

}

}

Output:

55

45

Real World Example: Account

*File: TestAccount.java*

//Java Program to demonstrate the working of a banking-system

//where we deposit and withdraw amount from our account.

//Creating an Account class which has deposit() and withdraw() methods

class Account{

int acc\_no;

String name;

float amount;

//Method to initialize object

void insert(int a,String n,float amt){

acc\_no=a;

name=n;

amount=amt;

}

//deposit method

void deposit(float amt){

amount=amount+amt;

System.out.println(amt+" deposited");

}

//withdraw method

void withdraw(float amt){

if(amount<amt){

System.out.println("Insufficient Balance");

}else{

amount=amount-amt;

System.out.println(amt+" withdrawn");

}

}

//method to check the balance of the account

void checkBalance(){System.out.println("Balance is: "+amount);}

//method to display the values of an object

void display(){System.out.println(acc\_no+" "+name+" "+amount);}

}

//Creating a test class to deposit and withdraw amount

class TestAccount{

public static void main(String[] args){

Account a1=new Account();

a1.insert(832345,"Ankit",1000);

a1.display();

a1.checkBalance();

a1.deposit(40000);

a1.checkBalance();

a1.withdraw(15000);

a1.checkBalance();

}}

Output:

832345 Ankit 1000.0

Balance is: 1000.0

40000.0 deposited

Balance is: 41000.0

15000.0 withdrawn

Balance is: 26000.0

Access Modifiers in Java

The access modifiers in Java specify 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.

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();

  }

}

Output:Hello

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();

  }

}

Output:Hello

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();

   }

}

# Constructors in Java

In [Java](https://www.javatpoint.com/java-tutorial), a constructor is a block of codes similar to the method. It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

**Note:** It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

### Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

## Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



## Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

### Syntax of default constructor:

1. <class\_name>(){}

## Example of default constructor

//Java Program to create and call a default constructor

class Bike1{

//creating a default constructor

Bike1(){System.out.println("Bike is created");}

//main method

public static void main(String args[]){

//calling a default constructor

Bike1 b=new Bike1();

}

}

Output:

Bike is created

#### Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



### Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

### Example of default constructor that displays the default values

//Let us see another example of default constructor

//which displays the default values

class Student3{

int id;

String name;

//method to display the value of id and name

void display(){System.out.println(id+" "+name);}

public static void main(String args[]){

//creating objects

Student3 s1=new Student3();

Student3 s2=new Student3();

//displaying values of the object

s1.display();

s2.display();

}

}

Output:

0 null

0 null

### Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

### Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

### Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

//Java Program to demonstrate the use of the parameterized constructor.

class Student4{

    int id;

    String name;

    //creating a parameterized constructor

    Student4(int i,String n){

    id = i;

    name = n;

    }

    //method to display the values

    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){

    //creating objects and passing values

    Student4 s1 = new Student4(111,"Karan");

    Student4 s2 = new Student4(222,"Aryan");

    //calling method to display the values of object

    s1.display();

    s2.display();

   }

}

Output:

111 Karan

222 Aryan

## Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor [overloading in Java](https://www.javatpoint.com/method-overloading-in-java) is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

### Example of Constructor Overloading

//Java program to overload constructors

class Student5{

    int id;

    String name;

    int age;

    //creating two arg constructor

    Student5(int i,String n){

    id = i;

    name = n;

    }

    //creating three arg constructor

    Student5(int i,String n,int a){

    id = i;

    name = n;

    age=a;

    }

    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){

    Student5 s1 = new Student5(111,"Karan");

    Student5 s2 = new Student5(222,"Aryan",25);

    s1.display();

    s2.display();

   }

}

Output:

111 Karan 0

222 Aryan 25

## Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

|  |  |
| --- | --- |
| Java Constructor | Java Method |
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |



## Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

* By constructor
* By assigning the values of one object into another
* By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

//Java program to initialize the values from one object to another object.

class Student6{

    int id;

    String name;

    //constructor to initialize integer and string

    Student6(int i,String n){

    id = i;

    name = n;

    }

    //constructor to initialize another object

    Student6(Student6 s){

    id = s.id;

    name =s.name;

    }

    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){

    Student6 s1 = new Student6(111,"Karan");

    Student6 s2 = new Student6(s1);

    s1.display();

    s2.display();

   }

}

Output:

111 Karan

111 Karan

## Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

class Student7{

    int id;

    String name;

    Student7(int i,String n){

    id = i;

    name = n;

    }

    Student7(){}

    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){

    Student7 s1 = new Student7(111,"Karan");

    Student7 s2 = new Student7();

    s2.id=s1.id;

    s2.name=s1.name;

    s1.display();

    s2.display();

   }

}

Output:

111 Karan

111 Karan

### Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

### Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

### Is there Constructor class in Java?

Yes.

### What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

# Inheritance in Java

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors 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.

### 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.

### Java Inheritance Example



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);

}

}

Programmer salary is:40000.0

Bonus of programmer is:10000

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.

*File: TestInheritance.java*

**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();

}}

Output:

barking...

eating...

## 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.

*File: TestInheritance2.java*

**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();

}}

Output:

weeping...

barking...

eating...

## 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.

*File: TestInheritance3.java*

**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

}}

Output:

meowing...

eating...

## 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?

}

}

Compile Time Error

# 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{}

1. A a=**new** B();//upcasting

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

1. **interface** I{}
2. **class** A{}
3. **class** B **extends** A **implements** I{}

Here, the relationship of B class would be:

B IS-A A

B IS-A I

B IS-A Object

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

### 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();

  }

}

Output:

running safely with 60km.

## 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());

}

}

Output:

SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7

## Java Runtime Polymorphism Example: Shape

**class** Shape{

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

}

**class** Rectangle **extends** Shape{

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

}

**class** Circle **extends** Shape{

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

}

**class** Triangle **extends** Shape{

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

}

**class** TestPolymorphism2{

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

Shape s;

s=**new** Rectangle();

s.draw();

s=**new** Circle();

s.draw();

s=**new** Triangle();

s.draw();

}

}

Output:

drawing rectangle...

drawing circle...

drawing triangle...

## Java Runtime Polymorphism Example: Animal

**class** Animal{

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

}

**class** Dog **extends** Animal{

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

}

**class** Cat **extends** Animal{

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

}

**class** Lion **extends** Animal{

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

}

**class** TestPolymorphism3{

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

Animal a;

a=**new** Dog();

a.eat();

a=**new** Cat();

a.eat();

a=**new** Lion();

a.eat();

}}

Output:

eating bread...

eating rat...

eating meat...

## 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.

#### 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");}

1. **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.

**Difference between Inheritance and Polymorphism**

| S.NO | Inheritance | Polymorphism |
| --- | --- | --- |
| 1. | Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class). | Whereas polymorphism is that which can be defined in multiple forms. |
| 2. | It is basically applied to classes. | Whereas it is basically applied to functions or methods. |
| 3. | Inheritance supports the concept of reusability and reduces code length in object-oriented programming. | Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding). |
| 4. | Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance. | Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding). |
| 5. | It is used in pattern designing. | While it is also used in pattern designing. |
| 6. | **Example :**  The class bike can be inherit from the class of two-wheel vehicles, which is turn could be a subclass of vehicles. | **Example :**  The class bike can have method name set\_color(), which changes the bike’s color based on the name of color you have entered. |

# 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.

*File: Student.java*

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

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

**package** com.javatpoint;

**public** **class** Student{

//private data member

**private** String name;

//getter method for name

**public** String getName(){

**return** name;

}

//setter method for name

**public** **void** setName(String name){

**this**.name=name

}

}

*File: Test.java*

//A Java class to test the encapsulated class.

**package** com.javatpoint;

**class** Test{

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

//creating instance of the encapsulated class

Student s=**new** Student();

//setting value in the name member

s.setName("vijay");

//getting value of the name member

System.out.println(s.getName());

}

}

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output:

vijay

### Read-Only class

//A Java class which has only getter methods.

**public** **class** Student{

//private data member

**private** String college="AKG";

//getter method for college

**public** String getCollege(){

**return** college;

}

}

Now, you can't change the value of the college data member which is "AKG".

s.setCollege("KITE");//will render compile time error

### Write-Only class

//A Java class which has only setter methods.

**public** **class** Student{

//private data member

**private** String college;

//getter method for college

**public** **void** setCollege(String college){

**this**.college=college;

}

}

Now, you can't get the value of the college, you can only change the value of college data member.

System.out.println(s.getCollege());//Compile Time Error, because there is no such method

System.out.println(s.college);//Compile Time Error, because the college data member is private.

//So, it can't be accessed from outside the class

### Another Example of Encapsulation in Java

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

*File: Account.java*

//A Account class which is a fully encapsulated class.

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

**class** Account {

//private data members

**private** **long** acc\_no;

**private** String name,email;

**private** **float** amount;

//public getter and setter methods

**public** **long** getAcc\_no() {

**return** acc\_no;

}

**public** **void** setAcc\_no(**long** acc\_no) {

**this**.acc\_no = acc\_no;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** **float** getAmount() {

**return** amount;

}

**public** **void** setAmount(**float** amount) {

**this**.amount = amount;

}

}

*File: TestAccount.java*

//A Java class to test the encapsulated class Account.

**public** **class** TestEncapsulation {

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

    //creating instance of Account class

    Account acc=**new** Account();

    //setting values through setter methods

    acc.setAcc\_no(7560504000L);

    acc.setName("Sonoo Jaiswal");

    acc.setEmail("sonoojaiswal@javatpoint.com");

    acc.setAmount(500000f);

    //getting values through getter methods

    System.out.println(acc.getAcc\_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());

}

}

Output:

7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0

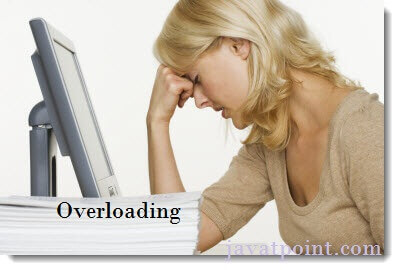
# Method Overloading in Java

If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the [program](https://www.javatpoint.com/java-programs).

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

So, we perform method overloading to figure out the program quickly.



## Advantage of method overloading

Method overloading increases the readability of the program.

### Different ways to overload the method

There are two ways to overload the method in java

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

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

### 1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating [static methods](https://www.javatpoint.com/static-keyword-in-java) so that we don't need to create instance for calling methods.

**class** Adder{

**static** **int** add(**int** a,**int** b){**return** a+b;}

**static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;}

}

**class** TestOverloading1{

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

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(11,11,11));

}}

Output:

22

33

### 2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in [data type](https://www.javatpoint.com/java-data-types). The first add method receives two integer arguments and second add method receives two double arguments.

**class** Adder{

**static** **int** add(**int** a, **int** b){**return** a+b;}

**static** **double** add(**double** a, **double** b){**return** a+b;}

}

**class** TestOverloading2{

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

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(12.3,12.6));

}}

Output:

22

24.9

### Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

**class** Adder{

**static** **int** add(**int** a,**int** b){**return** a+b;}

**static** **double** add(**int** a,**int** b){**return** a+b;}

}

**class** TestOverloading3{

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

System.out.println(Adder.add(11,11));//ambiguity

}}

Output:

Compile Time Error: method add(int,int) is already defined in class Adder

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

#### Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

### Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main() method which receives string array as arguments only. Let's see the simple example:

**class** TestOverloading4{

**public** **static** **void** main(String[] args){System.out.println("main with String[]");

**public** **static** **void** main(String args){System.out.println("main with String");}

**public** **static** **void** main(){System.out.println("main without args");}

}

Output:

main with String[]

## Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

### Example of Method Overloading with TypePromotion

**class** OverloadingCalculation1{

**void** sum(**int** a,**long** b){System.out.println(a+b);}

**void** sum(**int** a,**int** b,**int** c){System.out.println(a+b+c);}

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

  OverloadingCalculation1 obj=**new** OverloadingCalculation1();

  obj.sum(20,20);//now second int literal will be promoted to long

  obj.sum(20,20,20);

  }

}

Output:40

60

### Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

**class** OverloadingCalculation2{

**void** sum(**int** a,**int** b){System.out.println("int arg method invoked");}

**void** sum(**long** a,**long** b){System.out.println("long arg method invoked");}

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

  OverloadingCalculation2 obj=**new** OverloadingCalculation2();

  obj.sum(20,20);//now int arg sum() method gets invoked

  }

}

Output:int arg method invoked

### Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

**class** OverloadingCalculation3{

**void** sum(**int** a,**long** b){System.out.println("a method invoked");}

**void** sum(**long** a,**int** b){System.out.println("b method invoked");}

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

  OverloadingCalculation3 obj=**new** OverloadingCalculation3();

  obj.sum(20,20);//now ambiguity

  }

}

Output:Compile Time Error

# 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

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

//Java Program to demonstrate why we need method overriding

//Here, we are calling the method of parent class with child

//class object.

//Creating a parent class

**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();

  }

}

Output:

Vehicle is running

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.

//Java Program to illustrate the use of Java Method Overriding

//Creating a parent class.

**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

  }

}

Output:

Bike is running safely

### 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.

//Java Program to demonstrate the real scenario of Java Method Overriding

//where three classes are overriding the method of a parent class.

//Creating a parent class.

**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());

}

}

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

### Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

### 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.

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

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

Java Method Overloading example

**class** OverloadingExample{

**static** **int** add(**int** a,**int** b){**return** a+b;}

**static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;}

}

Java Method Overriding example

**class** Animal{

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

}

**class** Dog **extends** Animal{

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

}

# this keyword in Java

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



## Usage of Java this keyword

Here is given the 6 usage of java this keyword.

1. [this can be used to refer current class instance variable.](https://www.javatpoint.com/this1)
2. [this can be used to invoke current class method (implicitly)](https://www.javatpoint.com/this2)
3. [this() can be used to invoke current class constructor.](https://www.javatpoint.com/this3)
4. [this can be passed as an argument in the method call.](https://www.javatpoint.com/this4)
5. [this can be passed as argument in the constructor call.](https://www.javatpoint.com/this5)
6. [this can be used to return the current class instance from the method.](https://www.javatpoint.com/this6)

### 1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

#### Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

**class** Student{

**int** rollno;

String name;

**float** fee;

Student(**int** rollno,String name,**float** fee){

rollno=rollno;

name=name;

fee=fee;

}

**void** display(){System.out.println(rollno+" "+name+" "+fee);}

}

**class** TestThis1{

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

Student s1=**new** Student(111,"ankit",5000f);

Student s2=**new** Student(112,"sumit",6000f);

s1.display();

s2.display();

}}

**Output:**

0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

#### Solution of the above problem by this keyword

**class** Student{

**int** rollno;

String name;

**float** fee;

Student(**int** rollno,String name,**float** fee){

**this**.rollno=rollno;

**this**.name=name;

**this**.fee=fee;

}

**void** display(){System.out.println(rollno+" "+name+" "+fee);}

}

**class** TestThis2{

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

Student s1=**new** Student(111,"ankit",5000f);

Student s2=**new** Student(112,"sumit",6000f);

s1.display();

s2.display();

}}

**Output:**

111 ankit 5000.0

112 sumit 6000.0

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

#### Program where this keyword is not required

**class** Student{

**int** rollno;

String name;

**float** fee;

Student(**int** r,String n,**float** f){

rollno=r;

name=n;

fee=f;

}

**void** display(){System.out.println(rollno+" "+name+" "+fee);}

}

**class** TestThis3{

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

Student s1=**new** Student(111,"ankit",5000f);

Student s2=**new** Student(112,"sumit",6000f);

s1.display();

s2.display();

}}

**Output:**

111 ankit 5000.0

112 sumit 6000.0

#### It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

### 2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



**class** A{

**void** m(){System.out.println("hello m");}

**void** n(){

System.out.println("hello n");

//m();//same as this.m()

**this**.m();

}

}

**class** TestThis4{

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

A a=**new** A();

a.n();

}}

**Output:**

hello n

hello m

### 3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Calling default constructor from parameterized constructor:**

**class** A{

A(){System.out.println("hello a");}

A(**int** x){

**this**();

System.out.println(x);

}

}

**class** TestThis5{

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

A a=**new** A(10);

}}

**Output:**

hello a

10

**Calling parameterized constructor from default constructor:**

**class** A{

A(){

**this**(5);

System.out.println("hello a");

}

A(**int** x){

System.out.println(x);

}

}

**class** TestThis6{

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

A a=**new** A();

}}

**Output:**

5

hello a

### Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

**class** Student{

**int** rollno;

String name,course;

**float** fee;

Student(**int** rollno,String name,String course){

**this**.rollno=rollno;

**this**.name=name;

**this**.course=course;

}

Student(**int** rollno,String name,String course,**float** fee){

**this**(rollno,name,course);//reusing constructor

**this**.fee=fee;

}

**void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}

}

**class** TestThis7{

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

Student s1=**new** Student(111,"ankit","java");

Student s2=**new** Student(112,"sumit","java",6000f);

s1.display();

s2.display();

}}

**Output:**

111 ankit java 0.0

112 sumit java 6000.0

#### Rule: Call to this() must be the first statement in constructor.

**class** Student{

**int** rollno;

String name,course;

**float** fee;

Student(**int** rollno,String name,String course){

**this**.rollno=rollno;

**this**.name=name;

**this**.course=course;

}

Student(**int** rollno,String name,String course,**float** fee){

**this**.fee=fee;

**this**(rollno,name,course);//C.T.Error

}

**void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}

}

**class** TestThis8{

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

Student s1=**new** Student(111,"ankit","java");

Student s2=**new** Student(112,"sumit","java",6000f);

s1.display();

s2.display();

}}

**Output:**

Compile Time Error: Call to this must be first statement in constructor

### 4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

**class** S2{

**void** m(S2 obj){

  System.out.println("method is invoked");

  }

**void** p(){

  m(**this**);

  }

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

  S2 s1 = **new** S2();

  s1.p();

  }

}

**Output:**

method is invoked

### Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

### 5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

**class** B{

  A4 obj;

  B(A4 obj){

**this**.obj=obj;

  }

**void** display(){

    System.out.println(obj.data);//using data member of A4 class

  }

}

**class** A4{

**int** data=10;

  A4(){

   B b=**new** B(**this**);

   b.display();

  }

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

   A4 a=**new** A4();

  }

}

Output:10

### 6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

### Syntax of this that can be returned as a statement

return\_type method\_name(){

**return** **this**;

}

### Example of this keyword that you return as a statement from the method

**class** A{

A getA(){

**return** **this**;

}

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

}

**class** Test1{

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

**new** A().getA().msg();

}

}

**Output:**

Hello java

### Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

**class** A5{

**void** m(){

System.out.println(**this**);//prints same reference ID

}

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

A5 obj=**new** A5();

System.out.println(obj);//prints the reference ID

obj.m();

}

}

**Output:**

A5@22b3ea59

A5@22b3ea59

# 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

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

## 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();

}}

**Output:**

black

white

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();

}}

Output:

eating...

barking...

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();

}}

Output:

animal is created

dog is created

#### 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.

**Another example of super keyword where super() is provided by the compiler implicitly.**

**class** Animal{

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

}

**class** Dog **extends** Animal{

Dog(){

System.out.println("dog is created");

}

}

**class** TestSuper4{

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

Dog d=**new** Dog();

}}

Output:

animal is created

dog is created

## 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();

}}

Output:

1 ankit 45000

# Java static keyword

The **static keyword** in [Java](https://www.javatpoint.com/java-tutorial) is used for memory management mainly. We can apply static keyword with [variables](https://www.javatpoint.com/java-variables), methods, blocks and [nested classes](https://www.javatpoint.com/java-inner-class). The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



## 1) Java static variable

If you declare any variable as static, it is known as a static variable.

* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.

### Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

#### Understanding the problem without static variable

**class** Student{

**int** rollno;

     String name;

     String college="ITS";

}

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all [objects](https://www.javatpoint.com/object-and-class-in-java). If we make it static, this field will get the memory only once.

### Example of static variable

//Java Program to demonstrate the use of static variable

**class** Student{

**int** rollno;//instance variable

   String name;

**static** String college ="ITS";//static variable

   //constructor

   Student(**int** r, String n){

   rollno = r;

   name = n;

   }

   //method to display the values

**void** display (){System.out.println(rollno+" "+name+" "+college);}

}

//Test class to show the values of objects

**public** **class** TestStaticVariable1{

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

 Student s1 = **new** Student(111,"Karan");

 Student s2 = **new** Student(222,"Aryan");

 //we can change the college of all objects by the single line of code

 //Student.college="BBDIT";

 s1.display();

 s2.display();

 }

}

Output:

111 Karan ITS

222 Aryan ITS



### Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

//Java Program to demonstrate the use of an instance variable

//which get memory each time when we create an object of the class.

**class** Counter{

**int** count=0;//will get memory each time when the instance is created

Counter(){

count++;//incrementing value

System.out.println(count);

}

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

//Creating objects

Counter c1=**new** Counter();

Counter c2=**new** Counter();

Counter c3=**new** Counter();

}

}

Output:

1

1

1

### Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

//Java Program to illustrate the use of static variable which

//is shared with all objects.

**class** Counter2{

**static** **int** count=0;//will get memory only once and retain its value

Counter2(){

count++;//incrementing the value of static variable

System.out.println(count);

}

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

//creating objects

Counter2 c1=**new** Counter2();

Counter2 c2=**new** Counter2();

Counter2 c3=**new** Counter2();

1. }

}

Output:

1

2

3

## 2) Java static method

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than the object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.

### Example of static method

//Java Program to demonstrate the use of a static method.

**class** Student{

**int** rollno;

     String name;

**static** String college = "ITS";

     //static method to change the value of static variable

**static** **void** change(){

     college = "BBDIT";

     }

     //constructor to initialize the variable

     Student(**int** r, String n){

     rollno = r;

     name = n;

     }

     //method to display values

**void** display(){System.out.println(rollno+" "+name+" "+college);}

}

//Test class to create and display the values of object

**public** **class** TestStaticMethod{

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

    Student.change();//calling change method

    //creating objects

    Student s1 = **new** Student(111,"Karan");

    Student s2 = **new** Student(222,"Aryan");

    Student s3 = **new** Student(333,"Sonoo");

    //calling display method

    s1.display();

    s2.display();

    s3.display();

    }

}

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

### Another example of a static method that performs a normal calculation

//Java Program to get the cube of a given number using the static method

**class** Calculate{

**static** **int** cube(**int** x){

**return** x\*x\*x;

  }

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

**int** result=Calculate.cube(5);

  System.out.println(result);

  }

}

Output:125

### Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

**class** A{

**int** a=40;//non static

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

  System.out.println(a);

 }

}

Output:Compile Time Error

### Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates an object first then call main() method that will lead the problem of extra memory allocation.

## 3) Java static block

* Is used to initialize the static data member.
* It is executed before the main method at the time of classloading.

### Example of static block

**class** A2{

**static**{System.out.println("static block is invoked");}

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

   System.out.println("Hello main");

  }

}

Output:static block is invoked

Hello main

### Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the [main method](https://www.javatpoint.com/java-main-method).

**class** A3{

**static**{

  System.out.println("static block is invoked");

  System.exit(0);

  }

}

Output:

static block is invoked

Since JDK 1.7 and above, output would be:

Error: Main method not found in class A3, please define the main method as:

public static void main(String[] args)

or a JavaFX application class must extend javafx.application.Application

# 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?

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

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

## 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.

### Syntax:

**interface** <interface\_name>{

    // declare constant fields

    // declare methods that abstract

    // by default.

}

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();

 }

}

Output:

Hello

## Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

*File: TestInterface1.java*

//Interface declaration: by first user

**interface** Drawable{

**void** draw();

}

//Implementation: by second user

**class** Rectangle **implements** Drawable{

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

}

**class** Circle **implements** Drawable{

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

}

//Using interface: by third user

**class** TestInterface1{

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

Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()

d.draw();

}}

Output:

drawing circle

## Java Interface Example: Bank

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

*File: TestInterface2.java*

**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());

}}

Output:

ROI: 9.15

## 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();

 }

}

Output:Hello

Welcome

## 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();

 }

}

Output:

Hello

Welcome

## Java 8 Default Method in Interface

*File: TestInterfaceDefault.java*

**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();

}}

Output:

drawing rectangle

default method

## Java 8 Static Method in Interface

*File: TestInterfaceStatic.java*

**interface** Drawable{

**void** draw();

**static** **int** cube(**int** x){**return** x\*x\*x;}

}

**class** Rectangle **implements** Drawable{

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

}

**class** TestInterfaceStatic{

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

Drawable d=**new** Rectangle();

d.draw();

System.out.println(Drawable.cube(3));

}}

Output:

drawing rectangle

27

## Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, [Serializable](https://www.javatpoint.com/serialization-in-java), 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 which is known as a nested interface. We will learn it in detail in the [nested classes](https://www.javatpoint.com/java-inner-class) chapter. For example:

**interface** printable{

**void** print();

**interface** MessagePrintable{

**void** msg();

 }

}