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**

1. **class** Subclass-name **extends** Superclass-name
2. {
3. //methods and fields
4. }

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.

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.

1. **class** Employee{
2. **float** salary=40000;
3. }
4. **Public class** Programmer **extends** Employee{
5. **int** bonus=10000;
6. **public** **static** **void** main(String args[]){
7. Programmer p=**new** Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Programmer)

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.

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **Public class** TestInheritance{
8. **public** **static** **void** main(String args[]){
9. Dog d=**new** Dog();
10. d.bark();
11. d.eat();
12. }}

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*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** weep(){System.out.println("weeping...");}
9. }
10. **class** TestInheritance2{
11. **public** **static** **void** main(String args[]){
12. BabyDog d=**new** BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}

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*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **void** eat()1{System.out.println("eating...");}
5. }
6. **class** Dog **extends** Animal{
7. **void** bark(){System.out.println("barking...");}
8. }
9. **class** Cat **extends** Animal{
10. **void** meow(){System.out.println("meowing...");}
11. }
12. **class** TestInheritance3{
13. **public** **static** **void** main(String args[]){
14. Cat c=**new** Cat();
15. c.eat();
16. c.meow();
17. //c.bark();//C.T.Error
18. }}

Output:

meowing...

eating...

## **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.

1. **class** A{
2. **void** msg(){
3. System.out.println("Hello");
4. }
5. }
6. **class** B{
7. **void** msg(){System.out.println("Welcome");
8. }
9. }
10. **class** C **extends** A,B{//suppose if it were
12. **public** **static** **void** main(String args[]){
13. C obj=**new** C();
14. obj.msg();//Now which msg() method would be invoked?
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=C)

Compile Time Error

# Aggregation in Java:

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

1. **class** Employee {
2. **int** id
3. String name;
4. Address address;
5. ...
6. }

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

### **Why use Aggregation?**

* For Code Reusability.

### **When use Aggregation?**

* Code reuse is also best achieved by aggregation when there is no is-a relationship.
* Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

### **Understanding meaningful example of Aggregation**

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

#### **Address.java**

1. **public** **class** Address {
2. String city,state,country;
4. **public** Address(String city, String state, String country) {
5. **this**.city = city;
6. **this**.state = state;
7. **this**.country = country;
8. }
10. }

#### **Emp.java**

1. **public** **class** Emp {
2. **int** id;
3. String name;
4. Address address;
6. **public** Emp(**int** id, String name,Address address) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.address=address;
10. }
12. **void** display(){
13. System.out.println(id+" "+name);
14. System.out.println(address.city+" "+address.state+" "+address.country);
15. }
17. **public** **static** **void** main(String[] args) {
18. Address address1=**new** Address("gzb","UP","india");
19. Address address2=**new** Address("gno","UP","india");
21. Emp e1=**new** Emp(111,"varun",address1);
22. Emp e2=**new** Emp(112,"arun",address2);
24. e1.display();
25. e2.display();
27. }
28. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Emp)

Output:111 varun

gzb UP india

112 arun

gno UP india

## **Java Polymorphism:**

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; [**Inheritance**](https://www.w3schools.com/java/java_inheritance.asp) lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

# 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.

## **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

### **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.

1. **class** Adder{
2. **static** **int** add(**int** a,**int** b){
3. **return** a+b;
4. }
5. **static** **int** add(**int** a,**int** b,**int** c){
6. **return** a+b+c;
7. }
8. }
9. **class** TestOverloading1{
10. **public** **static** **void** main(String[] args){
11. System.out.println(Adder.add(11,11));
12. System.out.println(Adder.add(11,11,11));
13. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestOverloading1)

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.

1. **class** Adder{
2. **static** **int** add(**int** a, **int** b){
3. **return** a+b;
4. }
5. **static** **double** add(**double** a, **double** b){
6. **return** a+b;
7. }
8. }
9. **class** TestOverloading2{
10. **public** **static** **void** main(String[] args){
11. System.out.println(Adder.add(11,11));
12. System.out.println(Adder.add(12.3,12.6));
13. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestOverloading2)

Output:

22

24.9

# 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).



### **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.

1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. **class** Vehicle{
4. //defining a method
5. **void** run(){
6. System.out.println("Vehicle is running");
7. }
8. }
9. //Creating a child class
10. **class** Bike2 **extends** Vehicle{
11. //defining the same method as in the parent class
12. **void** run(){System.out.println("Bike is running safely");}
14. **public** **static** **void** main(String args[]){
15. Bike2 obj = **new** Bike2();//creating object
16. obj.run();//calling method
17. }
18. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Bike2)

Output:

Bike is running safely

1. **class** Bank{
2. **int** getRateOfInterest(){**return** 0;}
3. }
4. //Creating child classes.
5. **class** SBI **extends** Bank{
6. **int** getRateOfInterest(){**return** 8;}
7. }
9. **class** ICICI **extends** Bank{
10. **int** getRateOfInterest(){**return** 7;}
11. }
12. **class** AXIS **extends** Bank{
13. **int** getRateOfInterest(){**return** 9;}
14. }
15. //Test class to create objects and call the methods
16. **class** Test2{
17. **public** **static** **void** main(String args[]){
18. SBI s=**new** SBI();
19. ICICI i=**new** ICICI();
20. AXIS a=**new** AXIS();
21. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
22. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
23. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
24. }
25. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Test2)

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

# 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. |

# 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.

## **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.

1. **class** Animal {
2. **void** eat(){
3. System.out.println("eating...");
4. }
5. }
6. **class** Dog **extends** Animal{
7. **void** eat(){
8. System.out.println("eating bread...");
9. }
10. **void** bark(){
11. System.out.println("barking...");
12. }
13. **void** work(){
14. **super**.eat();
15. bark();
16. }
17. }
18. **class** TestSuper2{
19. **public** **static** **void** main(String args[]){
20. Dog d=**new** Dog();
21. d.work();
22. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestSuper2)

Output:

eating...

barking...

# Final Keyword In Java:

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

## **Java final method**

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

### **Example of final method**

1. **class** Bike{
2. **final** **void** run(){System.out.println("running");}
3. }
5. **class** Honda **extends** Bike{
6. **void** run(){System.out.println("running safely with 100kmph");}
8. **public** **static** **void** main(String args[]){
9. Honda honda= **new** Honda();
10. honda.run();
11. }
12. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Honda)

Output:Compile Time Error

### **Class Names**

* **Convention**: Use PascalCase (also known as UpperCamelCase).
* **Rule**: The first letter of each word is capitalized, and no spaces or underscores are used.
* **Examples**: CustomerOrder, BankAccount, UserProfile

### 2. **Interface Names**

* **Convention**: Use PascalCase and often prefixed with I (optional and less common in modern Java).
* **Rule**: Same as class names.
* **Examples**: Runnable, Serializable, List, IUserService (less common)

### 3. **Method Names**

* **Convention**: Use camelCase.
* **Rule**: The first word is in lowercase, and the first letter of each subsequent word is capitalized.
* **Examples**: calculateTotal, getUserName, findMaximumValue

### 4. **Variable Names**

* **Convention**: Use camelCase.
* **Rule**: Same as method names.
* **Examples**: firstName, accountBalance, numberOfItems

### 5. **Constant Names**

* **Convention**: Use ALL\_UPPERCASE\_WITH\_UNDERSCORES.
* **Rule**: All letters are uppercase, and words are separated by underscores.
* **Examples**: PI, MAX\_VALUE, DEFAULT\_TIMEOUT

### 6. **Package Names**

* **Convention**: Use all lowercase.
* **Rule**: Typically start with the reversed domain name of the organization, followed by more specific names.
* **Examples**: com.example.project, org.openjdk.tools, edu.university.departm