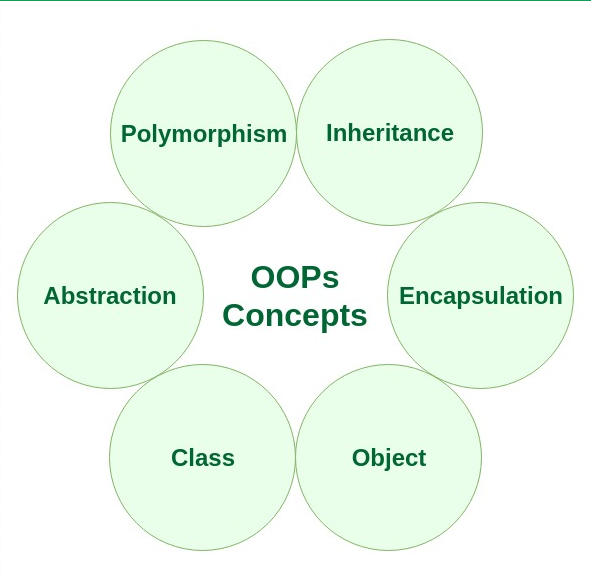
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* JAVA Oops \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*



* Inheritance in Java

Understand the requirement:

Suppose we have two application like whats-up and facebook

**Facebook**

**friends**

**Post**

**Comments**

**timeline**

**Notification**

**Like &rejection**

**Whats-up**

**Activitation**

**Status**

**Call**

**Video call**

**chat**

extends

Facebook class whats-up class

Noe suppose Facebook want the all the feature of the what’s- up then how will you do that?

We will inherit the what’s-up class in Facebook class where it will copy the functionality of whats class to Facebook class.

* If we want only video call in Facebook how we will do that? (Aggregation)

It is not required to copy all code of the whatsup we simply aggregate that video call class in face and use that feature

So using this they can use the individual feature instead of sharing all the feature

(if we want only particular functionality we use aggregation so aggregation for particular functionality.

But when we inheritance then all the base class functionality is copied in child class)

* **Inheritance have “is a” relationship.**

**E.g. video call is a extended by Facebook or class A extended by class C.**

* **Aggregation have “ has a “ relationship. // see later**
* **In inheritance child class acquires all the properties of base/parent class.**

### **syntax of Java Inheritance**

**class** Subclass-name **extends** Superclass-name

{

   //methods and fields

}

Class A{

Int a=10;

Void job (){

System.out.println(“inside job method of class A”)

}

Void factory (){

System.out.println(“inside factory of class A”)

}

}

Class B extends class A{

Inr b =20;

Void test(){

System.out.println(“test class B”)

}} +

Copy of class A

We can use all the variable and method of class A using object of class B

Class A class B

**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 (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes 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 (so runtime polymorphism 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 **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.

### **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();   using object of programmer class we access salary which is base class variable

   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

## **Types of inheritance in java**

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

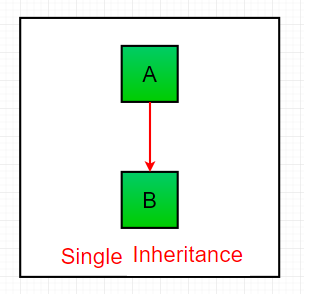
In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

#### Types of inheritance in Java **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**

* In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.
* [](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/inheritance1.png)

Advantage : you can use variable and method of base class in child class.

* Syntax : className extends baseClass name{ }

e.g. class A extends class B{ }

class boy{ class girl extends boy{

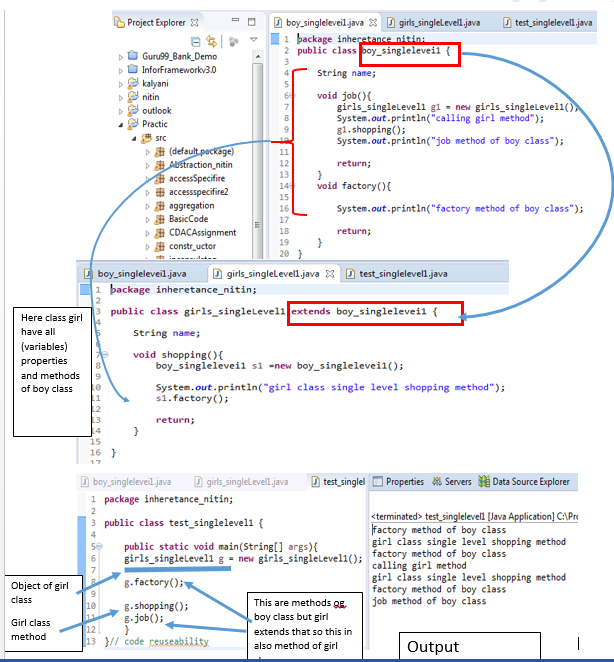
String Name; int x;

Void job(){

System.out.println(“nitin”); ---- variables

} ---- methods

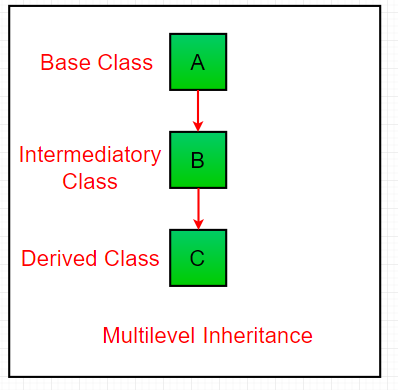
} }



## **Multilevel Inheritance**

1. Here class A extended by class B { so Class B have all the properties and method of class A}

2. and class c extends class B so class C have properties and behavior of both class A and class B



Class c extends class b

Class B extends class A

Here class A extend by class B and that class B extend by class C

Class A{ class B extends class A{ class C extends class B{

String Name; int x; int z=20;

Void job(){ void display(){ }

Sys.out.println(“ job class A”)

} **{it have all the class A {it have all the class A and**

Int sum(){ **properties} class B properties)**

Int a=10,b=20 }

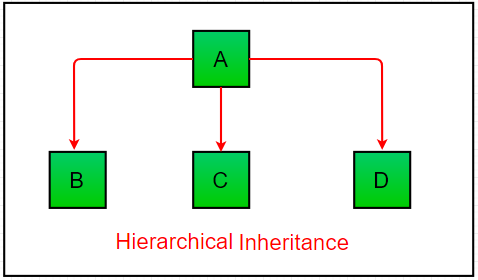
Return a=b;

}

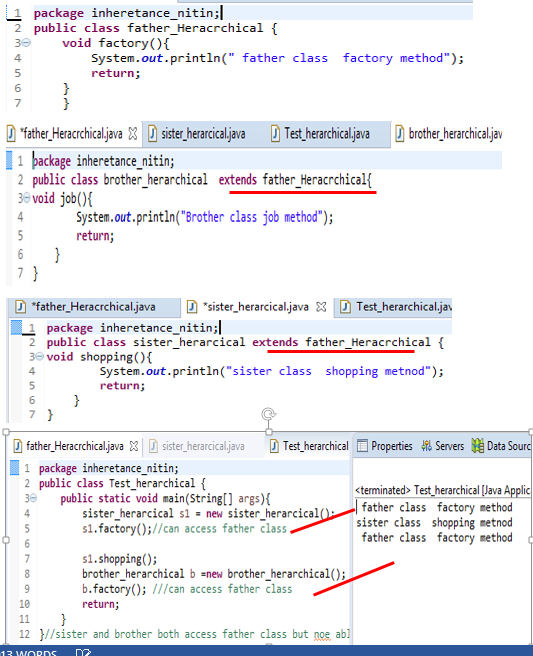
} }



## **Hierarchical Inheritance Example**



Refer to child and parent class relationship where more than one class extends one class.



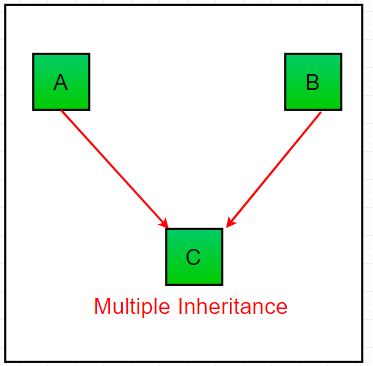
* [**Multiple Inheritance**](https://www.geeksforgeeks.org/java-and-multiple-inheritance/)
* Multiple inheritance supported in C++ only java not support it.
* Multiple inheritance refers to concept that one class extend more than one class.

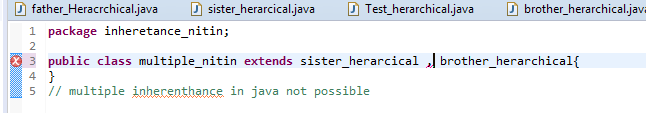
## **Why multiple inheritance is not supported in java?**

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.

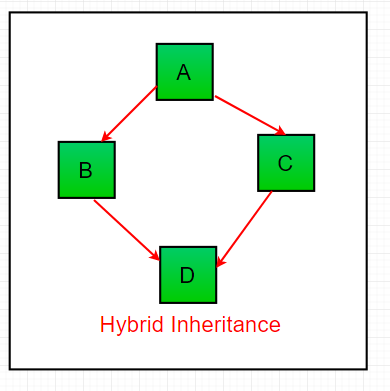
Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.





[**Multiple Inheritance**](https://www.geeksforgeeks.org/java-and-multiple-inheritance/)**(Through Interfaces) :**In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support [multiple inheritance](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. In java, we can achieve multiple inheritance only through [Interfaces](http://quiz.geeksforgeeks.org/interfaces-in-java/). In image below, Class C is derived from interface A and B.

* **Hybrid Inheritance(Through Interfaces) :**It is a mix of two or more of the above types of inheritance. Since java doesn’t support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through [Interfaces](http://quiz.geeksforgeeks.org/interfaces-in-java/).

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/inheritance-1.png)

**Important facts about inheritance in Java**

* **Default superclass**: Except [Object](https://www.geeksforgeeks.org/object-class-in-java/) class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of [Object](https://www.geeksforgeeks.org/object-class-in-java/) class.
* **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support [multiple inheritance](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. Although with interfaces, multiple inheritance is supported by java.
* **Inheriting Constructors:**A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
* **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

**What all can be done in a Subclass?**

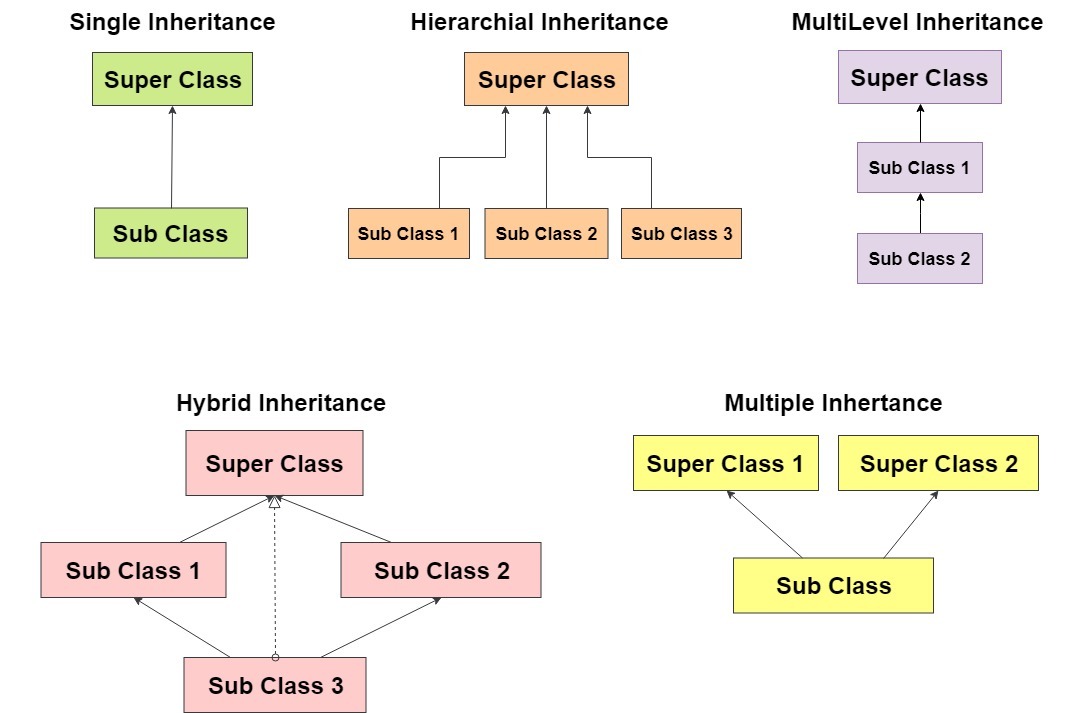
In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

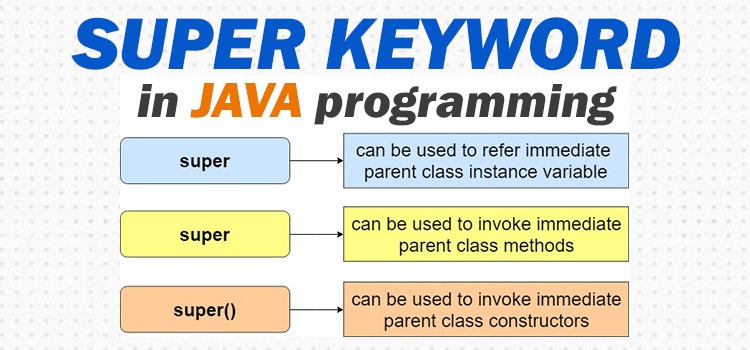
* The inherited fields can be used directly, just like any other fields.
* We can declare new fields in the subclass that are not in the superclass.
* The inherited methods can be used directly as they are.
* We can write a new instance method in the subclass that has the same signature as the one in the superclass, thus [overriding](https://www.geeksforgeeks.org/overriding-in-java/) it (as in example above, toString() method is overridden).
* We can write a new static method in the subclass that has the same signature as the one in the superclass, thus [hiding](https://www.geeksforgeeks.org/g-fact-63/) it.
* We can declare new methods in the subclass that are not in the superclass.
* We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword [super](http://quiz.geeksforgeeks.org/super-keyword/).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* **Super keyword (not in c++)**
* The SUPER keyword in java is a reference variable which is used to refer immediate parent class object.
* Whenere you create the instance of subclass, an instance of parent class is created which is refer by super class reference variable.
* Or







## **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.

2 defination:

The super keyword in java is a reference variable that is used to refer parent class objects. The keyword “super” came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

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

This scenario occurs when a derived\child class and base\parent class has same data member, In that case there is a possibility of ambiguity for the JVM

Eg:

**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

}

}

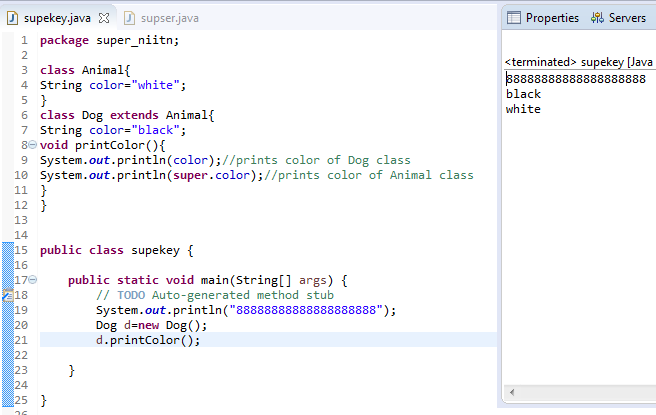
**class** TestSuper1{

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

Dog d=**new** Dog();

d.printColor();

}}



## **Super can be used to invoke parent class method**

This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword.

**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();

}}

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

Output:

eating...

barking...

## **Super is used to invoke parent class constructor.**

Super keyword can also be used to access the parent class constructor. One more important thing is that, ‘’super’ can call both parametric as well as non-parametric constructors depending upon the situation

**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.

## **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

**Important points:**

1. Call to super() must be first statement in Derived(Student) Class constructor.
2. If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.
3. If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called *constructor chaining*..

# Aggregation in Java



* When we want all functionality of other class we use inheritance
* When we want some functionality we use Aggregation.
* Aggregation have “has a” relationship.
* Here we are passing object of aggregate class while creating object
* **In Heratance : Here class girl have all the variable and method of class boy**

e.g. class A extends class B{ }

class boy{ class girl extends boy{

String Name; int x;

Void job(){

System.out.println(“nitin”); ---- variables

} ---- methods

} }

But in aggregation we aggregate one or some functionality

* In Aggregation,**both the entries can survive individually** which means ending one entity will not effect the other entity

Aggregation:

Whats-up Facebook{

{ whats-up a;

Calling();

Status()

Video call() }

Chat()

}

[ Here like Facebook take service from whats-up]

* What is Aggregation : aggregation is the special from of association , it is a relation between two classes howevet it is directional association.(one way , one to other)

FB{ Whats-up{

* } }
* Syntax : className variableName ;
* Need : to maintain code reusability.
* Then what is difference between aggregation and inheritance

Inheritance aggregation

FB{ wA{

Class A{

WA x; callinh();

Int num=10,a=2,b-5; x.calling; chat();

Void add(){

Num=a+b

}} } }

Class B extend A{

} **inside Class B object inside class FB object**

FB{

}

+ reference to Calling() method of WA object

B{

}

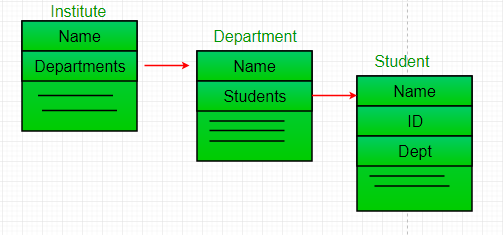
A{ } //a also created with

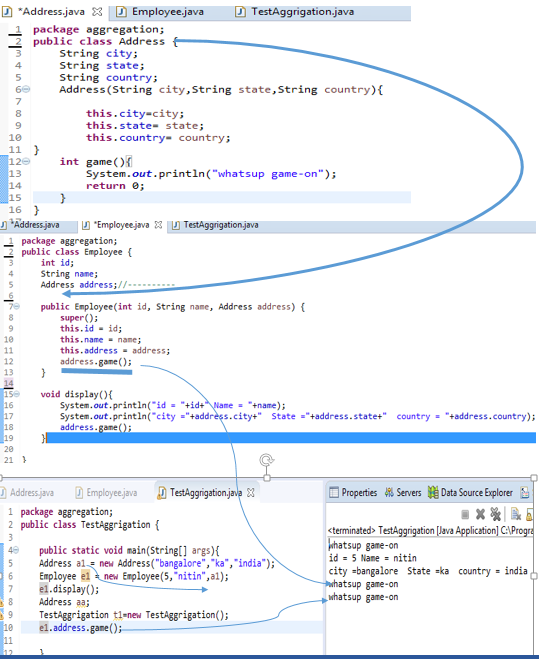
B

It is a special form of Association where:

* It represents **Has-A** relationship.
* It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation,**both the entries can survive individually** which means ending one entity will not effect the other entity

### **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.
* example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make a Institute class which has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s).  
  It represents a **Has-A** relationship.  
  [](https://www.geeksforgeeks.org/cdncontribute.geeksforgeeks.org/wp-content/uploads/Reference.png)



### **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

**public** **class** Address {

String city,state,country;

**public** Address(String city, String state, String country) {

**this**.city = city;

**this**.state = state;

**this**.country = country;

}

}

Emp.java

**public** **class** Emp {

**int** id;

String name;

Address address;

**public** Emp(**int** id, String name,Address address) {

**this**.id = id;

**this**.name = name;

    this.address=address;

}

**void** display(){

System.out.println(id+" "+name);

System.out.println(address.city+" "+address.state+" "+address.country);

}

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

Address address1=new Address("gzb","UP","india");

Address address2=new Address("gno","UP","india");

Emp e=**new** Emp(111,"varun",address1);

Emp e2=**new** Emp(112,"arun",address2);

e.display();

e2.display();

}

}

Output:111 varun

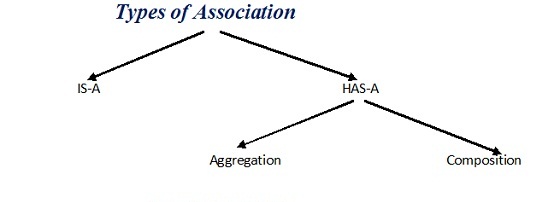
gzb UP india

112 arun

gno UP india

# Association

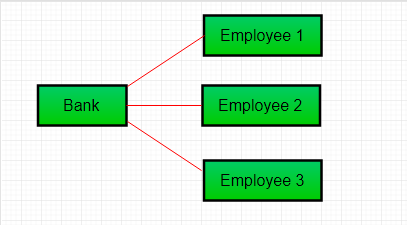
Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.  
In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.



|  |
| --- |
| // Java program to illustrate the  \*\*\*\*\*\*\*\*see main method rest is common cass \*\*\*\*  // concept of Association  import java.io.\*;    // class bank  class Bank  {      private String name;        // bank name      Bank(String name)      {          this.name = name;      }        public String getBankName()      {          return this.name;      }  }    // employee class  class Employee  {      private String name;        // employee name      Employee(String name)      {          this.name = name;      }        public String getEmployeeName()      {          return this.name;      }  }    // Association between both the  // classes in main method  class Association  {      public static void main (String[] args)      {          Bank bank = new Bank("Axis");          Employee emp = new Employee("Neha");            System.out.println(emp.getEmployeeName() +                 " is employee of " + bank.getBankName()); /// only this we get using +      }  } |

Output:

Neha is employee of Axis

In above example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/Aggre.png)

* It represents **Has-A** relationship.
* It is a **unidirectional association** i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation,**both the entries can survive individually** which means ending one entity will not effect the other entity

## Association vs Aggregation vs Composition

Lets discuss **difference between Association, Aggregation and Composition**:

Although all three are related terms, there are some major differences in the way they relate two classes. **Association** is a relationship between two separate classes and the association can be of any type say one to one, one to may etc. It joins two entirely separate entities.

* [Aggregation](https://beginnersbook.com/2013/05/aggregation/) is a special form of association which is a unidirectional one way relationship between classes (or entities), for e.g. Wallet and Money classes. Wallet has Money but money doesn’t need to have Wallet necessarily so its a one directional relationship. In this relationship both the entries can survive if other one ends. In our example if Wallet class is not present, it does not mean that the Money class cannot exist.**Composition** is a restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other. For e.g. Human and Heart. A human needs heart to live and a heart needs a Human body to survive. In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one too) then its a composition. Heart class has no sense if Human class is not present.
* **Composition**

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

* It represents **part-of** relationship.
* In composition, both the entities are dependent on each other.
* When there is a composition between two entities, the composed object **cannot exist**without the other entity.

**Composition** is the design technique to implement has-a relationship in classes. We can use **java** inheritance or Object **composition** for code reuse. **Java composition**is achieved by using instance variables that refers to other objects. Job.**java**

class A   
{ int x=10;}

class B   
{ int y=20;   
A a=new A();   
System.out.println(a.x+" "+y);   
}

its a sample example to understand composition...

------------------------------------------------------------------------------------

public class Job {   
private String role;   
private long salary;   
private int id;   
  
public String getRole() {   
return role;   
}   
public void setRole(String role) {   
this.role = role;   
}   
public long getSalary() {   
return salary;   
}   
public void setSalary(long salary) {   
this.salary = salary;   
}   
public int getId() {   
return id;   
}   
public void setId(int id) {   
this.id = id;   
}   
  
  
}   
  
  
Person.java   
  
package com.journaldev.composition;   
  
  
public class Person {   
  
//composition has-a relationship   
private Job job;   
  
public Person(){   
this.job=new Job();   
job.setSalary(1000L);   
}   
public long getSalary() {   
return job.getSalary();   
}   
  
}

------------------------------------------------------------------------------------

Lets take example of**Library**.

|  |
| --- |
| // Java program to illustrate  // the concept of Composition  import java.io.\*;  import java.util.\*;    // class book  class Book  {      public String title;      public String author;        Book(String title, String author)      {          this.title = title;          this.author = author;      }  }    ------------------------------------------------------------------------  // Libary class contains  // list of books.  class Library  {        // reference to refer to list of books.      private final List<Book> books;        Library (List<Book> books)      {          this.books = books;      }        public List<Book> getTotalBooksInLibrary(){           return books;      }    } --------------------------------------------------------------------------------    // main method  class GFG  {      public static void main (String[] args)      {            // Creating the Objects of Book class.          Book b1 = new Book("EffectiveJ Java", "Joshua Bloch");          Book b2 = new Book("Thinking in Java", "Bruce Eckel");          Book b3 = new Book("Java: The Complete Reference", "Herbert Schildt");            // Creating the list which contains the          // no. of books.          List<Book> books = new ArrayList<Book>();          books.add(b1);          books.add(b2);          books.add(b3);            Library library = new Library(books);            List<Book> bks = library.getTotalBooksInLibrary();          for(Book bk : bks){                System.out.println("Title : " + bk.title + " and "              +" Author : " + bk.author);          }      }  } |

Output

Title : EffectiveJ Java and Author : Joshua Bloch

Title : Thinking in Java and Author : Bruce Eckel

Title : Java: The Complete Reference and Author : Herbert Schildt

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Aggregation vs Composition**

1. **Dependency:** Aggregation implies a relationship where the child **can exist independently** of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child **cannot exist independent** of the parent. Example: Human and heart, heart don’t exist separate to a Human
2. **Type of Relationship:** Aggregation relation is **“has-a”** and composition is **“part-of”**relation.
3. **Type of association:**Composition is a **strong** Association whereas Aggregation is a **weak** Association.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Polymorphism in Java

# Poly – multiple.

# Polymorphism – Greek meaning ”having multiple forms”

# Type of polymorphism

# There are two type of polymorphism

# 1. static polymorphism Also called complier time polymorphism

# 2. Dynamic polymorphism also called Run time polymorphism

# Complier time polymorphism ---- Method overloading.

# This type of polymorphism achieved by method overloading

# Overloading -🡪 only inside one class.

# Method Overloading

# 1. Multiple method having the same name but different no of parameter/argument or data type is called overloading.

# 2. Function can be overloaded by change in no of argument or change in type of argument

# 3.it dissent check return type of the method, it check no of argument or argument data type.

# Inheritance not required for static/compiler time polymorphism

# Run time polymorphism ----- Method overriding.

# Inheritance is required for Run time polymorphism

# First complete overloading then see this.

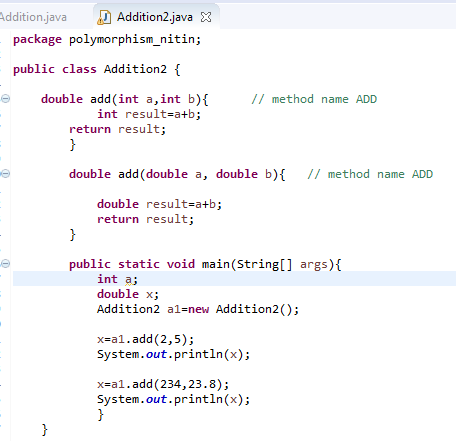
# Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

# This allows us to perform a single action in different ways.

# Complier time polymorphism

In Java, static polymorphism is achieved through method overloading. Method overloading means there are several methods present in a class having the same name but different types/order/number of parameters.

At compile time, Java knows which method to invoke by checking the method signatures.  So, this is called compile time polymorphism or static binding.



# Here depending on method argument and type complier select the method for operation.(we not check return type)

# 

No of argument is sae also argumrnt type is same which create ambiguity so its give an error

# This eror is came because there is not any method who accept three int argument

## **Runtime Polymorphism in Java**

* This type of polymorphism achieved method overriding
* Function Overriding : subclass method overwrite parent class method called overriding.
* Overriding :

1. Only if there is inheritance.

2. Method name and parameter should same.

3. parent class method is overwritten.

* **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:



1. **class** A{}
2. **class** B **extends** A{}
3. 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:

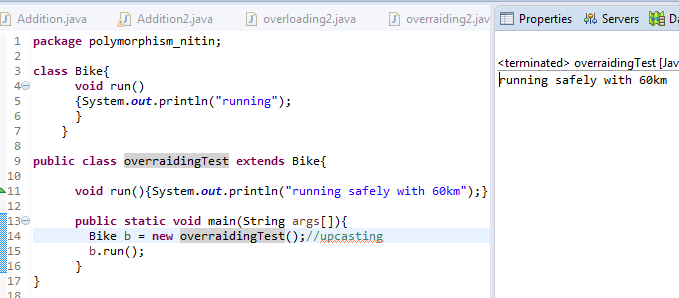
B IS-A A

B IS-A I

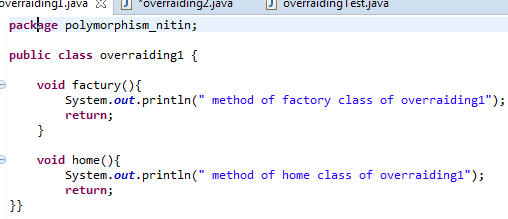
B IS-A Object

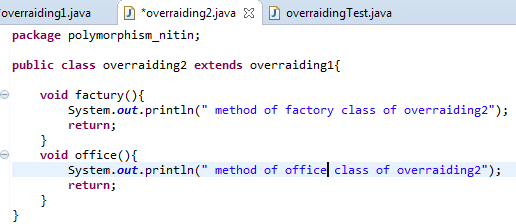
Since Object is the root class of all classes in Java, so we can write B IS-A Object.

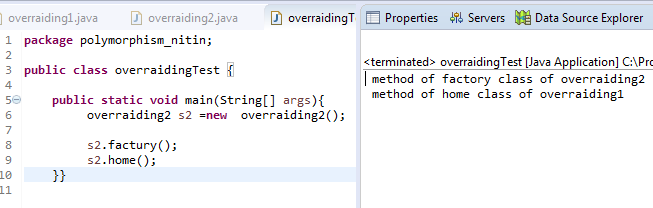
* **Example:**



2.







Parent method is overwritten by chield method

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");}

**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();

}}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=BabyDog1)

Output:

Dog is eating

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **Abstraction in Java**

It is the process of hiding internal details or inplimentation details & showing only functionality to the user

Types : you can achieve the abstraction using following type.

1. Abstract class - (0 to 100%)abstraction can achieve.

2. Interface --- 100% abstraction

1> Abstract class : RULES

1> if there is any abstract method n class then that class should be abstract class

2> abstract method only provide declaration it should not have Body

3. IF you are exteactign any abstract class you have to check is any abstract method in that class.

If it is there then you have to provide the implementation of that method. .

4. we can not create object of abstract class.

2> If chield does not implement all abstract method of abstract parent class then the chield must be an abstract class.

3. Abstract class have abstract method as well as (normal) concrete method.

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

### **Abstract class in Java**

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

#### **Points to Remember**

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

### **Abstract Method in Java**

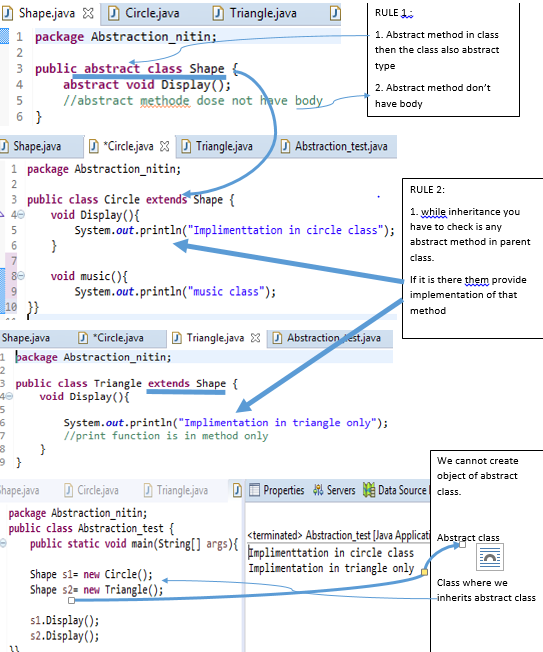
A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract method**

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

**Abstract classes and Abstract methods :**

1. An abstract class is a class that is declared with [abstract keyword.](https://www.geeksforgeeks.org/abstract-keyword-in-java/)
2. An abstract method is a method that is declared without an implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method defined abstract must always be redefined in the subclass,thus making [overriding](http://contribute.geeksforgeeks.org/overriding-in-java/) compulsory OR either make subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with abstract keyword.
6. There can be no object of an abstract class.That is, an abstract class can not be directly instantiated with the [*new operator*](https://www.geeksforgeeks.org/new-operator-java/).
7. An abstract class can have parametrized constructors and default constructor is always present in an abstract class.



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

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

**abstract** **class** Bike{

**abstract** **void** run();

}

**class** Honda4 **extends** Bike{

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

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

 Bike obj = **new** Honda4();

 obj.run();

}

}

running safely

n this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

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

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

*File: TestAbstraction1.java*

**abstract** **class** Shape{

**abstract** **void** draw();

}

//In real scenario, implementation is provided by others i.e. unknown by end user

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

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

}

**class** Circle1 **extends** Shape{

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

}

//In real scenario, method is called by programmer or user

**class** TestAbstraction1{

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

Shape s=**new** Circle1();//In a real scenario, object is provided through method, e.g., getShape() method

s.draw();

}

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction1)

drawing circle

### **Another example of Abstract class in java**

*File: TestBank.java*

**abstract** **class** Bank{

**abstract** **int** getRateOfInterest();

}

**class** SBI **extends** Bank{

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

}

**class** PNB **extends** Bank{

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

}

**class** TestBank{

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

Bank b;

b=**new** SBI();

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

b=**new** PNB();

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

}}

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

Rate of Interest is: 7 %

Rate of Interest is: 8 %

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

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

*File: TestAbstraction2.java*

//Example of an abstract class that has abstract and non-abstract methods

**abstract** **class** Bike{

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

**abstract** **void** run();

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

 }

//Creating a Child class which inherits Abstract class

**class** Honda **extends** Bike{

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

 }

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

**class** TestAbstraction2{

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

  Bike obj = **new** Honda();

  obj.run();

  obj.changeGear();

 }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction2)

bike is created

running safely..

gear changed

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

**class** Bike12{

**abstract** **void** run();

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike12)

compile time error

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

### **Another real scenario of abstract class**

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

#### **Note: If you are beginner to java, learn interface first and skip this example.**

**interface** A{

**void** a();

**void** b();

**void** c();

**void** d();

}

**abstract** **class** B **implements** A{

**public** **void** c(){System.out.println("I am c");}

}

**class** M **extends** B{

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

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

**public** **void** d(){System.out.println("I am d");}

}

**class** Test5{

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

A a=**new** M();

a.a();

a.b();

a.c();

a.d();

}}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Test5)

Output:I am a

I am b

I am c

I am d

**Encapsulation vs Data Abstraction**

1. [Encapsulation](http://contribute.geeksforgeeks.org/encapsulation-in-java/) is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).
2. While encapsulation groups together data and methods that act upon the data, data abstraction deals with exposing the interface to the user and hiding the details of implementation.

**Advantages of Abstraction**

1. It reduces the complexity of viewing the things.
2. Avoids code duplication and increases reusability.
3. Helps to increase security of an application or program as only important details are provided to the user.

* **Why can’t we create the object of an abstract class**

Because this class are incomplete, they have abstract method that has no body so if java allow you to create object of this class then if someone calls the abstract method using that object then what would happen?

There would be no actual implementation of the method to invoke.

Also because an object in concreate an abstract class is like template so you have to extent it and build on it before you can use it

# Difference between Abstract Class and Concrete Class in Java

[**Abstract Class**](https://www.geeksforgeeks.org/pure-virtual-functions-and-abstract-classes/)**:** An abstract class is a type of class in Java that is declared by the abstract keyword. An abstract class cannot be instantiated directly, i.e. object of such class cannot be created directly using new keyword. An abstract class can be instantiated either by concrete subclass, or by defining all the abstract method along with the new statement. It may or may not contain abstract method. An abstract method is declared by abstract keyword, such methods cannot have a body. If a class contains abstract method, then it also needs to be abstract.

**Concrete Class:** A concrete class in Java is a type of subclass, which implements all the abstract method of its super abstract class which it extends to. It also has implementations of all methods of interfaces it implements.

**Abstract Class vs Concrete Class**

1. **Modifier:** An abstract class is declared using abstract modifier. Concrete class should not be declared using abstract keyword, on doing so, it will also become abstract class.
2. **Instantiation:** An abstract class cannot be instantiated directly, i.e. object of such class cannot be created directly using new keyword. An abstract class can be instantiated either by concrete subclass, or by defining all the abstract method along with the new statement. A concrete class can be instantiated directly, using a new keyword.
3. **Abstract methods:** An abstract class may or may not, have an abstract method. A concrete class cannot have an abstract method, because class containing an abstract method must also be abstract.
4. **Final:** An abstract class cannot be **final**, because all its abstract methods must defined in the subclass. A concrete class can be declared as **final**.
5. **Interface:** Interface implementation is not possible with abstract class, however, it is possible with concrete class.

| **ABSTRACT CLASS** | **CONCRETE CLASS** |
| --- | --- |
| An abstract class is declared using abstract modifier. | A concrete class is note declared using abstract modifier. |
| An abstract class cannot be directly instantiated using the new keyword. | A concrete class can be directly instantiated using the new keyword. |
| An abstract class may or may not contain abstract methods. | A concrete class cannot contain an abstract method. |
| An abstract class cannot be declared as final. | A concrete class can be declared as final. |
| Interface implementation is not possible | Interface implementation is possible. |

**Some important points:**

* A concrete class is a subclass of an abstract class, which implements all its abstract method.
* Abstract methods cannot have body.
* Abstract class can have static fields and static method, like other classes.
* An abstract class cannot be declared as final.
* Only abstract class can have abstract methods.
* A private, final, static method cannot be abstract, as it cannot be overridden in a subclass.
* Abstract class cannot have abstract constructors.
* Abstract class cannot have abstract static methods.
* If a class extends an abstract class, then it should define all the abstract methods (override) of the base abstract class. If not, the subclass(the class extending abstract class) must also be defined as abstract class.

# 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*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple 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.

## **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.

---------------------------------------------------------------------------

Interface Test{ ------🡪 JVM -------------🡪 interface Test{

Int I = 10; --------------------------------------🡪 public static final int I =10;

Void Drow(); -------------------------------------🡪 public abstract void Drow();

} }

Complier make instance complier make every method as public abstract

Variable as “public **static final** “

----------------------------------------------------------------------------------

* It provide 100% of abstraction.

## **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>{

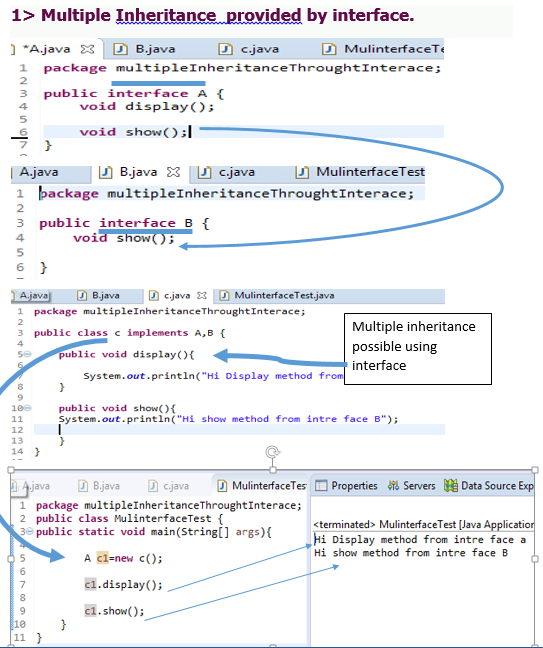
     // declare constant fields

 // declare methods that abstract

     // by default.  }

**1> Interface to achieve Abstraction**





**1. Interfac -🡪 Impliments**

**2. Abstract class 🡪 extends**

**3. we cannot create object of interface or Abstract class**

## **Internal addition by the compiler**

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

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



#### **The relationship between classes and interfaces**

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



## **Java Interface Example**

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

**interface** printable{

**void** print();

}

**class** A6 **implements** printable{

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

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

A6 obj = **new** A6();

obj.print();

 }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A6)

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();

}}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface1)

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());

}}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface2)

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();

 }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=A7)

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 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();

 }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface3)

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();

 }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterface4)

Output:

Hello

Welcome

## **Java 8 Default Method in Interface**

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

*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();

}}

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

Output:

drawing rectangle

default method

## **Java 8 Static Method in Interface**

Since Java 8, we can have static method in interface. Let's see an example:

*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));

}}

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

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, 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{

}

Difference Between Interface and Abstract Class

Last modified on September 7th, 2014 by Joe.

1. Main difference is methods of a Java interface are implicitly abstract and cannot have implementations. A Java abstract class can have instance methods that implements a default behavior.
2. Variables declared in a Java interface is by default final. An  abstract class may contain non-final variables.
3. Members of a Java interface are public by default. A Java abstract class can have the usual flavors of class members like private, protected, etc..
4. Java interface should be implemented using keyword “implements”; A Java abstract class should be extended using keyword “extends”.
5. An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
6. A Java class can implement multiple interfaces but it can extend only one abstract class.
7. Interface is absolutely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
8. In comparison with java abstract classes, java interfaces are slow as it requires extra indirection.

# Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract**methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

* Access modifiers in java

There are two types of modifiers in java.

1 .Access Modifiers

2. Non access modifiers.

The access modifiers in java specifies the accessibility/scope of data member, method, constructor or class.

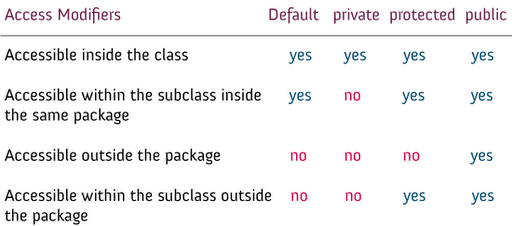
--Private constructor : if you make any class constructor private, you cannot create the instance of that class from outside the class

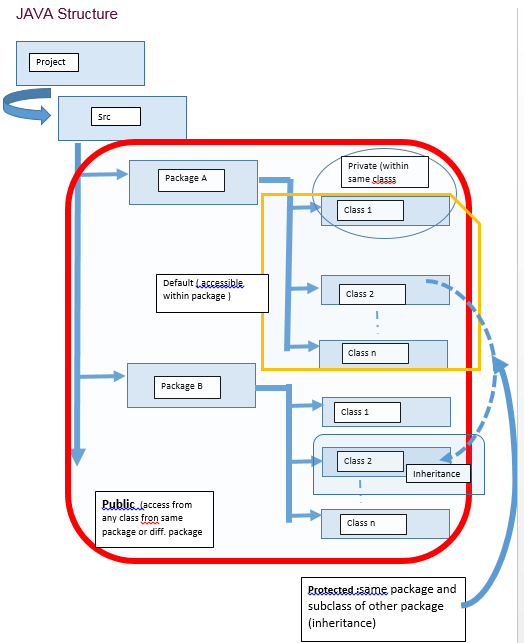
The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

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





### **1. private access modifier**

The private access modifier is specified using the keyword private.

* The methods or data members declared as private are accessible only within the class in which they are declared.
* Any other class of same package will not be able to access these members.
* Top level Classes or interface can not be declared as private because
  1. private means “only visible within the enclosing class”.
  2. protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, they apply only to nested classes and not on top level classes

**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 access modifier**

When no access modifier is specified for a class, method or data member – It is said to be having the default access modifier by default.

* The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are **accessible only within the same package.**
* **Default classes are not public so it cannot be access from outside the package**

|  |
| --- |
| * **With different package**   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.\*;

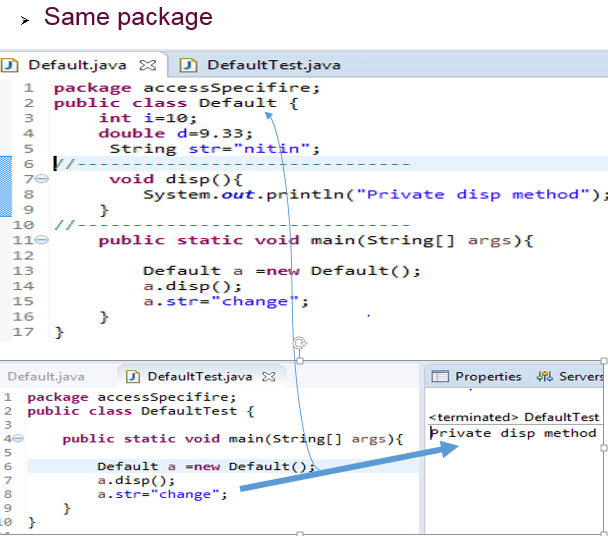
**class** B{

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

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

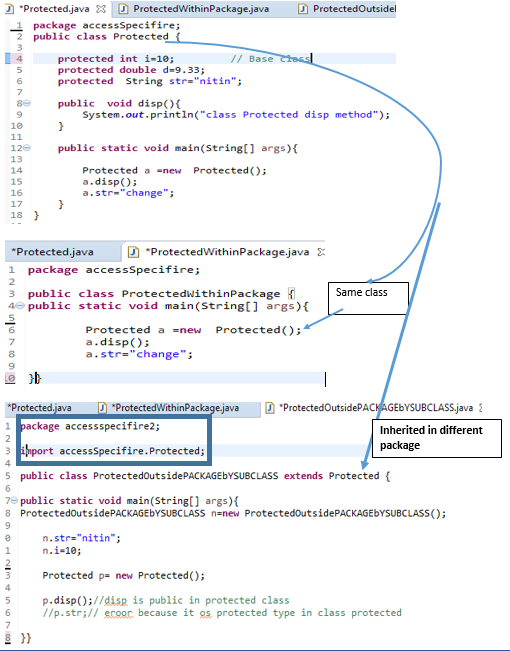
   obj.msg();//Compile Time Error

  }  }



### **3) Protected access modifier**

* 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.



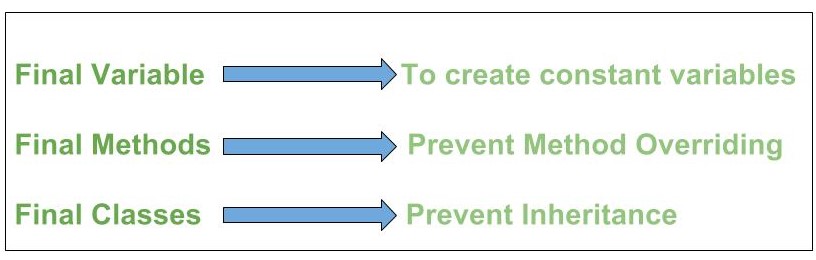
### **4) Public access modifier**

|  |  |
| --- | --- |
| * The **public access modifier** is accessible everywhere. It has the widest scope among all   other modifiers.   1. //save by A.java 3. **package** pack; 4. **public** **class** A{ 5. **public** **void** msg(){System.out.println("Hello");} 6. } 7. //save by B.java 9. **package** mypack; 10. **import** pack.\*; 12. **class** B{ 13. **public** **static** **void** main(String args[]){ 14. A obj = **new** A(); 15. obj.msg(); 16. } 17. }   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.   1. **class** A{ 2. **protected** **void** msg(){System.out.println("Hello java");} 3. } 5. **public** **class** Simple **extends** A{ 6. **void** msg(){System.out.println("Hello java");}//C.T.Error 7. **public** **static** **void** main(String args[]){ 8. Simple obj=**new** Simple(); 9. obj.msg(); 10. } 11. }  |  | | --- | | The default modifier is more restrictive than protected. That is why there is compile time error. |  * **Non-access modifiers :**In java, we have 7 non-access   modifiers. They are used with classes, methods, variables,\ constructors etcto provide  information about their behavior to JVM.They are   * [static](https://www.geeksforgeeks.org/static-keyword-java/) * [final](https://www.geeksforgeeks.org/final-keyword-java/) * abstract * [synchronized](https://www.geeksforgeeks.org/synchronized-in-java/) * [transient](https://www.geeksforgeeks.org/transient-keyword-java/) * [volatile](https://www.geeksforgeeks.org/volatile-keyword-in-java/) * native |

* Final

The final keyword in java is used to restrict the user.

*final* keyword is used in different contexts. First of all, *final* is a [non-access modifier](https://www.geeksforgeeks.org/access-and-non-access-modifiers-in-java/) applicable **only to a variable, a method or a class**.



The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

## **1) Java final variable**

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



1. It is good practice to represent final variables in all uppercase,

2.  If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from [final array](https://www.geeksforgeeks.org/final-arrays-in-java/) or final collection.

3. **Examples :**

// a final variable

final int THRESHOLD = 5;

// a blank final variable

final int THRESHOLD;

// a final static variable PI

static final double PI = 3.141592653589793;

// a blank final static variable

static final double PI;

**Initializing a final variable :**  
We must initialize a final variable, otherwise compiler will throw compile-time error.A final variable can only be initialized once, either via an [initializer](https://www.geeksforgeeks.org/g-fact-26-the-initializer-block-in-java/) or an assignment statement. There are three ways to initialize a final variable :

1. You can initialize a final variable when it is declared. This approach is the most common. A final variable is called **blank final variable**, if it is **not** initialized while declaration. Below are the two ways to initialize a blank final variable.
2. A blank final variable can be initialized inside [instance-initializer block](https://www.geeksforgeeks.org/instance-initialization-block-iib-java/) or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
3. A blank final static variable can be initialized inside [static block](https://www.geeksforgeeks.org/g-fact-79/).

**When to use a final variable :**

The only difference between a normal variable and a final variable is that we can re-assign value to a normal variable but we cannot change the value of a final variable once assigned. Hence final variables must be used only for the values that we want to remain constant throughout the execution of program.

//Java program to demonstrate different

// ways of initializing a final variable

class Gfg

{

    // a final variable

    // direct initialize

    final int THRESHOLD = 5;

    // a blank final variable

    final int CAPACITY;

    // another blank final variable

    final int  MINIMUM;

    // a final static variable PI

    // direct initialize

    static final double PI = 3.141592653589793;

    // a  blank final static  variable

    static final double **EULERCONSTANT;**

    // instance initializer block for

    // initializing CAPACITY

    {

        CAPACITY = 25;

    }

    // static initializer block for

    // initializing EULERCONSTANT

    static{

**EULERCONSTANT** = 2.3;

    }

    // constructor for initializing MINIMUM

    // Note that if there are more than one

    // constructor, you must initialize MINIMUM

    // in them also

    public GFG()

    {

**MINIMUM** = -1;

    }

}

* Whati is diffrence between “ static final” and “final static”

No difference at all.

If two or more (distinct) field modifiers appear in a field declaration, it is customary, though not required, that they appear in the order consistent with that shown above in the production for FieldModifier.

* **Reference final variable :**
* When a final variable is a reference to an object, then this final variable is called reference final variable. For example, a final StringBuffer variable looks like
* final StringBuffer sb;
* As you know that a final variable cannot be re-assign. But in case of a reference final variable, internal state of the object pointed by that reference variable can be changed. Note that this is not re-assigning. This property of final is called non-transitivity. To understand what is mean by internal state of the object, see below example :

|  |
| --- |
| // Java program to demonstrate  // reference final variable    class Gfg  {      public static void main(String[] args)      {          // a final reference variable sb          final StringBuilder sb = new StringBuilder("Geeks");            System.out.println(sb);            // changing internal state of object          // reference by final reference variable sb          sb.append("ForGeeks");            System.out.println(sb);      }  } |

Output:

Geeks

GeeksForGeeks

The non-transitivity property also applies to arrays, because [arrays are objects in java](https://www.geeksforgeeks.org/arrays-in-java/). Arrays with final keyword are also called [final arrays](https://www.geeksforgeeks.org/final-arrays-in-java/).

**Note :**

1. As discussed above, a final variable cannot be reassign, doing it will throw compile-time error.

|  |
| --- |
| // Java program to demonstrate re-assigning  // final variable will throw compile-time error    class Gfg  {      static final int CAPACITY = 4;        public static void main(String args[])      {          // re-assigning final variable          // will throw compile-time error          CAPACITY = 5;      }  } |

Output

Compiler Error: cannot assign a value to final variable CAPACITY

When a final variable is created inside a method/constructor/block, it is called local final variable, and it must initialize once where it is created. See below program for local final variable

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate  // local final variable    // The following program compiles and runs fine    class Gfg  {      public static void main(String args[])      {          // local final variable          final int i;          i = 20;          System.out.println(i);      }  } |

Output:

20

1. Note the difference between C++ *const* variables and Java *final* variables. const variables in C++ must be assigned a value when declared. For final variables in Java, it is not necessary as we see in above examples. A final variable can be assigned value later, but only once.

final with [foreach loop](https://www.geeksforgeeks.org/for-each-loop-in-java/) : final with for-each statement is a legal statement.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate final  // with for-each statement  class Gfg  {      public static void main(String[] args)      {          int arr[] = {1, 2, 3};          // final with for-each statement          // legal statement          for (final int i : arr)              System.out.print(i + " ");      }  } |

Output:

1 2 3

**Explanation :**Since the i variable goes out of scope with each iteration of the loop, it is actually re-declaration each iteration, allowing the same token (i.e. i) to be used to represent multiple variables.

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

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

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

It can be initialized only in constructor.

### **Example of blank final variable**

**class** Student{

**int** id;

String name;

**final** String PAN\_CARD\_NUMBER;

...

}

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

Yes, but only in constructor. For example:

**class** Bike10{

**final** **int** speedlimit;//blank final variable

  Bike10(){

  speedlimit=70;

  System.out.println(speedlimit);

  }

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

**new** Bike10();

 }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike10)

Output: 70

### **static blank final variable**

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

### **Example of static blank final variable**

**class** A{

**static** **final** **int** data;//static blank final variable

**static**{ data=50;}

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

    System.out.println(A.data);

 }

}

### **What is final parameter?**

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

**class** Bike11{

**int** cube(**final** **int** n){

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

   n\*n\*n;

  }

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

    Bike11 b=**new** Bike11();

    b.cube(5);

 }

}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Bike11)

Output: Compile Time Error

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

No, because constructor is never inherited.

## **2) Java final class**

* When a class is declared with final keyword, it is called a final class. A final class cannot be extended(inherited).
* **1. One is definitely to prevent**[**inheritance**](https://www.geeksforgeeks.org/inheritance-in-java/)**, as final classes cannot be extended.**

**final** **class** Bike{}

**class** Honda1 **extends** Bike{

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

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

  Honda1 honda= **new** Honda1();

  honda.run();

  }

}

Output:Compile Time Error

* **The other use of final with classes is to**[**create an immutable class**](https://www.geeksforgeeks.org/create-immutable-class-java/)

**What is Immutable class:** Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like String, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

Following are the requirements:  
• Class must be declared as final (So that child classes can’t be created)  
• Data members in the class must be declared as final (So that we can’t change the value of it after object creation)  
• A parameterized constructor  
• Getter method for all the variables in it  
• No setters(To not have option to change the value of the instance variable)

|  |
| --- |
| // An immutable class  public final class Student  {      final String name;      final int regNo;        public Student(String name, int regNo)      {          this.name = name;          this.regNo = regNo;      }      public String getName()      {          return name;      }      public int getRegNo()      {          return regNo;      }  }    // Driver class  class Test  {      public static void main(String args[])      {          Student s = new Student("ABC", 101);          System.out.println(s.getName());          System.out.println(s.getRegNo());            // Uncommenting below line causes error          // s.regNo = 102;      }  } |

Output :

ABC

101

## 3. Final methods

### **Is final method inherited?**

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

* When a method is declared with *final* keyword, it is called a final method.
* final method cannot be [overridden](https://www.geeksforgeeks.org/overriding-in-java/).
* The [Object](https://www.geeksforgeeks.org/object-class-in-java/) class does this a number of its methods are final.

class A

{

final void m1()

{

System.out.println("This is a final method.");

}

}

class B extends A

{

void m1()

{

// COMPILE-ERROR! Can't override.

System.out.println("Illegal!");

}}

**final vs abstract**

Please see [abstract in java](https://www.geeksforgeeks.org/abstract-keyword-in-java/) article for differences between final and abstract.

Related Interview Question(Important) : [Difference between final, finally and finalize in Java](https://www.geeksforgeeks.org/g-fact-24-finalfinally-and-finalize-in-java/)

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](http://www.contribute.geeksforgeeks.org/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# final vs Immutability in Java

**final :**In Java, [final](https://www.geeksforgeeks.org/final-keyword-java/) is a modifier which is used for class, method and variable also. When a variable is declared with final keyword, it’s value can’t be modified, essentially, a constant.

[**Immutability**](https://www.geeksforgeeks.org/create-immutable-class-java/)**:**In simple terms, immutability means unchanging over time or unable to be changed. In Java, we know that String objects are immutable means we cant change anything to the existing String objects.

**Differences between final and immutability**

* final means that you can’t change the object’s reference to point to another reference or another object, but you can still mutate its state (using setter methods e.g). Whereas immutable means that the object’s actual value can’t be changed, but you can change its reference to another one.
* final modifier is applicable for variable but not for objects, Whereas immutability applicable for an object but not for variables.
* By declaring a reference variable as final, we won’t get any immutability nature, Even though reference variable is final. We can perform any type of change in the corresponding Object. But we cant perform reassignment for that variable.
* final ensures that the address of the object remains the same whereas the Immutable suggests that we can’t change the state of the object once created.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to illustrate  // difference between final  // and immutability    class Geeks {      public static void main(String[] args)      {          final StringBuffer sb = new StringBuffer("Hello");            // Even though reference varibale sb is final          // We can perform any changes          sb.append("GFG");            System.out.println(sb);            // Here we will get Compile time error          // Because reassignment is not possible for final variable          sb = new StringBuffer("Hello World");            System.out.println(sb);      }  } |

Output:

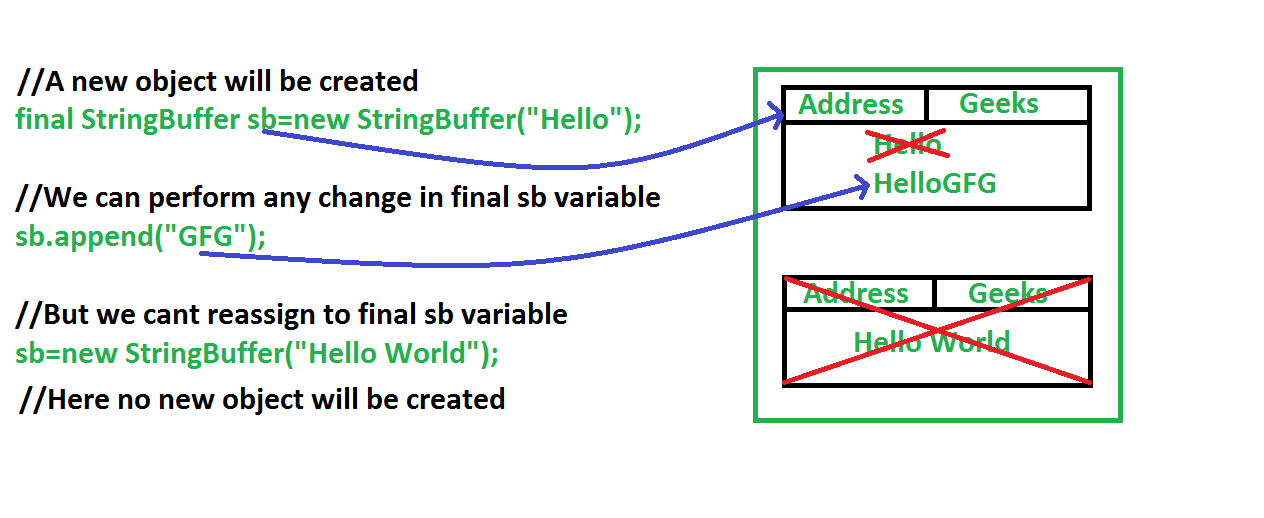
Geeks.java:14: error: cannot assign a value to final variable sb

sb = new StringBuffer("Hello World");

^

1 error

**Pictorial Representation of the above Program**



**Explanation:** In the above picture, we can see that we are creating an object of StringBuffer class by making reference final.

* Declaring reference variable as final, does not mean that the object is immutable.
* In the next line we are performing append() operation on the created object and it is successfully changed.
* If the object is immutable, then the above append operation can’t be done.
* But it is executed successfully as we declare reference variable as final. final means we can’t reassign anything to that reference variable again.
* Therefore when we try to create a new object of BufferedReader then it wont created any object by throwing an error to the console.

Private and final methods in Java

When we use *final* specifier with a method, the method cannot be overridden in any of the inheriting classes. Methods are made final due to design reasons.   
Since private methods are inaccessible, they are implicitly final in Java. So adding *final*specifier to a private method doesn’t add any value. It may in-fact cause unnecessary confusion.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| class Base {       private final void foo() {}       // The above method foo() is same as following. The keyword     // final is redundant in above declaration.       // private void foo() {}  } |

For example, both ‘program 1’ and ‘program 2’ below produce same compiler error “foo() has private access in Base”.

**Program 1**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // file name: Main.java  class Base {      private final void foo() {}  }    class Derived extends Base {      public void foo() {}  }    public class Main {      public static void main(String args[]) {          Base b = new Derived();          b.foo();      }  } |

**Program 2**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // file name: Main.java  class Base {      private void foo() {}  }    class Derived extends Base {      public void foo() {}  }    public class Main {      public static void main(String args[]) {          Base b = new Derived();          b.foo();      }  } |

Please write comments if you find anything incorrect, or you want to share more information about the topic discu

* Abstract

### Keyword:

* abstract

### Applicable to:

1. Class
2. Method

#### Abstract Class:

An abstract Class can have abstract Methods. A Class can also be an abstract class without having any abstract Methods in it. If a Class has an abstract Method, the Class becomes an abstract Class.

#### Abstract Method :

Abstract Methods are those Methods which does not have a body but only a signature.

### **Abstract Class**

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise, a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

**Example**

abstract class Caravan {

private double price;

private String model;

private String year;

public abstract void goFast(); // an abstract method

public abstract void changeColor();

}

### **Abstract Methods**

An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class, unless the subclass is also an abstract class.

If a class contains one or more abstract methods, then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon. Example: public abstract sample();

**Example**

public abstract class SuperClass {

abstract void m(); // abstract method

}

class SubClass extends SuperClass {

// implements the abstract method

void m() {

.........

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Synchronized Non Access Modifier

### Applicable to

1. Method

#### Synchronized Method

Synchronized Methods can be accessed by only one thread at a time.

### **The Synchronized Modifier**

The synchronized keyword used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

**Example**

public synchronized void showDetails() {

.......

}

[Multi-threaded](http://quiz.geeksforgeeks.org/multithreading-in-java/)programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Following is the general form of a synchronized block:

// Only one thread can execute at a time.

// sync\_object is a reference to an object

// whose lock associates with the [monitor](http://quiz.geeksforgeeks.org/monitors/).

// The code is said to be synchronized on

// the monitor object

synchronized(sync\_object)

{

// Access shared variables and other

// shared resources

}

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Following is an example of multi threading with synchronized.

|  |
| --- |
| // A Java program to demonstrate working of  // synchronized.  import java.io.\*;  import java.util.\*;    // A Class used to send a message  class Sender  {      public void send(String msg)      {          System.out.println("Sending\t"  + msg );          try          {              Thread.sleep(1000);          }          catch (Exception e)          {              System.out.println("Thread  interrupted.");          }          System.out.println("\n" + msg + "Sent");      }  }    // Class for send a message using Threads  class ThreadedSend extends Thread  {      private String msg;      private Thread t;      Sender  sender;        // Recieves a message object and a string      // message to be sent      ThreadedSend(String m,  Sender obj)      {          msg = m;          sender = obj;      }        public void run()      {          // Only one thread can send a message          // at a time.          synchronized(sender)          {              // synchronizing the snd object              sender.send(msg);          }      }  }    // Driver class  class SyncDemo  {      public static void main(String args[])      {          Sender snd = new Sender();          ThreadedSend S1 =              new ThreadedSend( " Hi " , snd );          ThreadedSend S2 =              new ThreadedSend( " Bye " , snd );            // Start two threads of ThreadedSend type          S1.start();          S2.start();            // wait for threads to end          try          {              S1.join();              S2.join();          }          catch(Exception e)          {              System.out.println("Interrupted");          }      }  } |

Output:

Sending Hi

Hi Sent

Sending Bye

Bye Sent

### **The Transient Modifier**

* **transient** is a variables modifier used in [serialization](http://quiz.geeksforgeeks.org/serialization-in-java/). At the time of serialization, if we don’t want to save value of a particular variable in a file, then we use **transient** keyword. When JVM comes across **transient**keyword, it ignores original value of the variable and save default value of that variable data type.
* **transient** keyword plays an important role to meet security constraints. There are various real-life examples where we don’t want to save private data in file. Another use of **transient**keyword is not to serialize the variable whose value can be calculated/derived using other serialized objects or system such as age of a person, current date, etc.  
  Practically we serialized only those fields which represent a state of instance, after all serialization is all about to save state of an object to a file. It is good habit to use **transient**keyword with private confidential fields of a class during serialization.
* filter\_none
* edit
* play\_arrow
* brightness\_4

|  |
| --- |
| // A sample class that uses transient keyword to  // skip their serialization.  class Test implements Serializable  {      // Making password transient for security      private transient String password;        // Making age transient as age is auto-      // computable from DOB and current date.      transient int age;        // serialize other fields      private String username, email;      Date dob;        // other code  } |

* **transient and static :**Since **static** fields are not part of state of the object, there is no use/impact of using **transient** keyword with static variables. However there is no compilation error.
* **transient and final :**final variables are directly serialized by their values, so there is no use/impact of declaring final variable as **transient**. There is no compile-time error though.
* filter\_none
* edit
* play\_arrow
* brightness\_4

|  |
| --- |
| // Java program to demonstrate transient keyword  // Filename Test.java  import java.io.\*;  class Test implements Serializable  {      // Normal variables      int i = 10, j = 20;        // Transient variables      transient int k = 30;        // Use of transient has no impact here      transient static int l = 40;      transient final int m = 50;        public static void main(String[] args) throws Exception      {          Test input = new Test();            // serialization          FileOutputStream fos = new FileOutputStream("abc.txt");          ObjectOutputStream oos = new ObjectOutputStream(fos);          oos.writeObject(input);            // de-serialization          FileInputStream fis = new FileInputStream("abc.txt");          ObjectInputStream ois = new ObjectInputStream(fis);          Test output = (Test)ois.readObject();          System.out.println("i = " + output.i);          System.out.println("j = " + output.j);          System.out.println("k = " + output.k);          System.out.println("l = " + output.l);          System.out.println("m = " + output.m);      }  } |

* Output :
* i = 10
* j = 20
* k = 0
* l = 40
* m = 50

### **The Volatile Modifier**

Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

Consider below simple example.

**class** SharedObj

{

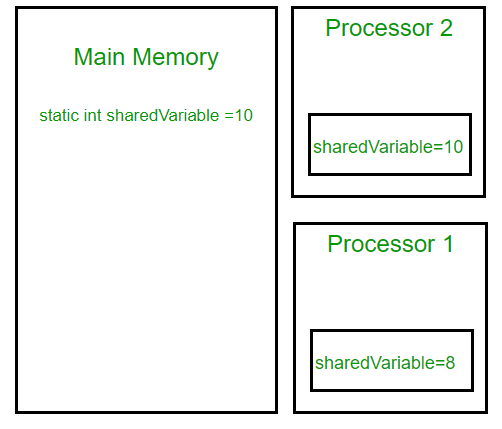
// Changes made to sharedVar in one thread

// may not immediately reflect in other thread

**static int** sharedVar = 6;

}

Suppose that two threads are working on **SharedObj**. If two threads run on different processors each thread may have its own local copy of **sharedVariable**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the [write policy](https://en.wikipedia.org/wiki/CPU_cache#Write_policies) of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

Below diagram shows that if two threads are run on different processors, then value of **sharedVariable** may be different in different threads.  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/volatile-keyword-in-java.png)

Note that write of normal variables without any synchronization actions, might not be visible to any reading thread (this behavior is called [sequential consistency](https://en.wikipedia.org/wiki/Sequential_consistency)). Although most modern hardware provide good cache coherence therefore most probably the changes in one cache are reflected in other but it’s not a good practice to rely on hardware for to ‘fix’ a faulty application.

class SharedObj

{

// volatile keyword here makes sure that

// the changes made in one thread are

// immediately reflect in other thread

static **volatile** int sharedVar = 6;

}

Note that volatile should not be confused with static modifier. static variables are class members that are shared among all objects. There is only one copy of them in main memory.

**volatile vs synchronized:**  
Before we move on let’s take a look at two important features of locks and synchronization.

1. **Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.
2. **Visibility**: It means that changes made by one thread to shared data are visible to other threads.

Java’s synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modifies the value of shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other thread trying to enter the block at the same time will be blocked and put to sleep.

In some cases we may only desire the visibility and not atomicity. Use of synchronized in such situation is an overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, use of volatile is limited to very restricted set of cases as most of the times atomicity is desired. For example a simple increment statement such as x = x + 1; or x++ seems to be a single operation but is s really a compound read-modify-write sequence of operations that must execute atomically.

**volatile in Java vs C/C++:**  
Volatile in java is different from [“volatile” qualifier in C/C++](https://www.geeksforgeeks.org/understanding-volatile-qualifier-in-c/). For Java, “volatile” tells the compiler that the value of a variable must never be cached as its value may change outside of the scope of the program itself. In C/C++, “volatile” is needed when developing embedded systems or device drivers, where you need to read or write a memory-mapped hardware device. The contents of a particular device register could change at any time, so you need the “volatile” keyword to ensure that such accesses aren’t optimized away by the compiler.

# Native keyword in java

The native keyword is applied to a method to indicates that the method is implemented in native code using JNI (Java Native Interface). native is a modifier applicable **only for method**s and we can’t apply it anywhere else. The methods which are implemented in C, C++ are called as native methods or foreign methods.

**The main objective of native keyword are:**

* To improve performance of the system.
* To achieve machine level/memory level communication.
* To use already existing legacy non-java code.

**Pseudo code to use native keyword in java:**

filter\_none

brightness\_4

|  |
| --- |
| Class Native  {      Static      {          System.LoadLibrary(“Native library path”);      }      Public native void m();  }  Class Test  {      Public static void main(String[] args)      {          Native n = new Native();          n.m();      }  } |

**Important points about native keyword:**

* For native methods implementation is already available in old languages like C, C++ and we are not responsible to provide implementation. Hence native method declaration should ends with ;( semi-colon).
* We can’t declare native method as [abstract](https://www.geeksforgeeks.org/abstract-classes-in-java/).

### **We can’t declare native method as**[**strictfp**](https://www.geeksforgeeks.org/strictfp-keyword-java/)**because there is no guarantee that old languages (C, C++) follow IEEE 754 standard. Hence native strictfp combination is illegal combination for methods.**

### **The main advantage of native keyword is performance will be improved but the main disadvantage of native keyword is it breaks platform independent nature of java.**

### Declaring Native Methods**: In this section we explain that how to declare a native method in Java and how to generate the corresponding C/C++ function prototype.**

### 

### **private native String getLine(String prompt);**

### From the Native Language Side:

### **javah -jni Prompt**

### **JNIEXPORT jstring JNICALL Java\_Prompt\_getLine(JNIEnv \*, jobject, jstring);**

### 

# Encapsulation in Java

# **Encapsulation in Java** is a process of wrapping code and data together into a single unit

# Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.

To achieve encapsulation in Java −

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.
* Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
* As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

# https://cdncontribute.geeksforgeeks.org/wp-content/uploads/Encapsulation.jpg

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

1. 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.

2. A class can have total control over what is stored in its fields.

3.Data Hiding: The user will have no idea about the inner implementation of

the class. It will not be visible to the user that how the class is storing values

in the variables.

He only knows that we are passing the values to a setter method and

variables are

getting initialized with that value.

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

### **imple 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

|  |
| --- |
| // Java program to demonstrate encapsulation  public class Encapsulate  {      // private variables declared      // these can only be accessed by      // public methods of class      private String geekName;      private int geekRoll;      private int geekAge;        // get method for age to access      // private variable geekAge      public int getAge()      {        return geekAge;      }        // get method for name to access      // private variable geekName      public String getName()      {        return geekName;      }        // get method for roll to access      // private variable geekRoll      public int getRoll()      {         return geekRoll;      }        // set method for age to access      // private variable geekage      public void setAge( int newAge)      {        geekAge = newAge;      }        // set method for name to access      // private variable geekName      public void setName(String newName)      {        geekName = newName;      }        // set method for roll to access      // private variable geekRoll      public void setRoll( int newRoll)      {        geekRoll = newRoll;      }  } |

In the above program the class EncapsulateDemo is encapsulated as the variables are declared as private. The get methods like getAge() , getName() , getRoll() are set as public, these methods are used to access these variables. The setter methods like setName(), setAge(), setRoll() are also declared as public and are used to set the values of the variables.

The program to access variables of the class EncapsulateDemo is shown below:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| public class TestEncapsulation  {      public static void main (String[] args)      {          Encapsulate obj = new Encapsulate();            // setting values of the variables          obj.setName("Harsh");          obj.setAge(19);          obj.setRoll(51);            // Displaying values of the variables          System.out.println("Geek's name: " + obj.getName());          System.out.println("Geek's age: " + obj.getAge());          System.out.println("Geek's roll: " + obj.getRoll());            // Direct access of geekRoll is not possible          // due to encapsulation          // System.out.println("Geek's roll: " + obj.geekName);      }  } |

Output:

Geek's name: Harsh

Geek's age: 19

Geek's roll: 51

### **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());

}

}

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

Output:

7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0