

Access Modifiers in java

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

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 accessible only within class. |

### **Simple example of private access modifier**

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

**class** A{

**private** **int** data=40;

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

}

**public** **class** Simple{

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

   A obj=**new** A();

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

   obj.msg();//Compile Time Error

   }

}

### Role of Private Constructor

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

**class** A{

**private** A(){}//private constructor

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

}

**public** **class** Simple{

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

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

 }

}

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

### **2) default access modifier**

|  |
| --- |
| If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package. |

### **Example of default access modifier**

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

//save by A.java

**package** pack;

**class** A{

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

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B{

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

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

   obj.msg();//Compile Time Error

  }

}

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

### **3) protected 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.

### **Example of protected access modifier**

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

//save by A.java

**package** pack;

**public** **class** A{

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

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B **extends** A{

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

   B obj = **new** B();

   obj.msg();

  }

}

Output:Hello

### **4) public access modifier**

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

### **Example of public access modifier**

//save by A.java

**package** pack;

**public** **class** A{

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

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B{

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

   A obj = **new** A();

   obj.msg();

  }

}

Output:Hello

### **Understanding all java access modifiers**

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

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

### **Java access modifiers with method overriding**

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

**class** A{

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

}

**public** **class** Simple **extends** A{

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

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

   Simple obj=**new** Simple();

   obj.msg();

   }

}

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

# Encapsulation in Java

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

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

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

#### Advantage of Encapsulation in java

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

### **Simple example of encapsulation in java**

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

//save as Student.java

**package** com.javatpoint;

**public** **class** Student{

**private** String name;

**public** String getName(){

**return** name;

}

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

**this**.name=name

}

}

//save as Test.java

**package** com.javatpoint;

**class** Test{

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

Student s=**new** Student();

s.setname("vijay");

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

}

}

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output: vijay

# Object class in Java

The **Object class** is the parent class of all the classes in java bydefault. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

Object obj=getObject();//we don't what object would be returned from this method

The Object class provides some common behaviours to all the objects such as object can be compared, object can be cloned, object can be notified etc.



### **Methods of Object class**

|  |
| --- |
| The Object class provides many methods. They are as follows: |

|  |  |
| --- | --- |
| Method | Description |
| **public final ClassgetClass()** | returns the Class class object of this object. The Class class can further be used to get the metadata of this class. |
| **public int hashCode()** | returns the hashcode number for this object. |
| **public boolean equals(Object obj)** | compares the given object to this object. |
| **protected Object clone() throws CloneNotSupportedException** | creates and returns the exact copy (clone) of this object. |
| **public String toString()** | returns the string representation of this object. |
| **public final void notify()** | wakes up single thread, waiting on this object's monitor. |
| **public final void notifyAll()** | wakes up all the threads, waiting on this object's monitor. |
| **public final void wait(long timeout)throws InterruptedException** | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| **public final void wait(long timeout,int nanos)throws InterruptedException** | causes the current thread to wait for the specified miliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| **public final void wait()throws InterruptedException** | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| **protected void finalize()throws Throwable** | is invoked by the garbage collector before object is being garbage collected. |

# constructor in javaObject Cloning in Java

The **object cloning** is a way to create exact copy of an object. For this purpose, clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

**protected** Object clone() **throws** CloneNotSupportedException

### Why use clone() method ?

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.

### Advantage of Object cloning

Less processing task.

### **Example of clone() method (Object cloning)**

Let's see the simple example of object cloning

**class** Student18 **implements** Cloneable{

**int** rollno;

String name;

Student18(**int** rollno,String name){

**this**.rollno=rollno;

**this**.name=name;

}

**public** Object clone()**throws** CloneNotSupportedException{

**return** **super**.clone();

}

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

**try**{

Student18 s1=**new** Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

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

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

}**catch**(CloneNotSupportedException c){}

}

}

Output:101 amit

101 amit

As you can see in the above example, both reference variables have the same value. Thus, the clone() copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

# Java Array

Normally, array is a collection of similar type of elements that have contiguous memory location.

**Java array** is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.



### Advantage of Java Array

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
* **Random access:** We can get any data located at any index position.

### Disadvantage of Java Array

* **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

### **Types of Array in java**

There are two types of array.

* Single Dimensional Array
* Multidimensional Array

### **Single Dimensional Array in java**

### Syntax to Declare an Array in java

dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];

### Instantiation of an Array in java

arrayRefVar=**new** datatype[size];

### **Example of single dimensional java array**

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

**class** Testarray{

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

**int** a[]=**new** **int**[5];//declaration and instantiation

a[0]=10;//initialization

a[1]=20;

a[2]=70;

a[3]=40;

a[4]=50;

//printing array

**for**(**int** i=0;i<a.length;i++)//length is the property of array

System.out.println(a[i]);

}}

Output: 10

20

70

40

50

### **Declaration, Instantiation and Initialization of Java Array**

We can declare, instantiate and initialize the java array together by:

**int** a[]={33,3,4,5};//declaration, instantiation and initialization

Let's see the simple example to print this array.

**class** Testarray1{

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

**int** a[]={33,3,4,5};//declaration, instantiation and initialization

//printing array

**for**(**int** i=0;i<a.length;i++)//length is the property of array

System.out.println(a[i]);

}}

Output:33

3

4

5

### **Passing Array to method in java**

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

**class** Testarray2{

**static** **void** min(**int** arr[]){

**int** min=arr[0];

**for**(**int** i=1;i<arr.length;i++)

**if**(min>arr[i])

  min=arr[i];

System.out.println(min);

}

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

**int** a[]={33,3,4,5};

min(a);//passing array to method

}}

Output:3

### **Multidimensional array in java**

In such case, data is stored in row and column based index (also known as matrix form).

### Syntax to Declare Multidimensional Array in java

dataType[][] arrayRefVar; (or)

dataType [][]arrayRefVar; (or)

dataType arrayRefVar[][]; (or)

dataType []arrayRefVar[];

### Example to instantiate Multidimensional Array in java

**int**[][] arr=**new** **int**[3][3];//3 row and 3 column

### Example to initialize Multidimensional Array in java

arr[0][0]=1;

arr[0][1]=2;

arr[0][2]=3;

arr[1][0]=4;

arr[1][1]=5;

arr[1][2]=6;

arr[2][0]=7;

arr[2][1]=8;

arr[2][2]=9;

### **Example of Multidimensional java array**

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

**class** Testarray3{

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

//declaring and initializing 2D array

**int** arr[][]={{1,2,3},{2,4,5},{4,4,5}};

//printing 2D array

**for**(**int** i=0;i<3;i++){

**for**(**int** j=0;j<3;j++){

   System.out.print(arr[i][j]+" ");

 }

 System.out.println();

}

}}

Output:1 2 3

2 4 5

4 4 5

### **What is class name of java array?**

In java, array is an object. For array object, an proxy class is created whose name can be obtained by getClass().getName() method on the object.

**class** Testarray4{

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

**int** arr[]={4,4,5};

Class c=arr.getClass();

String name=c.getName();

System.out.println(name);

}}

Output:I

### **Copying a java array**

We can copy an array to another by the arraycopy method of System class.

### Syntax of arraycopy method

**public** **static** **void** arraycopy(

Object src, **int** srcPos,Object dest, **int** destPos, **int** length

)

### Example of arraycopy method

**class** TestArrayCopyDemo {

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

**char**[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',

                'i', 'n', 'a', 't', 'e', 'd' };

**char**[] copyTo = **new** **char**[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);

        System.out.println(**new** String(copyTo));

    }

}

Output:caffein

### **Addition of 2 matrices in java**

Let's see a simple example that adds two matrices.

**class** Testarray5{

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

//creating two matrices

**int** a[][]={{1,3,4},{3,4,5}};

**int** b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices

**int** c[][]=**new** **int**[2][3];

//adding and printing addition of 2 matrices

**for**(**int** i=0;i<2;i++){

**for**(**int** j=0;j<3;j++){

c[i][j]=a[i][j]+b[i][j];

System.out.print(c[i][j]+" ");

}

System.out.println();//new line

}

}}

Output:2 6 8

6 8 10

# Call by Value and Call by Reference in Java

|  |
| --- |
| There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method. |
|  |

### **Example of call by value in java**

|  |
| --- |
| In case of call by value original value is not changed. Let's take a simple example: |

**class** Operation{

**int** data=50;

**void** change(**int** data){

 data=data+100;//changes will be in the local variable only

 }

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

   Operation op=**new** Operation();

   System.out.println("before change "+op.data);

   op.change(500);

   System.out.println("after change "+op.data);

 }

}

Output:before change 50

after change 50

### **Another Example of call by value in java**

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

**class** Operation2{

**int** data=50;

**void** change(Operation2 op){

 op.data=op.data+100;//changes will be in the instance variable

 }

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

   Operation2 op=**new** Operation2();

   System.out.println("before change "+op.data);

   op.change(op);//passing object

   System.out.println("after change "+op.data);

 }

}

Output:before change 50

after change 150

# strictfp keyword

|  |
| --- |
| The strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic. |

### **Legal code for strictfp keyword**

|  |
| --- |
| The strictfp keyword can be applied on methods, classes and interfaces. |

**strictfp** **class** A{}//strictfp applied on class

**strictfp** **interface** M{}//strictfp applied on interface

**class** A{

**void** m(){}//strictfp applied on method

}

### **Illegal code for strictfp keyword**

|  |
| --- |
| The strictfp keyword can be applied on abstract methods, variables or constructors. |

**class** B{

**strictfp** **abstract** **void** m();//Illegal combination of modifiers

}

**class** B{

**strictfp** **int** data=10;//modifier strictfp not allowed here

}

**class** B{

**strictfp** B(){}//modifier strictfp not allowed here

}

Creating API Document | javadoc tool

We can create document api in java by the help of **javadoc** tool. In the java file, we must use the documentation comment /\*\*... \*/ to post information for the class, method, constructor, fields etc.

Let's see the simple class that contains documentation comment.

**package** com.abc;

/\*\* This class is a user-defined class that contains one methods cube.\*/

**public** **class** M{

/\*\* The cube method prints cube of the given number \*/

**public** **static** **void**  cube(**int** n){System.out.println(n\*n\*n);}

}

To create the document API, you need to use the javadoc tool followed by java file name. There is no need to compile the javafile.

On the command prompt, you need to write:

javadoc M.java

to generate the document api. Now, there will be created a lot of html files. Open the index.html file to get the information about the classes.

# Java Command Line Arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

### **Simple example of command-line argument in java**

|  |
| --- |
| In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt. |

**class** CommandLineExample{

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

System.out.println("Your first argument is: "+args[0]);

}

}

compile by > javac CommandLineExample.java

run by > java CommandLineExample sonoo

Output: Your first argument is: sonoo

### **Example of command-line argument that prints all the values**

|  |
| --- |
| In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop. |

**class** A{

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

**for**(**int** i=0;i<args.length;i++)

System.out.println(args[i]);

}

}

compile by > javac A.java

run by > java A sonoo jaiswal 1 3 abc

Output: sonoo

jaiswal

1

3

abc

# Difference between object and class

There are many differences between object and class. A list of differences between object and class are given below:

|  |  |  |
| --- | --- | --- |
| No. | Object | Class |
| 1) | Object is an **instance** of a class. | Class is a **blueprint or template**from which objects are created. |
| 2) | Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. | Class is a **group of similar objects**. |
| 3) | Object is a **physical** entity. | Class is a **logical** entity. |
| 4) | Object is created through **new keyword** mainly e.g. Student s1=new Student(); | Class is declared using **class keyword** e.g. class Student{} |
| 5) | Object is created **many times** as per requirement. | Class is declared **once**. |
| 6) | Object **allocates memory when it is created**. | Class **doesn't allocated memory when it is created**. |
| 7) | There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization. | There is only **one way to define class** in java using class keyword. |

# Difference between method overloading and method overriding in java

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

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

## **Java Method Overloading example**

**class** OverloadingExample{

**static** **int** add(**int** a,**int** b){

**return** a+b;

}

**static** **int** add(**int** a,**int** b,**int** c){

**return** a+b+c;

}

}

## **Java Method Overriding example**

**class** Animal{

**void** eat(){

System.out.println("eating...");

}

}

**class** Dog **extends** Animal{

**void** eat(){

System.out.println("eating bread...");

}

}