Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.



int[] arr = new int[10];

0-9; arr[0]=1; arr[1] = ‘a’, arr[2]=”Sam”, arr[3]=2.0

int x =arr[1];

### **Advantages**

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

### **Disadvantages**

* **Size Limit:** We can store only the 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 which grows automatically.

### **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**

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];
4. int[] arr = new int[3];
5. int []arr;
6. int arr[]

**Instantiation of an Array in Java**

1. arrayRefVar=**new** datatype[size];

### **Example of Java Array**

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

//Java Program to illustrate how to declare, instantiate, initialize

//and traverse the Java 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;

//traversing array 5

**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

## **For-each Loop for Java Array**

**for**(data\_type variable:array){

//body of the loop

}

## **Passing Array to a Method in Java**

**class** Testarray2{  a{6,2,7,8}

//creating a method which receives an array as a parameter

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

**int** min=arr[0];  //2

**for**(**int** i=1;i<arr.length;i++)

**if**(min>arr[i]) {// 2 >8

  min=arr[i]; // min=2

}

System.out.println(arr[4]);

System.out.println(min);  //3

}

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

**int** a[]={33,3,4,5};//declaring and initializing an array  //0,1,2,3

min(a);//passing array to method

}}

Output:

3

## **ArrayIndexOutOfBoundsException**

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

## **2) Multidimensional Array in Java(2-D array)**

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

**Syntax to Declare Multidimensional Array in Java**

1. dataType[][] arrayRefVar; (or)  int[][] arr;
2. dataType [][]arrayRefVar; (or)  int [][]arr;
3. dataType arrayRefVar[][]; (or)   int arr[][];
4. dataType []arrayRefVar[];   int []arr[];

**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**

//Java Program to illustrate the use of multidimensional array

**class** Testarray3{

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

//declaring and initializing 2D array

**int** arr[][]={{1,2,3},//0

{2,4,5},//1

{4,4,5}};  //2

//0,1,2

//printing 2D array

**for**(**int** i=0;i<3;i++){

**for**(**int** j=0;j<3;j++){

   System.out.print(arr[i][j]+" ");  //0,0//0,1///0,2////123

 }

 System.out.println();

}

}}

Output:

1 2 3

2 4 5

4 4 5

## **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

)

EX:

**class** TestArrayCopyDemo {

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

        //declaring a source array

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

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

        //declaring a destination array

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

**copytp[0]=’c’**

        //copying array using System.arraycopy() method

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

        //printing the destination array

        System.out.println(String.valueOf(copyTo));  }  }

Output:

caffein

## **Cloning an Array in Java**

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

1. //Java Program to clone the array
2. **class** Testarray1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. System.out.println("Printing original array:");
6. **for**(**int** i:arr)
7. System.out.println(i);
9. System.out.println("Printing clone of the array:");
10. **int** carr[]=arr.clone();  ///{33,3,4,5}
11. **for**(**int** i:carr)
12. System.out.println(i);
14. System.out.println("Are both equal?");
15. System.out.println(arr==carr);
17. }}

Output:

Printing original array:

33

3

4

5

Printing clone of the array:

33

3

4

5

Are both equal?

false

## **Addition of 2 Matrices in Java**

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

//Java Program to demonstrate the addition of two matrices in Java

**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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Polymorphism**

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

### **Example**

Let us look at an example.

public interface Vegetarian{}

public class Animal{}

public class Deer extends Animal implements Vegetarian{}

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples −

* A Deer IS-A Animal
* A Deer IS-A Vegetarian
* A Deer IS-A Deer
* A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal −

### **Example**

Deer d = new Deer();

Animal a = d;

Vegetarian v = d;

Object o = d;

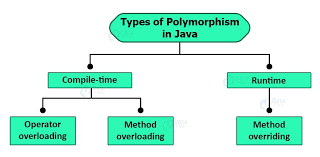
All the reference variables d, a, v, o refer to the same Deer object in the heap.

In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

**Types of polymorphism**

In Java polymorphism is mainly divided into two types:

* Compile-time Polymorphism
* Runtime Polymorphism



**Type 1:**Compile-time polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

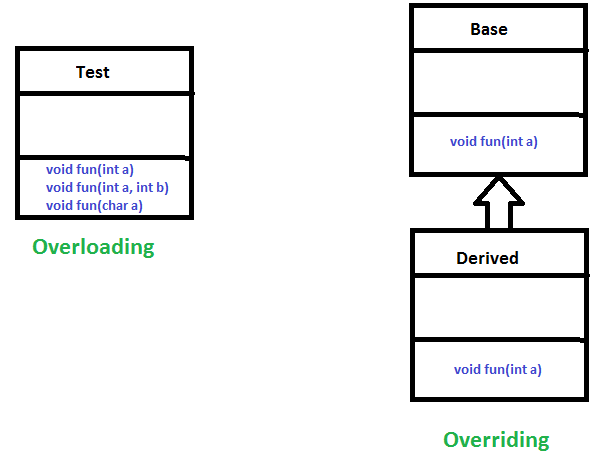
***Note:****But Java doesn’t support the Operator Overloading.*

*method(int o, int a);// arg:2 method(1,2);*

*method(int o);//arg 1*

*method(int a, int b, int c);//3*

*method(int a, String b);//2 //compile error// method(1,”Train”)*



**Method Overloading**: When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.

**Example 1**

IDE1, IDE2

**Type 2:**[Runtime polymorphism](https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/)

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. [**Method overriding**](https://www.geeksforgeeks.org/overriding-in-java/), on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

**Example**

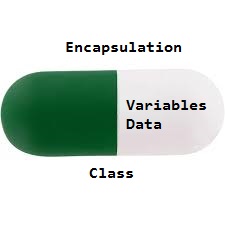
IDE

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Encapsulation in Java**

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield. 

* 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 its own class in which it is declared.
* As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a **combination of data-hiding and abstraction**.
* Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables
* It is more defined with setter and getter method.



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.

**Advantages of Encapsulation**:

* **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
* **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirement. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
* **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
* **Testing code is easy:** Encapsulated code is easy to test for unit testing.