# Arrays in Java

**Array in java** is a group of like-typed variables referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.

* In Java, all arrays are dynamically allocated. (discussed below).
* Arrays are stored in contiguous memory [consecutive memory locations].
* Since arrays are objects in Java, we can find their length using the object property *length*. This is different from C/C++, where we find length using sizeof.
* A Java array variable can also be declared like other variables with [] after the data type.
* The variables in the array are ordered, and each has an index beginning with 0.
* Java array can also be used as a static field, a local variable, or a method parameter.
* The **size** of an array must be specified by int or short value and not long.
* The direct superclass of an array type is **Object**.
* Every array type implements the interfaces Cloneable and **java.io.Serializable**.
* This storage of arrays helps us randomly access the elements of an array [Support Random Access].
* The size of the array cannot be altered(once initialized).  However, an array reference can be made to point to another array.

An array can contain primitives (int, char, etc.) and object (or non-primitive) references of a class depending on the definition of the array. In the case of primitive data types, the actual values are stored in contiguous memory locations. In the case of class objects, **the actual objects are stored in a heap segment**.



### **Creating, initializing, and accessing an Array**

### **One-Dimensional Arrays:**

The general form of a one-dimensional array declaration is

type var-name[];

OR

type[] var-name;

An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc., or user-defined data types (objects of a class). Thus, the element type for the array determines what type of data the array will hold.

**Example:**

// both are valid declarations

int intArray[];

or int[] intArray;

byte byteArray[];

short shortsArray[];

boolean booleanArray[];

long longArray[];

float floatArray[];

double doubleArray[];

char charArray[];

// an array of references to objects of

// the class MyClass (a class created by

// user)

MyClass myClassArray[];

Object[] ao, // array of Object

Collection[] ca; // array of Collection

// of unknown type

Although the first declaration establishes that int Array is an array variable, **no actual array exists**. It merely tells the compiler that this variable (int Array) will hold an array of the integer type. To link int Array with an actual, physical array of integers, you must allocate one using **new** and assign it to int Array.

### **Instantiating an Array in Java**

When an array is declared, only a reference of an array is created. To create or give memory to the array, you create an array like this: The general form of *new* as it applies to one-dimensional arrays appears as follows:

var-name = new type [size];

Here, *type* specifies the type of data being allocated, *size* determines the number of elements in the array, and *var-name* is the name of the array variable that is linked to the array. To use *new* to allocate an array, **you must specify the type and number of elements to allocate.**

**Example:**

int intArray[]; //declaring array

intArray = new int[20]; // allocating memory to array

OR

int[] intArray = new int[20]; // combining both statements in one

***Note:***

*The elements in the array allocated by new will automatically be initialized to****zero****(for numeric types),****false****(for boolean), or****null****(for reference types). Do refer to****default array values in Java****.*

*Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory to hold the array, using new, and assign it to the array variable. Thus,****in Java****,****all arrays are dynamically allocated.***

### **Array Literal**

In a situation where the size of the array and variables of the array are already known, array literals can be used.

int[] intArray = new int[]{ 1,2,3,4,5,6,7,8,9,10 };

// Declaring array literal

* The length of this array determines the length of the created array.
* There is no need to write the new int[] part in the latest versions of Java.

### **Accessing Java Array Elements using for Loop**

Each element in the array is accessed via its index. The index begins with 0 and ends at (total array size)-1. All the elements of array can be accessed using Java for Loop.

// accessing the elements of the specified array

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

System.out.println("Element at index " + i +

" : "+ arr[i]);

**Implementation:**

|  |
| --- |
| // Java program to illustrate creating an array  // of integers,  puts some values in the array,  // and prints each value to standard output.    **class** Arrays {  **public** **static** **void** main(String[] args)      {          // declares an Array of integers.  **int**[] arr;            // allocating memory for 5 integers.          arr = **new** **int**[5];            // initialize the first elements of the array          arr[0] = 10;            // initialize the second elements of the array          arr[1] = 20;            // so on...          arr[2] = 30;          arr[3] = 40;          arr[4] = 50;            // accessing the elements of the specified array  **for** (**int** i = 0; i < arr.length; i++)              System.out.println("Element at index " + i                                 + " : " + arr[i]);      }  } |

**Output**

Element at index 0 : 10

Element at index 1 : 20

Element at index 2 : 30

Element at index 3 : 40

Element at index 4 : 50

**You can also access java arrays using for each loops.**

For-each is another array traversing technique like for loop, while loop, do-while loop introduced in Java5. 

* It starts with the keyword **for** like a normal for-loop.
* Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
* In the loop body, you can use the loop variable you created rather than using an indexed array element.
* It’s commonly used to iterate over an array or a Collections class (eg, ArrayList)

**Syntax:**

for (type var : array)

{

statements using var;

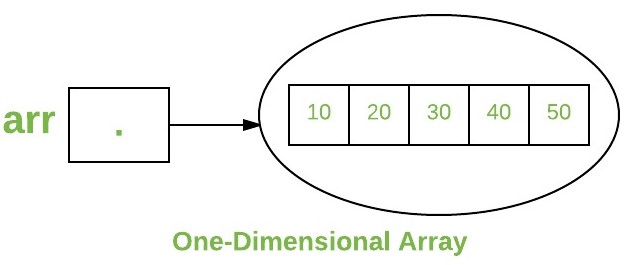
}

**Simple program with for each loop:**

|  |
| --- |
| /\*package whatever //do not write package name here \*/    **import** java.io.\*;    **class** Easy    {    **public** **static** **void** main(String[] args)        {            // array declaration    **int** ar[] = { 10, 50, 60, 80, 90 };    **for** (**int** element : ar)                System.out.print(element + " ");      }  } |

**Output**

10 50 60 80 90



### **Arrays of Objects**

An array of objects is created like an array of primitive-type data items in the following way.

Student[] arr = new Student[5]; //student is a user-defined class

**Syntax:**

1) data type[] arrName;

2) datatype arrName[];

3) datatype [] arrName;

The student Array contains five memory spaces each of the size of student class in which the address of five Student objects can be stored. The Student objects have to be instantiated using the constructor of the Student class, and their references should be assigned to the array elements in the following way.

**Example**

|  |
| --- |
| // Java program to illustrate creating  //  an array of objects    **class** Student {  **public** **int** roll\_no;  **public** String name;      Student(**int** roll\_no, String name)      {  **this**.roll\_no = roll\_no;  **this**.name = name;      }    // Elements of the array are objects of a class Student.  **public** **static** **void** main(String[] args)      {          // declares an Array of integers.          Student[] arr;            // allocating memory for 5 objects of type Student.          arr = **new** Student[5];            // initialize the first elements of the array          arr[0] = **new** Student(1, "aman");            // initialize the second elements of the array          arr[1] = **new** Student(2, "vaibhav");            // so on...          arr[2] = **new** Student(3, "shikar");          arr[3] = **new** Student(4, "dharmesh");          arr[4] = **new** Student(5, "mohit");            // accessing the elements of the specified array  **for** (**int** i = 0; i < arr.length; i++)              System.out.println("Element at " + i + " : "                                 + arr[i].roll\_no + " "                                 + arr[i].name);      }  } |

**Output**

Element at 0 : 1 aman

Element at 1 : 2 vaibhav

Element at 2 : 3 shikar

Element at 3 : 4 dharmesh

Element at 4 : 5 mohit

### **What happens if we try to access elements outside the array size?**

JVM throws **ArrayIndexOutOfBoundsException** to indicate that the array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of an array.

Below code shows what happens if we try to access elements outside the array size.

|  |
| --- |
| // Code for showing error "ArrayIndexOutOfBoundsException"    **public** **class** AIOB {  **public** **static** **void** main(String[] args)      {  **int**[] arr = **new** **int**[4];          arr[0] = 10;          arr[1] = 20;          arr[2] = 30;          arr[3] = 40;            System.out.println(              "Trying to access element outside the size of array");          System.out.println(arr[5]);      }  } |

**Output**

Trying to access element outside the size of array

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 4

at GFG.main(GFG.java:13)

|  |
| --- |
| **public** **class** AIOBA {  **public** **static** **void** main(String[] args)      {  **int**[] arr = **new** **int**[2];          arr[0] = 10;          arr[1] = 20;    **for** (**int** i = 0; i < arr.length; i++)              System.out.println(arr[i]);      }  } |

**Output**

10

20

### **Multidimensional Arrays:**

Multidimensional arrays are **arrays of arrays** with each element of the array holding the reference of other arrays. These are also known as **Jagged Arrays**. A multidimensional array is created by appending one set of square brackets ([]) per dimension.

**Syntax :**

datatype [][] arrayrefvariable;

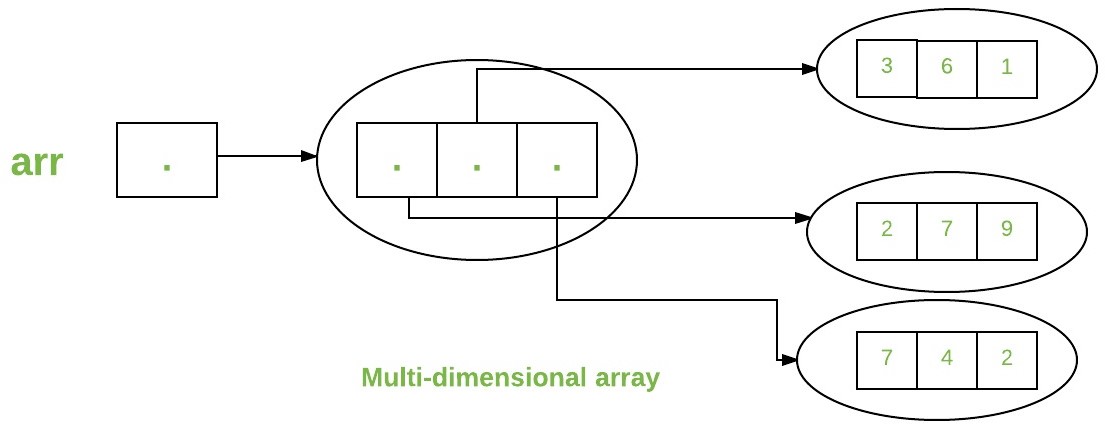
or

datatype arrayrefvariable[][];

**Syntax:**

|  |
| --- |
| **import** java.io.\*;    **class** MDA {  **public** **static** **void** main (String[] args) {        // Syntax  **int** [][] arr= **new** **int**[3][3];        // 3 row and 3 column      }  } |

**Output**



int[][] intArray = new int[10][20]; //a 2D array or matrix

int[][][] intArray = new int[10][20][10]; //a 3D array

**Example**

|  |
| --- |
| **public** **class** multiDimensional {  **public** **static** **void** main(String args[])      {          // declaring and initializing 2D array  **int** arr[][]              = { { 2, 7, 9 }, { 3, 6, 1 }, { 7, 4, 2 } };            // 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**

2 7 9

3 6 1

7 4 2

## **Three-Dimensional Arrays In Java**

We already discussed that an array gets more complex as their dimensions increase. Three-dimensional arrays are complex for multi-dimensional arrays. A three dimensional can be defined as an array of two-dimensional arrays.

**The general definition of a Three-dimensional array is given below:**

data\_type [] [] [] array\_name = new data\_type [d1][d2][d3];

**Here,**

d1, d2, d3 = sizes of the dimensions  
data\_type = data type of the elements of the array  
array\_name = name of the 3d array

**Example of 3d array definition is:**

int [] [] [] intArray = new int[2][3][4];

The above definition of 3d array can be interpreted as having 2 tables or arrays, 3 rows and 4 columns that totals up to 2x3x4 = 24 elements.

**This means that in a 3d array, the three dimensions are interpreted as:**

* **The number of Tables/Arrays:** The first dimension indicates how many tables or arrays a 3d array will have.
* **The number of Rows:** The second dimension signifies the total number of rows an array will have.
* **The number of Columns:** The third dimension indicates the total columns in the 3d array.

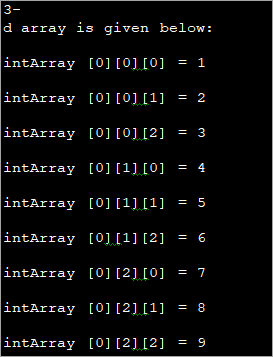
### **Initialize 3d Array**

The approaches used to initialize a 3d array are the same as the ones used for initializing Two-dimensional arrays.

You can either initialize the array by assigning values to individual array elements or initialize the array during the declaration.

|  |
| --- |
| **public** **class** Main  {  **public** **static** **void** main(String[] args) {            //initialize 3-d array  **int**[][][] intArray = { { { 1, 2, 3}, { 4, 5, 6 } ,  { 7, 8, 9 } } };          System.out.println ("3-d array is given below :");          //print the elements of array  **for** (**int** i = 0; i < 1; i++)  **for** (**int** j = 0; j < 3; j++)  **for** (**int** z = 0; z < 3; z++)                  System.out.println ("intArray [" + i              + "][" + j + "][" + z + "] = " + intArray [i][j][z]);      }  } |

**Output:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/Initialization-of-the-3d-array-while-declaration.png)

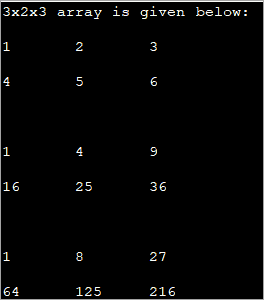
After initializing the 3d array during declaration, we have accessed the individual elements of the array and printed them.

### **Acces And Print 3d Array**

Again, printing and accessing array elements in a three-dimensional array is similar to that in two-dimensional arrays.

|  |
| --- |
| **public** **class** Main  {  **public** **static** **void** main(String[] args) {          //initialize 3-d array  **int**[][][] myArray = { { { 1, 2, 3 }, { 4, 5, 6 } },  { { 1, 4, 9 },  { 16, 25, 36 } },                  { { 1, 8, 27 }, { 64, 125, 216 } } };          System.out.println("3x2x3 array is given below:");          //print the 3-d array  **for** (**int** i = 0; i < 3; i++) {  **for** (**int** j = 0; j < 2; j++) {  **for** (**int** k = 0; k < 3; k++) {          System.out.print(myArray[i][j][k] + "\t");          }          System.out.println();                      }          System.out.println();          }      }  } |

**Output:**

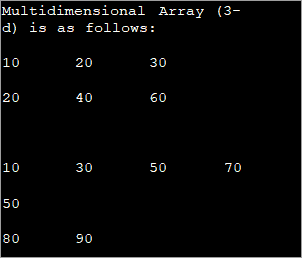
[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/For-loops-to-access-the-array-elements.png)

The above program displays a tabular representation of a three-dimensional array. As shown, it is a 3x2x3 array which means that it has 3 tables, 2 rows and 3 columns and thus 18 elements.

It is already mentioned that the column size can vary in a multi-dimensional array. The example below demonstrates a three-dimensional array with varied column sizes.

|  |
| --- |
| **public** **class** Main  {  **public** **static** **void** main(String[] args) {          //initialize 3-d array  **int**[][][] intArray = {                {{10, 20, 30},{20, 40, 60}},                { {10, 30,50,70},{50},{80, 90}}                };      System.out.println("Multidimensional Array (3-d) is as follows:");      // use for..each loop to iterate through elements of 3d array  **for** (**int**[][] array\_2D: intArray) {  **for** (**int**[] array\_1D: array\_2D) {  **for**(intelem: array\_1D) {              System.out.print(elem + "\t");               }             System.out.println();          }          System.out.println();      }      }  } |

**Output:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/For-loop-to-traverse-through-the-array.png)

The input array used is a Three-dimensional array with a varied length of columns. The enhanced for each loop used for each dimension displays the contents of the array in a tabular format.

### **Passing Arrays to Methods**

Like variables, we can also pass arrays to methods. For example, the below program passes the array to method *sum* to calculate the sum of the array’s values.

|  |
| --- |
| // Java program to demonstrate  // passing of array to method    **public** **class** PATM {      // Driver method  **public** **static** **void** main(String args[])      {  **int** arr[] = { 3, 1, 2, 5, 4 };            // passing array to method m1          sum(arr);      }    **public** **static** **void** sum(**int**[] arr)      {          // getting sum of array values  **int** sum = 0;    **for** (**int** i = 0; i < arr.length; i++)              sum += arr[i];            System.out.println("sum of array values : " + sum);      }  } |

**Output**

sum of array values : 15

### **Returning Arrays from Methods**

As usual, a method can also return an array. For example, the below program returns an array from method *m1*.

|  |
| --- |
| // Java program to demonstrate  // return of array from method    **class** RAFM {      // Driver method  **public** **static** **void** main(String args[])      {  **int** arr[] = m1();    **for** (**int** i = 0; i < arr.length; i++)              System.out.print(arr[i] + " ");      }    **public** **static** **int**[] m1()      {          // returning  array  **return** **new** **int**[] { 1, 2, 3 };      }  } |

**Output**

1 2 3

### **Class Objects for Arrays**

Every array has an associated Class object, shared with all other arrays with the same component type.

|  |
| --- |
| // Java program to demonstrate  // Class Objects for Arrays    **class** COFA {  **public** **static** **void** main(String args[])      {  **int** intArray[] = **new** **int**[3];  **byte** byteArray[] = **new** **byte**[3];  **short** shortsArray[] = **new** **short**[3];            // array of Strings          String[] strArray = **new** String[3];            System.out.println(intArray.getClass());          System.out.println(              intArray.getClass().getSuperclass());          System.out.println(byteArray.getClass());          System.out.println(shortsArray.getClass());          System.out.println(strArray.getClass());      }  } |

**Output**

class [I

class java.lang.Object

class [B

class [S

class [Ljava.lang.String;

**Explanation:**

1. The string “[I” is the run-time type signature for the class object “array with component type *int*.”
2. The only direct superclass of an array type is **java.lang.Object**.
3. The string “[B” is the run-time type signature for the class object “array with component type *byte*.”
4. The string “[S” is the run-time type signature for the class object “array with component type *short*.”
5. The string “[L” is the run-time type signature for the class object “array with component type of a Class.” The Class name is then followed.

### **Array Members**

Now, as you know that arrays are objects of a class, and a direct superclass of arrays is a class Object. The members of an array type are all of the following:

* The public final field *length* contains the number of components of the array. Length may be positive or zero.
* All the members are inherited from class Object; the only method of Object that is not inherited is its **clone** method.
* The public method *clone()* overrides the clone method in class Object and throws no **checked exceptions**.

### **Arrays Types and Their Allowed Element Types**

| **Array Types** | **Allowed Element Types** |
| --- | --- |
| Primitive Type Arrays | Any type which can be implicitly promoted to declared type. |
| Object Type Arrays | Either declared type objects or it’s child class objects. |
| Abstract Class Type Arrays | Its child-class objects are allowed. |
| Interface Type Arrays | Its implementation class objects are allowed. |