Java - Arrays

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable −

Syntax

```
dataType[] arrayRefVar;   // preferred way.
or
dataType arrayRefVar[];  // works but not preferred way.
```

Note − The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax −

```
double[] myList;   // preferred way.
or
double myList[];   // works but not preferred way.
```

Creating Arrays

You can create an array by using the new operator with the following syntax −

Syntax

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things −

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below −

```
dataType[] arrayRefVar = new dataType[arraySize];
```

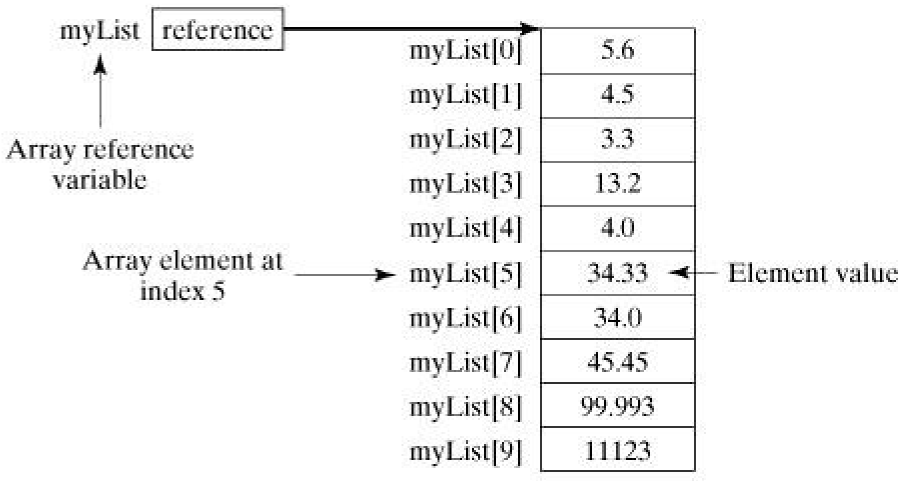
Alternatively you can create arrays as follows −

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList −
double[] myList = new double[10];
Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays
When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.
Example
Here is a complete example showing how to create, initialize, and process arrays −

```
public class TestArray {

  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (int i = 0; i < myList.length; i++) {
      System.out.println(myList[i] + " ");
    }

    // Summing all elements
    double total = 0;
    for (int i = 0; i < myList.length; i++) {
      total += myList[i];
    }
    System.out.println("Total is " + total);

    // Finding the largest element
    double max = myList[0];
    for (int i = 1; i < myList.length; i++) {
      if (myList[i] > max) max = myList[i];
    }
    System.out.println("Max is " + max);
```

```
   }
}
```

This will produce the following result −

Output

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

The foreach Loops

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example

The following code displays all the elements in the array myList −

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // Print all the array elements
      for (double element: myList) {
         System.out.println(element);
      }
   }
}
```

This will produce the following result −

Output

```
1.9
2.9
3.4
3.5
```

Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an **int** array −

Example

```
public static void printArray(int[] array) {
   for (int i = 0; i < array.length; i++) {
      System.out.print(array[i] + " ");
   }
}
```

Returning an Array from a Method

A method may also return an array. For example, the following method returns an array that is the reversal of another array −

Example

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
    result[j] = list[i];
  }
  return result;
}
```

The Arrays Class

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

public static boolean equals(long[] a, long[] a2)

Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.)

public static void sort(Object[] a)

Sorts the specified array of objects into an ascending order, according to the natural ordering of its elements. The same method could be used by all other primitive data types (Byte, short, Int, etc.)

Java - Numbers Class

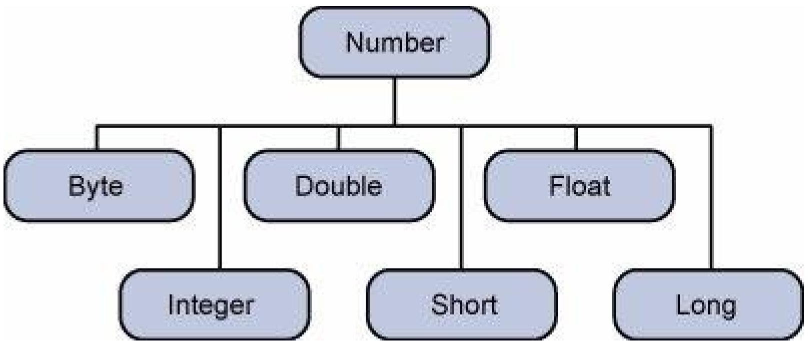
Normally, when we work with Numbers, we use primitive data types such as byte, int, long, double, etc.

Example

```
int i = 5000;
float gpa = 13.65;
double mask = 0xaf;
```

However, in development, we come across situations where we need to use objects instead of primitive data types. In order to achieve this, Java provides **wrapper classes**.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.



The object of the wrapper class contains or wraps its respective primitive data type. Converting primitive data types into object is called **boxing**, and this is taken care by the compiler. Therefore, while using a wrapper class you just need to pass the value of the primitive data type to the constructor of the Wrapper class.

And the Wrapper object will be converted back to a primitive data type, and this process is called unboxing. The **Number** class is part of the java.lang package.

Following is an example of boxing and unboxing −

Example

```
public class Test {

   public static void main(String args[]) {
      Integer x = 5; // boxes int to an Integer object
      x =  x + 10;   // unboxes the Integer to a int
      System.out.println(x);
   }
}
```

This will produce the following result −

Output

```
15
```

When x is assigned an integer value, the compiler boxes the integer because x is integer object. Later, x is unboxed so that they can be added as an integer.

Number Methods

Following is the list of the instance methods that all the subclasses of the Number class implements −

Sr.No.	Method & Description
1	xxxValue() Converts the value of *this* Number object to the xxx data type and returns it.
2	compareTo()

		Compares *this* Number object to the argument.
3	equals()	Determines whether *this* number object is equal to the argument.
4	valueOf()	Returns an Integer object holding the value of the specified primitive.
5	toString()	Returns a String object representing the value of a specified int or Integer.
6	parseInt()	This method is used to get the primitive data type of a certain String.
7	abs()	Returns the absolute value of the argument.
8	ceil()	Returns the smallest integer that is greater than or equal to the argument. Returned as a double.
9	floor()	Returns the largest integer that is less than or equal to the argument. Returned as a double.
10	rint()	Returns the integer that is closest in value to the argument. Returned as a double.
11	round()	Returns the closest long or int, as indicated by the method's return type to the argument.
12	min()	Returns the smaller of the two arguments.
13	max()	Returns the larger of the two arguments.
14	exp()	Returns the base of the natural logarithms, e, to the power of the argument.
15	log()	Returns the natural logarithm of the argument.
16	pow()	Returns the value of the first argument raised to the power of the second argument.
17	sqrt()	

		Returns the square root of the argument.
18	sin()	Returns the sine of the specified double value.
19	cos()	Returns the cosine of the specified double value.
20	tan()	Returns the tangent of the specified double value.
21	asin()	Returns the arcsine of the specified double value.
22	acos()	Returns the arccosine of the specified double value.
23	atan()	Returns the arctangent of the specified double value.
24	atan2()	Converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta.
25	toDegrees()	Converts the argument to degrees.
26	toRadians()	Converts the argument to radians.
27	random()	Returns a random number.