**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};**

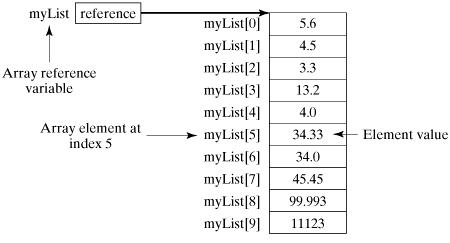
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

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] + " ");

}

}

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2 −

Example

printArray(new int[]{3, 1, 2, 6, 4, 2});

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.

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public static int binarySearch(Object[] a, Object key)**  Searches the specified array of Object ( Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, it returns ( – (insertion point + 1)). |
| 2 | **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.) |
| 3 | **public static void fill(int[] a, int val)**  Assigns the specified int value to each element of the specified array of ints. The same method could be used by all other primitive data types (Byte, short, Int, etc.) |
| 4 | **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.) |

**One D Array:**

You can either use array declaration or array literal (but only when you declare and affect the variable right away, array literals cannot be used for re-assigning an array).

For primitive types:

int[] myIntArray = new int[3];

int[] myIntArray = {1,2,3};

int[] myIntArray = new int[]{1,2,3};

For classes, for example String, it's the same:

String[] myStringArray = new String[3];

String[] myStringArray = {"a","b","c"};

String[] myStringArray = new String[]{"a","b","c"};

The third way of initialising is useful when you declare the array first and then initialise it. Cast is necessary here.

String[] myStringArray;

myStringArray = new String[]{"a","b","c"};

A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

type var-name[ ];

Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named **month\_days**with the type "array of int":

int month\_days[];

Although this declaration establishes the fact that **month\_days**is an array variable, no array actually exists. In fact, the value of **month\_days**is set to **null**, which represents an array with no value. To link **month\_days**with an actual, physical array of integers, you must allocate one using **new**and assign it to **month\_days**. **new**is a special operator that allocates memory.

You will look more closely at **new**in a later chapter, but you need to use it now to allocate memory for arrays. The general form of **new**as it applies to one-dimensional arrays appears as follows:

array-var = new type[size];

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array. That is, to use **new**to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new**will automatically be initialized to zero. This example allocates a 12-element array of integers and links them to **month\_days**.

month\_days = new int[12];

After this statement executes, **month\_days**will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Let's review: 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 that will hold the array, using **new**, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **month\_days**.

month\_days[1] = 28;

The next line displays the value stored at index 3.

System.out.println(month\_days[3]);

Putting together all the pieces, here is a program that creates an array of the number of days in each month.

**// Demonstrate a one-dimensional array.   
class Array {   
public static void main(String args[]) {   
int month\_days[];   
month\_days = new int[12];   
month\_days[0] = 31;   
month\_days[1] = 28;   
month\_days[2] = 31;   
month\_days[3] = 30;   
month\_days[4] = 31;   
month\_days[5] = 30;**

**month\_days[6] = 31;   
month\_days[7] = 31;   
month\_days[8] = 30;   
month\_days[9] = 31;   
month\_days[10] = 30;   
month\_days[11] = 31;   
System.out.println("April has " + month\_days[3] + " days.");   
}   
}**

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is **month\_days[3]**or 30. It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

int month\_days[] = new int[12];

This is the way that you will normally see it done in professionally written Java programs. Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**. For example, to store the number of days in each month, the following code creates an initialized array of integers:

**// An improved version of the previous program.   
class AutoArray {   
public static void main(String args[]) {   
int month\_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,   
30, 31 };   
System.out.println("April has " + month\_days[3] + " days.");   
}   
}**

When you run this program, you see the same output as that generated by the previous version.   
Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range. (In this regard, Java is fundamentally different from C/C++, which provide no run-time boundary checks.) For example, the run-time system will check the value of each index into**month\_days**to make sure that it is between 0 and 11 inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a runtime error.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

**// Average an array of values.   
class Average {   
public static void main(String args[]) {   
double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};   
double result = 0;   
int i;   
for(i=0; i<5; i++)   
result = result + nums[i];   
System.out.println("Average is " + result / 5);   
}   
}**

**Processing Arrays using Loops**

It would be quite difficult to process each of the array's elements individually as we have done in the previous chapter. Since these are repetitive tasks, repetition statements like for and while come to our rescue. The common approach is to use a for loop with a variable used to keep track of the index number and iterate through the entire array.

For example, the following code stores the numbers from 1 to 7 in a seven element int array.

int[] a =new int[7];  
for ( int i=0; i<a.length; i++) {  
    a[i]= i+1;  
}

In a similar way, a for loop can be used to print the elements of the array.

for ( int i=0; i<a.length; i++) {  
    System.out.println("Element at index "+i+" is  "+a[i]);  
}

And the code below stores the first seven even numbers in the array a, starting from 2.

for ( int i=0; i<a.length; i++) {  
    a[i] = (i+1) \* 2;  
}

Here are a few more example which use loops to process arrays.

examples

We may also use the enhanced for loop ( also known as the for each loop) to access the elements of the array in a more convenient way. The for each loop has the following syntax.

for ( <data type> <variable name>:< array name>) {  
    // code  
}

The data type should be the same as the data type of the variables that are stored in the specified array or it can be a higher data type. For example, when the array is of type int, the data type specified can be long.

For each iteration of the for loop, an element of the array starting from the zeroth index is stored in the variable specified in the header and the body of the for loop is executed. In other words, the body of the for loop executes for each element of the array.

The following enhanced for loop is used to print the elements of the array.

int[] a ={3,4,7,9};  
for ( int x: a ) {  
    System.out.println(x);  
}

This is how the above code works. That value of x is first initialised to a[0] i.e. 3 and the body of the for loop is executed which will causes the integer 3 to be printed on the screen. Next a[1] i.e. 4 is assigned to x and the body executed again. In this the loop continues until all the elements are printed.

Note that the enhanced for loop cannot replace every for loop that is used to manipulate an array. This is because, the enhanced for loop only provides us the values held in the array and not a means to manipulate those values unless the array contains reference type variable. For example, we cannot use the for loop to initialise the array variables with the natural number from 1 to 7, as done above using a normal for loop. This is because elements here are of type int which is a primitive data type. Hence the variable x holds the copy of the integer variable of the array and not a reference to the variable. Moreover, we have no counter variables that would help in manipulating the array. However, if the array of of a reference type, like Student, the values stored in the array cannot not just be accessed but also modified, because in that case the variable specified in the for header (say x) holds a reference to the Student object stored in the array.

The enhanced for loop is used to iterate through not just an array, but a collection in general. We will see what collections are in a later chapter.

**ArrayList:**

Java ArrayList class

Java ArrayList class hierarchy

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non synchronized.
* Java ArrayList allows random access because array works at the index basis.
* In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

Hierarchy of ArrayList class

As shown in above diagram, Java ArrayList class extends AbstractList class which implements List interface. The List interface extends Collection and Iterable interfaces in hierarchical order.

ArrayList class declaration

Let's see the declaration for java.util.ArrayList class.

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| ArrayList() | It is used to build an empty array  list. |
| ArrayList(Collection c) | It is used to build an array list  that is initialized with the  elements of the collection c. |
| ArrayList(int capacity) | It is used to build an array list  that has the specified initial  capacity. |

1. **public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

Constructors of Java ArrayList

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a  list. |
| boolean addAll(Collection c) | It is used to append all of the  elements in the specified collection  to the end of this list, in the order  that they are returned by the  specified collection's iterator. |
| void clear() | It is used to remove all of the  elements from this list. |
| int lastIndexOf(Object o) | It is used to return the index in this  list of the last occurrence of the  specified element, or -1  if the list does not contain this  element. |
| Object[] toArray() | It is used to return an array  containing all of the elements in this  list in the correct order. |
| Object[] toArray(Object[] a) | It is used to return an array  containing all of the elements in this  list in the correct order. |
| boolean add(Object o) | It is used to append the  specified element to the  end of a list. |
| boolean addAll(int index, Collection c) | It is used to insert all of the elements  in the specified collection into this  list, starting at the specified position. |
| Object clone() | It is used to return a shallow copy of  an ArrayList. |
| int indexOf(Object o) | It is used to return the index in this  list of the first occurrence of  the specified element, or -1 if the  List does not contain this element. |
| void trimToSize() | It is used to trim the capacity of  this ArrayList instance to be the  list's current size. |

Methods of Java ArrayList

Java Non-generic Vs Generic Collection

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Let's see the old non-generic example of creating java collection.

1. ArrayList al=**new** ArrayList();//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

1. ArrayList<String> al=**new** ArrayList<String>();//creating new generic arraylist

In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives compile time error.

For more information of java generics, click here [Java Generics Tutorial](https://www.javatpoint.com/generics-in-java).

Java ArrayList Example

1. **import** java.util.\*;
2. **class** TestCollection1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

Ravi

Vijay

Ravi

Ajay

Two ways to iterate the elements of collection in java

There are two ways to traverse collection elements:

1. By Iterator interface.
2. By for-each loop.

In the above example, we have seen traversing ArrayList by Iterator. Let's see the example to traverse ArrayList elements using for-each loop.

Iterating Collection through for-each loop

1. **import** java.util.\*;
2. **class** TestCollection2{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. **for**(String obj:al)
10. System.out.println(obj);
11. }
12. }

Ravi

Vijay

Ravi

Ajay

User-defined class objects in Java ArrayList

Let's see an example where we are storing Student class object in array list.

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }
11. **import** java.util.\*;
12. **public** **class** TestCollection3{
13. **public** **static** **void** main(String args[]){
14. //Creating user-defined class objects
15. Student s1=**new** Student(101,"Sonoo",23);
16. Student s2=**new** Student(102,"Ravi",21);
17. Student s2=**new** Student(103,"Hanumat",25);
18. //creating arraylist
19. ArrayList<Student> al=**new** ArrayList<Student>();
20. al.add(s1);//adding Student class object
21. al.add(s2);
22. al.add(s3);
23. //Getting Iterator
24. Iterator itr=al.iterator();
25. //traversing elements of ArrayList object
26. **while**(itr.hasNext()){
27. Student st=(Student)itr.next();
28. System.out.println(st.rollno+" "+st.name+" "+st.age);
29. }
30. }
31. }

101 Sonoo 23

102 Ravi 21

103 Hanumat 25

Example of addAll(Collection c) method

1. **import** java.util.\*;
2. **class** TestCollection4{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Sonoo");
10. al2.add("Hanumat");
11. al.addAll(al2);//adding second list in first list
12. Iterator itr=al.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }

Ravi

Vijay

Ajay

Sonoo

Hanumat

Example of removeAll() method

1. **import** java.util.\*;
2. **class** TestCollection5{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Ravi");
10. al2.add("Hanumat");
11. al.removeAll(al2);
12. System.out.println("iterating the elements after removing the elements of al2...");
13. Iterator itr=al.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
18. }
19. }

iterating the elements after removing the elements of al2...

Vijay

Ajay

Example of retainAll() method

1. **import** java.util.\*;
2. **class** TestCollection6{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Ravi");
10. al2.add("Hanumat");
11. al.retainAll(al2);
12. System.out.println("iterating the elements after retaining the elements of al2...");
13. Iterator itr=al.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. }
18. }

iterating the elements after retaining the elements of al2...

Ravi

Java ArrayList Example: Book

Let's see an ArrayList example where we are adding books to list and printing all the books.

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ArrayListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

Java - Strings Class

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

## Creating Strings

The most direct way to create a string is to write −

String greeting = "Hello world!";

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

### Example

public class StringDemo {

public static void main(String args[]) {

char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };

String helloString = new String(helloArray);

System.out.println( helloString );

}

}

This will produce the following result −

### Output

hello.

**Note** − The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use [String Buffer & String Builder](https://www.tutorialspoint.com/java/java_string_buffer.htm) Classes.

## String Length

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

The following program is an example of **length()**, method String class.

### Example

public class StringDemo {

public static void main(String args[]) {

String palindrome = "Dot saw I was Tod";

int len = palindrome.length();

System.out.println( "String Length is : " + len );

}

}

This will produce the following result −

### Output

String Length is : 17

## Concatenating Strings

The String class includes a method for concatenating two strings −

string1.concat(string2);

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in −

"My name is ".concat("Zara");

Strings are more commonly concatenated with the + operator, as in −

"Hello," + " world" + "!"

which results in −

"Hello, world!"

Let us look at the following example −

### Example

public class StringDemo {

public static void main(String args[]) {

String string1 = "saw I was ";

System.out.println("Dot " + string1 + "Tod");

}

}

This will produce the following result −

### Output

Dot saw I was Tod

## Creating Format Strings

You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of −

### Example

System.out.printf("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

You can write −

String fs;

fs = String.format("The value of the float variable is " +

"%f, while the value of the integer " +

"variable is %d, and the string " +

"is %s", floatVar, intVar, stringVar);

System.out.println(fs);

## String Methods

Here is the list of methods supported by String class −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | [**char charAt(int index)**](https://www.tutorialspoint.com/java/java_string_charat.htm)  Returns the character at the specified index. |
| 2 | [**int compareTo(Object o)**](https://www.tutorialspoint.com/java/java_string_compareto.htm)  Compares this String to another Object. |
| 3 | [**int compareTo(String anotherString)**](https://www.tutorialspoint.com/java/java_string_compareto_anotherstring.htm)  Compares two strings lexicographically. |
| 4 | [**int compareToIgnoreCase(String str)**](https://www.tutorialspoint.com/java/java_string_comparetoignorecase.htm)  Compares two strings lexicographically, ignoring case differences. |
| 5 | [**String concat(String str)**](https://www.tutorialspoint.com/java/java_string_concat.htm)  Concatenates the specified string to the end of this string. |
| 6 | [**boolean contentEquals(StringBuffer sb)**](https://www.tutorialspoint.com/java/java_string_contentequals.htm)  Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer. |
| 7 | [**static String copyValueOf(char[] data)**](https://www.tutorialspoint.com/java/java_string_copyvalueof.htm)  Returns a String that represents the character sequence in the array specified. |
| 8 | [**static String copyValueOf(char[] data, int offset, int count)**](https://www.tutorialspoint.com/java/java_string_copyvalueof_dataoffsetcount.htm)  Returns a String that represents the character sequence in the array specified. |
| 9 | [**boolean endsWith(String suffix)**](https://www.tutorialspoint.com/java/java_string_endswith.htm)  Tests if this string ends with the specified suffix. |
| 10 | [**boolean equals(Object anObject)**](https://www.tutorialspoint.com/java/java_string_equals.htm)  Compares this string to the specified object. |
| 11 | [**boolean equalsIgnoreCase(String anotherString)**](https://www.tutorialspoint.com/java/java_string_equalsignorecase.htm)  Compares this String to another String, ignoring case considerations. |
| 12 | [**byte getBytes()**](https://www.tutorialspoint.com/java/java_string_getbytes.htm)  Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array. |
| 13 | [**byte[] getBytes(String charsetName)**](https://www.tutorialspoint.com/java/java_string_getbytes_charsetname.htm)  Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array. |
| 14 | [**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**](https://www.tutorialspoint.com/java/java_string_getchars.htm)  Copies characters from this string into the destination character array. |
| 15 | [**int hashCode()**](https://www.tutorialspoint.com/java/java_string_hashcode.htm)  Returns a hash code for this string. |
| 16 | [**int indexOf(int ch)**](https://www.tutorialspoint.com/java/java_string_indexof.htm)  Returns the index within this string of the first occurrence of the specified character. |
| 17 | [**int indexOf(int ch, int fromIndex)**](https://www.tutorialspoint.com/java/java_string_indexof_fromindex.htm)  Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index. |
| 18 | [**int indexOf(String str)**](https://www.tutorialspoint.com/java/java_string_indexof_str.htm)  Returns the index within this string of the first occurrence of the specified substring. |
| 19 | [**int indexOf(String str, int fromIndex)**](https://www.tutorialspoint.com/java/java_string_indexof_strfromindex.htm)  Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. |
| 20 | [**String intern()**](https://www.tutorialspoint.com/java/java_string_intern.htm)  Returns a canonical representation for the string object. |
| 21 | [**int lastIndexOf(int ch)**](https://www.tutorialspoint.com/java/java_string_lastindexof.htm)  Returns the index within this string of the last occurrence of the specified character. |
| 22 | [**int lastIndexOf(int ch, int fromIndex)**](https://www.tutorialspoint.com/java/java_string_lastindexof_chfromindex.htm)  Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index. |
| 23 | [**int lastIndexOf(String str)**](https://www.tutorialspoint.com/java/java_string_lastindexof_str.htm)  Returns the index within this string of the rightmost occurrence of the specified substring. |
| 24 | [**int lastIndexOf(String str, int fromIndex)**](https://www.tutorialspoint.com/java/java_string_lastindexof_fromindex.htm)  Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index. |
| 25 | [**int length()**](https://www.tutorialspoint.com/java/java_string_length.htm)  Returns the length of this string. |
| 26 | [**boolean matches(String regex)**](https://www.tutorialspoint.com/java/java_string_matches.htm)  Tells whether or not this string matches the given regular expression. |
| 27 | [**boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)**](https://www.tutorialspoint.com/java/java_string_regionmatches_ignorecase.htm)  Tests if two string regions are equal. |
| 28 | [**boolean regionMatches(int toffset, String other, int ooffset, int len)**](https://www.tutorialspoint.com/java/java_string_regionmatches.htm)  Tests if two string regions are equal. |
| 29 | [**String replace(char oldChar, char newChar)**](https://www.tutorialspoint.com/java/java_string_replace.htm)  Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar. |
| 30 | [**String replaceAll(String regex, String replacement**](https://www.tutorialspoint.com/java/java_string_replaceall.htm)  Replaces each substring of this string that matches the given regular expression with the given replacement. |
| 31 | [**String replaceFirst(String regex, String replacement)**](https://www.tutorialspoint.com/java/java_string_replacefirst.htm)  Replaces the first substring of this string that matches the given regular expression with the given replacement. |
| 32 | [**String[] split(String regex)**](https://www.tutorialspoint.com/java/java_string_split.htm)  Splits this string around matches of the given regular expression. |
| 33 | [**String[] split(String regex, int limit)**](https://www.tutorialspoint.com/java/java_string_split_regexlimit.htm)  Splits this string around matches of the given regular expression. |
| 34 | [**boolean startsWith(String prefix)**](https://www.tutorialspoint.com/java/java_string_startswith.htm)  Tests if this string starts with the specified prefix. |
| 35 | [**boolean startsWith(String prefix, int toffset)**](https://www.tutorialspoint.com/java/java_string_startswith_prefixtoffset.htm)  Tests if this string starts with the specified prefix beginning a specified index. |
| 36 | [**CharSequence subSequence(int beginIndex, int endIndex)**](https://www.tutorialspoint.com/java/java_string_subsequence.htm)  Returns a new character sequence that is a subsequence of this sequence. |
| 37 | [**String substring(int beginIndex)**](https://www.tutorialspoint.com/java/java_string_substring.htm)  Returns a new string that is a substring of this string. |
| 38 | [**String substring(int beginIndex, int endIndex)**](https://www.tutorialspoint.com/java/java_string_substring_beginendindex.htm)  Returns a new string that is a substring of this string. |
| 39 | [**char[] toCharArray()**](https://www.tutorialspoint.com/java/java_string_tochararray.htm)  Converts this string to a new character array. |
| 40 | [**String toLowerCase()**](https://www.tutorialspoint.com/java/java_string_tolowercase.htm)  Converts all of the characters in this String to lower case using the rules of the default locale. |
| 41 | [**String toLowerCase(Locale locale)**](https://www.tutorialspoint.com/java/java_string_tolowercase_locale.htm)  Converts all of the characters in this String to lower case using the rules of the given Locale. |
| 42 | [**String toString()**](https://www.tutorialspoint.com/java/java_string_tostring.htm)  This object (which is already a string!) is itself returned. |
| 43 | [**String toUpperCase()**](https://www.tutorialspoint.com/java/java_string_touppercase.htm)  Converts all of the characters in this String to upper case using the rules of the default locale. |
| 44 | [**String toUpperCase(Locale locale)**](https://www.tutorialspoint.com/java/java_string_touppercase_locale.htm)  Converts all of the characters in this String to upper case using the rules of the given Locale. |
| 45 | [**String trim()**](https://www.tutorialspoint.com/java/java_string_trim.htm)  Returns a copy of the string, with leading and trailing whitespace omitted. |
| 46 | [**static String valueOf(primitive data type x)**](https://www.tutorialspoint.com/java/java_string_valueof.htm)  Returns the string representation of the passed data type argument. |

# Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

## Important Constructors of StringBuilder class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuilder() | creates an empty string  Builder with the initial  capacity of 16. |
| StringBuilder(String str) | creates a string Builder  with the specified string. |
| StringBuilder(int length) | creates an empty string  Builder with the specified  capacity as length. |

## Important methods of StringBuilder class

|  |  |
| --- | --- |
| **Method** | **Description** |
| public StringBuilder append(String s) | is used to append the specified  string with this string. The  append() method is  overloaded like append(char),  append(boolean),  append(int), append(float),  append(double) etc. |
| public StringBuilder insert(int offset, String s) | is used to insert the specified  string with this string at the  specified position. The insert()  method is overloaded like  insert(int, char), insert(int, boolean),  insert(int, int), insert(int, float),  insert(int, double)  etc. |
| public StringBuilder replace(int startIndex,int endIndex, String str) | is used to replace the string from  specified startIndex and endIndex. |
| public StringBuilder delete(int startIndex, int endIndex) | is used to delete the string from  specified startIndex and endIndex. |
| public StringBuilder reverse() | is used to reverse the string. |
| public int capacity() | is used to return the current  capacity. |
| Public void ensureCapacity(int minimumCapacity) | is used to ensure the capacity at  least equal to the given minimum. |
| public char charAt(int index) | is used to return the character  at the specified position. |
| public int length() | is used to return the length of  the string i.e. total number of  characters. |
| public String substring(int beginIndex) | is used to return the substring  from the specified beginIndex. |
| public String substring(int beginIndex, int endIndex) | is used to return the substring from  the s |

## Java StringBuilder Examples

Let's see the examples of different methods of StringBuilder class.

### 1) StringBuilder append() method

The StringBuilder append() method concatenates the given argument with this string.

1. **class** StringBuilderExample{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }

### 2) StringBuilder insert() method

The StringBuilder insert() method inserts the given string with this string at the given position.

1. **class** StringBuilderExample2{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }

### 3) StringBuilder replace() method

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

1. **class** StringBuilderExample3{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }

### 4) StringBuilder delete() method

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

1. **class** StringBuilderExample4{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello");
4. sb.delete(1,3);
5. System.out.println(sb);//prints Hlo
6. }
7. }

### 5) StringBuilder reverse() method

The reverse() method of StringBuilder class reverses the current string.

1. **class** StringBuilderExample5{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }

### 6) StringBuilder capacity() method

The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

1. **class** StringBuilderExample6{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
9. }
10. }

### 7) StringBuilder ensureCapacity() method

The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity\*2)+2. For example if your current capacity is 16, it will be (16\*2)+2=34.

1. **class** StringBuilderExample7{
2. **public** **static** **void** main(String args[]){
3. StringBuilder sb=**new** StringBuilder();
4. System.out.println(sb.capacity());//default 16
5. sb.append("Hello");
6. System.out.println(sb.capacity());//now 16
7. sb.append("java is my favourite language");
8. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
9. sb.ensureCapacity(10);//now no change
10. System.out.println(sb.capacity());//now 34
11. sb.ensureCapacity(50);//now (34\*2)+2
12. System.out.println(sb.capacity());//now 70
13. }
14. }

Wrapper class in Java

**Wrapper class in java** provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing.

The eight classes of java.lang package are known as wrapper classes in java. The list of eight wrapper classes are given below:

|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper class** |
| boolean | Boolean |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

Wrapper class Example: Primitive to Wrapper

1. **public** **class** WrapperExample1{
2. **public** **static** **void** main(String args[]){
3. //Converting int into Integer
4. **int** a=20;
5. Integer i=Integer.valueOf(a);//converting int into Integer
6. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
8. System.out.println(a+" "+i+" "+j);
9. }}

Output:

20 20 20

Wrapper class Example: Wrapper to Primitive

1. **public** **class** WrapperExample2{
2. **public** **static** **void** main(String args[]){
3. //Converting Integer to int
4. Integer a=**new** Integer(3);
5. **int** i=a.intValue();//converting Integer to int
6. **int** j=a;//unboxing, now compiler will write a.intValue() internally
8. System.out.println(a+" "+i+" "+j);
9. }}

Output:

3 3 3

# Autoboxing and Unboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called unboxing.

Here is the simplest example of autoboxing:

Character ch = 'a';

The rest of the examples in this section use generics. If you are not yet familiar with the syntax of generics, see the [Generics (Updated)](https://docs.oracle.com/javase/tutorial/java/generics/index.html) lesson.

Consider the following code:

List<Integer> li = new ArrayList<>();

for (int i = 1; i < 50; i += 2)

li.add(i);

Although you add the int values as primitive types, rather than Integer objects, to li, the code compiles. Because li is a list of Integer objects, not a list of int values, you may wonder why the Java compiler does not issue a compile-time error. The compiler does not generate an error because it creates an Integer object from i and adds the object to li. Thus, the compiler converts the previous code to the following at runtime:

List<Integer> li = new ArrayList<>();

for (int i = 1; i < 50; i += 2)

li.add(Integer.valueOf(i));

Converting a primitive value (an int, for example) into an object of the corresponding wrapper class (Integer) is called autoboxing. The Java compiler applies autoboxing when a primitive value is:

* Passed as a parameter to a method that expects an object of the corresponding wrapper class.
* Assigned to a variable of the corresponding wrapper class.

Consider the following method:

public static int sumEven(List<Integer> li) {

int sum = 0;

for (Integer i: li)

if (i % 2 == 0)

sum += i;

return sum;

}

Because the remainder (%) and unary plus (+=) operators do not apply to Integer objects, you may wonder why the Java compiler compiles the method without issuing any errors. The compiler does not generate an error because it invokes the intValue method to convert an Integer to an int at runtime:

public static int sumEven(List<Integer> li) {

int sum = 0;

for (Integer i : li)

if (i.intValue() % 2 == 0)

sum += i.intValue();

return sum;

}

Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value is called unboxing. The Java compiler applies unboxing when an object of a wrapper class is:

* Passed as a parameter to a method that expects a value of the corresponding primitive type.
* Assigned to a variable of the corresponding primitive type.

The [Unboxing](https://docs.oracle.com/javase/tutorial/java/data/examples/Unboxing.java) example shows how this works:

import java.util.ArrayList;

import java.util.List;

public class Unboxing {

public static void main(String[] args) {

Integer i = new Integer(-8);

// 1. Unboxing through method invocation

int absVal = absoluteValue(i);

System.out.println("absolute value of " + i + " = " + absVal);

List<Double> ld = new ArrayList<>();

ld.add(3.1416); // Π is autoboxed through method invocation.

// 2. Unboxing through assignment

double pi = ld.get(0);

System.out.println("pi = " + pi);

}

public static int absoluteValue(int i) {

return (i < 0) ? -i : i;

}

}

The program prints the following:

absolute value of -8 = 8

pi = 3.1416

Autoboxing and unboxing lets developers write cleaner code, making it easier to read. The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing:

|  |  |
| --- | --- |
| **Primitive type** | **Wrapper class** |
| boolean | Boolean |
| byte | Byte |
| char | Character |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |
| double | Double |