# **Java String :**

In [Java](https://www.javatpoint.com/java-tutorial), string is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. For example:

1. **char**[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=**new** String(ch);

is same as:

1. String s="javatpoint";

**Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

## CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, [StringBuffer](https://www.javatpoint.com/StringBuffer-class) and [StringBuilder](https://www.javatpoint.com/StringBuilder-class) classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

What is String in Java?

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

1. String s="welcome";

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

### Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

### 2) By new keyword

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

### By New Keyword

Strings can indeed be created using the new keyword in Java. When a string is created with new, a new object of the String class is created in the heap memory, outside the string constant pool.

Unlike string literals, these objects are allocated separate memory space in the heap, regardless of whether the same value already exists in the heap or not.

**Syntax:**

String stringName = new String("string\_value");

**Example: Creating Java Strings using the new keyword**

String str = new String("Program");

System.out.println(str);

***Image for Reference:***

### Java String Example

**StringExample.java**

1. **public** **class** StringExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java";//creating string by Java string literal
4. **char** ch[]={'s','t','r','i','n','g','s'};
5. String s2=**new** String(ch);//converting char array to string
6. String s3=**new** String("example");//creating Java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=StringExample)

**Output:**

java

strings

example

# **Immutable String in Java**

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once String object is created its data or state can't be changed but a new String object is created.

Let's try to understand the concept of immutability by the example given below:

**Testimmutablestring.java**

1. **class** Testimmutablestring{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin";
4. s.concat(" Tendulkar");//concat() method appends the string at the end
5. System.out.println(s);//will print Sachin because strings are immutable objects
6. }
7. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring)

**Output:**

Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with Sachin Tendulkar. That is why String is known as immutable.



As you can see in the above figure that two objects are created but **s** reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

For example:

**Testimmutablestring1.java**

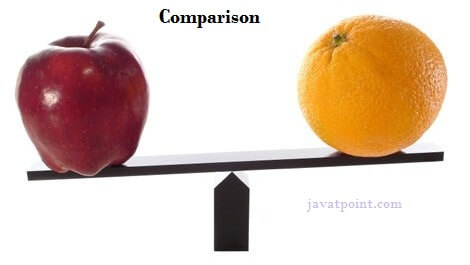
1. **class** Testimmutablestring1{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin";
4. s=s.concat(" Tendulkar");
5. System.out.println(s);
6. }
7. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring1)

**Output:**

Sachin Tendulkar

# **Java String compare**



We can compare String in Java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare String in Java:

1. By Using equals() Method
2. By Using == Operator
3. By compareTo() Method

## 1) By Using equals() Method

The String class equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

1. **class** Teststringcomparison1{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3=**new** String("Sachin");
6. String s4="Saurav";
7. System.out.println(s1.equals(s2));//true
8. System.out.println(s1.equals(s3));//true
9. System.out.println(s1.equals(s4));//false
10. }
11. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison1)

**Output:**

true

true

false

In the above code, two strings are compared using **equals()** method of **String** class. And the result is printed as boolean values, **true** or **false**.

**Teststringcomparison2.java**

1. **class** Teststringcomparison2{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="SACHIN";
6. System.out.println(s1.equals(s2));//false
7. System.out.println(s1.equalsIgnoreCase(s2));//true
8. }
9. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison2)

**Output:**

false

true

In the above program, the methods of **String** class are used. The **equals()** method returns true if String objects are matching and both strings are of same case. **equalsIgnoreCase()** returns true regardless of cases of strings

## 2) By Using == operator

The == operator compares references not values.

1. **class** Teststringcomparison3{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3=**new** String("Sachin");
6. System.out.println(s1==s2);//true (because both refer to same instance)
7. System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
8. }
9. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison3)

**Output:**

true

false

## 3) String compare by compareTo() method

The above code, demonstrates the use of **==** operator used for comparing two **String** objects.

## 3) By Using compareTo() method

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:

* **s1 == s2** : The method returns 0.
* **s1 > s2** : The method returns a positive value.
* **s1 < s2** : The method returns a negative value.
* **class** Teststringcomparison4{
* **public** **static** **void** main(String args[]){
* String s1="Sachin";
* String s2="Sachin";
* String s3="Ratan";
* System.out.println(s1.compareTo(s2));//0
* System.out.println(s1.compareTo(s3));//1(because s1>s3)
* System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
* }
* }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison4)

**Output:**

0

1

-1

# **String Concatenation in Java**

In Java, String concatenation forms a new String that is the combination of multiple strings. There are two ways to concatenate strings in Java:

1. By + (String concatenation) operator
2. By concat() method

## 1) String Concatenation by + (String concatenation) operator

Java String concatenation operator (+) is used to add strings. For Example:

#### . By + (String concatenation) operator:

In JavaScript, the **+** operator is used not only for addition but also for string concatenation. When the **+** operator is used with strings, it concatenates them together. Here's how it works:

let str1 = "Hello";

let str2 = "world";

let result = str1 + " " + str2;

console.log(result); // Output: Hello world

In this example:

* We have two strings, **"Hello"** and **"world"**.
* The **+** operator is used to concatenate **str1** and **str2**, with a space **" "** in between them.
* The result is **"Hello world"**, which is stored in the **result** variable.

#### 2. By concat() method:

JavaScript provides a method called **concat()** for concatenating strings. This method takes one or more strings as arguments and returns a new string containing the concatenated strings. Here's how it works:

let str1 = "Hello";

let str2 = "world";

let result = str1.concat(" ", str2);

console.log(result); // Output: Hello world

In this example:

* We have two strings, **"Hello"** and **"world"**.
* The **concat()** method is called on **str1** with **str2** as an argument, along with a space **" "** as an additional argument.
* The **concat()** method joins **str1**, the space, and **str2** together, resulting in the string **"Hello world"**.
* The result is stored in the **result** variable.

**TestStringConcatenation1.java**

1. **class** TestStringConcatenation1{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin"+" Tendulkar";
4. System.out.println(s);//Sachin Tendulkar
5. }
6. }

**Test it Now**

S SaSadfkfgk Sachin Tendulkar

### Sasgdfkg 2) String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

1. **public** String concat(String another)

Let's see the example of String concat() method.

**TestStringConcatenation3.java**

1. **class** TestStringConcatenation3{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin ";
4. String s2="Tendulkar";
5. String s3=s1.concat(s2);
6. System.out.println(s3);//Sachin Tendulkar
7. }
8. }

# **Substring in Java**

A part of String is called **substring**. In other words, substring is a subset of another String. Java String class provides the built-in substring() method that extract a substring from the given string by using the index values passed as an argument. In case of substring() method startIndex is inclusive and endIndex is exclusive.

### Example of Java substring() method

**TestSubstring.java**

1. **public** **class** TestSubstring{
2. **public** **static** **void** main(String args[]){
3. String s="SachinTendulkar";
4. System.out.println("Original String: " + s);
5. System.out.println("Substring starting from index 6: " +s.substring(6));//Tendulkar
6. System.out.println("Substring starting from index 0 to 6: "+s.substring(0,6)); //Sachin
7. }
8. }

**Output:**

Original String: SachinTendulkar

Substring starting from index 6: Tendulkar

Substring starting from index 0 to 6: Sachin

# **Java toString() Method**

If you want to represent any object as a string, **toString() method** comes into existence.

The toString() method returns the String representation of the object.

If you print any object, Java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on your implementation.

### Advantage of Java toString() method

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

### Understanding problem without toString() method

Let's see the simple code that prints reference.

**Student.java**

1. **class** Student{
2. **int** rollno;
3. String name;
4. String city;
6. Student(**int** rollno, String name, String city){
7. **this**.rollno=rollno;
8. **this**.name=name;
9. **this**.city=city;
10. }
12. **public** **static** **void** main(String args[]){
13. Student s1=**new** Student(101,"Raj","lucknow");
14. Student s2=**new** Student(102,"Vijay","ghaziabad");
16. System.out.println(s1);//compiler writes here s1.toString()
17. System.out.println(s2);//compiler writes here s2.toString()
18. }
19. }

**Output:**

Student@1fee6fc

Student@1eed786

As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since Java compiler internally calls toString() method, overriding this method will return the specified values. Let's understand it with the example given below:

1. **String** class: It overrides **toString()** to return itself because a string representation of a string is the string itself.
2. **Integer**, **Double**, **Boolean**, etc.: These wrapper classes override **toString()** to return the string representation of the wrapped primitive value.
3. **ArrayList**, **HashMap**, **HashSet**, etc.: These collection classes override **toString()** to provide a string representation of their elements.

## Example of Java toString() method

Let's see an example of toString() method.

**Student.java**

1. **class** Student{
2. **int** rollno;
3. String name;
4. String city;
6. Student(**int** rollno, String name, String city){
7. **this**.rollno=rollno;
8. **this**.name=name;
9. **this**.city=city;
10. }
12. **public** String toString(){//overriding the toString() method
13. **return** rollno+" "+name+" "+city;
14. }
15. **public** **static** **void** main(String args[]){
16. Student s1=**new** Student(101,"Raj","lucknow");
17. Student s2=**new** Student(102,"Vijay","ghaziabad");
19. System.out.println(s1);//compiler writes here s1.toString()
20. System.out.println(s2);//compiler writes here s2.toString()
21. }
22. }

**Output:**

101 Raj lucknow

102 Vijay ghaziabad

# **Java StringBuffer Class**

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

### What is a mutable String?

A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

### 1) StringBuffer Class append() Method

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

**StringBufferExample.java**

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

**Output:**

Hello Java

### 2) StringBuffer insert() Method

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

**StringBufferExample2.java**

1. **class** StringBufferExample2{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello ");
4. sb.insert(1,"Java");//now original string is changed
5. System.out.println(sb);//prints HJavaello
6. }
7. }

**Output:**

HJavaello

### 3) StringBuffer replace() Method

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

**StringBufferExample3.java**

1. **class** StringBufferExample3{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello");
4. sb.replace(1,3,"Java");
5. System.out.println(sb);//prints HJavalo
6. }
7. }

**Output:**

HJavalo

### 5) StringBuffer reverse() Method

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

**StringBufferExample5.java**

1. **class** StringBufferExample5{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello");
4. sb.reverse();
5. System.out.println(sb);//prints olleH
6. }
7. }

**Output:**

# **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.

## 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.

**StringBuilderExample.java**

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. }

**Output:**

Hello Java

### 2) StringBuilder insert() method

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

**StringBuilderExample2.java**

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. }

**Output:**

HJavaello

# **Difference between StringBuffer and StringBuilder**

Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

### 1. length()

This method returns the length of the string:

String str = "Hello";

int length = str.length(); // length is 5

### charAt(int index)

This method returns the character at the specified index:

String str = "Hello";

char ch = str.charAt(0); // ch is 'H'

### 3. substring(int beginIndex)

This method returns a substring starting from the specified index:

String str = "Hello";

String subStr = str.substring(2); // subStr is "llo"

### 4. substring(int beginIndex, int endIndex)

This method returns a substring starting from the specified begin index and ending at the specified end index (exclusive):

String str = "Hello";

String subStr = str.substring(1, 4); // subStr is "ell"

### indexOf(String str)

This method returns the index of the first occurrence of the specified substring:

String str = "Hello";

int index = str.indexOf("l"); // index is 2

### indexOf(String str, int fromIndex)

This method returns the index of the first occurrence of the specified substring, starting the search at the specified index:

String str = "Hello";

int index = str.indexOf("l", 3); // index is 3

### toUpperCase()

This method returns a string with all the characters converted to uppercase:

String str = "hello";

String upperCaseStr = str.toUpperCase(); // upperCaseStr is "HELLO"

### . toLowerCase()

This method returns a string with all the characters converted to lowercase:

String str = "HELLO";

String lowerCaseStr = str.toLowerCase(); // lowerCaseStr is "hello"

### trim()

This method returns a copy of the string with leading and trailing whitespace removed:

String str = " Hello ";

String trimmedStr = str.trim(); // trimmedStr is "Hello"

### 10. replace(CharSequence target, CharSequence replacement)

This method returns a new string resulting from replacing all occurrences of **target** with **replacement**:

String str = "Hello, world!";

String newStr = str.replace("world", "Java"); // newStr is "Hello, Java!"

Java Arrays

An array is a data structure that stores a fixed-size collection of elements of the same type. Key characteristics include:

* **Fixed Size**: Arrays have a fixed size determined at the time of creation. Once created, the size cannot be changed.
* **Contiguous Memory Allocation**: Elements are stored in adjacent memory locations, allowing for efficient access based on index.
* **Random Access**: Elements can be accessed directly using their index positions, providing constant-time access.
* **Homogeneous Elements**: All elements in an array must be of the same data type.

### Implement Basic Operations:

#### Access:

Accessing an element in an array involves using its index. Indexing starts at 0 in most programming languages

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

int thirdElement = numbers[2]; // Accessing the third element (index 2)

System.out.println(thirdElement); // Output: 3

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

#### Insert:

Inserting an element at a specific index involves shifting existing elements to make space for the new one.

int[] numbers = {1, 2, 4, 5};

int newValue = 3;

int insertIndex = 2;

// Shift elements to the right

for (int i = numbers.length - 1; i > insertIndex; i--) {

numbers[i] = numbers[i - 1];

}

numbers[insertIndex] = newValue;

Explantion:

1. We have an array called **numbers** initially containing **{1, 2, 4, 5}**.
2. We want to insert a new value, **3**, into the array at a specific index, **2**.
3. We initialize a variable **newValue** to **3**, representing the value to be inserted.
4. We initialize a variable **insertIndex** to **2**, representing the index where the new value will be inserted.
5. We start a **for** loop from the end of the array (**numbers.length - 1**) and iterate backwards until we reach the **insertIndex**.
   * In each iteration, we move the value at index **i - 1** to index **i**. This effectively shifts elements to the right to make space for the new value.
6. After shifting elements, we assign the **newValue** to the **insertIndex** position in the array.

Let's visualize the array before and after insertion:

Before insertion: **{1, 2, 4, 5}**

After insertion: **{1, 2, 3, 4, 5}**

Explanation:

* Initially, the array contains elements **{1, 2, 4, 5}**.
* We want to insert **3** at index **2**.
* We start shifting elements from the end of the array towards the **insertIndex** (from index **3** to index **2**).
* During each iteration, the value at index **i - 1** is moved to index **i**, creating space for the new value.
* After shifting, we assign the **newValue** (which is **3**) to index **2**.
* The final array becomes **{1, 2, 3, 4, 5}**, with **3** inserted at the desired index.

#### Delete:

Deleting an element involves removing it from the array and shifting subsequent elements to fill the gap.

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

int deleteIndex = 2;

// Shift elements to the left

for (int i = deleteIndex; i < numbers.length - 1; i++) {

numbers[i] = numbers[i + 1];

}

#### Update:

Updating an element involves assigning a new value to the element at a specific index.

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

numbers[2] = 10; // Updating the value at index 2

System.out.println(numbers[2]); // Output: 10

#### Reverse:

Reversing an array involves swapping elements from both ends until reaching the middle.

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

for (int i = 0, j = numbers.length - 1; i < j; i++, j--) {

int temp = numbers[i];

numbers[i] = numbers[j];

numbers[j] = temp;

}

**3. Sorting an Array - Brute Force Method:**

A simple way to sort an array is the Bubble Sort algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

int[] numbers = {5, 2, 7, 1, 3};

boolean swapped;

do {

swapped = false;

for (int i = 0; i < numbers.length - 1; i++) {

if (numbers[i] > numbers[i + 1]) {

int temp = numbers[i];

numbers[i] = numbers[i + 1];

numbers[i + 1] = temp;

swapped = true;

}

}

} while (swapped);

Explanation:

1. We have an array called **numbers** initially containing **{5, 2, 7, 1, 3}**.
2. We want to sort the array in ascending order.
3. We initialize a boolean variable **swapped** to **false**. This variable will be used to track whether any elements were swapped during a pass through the array.
4. We enter a **do-while** loop. This loop will execute at least once and continue executing as long as the **swapped** variable remains **true**, indicating that elements were swapped during the previous pass through the array.
5. Inside the loop:
   * We set **swapped** to **false** at the beginning of each iteration to reset it.
   * We iterate through the array using a **for** loop from index **0** to **length - 1**.
   * For each pair of adjacent elements **numbers[i]** and **numbers[i + 1]**, if **numbers[i]** is greater than **numbers[i + 1]**, we swap them.
   * After each swap, we set **swapped** to **true** to indicate that a swap occurred.
6. The loop continues until no elements are swapped during a pass through the array, indicating that the array is sorted.

Let's visualize how the array is sorted:

1. Pass 1:
   * **{5, 2, 7, 1, 3}** -> **{2, 5, 7, 1, 3}** (swapped)
   * **{2, 5, 7, 1, 3}** -> **{2, 5, 1, 7, 3}** (swapped)
   * **{2, 5, 1, 7, 3}** -> **{2, 5, 1, 3, 7}** (swapped)
2. Pass 2:
   * **{2, 5, 1, 3, 7}** -> **{2, 1, 5, 3, 7}** (swapped)
   * **{2, 1, 5, 3, 7}** -> **{2, 1, 3, 5, 7}** (swapped)
3. Pass 3:
   * **{2, 1, 3, 5, 7}** -> **{1, 2, 3, 5, 7}** (swapped)

After the third pass, no elements were swapped, so the loop exits. The array **{1, 2, 3, 5, 7}** is now sorted in ascending order.

**4. Searching an Array - Linear Search Method:**

Linear search scans each element of the array sequentially until the target element is found or the end of the array is reached.

int[] numbers = {5, 2, 7, 1, 3};

int target = 7;

int index = -1;

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

if (numbers[i] == target) {

index = i;

break;

}

}

1. We have an array called **numbers** initially containing **{5, 2, 7, 1, 3}**.
2. We want to search for a specific target value, **7**, in the array.
3. We initialize a variable **index** to **-1**. This variable will store the index of the target value if found, or remain **-1** if the target value is not found.
4. We enter a **for** loop that iterates through each element of the array.
5. Inside the loop:
   * We compare each element **numbers[i]** with the target value **7**.
   * If the current element is equal to the target value, we update the **index** variable with the current index **i** and exit the loop using the **break** statement.
6. After the loop:
   * If the target value is found in the array, the **index** variable will hold the index of the target value.
   * If the target value is not found, the **index** variable will remain **-1**.

Let's visualize how the program works:

1. Iteration 1: **numbers[0]** is **5**, not equal to **7**.
2. Iteration 2: **numbers[1]** is **2**, not equal to **7**.
3. Iteration 3: **numbers[2]** is **7**, equal to **7**. We update **index** to **2** and exit the loop.

After the loop, **index** is **2**, indicating that the target value **7** was found at index **2** in the array **{5, 2, 7, 1, 3}**. If the target value was not found, **index** would remain **-1**.

To declare an array, define the variable type with **square brackets**:

String[] cars;

We have now declared a variable that holds an array of strings. To insert values to it, you can place the values in a comma-separated list, inside curly braces:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

To create an array of integers, you could write:

int[] myNum = {10, 20, 30, 40};

Access the Elements of an Array

You can access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars[0]);

// Outputs Volvo

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_array)

**Note:** Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

cars[0] = "Opel";

Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

cars[0] = "Opel";

System.out.println(cars[0]);

// Now outputs Opel instead of Volvo

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_array_change)

Array Length

To find out how many elements an array has, use the length property:

Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

System.out.println(cars.length);

// Outputs 4

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_array_length)

## Loop Through an Array

You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

### Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

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

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

}

The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

### Example

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (String i : cars) {

System.out.println(i);

}

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_array_for_each)

The example above can be read like this: **for each** String element (called **i** - as in **i**ndex) in **cars**, print out the value of**i**.

If you compare the for loop and **for-each** loop, you will see that the **for-each** method is easier to write, it does not require a counter (using the length property), and it is more readable.