# **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**

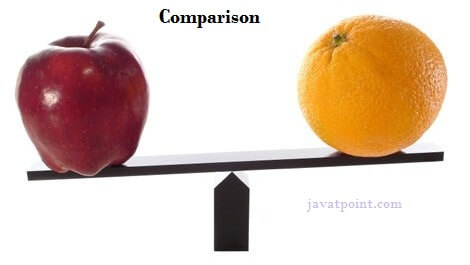
1. **class** Testimmutablestring1{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin";
4. S1=s.concat(" Tendulkar");
5. System.out.println(s);
6. }
7. }

[**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));
8. System.out.println(s1.equals(s3));
9. System.out.println(s1.equals(s4));
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));
7. System.out.println(s1.equalsIgnoreCase(s2));
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);
7. System.out.println(s1==s3);
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 collection of similar types of data.

For example, if we want to store the names of 100 people then we can create an array of the string type that can store 100 names.

String[] array = new String[100];

Here, the above array cannot store more than 100 names. The number of values in a Java array is always fixed.

**How to declare an array in Java?**

In Java, here is how we can declare an array.

dataType[] arrayName;

* dataType - it can be [primitive data types](https://www.programiz.com/java-programming/variables-primitive-data-types#data-types) like int, char, double, byte, etc. or [Java objects](https://www.programiz.com/java-programming/class-objects)
* arrayName - it is an [identifier](https://www.programiz.com/java-programming/keywords-identifiers#identifiers)

For example,

double[] data;

Here, data is an array that can hold values of type double.

**But, how many elements can array this hold?**

Good question! To define the number of elements that an array can hold, we have to allocate memory for the array in Java. For example,

// declare an array

double[] data;

// allocate memory

data = new double[10];

Here, the array can store **10** elements. We can also say that the **size or length** of the array is 10.

In Java, we can declare and allocate the memory of an array in one single statement. For example,

double[] data = new double[10];

**How to Initialize Arrays in Java?**

In Java, we can initialize arrays during declaration. For example,

//declare and initialize and array

int[] age = {12, 4, 5, 2, 5};

Here, we have created an array named age and initialized it with the values inside the curly brackets.

Note that we have not provided the size of the array. In this case, the Java compiler automatically specifies the size by counting the number of elements in the array (i.e. 5).

In the Java array, each memory location is associated with a number. The number is known as an array index. We can also initialize arrays in Java, using the index number. For example,

// declare an array

int[] age = new int[5];

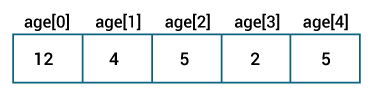
// initialize array

age[0] = 12;

age[1] = 4;

age[2] = 5;

..

Java Arrays initialization

**Note**:

* Array indices always start from 0. That is, the first element of an array is at index 0.
* If the size of an array is n, then the last element of the array will be at index n-1.

**How to Access Elements of an Array in Java?**

We can access the element of an array using the index number. Here is the syntax for accessing elements of an array,

// access array elements

array[index]

Let's see an example of accessing array elements using index numbers.

**Example: Access Array Elements**

class Main {

public static void main(String[] args) {

// create an array

int[] age = {12, 4, 5, 2, 5};

// access each array elements

System.out.println("Accessing Elements of Array:");

System.out.println("First Element: " + age[0]);

System.out.println("Second Element: " + age[1]);

System.out.println("Third Element: " + age[2]);

System.out.println("Fourth Element: " + age[3]);

System.out.println("Fifth Element: " + age[4]);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Accessing Elements of Array:

First Element: 12

Second Element: 4

Third Element: 5

Fourth Element: 2

Fifth Element: 5

In the above example, notice that we are using the index number to access each element of the array.

We can use loops to access all the elements of the array at once.

**Looping Through Array Elements**

In Java, we can also loop through each element of the array. For example,

**Example: Using For Loop**

class Main {

public static void main(String[] args) {

// create an array

int[] age = {12, 4, 5};

// loop through the array

// using for loop

System.out.println("Using for Loop:");

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

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

}

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Using for Loop:

12

4

5

In the above example, we are using the [for Loop in Java](https://www.programiz.com/java-programming/for-loop) to iterate through each element of the array. Notice the expression inside the loop,

age.length

Here, we are using the length property of the array to get the size of the array.

We can also use the [for-each loop](https://www.programiz.com/java-programming/enhanced-for-loop) to iterate through the elements of an array. For example,

**Example: Using the for-each Loop**

class Main {

public static void main(String[] args) {

// create an array

int[] age = {12, 4, 5};

// loop through the array

// using for loop

System.out.println("Using for-each Loop:");

for(int a : age) {

System.out.println(a);

}

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Using for-each Loop:

12

4

5

**Example: Compute Sum and Average of Array Elements**

class Main {

public static void main(String[] args) {

int[] numbers = {2, -9, 0, 5, 12, -25, 22, 9, 8, 12};

int sum = 0;

Double average;

// access all elements using for each loop

// add each element in sum

for (int number: numbers) {

sum += number;

}

// get the total number of elements

int arrayLength = numbers.length;

// calculate the average

// convert the average from int to double

average = ((double)sum / (double)arrayLength);

System.out.println("Sum = " + sum);

System.out.println("Average = " + average);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Sum = 36

Average = 3.6

In the above example, we have created an array of named numbers. We have used the for...each loop to access each element of the array.

Inside the loop, we are calculating the sum of each element. Notice the line,

int arrayLength = number.length;

Here, we are using the [length attribute](http://stackoverflow.com/questions/8755812/array-length-in-java) of the array to calculate the size of the array. We then calculate the average using:

average = ((double)sum / (double)arrayLength);

As you can see, we are converting the int value into double. This is called type casting in Java. To learn more about typecasting, visit [Java Type Casting](https://www.programiz.com/java-programming/typecasting).

**Multidimensional Arrays**

Arrays we have mentioned till now are called one-dimensional arrays. However, we can declare multidimensional arrays in Java.

A multidimensional array is an array of arrays. That is, each element of a multidimensional array is an array itself. For example,

double[][] matrix = {{1.2, 4.3, 4.0},

{4.1, -1.1}

};

Here, we have created a multidimensional array named matrix. It is a 2-dimensional array. To learn more, visit the [Java multidimensional array](https://www.programiz.com/java-programming/multidimensional-array).

### ****Functions in Java****

**Functions** (also called methods in Java) are reusable blocks of code designed to perform a specific task. They make programs modular and easier to maintain.

#### **Key Points about Functions:**

1. **Declaration:** Functions are defined with a return type, name, and parameters (optional).
2. **Calling:** Functions are called by their name followed by parentheses. If they have parameters, values must be passed in the same order as defined.
3. **Return Type:** Specifies what type of value the function returns. Use void if the function doesn’t return anything.

#### **Example 1: Function to Calculate the Square of a Number**

java

Copy code

public class FunctionExample {

public static void main(String[] args) {

int number = 5;

int result = calculateSquare(number); // Call the function

System.out.println("Square of " + number + " is: " + result);

}

// Function to calculate the square

public static int calculateSquare(int num) {

return num \* num; // Return the square of the number

}

}

#### **Example 2: Function to Greet a User**

java

Copy code

public class GreetingExample {

public static void main(String[] args) {

greetUser("Lynn"); // Call the function with a parameter

}

// Function to greet a user

public static void greetUser(String name) {

System.out.println("Hello, " + name + "! Welcome to Java programming.");

}

}

### ****Arrays - Parallel Array Sorting****

**Parallel Array Sorting** is a method introduced in Java 8 to sort arrays using parallelism for better performance. It divides the array into parts, sorts them concurrently, and merges the results.

#### **Advantages:**

* Faster sorting for large arrays.
* Automatically uses multithreading for efficiency.

#### **Example: Sorting an Array of Integers**

import java.util.Arrays; // Import Arrays utility class

public class ParallelArraySortingExample {

public static void main(String[] args) {

int[] numbers = {42, 15, 8, 23, 4, 16};

System.out.println("Before Sorting: " + Arrays.toString(numbers));

// Sort the array using parallel sort

Arrays.parallelSort(numbers);

System.out.println("After Sorting: " + Arrays.toString(numbers));

}

}

**Explanation:**

1. **Arrays.toString(numbers)** converts the array into a readable string format for display.
2. **Arrays.parallelSort(numbers)** sorts the array in ascending order using parallel sorting.

#### **Example: Sorting an Array of Strings**

java

Copy code

import java.util.Arrays;

public class ParallelStringSorting {

public static void main(String[] args) {

String[] names = {"Lynn", "Zara", "Adam", "Cathy"};

System.out.println("Before Sorting: " + Arrays.toString(names));

// Sort the array of strings

Arrays.parallelSort(names);

System.out.println("After Sorting: " + Arrays.toString(names));

}

}

### ****Combining Functions and Parallel Array Sorting****

#### **Example: Function to Sort an Array**

java

Copy code

import java.util.Arrays;

public class SortFunctionExample {

public static void main(String[] args) {

int[] scores = {95, 70, 88, 65, 99};

System.out.println("Original Scores: " + Arrays.toString(scores));

// Call the function to sort the array

sortArray(scores);

System.out.println("Sorted Scores: " + Arrays.toString(scores));

}

// Function to sort an array using parallel sort

public static void sortArray(int[] array) {

Arrays.parallelSort(array); // Sort the array

}

}

**Explanation:**

1. The function sortArray takes an integer array as input.
2. Inside the function, Arrays.parallelSort is used to sort the array.
3. The sorted array is displayed in the main method.

### ****Output Examples****

#### **Example 1 (Integer Sorting):**

**Input Array:** {42, 15, 8, 23, 4, 16}  
**Output:**

less

Copy code

Before Sorting: [42, 15, 8, 23, 4, 16]

After Sorting: [4, 8, 15, 16, 23, 42]

#### **Example 2 (String Sorting):**

**Input Array:** {"Lynn", "Zara", "Adam", "Cathy"}  
**Output:**

less

Copy code

Before Sorting: [Lynn, Zara, Adam, Cathy]

After Sorting: [Adam, Cathy, Lynn, Zara]

In Java 8, the Optional class is introduced in the java.util package to address the problem of NullPointerException. It provides a way to explicitly specify that a value may or may not be present.

Here’s a detailed explanation:

**What is Optional?**

Optional is a container object that may or may not contain a non-null value. If a value is present, it provides methods to access the value; otherwise, it indicates that the value is absent.

**Key Features**

1. **Null Safety**: Helps avoid NullPointerException.
2. **Readability**: Makes it clear in the code when a value can be absent.
3. **Functional Style**: Supports functional programming with methods like map, filter, and flatMap.

**Creating an Optional**

You can create an Optional using the following static methods:

1. **Optional.of(value)**  
   Used when the value is non-null. Throws NullPointerException if the value is null.

java

Copy code

Optional<String> optional = Optional.of("Hello, World!");

1. **Optional.ofNullable(value)**  
   Used when the value can be null. Creates an empty Optional if the value is null.

java

Copy code

Optional<String> optional = Optional.ofNullable(null);

1. **Optional.empty()**  
   Creates an empty Optional.

java

Copy code

Optional<String> optional = Optional.empty();

**Common Methods in Optional**

1. **isPresent()**  
   Checks if a value is present.

java

Copy code

if (optional.isPresent()) {

System.out.println(optional.get());

}

1. **ifPresent(Consumer<? super T> action)**  
   Executes the given action if the value is present.

java

Copy code

optional.ifPresent(value -> System.out.println("Value: " + value));

1. **orElse(T other)**  
   Returns the value if present; otherwise, returns the default value.

java

Copy code

String result = optional.orElse("Default Value");

1. **orElseGet(Supplier<? extends T> supplier)**  
   Similar to orElse, but the default value is computed using a supplier.

java

Copy code

String result = optional.orElseGet(() -> "Computed Default Value");

1. **orElseThrow(Supplier<? extends X> exceptionSupplier)**  
   Throws an exception if the value is absent.

java

Copy code

String result = optional.orElseThrow(() -> new IllegalStateException("No value present"));

1. **map(Function<? super T, ? extends U> mapper)**  
   Applies a function to the value if present and returns an Optional of the result.

java

Copy code

Optional<Integer> length = optional.map(String::length);

1. **filter(Predicate<? super T> predicate)**  
   Returns an Optional describing the value if it matches the predicate.

java

Copy code

Optional<String> filtered = optional.filter(value -> value.startsWith("H"));

1. **flatMap(Function<? super T, Optional<U>> mapper)**  
   Similar to map, but avoids nested Optional.

java

Copy code

Optional<Integer> result = optional.flatMap(value -> Optional.of(value.length()));

**Example: Using Optional to Avoid Null Checks**

Here’s a complete example of how to use Optional in a practical scenario:

java

Copy code

import java.util.Optional;

public class OptionalExample {

public static void main(String[] args) {

// Example of Optional.ofNullable

String name = null;

Optional<String> optionalName = Optional.ofNullable(name);

// Using orElse

String result = optionalName.orElse("Default Name");

System.out.println(result); // Output: Default Name

// Using ifPresent

optionalName.ifPresent(value -> System.out.println("Name: " + value));

// Example with a map

Optional<String> nonNullName = Optional.of("Java 8");

Optional<Integer> nameLength = nonNullName.map(String::length);

nameLength.ifPresent(length -> System.out.println("Name length: " + length));

// Using orElseThrow

String value = nonNullName.orElseThrow(() -> new IllegalArgumentException("Name not present"));

System.out.println(value); // Output: Java 8

}

}

**Benefits of Optional**

1. **Avoids NullPointerException**: Eliminates the need for explicit null checks.
2. **Readable Code**: Explicitly conveys the possibility of absence.
3. **Functional Programming Style**: Allows chaining operations for more concise and expressive code.

**Limitations**

1. Should not be used for every nullable field, especially in performance-critical scenarios.
2. It is not a replacement for all null checks, but a complement.

# **ava Garbage Collection**

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

### Advantage of Garbage Collection

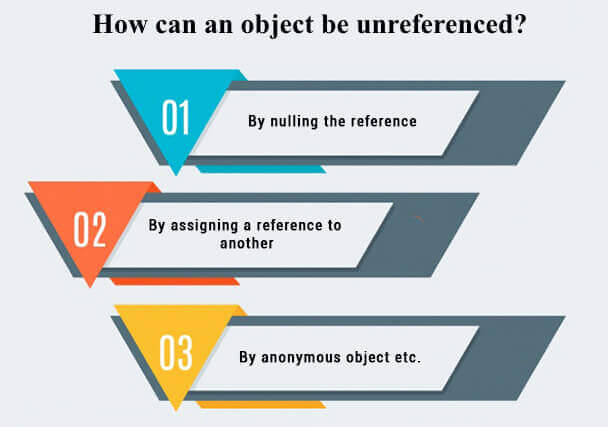
* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

## How can an object be unreferenced?

There are many ways:

Backward Skip 10sPlay VideoForward Skip 10s

* By nulling the reference
* By assigning a reference to another
* By anonymous object etc.



### 1) By nulling a reference:

1. Employee e=**new** Employee();
2. e=**null**;

### 2) By assigning a reference to another:

1. Employee e1=**new** Employee();
2. Employee e2=**new** Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection

### 3) By anonymous object:

1. **new** Employee();

## finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

1. **protected** **void** finalize(){}

#### **Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).**

## gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

1. **public** **static** **void** gc(){}

#### **Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.**

### Simple Example of garbage collection in java

1. **public** **class** TestGarbage1{
2. **public** **void** finalize(){System.out.println("object is garbage collected");}
3. **public** **static** **void** main(String args[]){
4. TestGarbage1 s1=**new** TestGarbage1();
5. TestGarbage1 s2=**new** TestGarbage1();
6. s1=**null**;
7. s2=**null**;
8. System.gc();
9. }
10. }

[Test it Now](javascript:void(0))

*object is garbage collected*

*object is garbage collected*

#### **Note: Neither finalization nor garbage collection is guaranteed.**