Strings in Java

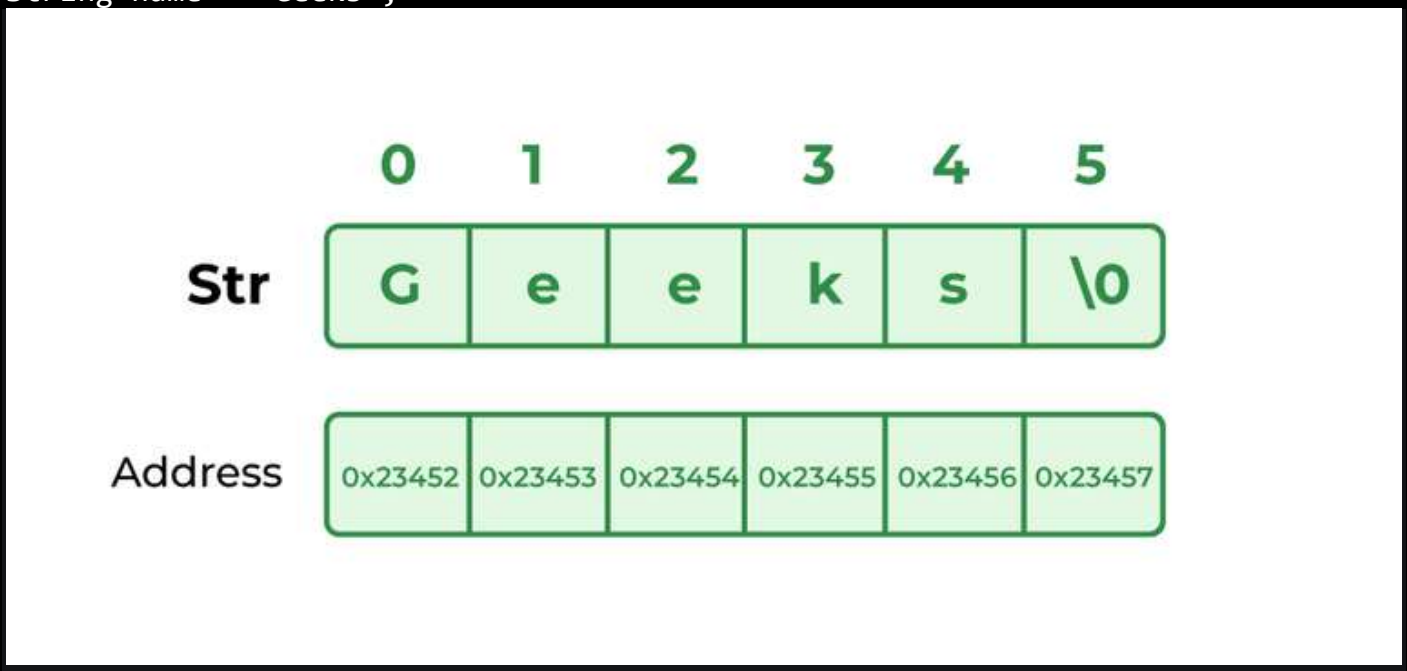
In the given example only one object will be created. Firstly JVM will not find any string object with the value “Welcome” in the string constant pool, so it will create a new object. After that it will find the string with the value “Welcome” in the pool, it will not create a new object but will return the reference to the same instance. In this article, we will learn about Java Strings.

What are Strings in Java?

Strings are the type of objects that can store the character of values and in Java, every character is stored in 16 bits i,e using UTF 16-bit encoding. A string acts the same as an array of characters in Java.

Example:

```
String name = "Geeks";
```



String Example in Java

Below is an example of a String in Java:

- Java

```java
// Java Program to demonstrate
// String
public class StringExample {


    // Main Function
    public static void main(String args[])
    {
        String str = new String("example");
```

```
        // creating Java string by new keyword
        // this statement create two object i.e
        // first the object is created in heap
        // memory area and second the object is
        // created in String constant pool.

        System.out.println(str);
    }
}
```

Output

```
example
```

Ways of Creating a String

There are two ways to create a string in Java:

- String Literal
- Using new Keyword

Syntax:

```
<String_Type> <string_variable> = "<sequence_of_string>";
```

1. String literal

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

Example:

```
String demoString = “GeeksforGeeks”;
```

2. Using new keyword

- String s = new String(“Welcome”);
- In such a case, JVM 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 the heap (non-pool)

Example:

```
String demoString = new String (“GeeksforGeeks”);
```

Interfaces and Classes in Strings in Java

CharBuffer: This class implements the CharSequence interface. This class is used to allow character buffers to be used in place of CharSequences. An example of such usage is the regular-expression package java.util.regex.

String: It is a sequence of characters. In Java, objects of String are immutable which means a constant and cannot be changed once created.

CharSequence Interface

CharSequence Interface is used for representing the sequence of Characters in Java. Classes that are implemented using the CharSequence interface are mentioned below and these provides much of functionality like substring, lastoccurence, first occurence, concatenate , toupper, tolower etc.

1. String
2. StringBuffer
3. StringBuilder

1. String

String is an immutable class which means a constant and cannot be changed once created and if wish to change , we need to create an new object and even the functionality it provides like toupper, tolower, etc all these return a new object , its not modify the original object. It is automatically thread safe.

Syntax

```
String str= "geeks";
          or
String str= new String("geeks")
```

2. StringBuffer

StringBuffer is a peer class of **String** that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences means it is immutable in nature and it is thread safe class , we can use it when we have multi threaded environment and shared object of string buffer i.e, used by mutiple thread. As it is thread safe so there is extra overhead, so it is mainly used for multithreaded program.

Syntax:

```
StringBuffer demoString = new StringBuffer("GeeksforGeeks");
```

3. StringBuilder

StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters and it is not thread safe and its used only within the thread , so there is no extra overhead , so it is mainly used for single threaded program.

Syntax:

```
StringBuilder demoString = new StringBuilder();
demoString.append("GFG");
```

StringTokenizer

StringTokenizer class in Java is used to break a string into tokens.

Example:

A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed. A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

StringJoiner is a class in *java.util* package which is used to construct a sequence of characters(strings) separated by a delimiter and optionally starting with a supplied prefix and ending with a supplied suffix. Though this can also be done with the help of the StringBuilder class to append a delimiter after each string, StringJoiner provides an easy way to do that without much code to write.

Syntax:

```
public StringJoiner(CharSequence delimiter)
```

Above we saw we can create a string by String Literal.

String demoString =”Welcome”;

Here the JVM checks the String Constant Pool. If the string does not exist, then a new string instance is created and placed in a pool. If the string exists, then it will not create a new object. Rather, it will return the reference to the same instance. The cache that stores these string instances is known as the String Constant pool or String Pool. In earlier versions of Java up to JDK 6 String pool was located inside PermGen(Permanent Generation) space. But in JDK 7 it is moved to the main heap area.

Immutable String in Java

In Java, string objects are immutable. Immutable simply means unmodifiable or unchangeable. Once a string object is created its data or state can’t be changed but a new string object is created.

Below is the implementation of the topic:

- Java

```java
// Java Program to demonstrate Immutable String in Java
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        String s = "Sachin";

        // concat() method appends
        // the string at the end
        s.concat(" Tendulkar");

        // This will print Sachin
        // because strings are
        // immutable objects
        System.out.println(s);
    }
}
```

Output

```
Sachin
```

Here Sachin is not changed but a new object is created with “Sachin Tendulkar”. That is why a string is known as immutable.
As you can see in the given figure that two objects are created but s reference variable still refers to “Sachin” and not to “Sachin Tendulkar”. But if we explicitly assign it to the reference variable, it will refer to the “Sachin Tendulkar” object.

For Example:

- Java

```java
// Java Program to demonstrate Explicitly assigned strings
```

```java
import java.io.*;


class GFG {
    public static void main(String[] args)
    {
        String name = "Sachin";
        name = name.concat(" Tendulkar");
        System.out.println(name);
    }
}
```

Output

```
Sachin Tendulkar
```

Memory Allotment of String

Whenever a String Object is created as a literal, the object will be created in the String constant pool. This allows JVM to optimize the initialization of String literal.

Example:

```
String demoString = "Geeks";
```

The string can also be declared using a **new** operator i.e. dynamically allocated. In case of String are dynamically allocated they are assigned a new memory location in the heap. This string will not be added to the String constant pool.

Example:

```
String demoString = new String("Geeks");
```

If you want to store this string in the constant pool then you will need to “intern” it.

Example:

```
String internedString = demoString.intern();
// this will add the string to string constant pool.
```

It is preferred to use String literals as it allows JVM to optimize memory allocation.
An example that shows how to declare a String

- Java

```java
// Java code to illustrate String
import java.io.*;
import java.lang.*;


class Test {
    public static void main(String[] args)
```

```
    {
        // Declare String without using new operator
        String name = "GeeksforGeeks";

        // Prints the String.
        System.out.println("String name = " + name);

        // Declare String using new operator
        String newString = new String("GeeksforGeeks");

        // Prints the String.
        System.out.println("String newString = " + newString);
    }
}
```

Output

```
String Stringname = GeeksforGeeks
String newString = GeeksforGeeks
```

Note: *String objects are stored in a special memory area known as string constant pool.*

Why did the String pool move from PermGen to the normal heap area?

PermGen space is limited, the default size is just 64 MB. it was a problem with creating and storing too many string objects in PermGen space. That's why the String pool was moved to a larger heap area. To make Java more memory efficient, the concept of string literal is used. By the use of the 'new' keyword, The JVM will create a new string object in the normal heap area even if the same string object is present in the string pool.

For example:

```
String demoString = new String("Bhubaneswar");
```

Let us have a look at the concept with a Java program and visualize the actual JVM memory structure:

Below is the implementation of the above approach:

- Java

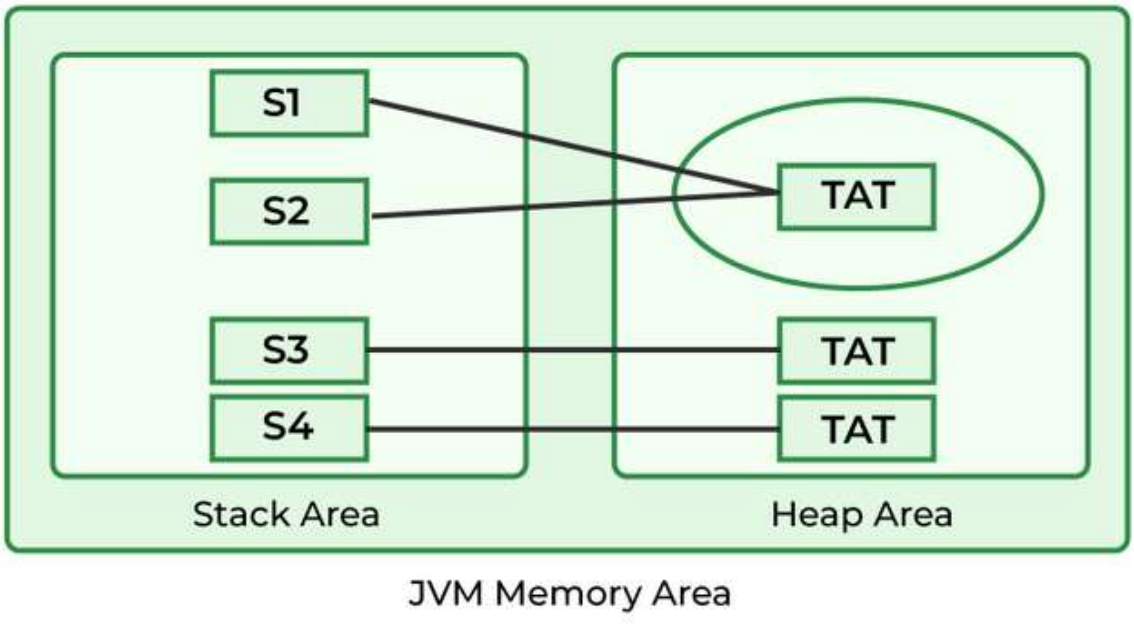
```
// Java program and visualize the
// actual JVM memory structure
// mentioned in image
class StringStorage {
```

```
    public static void main(String args[])
    {
        // Declaring Strings
        String s1 = "TAT";
        String s2 = "TAT";
        String s3 = new String("TAT");
        String s4 = new String("TAT");

        // Printing all the Strings
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
        System.out.println(s4);
    }
}
```

Output

```
TAT
TAT
TAT
TAT
```



String Pool in Java

Note: *All objects in Java are stored in a heap. The reference variable is to the object stored in the stack area or they can be contained in other objects which puts them in the heap area also.*

Example 1:

- Java

```
// Construct String from subset of char array


// Driver Class
class GFG {
    // main function
    public static void main(String args[])
    {
        byte ascii[] = { 71, 70, 71 };

        String firstString = new String(ascii);
        System.out.println(firstString);

        String secondString = new String(ascii, 1, 2);
        System.out.println(secondString);
    }
}
```

Output

```
GFG
FG
```

Example 2:

- Java

```
// Construct one string from another


class GFG {
```

```
    public static void main(String args[])
    {

        char characters[] = { 'G', 'f', 'g' };

        String firstString = new String(characters);
        String secondString = new String(firstString);

        System.out.println(firstString);
        System.out.println(secondString);
    }
}
```

Output

```
Gfg
Gfg
```

Frequently Asked Questions

1. What are strings in Java?

Strings are the types of objects which can store characters as elements.

2. Why string objects are immutable in Java?

Because java uses the concept of string literal. Suppose there are 5 reference variables, all refer to one object “Sachin”. If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why string objects are immutable in Java.

3. 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).

String class in Java

The string is a sequence of characters. In Java, objects of String are immutable which means a constant and cannot be changed once created.

Creating a String

There are two ways to create string in Java:

1. String literal

```
String s = “GeeksforGeeks”;
```

2. Using new *keyword*

```
String s = new String (“GeeksforGeeks”);
```

String Constructors in Java

1. String(byte[] byte_arr)

Construct a new String by decoding the *byte array*. It uses the platform’s default character set for decoding.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};
String s_byte =new String(b_arr); //Geeks
```

2. String(byte[] byte_arr, Charset char_set)

Construct a new String by decoding the *byte array*. It uses the *char_set* for decoding.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};
Charset cs = Charset.defaultCharset();
String s_byte_char = new String(b_arr, cs); //Geeks
```

3. String(byte[] byte_arr, String char_set_name)

Construct a new String by decoding the *byte array*. It uses the *char_set_name* for decoding. It looks similar to the above constructs and they appear before similar functions but it takes the *String(which contains char_set_name)* as parameter while the above constructor takes *CharSet*.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};
String s = new String(b_arr, "US-ASCII"); //Geeks
```

4. String(byte[] byte_arr, int start_index, int length)

Construct a new string from the *bytes array* depending on the *start_index(Starting location)* and *length(number of characters from starting location)*.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};
String s = new String(b_arr, 1, 3); // eek
```

5. String(byte[] byte_arr, int start_index, int length, Charset char_set)

Construct a new string from the *bytes array* depending on the *start_index(Starting location)* and *length(number of characters from starting location)*.Uses *char_set* for decoding.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};
Charset cs = Charset.defaultCharset();
String s = new String(b_arr, 1, 3, cs); // eek
```

6. String(byte[] byte_arr, int start_index, int length, String char_set_name)

Construct a new string from the *bytes array* depending on the *start_index(Starting location)* and *length(number of characters from starting location)*.Uses *char_set_name* for decoding.

Example:

```
byte[] b_arr = {71, 101, 101, 107, 115};
String s = new String(b_arr, 1, 4, "US-ASCII"); // eeks
```

7. String(char[] char_arr)

Allocates a new String from the given *Character array*

Example:

```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};
String s = new String(char_arr); //Geeks
```

8. String(char[] char_array, int start_index, int count)

Allocates a String from a given *character array* but choose *count* characters from the *start_index*.

Example:

```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};
String s = new String(char_arr , 1, 3); //eek
```

9. String(int[] uni_code_points, int offset, int count)

Allocates a String from a *uni_code_array* but choose *count* characters from the *start_index*.

Example:

```
int[] uni_code = {71, 101, 101, 107, 115};
String s = new String(uni_code, 1, 3); //eek
```

10. String(StringBuffer s_buffer)

Allocates a new string from the string in *s_buffer*

Example:

```
StringBuffer s_buffer = new StringBuffer("Geeks");
String s = new String(s_buffer); //Geeks
```

11. String(StringBuilder s_builder)

Allocates a new string from the string in *s_builder*

Example:

```
StringBuilder s_builder = new StringBuilder("Geeks");
String s = new String(s_builder); //Geeks
```

String Methods in Java

1. int length()

Returns the number of characters in the String.

```
"GeeksforGeeks".length();  // returns 13
```

2. Char charAt(int i)

Returns the character at ith index.

```
"GeeksforGeeks".charAt(3); // returns  ‘k’
```

3. String substring (int i)

Return the substring from the ith index character to end.

```
"GeeksforGeeks".substring(3); // returns “ksforGeeks”
```

4. String substring (int i, int j)

Returns the substring from i to j-1 index.

```
"GeeksforGeeks".substring(2, 5); // returns “eks”
```

5. String concat(String str)

Concatenates specified string to the end of this string.

```
String s1 = ”Geeks”;
String s2 = ”forGeeks”;
String output = s1.concat(s2); // returns “GeeksforGeeks”
```

6. int indexOf (String s)

Returns the index within the string of the first occurrence of the specified string.

```
String s = ”Learn Share Learn”;
int output = s.indexOf(“Share”); // returns 6
```

7. int indexOf (String s, int i)

Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

```
String s = ”Learn Share Learn”;
int output = s.indexOf("ea",3);// returns 13
```

8. Int lastIndexOf(String s)

Returns the index within the string of the last occurrence of the specified string.

```
String s = ”Learn Share Learn”;
int output = s.lastIndexOf("a"); // returns 14
```

9. boolean equals(Object otherObj)

Compares this string to the specified object.

```
Boolean out = “Geeks”.equals(“Geeks”); // returns true
Boolean out = “Geeks”.equals(“geeks”); // returns false
```

10. boolean equalsIgnoreCase (String anotherString)

Compares string to another string, ignoring case considerations.

```
Boolean out= “Geeks”.equalsIgnoreCase(“Geeks”); // returns true
Boolean out = “Geeks”.equalsIgnoreCase(“geeks”); // returns true
```

11. int compareTo(String anotherString)

Compares two string lexicographically.

```
int out = s1.compareTo(s2);
// where s1 and s2 are
// strings to be compared
This returns difference s1-s2. If :
out < 0  // s1 comes before s2
out = 0  // s1 and s2 are equal.
out > 0   // s1 comes after s2.
```

12. int compareToIgnoreCase(String anotherString)

Compares two string lexicographically, ignoring case considerations.

```
int out = s1.compareToIgnoreCase(s2);
// where s1 and s2 are
// strings to be compared
This returns difference s1-s2. If :
out < 0  // s1 comes before s2
out = 0   // s1 and s2 are equal.
out > 0   // s1 comes after s2.
```

Note: *In this case, it will not consider case of a letter (it will ignore whether it is uppercase or lowercase).*

13. String toLowerCase()

Converts all the characters in the String to lower case.

```
String word1 = “HeLLo”;
String word3 = word1.toLowerCase(); // returns “hello"
```

14. String toUpperCase()

Converts all the characters in the String to upper case.

```
String word1 = “HeLLo”;
String word2 = word1.toUpperCase(); // returns “HELLO”
```

15. String trim()

Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

```
String word1 = “ Learn Share Learn “;
String word2 = word1.trim(); // returns “Learn Share Learn”
```

16. String replace (char oldChar, char newChar)

Returns new string by replacing all occurrences of *oldChar* with *newChar*.

```
String s1 = “feeksforfeeks“;
String s2 = “feeksforfeeks”.replace(‘f’ ,’g’); // returns “geeksgorgeeks”
```

Note: *s1 is still feeksforfeeks and s2 is geeksgorgeeks*

17. String contains(string):

Returns true if string contains contains the given string

```
String s1="geeksforgeeks"
String s2="geeks"
s1.contains(s2) // return true
```

Example of String Constructor and String Methods

Below is the implementation of the above mentioned topic:

- Java

```java
// Java code to illustrate different constructors and methods

// String class.

import java.io.*;

import java.util.*;


// Driver Class

class Test

{

      // main function

    public static void main (String[] args)

    {
```

```
        String s= "GeeksforGeeks";
        // or String s= new String ("GeeksforGeeks");

        // Returns the number of characters in the String.
        System.out.println("String length = " + s.length());

        // Returns the character at ith index.
        System.out.println("Character at 3rd position = "
                           + s.charAt(3));

        // Return the substring from the ith  index character
        // to end of string
        System.out.println("Substring " + s.substring(3));

        // Returns the substring from i to j-1 index.
        System.out.println("Substring  = " + s.substring(2,5));

        // Concatenates string2 to the end of string1.
        String s1 = "Geeks";
        String s2 = "forGeeks";
        System.out.println("Concatenated string  = " +
                            s1.concat(s2));

        // Returns the index within the string
        // of the first occurrence of the specified string.
        String s4 = "Learn Share Learn";
        System.out.println("Index of Share " +
                           s4.indexOf("Share"));

        // Returns the index within the string of the
        // first occurrence of the specified string,
        // starting at the specified index.
        System.out.println("Index of a  = " +
```

```
                    s4.indexOf('a',3));

        // Checking equality of Strings
        Boolean out = "Geeks".equals("geeks");
        System.out.println("Checking Equality  " + out);
        out = "Geeks".equals("Geeks");
        System.out.println("Checking Equality  " + out);

        out = "Geeks".equalsIgnoreCase("gEeks ");
        System.out.println("Checking Equality " + out);

        //If ASCII difference is zero then the two strings are similar
        int out1 = s1.compareTo(s2);
        System.out.println("the difference between ASCII value is="+out1);
        // Converting cases
        String word1 = "GeeKyMe";
        System.out.println("Changing to lower Case " +
                           word1.toLowerCase());

        // Converting cases
        String word2 = "GeekyME";
        System.out.println("Changing to UPPER Case " +
                           word2.toUpperCase());

        // Trimming the word
        String word4 = " Learn Share Learn ";
        System.out.println("Trim the word " + word4.trim());

        // Replacing characters
        String str1 = "feeksforfeeks";
        System.out.println("Original String " + str1);
        String str2 = "feeksforfeeks".replace('f' ,'g') ;
        System.out.println("Replaced f with g -> " + str2);
```

```
    }
}
```

Output :

```
String length = 13
Character at 3rd position = k
Substring ksforGeeks
Substring = eks
Concatenated string = GeeksforGeeks
Index of Share 6
Index of a = 8
Checking Equality false
Checking Equality true
Checking Equality false
the difference between ASCII value is=-31
Changing to lower Case geekyme
Changing to UPPER Case GEEKYME
Trim the word Learn Share Learn
Original String feeksforfeeks
Replaced f with g -> geeksgorgeeks
```