Contents

[Java String 1](#_Toc162615349)

[How to create a string object? 2](#_Toc162615350)

[1) String Literal 2](#_Toc162615351)

[2) By new keyword 2](#_Toc162615352)

[String constantpool 2](#_Toc162615353)

[Need of String Constant Pool: 4](#_Toc162615354)

[Interning of String in Java 6](#_Toc162615355)

[Java StringBuffer class 7](#_Toc162615356)

[Java StringBuilder class 8](#_Toc162615357)

[StringTokenizer class in Java with example 10](#_Toc162615358)

[Immutable: 11](#_Toc162615359)

# 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:

char[] ch={'j','a','v','a','t','p','o','i','n','t'};

String s=new String(ch);

String s=new String(“dfsf”);

is same as:

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.

The java.lang.String class implements Serializable, Comparable and CharSequence [interfaces](https://www.javatpoint.com/interface-in-java).



# 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:

String s="welcome";

## 2) By new keyword

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

# String constantpool

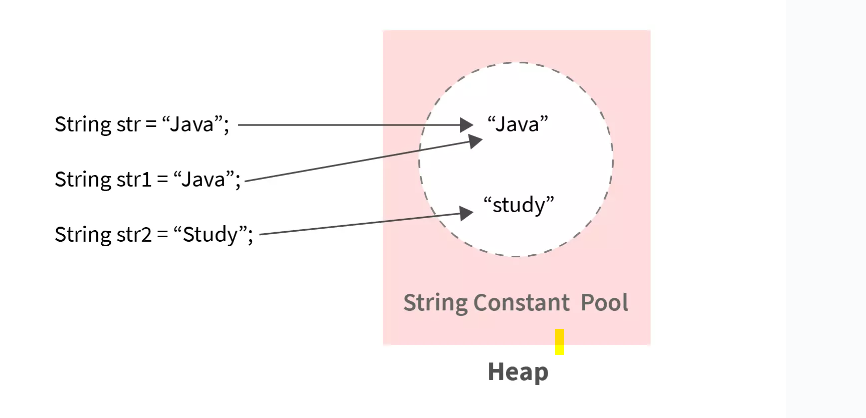
String Pool in Java is a special storage space in [Java](https://www.scaler.com/topics/java/) Heap memory where string literals are stored. It is also known by the names - String Constant Pool or String Intern Pool. Whenever a string literal is created, the JVM first checks the String Constant Pool before creating a new String object corresponding to it

* When we create a string literal, the JVM first checks that literal in the String Constant Pool. If the literal is already present in the pool, its reference is stored in the variable.
* However, if the string literal is not found, the JVM creates a new string object in the String Constant Pool and returns its reference.

String str = "Java";

String str1 = "Java";

String str2 = "Study";

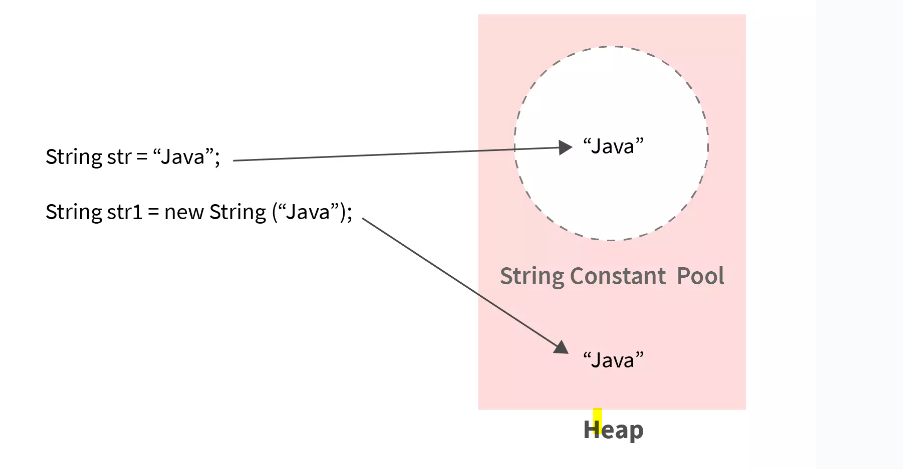


String str = "Java";

String str1 = new String("Java");

Both expressions give you a String object, but there is a subtle difference between them. When you create a String object using the new() operator, it always creates a new object in [heap memory](http://java67.blogspot.sg/2013/08/guide-of-javalangoutofmemoryerror-java-heap-space-tomcat-eclipse-minecraft-jboss.html).

On the other hand, if you create an object using String literal syntax e.g. "Java", it may return an existing object from String pool (a cache of String object in Perm gen space, which is now moved to heap space in recent Java release), if it already exists.



String a = "Java";

String b = "Java";

System.out.println(a == b); // True

Here two different objects are created and they have different references:

String c = new String("Java");

String d = new String("Java");

System.out.println(c == d); // False

Similarly, when you compare a string literal with a String object created using new() operator using == operator, it will return false, as shown below :

String e = "JDK";

String f = new String("JDK");

System.out.println(e == f); // False

In this code snippet, how many String objects are created?

String str = "This is a string";

String str2 = "This is a string";

String str3 = new String("This is a string");

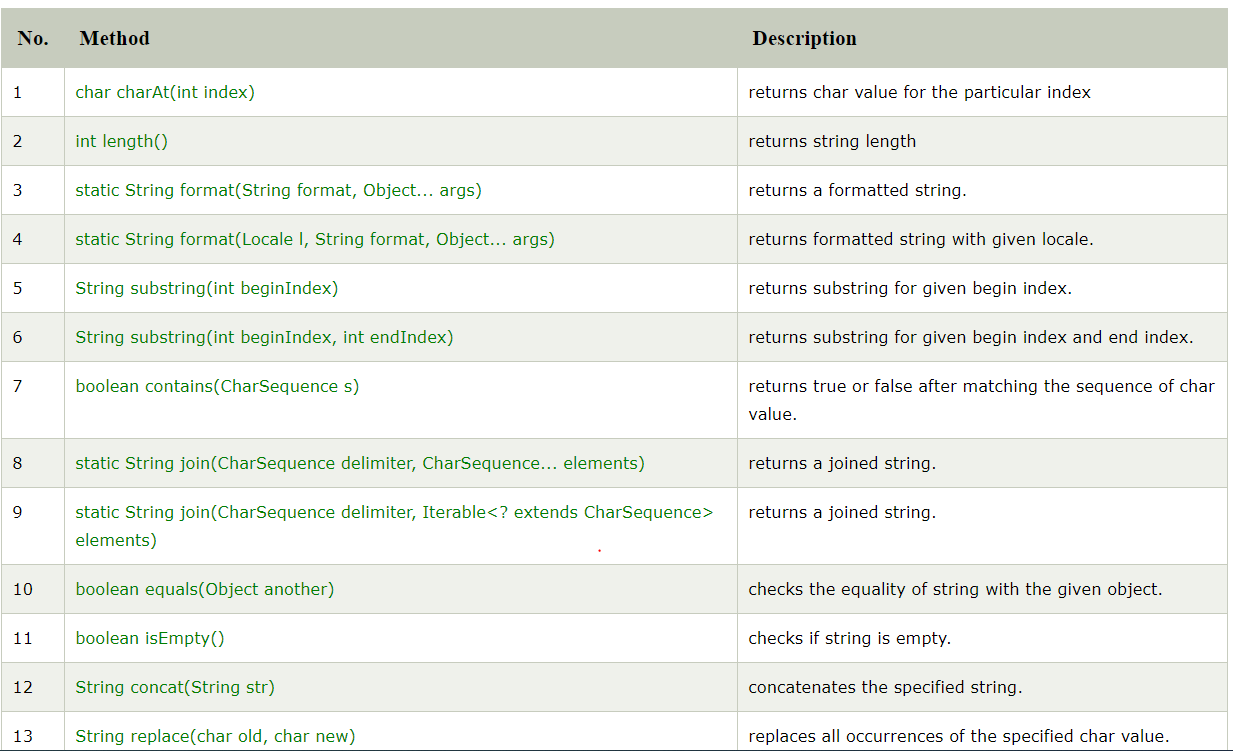
The answer is: 2 String objects are created. str and str2 both refer to the same object. str3 has the same content but using new forced the creation of a new, distinct, object.

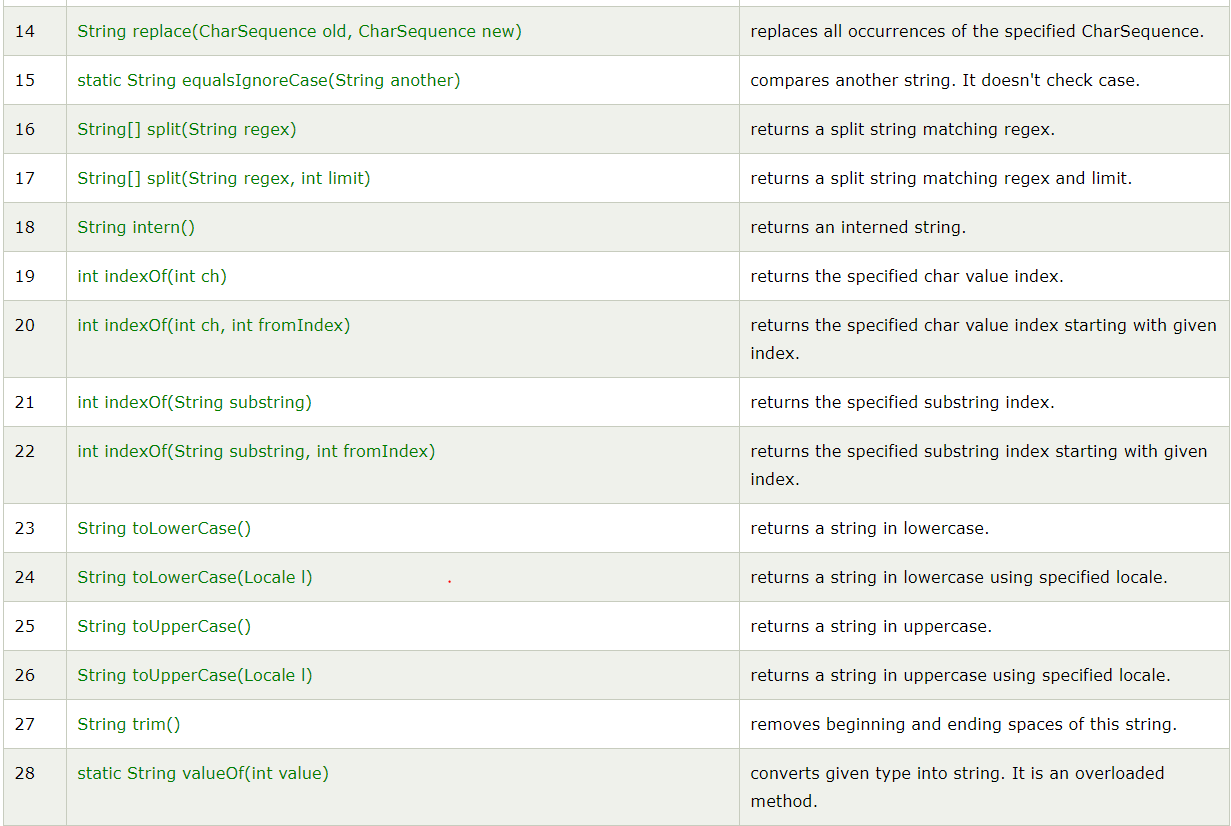
When you create a String literal, the JVM internally checks, what is known as the String pool, to see if it can find a similar (content wise) String object. If it finds it, it returns the same reference. Otherwise, it just goes ahead and creates a new String object in the pool so that the same check can be performed in the future.

## Need of String Constant Pool:

* When we create a String object, it uses some amount of space in the heap memory.
* Let's say we are creating n number of String objects with the same value, and distinct memory is allocated to each of these string objects (though they all contain the same string).
* This is an inefficient usage of heap memory. In order to escalate the performance of our code and reduce memory usage, JVM optimizes the way in which strings are stored with the help of a string constant pool.

Strings are immutable in nature. Immutable means they have a constant value, and even if they are altered, instead of reflecting the alterations in the original string, a new object is created. This immutability is achieved through String Pool





# Interning of String in Java

String Interning is a method of storing only one copy of each distinct String Value, which must be immutable.

The String interning ensures that all strings having the same contents use the same memory.

String str1 = "xyz";

String str2 = "xyz";

When we execute above code both will share same memory. So here string literals by defaults calls intern and makes sure both use same memory.

However, if you create strings with using the new keyword, these strings won't share the same memory.

String str1 = new String("xyz");

String str2 = new String("xyz");

Above code will use two different references in heap.

As you can see from this example, both str1 and str2 have the same content. However, they are not equal because they don't share the same memory.

In this case, you can manually use the intern() method so that the same memory is used for strings having the same content.

String str1 = new String("xyz");

String str2 = new String("xyz");

// str1 and str2 doesn't share the same memory pool

System.out.println(str1 == str2); // false

// using the intern() method

// now both str1 and str2 share the same memory pool

str1 = str1.intern();

str2 = str2.intern();

System.out.println(str1 == str2); // true

By applying String.intern() on a couple of strings will ensure that all strings having the same contents share the same memory. For example, if a name ‘Amy’ appears 100 times, by interning you ensure only one ‘Amy’ is actually allocated memory.

public class InternExample2 {

    public static void main(String[] args) {

        String s1 = "Javatpoint";          String s2 = s1.intern();

      String s3 = new String("Javatpoint");

        String s4 = s3.intern();

        System.out.println(s1==s2); // True

        System.out.println(s1==s3); // False

        System.out.println(s1==s4); // True

        System.out.println(s2==s3); // False

        System.out.println(s2==s4); // True

        System.out.println(s3==s4); // False

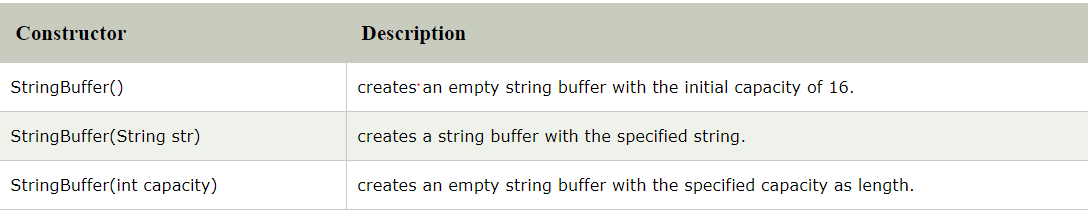
    }

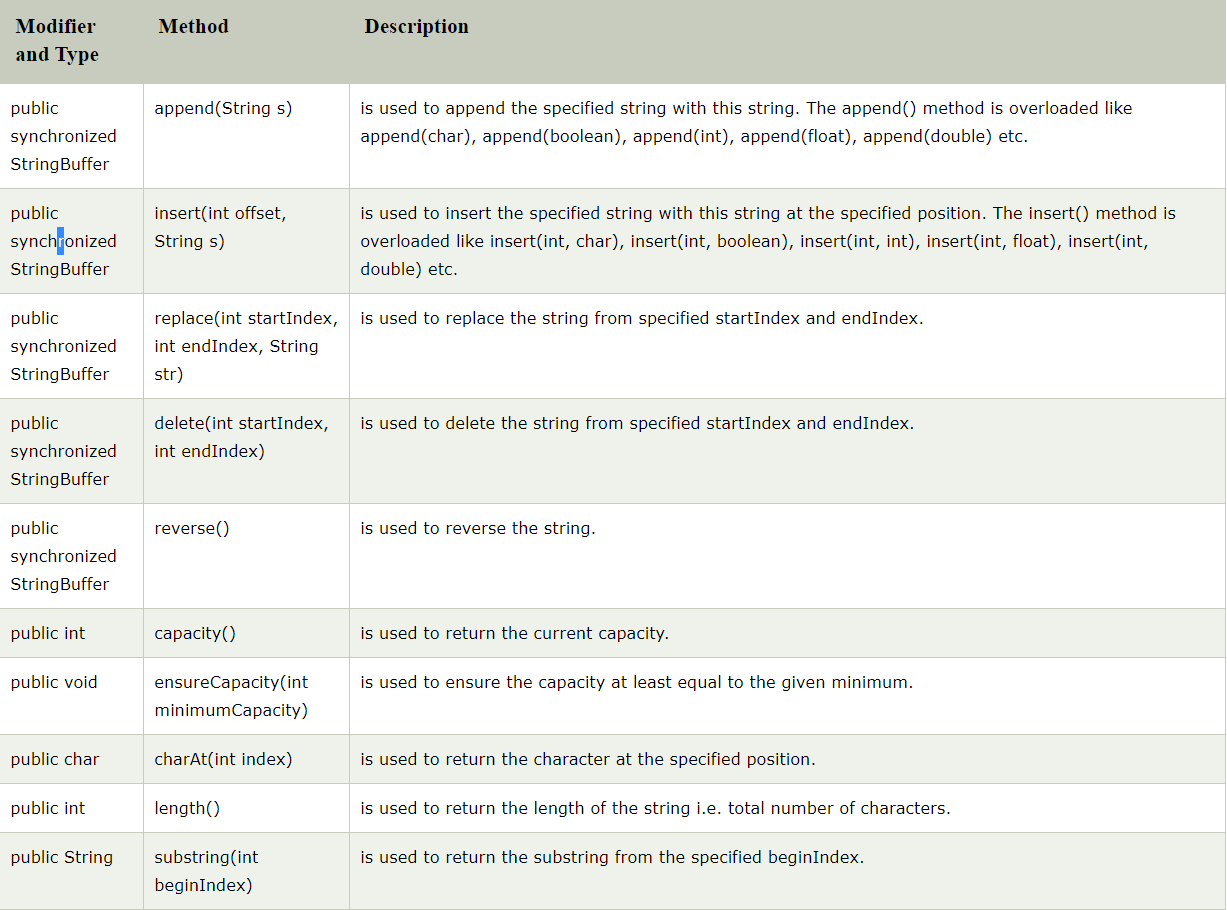
}

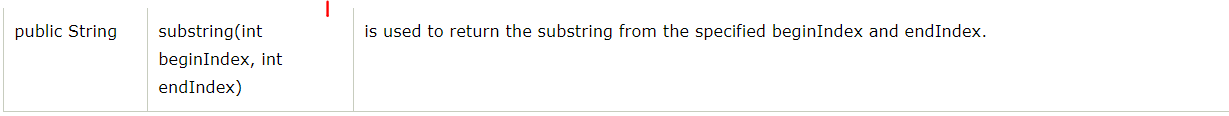
# Java StringBuffer class

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

StringBuffer is designed to minimize this copying. It uses a simple but clever internal data structure to avoid doing any copying at all until the very end, when you ask for the final String with a toString() call:

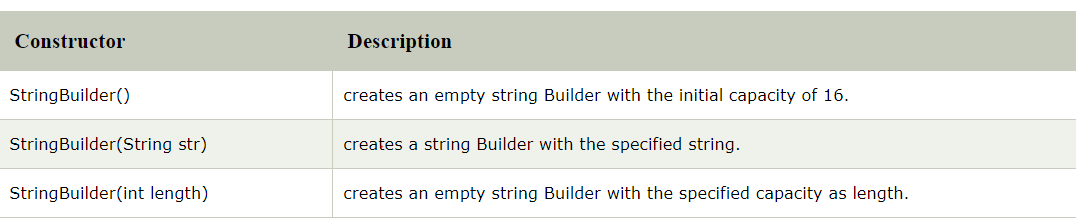
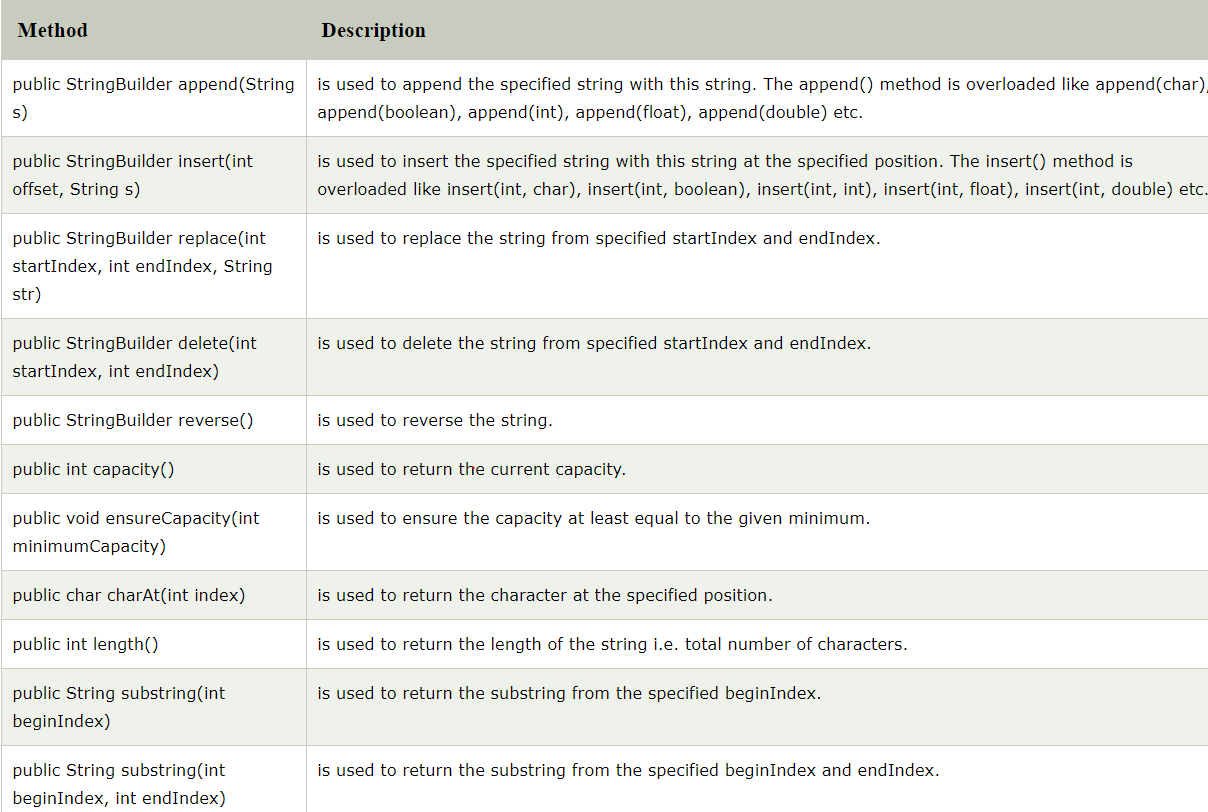


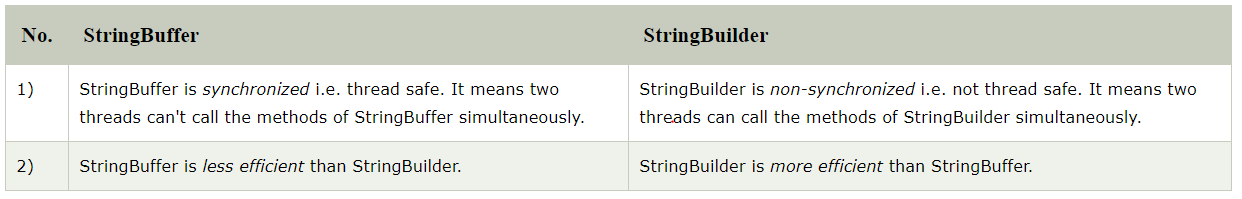




# Java StringBuilder class

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

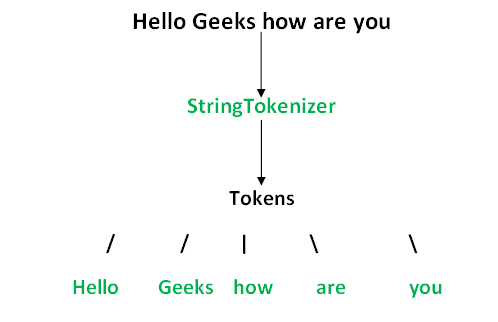
 



String is immutable ( once created can not be changed )object . The object created as a String is stored in the Constant String Pool. Every immutable object in Java is thread safe ,that implies String is also thread safe . String can not be used by two threads simultaneously.

# StringTokenizer class in Java with example

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.

**Constructors:**

**StringTokenizer(String str) :**

**str** is string to be tokenized.

Considers default delimiters like new line, space, tab,

carriage return and form feed.

**StringTokenizer(String str, String delim) :**

**delim** is set of delimiters that are used to tokenize

the given string.

**StringTokenizer(String str, String delim, boolean flag):**

The first two parameters have same meaning. The flag

serves following purpose.

If the **flag** is **false**, delimiter characters serve to

separate tokens. For example, if string is "hello geeks"

and delimiter is " ", then tokens are "hello" and "geeks".

If the **flag** is **true**, delimiter characters are

considered to be tokens. For example, if string is "hello

geeks" and delimiter is " ", then tokens are "hello", " "

and "geeks".

# Immutable:

**Immutable objects** are instances whose state doesn’t change after it has been initialized. For example, [String](https://www.journaldev.com/16928/java-string) is an immutable class and once instantiated its value never changes

An immutable class is good for caching purposes because you don’t have to worry about the value changes.

Another benefit of immutable class is that it is inherently [**thread-safe**](https://www.journaldev.com/1061/thread-safety-in-java), so you don’t have to worry about thread safety in case of multi-threaded environment.

Create custom immutable class

To create an immutable class in Java, you have to do the following steps.

1. Declare the class as final so it can’t be extended.
2. Make all fields private so that direct access is not allowed.
3. Don’t provide setter methods for variables.
4. Make all **mutable fields final** so that its value can be assigned only once.
5. Initialize all the fields via a [constructor](https://www.journaldev.com/18899/constructor-in-java) performing deep copy.
6. Perform [cloning](https://www.journaldev.com/60/java-clone-object-cloning-java) of objects in the getter methods to return a copy rather than returning the actual object reference.

To understand points 4 and 5, let’s run the sample Final class that works well and values don’t get altered after instantiation.

// Java Program to Create An Immutable Class

// Importing required classes

import java.util.HashMap;

import java.util.Map;

// Class 1

// An immutable class

final class Student {

// Member attributes of final class

private final String name;

private final int regNo;

private final Map<String, String> metadata;

// Constructor of immutable class

// Parameterized constructor

public Student(String name, int regNo,

Map<String, String> metadata)

{

// This keyword refers to current instance itself

this.name = name;

this.regNo = regNo;

// Creating Map object with reference to HashMap

// Declaring object of string type

Map<String, String> tempMap = new HashMap<>();

// Iterating using for-each loop

for (Map.Entry<String, String> entry :

metadata.entrySet()) {

tempMap.put(entry.getKey(), entry.getValue());

}

this.metadata = tempMap;

}

// Method 1

public String getName() { return name; }

// Method 2

public int getRegNo() { return regNo; }

// Note that there should not be any setters

// Method 3

// User -defined type

// To get meta data

public Map<String, String> getMetadata()

{

// Creating Map with HashMap reference

Map<String, String> tempMap = new HashMap<>();

for (Map.Entry<String, String> entry :

this.metadata.entrySet()) {

tempMap.put(entry.getKey(), entry.getValue());

}

return tempMap;

}

}

// Class 2

// Main class

class GFG {

// Main driver method

public static void main(String[] args)

{

// Creating Map object with reference to HashMap

Map<String, String> map = new HashMap<>();

// Adding elements to Map object

// using put() method

map.put("1", "first");

map.put("2", "second");

Student s = new Student("ABC", 101, map);

// Calling the above methods 1,2,3 of class1

// inside main() method in class2 and

// executing the print statement over them

System.out.println(s.getName());

System.out.println(s.getRegNo());

System.out.println(s.getMetadata());

// Uncommenting below line causes error

// s.regNo = 102;

map.put("3", "third");

// Remains unchanged due to deep copy in constructor

System.out.println(s.getMetadata());

s.getMetadata().put("4", "fourth");

// Remains unchanged due to deep copy in getter

System.out.println(s.getMetadata());

}

}