Strings: Java

Java is a computer Object oriented programming language. It contains different types of objects namely primitive, derivative and customized. Each object has its own class and can be accessed by a reference variable. Methods can be applied on those reference variables to manipulate the objects which are placed in a memory called Java Heap. Each and every object of Java is inherited from the base Object. Among various objects in Java String is an interesting object and widely used probably in each and every java program ever written.

**What is a String object?**

String is a derivative object in Java. It represents a sequence of characters (16 bit Unicode), but unlike C/C++ it doesn’t end with a null value. Like every object String has its own constructors through which String objects can be created using the ‘new’ keyword. There are quite a lot of constructors. But the interesting part is in most of the cases a String object is not created using the ‘new’ keyword. Rather it can be created like primitive objects though JVM will ultimately create a new String object. Here comes the one of the many interesting parts of String object in Java.

First and foremost thing to remember String objects are immutable, i.e. once created cannot be changed. Then how come it can be used in codes so many times? We will come to it shortly. String objects can be created without using the ‘new’ keyword. Also String objects like all other objects though are placed in the heap memory String literals has special space in the heap known as “String pool constant” unlike any other objects. Also String class has his two very useful brother classes StringBuffer and StringBuilder. Both the classes contains exactly same methods, but the difference is StringBuffer is thread safety i.e. all methods are synchronized while StringBuilder aren’t and is little faster. Also StringBuilder and StringBuffer allows to manipulate objects unlike their elder brother String class.

**Creating String Objects**

String like any objects can be created by the ‘new’ keyword followed by any of the various constructors. Below are few of the commonly used examples of creating a String object in a Java program.

* String str = new String();
* String str = new String(“Abcd”);
* Char[] char = {‘A’, ‘b’, ‘c’, ‘d,};

String str = new String (char)

* String str = “Abcd”;

The last one is the most commonly and preferable way used by developers to create String objects. There are some differences in between all these options but the common thing is that they all create a new String object with a value “Abcd” and assign it to the reference variable str. But as said earlier all the String objects are immutable or frozen. Once a value is assigned to it, that value cannot be changed. But their reference variable is not. So how does it work? We will see look into it just now.

**How String object reference variable works and how immutable String objects can be shown differently:**

Suppose we want to create a String. So we would write the code as,

String str = “Hello World!”;

Now the JVM will create a new String object with String literals “Hello World”. If we write,

System.out.println (“Value of str: ” + str);

//Output: “Hello World!”

Up to this everything seems to be same like any other normal java objects. But let’s say we want to add some more String literals or characters after “Hello Word!”. We want the String to change to “Hello World! How are you?” i.e. we want to *change* String object which is immutable, unchangeable. It seems like it is impossible and if that is the case no code developed by Java which uses String objects is scalable or changeable. But it is possible.

String class has a method called concat(String str), which appends one string to the end of another. Let’s see the below code using String#concat(String str) method.

String str = “Hello World!”;

Now the JVM will create a new String object with String literals “Hello World”. If we write,

System.out.println (“Value of str: ” + str);

//Output: “Value of str: Hello World!”

str = str.concat (”How are you?”)

System.out.println (“Value of str: ” + str);

//Output: “Value of str: Hello World! How are you?” ---- Same reference different output.

So how does the *immutable* String object gets mutable or changed here?

Actually in this case not one but 2 String objects got created and the reference variable is assigned to the last one and printed it. The first object contains “Hello World!” and the second one “Hello World! How are you?”

The first was is still there as an object in the memory unless garbage collection has happened. But since no reference variable is attached to it, it cannot be accessed. The below picture explain it.

Step 1: New String object is created with reference variable str



Step 2: Another String Object is created as "Hello World! How are you?" The dotted line represents deleted reference



Here we can see, the immutable first string remained same. Instead the String#concat (String str) method created a new String object with new literal values and assigned the reference variable to it from the older one. The older one though present sadly now has now has no way to get access. But what if we need to access both the objects? This can be done by assigning a 2nd variable which will point towards the first Object as well or assign it to the reference variable already created. The String#concat (String str) method will be applied to any of the one, which will eventually create a newer String object. Now the one will point towards the older String (“Hello String”), the other will point towards the newer String (“Hello World! How are you?”). This ways with both the reference variable we can

Let’s see the below code.

String str = “Hello World!”;

Now the JVM will create a new String object with String literals “Hello World”. If we write,

System.out.println (“Value of str: ” + str);

//Output: “Value of str: Hello World!”

String str2 = str;

System.out.println (“Value of str: ” + str);

System.out.println (“Value of str2: ” + str2);

//Output: “Value of str: Hello World!”

//Output: “Value of str2: Hello World!”

str = str.concat (”How are you?”)

System.out.println (“Value of str: ” + str);

System.out.println (“Value of str2: ” + str2);

//Output:

“Value of str: Hello World! How are you?” ---- Reference str pointing to the second object.

“Value of str2: Hello World!” ---- Reference str2 still pointing towards the first object

The below picture:

1Step 1: New String object is created with reference variable str



Step 1: No new String object is created but here both the reference variables points towards the sane String object



Step 1: New String object is created ("How are you?"). Reference variable str now refers newer String object while the reference object str2 points to the first String object ("Hello World!"). str has lost its connection with first String object.



Now we have two Java objects and two reference variables. Each variable is pointing to one of the objects. So we can access both of them. None of the object is lost for us.

**Differences of assigning and non-assigning reference variables while working on them:**

We have already seen how the method String#concat(String str) can create a newer String and display it. But a very small change in the code can return the same old value even after concatenating it with a newer string value. Let’s see the following code.

String str = “Hello World!”;

Now the JVM will create a new String object with String literals “Hello World”. If we write,

System.out.println (“Value of str: ” + str);

//Output: “Value of str: Hello World!”

str.concat (”How are you?”) --- This line is important

System.out.println (“Value of str: ” + str);

//Output: “Value of str: Hello World!” ---- Same reference, older output.

Here the method String#concat (String str) is applied on the reference variable str. So JVM will create a new String object and keep it in the heap. But no reference variable is assigned to the newer String. So even though the newer String is present we cannot access it. See the first code example where we assigned str to str.concat (“How are you?”), so the reference variable got linked with the newer String and gave the new output. Same thing will happen if we use any other method without assigning it to a variable.

**Important facts about Strings and memory:**

Memory management and efficient use of memory is one of the vital and key goals of any programming language and software. As in Java code Strings are used in high numbers it is expected that String objects might take up a large amount of space often with lot of redundancy within the whole gamut of String literals used in the program, eventually affecting the program/software.

But Java has a beautiful thing called “String constant pool” within the main heap memory to overcome the above mentioned issue. All String literals are stored in the specially created “String constant pool”.

When a new String object is get created the compiler encounters the String literal of the new object and checks it with the existing String literals in the pool. If the newer String literal matches with an already existing one in the pool, no new String object gets created. The compiler simply directed the new reference variable with the existing one and returns it. The already existing String object just have a new reference variable linked to it.

This also makes sense the reason behind String objects being immutable. If one reference variable makes changes the other still remains as it is. But in some cases it might be required to change a String object at one place and reflect it to each and every places or reference variables directed to it. For that we have the StringBuilder/ StringBuffer classes. We will discuss it soon.

//TODO: Add image

But here there is something we can delve into different ways to String creation or the subtle differences between the creations of String objects in different ways. Let’s take a look at the code below.

Step 1:

String str = “Hello World!”;

This simple creates one String object and one reference variable directed to it. The String literal “Hello World” will go in the pool. Any new String created with same content will refer to it from the ‘String Constant Pool’.

Step 2:

String str = new String (“Hello World!”);

In this case as the ‘new’ keyword is used JVM will create a new String object in the normal heap, not in the “String constant pool” and the reference variable str will be linked to it. Additionally the String literal “Hello World!” will be placed in the “String constant pool” as well. Basically two objects and one reference variable are created in this case. That’s why developers mostly use the previous one while creating String.

**Brother classes: StringBuffer and StringBuilder**

StringBuffer and StringBuilder classes are used when lots of manipulation and modification are required to strings of characters. Strings are immutable. So if a great deal of manipulation on String objects are required it will eventually end up with lots of abandoned Strings objects in the String pool, which is definitely not a good practice of programming. But, on the other side these two classes allows to modify the object repeatedly without leaving behind a large number of discarded String objects.

**StringBuffer vs StringBuilder**