Understanding String in Java

Abhiroop Nandi Ray
Sep 10, 2018 · 14 min read

Understanding String, StringBuilder, StringBuffer: Java

Java is a Object oriented computer 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 or not, 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 Object'* 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. in C Strings are basically null terminated (\'0')character arrays. In Java however Strings are object. Internally it might be converted to character arrays but that will not be exposed to developers. Though if required String objects can be transferred to character arrays.

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 primitives, though JVM will ultimately create a new String object. Here comes one of the many interesting parts of String object in Java.

First and foremost thing to remember **Java 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.

As said earlier 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. Along with that String class has his two very useful sibling(not exactly, but in metamorphic way) 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 the objects directly unlike their elder sibling(again in metamorphic way) String class.

Creating String Objects

String like any objects can be created by the ‘new’ keyword followed by any of the various constructors and without the ‘new’ keyword. 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. So in the above example, the String objects with string literals “Abcd” are now rigid and inflexible. But with their reference variable we do can print changed String values. 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 and permanent. 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?”) ———— This line is creating the magic

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 assigned to the last one gets assigned to the newer one and printed. 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 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 a or assign it to the reference variable already created, which will anyway direct the second variable to the first Object. The String#concat (String str) method will be applied to any of the one variables, which will eventually create a newer String object. Now 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);

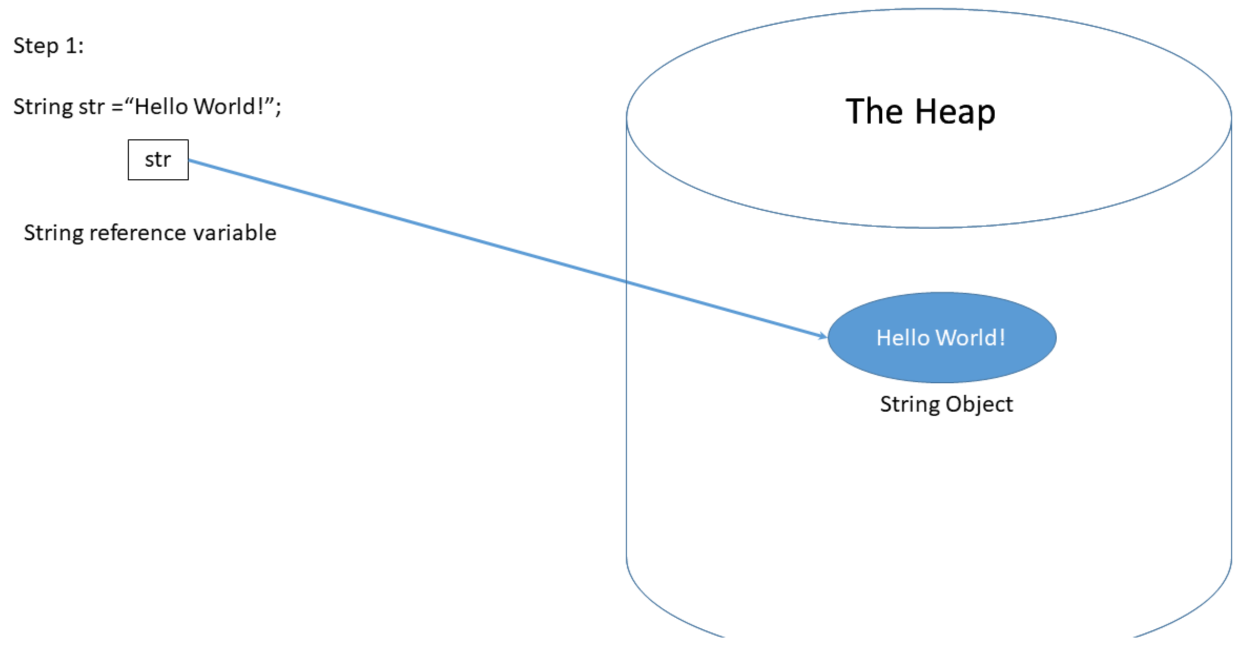
//Output:

“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

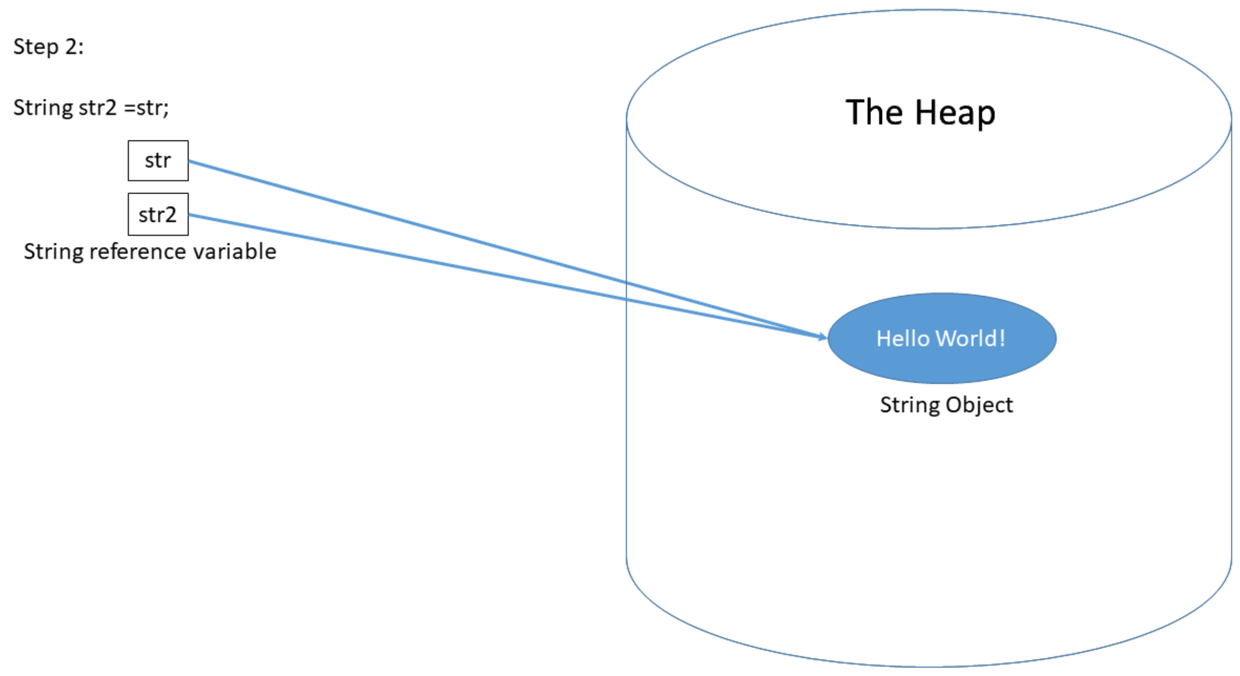
The below picture:

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



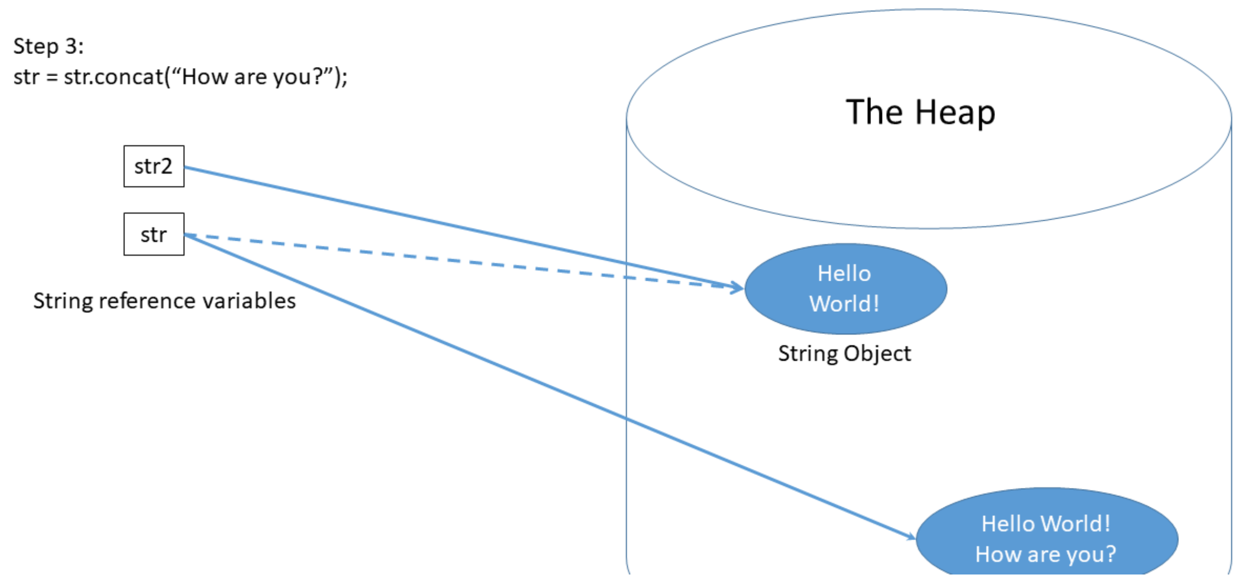
Single String object pointing to the object

Step 2: No new String object is created but here both the reference variables points towards the sane String object.



Two String objects pointing to same String object

Step 3: 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.


Step 3:
str = str.concat("How are you?");
str2
str
String reference variables
The Heap
Hello World!
String Object
Hello World! How are you?

Two String objects pointing to two different Strings

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. Remember String objects are immutable, so the objects remained same.

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.

String objects can also be concatenated using the '+' operator:

```
String str = “Hello World!”

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

//Output: Value of str: Hello World!

str = str + “ How are you?”

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

//Output: Value of str: Hello World! How are you?
```

Both the String#concat(String str) method and ‘+’ operator will give same output but internally they both works differently.

For String#concat(String str), it internally creates a new char array buffer, and returns a new string based on that char array.

For the + operator, the compiler in fact translate it to use StringBuffer/StringBuilder.
`str1 = new StringBuilder().append(str1).append(str2).toString();`

But, here is an important fact to remember. In case of loop ‘+’ operator may gives unwanted results. Please do not use ‘+’ operator inside a loop. Using String with ‘+’ operator instead of StringBuilder will slow down as Java itself needs to convert it to StringBuilder.

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 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 of the many reference variables makes changes the others still remain 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 place or reference variables directed to it. For that we have the StringBuilder/ StringBuffer classes. We will discuss it soon.

//TODO: Add image (Is it required !!??)

But here there is something we can delve into the different ways to String creation and the subtle differences between them. 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.

String comparisons:

Normally when we compare to objects in Java we use the ‘==’ operator. But in string it is advisable to use the .equals() method to compare two strings instead of the ‘==’ operator. Why? Let’s check the code below:

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

        String str1 = "Abcd";

        String str2 = "Abcd";

        System.out.println("String comparison with == oprator:  " +
str1 == str2);

        System.out.println("String comparison with .equalus() " +
str1.equals(str2));

        int a = 100;

        int b = 100;

        char char1 = 'a';

        char char2 = 'a';

        System.out.println("int comparison with == operator : " + (a
==b));

        System.out.println("char comparison with == operator : " +
(char1 == char2));

    }

}
```

Now let us check the output of the program:

```
false

String comparison with .equalus() true

int comparison with == operator : true

char comparison with == operator : true
```

We can see the '==' worked perfectly fine at the case of int or char but failed in case of String. Instead when we used the .equals() method on String objects they worked exactly as expected. First of all, equals() is a method and '==;.This is because The operator '==' operator compares memory locations, while equals() method compares the content stored in two objects.

Here str1 and str2 are two objects located at different memory locations even though they have same String literals. So when compared with '==' method it is giving output as *false* as they are actually not same objects according to the operator's job.

But as .equals() method compared the contents of str1 and str2, which is same in this case. It found them same and returned as *true*.

Few important things about .equals method:

· It only compares what is written to compare

- If a class does not override the .equals() method of it, then by default it goes to the equals(Object o)method of the closest parent class that has overridden this method.
- If neither of any parent classes have provided an overridden method of .equals(), then by default it points to the method from the ultimate parent class, Object.

As per the Object API this works same as ==; that is, it returns true *if and only if* both variables refer to the same object, if their references are one and the same. Thus it will test for **object equality** and not **functional equality**.

If a class overrides the .equals method, the hashcode() method is necessary to be overridden too.

Sibling 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 itself, repeatedly without leaving behind a large number of discarded String objects.

StringBuffer vs StringBuilder:

Both the classes are immutable (the class, not the objects) have exactly same methods, performs same features and returns same value. But StringBuffer class is thread safe and all the methods are synchronized, whereas StringBuilder methods are not thread safe and are not synchronized.

StringBuilder as non-synchronized runs a little faster than StringBuffer. It depends on the program requirement which class to use when. *Apart from that anything and everything is same for both the classes. From now onwards in code examples StringBuilder will be used, but all of then applies to StringBuffer class as well.*

Usages of StringBuffer/StringBuffer:

In these classes the methods invoke directly on the object unlike the String class. So when a method is invoked no newer object is created like String. Let's see the following code

Like String#concat(String str) which returns a new String literal with the value of the String passed in the method parameters and append it to the end of the older String(by creating a new String as again String objects are adamant and immutable) , there is a similar method in the StringBuilder class. In StringBuilder class there is a StringBuilder#append(String str) method which updates the value of the object on which the method is invoked. Basically if an old string needs to add newer character to its end, both the methods return similar (but not the same) value.

Code 1:

Using String class

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 with String class: " + str);

//Output: "Value of str: Hello World!"

str.concat ("How are you?") — — — — — *This line is important*

System.out.println ("Value of str with String class: " + str);

//Output: "Value of str with String class: Hello World!" — — Same reference different output.

Using StringBuilder class

StringBuilder 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 with StringBuilder class: ” + str);

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

str.append (”How are you?”) — — — — *This line is important*

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

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

As discussed earlier, in String class unless the reference variable is assigned it won’t change it, even though a newer String is object get created by the JVM, because the newer object doesn’t have a reference variable to access it. It is lost to us. But in the case of StringBuilder without assigning the value, we got the required new result. It happened because in StrinBuilder object the method acts directly on the object itself. So as we made the call, the same object got manipulated, appending on itself unlike String class, as well as not creating new object.

Why String class along with StringBuilder and StringBuffer:

1> String objects are immutable while StringBuilder/StringBuffer objects are mutable.

2> It is recommended to use String when value of string is non-changeable.

3> For simple string concatenation, StringBuilder is not a better choice, java compiler will help by itself and do the trick. However, if the requirement is to concatenate inside a loop, it is required to manually apply StringBuilder/StringBuffer.

4> StringBuffer is thread safe. It is recommended to use StringBuffer when value of string is changeable and application is implemented in multithreaded environment.

5> It is recommended to use StringBuilder when value of string is changeable but application is not using multiple threads.

6> Immutable String uses String Literal pool to store the objects whereas StringBuffer StringBuilder both use heap memory. So if similar string objects are created String saves memory that is why is StringBuffer/StringBuilder is used many times in the code it will lead to “memory out of bound error”.

7> As String is immutable CPU can cache the value of String object which makes the process run faster and improves speed.

8> String being immutable if multiple reference value points to the same String literal there is a chance of unwanted change in the String object(all places) if one reference value is manipulated.

#Java, #String, #JavaStringObjects, #JavaString

I am thankful to :

First of all my parents, family and friends

1> SCJP book by Kathy Sierra and Bert Bates

2> https://docs.oracle.com/javase/7/docs/api/

3> https://www.stackoverflow.com

4> https://www.quora.com

5> My employers and the projects they assigned to me.

Programming Java String Software Development