*“It is an immutable law in business that words are words, explanations are explanations, promises are promises-but only performance is reality.”*

* *Harold S. Geneen*

Dear readers, I am bringing to you what I had promised, ***memory management*** in Strings and ***Immutable*** nature of Strings.

Please watch this introductory video on [Strings](https://www.pepcoding.com/resources/online-java-foundation/string,-string-builder-and-arraylist/strings-intro-official/video) in Java, to learn about the basics of strings before jumping onto this article.

After watching the basic introduction to strings, you should watch [this video](https://www.pepcoding.com/resources/online-java-foundation/string,-string-builder-and-arraylist/string-interning-immutability/video) to learn about the concepts which I am going in this article as well.

***Memory Management & String Interning***

There are majorly two types of memory associated with a program, that are ***stack memory*** and ***heap memory***. You should keep in mind that only primitive data-types: *Byte, Short, Int, Long, Double, Float, Char* and *Boolean* are stored directly in stack memory.

Since, strings are non-primitive data types in Java, hence they are not stored directly in stack memory. Instead, they are created in the heap memory and just a reference to this location is held by the stack memory.

But, there is a catch here! There is a special area or segment of heap memory known as ***intern pool.*** Strings in Java are created in this area known as intern pool.

Let us take an example of three strings declared in the following manner:

1. String s1 = "hello";

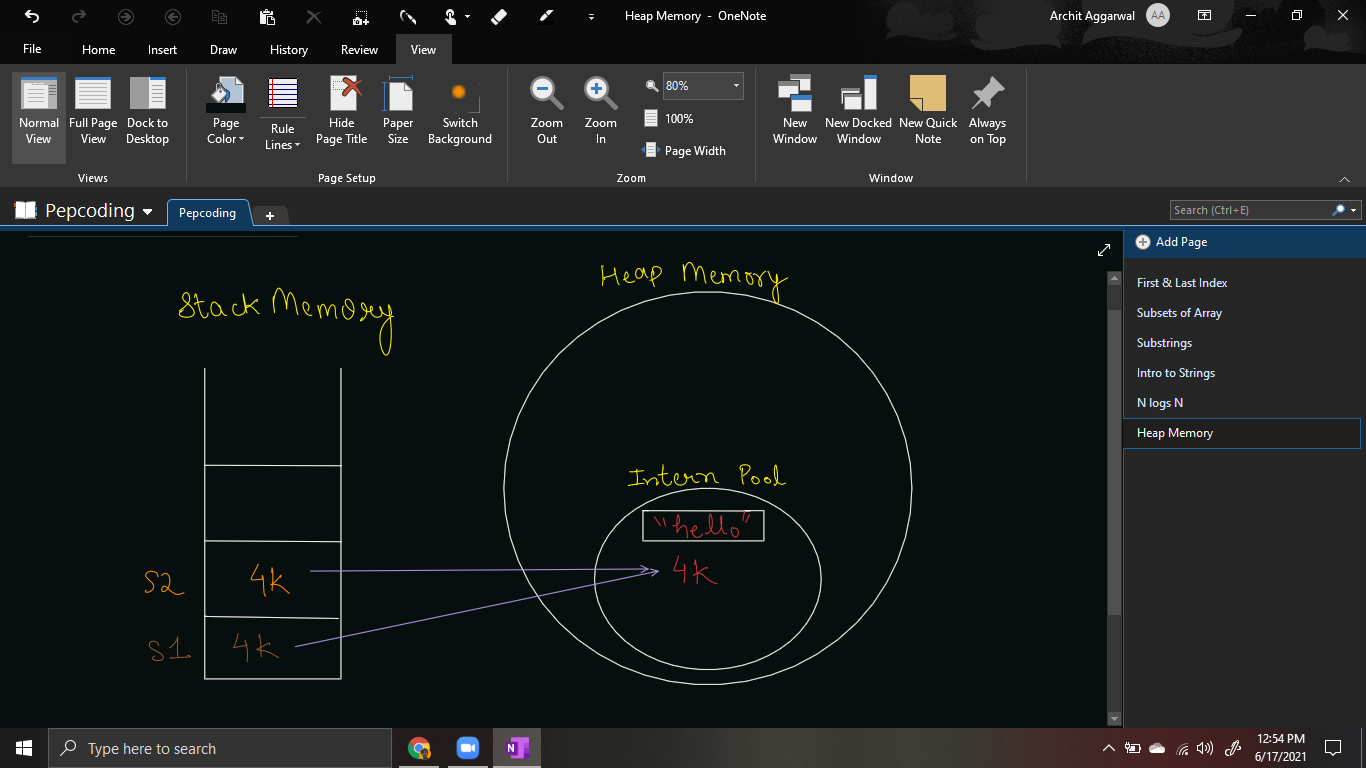
2. String s2 = "hello";

3. String s3 = new String("hello");

Statement 1, will make a string of 5 characters “hello” in the intern pool, let us suppose, at starting address = 4k. And this address (4k) is stored in the stack memory with s1 holding the address.

Now, Statement 2, will make another string of 5 characters “hello” in the intern pool….! Not really my friend, it is not that simple. Read carefully!

First, a character array *{‘h’,’e’,’l’,’l’,’o’}* is searched in the intern pool. If it already exists (which in this case, is yes as string at location 4k = “hello” is already present), then JVM (Java Virtual Machine) will not create another string “hello” in the intern pool. Instead, it will store the reference of the same 4k location for the s2 string as well. Hence, now 2 strings s1 and s2 point to the same “hello” literal in the intern pool.



If it would have been not present, then JVM would have created another string literal and allocated a new memory location to the String.

This process of searching for an identical string literal in the intern pool, and if present, *sharing* the same memory of string literal for different strings is known as ***String Interning in Java***.

We have covered ***WHAT*** is interning and ***HOW*** it takes place, but ***WHY*** interning takes place? What is the need or it’s advantages?

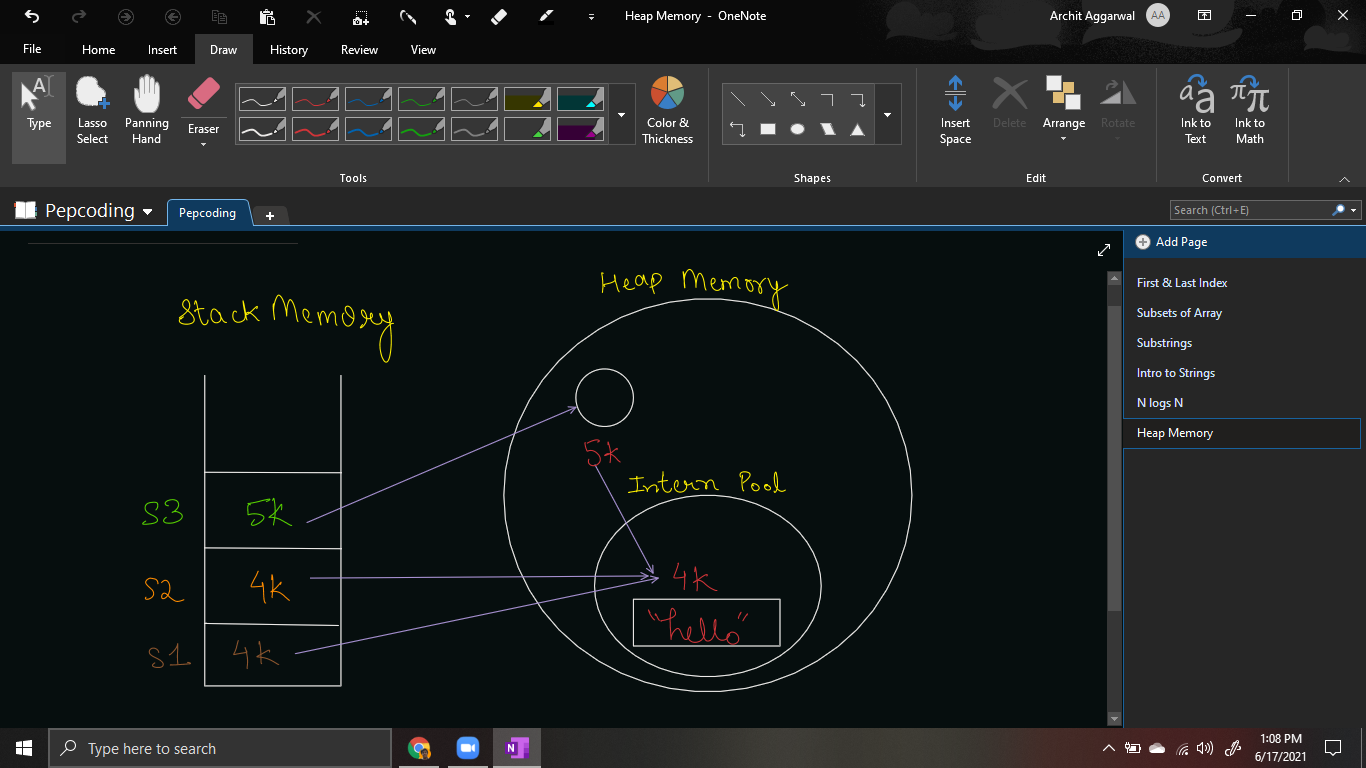
Well, it has a simple answer - To ***Optimize Space***. There can be multiple duplicate strings created throughout the program. We can save some space from heap memory by not allocating different character arrays for the same string literal.

Is there a way to ***avoid* String Interning**?

Yes, we can create the strings using the ***new*** keyword. This will create a new shell object outside of the intern pool (with a new address) for the string.

But, there is still a problem. If there will be an identical character array, then this shell object will point to the same character array.

For statement 3, a new shell object will be created at location, let it be 5k, but, in this case also, the *shell object* will point to the same character array at location 4k.



So, we have seen WHAT, HOW & WHY of Interning. But what are the impacts of doing ***String Interning?*** Two major impacts are the unexpected result in String comparisons and another impact is that it makes Strings Immutable.

**Comparison operator (==) vs equals() method**

We should avoid using == operator for comparing strings. Let us know why?

What do you think will be output of two comparison operations:

* Is s1 == s2?
* Is s1 == s3?

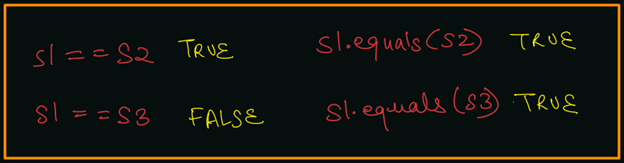
You might have guessed that those operations will give the result ‘true’. But, only the first operation will meet your expectations. The result of s1 == s3 will be ‘false’.

It is because, in stack memory, s1 is pointing to 4k memory location whereas s3 is pointing to a different memory location (5k), now JVM will not check whether the shell object of s3 at 5k is pointing to the same character array or not. It will just return false.

Hence, even after s1 = “hello” and s3 = “hello”, the result will be false as s3 is initialized using the new keyword.

To compare strings in java, we should make a habit to ***always use equals()*** method. *s1.equals(s2)* will be directly returned true as both point to the same (4k) memory location.

Now, *s1.equals(s3)* will not be returned false even if both point to different memory locations, instead they will now be checked whether they are equal character-by-character or not. Since both are “hello”, it will also return true.



**Q) What is IMMUTABILITY of Strings?**

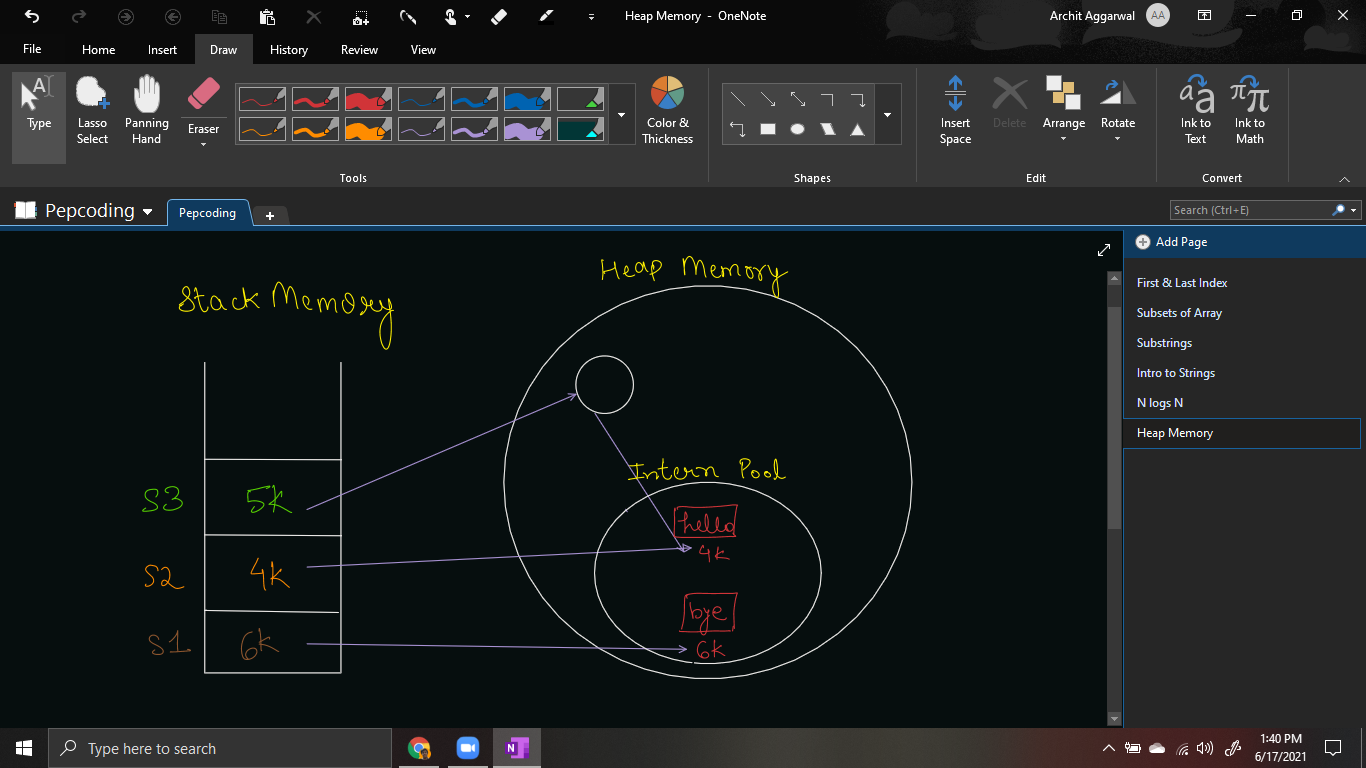
If you look for the meaning of immutability in English dictionary, you will find it means ‘which cannot be changed/modified’. Does it mean that Strings cannot be changed in Java? Well, a proper answer is:

***‘Reference is mutable, instance is not.’***

Once a string object is created, its data or state can't be changed but the string can point to another character array.

If we try to modify s1 as : s1 = “bye”, then what will happen? Will the character array {h, e, l, l, o} which is not only being pointed by s1 but also by s2 change?

No, but a new object will be created with a character array {b, y, e} and s1 will now start pointing to this new address location.



The original character array {h, e, l, l, o} will remain as it is. We cannot modify this instance, i.e. we cannot say set character at some index to some other character.

Hence, the reference of string (in stack memory) is mutable, but the instance (character array in intern pool) is immutable.

**Q) But, why does Immutability occur in the first place?**

Let us assume, if you were allowed to modify the instance of string, i.e. modify the character array {h, e, l, l, o} in the intern pool, then what would have been the consequences?

Since the character array at 4k location is pointed by more than 1 string, modifying the character array itself would have led to modification of all the strings which are pointing it.

For example), if we would have changed s1’s character at 3rd index to say ‘d’, then character array would have been modified to {h, e, d, l, o}, hence the character at 3rd index for the string s2 would also have changed to ‘d’.

This can be a surprise to the developer who does not know about the existence of string s1, but is working on a module which uses string s2. We would wonder why s2 has become “hedlo”, but he himself initialized it to “hello”.

Thus, to avoid such surprises, it is not allowed to modify the instance of strings in Java. ***Strings are immutable in java as an immediate cause of string interning***.

**Q) What is the impact of Immutability?**

Immutability can lead to ***low performance issues***, both in terms of space and time. Consider the following example:

Scanner scn = new Scanner(System.in);

int n = scn.nextInt();

String s = "";

for(int i=0; i<n; i++){

s += i;

}

System.out.println(s);

What do you think will be the time complexity of the above code? Will it be O(n), as we are adding only 1 character n times?

No, the time complexity of the above piece of code is as bad as O(n2). As we have discussed earlier, that instance of string cannot be modified, hence adding a character to the end of string does not mean it will get appended in the end of the character array.

Instead, even if a single character is appended, firstly JVM will create a new string object, copy all of the characters present to the new string, and then append the single character to the newly created string object. Hence we are first copying the entire string which is O(n) work for appending a single character.

And since we are appending single character for n times, the overall time complexity will turn out to be O(n \* n) = O(n2).

For example) to add character ‘d’ to string s = “hello”, we have to first make a new string object, then copy all characters {h, e, l, l, o} to the new string and append ‘d’ to the new string. Hence we are not operating on 1 character, but s.size() + 1 characters.

So, consider if the current string is taking 2GB space in memory, then adding a single character will lead to creation of a copy of the string of entire 2GB and appending 1 character (of few byte(s)) to it.

This is not what we expected! So, is there any way to make MUTABLE STRINGS IN JAVA? Yes, there is a concept of ***StringBuilder Class*** which we will learn in our next video lecture.

This much information would have been difficult to swallow in one go. Hence, I recommend you to watch the video and read the article ***again*** to have a better understanding.

Contributor : Archit Aggarwal