* **string s = "abcdef"** - this will create and address in stack (say at 4k).

Now that 4k will point to an address in heap which would contain "abc"

Heap

Stack

4000

4000

* **str.substr(2) = cdef;**//suru ka 2 chhor kar k baki sara
* will collect input until the user enters a '#'. Or any character we wish to

**getline(cin, mystring,'#');**

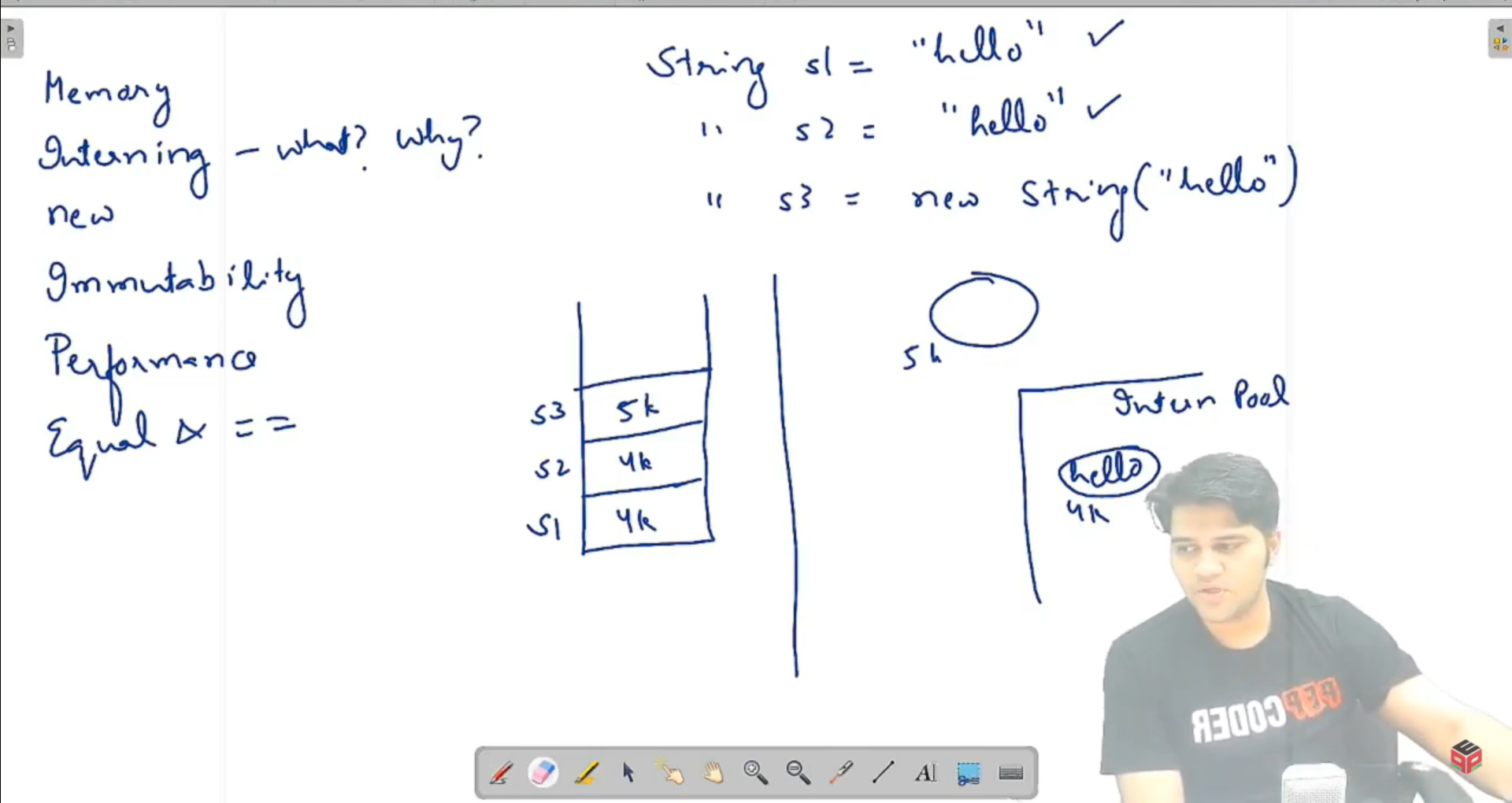
* After taking an integer an input, if you want to take a string as input, you need to take two inputs in two different strings

**using getline(cin, dummy); getline(cin, str);**

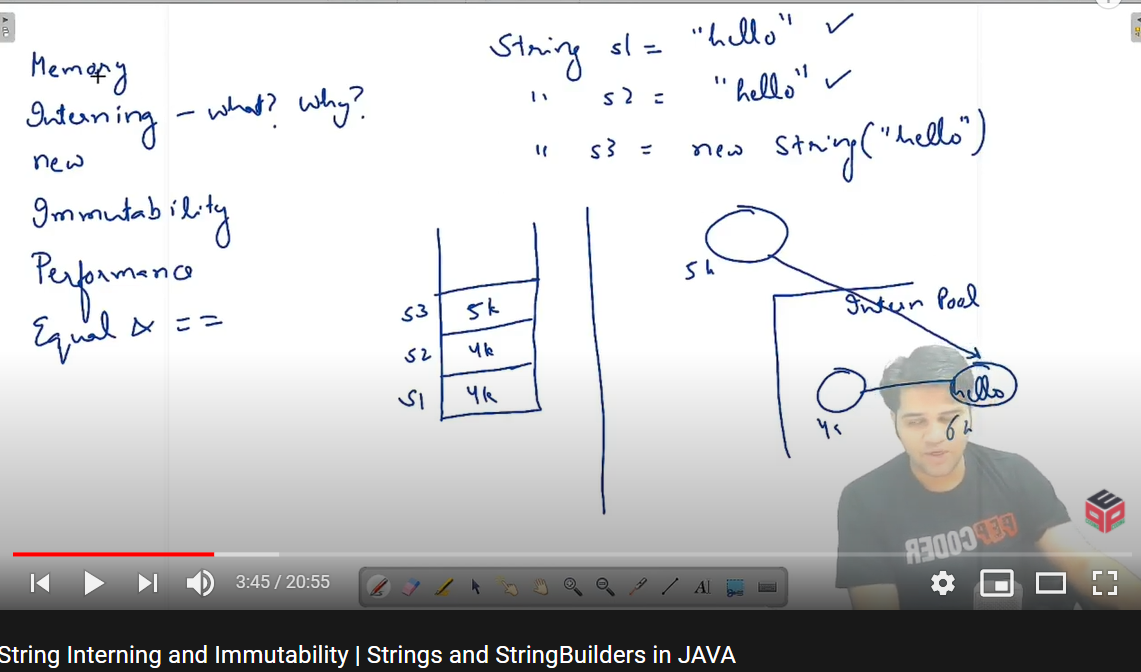
First one stores the pressing of enter and 2nd one actually st

MEMORY/NEW/IMMUTABILITY/PERFORMANCE ISSUES/EQUALS/DOUBLE EQUALS/INTERNITY

Interning: it means sharing memory, done by by strings(to optimize space)

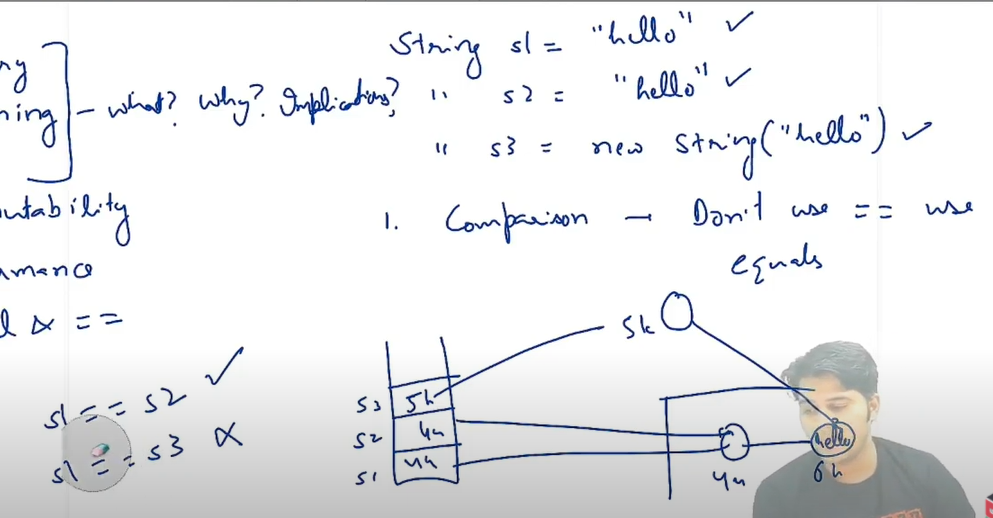


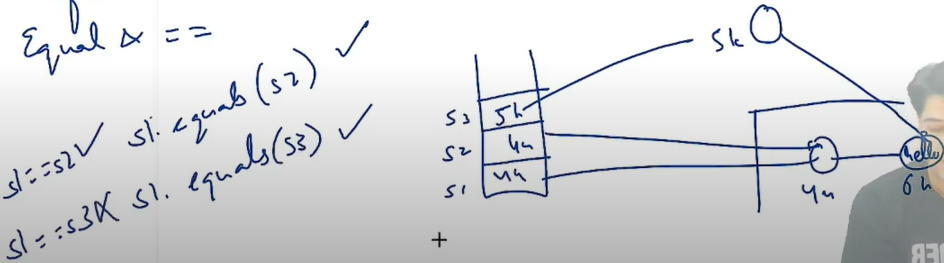
This is called interning. i.e. unless we use “new” keyword, s1 and s2 will be stored at only 1 address and not 2 addr. If their content is same



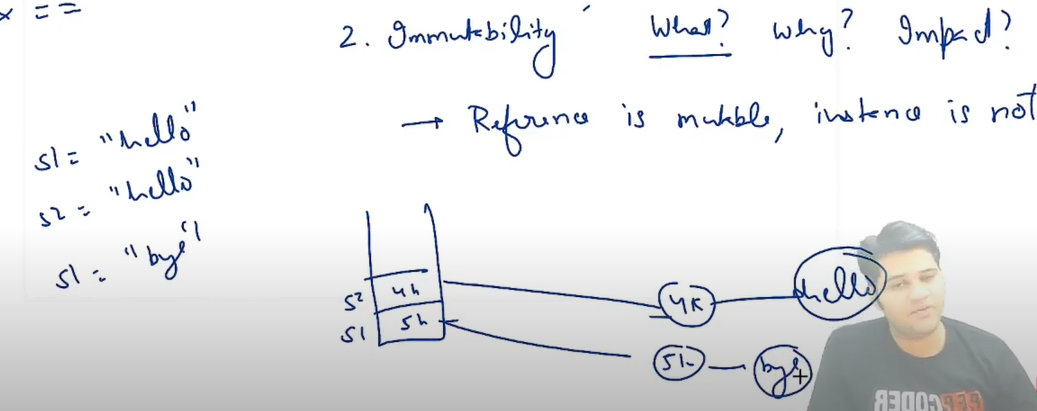
Actually 4k also doesn’t store “hello”. And in case we use “new”, 5k address is allocated to a shell and it also points to same “hello”

String comparison: “==” checks address. “equals” checks the actual content. So, always “equals ”





IMMUTABILITY:



S1 = “bye”; It will make a new address and store bye in that. So, reference/address is mutable but contents of strings are not.

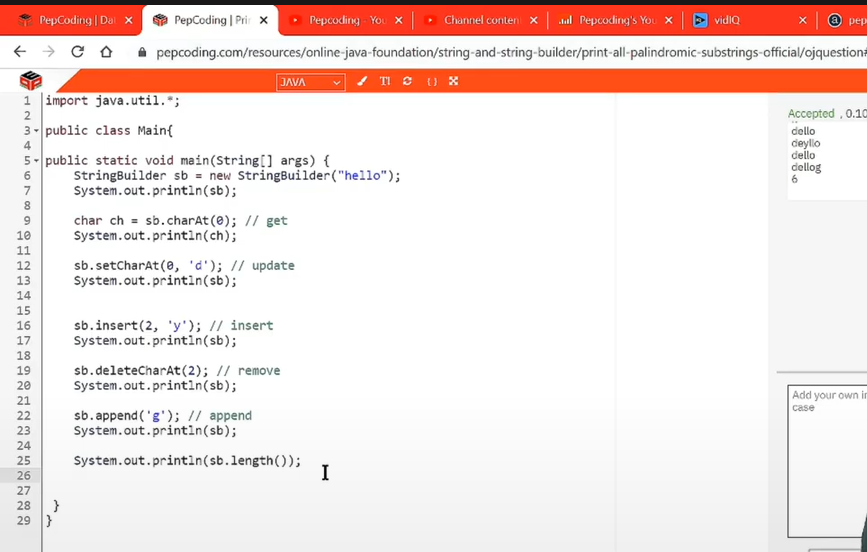
Why? So that if we change s1 characters, s2 would also get changed if they were mutable.

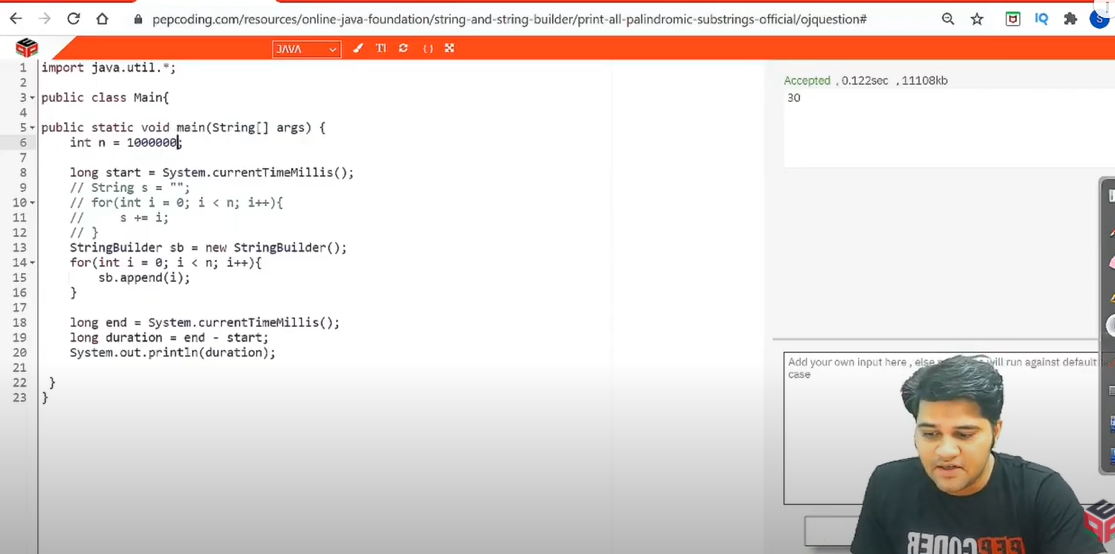
Impact: Performance issues. But we’ll optimize that using string builder

Note: C++ **std::string is mutable** and assignment (generally) copies the string data. Java String is immutable and assignment copies a reference to the string.

String Builders:

They’re mutable, unlike strings.





Here we see that string gave time limit exceeded on n = 105 (because in actual, the for loop would run for O(n2) which is 1010) whereas StringBuilder just worked fine. Because it runs in O(n) and that is because string builders are mutable.

**Questions below:**

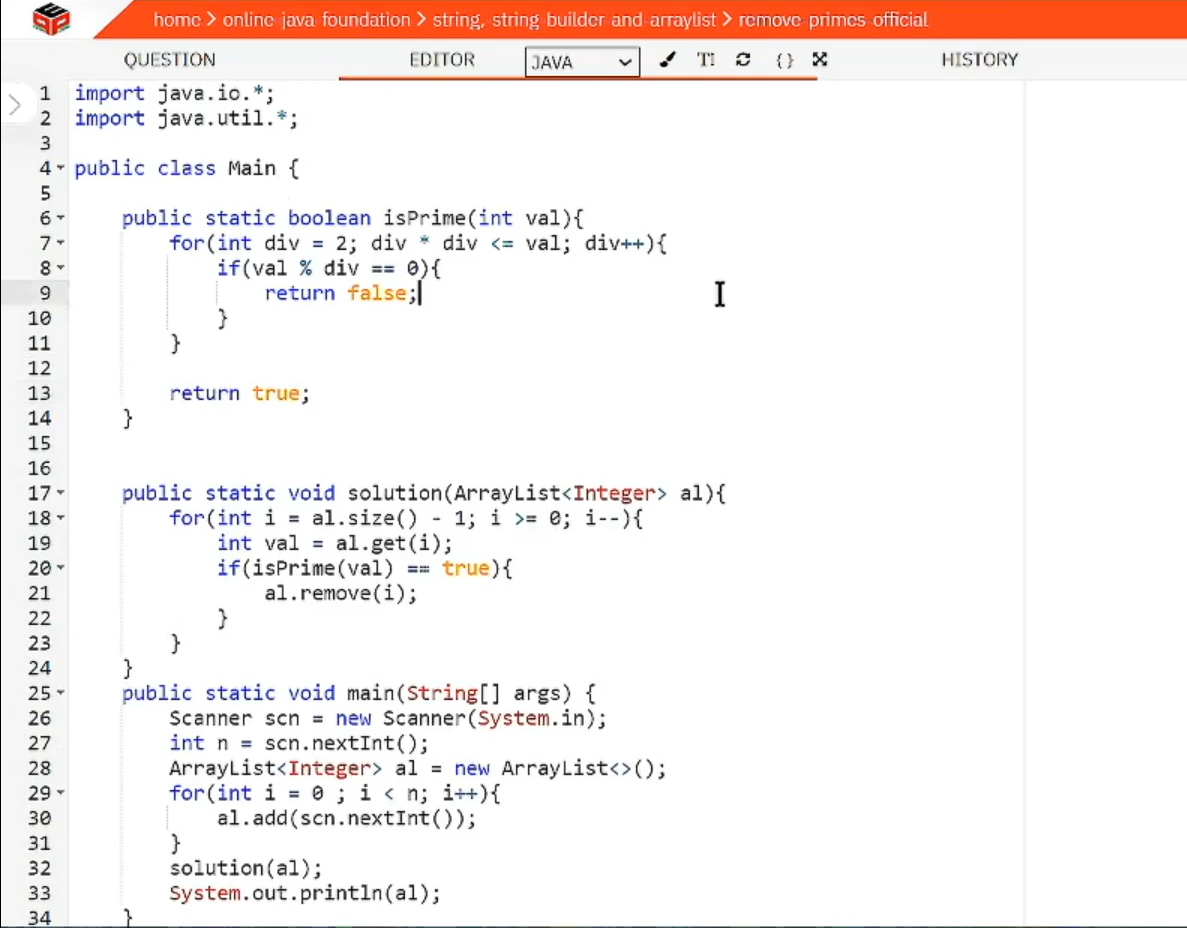
**Q: Remove Primes:**

Instead of iterating the loop from 0 - v.size() -1, running it from v.size() - 1 to 0 would be

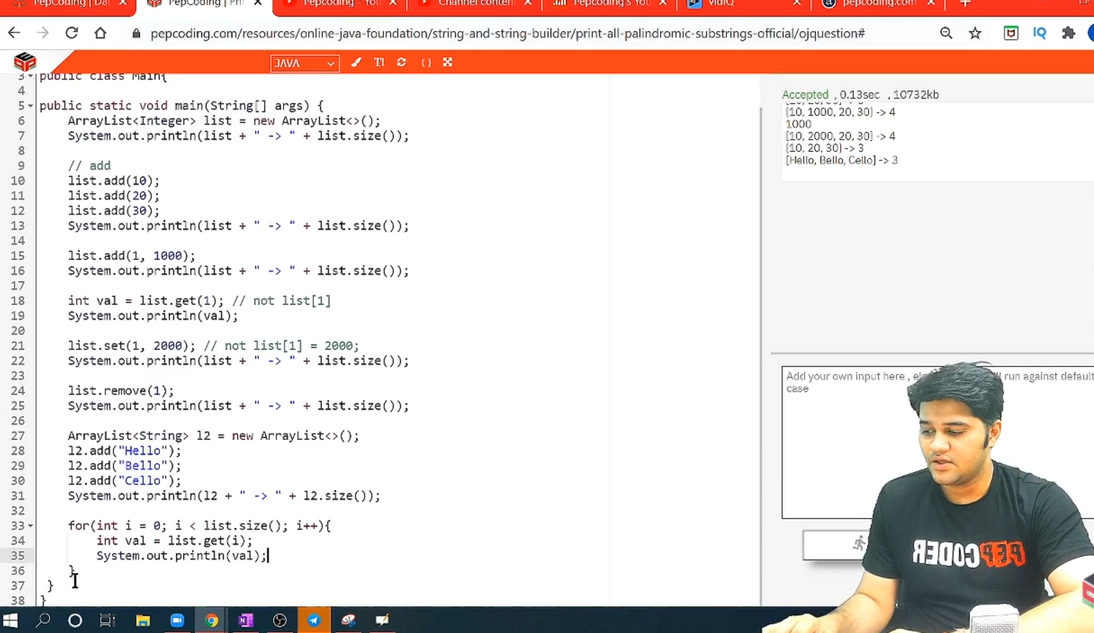
efficient. Because even if the position of elements get modified after deletion, it won't make us do

extra calculation

(Sumeet sir's code is much clean in this)



ArrayList in java (basic operations):



Loop in a vector or an arraylist:

For(int x: vector\_name)

{

Cout<<x<<”\n”;

}