Agenda : strings

↳ Memory Management

↳ Interning

↳ Implications of Interning

↳ Immutability

↳ Performance of Strings.

↳ String Builders

class starts at 9:05 PM

---

① Memory Management →

⇒ Non primitive data type.

⇒ Primitive data types are stored in Stack

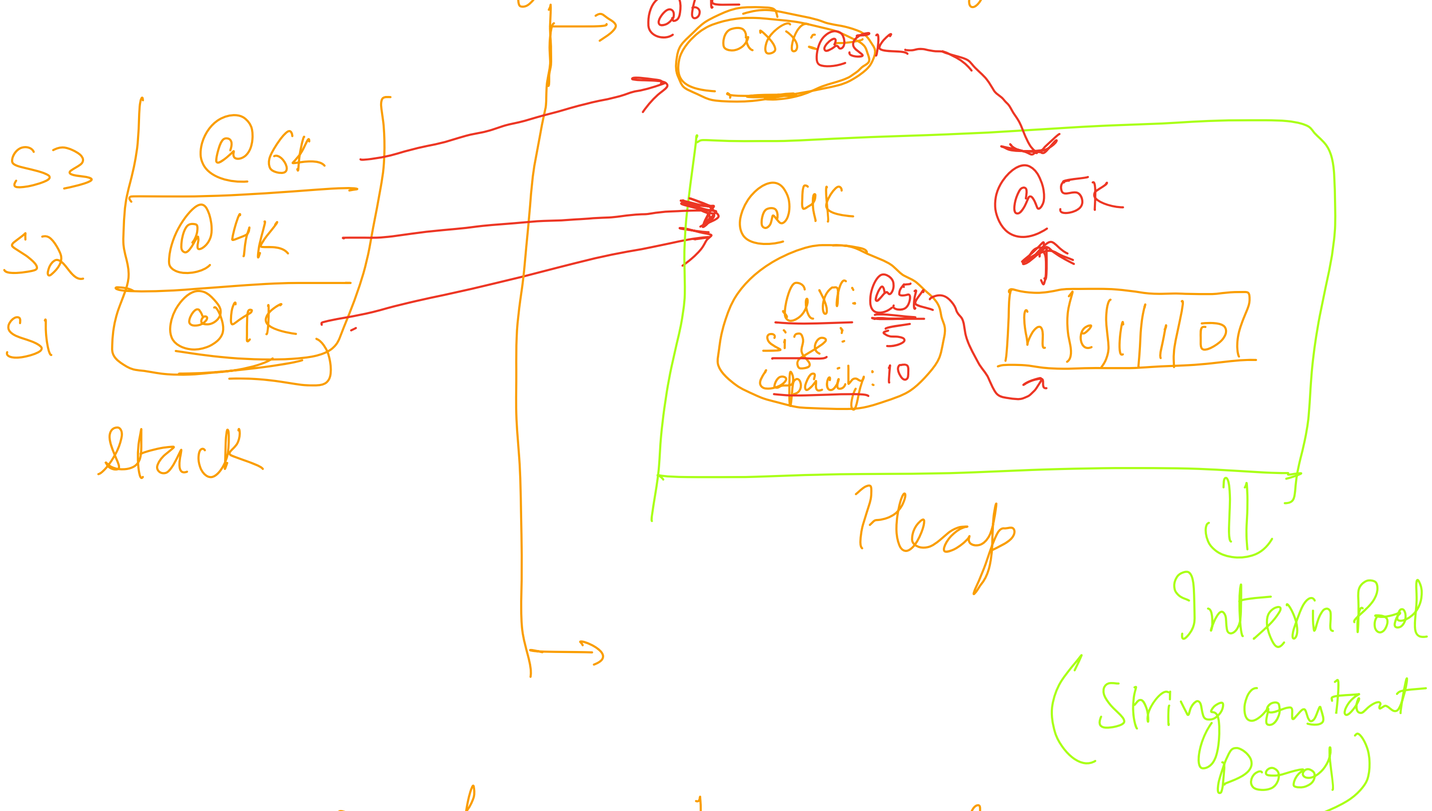
⇒ Non-primitive ⇒ actual data is stored in Heap & Address is stored in stack.

⇒ String is a non-primitive type

① String s1 = "Hello";

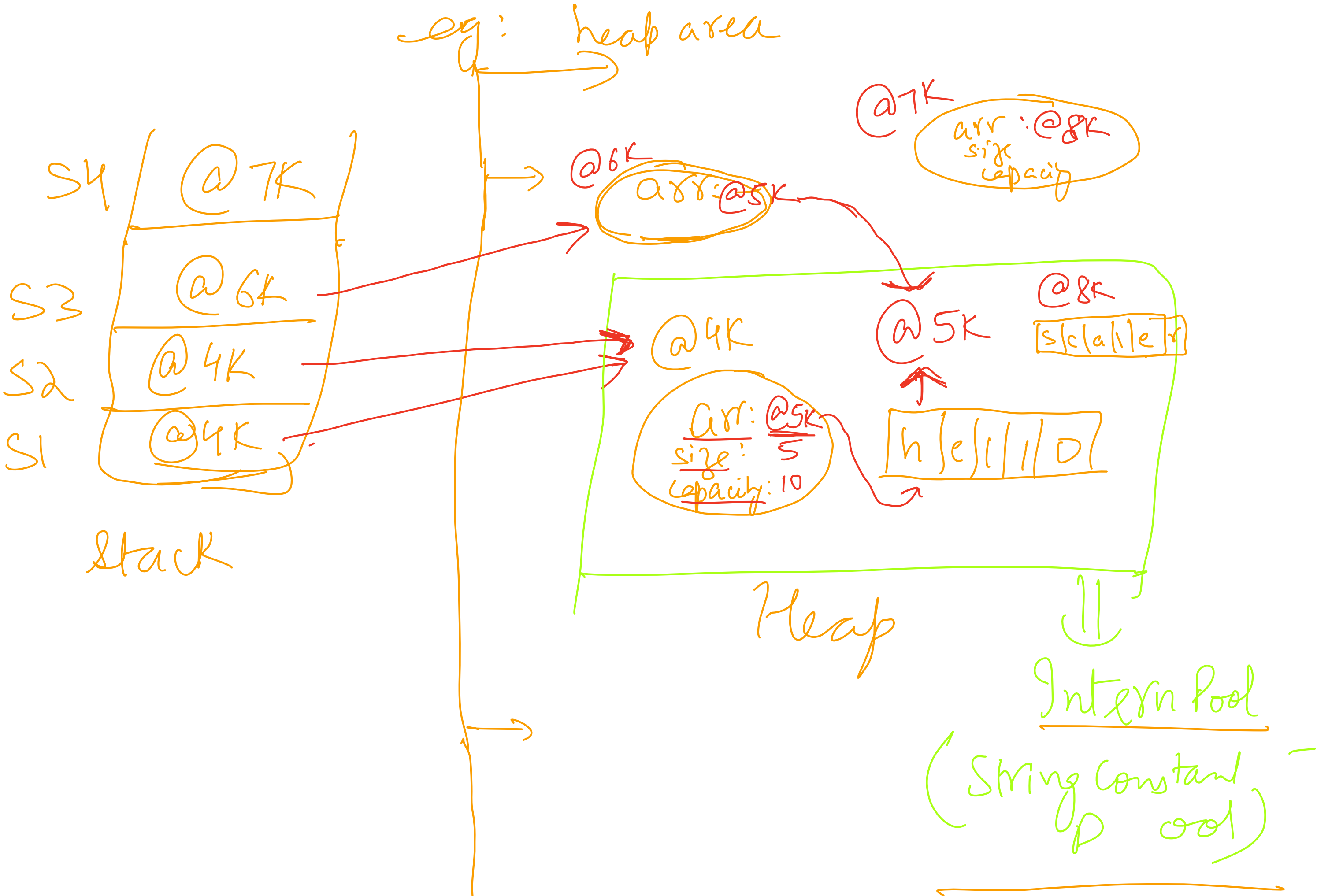
② String s2 = "Hello";

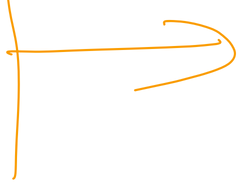
③ String s3 = $\boxed{\text{new}}$ String ("Hello");



new ⇒ forces to create a new object outside the Intern pool, but within the heap memory only.

eg:





⇒ This concept where all the strings containing the same content point to the same address is know as Interning.

⇒ Purpose of interning ⇒ to save memory.

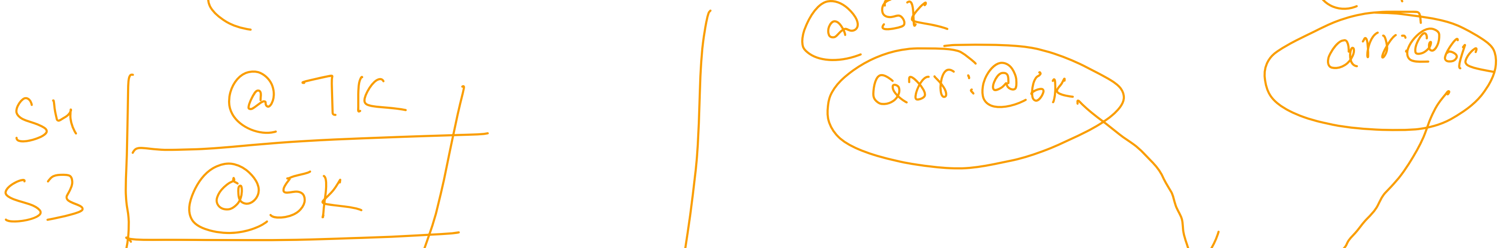
Implications of Interning

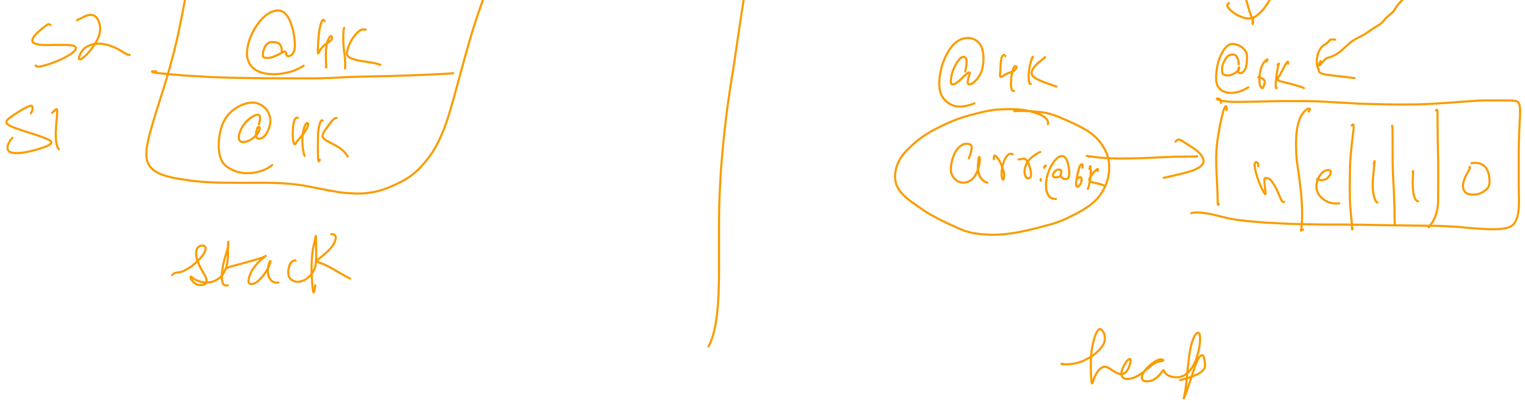
① Dont use == for doing a string Comparison.

```
✓ S1 = "Hello";
✓ S2 = "Hello";
✓ S3 = new String("Hello");
```

(S1 == S2) ⇒ true

(S2 == S3) ⇒ false





S4 = new String ("Hello");

(S3 = = S4) => false

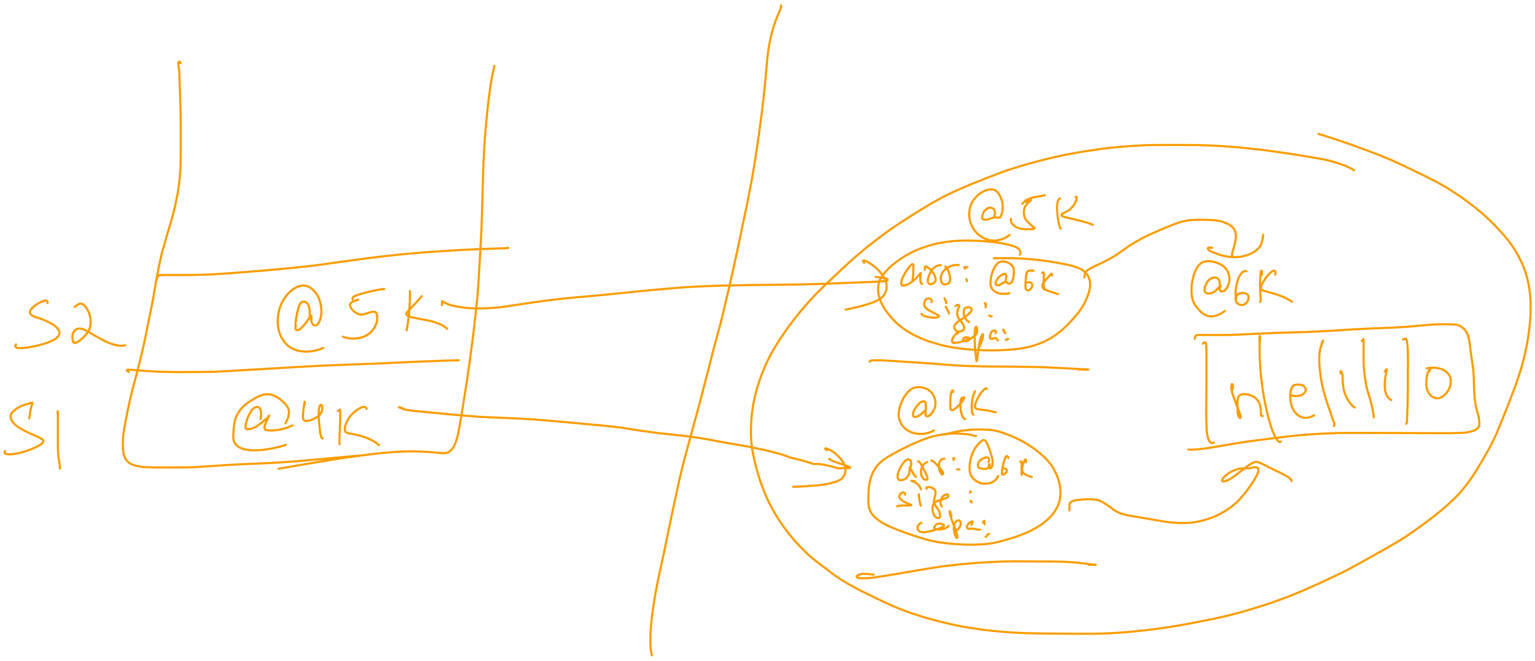
② equals => address comparison or content comparison.

-> actually does both of the above.

{ S1. equals (S2) => true

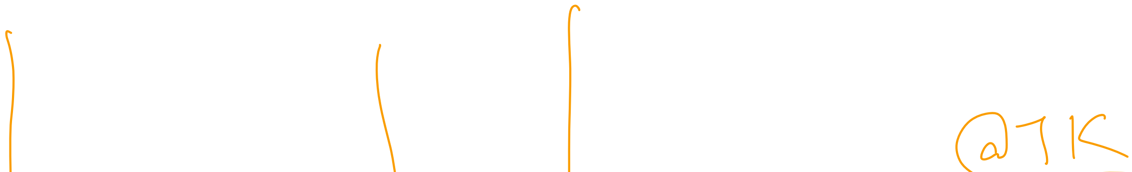
S2. equals (S3) => true

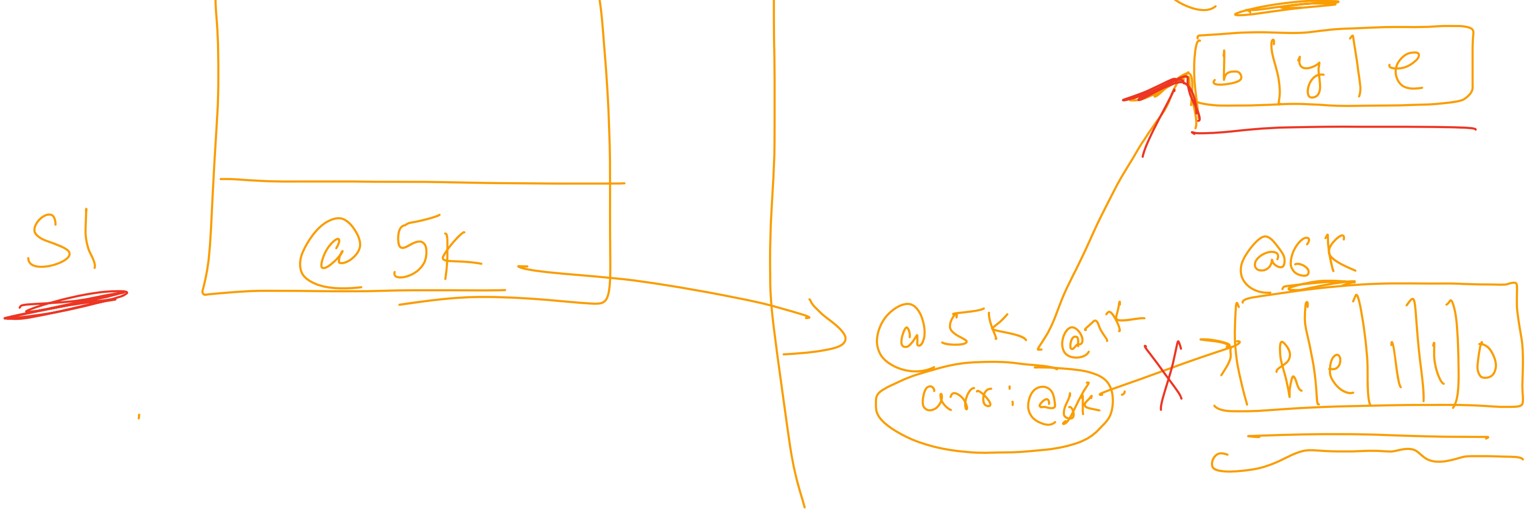
```
boolean equals ( String other) {

    if ( this == other)
        return true;

    int length1 = this. value. length();
    int length2 = Other. value. length();
    if ( length1 != length2) {
            return false;
        }
    // go for char by char comparison
}
```



⟹ **Immutability** →

```
String s1 = "hello";
sout(s1)  ⟹  hello
s1 = "bye";
sout(s1)  ⟹  bye.
```





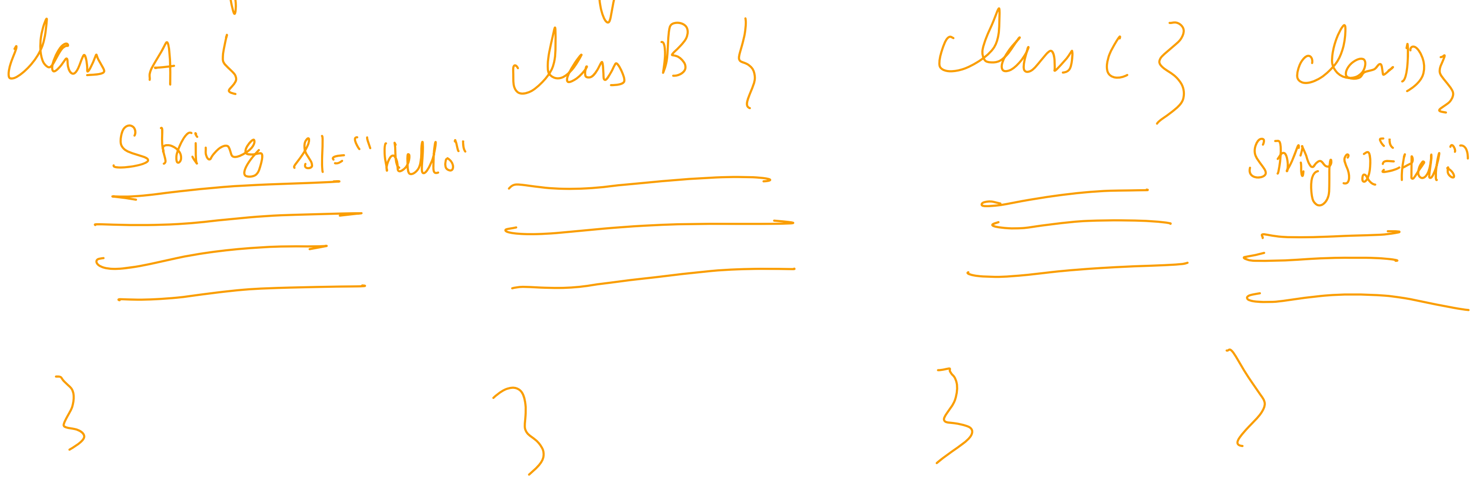
String instances | content are immutable whereas the string references are mutable.
& hello would be garbage collected.

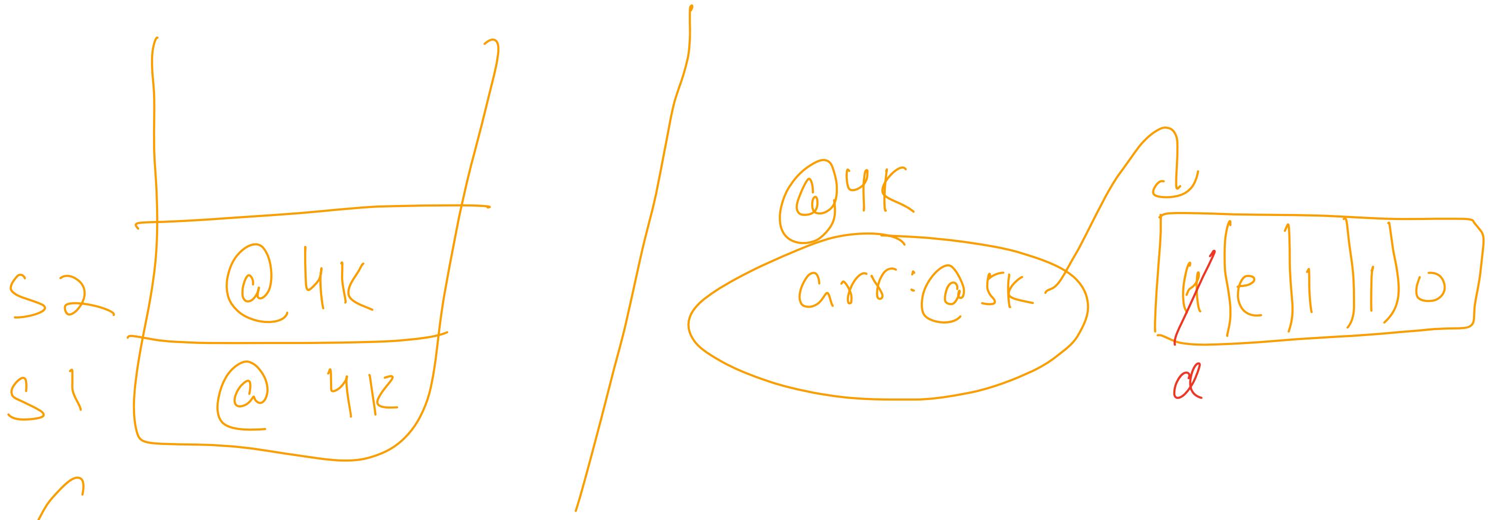
⇒ String classes do not provide us with any method that can alter the value of this character array.

```
⇒      String s1 = "hello";
        s1.replace('l', 'd');  // heddo
──→ sout(s1) ⇒ hello;
      s2 = s1.replace('l', 'd');
──→ sout(s2) ⇒ heddo.
```

Class resumes at 10:19

⇒ Why are Strings immutable, because of interning.





If s1 somehow alters the value of character array, then when s2 tries to access the value →, s2 will be met with a very bad surprise.

Save memory → Interning →

String immutability natural consequence of interning

⟹ Implications of Immutability. ⟹

① Operations on Strings in Java have very bad performance.

eg:-

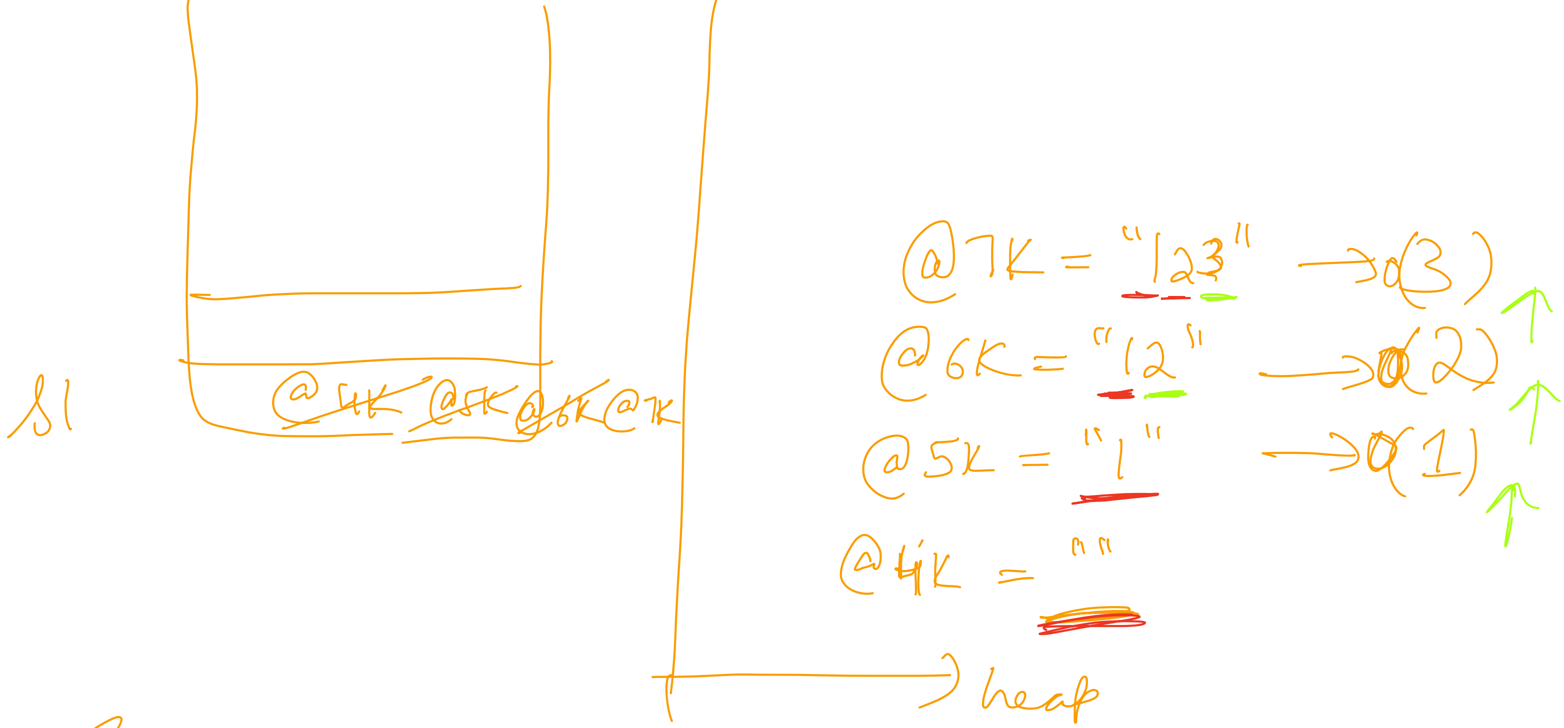
```
String s1 = "hello";
String s2 = s1.replace('l', 'd');
```

⟹ replace will copy all chars from s1 to s2 while creating a new String and that's why the time complexity would be $O(n)$

eg2:

```
String s = "";
for(int i = 1; i <= n; i++) {

    s += i;

}
```

Time complexity of above for loop is $O(n^2)$.



⇒ $1 + 2 + 3 + \dots\dots + n$

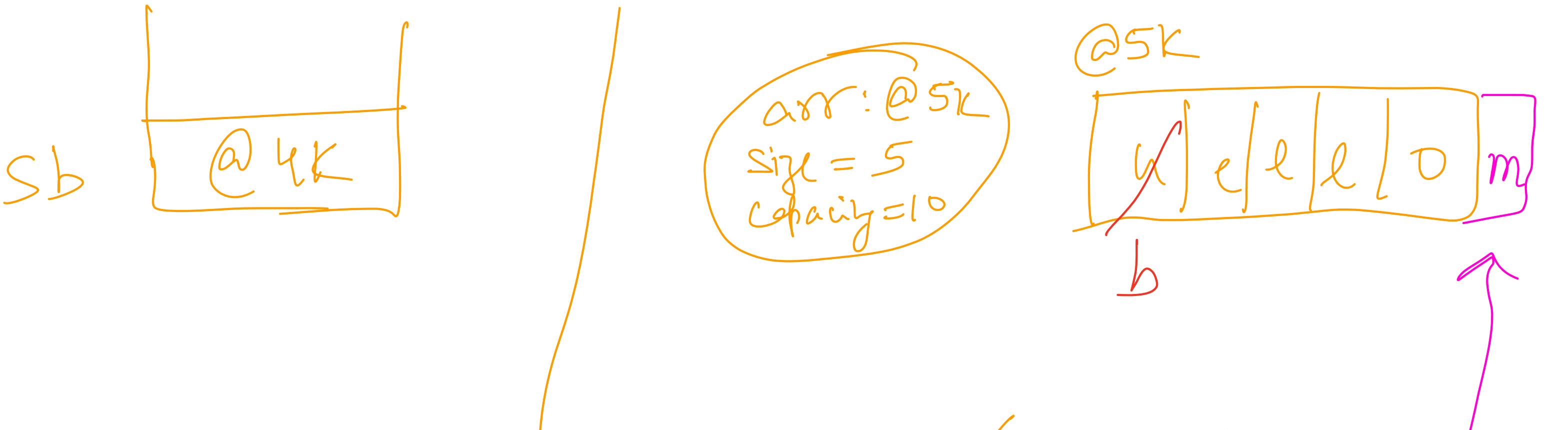
⇒ $\frac{n(n+1)}{2}$ ⇒ $O(n^2)$

StringBuilders →

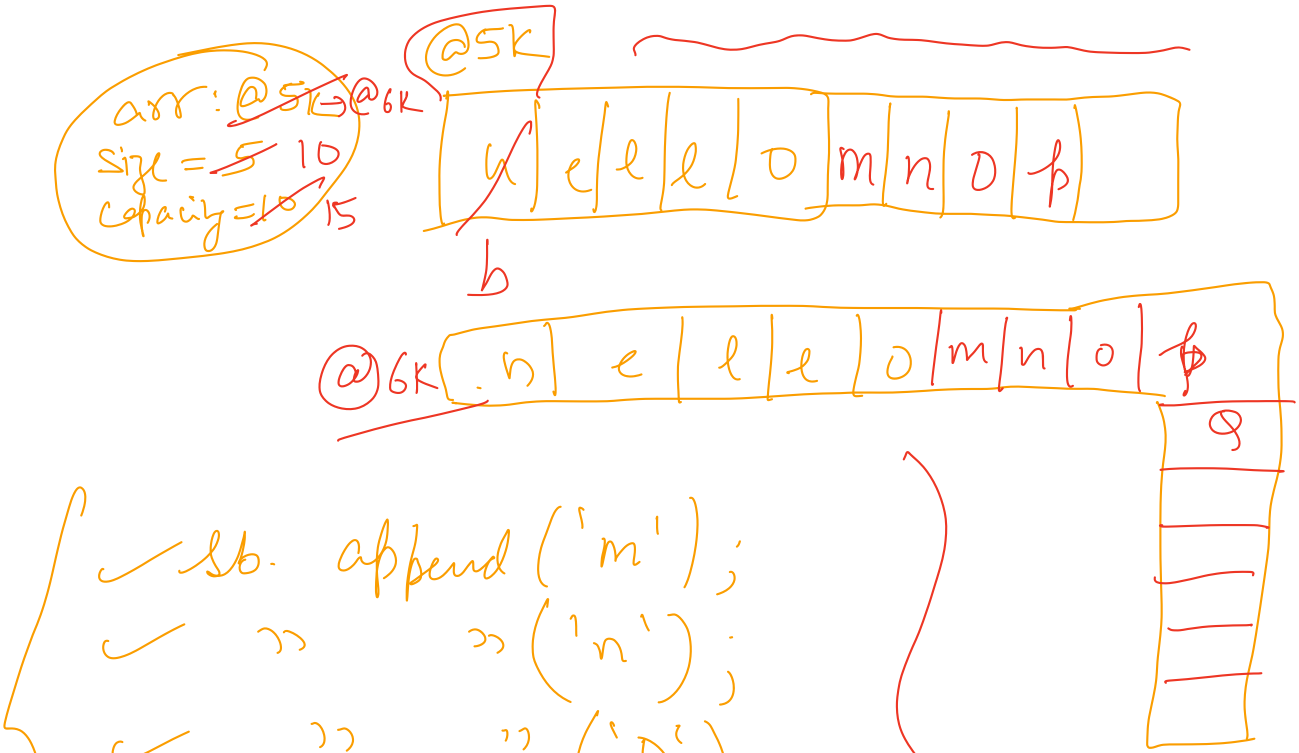
StringBuilder sb = new StringBuilder("hello");

⇒ StringBuilder methods can mutate the strings.

⇒ no concept of Intern pool.



```
sb.charAt(0,'b');
sb.append('m');
```



sb

```
①-④ {  sb.append('m');
        "       " ('n');
        "       " ('o');
        "       " ('p');
⑤    {  "       " ('q');
```

⇒ StringBuilder works on dynamic array.

⇒ Time complexities of ①-④ ⇒ O(1)

⇒ Time complexity of ⑤ ⇒ O(n)

⇒ So overall the time complexity is O(1) only and there's only 1 costly operation that happens at the time of array getting filled O(n).

eg:

```
StringBulder sb = new StringBuilder("");
for(int i = 1; i <= n; i++) {
    sb.append(i)
}
```

O(n)

⟹ String Builders are not synchronized whereas operations on String Buffer are synchronized in nature.

⟹ In a multi-threaded environment use String Buffer.

⟹ One positive implication ⟹ Strings are immutable, so they can be used in multi-threaded environment also.