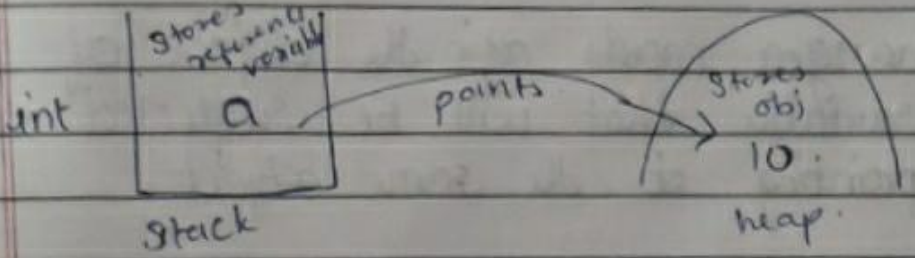
Strings & Stringbuilder ✍

① Stack memory :- When we declare a variable
eg :- int a = 10.
so, here the reference variable is stored in stack memory.

② Heap memory :- Reference variable stored in stack memory is pointing to the object of that variable are stored in heap memory.

int a = 10
s h



③ Memory allocations :- Memory allocations specifies the memory address to a program.

- There are two types of memory
  a) Stack memory & b) Heap memory

a) Static memory allocations :- It performs type checking at compile time.
- Here, errors will show at compile time.
- Declare datatype before you use it.
- More control & Runtime errors are reduced.
- Here you've to write a little bit more code. but you've more control over the type of data.

4d) Dynamic memory allocations:- Performs type checking at runtimes.

- Here, errors might not shown at till the program runs.
- No need to del declare a datatype of a variable
- It saves time in writing code but might give error at runtime.

5) Garbage collections:-

- More than one reference variable can points to the same object.
- If any changes made in the object of an reference variable that will be reflected to all others pointing to the same object.
- If there is an object without reference variable then the object will be disposed by "Garbage collection"
- So that's how garbage collection works.

Q What are strings? Strings will be in double quote " "

Ans:

- Strings are pile or a sequence of characters for Eg "kunal".
- It's a datatype & it is a non-primitive. so, a non-primitive datatype.
- Syntax :

```
String name = "Deepa Chaurasiya";
System.out.println(name);
```

↑ Everything that starts with a capital letter is a class.

- It is the most commonly used class in the JAVA's class library.
- Every string that you create it's actually an object of type String.

** Concepts :

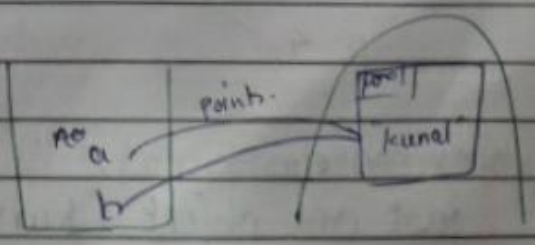
① String pool : It is a seperate memory structure inside the heap.

Q Why seperate pool? Why not just putting it out in the heap normally? like of every other object?

Ans: All the similar values of strings are not like recreated in the pool.

Eg:

String a = "kunal"
String b = "kunal"



so it's gonna be like it already exits in the pool no need to create it again pt b to the kunal.

Use case :- It makes our program more optimised.

Note :-

- If you try to change this object via this reference variable it will it not change for b.

Q Why ?

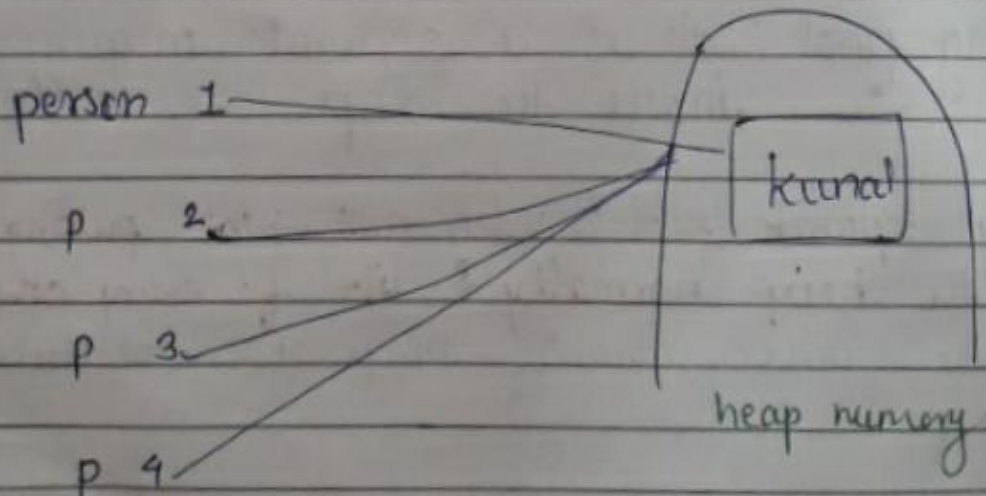
Ans :- Immutability.

- Strings are immutable in java, we can't change the object or modify object.
- If you wanna rename to "kunal" then you've to creat a new object for that.
- Strings are immutable for security purpose.

***

Why Strings are immutable ?

Ans :-



all person's named as kunal. ∴ All of them will be having just one object "kunal".

- Suppose one person decides to change their name if it was not immutable, if a person 1 decided to change their name to karan, so, that will change his name to karan
- If it was allowed to modify this then all person 1, 2, 3, 4 ~~will~~ name will be karan in the database.
- Therefore, for the security reasons strings are immutable.

* Comparisons of Strings.

① == method : comparator

Eg ① a ⟶ "kunal" } In this case == will give false.
b ⟶ "kunal"

② a ⟶ "kunal" } in this case == will be give true
b ⟶

- Comparator actully checks for value and reference variable if the reference variable is pointing to the same object

② How to creat different objects of same values.

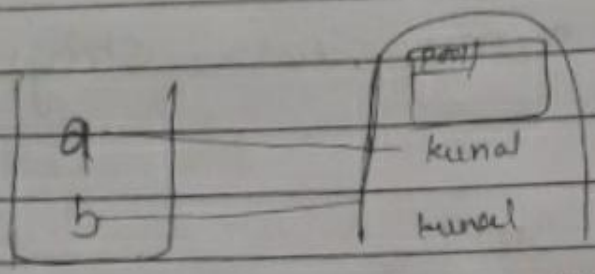
Ans:-

Syn Suppose, take an i/p as a string.
& by using new keyword we can create a new obj. →

String a = new String ("kunal")

String b = new String ("kunal")

// creating these values outside the pool but in heap because it is object so it will be in heap only.



a==b // false.

even though the values are same the but these two a & b are not pointing to the same object in that case it will give false.

- When you only need to check value, use .equals method or function.

so,

```
String name1 = new String ("kunal");
String name2 = new String ("kunal");

System.out.println(name1.equals(name2));
                                o/p :- true
```

Here does not care whether the reference variable are pointing to same object or not it just cares about the value.

* class is a name group of properties and funct^{n}.

* We can't do :

```
System.out.println(name1[0]);
```

- But in arrays we can do this.

- Though it's sort of collection of characters but we cannot do ↑ do this.

• So, we have to use a method called charAt().

```
System.out.println(name1.charAt(0));   o/p = K.
```

out : variable of type printstream.

printstream : class : it has method in it called println

* function overloading ! or method overloading.

Ans: Function overloading happens when two functions have same name.

eg:

```
fun () {
    // code
}
fun () {
    // code
}
```

} ✗ function overloading.

```
// fun (int a) {
    // code
}

fun (int a, int b) {
    // code
}
```

} It is allowed having different arguments with same function/ method name.

- At compile time, it decides which fish function to run.

* Pretty printing :-

```
float a = 453.1234f;
System.out.printf("Formatted number is %.2f", a);
```

Place holder

printf :- Formatted string

name declared

- Here % means a place holder & till how many decimal value do we want. for eg :- 2. % .2f (f because it's float)
- It rounds off all as well.

* for π

```
System.out.printf(Math.PI);
```

* for String

~~System.out.printf~~

System.out.printf("Hello my name is %s and I am %s.", "kunal", "student");

o/p :- Hello my name is kunal and I am student.

So, the order in which you have place the placeholders, int that order only you've to put the variable

- Some common formate specifiers :-

(1) %c :- Character
(2) %d :- Decimal number (base 10)
(3) %e :- Exponential floating-point number
(4) %f :- floating-point number
(5) %i :- Integer (base 10)
(6) %o :- Octal number (base 8)
(7) %s :- String
(8) %u :- Unsigned decimal (integer) number
(9) %x :- Hexadecimal number (base 16)
(10) %t :- Date/time
(11) %n :- Newline.

Operator overloading:

* Operater + ("Overloaaded for String type")

① sout ('a' + 'b'); o/p = 195

(character)

↓

Here the operater is converting this into its ~~character value~~ integer value. and then adding the "unicode or Ascii value"

② sout ("a" + "b"); o/p = ab

It concatenate the strings.

③ sout ('a' + 3); o/p = 100

④ Sout ((Char) ('a' + 3)); o/p = d

converting the 100 into a character (Explicit casting).

Note:- 1) When you're doing the addition with characters it converts into its value into a number then it use that number to solve the problem.

2) But with string it's not doing that, it actually taking the string value

⑤ Sout ("a" + 1); o/p = a1

// integer will be converted to integer that will call toString()

Note:- 1) When an integer is concatenated ("added") with a string it is converted to its wrapper class integer

2) This is same as: "a" + "1".

⑥ Sout ("kunal" + new Arraylist <>());

o/p :- kunal []

Initially arraylist is empty. then it's converted into kunal

So, we know it is will be like an object of type integer hence, it's calling the toString. which is returning a normal brackets. since it empty it will return a empty array.

⑦ Sout (new Integer (56) + new Arraylist <>());

error.

expression

Operator '+' cannot be applied to integer and array list.

* Operator '+' in Java is only defined for primitives and when any one of these values is a string

* Operator '+' you can only use with primitives and you can only use this with all the complex objects as well.

But, the only condition is at least one of these object should be of type String.

Eg: Sout (new Integer(56) + "" + new Arraylist <>());

this will work

complex obj

complex obj

obj of type string

Here the entire result will be of string type. o/p: 56[]

So, On string objects the plus operator is being overloaded, because it concatenate more than one strings.

- In java operater overloading is not supported for some software engineering but consideration But, in C++ it is supported.
- So, ~~you basically~~ + operater, you can basically modify what the plus '+' operater is doing in C++ & also in python
- you can also make it act like as a multiplication or subtring subtraction, you can add complex data type at as well.
- But this results in poor code, that is why in Java it is not supported.
- Java has only given us operator overloading exception for strings, but you cannot do it on your own if you want to merge two complex objects of your own type like array, hashmap it will not allow ever for the modification.
- It's only operater that is intentionally overloaded in java to support string concatenation or string joining.

- new Integer() : In future it's going to be removed.

(8) sout ("a" + 'a'); o/p : aa if one one of the data type is string ans will be String

String performance

loop

① Series : " " . . .

② Series : "" + "a" : "a"

③ series : "a" + 'b' : "ab"

④ series : "ab" + 'c' : "abc"

⑤. Here the new object is being created everytime, it is not changing the original object because the Strings are immutable. Therefore this one one copying the one old one and the appending it one with the new one.

- So much of waste of memory because of these and it we can't be have having any reference val variable. So time complexity : $O(n^2)$ worst.

- Solution :- wouldn't it be great if there was sort of datatype that will allows us to modify it value because strings fails to allow this. So, the answer is :- Stringbuilder

* The Here in Stringbuilder one object is made and the changes are done in that object only & the reference is also the same and it will not changed

* Stringbuilder is a seperate class.

* Methods that string provide (Some).

1) .toCharArray() :- It converts char into array of strings characters

Eg:- string name = "kunal kushwaha";
sout (Arrays.toString(name.toCharArray()));
o/p : [k, u, n, a, l, , , K, u, s, h, w, a, h, a]

PS Even that space will be counted.

2) .length() :- It will gi provide you the length.

3) .toLowerCase() :- It will convor into lowercase.
It's not actully going to convert the origin object because of immutability.

4) .indexOf :- It will give the index value.

5) .lastIndexOf :- It will give the last index value.

6) .strip() :- White spaces are removed.

7) .split() :- add a regex first :- After adding regex it will split ~~on it~~ over here

.Split(regex :)
↑
blueed

Code of plr palindrom string. Time On

```
public class Palindromstring{
    Public static void main(String[] args){
        String str = "abcba";
        System.out.println(isPalin(str));
    }

    Static boolean isPalin(String str){
        if (str == null || str.length() == 0){
            return true;
        }
        str = str.toLowerCase();
        for(int i=0; i<=str.length()/2; i++){
            char start = str.charAt(i);
            char end = str.charAt(str.length()
                       -1-i);
            if(strat != end)
                return flase;
            }
        }
        return ftrue;
    }
}
```

this will be having null.

If this is = 0, that mean for empty str

here length is a method

you can also put null as false completely depends upon interviewer but the ans will be same

(Str.length-1) = last index every time we will be doing -i b, Suppose i=2, then end index should be, end index -2 Hence ...

Strings and StringBuilder in Java

* **What is String?**

String is basically a collection/sequence of characters. and it is stored in String data type.

Example

```
String name = "Kunal Kushwaha"
```

- String → datatype
- name → reference variable
- = → String declaration
- "Kunal Kushwaha" → value (collection of character) / Object

⇒ String is the most commonly used class in the Java's class library.

```
String name = "Kunal Kushwaha"
```

Everything that start with capital letter is a class.

⇒ Every String that we create, it's actually an object of type String.

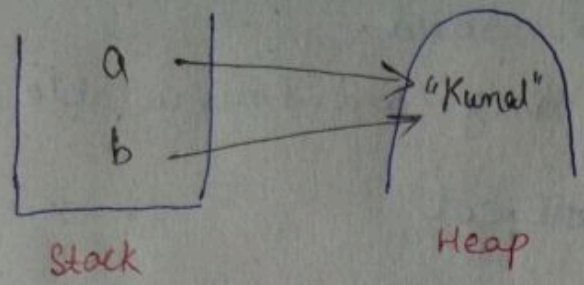
* **Internal Working of String :—**

Let say,

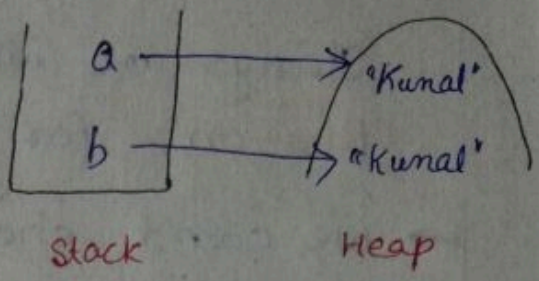
```
String a = "Kunal"
String b = "Kunal"
```

Q. Is this creating two different objects or is it pointing to same object?

How it is stored (A) or (B) ?



OR

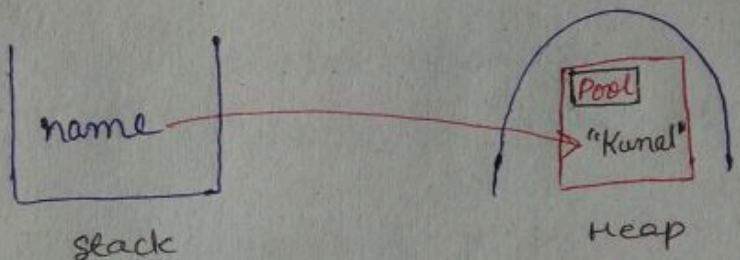


(A) (B)

Regarding this let's understand some concepts :-

1. String Pool :- It is a separate memory structure inside the heap.

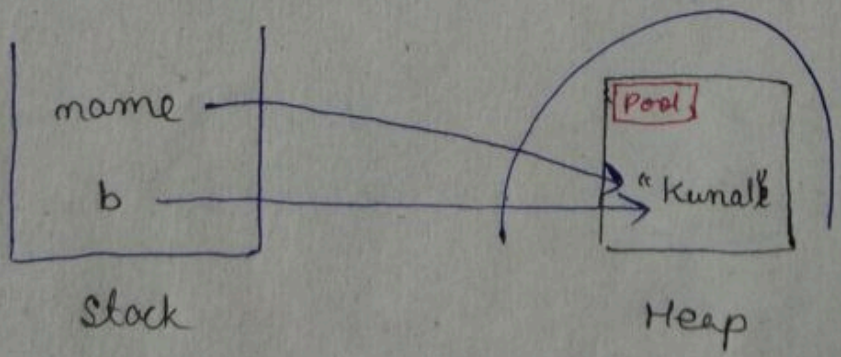
Ex- String name = "Kunal"



- Use of Pool :-

→ All the similar values of strings are not recreated in the pool. That makes our programs more optimized.

Ex→ String name = "Kunal" ; String b = "Kunal"



Here, it says that "Kunal" already exists in the pool. So, no need to create it again. Hence, point b to 'Kunal'.

2. Immutability :-

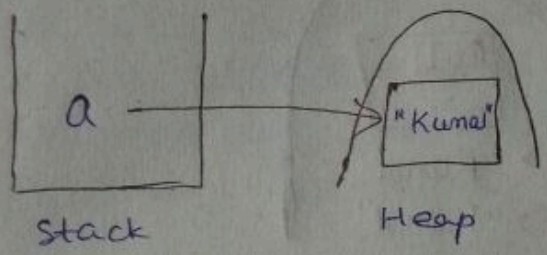
→ Strings are immutable in Java.

Reason:- For Security

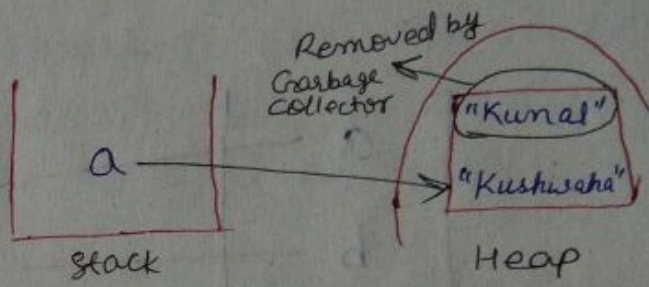
→ we can't change any object.

* Let's say :-

Initially; String a = "Kunal"



Then, a = "Kushwaha"



Here, we haven't change the object i.e "Kunal".
we have created a new object i.e, "Kushwaha"

* String Comparison Methods :-

(1) == method :-

== ⇒ a comparator

It checks ~~the~~ if the reference variables are pointing to same object

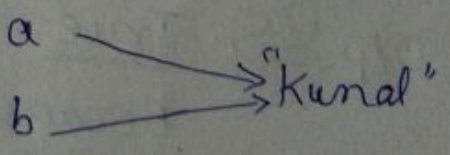
case-A

a ⟶ "Kunal"

b ⟶ "Kunal"

⇒ a==b will give False

case-B



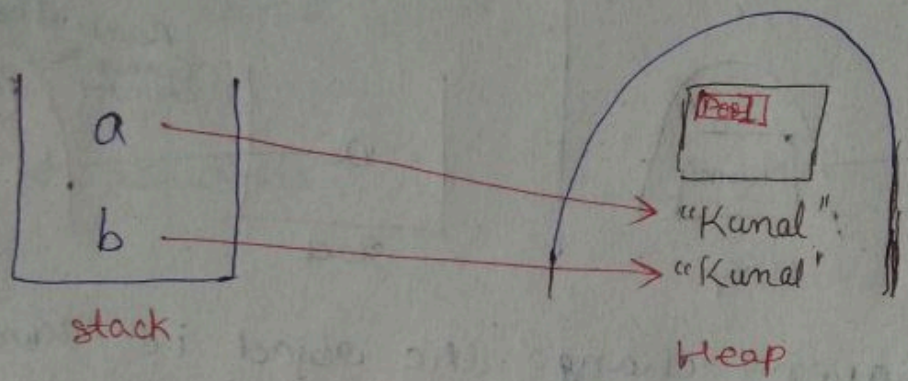
⇒ a==b will give True

* How to create different objects of same value :-

⇒ For this, we use "new" keyword.

```
String a = new String ("Kunal")
String b = new String ("kunal")
```

// creating these values outside the pool but in heap.



NOTE :- This time a==b will give False.

② .equals method :- When we only need to check value, we use .equals method.

Ex

```
String a = new String ("Kunal");
String b = new String ("Kunal");
System.out.println (a.equals(b));
```

O/P ⇒ True

Because, it just checks the values are same or not.

* PrintStream — a class in java.
* out — a variable of type PrintStream.
* println — a method of PrintStream class.

NOTE:
Internal working of println → println calling the valueOf ~~function~~ method and that is calling toString() and then it's returning the string.

* Pretty Printing :— It prints/present the source code in an attractive way, so that it can be easily analyzed by the interpreter as well as easily read by humans.

Ex→ Print the value of π till 3 digit after decimal.

```
System.out.printf("Pie: %.3f", Math.PI);
```

printf → Print formatted string
%.3f → placeholder

* `System.out.println('a' + 'b');`
  O/P ⇒ 195 [ASCII value of a = 97, ASCII value of b = 98]

* `System.out.println("a" + "b");` // String concatenation
  O/P ⇒ ab

* `System.out.println('a' + 3);`
  O/P ⇒ 100 [ASCII value of a = 97]

* `System.out.println((char)('a' + 3));`
  O/P ⇒ d

* System.out.println("a" + 1);
O/P ⇒ a1

// String "a" is not converting into its ASCII value .....

NOTE: when an integer is added with a string it is converted to its rapper class integer. i.e, it is going to use toString().

** String Performance ** V.V.I

Ex

```
public static void main(String[] args) {
        String series = "";
        for(int i=0; i<26; i++) {
            char ch = (char)('a'+ i);
            series = series +ch;
        }
        System.out.println(series);
}
```

O/P ⇒ abcdefghijklmnopqrstuvwxyz

Let's see the working of above code, And what is the problem? why it is not a very good solution?

⇒ Initially, series = "" // empty string

⇒ After 1st iteration ⇒ series = "" + 'a' = "a"

⇒ After 2nd iteration ⇒ series = "a" + 'b' = "ab"

⇒ After 3rd iter. ⇒ series = "ab" + 'c' = "abc"

** Explaination **

⇒ So, we noticed that, new object is created everytime It is not changing the original object as we know that Strings are immutable.
So, it's actually creating new string object and copying the old one and then appending the new changes..

⇒ That's why, there is so much wastage of memory becoz, all the objects are dereferenced.
It happens like ↓

a, ab, abc, abcd, abcde, abcdef, ----------
------ abcdefghijklmnopqrstuvwxy

All these above large strings will have no reference variable. i.e, wastage of memory.

⇒ These are of size ↓

$1 + 2 + 3 + 4 + 5 + 6 + \dots\dots\dots + N$

$= \frac{N(N+1)}{2} = O\left(\frac{N^2+N}{2}\right) = O(N^2)$

* Solution →

StringBuilder :- It is a class just like String.
⇒ A datatype that allow us to modify the value.
⇒ It will not create a new object like string. but actually add in the original one.
i.e, StringBuilder is mutable.

```
public static void main(String[] args) {
        StringBuilder builder = new StringBuilder();
        for (int i=0; i<26; i++) {
            char ch = (char) ('a' + i);
            builder.append(ch);
        }
        System.out.println(builder.toString());
}
```

NOTE :- It gives O(N) complexity.

* String Methods :-

* toCharArray() :→ It converts the String into character array.
* length() :→ gives the length of String
* getBytes()
* toLowerCase() :→ prints the String into lowercase.

There are many more such methods .....