# Constructors of String class:

- ```
  String s = new String();
  ```

  This code creates an empty `String` object.

- ```
  String s = new String(String literal);
  ```

  This code creates a `String` object in the heap for the given string literal.

- ```
  String s = new String(StringBuffer sb);
  ```

  This code creates an equivalent `String` object for the given `StringBuffer`.

- ```
  String s = new String(char[] ch);

  // Example
  char[] ch = {'a', 'b', 'c', 'd'};
  String s = new String(ch);
  System.out.println(s); => abcd
  ```

  This code creates an equivalent `String` object for the given `char[]`.

- ```
  String s = new String(byte[] b);

  // Example
  byte[] ch = {100, 101, 102, 103};
  String s = new String(b);
  System.out.println(s); => defg
  ```

  This code creates an equivalent `String` object for the given `byte[]`.

# Important methods of String class:

`public char charAt(int index);` returns the character located at a specified *index*

***Example:***
  ```

```
    String s = "Ahmed";
    System.out.println(s.chatAt(3));     => e
    System.out.println(s.chatAt(30));    => runtime exception:
  StringIndexOutOfBoundsException
```

`public String concat(String s);` The overloaded `+` and `+=` operators also meant for concatenation purpose.

***Example:***

```
    String s = "Ahmed";
    s = s.concat(" Elhilali");
    // s = s + " Elhilali";
    // s -= " Elhilali";
    System.out.println(s);    => Ahmed Elhilali
```

`public boolean equals(Object o);` The perform content comparison where ***case** is important. This is overriding versopn of `Object` class `equals()` method.
`public boolean equalsIgnoreCase(String s);` The perform content comparison where ***case*** is not important.

**Note:**

- In general we can use `equalsIgnoreCase()` method to validate usernames, where *case* is not important, whereas we can use `equals()` to validate passwords where *case* is important.

***Example:***

```
    String s = "Ahmed";
    System.out.println(s.equals(AHMED));            => false
    System.out.println(s.equalsIgnoreCase(AHMED));  => true
```

`public String substring(int begin);` Returns a substring from begin index to end of the `String` ***[begin, last index]***.
`public String substring(int begin, int end);` Returns a substring from begin index to end - 1 index ***[begin, end[***.

***Example:***

```
    String s = "abcdefg";
    System.out.println(s.substring(3));     => defg
    System.out.println(s.substring(2, 6));  => cdef
```

`public String length();` Returns the number of character present in a string.

**Note:**

- `length` variable is applicable for *arrays* but not for `String` objects. Whereas `length()` method is applicable for `String` objects but not for *arrays*

***Example:***

```
  String s = "Ahmed";
  System.out.println(s.length);    => compile-time error: cannot find
sysmbol | symbol: variable length | location: java.lang.String
  System.out.println(s.length());  => 5
```

`public String replace(char oldCh, char newCh);` Replace the old character with a new character.

***Example:***

```
  String s = "ababa";
  System.out.println(s.replace('a', 'b')); => bbbbb
```

`public String toLowerCase();` Converts the string to upper case.
`public String toUpperCase();` Converts the string to lower case.
`public String trim();` Removes the whitespace in the beginning and at the end of the string.\

`public int indexOf(char ch);` Returns index of first occurence of specified character. `public int lastIndexOf(char ch);` Returns index of last occurence of specified character.

***Example:***

```
String s = "ababa";
System.out.println(s.indexOf('a'));      => 0
System.out.println(s.lastIndexOf('a'));  => 4
```

**Note:**

- Because of runtime operation if there is a change in the content, then with those changes a new object will be created in the *heap*. If there is not change in the content, the existing object will be reused and a new object won't be created. Wether the object is present in the *heap* or *SCP* the rule is the same.

***Example:***

```
String s1 = new String("Ahmed");
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();

System.out.println(s1 == S2);   => false
System.out.println(s1 == S3);   => true

String s4 = s2.toLowerCase();
String s5 = s4.toUpperCase();
```

| Heap | SCP |
|---|---|
| s1, s3 -> Ahmed | Ahmed |
| s2 -> AHMED | |
| s4 -> ahmed | |
| s5 -> AHMED | |

***Example:***

```
String s1 = "Ahmed";
String s2 = s1.toString();
String s3 = s1.toLowerCase();
String s4 = s1.toUpperCase();
String s4 = s1.toUpperCase();
String s5 = s4.toLowerCase();
```

| Heap | SCP |
|---|---|
| s4 -> AHMED | s1, s2, s3 -> Ahmed |
| s5 -> ahmed | |

# How to create our own immutable class:

- Once we create an object we can't perfom any change in the object. If we are trying to perfom any change and if there's a change in the content then with those change a new object will be created. If there's no change in the content then existing object will be reused. This behaviour is nothing but ***immutability***

***Example:***

```
String s1 = new String("Ahmed");
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
```

| Heap |
| --- |
| s1, s3 -> Ahmed |
| s2 -> AHMED |

- We can create our own *immutable* class like this:

```
public final class Test {
    private in i;
    Test (int i) {
        this.i = i;
    }

    public Test modify(int i) {
        if (this.i == i) {
            return this;
        }

        else {
            return new Test(i);
        }
    }

    public static void main(String[] args) {

        Test t1 = new Test(10);
        Test t2 = t1.modify(100);
        Test t3 = t1.modify(10);

        System.out.println(t1 == t2); => false
        System.out.println(t1 == t3); => true
    }
}
```

| Heap |
| --- |
| t1, t3 -> i = 10 |
| t2 -> i = 100 |

- Once we create a Test object we can't perform any changes in the existing object. IF we are trying to perfom any change and if there is change in the content then with those change a new object will be created, and if there is not change in the content then the existing object will be reused.

# final vs immutability:

- Final applicable for variable but not objects, whereas immutability applicable for objects but not for variable.
- By declaring a *reference variable* as final we won't get any immutability nature, even though reference variable is final we can perfom any type of change in the corresponding object but we can't perform *reassignment* for that variable.
- Hence final and immutability are different concepts.

```
    class Test {
        public static void main(String[] args) {
            final StringBuffer sb = new StringBuffer("Ahmed");
            sb.apped(" Elhilali");

            System.out.println(sb);             => Ahmed Elhilali
            sb = new StringBuffer("Keller");    => compile-time error:
cannot assign a value to final variable sb.
        }
    }
```

- Which of the following are meaningful?

```
    final variable;         => valid
    immutable variable;     => invalid
    final object;           => invalid
    immutable object;       => valid
```