# The difference between String and StringBuffer:

- If the content is fixed and won't change frequently then it is recommended to go for `String`.
- If the content is not fixed and keep changes frequently then it is not recommended to use `String`, because for every change a new object will be created which effects the perfermance of the system. To handle this requirements we should go for `StringBuffer`.
- The main advantage of `StringBuffer` over `String` is all required changes will be performed on the existing object only.

# StringBuffer constructors:

`StingBuffer sb = new StringBuffer();` creates an empty `StringBuffer` object default initial `capacity` equals 16, once `StringBuffer` reaches its maximum *capacity* a new `StingBuffer` object will be created with a `newCapacity = (current capacity + 1) * 2;`

```
    StringBuffer sb = new StringBuffer();
    System.out.println(sb.capacity());      => 16

    sb.append("abcdefghijklmnop");
    System.out.println(sb.capacity());      => 16

    sb.append("q");
    System.out.println(sb.capacity());      => 34
```

`StringBuffer sb = new StringBuffer(int initialCapacity);` creates an empty `StrinBuffer` object with specified *initial capacity*.
`StringBuffer sb = new StringBuffer(String s);` creates an equivalent `StringBuffer` for the given `String` with `capacity = s.length() + 16`

***Example:***

```
    StringBuffer sb = new StringBuffer("Ahmed");
    System.out.println("sb.capacity())   => 21
```

# Important methods of StringBuffer:

```
public int length();
public int capacity();
public char charAt(int index);
```

***Example:***

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

`public void setCharAt(int index, char ch);` to replace the character located at a specified index with provided character.

```
public StringBuffr append(String s);
public StringBuffr append(int i);
public StringBuffr append(long l);
public StringBuffr append(char ch);
public StringBuffr append(boolean b);
```

... All these methods are *overloaded* methods.

***Example:***

```
    StringBuffer sb = new StringBuffer();
    sb.append("PI value is: ");
    sb.appedn(3.14);
    sb.append(" It is exactly: ");
    sb.append(true);
    System.out.println(sb);     => PI value is: 3.14 It is exactly true
```

```
public StringBuffr insert(int index, String s);
public StringBuffr insert(int index, int i);
public StringBuffr insert(int index, long l);
public StringBuffr insert(int index, char ch);
public StringBuffr insert(int index, boolean b);
```

... All these methods are *overloaded* methods.

***Example:***

```
    StringBuffer sb = new StringBuffer("abcdefgh");
    sb.insert(2, "xyz");
    System.out.println(sb);     => abxyzcdefgh
```

`public StringBuffer delete(int begin, int end);` to delete characters located from begin index to end - 1 index, ***[begin, end[***.

`public StringBuffer deleteCharAt(int index);` to delete the character located at specified index.

`public StringBuffer reverse();`

```
StringBuffer sb = new StringBuffer("Ahmed");
System.out.println(sb.reverse());  => demha
```

`public void setLenth(int length);`

```
StringBuffer sb = new StringBuffer("Ahmed Elhilali");
sb.setLength(8);
System.out.println(sb);  => Ahmed El
```

`public void ensureCapacity(int capacity)`; to increase capacity on fly based on our requirement

```
StringBuffer sb = new StringBuffer();
System.out.println(sb.capacity());      => 16

sb.ensureCapacity(1000);
System.out.println(sb.capacity()):      => 1000
```

`public void trimToSize()`; to deallocate extra allocated free memory

```
StringBuffer sb = new StringBuffer(1000);
sb.append("abc");
sb.trimToSize();
System.out.println(sb.capacity());      => 3
```

# StringBuffer vs StringBuilder:

- Every method that is present in `StringBuffer` is `synchronized`, hence only one `thread` is allowed to operate on `StringBuffer` object at a time, which may create performance problems.
- To handle this requirement some people introduced `StringBuilder` concept in 1.5v.
- `StringBuilder` is exactly the same as StringBuffer except the following differences:
- Every method present in `StringBuffer` is `synchronized`.
- Every method present in `StringBuilder` is `non-synchronized`.
- At a time only one `thread` is allowed to operate on a `StringBuffer` object. Hence `StringBuffer` is `thread-safe`.

- At a time multiple `threads` are allowed to operate on a `StringBuilder` object. Hence `StringBuilder` is `not thread-safe`.
- `threads` are required to wait to operate on `StringBuffer` object, hence relatively performance is low.
- `threads` are not required to wait to operate on `StringBuilder` object, hence relatively performance is high.
- `StringBuffer` was introduced in 1.0v.
- `StringBuilder` was introduced in 1.5v.

| StringBuffer | StringBuilder |
|---|---|
| 1. synchronized | 1. non-synchronized |
| 2. thread-safe | 2. thread-unsafe |
| 3. low performance | 3. high performance |
| 4. 1.0v | 4. 1.5v |

**Note:**

- Except the above difference everything is the same in `StringBuffer` and `StringBuilder`, including methods and constructors.

# String vs StringBuffer vs StringBuilder:

1. If content is fixed and won't change frequently then we should go for `String`.
2. If the content is not fixed and changes frequently but *thread safety* is **required** then we should go for `StringBuffer`.
3. If the content is not fixed and changes frequently but *thread safety* is **not required** then we should go for `StringBuilder`.

# Method chaining:

- For most of the methods in `StringBuffer` and `StringBuilder` return type same type, hence after applying a method we can call another method on the result, which forms *method chaining*.
- `sb.m1().m2().m3().m4()...` in method chaining method calls will be executed from *left* to *right*.

```
   StringBuffer sb = new StringBuffer();
   sb.append("Ahmed").append(" Elhilali").append(" Keller").insert(2,
"abc").reverse().delete(2, 10);
   System.out.println(sb);       => relihlE demcbahA
```