String, StringBuffer, StringBuilder

- **Mutability Vs Immutability**

String objects are immutable(non-changeable). Once a String object is created, we can't perform any changes in that object. If we try to perform any changes then a new object will get created and reference will point to that newly created object.

Example – String s=new String("xyz");

```
s.concat("abc");
System.out.print(s);
```

Output : xyz

StringBuffer objects are mutable(changeable)

Example – StringBuffer sb=new StringBuffer ("xyz");

```
sb.append("abc");
System.out.print(sb);
```

Output : xyzabc

- **== operator Vs equals() method**

= = : reference comparison (true, if pointing to same reference else false)

equals() : content comparison

Example –

```
String s1=new String("xyz");
String s2=new String("xyz");
System.out.print(s1==s2);              Output : false
System.out.print(s1.equals(s2));       Output : true
```

```
StringBuffer sb1=new StringBuffer ("xyz");
StringBuffer sb2=new StringBuffer ("xyz");
System.out.print(sb1==sb2);            Output : false
System.out.print(sb1.equals(sb2));     Output : false
```

Note : Object class equals() is meant for reference/address comparison.
But, in String class it is overridden for content comparison only.
StringBuffer and StringBuilder does not override equals method so here it will use Object class equals() i.e., reference comparison.

- **Heap and SCP**

String s1= *new* String("xyz");
String s2= *new* String("xyz");

Heap	SCP
s1---> xyz	xyz
S2---> xyz	

When we create object using new keyword it gets stored in both heap & copy of the same in SCP but, reference will point to heap area only.
Here object creation is *mandatory*.
SCP object will *not* be eligible for garbage collection because internal implicit reference will be maintained by JVM.

String s3= "xyz";
String s4= "xyz";

Heap	SCP
	s3, s4 ---> xyz

Here we have used literal. So, s3 get stored in SCP *only*, and reference will point to SCP area only.
Here object creation is *not mandatory*. In SCP before creating object JVM first checks whether the same content is available or not, if not then object will be created but if the same content is already there in SCP it will point to the same available object(re-usability).

Combined above all examples:

Heap	SCP
s1---> xyz	s3, s4 ---> xyz
S2---> xyz	

Note: If at runtime any new String object get created then that object is stored *only* in *Heap* area. And if a particular object is not pointing to any references then it is eligible for GC.

Example:

String s1= *new* String("xyz");

s1.concat("abc");

s1= s1.concat("pqr");

Heap	SCP
s1---> xyz	xyz
xyzabc(*GC*)	abc
s1---> xyzpqr	pqr

Here, s1 will first point to *xyz*, but at the end it will point to *xyzpqr* and then *xyz* will be eligible for GC as it will not have any references pointing to it.

Example 1:

String s1= *new* String(“spring”);

s1.concat(“fall”);

String s2= s1.concat(“winter”);

s2.concat(“summer”);

Heap	SCP
s1---> spring	spring
springfall(*GC*)	fall
s2---> springwinter	winter
springwintersummer(*GC*)	summer

System.out.print(s1); ----------> spring

System.out.print(s2); ----------> springwinter

Example 2:

String s1= *new* String(“springfall”);

String s2= *new* String(“springfall”);

1.SOP(s1==s2);

String s3=”springfall”;

2.SOP(s1==s3);

String s4=”springfall”;

3.SOP(s4==s3);

String s5= ”spring” + ”fall”;

4.SOP(s4==s5);

String s6=” spring”;

String s7=s6 + ”fall”;

5.SOP(s4==s7);

Heap	SCP
s1---> springfall	s3, s4, s5, s9--->springfall
s2---> springfall	
	s6, s8---> spring
s7---> springfall	fall

1.System.out.print(s1==s2); ----------> false

2.System.out.print(s1==s3); ----------> false

3.System.out.print(s4==s3); ----------> true

4.System.out.print(s4==s5); ----------> true

5.System.out.print(s4==s7); ----------> false

5.System.out.print(s4==s9); ----------> true

final String s8=” spring”;(Note- Every *final* variable is constant literal)

String s9=s8 + ”fall”;

6.SOP(s4==s9);

Case1: String s5=”spring” + ”fall”; --->Here both are literals and processed at compile time. Hence stored only at SCP.

Case2: String s7=s6 + ”fall”; --->Here one is literal and other is variable. So, result is obtained at runtime by creating new object. Hence stored at heap as well as SCP.

- ## Importance of SCP

  Advantage: One object can be used with multiple references. So provides better memory utilization which ultimately improves performance.

  Disadvantage: If among all those multiple references any one reference tries to change the content of the common object to which all references are pointing, then all the remaining references will get affected and to prevent that *immutability* concept came in picture.

Questions:

1.Why SCP is available only for String and not to StringBuffer?

Ans. String is used more frequently than StringBuffer. Hence memory management is provided for String only.

2.Why String objects are immutable but StringBuffer objects are mutable?

Ans. Because of SCP. String provides SCP in which multiple references access same object(reusability). So, if try to change the object content with any of the references remaining will get affected. Hence String objects are immutable. In case of StringBuffer there is no such concept of SCP. Here every time new object is created means each object is individual/independent and hence StringBuffer objects are mutable.

3.Apart from String object is any other object immutable in java?

Ans. Yes. All wrapper class objects are immutable.

Note : All immutable objects in java are always *thread-safe*.

- **Important constructors of String class**

String is sequence of characters.

1. String s =new String();
   -Creates an empty String object.
2. String s =new String(String literal);
3. String s =new String(StringBuffer sb);
   -String equivalent of StringBuffer.
4. String s =new String(StringBuilder sb);
   -String equivalent of StringBuilder.
5. String s =new String(char[] ch);
   -Generate String for char array.
6. String s =new String(byte[] b);
   -Generate String for char array by converting byte value to ASCII.

- **Important methods of String class**

1. public char *charAt*(int index)
   -returns character at specified index. If index not applicable then *ArrayIndexOutOfBoundException*
   -e.g., s.charAt(3);
2. public String *concat*(String literal/object)
   -to merge two strings, we can also use "+" operator
   -e.g. s.concat("string");
3. public boolean *equals*(object o)
4. public boolean *equalsIgnoreCase*(object o)
5. public boolean *isEmpty*(object o)
6. public int *length()*
   Note: *length* variable to get array length.
   length() method to get length of String.
7. public String replace(char *old*, char *new*)
   -Every *old* will get replaced by *new*.
8. public String *substring*(int beginIndex)
   - returns string from specified index till the end of string.
9. public String *substring*(int beginIndex, int endIndex)
   - returns string from beginIndex to endIndex-1
10. public int *indexOf*(char ch)
    -returns index of first occurrence. If specified char not present returns -1

11. public int *lastIndexOf*(char ch)
12. public String *toLowerCase*()
13. public String *toUpperCase*()
14. public String *trim*()
    -removes blank spaces at the beginning and at the end of the String but, not in the middle of String.

Note: Once we create a string object, we are not allowed to perform any changes in that object. If we are trying to perform any changes, and if there is a change in content, with those changes a new object will get created. And if there is no change in content existing object will be reused irrespective of whether the object is in heap or SCP.

Note: Because of runtime operation if a new object is required to create, compulsory that new object will be placed in heap area only and not in SCP.

Note: If after certain operation on current object there is no change in that object then the current object will be reused.

Example 1:

String s1= *new* String("spring");

String s2= s1.toUpperCase();

String s3= s1.toLowerCase();

Heap	SCP
s1, s3---> spring	spring
s2---> SPRING	

System.out.print(s1==s2); ----------> false

System.out.print(s1==s3); ----------> true

Example 2:

String s1= "spring";

String s2= s1.toString();

String s3= s1.toLowerCase();

String s4= s1.toUpperCase();

String s5=s4.toUpperCase();

String s6=s5.toLowerCase();

Heap	SCP
	s1, s2, s3---> spring
s4, s5---> SPRING	
s6---> spring	

System.out.print(s1==s2 && s1==s3 && s2==s3); ---------> true

System.out.print(s1==s6); ----------> false

System.out.print(s1==s4); ----------> false

System.out.print(s5==s4); ----------> true

System.out.print(s1.equals(s6)); ----------> true

- **How to create own immutable class**

```
final public class Test {
	private int 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
	}
}
```

final – So that child class cannot be created and method will not be override.

private – So that variable can't be accessed outside the class.

All immutable classes are declared as *final.*

- **final vs immutable**

  final is related to a variable and *immutability* is related to an object.
  e.g., *final* StringBuffer sb=new StringBuffer("xyz");
  here *sb* is a reference variable which is declared as final which doesn't mean its immutable. It means you can't assign same reference to the new object, but we can perform changes to available object.
  e.g., sb=new StringBuffer("abc");

Conclusion- We cannot make StringBuffer object immutable.

• Need of StringBuffer

If the content is not fixed and keep on changing it is recommended not to use String concept because of immutability property. To handle such requirement StringBuffer can be used where objects are mutable i.e., all frequent changes will be performed in existing object only and no new object will get created.

• Important constructors of StringBuffer class

- *length* : number of characters present
- *capacity* : number of characters the object can hold

Note : In case of String capacity is same as length or can say capacity concept is not explicitly applicable like StringBuffer.

1. StringBuffer sb =new StringBuffer();
   -Default initial capacity=**16**
   -New capacity=**(CurrentCapacity + 1)*2**
   Note: if we try to add 17th element then only new capacity will get incremented automatically i.e., only after complete filling of initial capacity.
2. StringBuffer sb =new StringBuffer(int initialCapacity);
3. StringBuffer sb =new StringBuffer(String s);
   -Creates an equivalent StringBuffer for given String with **capacity=*s.length()* + Capacity of StringBuffer**

• Important methods of StringBuffer class

1. public int *length*()
2. public int *capacity*()
3. public char *charAt*(int index)
   -returns character at specified index. If index not applicable then *StringIndexOutOfBoundException*
4. public void *setCharAt*(int index, char ch)
5. public StringBuffer *append*(any data type will work)[overloaded method]
   -inserts data at last.
6. public StringBuffer *insert*(int index, any type data)[overloaded method]
   -inserts data at specified index.
7. public StringBuffer *delete*(int beginIndex, int endIndex)
   -deletes data from beginIndex to endIndex-1.
8. public StringBuffer *deleteCharAt*(int Index)
9. public StringBuffer *reverse*()
   -order of characters will be reversed

10. public void *setLength*(int length)
    -to get characters at specified length
11. public void *ensureCapacity*(int capacity)
    -to increased capacity dynamically
12. public void *trimToSize*()
    -to de-allocate unused memory

- ## Need of StringBuilder

  All methods in the StringBuffer are *synchronized*. So, at a time only one thread can operate on an object which slows down the performance. StringBuffer cannot be used for multi-threaded environment. To overcome this problem StringBuilder (*1.5v*) concept is used.
  StringBuilder methods and constructors are same as that of StringBuffer.
  StringBuilder is *non-synchronized* version of StringBuffer.

- ## StringBuffer vs StringBuilder

StringBufffer	StringBuilder
1.Most of the methods are *synchronized*.	1.Methods are *not synchronized*.
2.Object are *thread safe*.	2.Object are *not thread safe*.
3.Relatively *low* performance as only one thread can operate at a time.	3.Relatively *high* performance as multiple threads can operate simultaneously.
4.Introduced in 1.0 version	4.Introduced in 1.5 version

- ## Method chaining

  Method chaining is possible in all three String, StringBuffer, StringBuilder.

Whenever we are trying to print any object reference internally *toString*() method is called.

-Object Class ----> toString()----> ClassName@<hashcode in hexadecimal>

-Custom Class ----> toString()-----> need to override in custom class to provide meaningful string representation otherwise object class toString() will get executed.