STRING VS STRINGBUFFER

STRING:

In Java, string is basically an object that represents sequence of char values. Strings in Java are Objects that are backed internally by a char array. string are basically *"immutable"* objects in JAVA.

Once we create String object we can't perform any changes in that object. If you trying to perform to change the String object with those changes a new object by default will be created. This non-changeable behavior is called as immutability.

Immutable means the object can not be changed to its value or state once it is created. Whenever we try to change a immutable object it creates a new object to store the changeable values.

STRINGBUFFER:

StringBuffer class is used for representing changing strings. Java StringBuffer class is used to create mutable String objects. StringBuffer is a peer class of String that provides much of the functionality of strings. StringBuffer represents growable and writable character sequences.

StringBuffer is *"mutable"*. In StringBuffer concept once we create a StringBuffer object we can perform any changes in that object. This changeable behavior is called as mutability.

Mutable means the object can be changed to any value or state without adding a new object.

```
public class test
{
public static void main(String args[])
{
String s = new String("ATISH");
s.concat("SAHU");
System.out.println(s);

System.out.println(s.concat("sahu"));

StringBuffer ss = new StringBuffer("LIPUN");
ss.append("sahu");
System.out.println(ss);
}
}
```

System.out.println(s);

The output of the above line is ATISH because it is a String object which is immutable.

System.out.println(s.concat(s));

The above line will create a new object to store a new updated value that happens to string object s because of immutability. The output is ATISHsahu. And this new object becomes garbage value cause of no reference value is attached to the operation.

System.out.println(ss);

The output of above line is LIPUNsahu. Because the object ss is StringBufffer object which is mutable. And it doesn't contain any new object for update the new changeable value.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
ATISH
ATISHsahu
LIPUNsahu
```

== Operator VS equals() method:

== operator always meant for reference comparison. Reference comparison defines if both references point into the same object then only it will returns true.

Object class equals() meant for reference comparison / address comparison. In String class equals() overridden for content comparison. In StringBuffer equals() is not overridden for reference comparison.

```
public class test
{
public static void main(String args[])
{
String s1 = new String("ATISH");
String s2 = new String("ATISH");

System.out.println(s1 == s2);
System.out.println(s1.equals(s2));

StringBuffer ss1 = new StringBuffer("ATISH");
StringBuffer ss2 = new StringBuffer("ATISH");

System.out.println(ss1 == ss2);
System.out.println(ss1.equals(ss2));
}
}
```

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
false
true
false
false
```

```
public class test
{
public static void main(String args[])
{

String s1 = "ATISH";
String s2 = "ATISH";

System.out.println("s1 = "+s1.hashCode());
System.out.println("s2 = "+s2.hashCode());
System.out.println("s1 == s2 -> "+(s1 == s2));
System.out.println("s1.equals(s2) -> "+s1.equals(s2));

System.out.println("==============================");

String s3 = "lipun";
String s4 = "LIPUN";

System.out.println("s3 = "+s3.hashCode());
System.out.println("s4 = "+s4.hashCode());
System.out.println("s3 == s4 -> "+(s3 == s4));
System.out.println("s3.equals(s4) -> "+s3.equals(s4));

}
}
```

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
s1 = 62604107
s2 = 62604107
s1 == s2 -> true
s1.equals(s2) -> true
==============================
s3 = 102979692
s4 = 72441932
s3 == s4 -> false
s3.equals(s4) -> false
```

hashCode() is overridden in String class and the hashCode is computed using the below formula

s[0]31^(n-1) + s[1]31^(n-2) + … + s[n-1]

So two different String object will give you same hashCode value, if they hold the same value because hashCode for Strings are computed based on the value.

```
public class test
{
public static void main(String args[])
{

StringBuffer s1 = new StringBuffer("ATISH");
StringBuffer s2 = new StringBuffer("ATISH");

System.out.println("s1 = "+s1.hashCode());
System.out.println("s2 = "+s2.hashCode());
System.out.println("s1 == s2 -> "+(s1 == s2));
System.out.println("s1.equals(s2) -> "+s1.equals(s2));

System.out.println("==============================");

StringBuffer s3 = new StringBuffer("lipun");
StringBuffer s4 = new StringBuffer("LIPUN");

System.out.println("s3 = "+s3.hashCode());
System.out.println("s4 = "+s4.hashCode());
System.out.println("s3 == s4 -> "+(s3 == s4));
System.out.println("s3.equals(s4) -> "+s3.equals(s4));

}
}
```

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
s1 = 366712642
s2 = 1829164700
s1 == s2 -> false
s1.equals(s2) -> false
==============================
s3 = 2018699554
s4 = 1311053135
s3 == s4 -> false
s3.equals(s4) -> false
```

The hashCode method of the Object class is a native method which is typically implemented by converting the internal address of the object into an integer as the hash code value or may not as it depends on the internal implementation of the JVM. This is because the StringBuffer is a mutable object unlike the String you can easily modify the state of the StringBuffer object after its creation. This makes it unsuitable for use in any "hash" based data structures like a HashMap as the it will be inconsistent.

```
public class test
{

public static void main(String args[])
{
String s1= "atish";
String s2 = "atish";
System.out.println("s1: "+s1.hashCode());
System.out.println("s2: "+s2.hashCode());
System.out.println("s1 == s2"+(s1 == s2));

String s3 = new String("LIPUN");
String s4 = new String("LIPUN");
System.out.println("s3: "+s3.hashCode());
System.out.println("s4: "+s4.hashCode());
System.out.println("s3 == s4"+(s3 == s4));
}
}
```

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
s1: 93141867
s2: 93141867
s1 == s2true
s3: 72441932
s4: 72441932
s3 == s4false
```

```
import java.lang.*;

public class test     {

public static void main(String args[])    {

        Integer i = new Integer(60);

        System.out.println(i);                  //60

        System.out.println(i.hashCode());                  //60

        Integer j = new Integer(60);

        System.out.println(j);                  //60

        System.out.println(j.hashCode());       //60

        System.out.println(i == j);//false

        System.out.println(i.equals(j));   //true

}       }
```

HEAP AND STRING CONSTANT POOL(SCP):

String s = new String("atish");

In the above statement two objects will created. In heap area there is a object will created and inside the object the value is store which is “atish” and “s” is the reference variable of the object. For every string literal for thee future purpose one more object will created in “String Constant Pool (SCP)” and in this no explicit reference variable, but internally implicit reference variable is maintained by JVM. Until 1.6 version SCP is the part of method area. From 1.7 version onwards in heap area some part will reserved for SCP. When it was in method area the SCP size was fixed after it was part of the heap area SCP can extend its size.

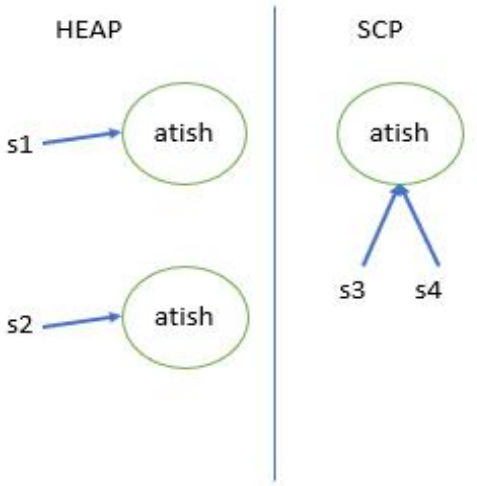
String a = "ATISH";

In the above statement one object will created in SCP area. But during execution first JVM will check is there any object with the content already in SCP or not. If it is already there then only it will reuse the same object. If the object is not already there then only a new object will be created and pointing to the object. In this above statement object creation is not mandatory.

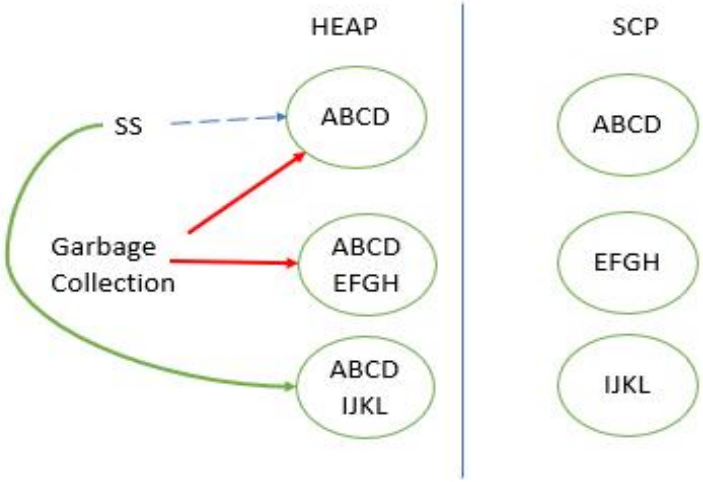
In runtime operation if an object is required to create that object is going to create only in the heap area but not in SCP area. For this new object there is no reference variable is connected so that object eligible for garbage collection.

```
public class test
{
public static void main(String args[])
{
	String s1 = new String("atish");
	String s2 = new String("atish");

	String s3 = "atish";
	String s4 = "atish";

	System.out.println(s1.hashCode());//93141867
	System.out.println(s2.hashCode());//93141867
	System.out.println(s3.hashCode());//93141867
	System.out.println(s4.hashCode());//93141867


	String ss = new String("ABCD");
	ss.concat("EFGH");
	ss = ss.concat("IJKL");
	System.out.println(ss);//ABCDIJKL

}
}
```

```
String s1 = new String("atish");
String s2 = new String("atish");

String s3 = "atish";
String s4 = "atish";
```




```
String ss = new String("ABCD");
ss.concat("EFGH");
ss = ss.concat("IJKL");
System.out.println(ss);//ABCDIJKL
```




```
public class test {
public static void main(String args[]) {
String s1 = new String("spring");
s1.concat("fall");
String s2 = s1.concat("winter");
s2.concat("summer");
System.out.println(s1);
System.out.println(s2);
}     }
```

```
C:\Users\Atish kumar
spring
springwinter
```


```
public class test {
public static void main(String args[]) {
String s1 = new String("spring");
s1.concat("fall");
String s2 = s1.concat("winter");
s2.concat("summer");
System.out.println(s1);
System.out.println(s2);
}     }
```


IF A OPERATION HAVE TWO CONSTANTS THEN THE OPERATION HAPPEN IN COMPILE TIME.

IF A OPERATION HAVE ATLEAST ONE VARIABLE THEN THE OPERATION HAPPEN IN RUNTIME.

EVERY FINAL VARIABLE CAN BE REPLACED AT COMPILE TIME ONLY. SO THE VALUE IS BECOME A CONSTANT VALUE.

HEAP

S1 -> (YCCM)

S2 -> (YCCM)

S7 -> (YCCM)

SCP

YCCM <- S3 S4 S5 S9

YC <- S6 S8

CM<- MAINTAINED BY JVM

```
public class test {
public static void main(String args[])  {

String s1 = new String("You cannot change Me");
String s2= new String("You cannot change Me");
System.out.println(s1 == s2);//FALSE

String s3 = "You cannot change Me";
System.out.println(s1 == s3);//FALSE
System.out.println(s2 == s3);//FALSE

String s4 = "You cannot change Me";
System.out.println(s1 == s4);//FALSE
System.out.println(s2 == s4);//FALSE
System.out.println(s3 == s4);//TRUE

String s5 = "You cannot " + "change Me";
System.out.println(s1 == s5);//FALSE
System.out.println(s2 == s5);//FALSE
System.out.println(s3 == s5);//TRUE
System.out.println(s4 == s5);//TRUE

String s6 = "You cannot ";
String s7 = s6 + "change Me";
System.out.println(s1 == s7);//FALSE
System.out.println(s2 == s7);//FALSE
System.out.println(s3 == s7);//FALSE
System.out.println(s4 == s7);//FALSE
System.out.println(s5 == s7);//FALSE

final String s8 = "You cannot ";
String s9 = s8 + "change Me";
System.out.println(s1 == s9);//FALSE
System.out.println(s2 == s9);//FALSE
System.out.println(s3 == s9);//TRUE
System.out.println(s4 == s9);//TRUE
System.out.println(s5 == s9);//TRUE
System.out.println(s7 == s9);//FALSE

}       }
```

IMPORTANCE OF STRING CONSTANT POOL:

ADVANTAGE/IMPORTANCE:

If any string object repeatedly required never recommended to create separate object for every requirement.

So in java create one string object and share the same object for all requirements and it is possible because of SCP concept.

Because of SCP performance will be improved, memory utilization is improved. If reusing concept is not there immutability not required.

DISADVANTAGE:

If multiple references are pointing to the same object but If any reference want to change its value then it will affect to all the other reference which are pointing to the object.

Once we create a string object we are not allowed to change its content. If any reference want to change its content with those changes a new object by default will be created. And that particular reference who want to change will be reassigned. This is nothing but immutability concept.

IMPORTANT QUESTION ON STRING AND STRING BUFFER:

Q. Why SCP concept is only available only for String object but not for StringBuffer?

ANS. String is a regular customer means the most commonly used object is String object. It is for the String special privileges are there. StringBuffer you may use or you may not use also. There are several application where StringBuffer is not used but there is no application without using String. That's why java provided a special memory management SCP for String but StringBuffer is very rarely used Object so special memory management not required for StringBuffer.

Q. Why String objects are immutable where as StringBuffer objects are mutable?

ANS. In the case of String just because of SCP same object can be reused multiple times. By using one reference if we try to change the content the remaining references will going to be affected. Once we create a String object we are not allowed to change its content. Because it shared with multiple references. If any person trying to change the content with those changes a new object will be created and the existing object you are not allowed to change.

In StringBuffer there is no SCP concept. If SCP concept is not there reusing same object such thing is not there. So every time a separate object have to be created. If we try to change the content of any object it will not affect to other objects that's why StringBuffer is mutable.

Q. In addition to String objects any other Objects are immutable in JAVA?

ANS. All Wrapper class Objects are also immutable. Same reusable of object is also here for a certain level. Example: Byte, Short, Integer, Long, Float, Double, Character, Boolean

IMPORTANT CONSTRUCTORS OF STRING CLASS:

Zero length strings are also allowed in java. for a given StringBuffer I want to create a equivalent String object.

String s = new String(); //empty string object in heap area.

String s1 = new String(String literal);

String S2 = new String(StringBuffer sb);

//for the given string object an equivalent for the given StringBuffer object an equivalent string will be created.

String s3 = new String(StringBuilder SD);

//for the given StringBuilder can you create a string object.

String s4 = new String(char[] ch);

//for given char array you can able to create an equivalent string object.

String s5 = new String(byte[] b); //for given byte array we can create a string object.

```
public class test {
public static void main(String args[])  {

char[] ch = {'j', 'a', 'v', 'a'};
String s = new String(ch);
System.out.println(s);  //java

byte[] bb = {96, 97, 98, 99, 100};
String ss = new String(bb);
System.out.println(ss); //`abcd

}     }
```

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
java
`abcd

C:\Users\Atish kumar sahu\desktop>
```

STRING & STRINGBUFFER IN JAVA PROGRAMMING

String Methods In Java:

charAt():

The charAt() method returns the character at the specified index in a string. The index of the first character is 0, the second character is 1, and so on.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
charAt() : F
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: 15
        at java.lang.String.charAt(Unknown Source)
        at test.main(test.java:10)

C:\Users\Atish kumar sahu\desktop>
```

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);
		String s = "ABCDEFGHIJKL";

		System.out.println("charAt() : "+s.charAt(5));
		System.out.println("charAt() : "+s.charAt(15));
	}
}
```

concat():

The concat() method appends (concatenate) a string to the end of another string.

Command Prompt

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
concat() : ABCDEFGHIJKL Atish
concat() : ABCDEFGHIJKL         Lipun

C:\Users\Atish kumar sahu\desktop>
```

test.java

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);
		String s = "ABCDEFGHIJKL";

		System.out.println("concat() : "+s.concat(" Atish"));
		System.out.println("concat() : "+s.concat("         Lipun"));
	}
}
```

equals():

The equals() method compares two strings, and returns true if the strings are equal, and false if not.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
equals() : true
equals() : false
equals() : false

C:\Users\Atish kumar sahu\desktop>
```

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);
		String s = "ABCDEFGHIJKL";

		System.out.println("equals() : "+s.equals("ABCDEFGHIJKL"));
		System.out.println("equals() : "+s.equals("abcdefghijkl"));
		System.out.println("equals() : "+s.equals("Atish"));
	}
}
```

equalsIgnoreCase():

The equalsIgnoreCase() method compares two strings, ignoring lower case and upper case differences. This method returns true if the strings are equal, and false if not.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
equalIgnoreCase() : true
equalIgnoreCase() : false
equalIgnoreCase() : true

C:\Users\Atish kumar sahu\desktop>
```

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);
		String s = "ABCDEFGHI";

		System.out.println("equalIgnoreCase() : "+s.equalsIgnoreCase("abcdefghi"));
		System.out.println("equalIgnoreCase() : "+s.equalsIgnoreCase("ATISH"));
		System.out.println("equalIgnoreCase() : "+s.equalsIgnoreCase("ABCDEFGHI"));
	}
}
```

isEmpty():

The isEmpty() method checks whether a string is empty or not. This method returns true if the string is empty (length() is 0), and false if not.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
isEmpty()  : true
isEmpty() : false
isEmpty() : false

C:\Users\Atish kumar sahu\desktop>
```

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);
		String s1 = "";
		System.out.println("isEmpty()  : "+s1.isEmpty());

		String s2 = " ";
		System.out.println("isEmpty() : "+s2.isEmpty());

		String s3 = "Atish";
		System.out.println("isEmpty() : "+s3.isEmpty());
	}
}
```

length():

The length() method returns the length of a specified string. The length of an empty string is 0.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
length() : 12

C:\Users\Atish kumar sahu\desktop>
```

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);
		String s1 = "Abcdefghijkl";

		System.out.println("length() : "+s1.length());
	}
}
```

replace():

The replace() method searches a string for a specified character, and returns a new string where the specified character(s) are replaced.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
replace() : abcdefghijkl

C:\Users\Atish kumar sahu\desktop>
```

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);
		String s1 = "Abcdefghijkl";

		System.out.println("replace() : "+s1.replace("A","a"));
	}
}
```

substring():

Returns a new string which is the substring of a specified string.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
substring() : efghijkl

C:\Users\Atish kumar sahu\desktop>
```

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);
		String s1 = "Abcdefghijklmnopqrst";
		System.out.println("substring() : "+s1.substring(4,12));
	}
}
```

indexOf():

The indexOf() method returns the position of the first occurrence of specified character(s) in a string. Use the lastIndexOf method to return the position of the last occurrence of specified character(s) in a string.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
indexOf() : 5
indexOf() : -1

C:\Users\Atish kumar sahu\desktop>
```

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);

		String s1 = "Abcdefghijofrst";
		System.out.println("indexOf() : "+s1.indexOf("f"));

		String s2 = "AbCdEf";
		System.out.println("indexOf() : "+s2.indexOf("B"));
	}
}
```

lastIndexOf():

The lastIndexOf() method returns the position of the last occurrence of specified character(s) in a string. Use the indexOf method to return the position of the first occurrence of specified character(s) in a string.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
lastIndexOf() : 11
lastIndexOf() : -1

C:\Users\Atish kumar sahu\desktop>
```

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);

		String s1 = "Abcdefghijofrst";
		System.out.println("lastIndexOf() : "+s1.lastIndexOf("f"));

		String s2 = "AbCdEfbb";
		System.out.println("lastIndexOf() : "+s2.lastIndexOf("B"));
	}
}
```

toLowerCase() & toUpperCase():

The toLowerCase() method converts a string to lower case letters. The toUpperCase() method converts a string to upper case letters.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
S1 : abcdefghij
toUpperCase() : ABCDEFGHIJ

S2 : ABCDEFGHIJKL
toLowerCase() : abcdefghijkl

C:\Users\Atish kumar sahu\desktop>
```

File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);

		String s1 = "abcdefghij";
		System.out.println("S1 : "+s1);
		System.out.println("toUpperCase() : "+s1.toUpperCase());

		String s2 = "ABCDEFGHIJKL";
		System.out.println("\nS2 : "+s2);
		System.out.println("toLowerCase() : "+s2.toLowerCase());
	}
}
```

trim():

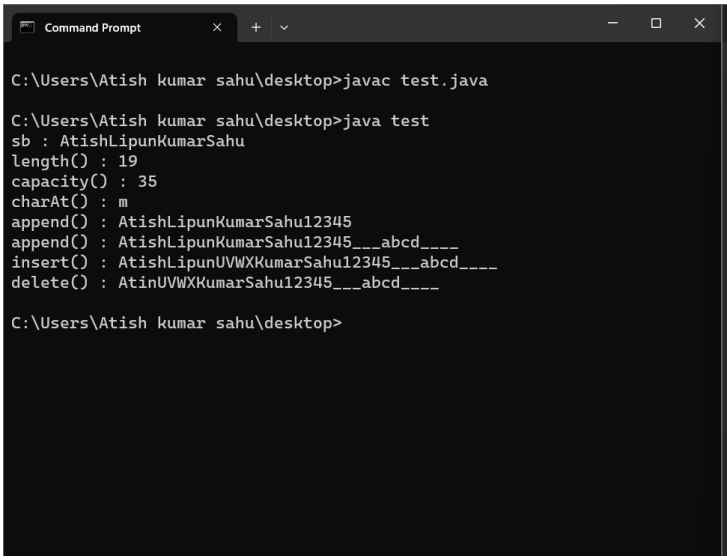
The trim() method removes whitespace from both ends of a string. This method does not change the original string.

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
S1 :                Atish Kumar Sahu
length : 43
S2 :  Atish Kumar Sahu
length : 16

C:\Users\Atish kumar sahu\desktop>
```

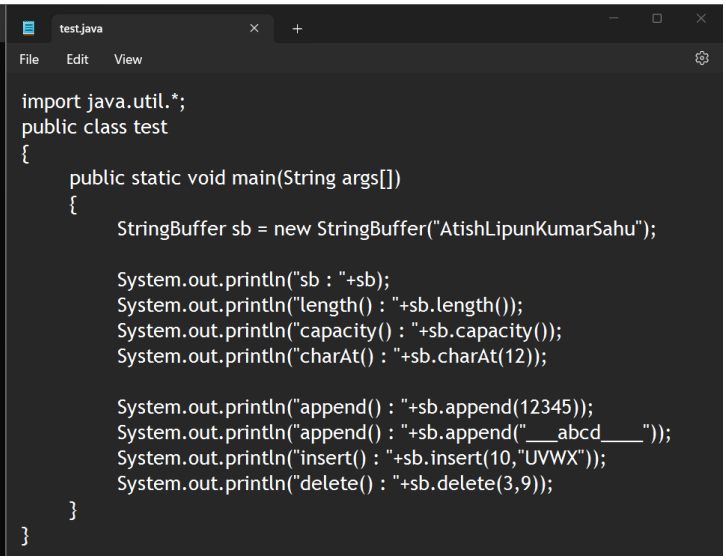
File Edit View

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		Scanner in = new Scanner(System.in);

		String s1 = "            Atish Kumar Sahu          ";
		System.out.println("S1 : "+s1);
		System.out.println("length : "+s1.length());

		String s2 = s1.trim();
		System.out.println("S2 :  "+s2);
		System.out.println("length : "+s2.length());
	}
}
```

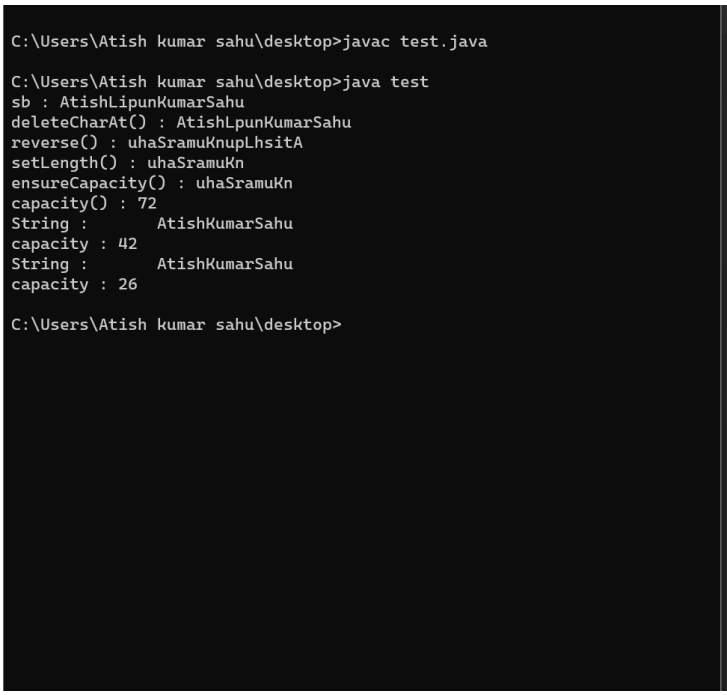
StringBuffer Method In Java:



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
sb : AtishLipunKumarSahu
length() : 19
capacity() : 35
charAt() : m
append() : AtishLipunKumarSahu12345
append() : AtishLipunKumarSahu12345___abcd____
insert() : AtishLipunUVWXKumarSahu12345___abcd____
delete() : AtinUVWXKumarSahu12345___abcd____

C:\Users\Atish kumar sahu\desktop>
```



```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		StringBuffer sb = new StringBuffer("AtishLipunKumarSahu");

		System.out.println("sb : "+sb);
		System.out.println("length() : "+sb.length());
		System.out.println("capacity() : "+sb.capacity());
		System.out.println("charAt() : "+sb.charAt(12));

		System.out.println("append() : "+sb.append(12345));
		System.out.println("append() : "+sb.append("___abcd____"));
		System.out.println("insert() : "+sb.insert(10,"UVWX"));
		System.out.println("delete() : "+sb.delete(3,9));
	}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
sb : AtishLipunKumarSahu
deleteCharAt() : AtishLpunKumarSahu
reverse() : uhaSramuKnupLhsitA
setLength() : uhaSramuKn
ensureCapacity() : uhaSramuKn
capacity() : 72
String :       AtishKumarSahu
capacity : 42
String :       AtishKumarSahu
capacity : 26

C:\Users\Atish kumar sahu\desktop>
```

```
import java.util.*;
public class test
{
	public static void main(String args[])
	{
		StringBuffer sb = new StringBuffer("AtishLipunKumarSahu");

		System.out.println("sb : "+sb);
		System.out.println("deleteCharAt() : "+sb.deleteCharAt(6));
		System.out.println("reverse() : "+sb.reverse());

		sb.setLength(10);
		System.out.println("setLength() : "+sb);

		sb.ensureCapacity(50);
		System.out.println("ensureCapacity() : "+sb);

		System.out.println("capacity() : "+sb.capacity());

		StringBuffer sb1 = new StringBuffer("      AtishKumarSahu      ");
		System.out.println("String : "+sb1);
		System.out.println("capacity : "+sb1.capacity());

		sb1.trimToSize();
		System.out.println("String : "+sb1);
		System.out.println("capacity : "+sb1.capacity());
	}
}
```

IMPORTANT CONCLUSIONS ABOUT STRING IMMUTABILITY:

```
public class test {
public static void main(String args[]) {

String s1 = new String("atish");
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
System.out.println(s1 == s2); //false
System.out.println(s2 == s3); //false
System.out.println(s1 == s3); //true

}    }
```

```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
false
false
true
```

```
public class test {
public static void main(String args[]) {

String s1 = "atish";
String s2 = s1.toString();
String s3 = s1.toLowerCase();
String s4 = s1.toUpperCase();

System.out.println(s1 == s2); //true
System.out.println(s1 == s3); //true
System.out.println(s1 == s4); //false
System.out.println(s2 == s3); //true
System.out.println(s2 == s4); //false
System.out.println(s3 == s4); //false

}    }
```

Once we create string object we are not allowed to perform any changes to content of that object. If we try to change the content of the object with those changes a new object will be created. If no change existing object will be reused whether the object is in heap area or in SCP area.

CREATION OF OWN IMMUTABLE CLASS:

```
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 VS IMMUTABILITY:

By declaring reference variable as final we are not going to get immutability nature. Final and immutability both are different. If you use final you can not reassign that reference variable to any new object.

Final terminology talks about variable but not object. Immutability concept talks about object but not for the variable. By declaring a reference variable as final we never gone to get immutability. How can we make a StringBuffer as immutable? It is impossible to make because internally the methods of StringBuffer is implemented for mutability purpose but not for immutability purpose. If you change the source code or any method present inside StringBuffer then it becomes immutable which is not gonna be happen.

```
public class test {

public static void main(String args[]) {

final StringBuffer SB = new StringBuffer("ABCDEFG");

SB.append("abcdefg");
System.out.println(SB); //ABCDEFGabcdefg

SB = new StringBuffer("ABCD123"); //compile time error
}     }
```

PRACTICE QUESTION AND ANSWER 01:

```
public class test {

public static void main(String args[]) {

String ta = "A";
ta = ta.concat("B");
String tb = "C";
ta = ta.concat(tb);
ta.replace('C', 'D');
ta = ta.concat(tb);
System.out.println(ta); //ABCC

}     }
```

PRACTICE QUESTION AND ANSWER 02:

```
public class test {

public static void main(String args[]) {

String str = " ";
str.trim();
System.out.println(str.equals("")+" "+str.isEmpty()); //false false

}     }
```

PRACTICE QUESTION AND ANSWER 03:

```
public class test {

public static void main(String args[]) {

String s = "ATISH SAHU";
int len = s.trim().length();
System.out.println(len); //10

}     }
```

PRACTICE QUESTION AND ANSWER 04:

```
public class test {

public static void main(String args[]) {

String s = "HELLO WORLD";
s.trim();
int i = s.indexOf(" ");
System.out.println(i); //5

}     }
```

PRACTICE QUESTION AND ANSWER 05:

```
public class test {

public static void main(String args[]) {

String s1 = "java";
String s2 = new String("java");
if(s1.equalsIgnoreCase(s2))
{
System.out.println("equal"); //equal
}
else
{
System.out.println("not equal");
}

}     }
```

NEED OF STRINGBUFFER:

If the content is not fixed keep on changing never recommended to use string concept. For every change it is mandatory to create a new object it cause reduction of performance and memory problem also occur. So if the content is keep on changing then it is highly recommended to use StringBuffer.

Incase of String concept for every change a new object created. Bu incase of StringBuffer all required changes will be performed in the excitable object only. It is introduced in 1.0 version.

STRINGBUFFER CLASS CONSTRUCTORS:

StringBuffer sb1 = new StringBuffer();

In the above statement the StringBuffer is an empty object but the default initial capacity is 16. Once we create a empty StringBuffer object how many characters it can be able to accommodate? 16 character it can able to accommodate. If the 16 character is completed but still if you are trying to add more than 16 character a new StringBuffer object will be created with bigger capacity all the 16 character will be copied also the other characters beyond 16 are placed and the reference variable is pointing to that new StringBuffer object and the old object automatically going to garbage collection. The total thing happen in internally. The capacity of the new object is equal to (current capacity + 1) * 2

```
StringBuffer sb1 = new StringBuffer();
System.out.println(sb1.capacity()); //16
sb1.append("abcdefghijklmnopqrstuvwxyz");
System.out.println(sb1.capacity()); //34
```

Whenever we add new characters to a StringBuffer object it will create a new object which have the more capacity to store but the continuous use of the this operation creates performance reduction and memory management problem. “Based on our requirement we can create a bigger StringBuffer object.” But after this if you want to create more character to that StringBuffer the rule is remain same.

```
StringBuffer sb = new StringBuffer(1000);
System.out.println(sb.capacity()); //1000
```

We can create for a given String an equivalent StringBufffer will be created.

```
StringBuffer sb = new StringBuffer("abcde");
System.out.println(sb.capacity());//21
```

In above statement the capacity is 21 because it add 5 characters to its initial capacity which is 16.

STRINGBUILDER VS STRINGBUFFER:

In StringBuilder no method is synchronized. At a time multiple thread are allowed to operate on StringBuilder object and hence it is not thread safe. Threads are not required to wait to operate on StringBuilder Object and hence relatively performance is high. It is introduced in 1.5 version.

In StringBuffer every method is synchronized. At a time only one thread is allow to operate on StringBuffer object and hence it is thread safe. Threads are required to wait to operate on StringBuffer object and hence relatively performance is slow. Introduced in 1.0 version.

STRING VS STRINHBUFFER VS STRINGBUILDER:

If the content is fixed and not going to change frequently highly recommended to use String. Because the same content reused multiple time we have not to create separate new object so the performance will be increased and memory utilization is also improved. String is always thread safe. Because once we create a String object we are not allowed to change anything to the existing object. If you try to change with that change a new object will be created. All immutable objects by default thread safe because no one is allowed to perform modification on the existing object.

If the content is not fixed and keep on changing but thread safety is required at a time only one thread is allow to operate then it is highly required to go for StringBuffer concept. Because every method present in StringBuffer is synchronized so by default StringBuffer is Thread safe.

If the content is not fixed and keep on changing but if you don't want thread safety multiple threads are allowed to operate at a time on my object then it is highly recommended to use StringBuilder.

METHOD CHAINING:

In method chaining all methods are execute from left to right.

```
public class test {

public static void main(String args[]) {

StringBuilder sb = new StringBuilder();
sb.append("apcd").append("hbdue").reverse().insert(2, "bbdirid").delete(1,4);
System.out.println(sb); //ediriddbhdcpa
}     }
```

PRACTICCE QUESTION 6:

In object class equals method if the objects are different type it gives result as false. Because object class equals method is for reference comparison.

```
public class test    {

public static void main(String args[]) {

StringBuilder sb = new StringBuilder(5);
String s = "";
if(sb.equals(s))
{
System.out.println("match 1");
}
else if(sb.toString().equals(s.toString()))
{
System.out.println("match 2");//match2
}
else
{
System.out.println("no match");
}
}      }
```

In StringBuffer equals() method is not overridden and hence object class() method will be executed. If arguments are different types then equal() method return false. Hence

"sb.equals(s)" return false

But in String class equals() method meant for content comparison. Hence

"sb.toString().equals(s.toString())" returns true.

PRACTICE QUESTION 7: == operator always meant for reference comparison.

```
public class test    {

public static void main(String args[])  {

StringBuilder sb1 = new StringBuilder("abcde");
String str1 = sb1.toString();

String str2 = str1;
System.out.println(str1 == str2); //true

}       }

/*
String str2 = sb1.toString();
System.out.println(str1 == str2);//false
*/

/*String str2 = str1;
System.out.println(str1 == str2); //true
*/

/*
String str2 = new String(str1);
System.out.println(str1 == str2); //false
*/

/*
String str2 = "abcde";
System.out.println(str1 == str2);//false
*/
```

String str2 = “abcde”; in this statement the abcde object is created on SCP area.

String str1 = sb1.toString(); in this statement str1 is available in heap area because of method call. If an object is to create compulsory that object is create into the heap area.

PRACTICE QUESTION AND ANSWER 8:

Which statement will empty the contents of a StringBuilder variable named sb?

Sb.deleteAll();

Sb.delete(0, sb.size());

Sb.delete(0,sb.length()); = correct answer

Sb.removeAll(); it is for collection not for string

PRACTICE QUESTION 9:

```
class mystring
{
String msg;
mystring(String msg)
{
this.msg = msg;
}
}

public class test    {

public static void main(String args[])  {

System.out.println("hello"+new StringBuilder("java se 8"));//hellojava se 8
System.out.println("hello"+new mystring("java se 8"));//hellomystring@15db9742

}       }
```

S.O.P("HELLO"+ new StringBuilder("java se 8")); in this S.O.P statement a new StringBuilder object and trying to print content.

S.O.P("HELLO"+ new mystring("java se 8")); in this S.O.P mystring object is created and trying to print the content.

toString() method whenever we trying to print any object reference internally toString() method will be called

test t1= new test();

sop(t1); ==> sop(t1.toString());

if the test class contain toString() method it will executed. If the test class does not contain toString() Method object class toString() method will be executed which is always going to print

"class name @ hash code in hexadecimal code"

If you want meaningful String representation we can override toString() method in our class.

For meaningful String representation highly recommended to override toString() method in our class. in String class, StringBuffer class, StringBuilder class all wrapper classes, all collection classes toString() is already overridden for meaningful representation.

Whenever we trying to concat a string with an object for the object again toString() method will be called and toString() method return string so concat going to be happen.

```
public class test    {

public static void main(String args[])  {

test t1 = new test();
System.out.println(t1);  //test@15db9742
System.out.println(t1.toString());//test@15db9742

/*here the test class contain object class toString() method*/

}       }
```

```
public class test    {

public String toString()
{
        return "test object";
}
public static void main(String args[])  {

test t1 = new test();
System.out.println(t1);  //test object
System.out.println(t1.toString());//test object

}       }
```

PRACTICE QUESTION 10:

You are developing banking module you have developed a class name masktest have a mask method. You must ensure the mask method return a string that hides all digits of credit card number except last four digit and hyphens that separate each group of 4 digit.

```
class masktest
{
public static String mask(String creditcard)
{
String x = "xxxx-xxxx-xxxx-";

return x + creditcard.substring(15,19);  //xxxx-xxxx-xxxx-5979
}
public static void main(String args[])
{
System.out.println(mask("1234-5678-9010-5979"));
}
}
```

```
class masktest
{
public static String mask(String creditcard)
{
String x = "xxxx-xxxx-xxxx-";

// return x + creditcard.substring(15,19); //xxxx-xxxx-xxxx-5979

StringBuilder sb = new StringBuilder(x);
sb.append(creditcard, 15, 19);
return sb.toString(); //xxxx-xxxx-xxxx-5979
}
public static void main(String args[])
{
System.out.println(mask("1234-5678-9010-5979"));
}
}
```