Module – 3

String Handling

- The String Constructors
- String Length
- Special String Operations
  - String Literals
  - String Concatenation
  - String Concatenation with Other Data Types
- String Conversion and toString( )
- Character Extraction
  - charAt( ), getChars( ), getBytes( ) toCharArray(),
- String Comparison
  - equals( ) and equalsIgnoreCase( ), regionMatches( ) startsWith( ) and endsWith( ), equals() Versus == , compareTo( )
- Searching Strings
- Modifying a String
  - substring( ), concat( ), replace( ), trim( )
- Data Conversion Using valueOf( )
- Changing the Case of Characters Within a String
- Additional String Methods
- StringBuffer ,
- StringBuffer Constructors
  - length( ) and capacity( ), ensureCapacity( ),
  - setLength( ), charAt( ) and setCharAt( ), getChars( ),append( ), insert( ), reverse( ), delete( ) and deleteCharAt( ), replace( ), substring( )
- Additional StringBuffer
- Methods, StringBuilder

String Handling:

String class:

- String is probably the most commonly used class in java library. String class is encapsulated under java.lang package.
- In java, every string that you create is actually an object of type **String**. One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be altered.

StringBuffer class:

- StringBuffer class is used to create a **mutable** string object i.e its state can be changed after it is created.
- It represents growable and writable character sequence. As we know that String objects are immutable, so if we do a lot of changes with **String** objects, we will end up with a lot of memory leak.
- So **StringBuffer** class is used when we have to make lot of modifications to our string. It is also thread safe.

StringBuilder class

StringBuilder is identical to StringBuffer except for one important difference that it is not synchronized, which means it is not thread safe. It's because StringBuilder methods are not synchronized.

1.The String Constructors:

The **String** class supports several constructors.

S.N.	**Constructor & Description**
1	**String()** To create an empty **String**, you call the default constructor. For example, String s = new String(); will create an instance of **String** with no characters in it.
2	**String(byte[] bytes)** The **String** class provides constructors that initialize a string when given a **byte** array. Their forms are shown here: String(byte *asciiChars*[ ]) Here, *asciiChars* specifies the array of bytes
3	**String(byte[] bytes, int startindex, int numchar)**

	This constructs a new String by decoding the specified subarray of bytes using the platform's default charset
4	**String(String obj)** This initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
5	**String(StringBuffer buffer)** This allocates a new string that contains the sequence of characters currently contained in the string buffer argument.
6	**String(StringBuilder builder)** This allocates a new string that contains the sequence of characters currently contained in the string builder argument.
7	**String(int[] codePoints, int startindex, int numchar)** This allocates a new String that contains characters from a subarray of the Unicode code point array argument.
8	**String(char[] value)** This allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
9	**String(char[] value, int startindex, int numchar)** This allocates a new String that contains characters from a subarray of the character array argument.

Program:

```
public class StrCon {
        public static void main(String[] args) {
        String a=new String();
        System.out.println("Empty String"+a);
        char ch[]={'a','b','c','d'};
        String b=new String(ch);
        System.out.println("String with one argument as Char="+b);
        String c=new String(ch,1,3);
        System.out.println("String with Three argument as Char="+c);
        String d=new String(b);
        System.out.println("String with String object="+d);
        byte e[]={65,66,67,68,69};
        String f=new String(e);
        System.out.println("byte to String="+e);
        String g=new String(e,1,3);
        System.out.println("byte to string for subbyte="+g);
        StringBuffer h=new StringBuffer("hello");
        String i=new String(h);
        System.out.println("StringBuffer to String="+i);
        StringBuilder j=new StringBuilder("welcome");
        String k=new String(j);
        System.out.println("StringBuilder to Stirng="+k);
        int l[]={66,67,68,69,70};
        String m=new String(l,1,3);
        System.out.println("codepoint to String="+m);
        }
}
```

output

```
Empty String
String with one argument as Char=abcd
String with Three argument as Char=bcd
String with String object=abcd
byte to String=[B@19821f
byte to string for subbyte=BCD
StringBuffer to String=hello
StringBuilder to Stirng=welcome
codepoint to String=CDE
```

2.String Length():

Methods used to obtain information about an object are known as **accessor methods**. One accessory method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

ex:

```
String a=”hello”;
System.out.println(“length of string”+a.length());  //5
```

3. Special String Operations:

These operations include

- The automatic creation of new string instances from string literals.
- Concatenation of multiple String objects by use of the + operator.
- The conversion of other data types to a string representation.

There are explicit methods available to perform all these functions, but JAVA does them automatically as a convenience for the programmer and to add clarity

3.1 String Literals:

String literal is a simple string enclosed in double quotes " ". A string literal is treated as a String object. we can create a **String** instance from an array of characters by using the **new** operator.

```
char ch[]={‘a’,’b’,’c’};
String a=new String(ch);
```

Java automatically constructs a **String** object. Thus, you can use a string literal to initialize a **String** object.

```
String b=”hello”;
System.out.println(“Length of string=”+“hello”.length());
```

3.2 String Concatenation:

The + operator, which concatenates 2 strings, producing a String object as the result.This allows you to chain together a seres of + operations.

ex:

```
String a=”wel come”;
String b=”cec”;
System.out.println(a+”to”+b);  //wel come to CEC
```

3.3 String Concatenation with Other Data Types

You can concatenate strings with other types of data.

```
ex:
String msg=”value of a”;
int a=10;
System.out.println(msg+a); // value of a 10
```

program:

```
public class A {
public static void main(String[] args) {
char ch[]={'a','b','c'};
String a=new String(ch);
System.out.println("String literals="+a);
String b="hello";
System.out.println("String literals="+b);
System.out.println("concatenation of string="+a+b);
int a1=10;
System.out.print("concatenation of string with other data type="+a1);
}
}
```

output:

```
String literals=abc
String literals=hello
concatenation of string=abchello
concatenation of string with other data type=10
```

4. String Conversion and toString( ):

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**. **valueOf( )** is overloaded for all the simple types and for type **Object**.

For the simple types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called. For objects, **valueOf( )** calls the **toString( )** method on the object. We will look more closely at **valueOf( )**

String.valueOf();

Every class implements **toString( )** because it is defined by **Object**. However, the default implementation of **toString( )** is seldom sufficient. For most important classes that you create, you will want to override **toString( )** and provide your own string representations. Fortunately, this is easy to do. The **toString( )** method has this general form:

String toString( )

program:

```
public class CO {
public static void main(String[] args) {
String a="hello";
int b=10;
char c='a';
System.out.println("STring to String as a object="+String.valueOf(a));
System.out.println("Int to String as a object="+String.valueOf(b));
System.out.println("char to String as a object="+String.valueOf(c));
Integer a1=10;
System.out.println("Integer to string"+a1.toString());
}
}
```

output:

```
String to String as a object=hello
Int to String as a object=10
char to String as a object=a
Integer to string=10
```

5. Character Extraction:

The String class provides a no.of ways in which characters can be extracted from a String object.

5.1.charAt()

To extract a single character from a **String** It has this general form:

char charAt(int *where*)

Here, *where* is the index of the character that you want to obtain. The value of *where* must be nonnegative and specify a location within the string. **charAt( )** returns the character at the specified location. For example,

char ch;

ch = "abc".charAt(1);

assigns the value "b" to ch.

5.2 **getChars( )**

If you need to extract more than one character at a time, you can use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*.

```
ex: String a=”hello “;
    char b[]=new char[10];
    a.getChars(1,3,b,0);
```

5.3 **getBytes( )**

There is an alternative to **getChars( )** that stores the characters in an array of bytes. This method is called **getBytes( )**, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

byte[ ] getBytes( )

Other forms of **getBytes( )** are also available. **getBytes( )** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters.

ex:

String a=”hello;

byte b[]=a.getBytes();

5.4 **toCharArray( )**

If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**. It returns an array of characters for the entire string. It has this general form:

char[ ] toCharArray( )

ex:

String a=”hello;

char b[]=a.toCharArray();

program:

```
public class CO {
        public static void main(String[] args) {
        String a="hello";
        char c=a.charAt(1);
        System.out.println("charAt="+c);
        char ch[]=new char[2];
        a.getChars(1, 3, ch, 0);
        System.out.println(ch);
        byte b[]=a.getBytes();
        System.out.println(b);
        char ch1[]=a.toCharArray();
        System.out.println(ch1);
        }
}
```

output:

```
charAt=e
el
[B@19821f
hello
```

6. String Comparison:

The **String** class includes several methods that compare strings or substrings within strings

6.1 equals( ) and equalsIgnoreCase( ):

To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

boolean equalsIgnoreCase(String *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

program:

```
public class CO {
        public static void main(String[] args) {
        String a="hello";
        String b="Hello";
        System.out.println("equals()="+a.equals(b));
        System.out.println("equalsIgnoreCase()="+a.equalsIgnoreCase(b));
        }
}
```

output:

```
equals()=false
equalsIgnoreCase()=true
```

8.2 regionMatches( ):

The **regionMatches( )** method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

boolean regionMatches(int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)

startIndex specifies the index at which the region begins within the invoking **String** object. The **String** being compared is specified by *str2*. The index at which the comparison will start within *str2* is specified by *str2StartIndex*. The length of the substring being compared is passed in *numChars*.

boolean regionMatches(boolean *ignoreCase*, int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)

if *ignoreCase* is **true**, the case of the characters is ignored. Otherwise, case is significant.

program:

```
public class CO {
	public static void main(String[] args) {
	String a="hello";
	String b="ELL";
	System.out.println("region match="+a.regionMatches(1, b, 0, 2));
	System.out.println("region match with IgnoreCase()="+a.regionMatches(true,1, b, 0, 2));
	}
}
```

output:

```
region match=false
region match with IgnoreCase()=true
```

8.3 startsWith( ) and endsWith( ):

The **startsWith( )** method determines whether a given **String** begins with a specified string. Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:

boolean startsWith(String *str*)

boolean endsWith(String *str*)

Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned.

boolean startsWith(String *str*, int *startIndex*)

Here, *startIndex* specifies the index into the invoking string at which point the search will begin

program:

```
public class CO {
                public static void main(String[] args) {
                String a="hello";
                String b="ELL";
                System.out.println("Start with ="+a.startsWith("h"));
                System.out.println("end with="+a.endsWith("llo"));
                System.out.println("start with index ="+a.startsWith("llo",2));
        }
}
```

output:

```
Start with =true
end with=true
start with index =true
```

8.4 equals( ) Versus ==:

the **equals( )** method compares the characters inside a **String** object. The **==** operator compares two object references to see whether they refer to the same instance.

program:

```
public class CO {
                public static void main(String[] args) {
                String a="hello";
                String b="hello";
                System.out.println("equals()"+a.equals(b));
                String c=new String(a);
                System.out.println("== operator:"+(c==a));
        }}
```

output:

```
equals()true
== operator:false
```

8.6 compareTo( ):

- The **java string compareTo()** method compares the given string with current string lexicographically. It returns positive number, negative number or 0.
- It compares strings on the basis of Unicode value of each character in the strings.

If first string is lexicographically greater than second string, it returns positive number (difference of character value). If first string is less than second string lexicographically, it returns negative number and if first string is lexicographically equal to second string, it returns 0.

int compareTo(String anotherString)

program:

```
public class CO {
        public static void main(String[] args) {
                String s1="hello";
                String s2="hello";
                String s3="meklo";
                String s4="hemlo";
                String s5="flag";
                System.out.println("value is="+s1.compareTo(s2));
                System.out.println("value is="+s1.compareTo(s3));
                System.out.println("value is="+s1.compareTo(s4));
                System.out.println("value is="+s1.compareTo(s5));
                }}
```

output:

```
value is=0
value is=-5
value is=-1
value is=2
```

9. Searching Strings:

The **String** class provides two methods that allow you to search a string for a specified character or substring:

•**indexOf( )** Searches for the first occurrence of a character or substring.

•**lastIndexOf( )** Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways

To search for the first occurrence of a character, use

int indexOf(int *ch*)

To search for the last occurrence of a character, use

int lastIndexOf(int *ch*)

Here, *ch* is the character being sought.

To search for the first or last occurrence of a substring, use

int indexOf(String *str*)

int lastIndexOf(String *str*)

Here, *str* specifies the substring.

You can specify a starting point for the search using these forms:

int indexOf(int *ch*, int *startIndex*)

int lastIndexOf(int *ch*, int *startIndex*)

int indexOf(String *str*, int *startIndex*)

int lastIndexOf(String *str*, int *startIndex*)

Here, *startIndex* specifies the index at which point the search begins. For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero.

program:

```
public class CO {
        public static void main(String[] args) {
        String s1="hello hru";
        System.out.println("indexof char="+s1.indexOf('e'));
        System.out.println("indexof String="+s1.indexOf("hru"));
        System.out.println("indexof char at start index="+s1.indexOf('e',1));
        System.out.println("indexof String at start index="+s1.indexOf("hru",1));
        System.out.println("lastindexof char="+s1.lastIndexOf('e'));
        System.out.println("lastindexof string="+s1.lastIndexOf("ll"));
        System.out.println("lastindex of char at start index="+s1.lastIndexOf('e',7));
        System.out.println("lastindexof string at start="+s1.lastIndexOf("ell",7));
```

```
                    }}
```

output:

```
indexof char=1
indexof String=6
indexof char at start index=1
indexof String at start index=6
lastindexof char=1
lastindexof string=2
lastindexof char at start index=1
lastindexof string at start=1
```

10. **Modifying a String**

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use one of the following **String** methods, which will construct a new copy of the string with your modifications complete.

10.1 **substring( )**

You can extract a substring using **substring( )**. It has two forms. The first is

String substring(int *startIndex*)

Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:

String substring(int *startIndex*, int *endIndex*)

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

program:

```
public class CO {
	public static void main(String[] args) {
	String s1="hello hru";
	System.out.println("Sub string  from index position="+s1.substring(1));
	System.out.println("Sub string  from between index position="+s1.substring(1,5));
	}}
```

output:

```
Sub string  from index position=ello hru
Sub string  from between index position=ello
```

10.1 concat( )

You can concatenate two strings using **concat( )**, shown here:

String concat(String *str*)

This method creates a new object that contains the invoking string with the contents of *str* appended to the end. **concat( )** performs the same function as **+**.

program:

```
public class CO {
		public static void main(String[] args) {
		String s1="wel come to";
		String s2=" cec";
		System.out.println("Concatenation="+s1.concat(s2));
		}}
```

output:

```
Concatenation=wel come to cec
```

10.3 replace( )

The **replace( )** method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char *original*, char *replacement*)

Here, *original* specifies the character to be replaced by the character specified by *replacement*.

The resulting string is returned. For example,

String s = "Hello".replace('l', 'w');

puts the string “Hewwo” into **s**.

The second form of **replace( )** replaces one character sequence with another. It has this general form:

String replace(CharSequence *original*, CharSequence *replacement*)

String s = "Hello".replace('l', 'L');

puts the string “HeLLo” into **s**.

program:

```
public class CO {
        public static void main(String[] args) {
        String s1="wel come to cec";
         CharSequence s2="hello hru";
        System.out.println("Replace string ="+s1.replace('e', 'E'));
        System.out.println("Replace charsequence="+s1.replace('e', 'E'));
        }}
```

output:

```
Replace string =wEl comE to cEc
Replace charsequence=wEl comE to cEc
```

10.5 trim( )

The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim( )

Here is an example:

String s = "   Hello World   ".trim();

This puts the string “Hello World” into **s**.

```
public class CO {
        public static void main(String[] args) {
                String s1="       wel come to cec       ";
                 System.out.println("trim() ="+s1.trim());
```

```
                                            }}
```

output:

```
trim() =wel come to cec
```

11. Data Conversion Using valueOf( )

The **valueOf( )** method converts data from its internal format into a human-readable form.
It is a static method that is overloaded within **String** for all of Java's built-in types so that each type can be converted properly into a string. **valueOf( )** is also overloaded for type **Object**, so an object of any class type you create can also be used as an argument

static String valueOf(double *num*)

static String valueOf(long *num*)

static String valueOf(Object *ob*)

static String valueOf(char *chars*[ ])

static String valueOf(char *chars*[ ], int *startIndex*, int *numChars*)

program:

```
public class CO {
            public static void main(String[] args) {
            int a=10;
            float b=10;
            double c=10.0;
            char d='a';
            char e[]={'a','b','c'};
            System.out.println("String.valueOf(int)="+String.valueOf(a));
            System.out.println("String.valueOf(float)="+String.valueOf(b));
            System.out.println("String.valueOf(double)="+String.valueOf(c));
            System.out.println("String.valueOf(char)="+String.valueOf(d));
            System.out.println("String.valueOf(char,index,index)="+String.valueOf(e,1,2));
                                            }}
```

output:

```
String.valueOf(int)=10
String.valueOf(float)=10.0
String.valueOf(double)=10.0
String.valueOf(char)=a
String.valueOf(char,index,index)=bc
```

12. Changing the Case of Characters Within a String

The method **toLowerCase( )** converts all the characters in a string from uppercase to lower case. The **toUpperCase( )** method converts all the characters in a string from lowercase to uppercase.

String toLowerCase( )

String toUpperCase( )

Both methods return a **String** object that contains the uppercase or lowercase equivalent of the invoking **String**.

program:

```
public class CO {
            public static void main(String[] args) {
            String a="hello";
            String b="HELLO";
            System.out.println("toUpperCase()="+a.toUpperCase());
            System.out.println("toLowerCase()="+b.toLowerCase());
            }}
```

output:

```
toUpperCase()=HELLO
toLowerCase()=hello
```

12. Additional String Methods

In addition to those methods discussed earlier, **String** includes several other methods. These are summarized in the following table.

Method	**Description**
int codePointAt(int i)	Returns the Unicode code point at the index i.
int codePointBefore(int i)	Returns the Unicode code point at the index which precedes i.

int codePointCount(int start, int end)	Returns the number of code points in the portion of invoking String between start and end-1.
int offsetByCodePoints(int start, int num)	Returns the index within the invoking string that is num codepoints beyond the starting index start.
boolean contains(CharSequence str)	Returns true if the invoking object contains the String specified by str, else it returnsfalse.
boolean contentEquals(CharSequence str)	Returns true if the invoking string contains the same String as str, else it returnsfalse.
boolean contentEquals(StringBuffer str)	Returns true if the invoking string contains the same String as str, else it returnsfalse.
static String format(String fmtstr, Object... args)	Returns a String formatted as specified by fmtstr.
static String format(Locale loc, String fmtstr, Object... args)	Returns a String formatted as specified by fmtstr. Formatting is specified by Locale loc.
boolean isEmpty()	Returns true if the invoking String contains no characters and is of length zero.
String replaceFirst(String regExp, String newStr)	Returns a Stringin which first substring that matches with regExp is replaced by newStr.
String replaceAll(String regExp, String newStr)	Returns a Stringin which all the substrings that matches with regExp are replaced by newStr.
String[] split(String regExp)	Returns an String array that decomposes the invoking string into parts on encountering regular expression specified by regExp.
String[] split(String regExp, int max)	Returns an String array that decomposes the invoking string into parts on encountering regular expression specified by regExp. The number of pieces are specified by max. If the max is negative or zero, then the invoking string is fully decomposed. If the max is positive, then the last returned

	array contains the remaining of the invoking string
CharSequence subSequence(int start, int stop)	Returns a substring of the invoking string that begins at start and ends at stop.

program:

```
class CO
{
   public static void main(String arg[])
   {
              String a = "Hello How are you?";
          System.out.println("codepoint of element at index 0 : " + a.codePointAt(0));
          System.out.println("code point of element before index 1 : " + a.codePointBefore(1));
          System.out.println("Number of codePoints " + a.codePointCount(1, 4));
          System.out.println("Checks if apple contains ppl : " + a.contains("how"));
          System.out.println("ContentEquals checks  : " + a.contentEquals("hello"));
          System.out.println(" string patterns match : " + a.matches("how"));
          System.out.println("Checks if the string is empty :" + " ".isEmpty());
       System.out.println("Replaces the first with argument passed : " + a.replaceFirst("Hello", 
"hello"));
       System.out.println("ReplacesAll passed in the string : " + a.replaceAll("e", "E"));
   }
}
```

output:

```
codepoint of element at index 0 : 72
code point of element before index 1 : 72
Number of codePoints 3
Checks if apple contains ppl : false
ContentEquals checks  : false
 string patterns match : false
```

Checks if the string is empty :false

Replaces the first with argument passed : hello How are you?

ReplacesAll passed in the string : HEllo How arE you?

13. StringBuffer:

- StringBuffer class is used to create a **mutable** string object i.e its state can be changed after it is created. It represents growable and writable character sequence. As we know that String objects are immutable, so if we do a lot of changes with **String** objects, we will end up with a lot of memory leak.
- So **StringBuffer** class is used when we have to make lot of modifications to our string. It is also thread safe.

StringBuffer Constructors:

StringBuffer defines these four constructors:

StringBuffer( )

StringBuffer(int *size*)

StringBuffer(String *str*)

StringBuffer(CharSequence *chars*)

S.N.	Constructor & Description
1	**StringBuffer()** This constructs a string buffer with no characters in it and an initial capacity of 16 characters.
2	**StringBuffer(CharSequence seq)** This constructs a string buffer that contains the same characters as the specified CharSequence.
3	**StringBuffer(int capacity)** This constructs a string buffer with no characters in it and the specified initial capacity.
4	**StringBuffer(String str)** This constructs a string buffer initialized to the contents of the specified string.

13.1 length() and capacity():

- **length()** method returns the length (character count) of the sequence of characters currently represented by this object.
- **capacity()** method returns the current capacity. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur.

```
ex: StringBuffer a=new StringBuffer(“hello”);
a.length();
a.capacity()
```

program:

```
class CO
{
   public static void main(String arg[])
   {
      StringBuffer a =new StringBuffer("hello");
      System.out.println("Lenthg of stringbuffer: " + a.length());
      System.out.println("capacity of stringBuffer : " + a.capacity());
        }
}
```

output:

```
Lenthg of stringbuffer: 5
capacity of stringBuffer : 21
```

13.2 SetLength( )

To set the length of the buffer within a **StringBuffer** object, use **setLength( )**. Its general form is shown here:

void setLength(int *len*)

Here, *len* specifies the length of the buffer. This value must be nonnegative.

program:

```
class CO
{
   public static void main(String arg[])
   {
      StringBuffer a =new StringBuffer("hello");
      System.out.println("Lenthg of stringbuffer: " + a.length());
      a.setLength(10);
      System.out.println("setLength of stringBuffer : "+a.length() );
      System.out.println("capacity of stringBuffer : "+a.capacity() );
   }
}
```

output:

```
Lenthg of stringbuffer: 5
setLength of stringBuffer : 10
capacity of stringBuffer : 21
```

13.3 charAt( ) and setCharAt( ):

The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method. You can set the value of a character within a **StringBuffer** using **setCharAt( )**. Their general forms are shown here:

char charAt(int *where*)

void setCharAt(int *where*, char *ch*)

For **charAt( )**, *where* specifies the index of the character being obtained. For **setCharAt( )**, *where* specifies the index of the character being set, and *ch* specifies the new value of that character. For both methods, *where* must be nonnegative and must not specify a location beyond the end of the buffer.

program:

```
class CO
{
   public static void main(String arg[])
   {
      StringBuffer a =new StringBuffer("hello");
      System.out.println("charAt of index : "+a.charAt(1) );
      a.setCharAt(2, 'L');
      System.out.println("SetCharAt to index : "+a);
   }
}
```

output:

```
charAt of index : e
SetCharAt to index : hello
```

<u>13.4 getChars( )</u>

To copy a substring of a **StringBuffer** into an array, use the **getChars( )** method. It has this general form:

void getChars(int sourceStart, int sourceEnd, char target[ ],int targetStart)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied

is passed in *targetStart*.

program:

```
public class CO {
  public static void main(String[] args) {
    StringBuffer buff = new StringBuffer("Hello");
    char c[]=new char[buff.length()];
    buff.getChars(1, 3, c, 0);
    System.out.println(c);
  }
}
```

output:

```
el_
```

13.5 append( ):

The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions. Here are a few of its forms:

StringBuffer append(String *str*)

StringBuffer append(int *num*)

StringBuffer append(Object *obj*)

String.valueOf( ) is called for each parameter to obtain its string representation. The result is appended to the current **StringBuffer** object. The buffer itself is returned by each version of **append( )**

program:

```
public class CO {
  public static void main(String[] args) {
    StringBuffer buff = new StringBuffer("wel");
    String a="come to cec at ";
    int b=9;
    String sb=buff.append(a).append(b).toString();
    System.out.println("Append():"+sb);
  }
}
```

output:

```
Append():welcome to cec at 9
```

13.6 insert( ):

The **insert( )** method inserts one string into another. It is overloaded to accept values of all the simple types, plus **String**s, **Object**s, and **CharSequence**s. Like **append( )**, it calls **String.valueOf( )** to obtain the string representation of the value it is called with. This string is then inserted into the invoking **StringBuffer** object. These are a few of its forms:

StringBuffer insert(int *index*, String *str*)

StringBuffer insert(int *index*, char *ch*)

StringBuffer insert(int *index*, Object *obj*)

Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object.

program:

```
public class CO {
  public static void main(String[] args) {
    StringBuffer b = new StringBuffer("wel");
    String a="come";
    char c[]={'t','o'};
    System.out.println("inser(String obj) :"+b.insert(1, a));
    System.out.println("inser(char) :"+b.insert(1, c));
    System.out.println("inser(String) :"+b.insert(1, "WEL"));
  }
}
```

output:

```
inser(String obj) :wcomeel
inser(char) :wtocomeel
inser(String) :wWELtocomeel
```

13.7 reverse( ) :

We can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here:

StringBuffer reverse( )

This method returns the reversed object on which it was called

program:

```
public class CO {
  public static void main(String[] args) {
    StringBuffer b = new StringBuffer("Hello");
    System.out.println("Original String:"+b);
    System.out.println("reverse String:"+b.reverse());
  }
}
```

output:

```
Original String:Hello
reverse String:olleH
```

13.8 delete( ) and deleteCharAt( ):

You can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**. These methods are shown here:

StringBuffer delete(int *startIndex*, int *endIndex*)

StringBuffer deleteCharAt(int *loc*)

The **delete( )** method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove.

The **deleteCharAt( )** method deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object

program:

```
public class CO {
  public static void main(String[] args) {
    StringBuffer b = new StringBuffer("Hello");
    System.out.println("Original String:"+b);
    System.out.println("delet Char :"+b.deleteCharAt(1));
    System.out.println("delet Char form start index to end index :"+b.delete(1, 3));
  }
}
```

```
output:
Original String:Hello
delet Char :Hllo
delet Char form start index to end index :Ho
```

13. 9 replace( ) :

we can replace one set of characters with another set inside a **StringBuffer** object by calling **replace( )**. Its signature is shown here:

StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*–1 is replaced. The replacement string is passed in *str*. The resulting **StringBuffer** object is returned

program:

```
public class CO {
  public static void main(String[] args) {
    StringBuffer b = new StringBuffer("Hello");
    System.out.println("Original String:"+b);
    System.out.println("replace String:"+b.replace(0, 3, "HEL"));
  }
}
```

```
output:
Original String:Hello
replace String:HELlo
```

13.10 substring( ) :

we can obtain a portion of a **StringBuffer** by calling **substring( )**. It has the following two forms:

String substring(int *startIndex*)

String substring(int *startIndex*, int *endIndex*)

The first form returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object. The second form returns the substring that starts at *startIndex* and runs through *endIndex*–1

program:

```
public class CO {
  public static void main(String[] args) {
    StringBuffer b = new StringBuffer("Hello");
    System.out.println("Original String:"+b);
    System.out.println("Sub String with index:"+b.substring(1));
    System.out.println("Sub String with start index to end :"+b.substring(1, 3));
  }
}
```

output:

```
Original String:Hello
Sub String with index:ello
Sub String with start index to end :el
```

14. **Additional StringBuffer Methods:**

Method	Description
`int codePointAt(int index)`	Returns the character (Unicode code point) at the specified index.
`int codePointBefore(int index)`	Returns the character (Unicode code point) before the specified index.
`int codePointCount(int beginIndex, int endIndex)`	Returns the number of Unicode code points in the specified text range of this sequence.
`int indexOf(String str)`	Returns the index within this string of the first occurrence of the specified substring.
`int indexOf(String str, int fromIndex)`	Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
`int lastIndexOf(String str)`	Returns the index within this string of the rightmost occurrence of the specified substring.
`int lastIndexOf(String str, int fromIndex)`	Returns the index within this string of the last occurrence of the specified substring.
`int offsetByCodePoints(int index, int codePointOffset)`	Returns the index within this sequence that is offset from the given `index` by `codePointOffset` code

	points.
`StringBuffer replace(int start, int end, String str)`	Replaces the characters in a substring of this sequence with characters in the specified `String`.
`StringBuffer reverse()`	Causes this character sequence to be replaced by the reverse of the sequence.
`CharSequence subSequence(int start, int end)`	Returns a new character sequence that is a subsequence of this sequence.
`String toString()`	Returns a string representing the data in this sequence.
`void trimToSize()`	Attempts to reduce storage used for the character sequence.

Program:

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

        StringBuffer sb=new StringBuffer("Canara Enhioneering college    ");


        System.out.println("Unicode = " + sb.codePointAt(5));
        System.out.println("Length " + sb.codePointAt(5));

        System.out.println("Substring Index = " + sb.indexOf("Can"));

        System.out.println("Substring Index = " + sb.lastIndexOf("can"));

        System.out.println("Reverse = " + sb.reverse());

    }
}
```

output:

```
Unicode = 97
Length 97
Substring Index = 0
Substring Index = -1
Reverse =     egelloc gnireenoihnE aranaC
```