Module 3

Syllabus -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.

1. **What are the different types of String Constructors available in Java?**

The String class supports several constructors.

a. To create an empty String
the default constructor is used.

Ex: String s = new String();
will create an instance of String with no characters in it.

b. To create a String initialized by an array of characters, use the constructor shown here:

String(char chars[ ])

ex: char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);

This constructor initializes s with the string “abc”.

c. To specify a subrange of a character array as an initializer using the following constructor:
String(char chars[ ], int startIndex, int numChars)

Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use. Here is an example:

char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
This initializes s with the characters cde.

d. To construct a String object that contains the same character sequence as another String object using this constructor:

String(String strObj)
Here, strObj is a String object.

```
class MakeString
{ public static void main(String args[])
{ char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c);
```

```
 String s2 = new String(s1);
System.out.println(s1);
 System.out.println(s2);
}
}
```

The output from this program is as follows:

```
Java
Java
```

As you can see, s1 and s2 contain the same string.

e. To Construct string using byte array:

Even though Java's char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.

Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array.

Ex: String(byte asciiChars[ ])
String(byte asciiChars[ ], int startIndex, int numChars)

The following program illustrates these constructors:

```
class SubStringCons
{ public static void main(String args[])
{
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);
System.out.println(s1);
String s2 = new String(ascii, 2, 3);
System.out.println(s2);
}
}
```

This program generates the following output:

```
ABCDEF
CDE
```


f. To construct a String from a StringBuffer by using the constructor shown here:
Ex: String(StringBuffer strBufObj)

g. Constructing string using Unicode character set and is shown here:
String(int codePoints[ ], int startIndex, int numChars)

codePoints is an array that contains Unicode code points.

h. Constructing string that supports the new StringBuilder class.
Ex : String(StringBuilder strBuildObj)

Note:
String can be constructed by using string literals.
String s1=”Hello World”

String concatenation can be done using + operator. With other data type also.

String Length

1. The length of a string is the number of characters that it contains. To obtain this value, call the length( ) method,
2. Syntax:
   int length( )
3. Example
   char chars[] = { 'a', 'b', 'c' }; String s = new String(chars);
   System.out.println(s.length());// 3

toString()

1. Every class implements toString( ) because it is defined by Object. However, the default Implementation of toString() is sufficient.
2. For most important classes that you create, will want to override toString() and provide your own string representations.
   String toString( )
3. To implement toString( ), simply return a String object that contains the human-readable string that appropriately describes an object of your class.
4. By overriding toString() for classes that you create, you allow them to be fully integrated into Java’s programming environment. For example, they can be used in print() and println() statements and in concatenation expressions.

5. The following program demonstrates this by overriding
   toString( ) for the Box class:

```
 class Box
{
double width; double height; double depth;
 Box(double w, double h, double d)
{ width = w; height = h; depth = d; }

public String toString()
{ return "Dimensions are " + width + " by " + depth + " by " + height + "."; }
}

class toStringDemo {
public static void main(String args[])
{
 Box b = new Box(10, 12, 14);
String s = "Box b: " + b;
```

```
System.out.println(b);
System.out.println(s);
}
}
```

The output of this program is shown here:

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

Character Extraction

The String class provides a number of ways in which characters can be extracted from a String object. String object can not be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

A. charAt( )

1. description:

To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method.

2. Syntax

char charAt(int where)

Here, where is the index of the character that you want to obtain.
charAt( ) returns the character at the specified location.

3. example,

```
char ch;
 ch = "abc".charAt(1);
```

assigns the value “b” to ch.

B. getChars( )

1. to extract more than one character at a time, you can use the getChars() method.
2. Syntax

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

Here, sourceStart specifies the index of the beginning of the substring, sourceEnd specifies an index that is one past the end of the desired 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.

3.
```
class getCharsDemo {
public static void main(String args[])
{ String s = "This is a demo of the getChars method.";
int start = 10;
 int end = 14;
 char buf[] = new char[end - start];
  s.getChars(start, end, buf, 0);
```

```
        System.out.println(buf);
      }
  }
```

Here is the output of this program:

demo

C. getBytes( )

1. This method is called getBytes( ), and it uses the default character-to-byte conversions provided by the platform.

   Syntax:

   byte[ ] getBytes( )

   Other forms of getBytes( ) are also available.

2. getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

D. 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( )

2. String Comparison:

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

equals( )

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. The comparison is case-sensitive.

A. equalsIgnoreCase( )

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.

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
 public static void main(String args[]) {
```

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
s1.equalsIgnoreCase(s4)); } }
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

B. regionMatches( )

1. 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.
2. Syntax:

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

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

3. For both versions, 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 str2 StartIndex. The length of the substring being compared is passed in numChars.

4. In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

C. startsWith( ) and endsWith( )

1. The startsWith( ) method determines whether a given String begins with a specified string.
2. endsWith( ) determines whether the String in question ends with a specified string.
3. Syntax

```
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.
For example,
"Foobar".endsWith("bar")

"Foobar".startsWith("Foo")
are both true.

4. A second form of startsWith( ), shown here, lets you specify a starting point:

boolean startsWith(String str, int startIndex)

Here, startIndex specifies the index into the invoking string at which point the search will begin. For example,
"Foobar".startsWith("bar", 3)
returns true.

D. **equals( ) Versus ==**

It is important to understand that the equals( ) method and the == operator perform two different operations.
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.


```
class EqualsNotEqualTo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
}
}
```

E. **compareTo( )**

1. Sorting applications, you need to know which is less than, equal to, or greater than the next.
2. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The String method compareTo() serves this purpose.
3. It has this general form:

int compareTo(String str)

Here, str is the String being compared with the invoking String. The result of the comparison is returned and is interpreted,

4. Less than zero when invoking string is less than str.
5. Greater than zero when invoking string is greater than str.
6. Zero The two strings are equal.

```
// A bubble sort for Strings.
class SortString
{ static String arr[] = { "Now", "is", "the", "time", "for", "all", "good", "men",
"to", "come", "to", "the", "aid", "of", "their", "country" };
 public static void main(String args[])
 { for(int j = 0; j < arr.length; j++)
{ for(int i = j + 1; i < arr.length; i++)
 { if(arr[i].compareTo(arr[j]) < 0)
 { String t = arr[j];
arr[j] = arr[i];
 arr[i] = t;
}
} System.out.println(arr[j]);
}
}
}
```

The output of this program is the list of words:

Now aid all come country for good is men of the the their time to to
As you can see

7. Ignore case differences when comparing two strings, use compareToIgnoreCase( ), This method returns the same results as compareTo(), except that case differences are ignored.

5. Searching String

A. indexOf( ) and lastIndexOf( )

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

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

3. These two methods are overloaded in several different ways

4. return the index at which the character or substring was found, or –1 on failure.

5. To search for the first occurrence of a character, int indexOf(int ch)
6. To search for the last occurrence of a character, int lastIndexOf(int ch) Here, ch is the character being sought

7. 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.

8. You can specify a starting point for the search using these forms:
   int indexOf(int ch, int startIndex)
   int lastIndexOf(int ch, int startIndex)

9. int indexOf(String str, int startIndex) int lastIndexOf(String str, int startIndex) Here, startIndex specifies the index at which point the search begins.
10. For indexOf( ), the search runs from startIndex to the end of the string. For lastIndexOf( ), the search runs from startIndex to zero. The following example shows how to use the various index methods to search inside of Strings:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
 public static void main(String args[])
{ String s = "Now is the time for all good men " + "to come to the aid
of their country.";
System.out.println(s);
System.out.println("indexOf(t) = " + s.indexOf('t'));
System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t'));
System.out.println("indexOf(the) = " + s.indexOf("the"));
System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " + s.indexOf('t', 10));
System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " + s.lastIndexOf("the",
60));
}
}
```

Output

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
 indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
 lastIndexOf(the, 60) = 55
```

6. Modifying a String

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.

A. Substring( )

1. You can extract a substring using substring( ). It has two forms. The first is String substring(int startIndex)
2. 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.
3. 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.
4. The string returned contains all the characters from the beginning index, up to, but not including, the ending index. The following program uses substring( ) to replace all instances of one substring with another within a string:


```
// Substring replacement.
class StringReplace {
public static void main(String args[]) {
String org = "This is a test. This is, too.";
String search = "is";
String sub = "was";
String result = "";
int i;
do  {
System.out.println(org);
i = org.indexOf(search);
 if(i != -1) { result = org.substring(0, i);
result = result + sub;
result = result + org.substring(i + search.length());
 org = result;
} } while(i != -1);
}
}
```

The output from this program is shown here:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
 Thwas was a test. Thwas is, too.
 Thwas was a test. Thwas was, too.
```

B. concat( )

1. concatenate two strings using concat( )

String concat(String str)

2. This method creates a new object that contains the invoking string with the contents of str appended to the end.
3. concat( ) performs the same function as +.
4. String s1 = "one";
   String s2 = s1.concat("two");

C. replace( )

1. The replace( ) method has two forms.
2. The first replaces all occurrences of one character in the invoking string with another character.

Syntax:

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.

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)

D. trim( )

The trim() method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

Syntax:

String trim( )

Example:

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

This puts the string "Hello World" into s.

The trim( ) method is quite useful when you process user commands.

```
// Using trim() to process commands.
import java.io.*;
class UseTrim
{ public static void main(String args[]) throws IOException {
BufferedReader br = new BufferedReader(new
nputStreamReader(System.in));
String str;
System.out.println("Enter 'stop' to quit.");
System.out.println("Enter State: ");
do { str = br.readLine();
```

```
 str = str.trim();
if(str.equals("Illinois"))
System.out.println("Capital is Springfield.");
 else if(str.equals("Missouri"))
System.out.println("Capital is Jefferson City.");
else if(str.equals("California"))
System.out.println("Capital is Sacramento.");
else if(str.equals("Washington"))
 System.out.println("Capital is Olympia."); // ... }
while(!str.equals("stop"));
}
}
```

5. Data Conversion

1. The valueOf( ) method converts data from its internal format into a human-readable form.
2. 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.

3. valueOf( ) is also overloaded for type Object, so an object of any class type you create can also be used as an argument

Syntax:

```
static String valueOf(double num)
static String valueOf(long num)
 static String valueOf(Object ob)
static String valueOf(char chars[ ])
```

4. valueOf( ) is called when a string representation of some other type of data is needed. example, during concatenation operations.

5. Any object that you pass to valueOf( ) will return the result of a call to the object's toString( ) method.

6. There is a special version of valueOf() that allows you tospecify a subset of a char array.

Syntax:

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

7. Here, chars is the array that holds the characters, startIndex is the index into the array of characters at which the desired substring begins, and numChars specifies the length of the substring.

6. Changing Case of Characters

A. toLowerCase( )

1. converts all the characters in a string from uppercase to lowercase.

2. This method return a String object that contains the lowercase equivalent of the invoking String.

3. Non alphabetical characters, such as digits, are unaffected.

Syntax

String toLowerCase( )

B. toUpperCase()

1. converts all the characters in a string from lowercase to uppercase.
2. This method return a String object that contains the uppercase equivalent of the invoking String.

3. Non alphabetical characters, such as digits, are unaffected.

Syntax

String toUpperCase( )


```
class ChangeCase {
 public static void main(String args[]) {
String s = "This is a test.";
System.out.println("Original: " + s);
String upper = s.toUpperCase();
String lower = s.toLowerCase();
System.out.println("Uppercase: " + upper);
System.out.println("Lowercase: " + lower);
}
}
```

Output:

Original: This is a test.

Uppercase: THIS IS A TEST.

Lowercase: this is a test.

StringBuffer

StringBuffer is a peer class of String that provides much of the functionality of strings. As you know, String represents fixed-length, immutable character sequences.

StringBuffer represents growable and writeable character sequences.

StringBuffer may have characters and substrings inserted in the middle or appended to the end.

StringBuffer will automatically grow to make room for such additions and often has more characters pre allocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer defines these four constructors:

StringBuffer( )

StringBuffer(int size)

StringBuffer(String str)

StringBuffer(CharSequence chars)

a. The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
b. The second version accepts an integer argument that explicitly sets the size of the buffer.
c. The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

d. StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time.

A. length( ) and capacity( )

a. The current length of a StringBuffer can be found via the length( ) method, while the total allocated capacity can be found through the capacity( ) method.

Syntax

int length( )

int capacity( )

b. Example:

```
class StringBufferDemo
{
  public static void main(String args[])
  {
     StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer = " + sb);
   System.out.println("length = " + sb.length());
  System.out.println("capacity = " + sb.capacity());
}
```

}

Output

buffer = Hello

length = 5

capacity = 21

B. ensureCapacity( )

a. If you want to pre allocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity( ) to set the size of the buffer.
b. This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer.
   Syntax

   void ensureCapacity(int capacity)

   Here, capacity specifies the size of the buffer.

C. setLength( )

a. To set the length of the buffer with in a StringBufferobject,

Syntax:

void setLength(int len)

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

When you increase the size of the buffer, null characters are added to the end of the existing buffer.

If you call setLength( ) with a value less than the current value returned by length(), then the characters stored beyond the new length will be lost.

D. charAt( ) and setCharAt( )

a. 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( ).
b. Syntax

   char charAt(int where)

   void setCharAt(int where, char ch)

c. For charAt( ), where specifies the index of the character being obtained.
d. For setCharAt( ), where specifies the index of the character being set, and ch specifies the new value of that character.

```
// Demonstrate charAt() and setCharAt().
 class setCharAtDemo {
public static void main(String args[])
 { StringBuffer sb = new StringBuffer("Hello");
 System.out.println("buffer before = " + sb);
 System.out.println("charAt(1) before = " + sb.charAt(1));
sb.setCharAt(1, 'i');
sb.setLength(2);
System.out.println("buffer after = " + sb);
System.out.println("charAt(1) after = " + sb.charAt(1)); } }
```

Output

```
buffer before = Hello
charAt(1) before = e
 buffer after = Hi
 charAt(1) after = i
```


E. getChars( )

a. To copy a substring of a StringBuffer into an array, use the getChars( ) method.

Syntax

Syntax

```
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.

b. This means that the substring contains the characters from sourceStart through sourceEnd–1.

c. The array that will receive the characters is specified by target.

The index within target which the substring will be copied is passed in targetStart.

d. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

F. append( )

1. 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)

2. The result is appended to the current StringBuffer object.
3. The buffer itself is returned by each version of append( ).
4. This allows subsequent calls to be chained together, as shown in the following example:

```
class appendDemo {
 public static void main(String args[])
 { String s; int a = 42;
  StringBuffer sb = new StringBuffer(40);
  s = sb.append("a = ").append(a).append("!").toString();
  System.out.println(s);
 }
}
```

Output

a = 42!

G. **insert( )**

1. The insert() method inserts one string in to another.
2. It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences.
3. Like append(),it calls String.valueOf() to obtain the string representation of the value it is called with.
4. This string is then inserted into the invoking StringBuffer object.
5. 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.

6. The following sample program inserts “like” between “I” and “Java”:

```
class insertDemo { public static void main(String args[]) {
  StringBuffer sb = new StringBuffer("I Java!");
 sb.insert(2, "like ");
```

```
        System.out.println(sb);
    }
}
```

7. Output

   I like Java!

H. reverse( )

You 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.

The following program demonstrates reverse( )

```
class ReverseDemo {
public static void main(String args[])
{ StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
s.reverse();
System.out.println(s);
}
}
```


Output

abcdef

fedcba

I. delete( ) and deleteCharAt( )

You can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ).

Syntax:

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.

Thus, the substring deleted runs from startIndex to endIndex–1. The resulting StringBuffer object is returned.

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

```
// Demonstrate delete() and deleteCharAt()

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

{ StringBuffer sb = new StringBuffer("This is a test.");

sb.delete(4, 7);

System.out.println("After delete: " + sb);

sb.deleteCharAt(0);

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

}

}
```

Output

After delete: This a test.

After deleteCharAt: his a test.

J. **replace( )**

a. You can replace one set of characters with another set inside a StringBuffer object by calling replace( ).

b. Syntax

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

The substring being replaced is specified by the indexes startIndex and endIndex.

c. Thus, the substringatstartIndexthroughendIndex–1 is replaced.The replacement string is passed in str.

The resulting StringBuffer object is returned.

```
class replaceDemo {

public static void main(String args[])

{ StringBuffer sb = new StringBuffer("This is a test.");

sb.replace(5, 7, "was");

System.out.println("After replace: " + sb);

}
```

}

Here is the output:

After replace: This was a test.

K. **substring( )**

1. It has the following two forms:
   Syntax
   String substring(int startIndex)

   String substring(int startIndex, int endIndex)

2. The first form returns the substring that starts at startIndex and runs to the end of the invoking StringBuffer object.

3.The second form returns the substring that starts at startIndex and runs through endIndex–1.

These methods work just like those defined for String that were described earlier.

Difference between StringBuffer and StringBuilder.

1. J2SE 5 adds a new string class to Java's already powerful string handling capabilities. This new class is called StringBuilder.
2. It is identical to StringBuffer except for one important difference: it is not synchronized, which means that it is not thread-safe.
3. The advantage of StringBuilder is faster performance. However, in cases in which you are using multithreading, you must use StringBuffer rather than StringBuilder.

Additional Methods in String which was included in Java 5

1. **int codePointAt(int i)**

Returns the Unicode code point at the location specified by i.

2. **int codePointBefore(int i)**
   Returns the Unicode code point at the location that precedes that specified by i.
3. **int codePointCount(int start , int end)**
   Returns the number of code points in the portion of the invoking String that are between start and end– 1.
4. **boolean contains(CharSequence str)**
   Returns true if the invoking object contains the string specified by str . Returns false, otherwise.
5. **boolean contentEquals(CharSequence str)**
   Returns true if the invoking string contains the same string as str. Otherwise, returns false.
6. **boolean contentEquals(StringBuffer str)**
   Returns true if the invoking string contains the same string as str. Otherwise, returns false.
7. **static String format(String fmtstr , Object ... args)**

Returns a string formatted as specified by fmtstr.

8. **static String format(Locale loc , String fmtstr , Object ... args)**
   Returns a string formatted as specified by fmtstr.
9. **boolean matches(string regExp)**
   Returns true if the invoking string matches the regular expression passed in regExp. Otherwise, returns false.
10. **int offsetByCodePoints(int start , int num)**
    Returns the index with the invoking string that is num code points beyond the starting index specified by start.
11. **String replaceFirst(String regExp , String newStr)**
    Returns a string in which the first substring that matches the regular expression specified by regExp is replaced by newStr.
12. **String replaceAll(String regExp , String newStr)**
    Returns a string in which all substrings that match the regular expression specified by regExp are replaced by newStr
13. **String[ ] split(String regExp)**
    Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp.
14. **String[ ] split(String regExp , int max)**
    Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp. The number of pieces is specified by max. If max is negative, then the invoking string is fully decomposed. Otherwise, if max contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If max is zero, the invoking string is fully decomposed.
15. **CharSequence subSequence(int startIndex , int stopIndex)**
    Returns a substring of the invoking string, beginning at startIndex and stopping at stopIndex . This method is required by the CharSequence interface, which is now implemented by String.

Additional Methods in StringBuffer which was included in Java 5

StringBuffer appendCodePoint(int ch)
Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.

int codePointAt(int i)
Returns the Unicode code point at the location specified by i.

int codePointBefore(int i)
Returns the Unicode code point at the location that precedes that specified by i.

int codePointCount(int start , int end)
Returns the number of code points in the portion of the invoking String that are between start and end– 1.

int indexOf(String str)
Searches the invoking StringBuffer for the first occurrence of str. Returns the index of the match, or –1 if no match is found.

int indexOf(String str , int startIndex)
Searches the invoking StringBuffer for the first occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.

int lastIndexOf(String str)
Searches the invoking StringBuffer for the last occurrence of str. Returns the index of the match, or –1 if no match is found.

int lastIndexOf(String str , int startIndex)
Searches the invoking StringBuffer for the last occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.