STRINGS IN JAVA

STRING AS AN OBJECT

- When you create a String object, you are creating a string that cannot be changed.
- That is, once a String object has been created, you cannot change the characters that comprise that string.
- Strings are Immutable.
- Each time you need an altered version of an existing string, a new String object is created that contains the modifications.

String Handling:syllabus

- String Constructor,
- String length,
- Special string Operations,
- Character Extraction,
- String comparison,
- Modifying a string,
- String Buffer

Syllabus(contd..)

- **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() & equalignorecase(), regionmatches(),startswith() and endswith(),Equal() v/s==,Compareto()

Syllabus(contd..)

- Modifying a String
- substring()
- concat()
- replace()
- trim()

Syllabus(contd..) String Buffer

- String buffer constructors
- Length() and capacity()
- Ensurecapacity()
- Setlength()
- Charat() and setcharat()
- Getchars()
- Append()
- Insert(), reverse(),delete(),deletecharat()
- Replace(),substring()

STRING AS AN OBJECT

- Java provides two options: StringBuffer and StringBuilder. Both hold strings that can be modified after they are created.
- The String , StringBuffer , and StringBuilder classes are defined in java.lang. Thus, they are available to all programs automatically.

STRING CONSTRUCTORS

- The String class supports several constructors.
- **String s = new String();**
- **String(char chars[ ]);**

Eg:char chars[] = { 'a', 'b', 'c' };

String s = new String(chars);

You can specify a subrange of a character array as an initializer

STRING CONSTRUCTORS

- **String(char chars[ ], int startIndex, int numCharS)**
- 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

- You can construct a String object that contains the same character sequence as another String object using this constructor:
  String(String strObj);

- // Construct one String from another.
- 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); } }

output

- Java
- Java

STRING CONSTRUCTORS

- String class provides constructors that initialize a string when given a bytearray.
- Their forms are shown here:
- **String(byte asciiChars[ ]);**
- **String(byte asciiChars[ ], int startIndex, int numChars);**
- Here,asciiChars specifies the array of bytes. The second form allows you to specify a subrange

STRING CONSTRUCTORS

- Eg Program
- / Construct string from subset of char array.
- 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

STRING LENGTH AND STRING LITERAL

The length( ) method, shown here:

int length( );

Eg1: char chars[] = { 'a', 'b', 'c' };

String s = new String(chars);

System.out.println(s.length())

Eg2:

char chars[] = { 'a', 'b', 'c' };

String s1 = new String(chars);

String s2 = "abc"; // use string literal

STRING LITERAL

- You can call methods directly on a quoted string as if it were an object reference.
- **Eg3 System.out.println("abc".length());**
- STRING CONCATENATION
- The + operator is used to concatenate two strings_to produce a String Object as the Result.

STRING CONCATENATION

- Eg: String age = "9";

```
String s = "He is " + age + " years old.";
System.out.println(s);
```

String Concatenation with Other Data Types

Eg:

```
int age = 9;
    String s = "He is " + age + " years old.";
    System.out.println(s);
```

Output :He is 9 years old.

STRING CONCATENATION

- String s = "four: " + 2 + 2;
- System.out.println(s);
- This fragment displays
- four: 22
- rather than the
- four: 4
- String s = "four: " + (2 + 2);
- Now s contains the string “four: 4”.

STRING CONVERSION toString()

- Every class implements **toString( )** because it is defined by Object.
- We can override toString() Method in classes and provide our own representation.

. General Method of toString() representation is given by.

String toString();

STRING CONVERSION toString()

- // Override toString() for 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 + ".";
 }
 }
```

STRING CONVERSION toString()

- class toStringDemo {

```
public static void main(String args[]) {
Box b = new Box(10, 12, 14);
String s = "Box b: " + b; // concatenate Box object
System.out.println(b); // convert Box to string
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

- **charAt( ).**
- To extract a single character from a String , you can refer directly to an individual character via the charAt() method.
- **char charAt(int where);**
- The general form is given above.
- Eg.
- char ch;
- ch = "abc".charAt(1);
- assigns the value “b” to ch.

getChars( )

- To extract more than one character at a time, you can use the getChars( ) method.
- General Form
- **void getChars(int sourceStart , int sourceEnd, char target [ ], int targetStart);**
- **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.

getChars( )

- 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.
- **String s = "This is a demo of the getChars method.";**
- int start = 10;// count from index 0
- int end = 14;//count from 1
- char buf[] = new char[end - start];
- s.getChars(start, end, buf, 0);
- System.out.println(buf); Output:demo

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( );
- **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( )

equals( ) and equalsIgnoreCase( )

- **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.
- **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 but ignoring cases.

- // 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

Hello equals Hello -> true

Hello equals Good-bye -> false

Hello equals HELLO -> false

Hello equalsIgnoreCase HELLO -> true

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

Example for first form

```
String str1="Welcome to Java";
 String otherstr="Java";
Boolean
ismatch=str1.regionMatches(11,otherstr,0,4);
if(ismatch)
     System.out.println("substring is matched");
else
  System.out.println("substring is not matched");
```

Output for first form

- substring is matched

Example for second form

```
String str1="Welcome to Java";
 String otherstr=“java";
Boolean
ismatch=str1.regionMatches(true,11,otherstr,0,4);
if(ismatch)
     System.out.println("substring is matched");
else
   System.out.println("substring is not matched");
```

Output for second form

- substring is matched

- 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 str2StartIndex.
- The length of the substring being compared is passed in numChars.
- In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant

startsWith( ) and endsWith( )

- The **startsWith( )** method determines whether a givenString begins with a specified string.
- **endsWith( )** determines whether the String in question ends with a specified string.
- **boolean startsWith(String str);**
- **boolean endsWith(String str);**

For example,

"Foobar".endsWith("bar");

and

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

- **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.

- **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.
- The following program shows how two different String objects can contain the same characters, but references to these objects will not compare as equal:

- // equals() vs ==

```
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));
  }
}
```

output

```
Hello equals Hello -> true
Hello == Hello -> false
```

- We need to know which String is equal to which string or less than or greater than another string.
- For this we use **compareTo()** method
- **General Syntax of compareTo() is**
- **int compareTo(String str);**
- 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.

Value	Meaning
Less than zero	The invoking string is less than *str*.
Greater than zero	The invoking string is greater than *str*.
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

- If you want to ignore case differences when comparing two strings, use
- **compareToIgnoreCase( )**, as shown here:

  int compareToIgnoreCase(String str);

  This method returns the same results as compareTo( ) , except that case differences are ignored.

You might want to try substituting it into the previous program. After doing so, “Now” will no longer be first.

Searching Strings

- indexOf( ) Searches for the first occurrence of a character or substring.
- lastIndexOf( ) Searches for the last occurrence of a character or substring.

In all cases, the methods return the index at which the character or substring was found, or –1 on failure.

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.

Modifying a String

- substring( )
- You can extract a substring using substring( ). It has two forms. The first is
- **String substring(int startIndex);**
- This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.
- **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.

```
// 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 { // replace all matching substrings
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.

- **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.
- String s1 = "one";
- String s2 = s1.concat("two");
- System.out.println(“The string is"+" " +s2);
- Output: The string is onetwo
- **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.
- **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.

String Buffer

- StringBuffer is a peer class of String that provides much of the functionality of strings.
- 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 preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

- **StringBuffer Constructors**
- StringBuffer defines these four constructors
- **StringBuffer( );**
- **StringBuffer(int size);**
- **StringBuffer(String str);**
- **StringBuffer(CharSequence chars);**
- The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
- The second version accepts an integer argument that explicitly sets the size of the buffer.
- 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.

- StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time.
- By allocating room for a few extra characters, StringBuffer reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in chars.
- **length( ) and capacity( )**
- 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.

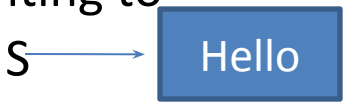
  They have the following

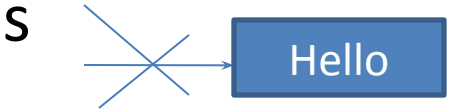
  general forms: **int length( ); int capacity( );**

- Here is an example:
- // StringBuffer length vs. capacity.
- 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());
- }
- }
- Here is the output of this program, which shows how StringBuffer reserves extra space
- for additional manipulations:
- buffer = Hello
- length = 5
- capacity = 21
- Since sbis initialized with the string “Hello” when it is created, its length is 5. Its capacity is 21
- because room for 16 additional characters is automatically added.

STRINGS vs STRINGBUFFER

- Strings :They are Immutable cannot be changed.
- Once they are created their values cannot be changed,you cannot add or delete a character to the same string.
- For Eg String s="hello";
- s=s+"world";
- earlier s was pointing to Hello i,e s
- when concatenated with world now s is pointing to





The unassigned Reference is proclaimed by and returned to Garbage Collector.

- When to Use String,StringBuffer classes
- 1. If the content is fixed and wont change frequently then we should go for string.
- 2.If the content is not fixed and keep on changing but thread safety is required then we go for String Buffer.
- String Buffer class is mutable the value can be manipulated.The Reference is not changed.

```
class stringsample{

    public static void main(String args[])
  {

      String s1="Hello";
   s1=s1+”World”;
        System.out.println("THE Value of s1 is"+s1);


         StringBuffer s2=new StringBuffer("Hello from BMSCE");

            System.out.println("THE Value of s2 is"+s2);
         s2.append("Enginnering");
         System.out.println("THE Value of s2 is"+s2);
  }
}
```

ensureCapacity()

- If you want to preallocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity( ) to set the size of the buffer.
- This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer.
- **ensureCapacity( ) has this general form:**
- **void ensureCapacity(int minCapacity) ;**
- Here, minCapacity specifies the minimum size of the buffer.

setLength( )

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

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( ) .
- **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.

- // 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));
- }
- }
- Here is the output generated by this program:
- buffer before = Hello
- charAt(1) before = e
- buffer after = Hi
- charAt(1) after = i

- **getChars( )**
- To copy a substring of a StringBuffer into an array, use the getChars( ) method.
- 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.
- This means that the substring contains the characters from sourceStart through sourceEnd–1
- .

```
class   getscharstring
{
        public static void main(string [] args){
  StringBuffer b=new StringBuffer("Hello from Bms College ");
char arr[]=new char[5];
//b.getChars(0,5,arr,0) ;
    // b.getChars(6,10,arr,0);
     b.getChars(6,8,arr,2);
    for(int i=0;i<arr.length;i++)
    {
      System.out.println(arr[i]);
     }

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

.

- 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

- **append( )**

The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object.

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( ).

- // Demonstrate append().

```
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);
 }
}
```

The output of this example is shown here:

a = 42!

The append( ) method is most often called when the + operator is used on String objects.

- **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 Char Sequences.

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.

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.

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

The output of this example is shown here:

I like Java!

- **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.
- // Using reverse() to reverse a StringBuffer.
- class ReverseDemo {
- public static void main(String args[]) {
- StringBuffer s = new StringBuffer("abcdef");
- System.out.println(s);
- s.reverse();
- System.out.println(s);
- }
- }
- Here is the output produced by the program
- abcdef
- fedcba

- delete( ) and deleteCharAt( )
- You can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ).
- **StringBuffer delete(int startIndex, int endIndex);**
- **StringBuffer deleteCharAt(int loc);**

  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);
- }
- }
- The following output is produced:
- After delete: is a test.
- After deleteCharAt: his a test.

- replace( );

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

```
/ Demonstrate replace()
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.

Features	String	String buffer	String Builder
Storage	Heap, String Constant Pool	Heap Area	Heap Area
Objects	immutable	Mutable	Mutable
Memory	More	Consumes less memory	Consumes less memory
Thread safe	not thread safe	All synchronized ,thread safe	Non synchronized, not thread safe
Performance	slow	Faster than string	Faster than string buffer