CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java** (Part 2)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- ☑ Java Library
  - ☑ **String Handling**
    - ☑ Character Extraction
    - ☑String Comparison
    - ☑Searching Strings
    - ☑Modifying Strings
    - ☑Using valueOf( )
    - ☑Comparison of String Buffer and String.

Character Extraction

- The String class provides the following methods through which characters can be extracted from a String object.

  charAt( )

  getChars( )

  getBytes( )

  toCharArray( )

Character Extraction(contd.)

charAt( )

- Used to **extract a single character** from a String
- we can refer directly to an individual character via the charAt( ) method.
- General form:

char **charAt**(int *where)*

Here *where* is the index of the character that you want to obtain.

e.g.

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

This assigns the value "**b**" to variable **ch.**

	abc
index	**012**

Character Extraction(contd.)

getChars( )

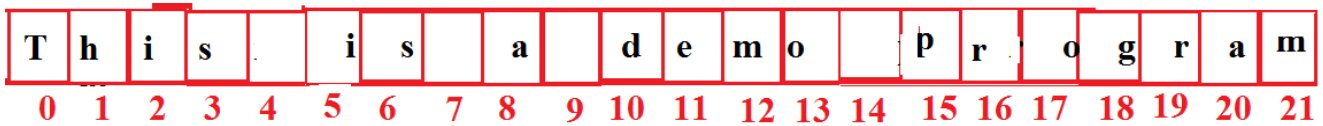
- Used to **extract more than one character** at a time.
- General form:

```
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 up to which character need to be extracted.
  - (the extracted substring contains the characters from *sourceStart* through *sourceEnd–1.)*
- This extracted substring is stored at ***target*** *array at location targetStart*.

Example program- getChars( )

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo program";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
}
}
```



This program will extract characters in string **s** from index 10 to 14-1(13) and store in character array **buf** and prints it.

OUTPUT
demo

Character Extraction(contd.)

getBytes( )

- Used to **extract the characters in an array of bytes.**
  - it uses the default character-to-byte conversions.
  - General form

```
byte[ ] getBytes( )
```

  - Most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

Character Extraction(contd.)

toCharArray( )

- Used **to convert all the characters in a String object into a character array.**
  - It returns an array of characters for the entire string.
- General form:

```
char[ ] toCharArray( )
```

Example program -toCharArray( )

```
public class CharArrayEg{
   public static void main(String args[]){
      String str = new String("Welcome to OOP");
      char[] a= str.toCharArray();
      System.out.print("Content of a is:");
      for(char c: a){
         System.out.print(c);
      }
   }
}
```

OUTPUT
Content of a is: Welcome to OOP

String Comparison

- The String class includes several methods that compare strings or substrings within strings.
- **equals( )**
- **equalsIgnoreCase( )**
- **regionMatches( )**
- **startsWith( )**
- **endsWith( )**
- **equals( ) Versus ==**
- **compareTo( )**

String Comparison(contd.)

equals( )

- To compare two strings for equality, use **equals( )**
- General form:

boolean **equals**(Object *str*)

  - Here, String object *str* is 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.

String Comparison(contd.)

equalsIgnoreCase( )

- This perform a comparison that **ignores case differences**(not case sensitive)
- When it compares two strings, it considers A-Z to be the same as a-z.
- General form:

boolean **equalsIgnoreCase**(String *str)*

String Comparison(contd.)

```
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 + " is " + s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " is " + s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " is " + s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " is " +s1.equalsIgnoreCase(s4));
}
}
```

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

String Comparison(contd.)

regionMatches( )

- The **regionMatches( )** method compares a specific region inside a string with another specific region in another string.

General forms:

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

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

- *startIndex* specifies the index at which the region begins within the invoking **String.** The String to be 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.

String Comparison(contd.)

- **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.

General forms:

boolean **startsWith**(String *str)*

boolean **endsWith**(String *str)*

System.out.println(“Football".endsWith("ball"));

This prints **true**. (because *bar* comes at the end of string *Football*

System.out.println(" Football ".startsWith("Foo"));

This prints **true**. (because *Foo* comes at the beginning of string *Football*

String Comparison(contd.)

- A second form of startsWith( ), 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,

System.println("Football".startsWith("ball", 4));

  - This prints **true**.

equals( ) Versus ==

- **equals**( ) method and the == operator perform two ***different operations***.
- the **equal**s( ) 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 + " is " + s1.equals(s2));
System.out.println(s1 + " = = " + s2 + " is " + (s1 ==s2));
}
}
```

OUTPUT

Hello equals Hello is true

Hello = = Hello is false

compareTo( )

- A string is **less than** another if it **comes before** the other in dictionary order.
  - E.g. “ant”<“bat” (ant comes beforr bat in dictionary)
- A string is **greater than** another if it **comes after** the other in dictionary order.
  - E.g. “bat”>“ant” (bat comes after ant in dictionary)
- **compareTo( )** method in String is used for comparing two strings. General form:

```
int compareTo(String str)
```

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

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

The uppercase letter has a lower value in the ASCII character set than lowercase letters.

Example program - compareTo( )


```
class CompareToEg
{
public static void main(String args[]) {
    String s1="ant";
    String s2="bat";
    if(s1.compareTo(s2) < 0)
    {System.out.println(s1 + " comes before "+s2);
    }
    else if(s1.compareTo(s2) > 0)
    {System.out.println(s1 + " comes after"+s2);
    }
    else
    System.out.println(s1 + " is same as "+s2);
}
}
```

OUTPUT
ant comes before bat

Bubblesort to sort strings


```
class SortString {
static String arr[] = {"This", "is", "best", "time", "for", "all"};
public static void main(String args[]) {
for(int i = 0; i < arr.length; i++)
{
    for(int j = i + 1; j < arr.length; j++)
     {
          if(arr[j].compareTo(arr[i]) < 0)
          {
          String temp = arr[i];
          arr[i] = arr[j];
          arr[j] = temp;
          }
    }
System.out.println(arr[i]);
}
}
}
```

OUTPUT

This
all
best
for
is
time

compareToIgnoreCase( )

- **compareToIgnoreCase( )** method is not case sensitive.

int **compareToIgnoreCase**(String *str)

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

- .

Using **compareToIgnoreCase**

```
class CompareToIgnoreEg{
public static void main(String args[]) {
    String s1="ant";
    String s2="Hat";
    if(s1.compareToIgnoreCase(s2) < 0)
    {System.out.println(s1 + " is before "+s2);
    }
    else if(s1.compareToIgnoreCase(s2) > 0)
    {System.out.println(s1 + " is after"+s2);
    }
    else
    System.out.println(s1 + " is same as "+s2);

}
}
```

OUTPUT

ant is before Hat

Using **compareTo**

```
class CompareToEg{
public static void main(String args[]) {
    String s1="ant";
    String s2="Hat";
    if(s1.compareTo (s2) < 0)
    {System.out.println(s1 + " is before "+s2);
    }
    else if(s1.compareToI (s2) > 0)
    {System.out.println(s1 + " is after"+s2);
    }
    else
    System.out.println(s1 + " is  same as "+s2);
    }
}
```

OUTPUT

ant is after Hat

Searching Strings

- The **String** class provides two methods 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.
  - In all cases, the methods return the index at which the character or substring was **found**. If the character or substring is **not found** then these method returns -1.

Searching Strings(contd.)

- 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 searched.*

- 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*.

Searching Strings(contd.)

- We can specify a **starting point for the search** using :

  int indexOf(char *ch, int startIndex)*

  int lastIndexOf(char *ch, int startIndex)*

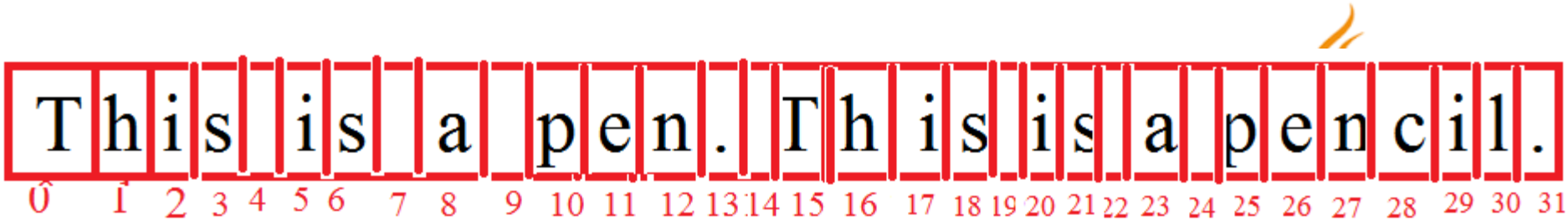
  int indexOf(String *str, int startIndex)*

  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.*

Searching Strings(contd.)


```
class indexOfDemo {
public static void main(String args[]) {
String s = “This is a pen. This is a pencil.";
System.out.println(s);
System.out.println("indexOf(i) = " +s.indexOf('i'));
System.out.println("lastIndexOf(i) = " +s.lastIndexOf('i'));
System.out.println("indexOf(This) = " +s.indexOf("This"));
System.out.println("lastIndexOf(This) = " +s.lastIndexOf("This"));
System.out.println("indexOf(i, 10) = " +s.indexOf('i', 10));
System.out.println("lastIndexOf(i, 23) = " + s.lastIndexOf('i', 23));
System.out.println("indexOf(This, 10) = " + s.indexOf("This", 10));
System.out.println("lastIndexOf(This, 13) = " + s.lastIndexOf("This", 13));
}
}
```



This is a pen. This is a pencil.

indexOf(i) = 2

lastIndexOf(i) = 29

indexOf(This) = 0

lastIndexOf(This) = 15

indexOf(i, 10) = 17

lastIndexOf(i, 23) = 20

indexOf(This, 10) = 15

lastIndexOf(This, 13) = 0

Modifying a String

- **String objects are immutable(cannot change a string.)**
- To modify a String, we must either
  - copy it into a StringBuffer or StringBuilder, or
  - use one of the following String methods:

    substring( )

    concat( )

    replace()

    trim( )

Modifying a String(contd.)

substring( ).

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

```
String substring(int startIndex)
```

    - Here, startIndex **specifies <u>the index at which the substring will begin</u>**. This form <u>returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.</u>
  - The second form of **substring( )** allows to specify both the beginning and ending index of the substring:

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

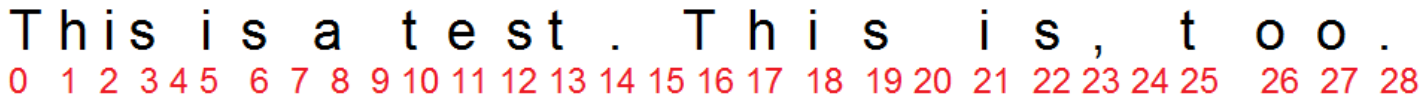
    - Here, *startIndex* specifies the beginning index, and endIndex specifies the stopping point.
    - The string returned contains all the characters from the beginning index, up to, but **not including, the ending index.**

Modifying a String(contd.)

```
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);
}
```

OUTPUT

```
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.
```



Modifying a String(contd.) concat()

concat( )

- We can use **concat()** method to concatenate two strings.

  String concat(String *str)*

- This method ***creates a new object*** that contains the invoking string with the value of *str appended to the end of it.*
- ***concat( ) performs the same function as* +.**
- ***For example,***

```
String s1 = "one";       // string s1 contains"one"
String s2 = s1.concat("two");
```

  - Here s1 is the invoking string that call the function concat() .
  - s1 contains "one" and is concatenated with argument string value "two" and form the string "onetwo". This result is stored in the String object **s2**

Modifying a String(contd.) replace( )

replace( ) -The **replace( ) method has two forms.**

1. The first form **replaces all occurrences of one character in** the invoking string with another character.

   String replace(char *original, char replacement)*

   – Here, *original* specifies the character that will be replaced by the character specified by *replacement.*

   – The resulting string is returned.

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

   – Here letter l is replaced by w. So "Hewwo" is put into String object s.

2. The second form of **replace( ) replaces one character sequence with another.**

String replace(CharSequence *original, CharSequence replacement)*

This form was added by J2SE 5.

Modifying a String(contd.) replace( )

replace( ) -The **replace( ) method has two forms.**

1. The first form **replaces all occurrences of one character in** the invoking string with another character.

   String replace(char *original, char replacement)*

   – Here, *original* specifies the character that will be replaced by the character specified by *replacement.*

   – The resulting string is returned.

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

   – Here letter l is replaced by w. So "Hewwo" is put into String object s.

2. The second form of **replace( ) replaces one character sequence with another.**

String replace(CharSequence *original, CharSequence replacement)*

**This form was added by J2SE 5.

Modifying a String(contd.) trim( )

trim( )

- The **trim( )** method returns a copy of the invoking string after **removing any leading and trailing whitespace**
- General form:

String trim( )

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

  - This puts the string "Hello World" into **s.**
- The **trim( )** method is quite useful when we process user commands.

- **E.g.** Write a program that prompts the user to **enter the name** of a state(Assam,Goa etc) and then **displays that state's capital**. Use **trim( )** to *remove any leading or trailing whitespace* that may have inadvertently been entered by the user.

```
import java.io.*;
class UseTrim {
public static void main(String args[]) throws
                                        IOException
{
BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
String str;
System.out.println("Enter 'stop' to quit.");
do {
System.out.println("Enter the State: ");
str = br.readLine();
str = str.trim();
if(str.equals("Assam"))
      System.out.println("Capital is  Dispur");
else if(str.equals("Goa"))
      System.out.println("Capital is Panaji");
else if(str.equals("Bihar"))
      System.out.println("Capital is Patna.");
else
System.out.println("Capital is not entered");
      } while(!str.equals("stop"));
  }
}
```

Data Conversion Using valueOf( )

- The valueOf( ) method **converts** data from its internal format **into a human-readable form.**
- It is a **static** method**.**
- **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.

valueOf()(contd.)

The valueOf() **returns the string representation of the corresponding argument.** Different overloaded form of valueOf() in String class.

- **valueOf(boolean b)** – Returns the string representation of boolean argument.
- **valueOf(char c)** – char argument.
- **valueOf(char[] data)** char array argument.
- **valueOf(char[] data, int offset, int count)** – specific subarray of the char array argument.
- **valueOf(double d)** – double argument.
- **valueOf(float f)** – float argument.
- **valueOf(int i)** – int argument.
- **valueOf(long l)** – long argument.
- **valueOf(Object obj)** – Object argument. (calls toString() method of the class Object(parent class of all classes n Java)

valueOf()(contd.)

- **valueOf( )** is called when **a string representation of some other type of data is needed**
  - example, during concatenation operations
- Any object that we pass to **valueOf( )** will return the result of a call to the object's **toString( )** method**.**
- For most arrays, **valueOf( )** returns a rather cryptic string, which indicates that it is an array of some type.
- For arrays of **char,** however, a String object is created that contains the characters in the char array

Changing the Case of Characters Within a String

- String toLowerCase( )
- 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.
```

Comparison of String Buffer and String.

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

String

- **String** is immutable.
- **String** represents **fixed-length, immutable character** sequences.
- Concatenation using String is slow.
- String class can override equals() method.

StringBuffer

- **StringBuffer** is **mutable.**
- **StringBuffer** represents **growable** and **writeable** character sequences
- Concatenation using StringBuffer is fast.
- StringBuffer class doesnot override equals() method.

String str = "Hello World";

str = "Hi World!";

- Here an object is created using string literal “Hello World”.
- In second statement when we assigned the new string literal “Hi World!” to str, the **object itself didn’t change** instead a **new object got created in memory** using string literal “Hi World!” and the reference to it is assigned to str.

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.

StringBuffer(contd.)

length( ) and capacity( )

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

int length( )
int 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());
}
}
```

OUTPUT
buffer = Hello
length = 5
capacity = 21

Here capacity is 21 because room for 16 additional characters is automatically added to value Hello

StringBuffer(contd.)

ensureCapacity( )

- **ensureCapacity**( ) is used to set the size of the buffer.
- This is useful if we know in advance that we will be appending a large number of small strings to a **StringBuffer.**

```
void ensureCapacity(int capacity)
```

- Here, *capacity specifies the size of the buffer.*

StringBuffer(contd.)

setLength( )

- Used to set the length of the buffer within a **StringBuffer object.**

```
void setLength(int len)
```

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

- When we increase the size of the buffer, **null characters** are added to the end of the existing buffer.
- If we call **setLength( )** with a value **less than the current value returned by length( ),** then the characters stored **beyond the new length will be lost.**

StringBuffer(contd.)

- **charAt( )** and **setCharAt( )**
- The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method.
- We can set the value of a character within a **StringBuffer** using **setCharAt( ).**

char charAt(int *where)*

void setCharAt(int *where, char ch)*

StringBuffer(contd.)


```
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
```

StringBuffer(contd.)

- **getChars( )**
- Used to copy a substring of a **StringBuffer.**

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

StringBuffer(contd.)

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
- The result is appended to the current **StringBuffer object.**
- The buffer itself is returned by each version of **append**( ).
  - **append() calls can be chained**

StringBuffer(contd.)


```
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!

StringBuffer(contd.)

insert( )

- The **insert( )** method inserts one string into another.
- It calls **String.valueOf( ).**
- 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)*

StringBuffer(contd.)


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

OUTPUT

I like Java!

StringBuffer(contd.)

reverse( )

- We can reverse the characters within a **StringBuffer** object using **reverse( ):**

StringBuffer **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

StringBuffer(contd.)

delete( ) and **deleteCharAt( )**

- We can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**

StringBuffer delete(int *startIndex, int endIndex)*

StringBuffer deleteCharAt(int *loc)*

- **delete()** deletes from *startIndex to endIndex–1*.
- The **deleteCharAt( )** method deletes the character at the index specified by *loc*

StringBuffer(contd.)


```
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: This a test.

After deleteCharAt: his a test.

StringBuffer(contd.)

replace( )

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

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

  - The substring at *startIndex through endIndex–1 is replaced.*

```
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);
}
}
```

OUTPUT

After replace: This was a test

StringBuffer(contd.)

substring( )

- We can obtain a portion of a **StringBuffer** by calling **substring( ).**

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*.

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.