STRING HANDLING

INTRODUCTION

- Java implements strings as objects of type string. This helps in building efficient string handling functions.
- String objects are immutable:
  - Each time you change the contents of the object a new object is created and object values cannot be changed.
  - Mutable objects can be created using StringBuffer and StringBuilder.
- All three classes are declared as final, so they don’t contain subclasses.
- They implement CharSequence interface.

STRING CONSTRUCTORS

- To create empty String
  - String s= new String();
- To create String initialized with array of characters.
  - String(char *chars[ ])*
    - Here is an example:
    - char chars[] = { 'a', 'b', 'c' };
    - String s = new String(chars);
    - This constructor initializes **s with the string "abc".**
- To create subrange of a character array as an initializer
  - String(char *chars[ ], int startIndex, int numChars)*
  - *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.**

STRING CONSTRUCTORS(CONTD..)

- construct a **String object that contains the same character sequence as another String object .**
  - String(String *strObj)*
  - Here, *strObj is a* ***String object.***
  - ***Consider this example:***

```
// 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(CONTD..)

- Using ASCII Character Set
  - String(byte *asciiChars[ ])*
  - String(byte *asciiChars[ ], int startIndex, int numChars)*
  - *asciiChars specifies the array of bytes.*
  - The byte-to-character conversion is done by using The default character encoding of the platform.

STRING CONSTRUCTORS(CONTD..)

- Using ASCII Character Set(Contd..)
- EX:

```
// 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);
}
}
```

- Output:
  - ABCDEF
  - CDE

STRING CONSTRUCTORS(CONTD..)

- String from StringBuffer.
  - String(StringBuffer *strBufObj)*
- J2SE 5 added two constructors to **String**
  - String(int *codePoints[ ], int startIndex, int numChars)*
  - String(StringBuilder *strBuildObj)*

STRING LENGTH

- The length of a string is the number of characters that it contains.
  - int length( )
  - EX: char chars[] = { 'a', 'b', 'c' };
    - String s = new String(chars);
    - System.out.println(s.length());
    - //output: 3

SPECIAL STRING OPERATIONS

- 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.

SPECIAL STRING OPERATIONS(CONTD..)

- **String Literals**
  - String s2 = "abc";
- Java automatically constructs a String object. Thus, you can use a string literal to initialize a String object.
- String object is created for every string literal, you can use a string literal any place you can use a String object.
- System.out.println("abc".length()); // Valid Function on string Literal Directly.

SPECIAL STRING OPERATIONS(CONTD..)

- **String Concatenation**
- + operator is used to concatenate the string.

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

SPECIAL STRING OPERATIONS(CONTD..)

- **String Concatenation with Other Data Types.**

  ```
  String s = "four: " + 2 + 2;
  System.out.println(s);
  ```

- String s = "four: " + (2 + 2);

SPECIAL STRING OPERATIONS(CONTD..)

- **String Conversion and toString( )**
- Every class implements toString( ) because it is defined by Object.
- If you want to override toString( ) and provide your own string representations.
  - Public String toString( )

CHARACTER EXTRACTION

- The String class provides a number of ways in which characters can be extracted from a String object.
- **charAt( )**
  - To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method. It has this general form:
  - char charAt(int *where)*
  - *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.

CHARACTER EXTRACTION(CONTD..)

- **getChars( )**
- to extract more than one character at a time.
- void getChars(int *sourceStart, int sourceEnd, char target[ ], int targetStart)*
- *sourceStart specifies the index of the beginning of the substring.*
- *sourceEnd* specifies an index that is one past the end of the desired substring.
- the substring contains the characters from *sourceStart through sourceEnd.* 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.*
- Care must be taken to assure that the *target array is large enough to hold the* number of characters in the specified substring.

CHARACTER EXTRACTION(CONTD..)

- **getChars( )** (Contd..)
- **EX:**

```
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);
}
}
```

output is: demo

CHARACTER EXTRACTION(CONTD..)

- **getBytes()**

- There is an alternative to getChars( ) that stores the characters in an array of bytes.
- it uses the default character-to-byte conversions provided by the platform.
- Here is its simplest form:
  - byte[ ] getBytes( )
- getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters.

CHARACTER EXTRACTION(CONTD..)

- **toCharArray( )**

- to convert all the characters in a **String object into a character array.**
- It returns an array of characters for the entire string.
- It has this general form:
  - char[ ] toCharArray( )
- getChars( ) can be used to achieve the same result.

STRING COMPARISON

- **equals( ) and equalsIgnoreCase( )**
- To compare two strings for equality.
- 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.
- To perform a comparison that ignores case differences, call equalsIgnoreCase( ).
- 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.*
  - *Returns* true if the strings contain the same characters in the same order, and false otherwise.

STRING COMPARISON(CONTD..)

equals( ) and equalsIgnoreCase( )(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 + " -> " +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));
	}
}
```

STRING COMPARISON(CONTD..)

- **equals( ) and equalsIgnoreCase( )(contd..)**
- 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

STRING COMPARISON(CONTD..)

- **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)*
  - *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.

STRING COMPARISON(CONTD..)

- **startsWith( ) and endsWith( )**
- The **startsWith( ) method determines whether a given String begins with a specified string.**
- **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.
- For example,
  "Foobar".endsWith("bar")
  and
  "Foobar".startsWith("Foo")
  are both **true.**
- A second form of **startsWith( ), shown here,**
  - 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.**

STRING COMPARISON(CONTD..)

- **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.

```
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

STRING COMPARISON(CONTD..)

- **compareTo( )**
- It has this 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.

STRING COMPARISON(CONTD..)

- **compareTo( ) (contd..)**

```
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]);
}}}          //Output: Elements in sorted Order.
```

STRING COMPARISON(CONTD..)

- **compareTo( ) (contd..)**
- If you want to ignore case differences when comparing two strings, use compareToIgnoreCase( ), as shown here:
  - int compareToIgnoreCase(String *str)*

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.
- To search for the first occurrence of a character,
  - int indexOf(char *ch)*
- To search for the last occurrence of a character, use
  - int lastIndexOf(char *ch)*
- To search for the first or last occurrence of a substring,
  - int indexOf(String *str)*
  - int lastIndexOf(String *str)*

SEARCHING STRINGS(CONTD..)

- You can specify a starting point for the search
  - int indexOf(char *ch, int startIndex)*
  - int lastIndexOf(char *ch, int startIndex)*
  - int indexOf(String *str, int startIndex)*
  - int lastIndexOf(String *str, int startIndex)*
- EX:

```
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.";
```

SEARCHING STRINGS(CONTD..)

```
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) = " +.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10) = " +s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " +s.lastIndexOf("the", 60));
}
}
```

MODIFYING A STRING

- **substring( )**
- Used to extract a substring from the string.
  - String substring(int startIndex)
  - startIndex specifies the index at which the substring will begin.
  - String substring(int startIndex, int endIndex)
  - 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..)

- **Concat()**
- Used to concatenate two strings.
  - 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 +
  - EX:
    - String s1 = "one";
    - String s2 = s1.concat("two");

      or
    - String s1 = "one";
    - String s2 = s1 + "two";

MODIFYING A STRING(CONTD..)

- **replace( )**
- String replace(char original, char replacement).
- original specifies the character to be replaced by the character specified by replacement.
- String s = "Hello".replace('l', 'w');
- The second form of replace( ) replaces one character sequence with another.
  - String replace(CharSequence original, CharSequence replacement) //from J2SE5

MODIFYING A STRING(CONTD..)

- **trim( )**
- The trim( ) method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.
  - String trim( )
  - EX: String s = " Hello World ".trim();

DATA CONVERSION USING VALUEOF( )

- The valueOf( ) method converts data from its internal format into a human-readable form.
- valueOf( ) is called when a string representation of some other type of data is needed.
- 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[ ])*
- Any object that you pass to valueOf( ) will return the result of a call to the object's toString( ) method.
- For arrays: static String valueOf(char *chars[ ], int startIndex, int numChars)*

CHANGING THE CASE OF CHARACTERS WITHIN A STRING

- String toLowerCase( )
- String toUpperCase( )
- Both methods return a String object that contains the uppercase or lowercase equivalent of the invoking String.

ADDITIONAL STRING METHODS

Method	Description
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*. Added by J2SE 5.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*. Added by J2SE 5.
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. Added by J2SE 5.
boolean contains(CharSequence *str*)	Returns **true** if the invoking object contains the string specified by *str*. Returns **false**, otherwise. Added by J2SE 5.
boolean contentEquals(CharSequence *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**. Added by J2SE 5.
boolean contentEquals(StringBuffer *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
static String format(String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. (See Chapter 18 for details on formatting.) Added by J2SE 5.
static String format(Locale *loc*, String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. Formatting is governed by the locale specified by *loc*. (See Chapter 18 for details on formatting.) Added by J2SE 5.
boolean matches(string *regExp*)	Returns **true** if the invoking string matches the regular expression passed in *regExp*. Otherwise, returns **false**.

ADDITIONAL STRING METHODS(CONTD..)

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*. Added by J2SE 5.
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*.
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*.

ADDITIONAL STRING METHODS(CONTD..)

Method	Description
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*.
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.
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**.

STRINGBUFFER

- 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 additions and often has more spcaes preallocated.
- **Constructors:**
  - StringBuffer( ) // Default size is 16 characters.
  - StringBuffer(int *size)*
  - StringBuffer(String *str) // String, with 16 characters extra.*
  - StringBuffer(CharSequence *chars)*

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.
- They have the following general forms:
  - **int length( )**
  - **int capacity( )**

STRINGBUFFER(CONTD..)

- **length( ) and capacity( ) (Contd..)**

```
// 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());
}
}
```

STRINGBUFFER(CONTD..)

- **ensureCapacity( )**
- If you want to preallocate room for a certain number of characters after a StringBuffer has been constructed.
- **void ensureCapacity(int *capacity)***
  - *capacity specifies the size of the buffer.*

- **setLength( )**
- To set the length of the buffer within a StringBuffer object.
- **void setLength(int *len)***
  - *len specifies the length of the buffer. This value must be nonnegative.*

STRINGBUFFER(CONTD..)

- **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( ).
- The general forms are:
  - char charAt(int *where)*
    - *where specifies the index of the character being obtained.*
  - void setCharAt(int *where, char ch)*
    - *where specifies the index of the character being set.*
    - *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.

STRINGBUFFER(CONTD..)

- **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)*
    - *sourceStart specifies the index of the beginning of the substring.*
    - *sourceEnd specifies* an index that is one past the end of the desired substring.
    - *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.*

STRINGBUFFER(CONTD..)

- **append( )**
- The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object.
- General 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 append( ) method is most often called when the + operator is used on String objects. Java automatically changes modifications to a String instance into similar operations on a StringBuffer instance. Thus, a concatenation invokes append( ) on a StringBuffer object. After the concatenation has been performed, the compiler inserts a call to toString( ) to turn the modifiable StringBuffer back into a constant String.

STRINGBUFFER(CONTD..)

- **append( )** (contd..)

```
// 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);
}
}
```

Output: a = 42!

STRINGBUFFER(CONTD..)

- **insert( )**
- The insert( ) method inserts one string into another. It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences.
- Like append( ), it calls String.valueOf( ) to obtain the string representation of the value it is called with.
- **General Form**
  - StringBuffer insert(int *index, String str)*
  - StringBuffer insert(int *index, char ch)*
  - StringBuffer insert(int *index, Object obj)*
  - index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

STRINGBUFFER(CONTD..)

- **insert( )** **(contd..)**

```
// Demonstrate insert().
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()**
- You can reverse the characters within a StringBuffer object using reverse( ).
  - 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);
}
}
```

- Output: abcdef

  fedcba

STRINGBUFFER(CONTD..)

- **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)*
  - *startIndex specifies the index of the first character to remove.*
  - *endIndex specifies an index* one past the last character to remove.

STRINGBUFFER(CONTD..)

- **delete( ) and deleteCharAt( ) (contd..)**
- // 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: This a test.

After deleteCharAt: his a test.

STRINGBUFFER(CONTD..)

- **replace( )**
- You 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 being replaced is specified by the indexes *startIndex and endIndex.*
  - substring at *startIndex through endIndex–1 is replaced. The replacement string is passed in str.*
  - The resulting StringBuffer object is returned.

STRINGBUFFER(CONTD..)

- **replace( ) (contd..)**

```
// 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.

STRINGBUFFER(CONTD..)

- **substring( )**
- You 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. These methods work just like those defined for String.*

STRINGBUFFER(CONTD..)

- **Additional String Buffer Methods.**

Method	Description
StringBuffer appendCodePoint(int *ch*)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned. Added by J2SE 5.
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*. Added by J2SE 5.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*. Added by J2SE 5.
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. Added by J2SE 5.
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.

STRINGBUFFER(CONTD..)

- **Additional String Buffer Methods.**

Method	Description
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*. Added by J2SE 5.
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 **StringBuffer**.
void trimToSize( )	Reduces the size of the character buffer for the invoking object to exactly fit the current contents. Added by J2SE 5.