Module 3

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

String

A *string* is a sequence of characters.

Some languages implement strings as character arrays, Java implements strings as an object (of type String).

Implementing strings as object facilitates Java to provide a full features that makes string handling convenient.
Ex: Java has methods to compare two strings, search for a substring, concatenate two
 strings, and change the case of letters within a string.

String objects can be constructed in different ways.

When a String object is created, it cannot be modified.

There is a restriction, that characters in the string are not modifiable, even so all types of String operations can be performed on a String object, each time a String object is modified a new String object is created that contains the modifications. The original string is left unchanged.

String

For those cases in which a string modification is required, Java provides two classes: **StringBuffer** and **StringBuilder**.

Both hold strings that can be modified after they contain value within it.

The **String**, **StringBuffer**, and **StringBuilder** classes are defined in java.lang package.

All these classes are declared as **final**, which means that none of these classes may be inherited.

All three implement the **CharSequence** interface.

Contents of the **String instance** cannot be modified after it’s instantiation.

However, a variable declared as a String reference can be changed to point at some other String object at any point of time.

String - The String Constructors

String class supports several constructors.
To create an empty **String**, default constructor can be used.

String s = new String();

will create an instance of **String** with empty set of characters in it.

Strings with initial values can also be created. The **String** class provides a variety of constructors to handle this.

To create a **String** initialized by an array of characters, following constructor is used:

String(char *chars*[ ])

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

This constructor initializes **s** with the string “abc”.

String - The String Constructors

A substring of a character array can be specified as an initializer

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.

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

A String object can be constructed, that contains the same character sequence as another **String** object using this constructor:

String(String *strObj*)

Here, *strObj* is a **String** object.

String - The String Constructors

```
// 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 is: Java
           Java
```

String - The String Constructors

Characters in Java can be represented using 8/16-bits, the **String** class provides constructors that initialize a string by a **byte** array.

String(byte *asciiChars*[ ])

String(byte *asciiChars*[ ], int *startIndex*, int *numChars*)

asciiChars specifies the array of bytes.

The second form allows to specify a subrange.

In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.

String - The String Constructors

```
// 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

The contents of the array are copied when a **String** object is created from an array.

If array contents are modified after creating the string, the **String** will be unchanged.

A **String** object can be created from a **StringBuffer** by using the following constructor

String(StringBuffer *strBufObj*)

String Length

Length of a string is the number of characters it contains.

To obtain this value, length( ) instance method can be used: **int length( )**

Following fragment prints "3", since there are three characters in the string s:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);      System.out.println(s.length());
```

Special String Operations

Strings are considered as **a common and important type** in Java language, hence Java has added special support for several string operations.

These operations include the **automatic creation of new String instances from string literals**, concatenation of multiple String objects by use of the + operator, and the conversion of other data types to a string representation.

Explicit methods are available to perform all these tasks, but Java does them automatically as a convenience for the programmer and to add clarity.

String Literals

String literals can be used as an initial value for the String object to be created.

For example, the following code fragment creates two equivalent strings:

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

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

s1==s2 =false
s1.equals(s2) =true

Because **a String object is created for every string litera**l, a string literal can be used in any place a String object is used.

For example, methods can be called directly on a quoted string as if it were an object reference,

```
System.out.println("abc".length());
```

String Concatenation

In general, Java does not allow operators to be applied to String objects.

The one exception to this rule is the + operator, which concatenates two strings, producing a String object as the result.

This allows to chain together a series of **+** operations.

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

Output: “He is 9 years old.”

One practical use of string concatenation is when creating very long strings.

Instead of letting long strings wrap around source code, it can be broken down into smaller pieces, using the **+** to concatenate them.

String Concatenation

```
// Using concatenation to prevent long lines.
class ConCat
{
  public static void main(String args[])
  {
    String longStr = "This could have been " +

     "a very long line that would have " +
     "wrapped around.  But string concatenation " +
     "prevents this.";

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

String Concatenation with Other Data Types

Strings can be concatenated with other data types.

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

int value in **age** is automatically converted into its string representation within a **String** object. This string is then concatenated.

The compiler will convert an operand to its string equivalent whenever the other operand of the **+** is an instance of **String**.

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

This fragment displays

```
four: 22
```

rather than the

```
four: 4
```

String Concatenation with Other Data Types

To complete the integer addition, parentheses must be used:

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

Now **s** contains the string “four: 4”.

String Conversion and toString( )

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**.

```
public static String valueOf(char c);
public static String valueOf(int c);
public static String valueOf(Object c);
….
```

valueOf( ) is overloaded for all the simple types (Wrapper classes) and for type **Object**. (valueOf( ) is class level function)

For simple types, **valueOf( )** returns a string that contains the information in String format.

String Conversion and toString( )

toString( ) method, is one of the ways by which the string representation for objects of classes can be determined.

Every class implements (built-in classes) **toString( )** because it is defined by **Object**. However, the default implementation of **toString( )** is **seldom sufficient**.

For user-defined classes **toString( )** can be overridden and provide customized string representations of the information present in the object.

The **toString( )** method has this general form:

String toString( )

To implement **toString( )**, return a **String** object that contains the readable string that appropriately describes an object of the class.

String Conversion and toString( )

By overriding **toString( )** for classes that are created, it is made fully integrated into Java's programming environment.

For example, they can be used in **print( )** and **println( )**statements and in concatenation expressions. The following program demonstrates this by overriding **toString( )** for the **Box** class:

Ex:

```
// Override toString() for Box class.
class Box {
 double width, height,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 and 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);
    }
}
```

Box's toString( ) method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println( )**.

Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object.

Characters that are part of a **String** object cannot be indexed.

Many of the **String** methods use an index (or offset) into the string, like arrays, to extract a character, string indexes begin at zero.

charAt( )

To extract a single character from a **String**, the character can be referred via the **charAt( )** method.

char charAt(int *where*)

Here, *where* is the index of the character that is to be retrieved.

The value of *where* must be nonnegative and specify a location within the string.

charAt( )

charAt( ) returns the character at the specified location.

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

assigns the value “**b**” to **ch**.

getChars( )

If there is a necessity to extract more than one character at a time, **getChars( )** method can be used.

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

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*-1.

The array that will receive the characters is specified by *target*. The index within the target at which the substring will be copied is passed in *targetStart*.

getChars( )

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

Ex:

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

byte[ ] getBytes( )

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

Ex:

```
class Main {
        public static void main(String args[]) {
         String s = "Engineering";
         byte [] b = new byte[50];

         b=s.getBytes();

         for(int i=0;i<s.length();i++)
          System.out.println(b[i]);
        }
}
```

Output: 69 110 113….

toCharArray( )

To convert all characters in a String object into a character array **toCharArray( )** method can be used. It returns an array of characters for the entire string.

char[ ] toCharArray( )

This function is provided as a convenience, since it is possible to use **getChars( )** to achieve the same result.

String Comparison

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

equals( ) and equalsIgnoreCase( )

To compare two strings for equality, **equals( )** method can be used.

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.

equals( ) and equalsIgnoreCase( )

To perform a comparison that ignores case differences, **equalsIgnoreCase( )** can be used. When it compares two strings, it considers **A-Z** to be the same as **a-z**.

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.

Ex:
```
// Demonstrate equals() and equalsIgnoreCase().
        class equalsDemo {
         public static void main(String args[]) {
            String s1 = "Hello", s2 = "Hello", s3 = "Good-bye", s4 = "HELLO";

            System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2)); //true
            System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3)); //false
            System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));  //false
            System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                                              s1.equalsIgnoreCase(s4)); //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 to ignore case in such comparisons.

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

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

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 given String begins with a specified string.

Conversely, endsWith( ) determines whether the String in question ends with a specified string.

```
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.

Ex:

```
"Foobar".endsWith("bar") and
"Foobar".startsWith("Foo")    are both true.
```

A second form of startsWith( ), shown here, lets programmer to 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.

```
"Foobar".startsWith("bar", 3)        returns true.
```

equals( ) Versus ==

The equals( ) method and the == operator perform two different operations.

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 two different String objects can contain the same characters, but references to these objects will not compare as equal:

Ex:
```
// equals() vs ==
class Main {
   public static void main(String args[]) {
       String s1 = "Hello";
       String s2 = new String(s1);
       System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));  //true
       System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));  //false
 }   }
```

equals( ) Versus ==

The variable **s1** refers to the **String** instance created by "**Hello**".
The object referred to by **s2** is created with **s1** as an initializer. Thus, the contents of the two **String** objects are identical, but they are distinct objects. This means that **s1** and **s2** do not refer to the same objects and are, therefore, not **==**.

compareTo( )

For sorting applications, it is necessary to know which value of the string is *less than, equal to,* or *greater than* the next string.

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.

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, as shown below

compareTo( )

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 {
    public static void main(String args[]) {
    String arr[] = {"Now", "is", "the", "time", "for", "all", "good", "men",
            "to", "come", "to", "the", "aid", "of", "their", "country"  };
        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]);    }   }   }
```

compareTo( )

compareTo( ) takes into account uppercase and lowercase letters.

The word “Now” came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

If case differences has to be ignored when comparing two strings, **compareToIgnoreCase( )** has to be used.

int compareToIgnoreCase(String *str*)

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

If the above method is replaced in the previous program, “Now” will no longer be first.

Searching Strings

The **String** class provides two methods that allows 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, 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.

Searching Strings

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.

Starting point for the search can be specified 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.

Searching Strings

Ex: // 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));
       }
}
```

Modifying a String

String objects are immutable, whenever a **String** object is supposed to be modified, it has to be copied either into a **StringBuffer** or **StringBuilder**, or one of the following **String** methods can be used, which will construct a new copy of the string with modifications.

substring( )

A substring can be extracted using **substring( )**. It has two forms.

String substring(int *startIndex*)

Here, *startIndex* specifies the index at which the substring will begin.

This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring( )** allows 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.

substring( )

The following program uses **substring( )** to replace all instances of one substring with another within a string:

```
     class StringReplace {
      public static void main(String args[]) {
         String org = "This is a test. This is, too.", search = "is",  sub = "was", 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);
} }
```

substring( )

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( )

Two strings can be concatenated using **concat( )**

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");
```

concat( )
puts the string “onetwo” into **s2**. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

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

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.
Ex:

String s = "Hello".replace('l', 'w'); puts the string “Hewwo” into **s**.

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

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

trim( )

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();

This puts the string "Hello World" into **s**.

The **trim( )** method is quite useful when user commands are processed.

Data Conversion Using valueOf( )

The **valueOf( )** method converts data from its internal format into a string form.

It is a static method that is overloaded within **String** for all of Java's built-in types so that each type can be converted properly into a string.

Data Conversion Using valueOf( )

valueOf( ) is also overloaded for type **Object**, so an object of any class type created 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*[ ]) ...**

valueOf( ) is called when a string representation of some other type of data is needed.

This method can be called directly with any data type and get a **String** representation. All of the simple types are converted to their common **String** representation.

Any object that is passed to **valueOf( )** will return the result of a call to the object's **toString( )** method. **toString( )** can be called directly and can obtain the same result.

Data Conversion Using valueOf( )

For arrays of **char**, a **String** object is created that contains the characters in the **char** array.

There is a special version of **valueOf( )** that allows to specify a subset of a **char** array.

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

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.

Changing the Case of Characters Within a String

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

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

Changing the Case of Characters Within a String

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

String toLowerCase( )

String toUpperCase( )

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

Ex:// Demonstrate toUpperCase() and toLowerCase().

```
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);
     }    }
```

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**.
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

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 is a peer class of **String** that provides much of the functionality of strings.

String represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writable 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 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

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.

Ex: StringBuffer sb = new StringBuffer( ); System.out.println(sb.capacity());

StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. 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.

int length( ) **int capacity( )**

StringBuffer

Ex:

```
// 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 **sb** is 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.

StringBuffer : ensureCapacity( )

If there is a need to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, **ensureCapacity( )** can be used to set the size of the buffer.

This is useful if it is known in advance the number of characters to be stored in **StringBuffer**

void ensureCapacity(int *capacity*)

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

setLength( ) (works on characters stored in StringBuffer)

To set the length of the buffer within a **StringBuffer** object, **setLength( )** can be used.

void setLength(int *len*)

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

When the size of the buffer is increased, null characters are added to the end of the existing buffer.

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

StringBuffer

```
StringBuffer sb = new StringBuffer("abc");
System.out.println(sb.capacity());  //19

sb.ensureCapacity(20); // increased to 40                    (19*2)+2

System.out.println(sb.capacity());//40
sb.setLength(2);// limits character's to only 2

System.out.println(sb.capacity());//40

System.out.println(sb.length()); //2
System.out.println(sb);  //ab
```

StringBuffer : charAt( ) and **setCharAt( )**

The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method. The value of a character can be set 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.

StringBuffer : charAt( ) and **setCharAt( )**

Ex:

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

StringBuffer : getChars( )

To copy a substring of a **StringBuffer** into an array, the **getChars( )** method can be used.

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.

The array that will receive the characters is specified by *target.* The index within the 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.

StringBuffer : append( )

The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions.

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( )**. .

Ex:
```
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 is :   a = 42!

StringBuffer : 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. 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.

Ex:
```
// 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 is :
```
I like Java!
```

StringBuffer :

reverse( )

Characters within a StringBuffer object can be reversed using reverse( )

StringBuffer reverse( )

This method returns the reversed object on which it was called.

Ex:

```
// 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 is:

```
abcdef
fedcba
```

StringBuffer : delete( ) and deleteCharAt( )

Characters within a StringBuffer can be deleted using the methods delete( ) and deleteCharAt( ).

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.

StringBuffer : delete( ) and deleteCharAt( )

Ex:

```
// 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 : replace( )

A set of characters can be replaced 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*. Thus, the substring at *startIndex* through *endIndex*–1 is replaced.

The replacement string is passed in str. The resulting StringBuffer object is returned.

Ex:

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

Output:

```
After replace: This was a test.
```

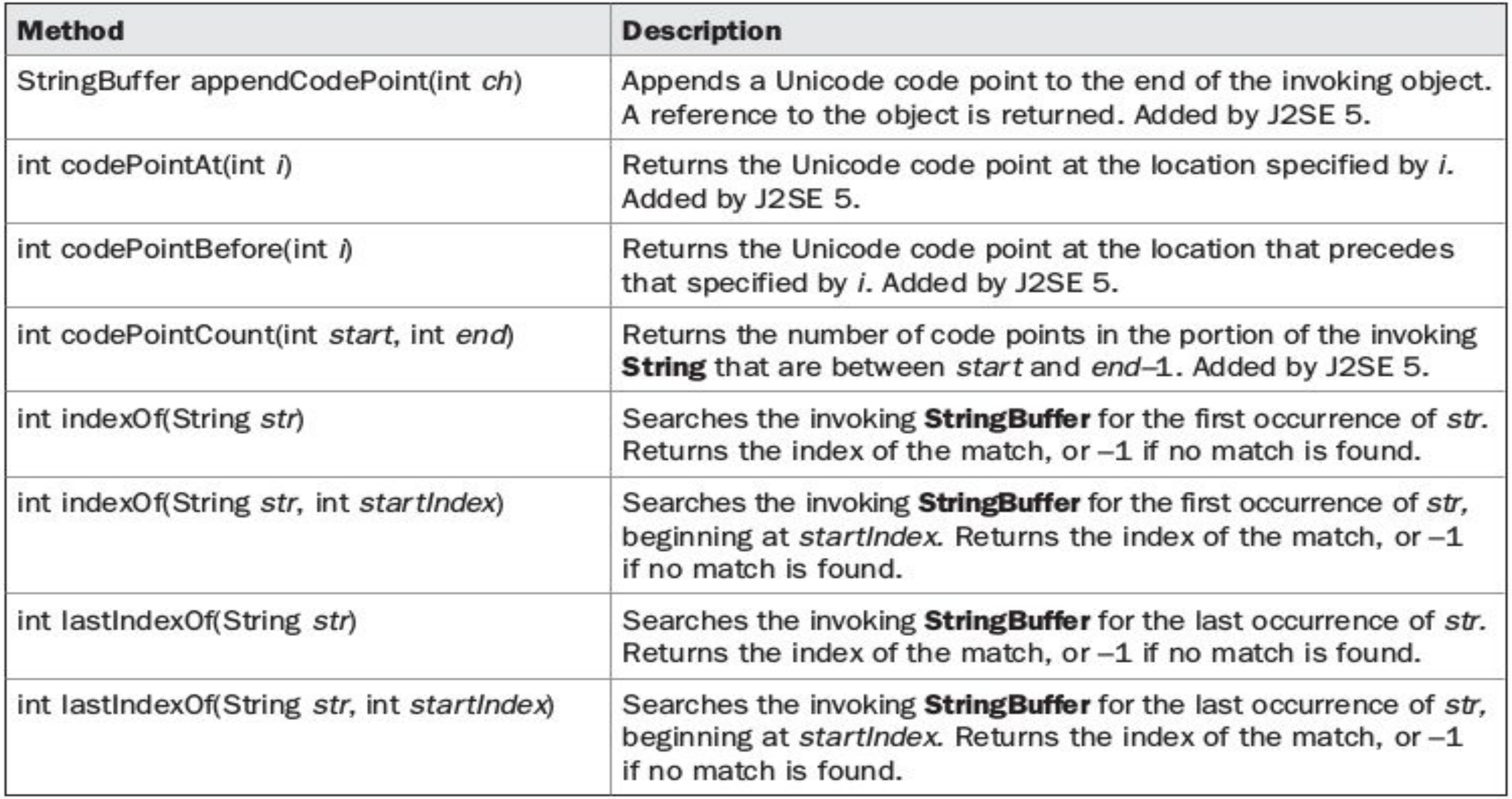
StringBuffer : substring( )
A portion of a StringBuffer can be obtained by calling substring( ).

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.

Additional StringBuffer 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.

Additional StringBuffer 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.

Additional StringBuffer Methods

Aside from subSequence( ), which implements a method required by the CharSequence interface, the other methods allow a StringBuffer to be searched for an occurrence of a String.

The following program demonstrates indexOf( ) and lastIndexOf( ):

```
class IndexOfDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("one two one");
        int i;
        i = sb.indexOf("one");
        System.out.println("First index: " + i);
        i = sb.lastIndexOf("one");
        System.out.println("Last index: " + i);
    }
}
```

The output is shown here:

```
First index: 0
Last index: 8
```

StringBuilder

J2SE 5 adds a new string class to Java.

This new class is called StringBuilder.

It is identical to StringBuffer except for one important difference: **it is not synchronized,** which means that it is not thread-safe.

The advantage of StringBuilder is faster performance.

However, in using multithreading, StringBuffer must be used rather than StringBuilder.