Java Lecture-6

String Handling

- ❖ In Java a string is a sequence of characters
- ❖ Java implements strings as objects of type **String**
- ❖ Java provides two options: **StringBuffer** and **StringBuilder**
- ❖ Both hold strings that can be modified after they are created
- ❖ The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**.

- ❖ The strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created

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

The String Constructors

To create an empty **String**, call the default constructor. For example,

```
String s = new String();
```

will create an instance of **String** with no characters in it.

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

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".

You can specify a subrange of a character array as an initializer using the following

Constructor:

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

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

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

This initializes **s** with the characters **cde**.

You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

String(String *strObj*)

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

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

The output from this program is as follows:

```
Java
Java
```

The **String** class provides constructors that initialize a string when given a **byte** array.

Two forms are shown here:

String(byte *chrs*[ ])

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

Here, *chrs* specifies the array of bytes.

The second form allows you 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.

```
// 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 Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:

int length( )

The 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

❖ String Literals

We can use a string literal to initialize a **String** object

```
char chars[] = { 'a', 'b', 'c' };
     String s1 = new String(chars);
    String s2 = "abc"; // use string literal
```

String object is created for every string literal, we can use a string literal any place we can use a **String** object

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

❖ String Concatenation

The **+** operator concatenates two strings, producing a **String** object as the result.

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

This displays the string "He is 9 years old."

```
// 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

We can concatenate strings with other types of data.

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

Here, the **int** value in **age** is automatically converted into its string representation within a **String** object.

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

Character Extraction

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*)

Here, *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**.

getChars( )

If you need to extract more than one character at a time, you can use the **getChars( )** method.

It has this general form:

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

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

Thus, the substring contains the characters from *sourceStart* through *sourceEnd*–1.

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

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

Here is the output of this program:

```
demo
```

getBytes( )

There is an alternative to **getChars( )** that stores the characters in an array of bytes.

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

Here is its simplest form:

byte[ ] getBytes( )

toCharArray( )

If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**.

It returns an array of characters for the entire string.

It has this general form:

char[ ] toCharArray( )

String Comparison

The **String** class includes a number of methods that compare strings or substrings within strings.

equals( ) and equalsIgnoreCase( )

To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object *str*)

Here, *str* is the **String** object being compared with the invoking **String** object.

It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

boolean equalsIgnoreCase(String *str*)

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "Good-bye";
    String s4 = "HELLO";
    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " equals " + s3 + " -> " +
                       s1.equals(s3));
    System.out.println(s1 + " equals " + s4 + " -> " +
                       s1.equals(s4));
    System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
                       s1.equalsIgnoreCase(s4));
  }
}
```

The output from the program is shown here:

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

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*)

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( )

String defines two methods that are, more or less, specialized forms of **regionMatches( )**.

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.

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**.

equals( ) Versus ==

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

As just explained, 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.

```
// equals() vs ==

class EqualsNotEqualTo {

public static void main(String args[]) {

   String s1 = "Hello";

    String s2 = new String(s1);

   System.out.println(s1 + " equals " + s2 + " -> " +

                       s1.equals(s2));

   System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));

  } }
```

The variable **s1** refers to the **String** instance created by **"Hello"**.

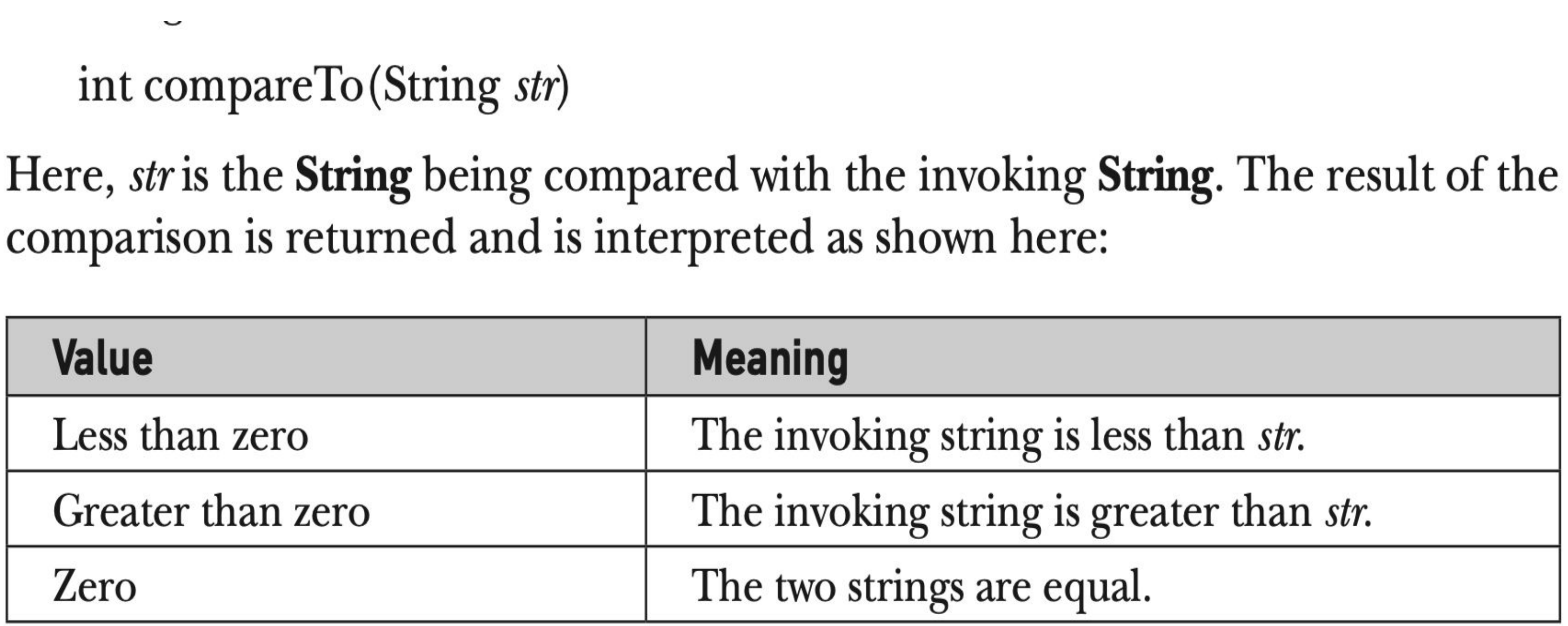
The object referred to by **s2** is created with **s1** as an initializer.

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 ==, as is shown here by the output of the preceding example:

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

compareTo( )



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 here:

Value	Meaning
Less than zero	The invoking string is less than *str*.
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

```
// A bubble sort for Strings.
class SortString {
 static String arr[] = {  "Now", "is", "the", "time", "for", "all", "good", "men",
    "to", "come", "to", "the", "aid", "of", "their", "country"};

  public static void main(String args[]) {

   for(int j = 0; j < arr.length(); j++) {
      for(int i = j + 1; i < arr.length(); i++) {
       if(arr[i].compareTo(arr[j]) < 0) {
          String t = arr[j];
            arr[j] = arr[i];

              arr[i] = t; }

}
          System.out.println(arr[j]);
         }

} }
```

The output of this program is the list of words:

```
Now
 aid
 all
 come
 country
 for
 good
is
 men
 of
 the
 the
 their
 time
 to

 to
```

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

int compareToIgnoreCase(String *str*)

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

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, use

int indexOf(char *ch*)

To search for the last occurrence of a character, use int lastIndexOf(char *ch*)

Here, *ch* is the character being sought.
To search for the first or last occurrence of a substring, use

int indexOf(String *str*)
int lastIndexOf(String *str*)

Here, *str* specifies the substring.

You can specify a starting point for the search using these forms:

int indexOf(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.

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

Here is the output of this program:

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


lastIndexOf(the) = 55
 indexOf(t, 10) = 11
 lastIndexOf(t, 60) = 55
 indexOf(the, 10) = 44
 lastIndexOf(the, 60) = 55
```

Modifying a String

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use a **String** method that constructs a new copy of the string with your modifications complete.

substring( )

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

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 you to specify both the beginning and ending index of the substring:

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

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point.

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

The output from this program is shown here:

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

concat( )

You can concatenate two strings using **concat( )**, shown here:

String concat(String *str*)

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

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

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. It has the following general form:

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

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

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

puts the string "Hewwo" into **s**.

trim( )

The **trim**( ) method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim( )

Here is an example:

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

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

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**.

StringBuffer

- **StringBuffer** supports a modifiable string.

- **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 additions.

StringBuffer Constructors

StringBuffer defines these four constructors:

StringBuffer( )

StringBuffer(int *size*)

StringBuffer(String *str*)

StringBuffer(CharSequence *chars*)

length( ) and capacity( )

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

They have the following general forms:

int length( )

int capacity( )

```
// 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
```

ensureCapacity( )

setLength( )

charAt( )

setCharAt( )

getChars( )

append( )

insert( )

reverse( )

delete( )

deleteCharAt( )

replace( )

substring( )