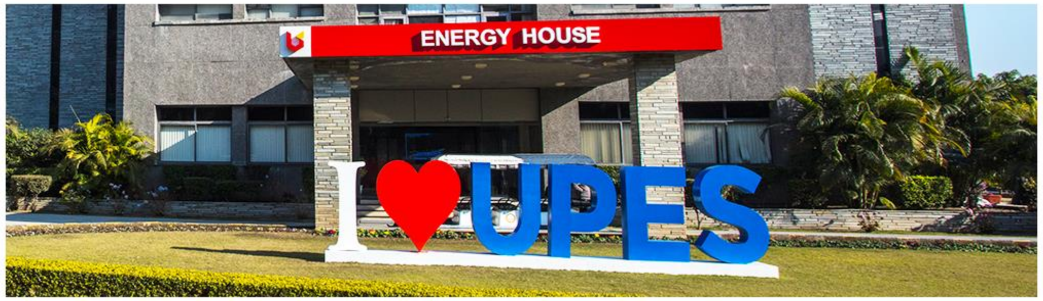


UPES

UNIVERSITY WITH A PURPOSE

Object Oriented Programming

Character Extraction

- The String class provides a number of ways in which characters can be extracted from a String object.
- Although the characters that comprise a string within a String object cannot be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation.
- Like arrays, the string indexes begin at zero.

UPES
UNIVERSITY WITH A PURPOSE

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**.

Character Extraction: 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 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*.

Character Extraction: getChars( )

Exp:

```
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:

```
demo
```

Character Extraction: 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( )
```

- Other forms of **getBytes( )** are also available. **getBytes( )** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

Character Extraction: 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( )
```

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

String Comparison: 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)
```

→ Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

String Comparison: equals( ) and equalsIgnoreCase( )

```
// 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
```

String Comparison: 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.

UPES

UNIVERSITY WITH A PURPOSE

String Comparison: 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**.

- A second form of **startsWith( )**, shown here, lets you specify a starting point:

```
boolean startsWith(String str, int startIndex)
```

→ Here, *startIndex* specifies the index into the invoking string at which point the search will begin.

→ For example, "Foobar".startsWith("bar", 3) returns **true**.

UPES
TY WITH A PURPOSE

String Comparison: 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)); } }
```

Output:

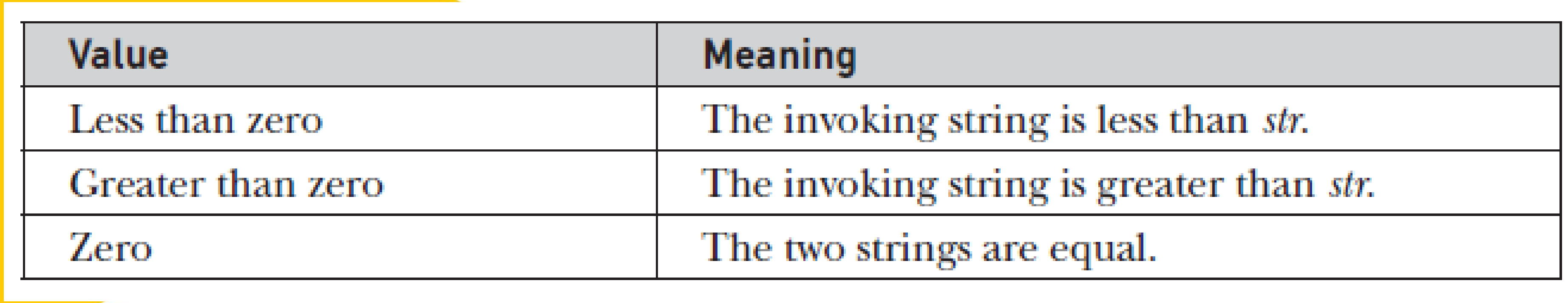
```
Hello equals Hello -> true
Hello == Hello -> false
```

String Comparison: compareTo( )

- Often, it is not enough to simply know whether two strings are identical.
- For sorting applications, you need to know which is *less than*, *equal to*, or *greater than* the next.
- 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 method **compareTo( )** serves this purpose. It is specified by the **Comparable<T>** interface, which **String** implements. It has this general form:

```
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.

String Comparison: compareTo( )

```
// 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:

```
aid
all
come
country
for
good
is
Men
now
of
the
the
their
time
to
to
```

References

Schildt, H. (2014). *Java: the complete reference*. McGraw-Hill Education Group.