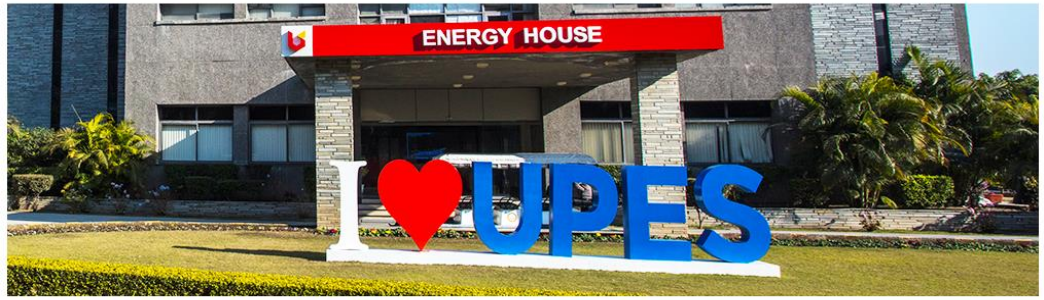

UPES

UNIVERSITY WITH A PURPOSE

Object Oriented Programming

UPES
UNIVERSITY WITH A PURPOSE

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.

→ 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.

- To search for the first or last occurrence of a substring, use

```
int indexOf(String str)
int lastIndexOf(String str)
```

UPES
UNIVERSITY WITH A PURPOSE

Searching Strings

Here, *str* specifies the substring. You can specify a starting point for the search 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

```
// 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: substring( )

- 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.
- 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. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

Modifying a String: substring( )

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

Output :

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

UPES
UNIVERSITY WITH A PURPOSE

Modifying a String: 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";
```

UPES
UNIVERSITY WITH A PURPOSE

Modifying a String: 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.
```

- The second form of **replace( )** replaces one character sequence with another. It has this general form:

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

Modifying a String: 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**.

Data Conversion Using valueOf( )

- The **valueOf( )** method converts data from its internal format into a human-readable 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.
- **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[ ])
```

- **valueOf( )** is called when a string representation of some other type of data is needed—for example, during concatenation operations.

Data Conversion Using valueOf( )

- For most arrays, **valueOf( )** returns a rather cryptic string, which indicates that it is an array of some type.
- For arrays of **char**, however, a **String** object is created that contains the characters in the **char** array.
- There is a special version of **valueOf( )** that allows you to specify a subset of a **char** array. It has this general form:

```
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.

Data Conversion Using valueOf( )

```
public class valueOfDemo {

   public static void main(String args[]) {
      double d = 102939939.939;
      boolean b = true;
      long l = 1232874;
      char[] arr = {'a', 'b', 'c', 'd', 'e', 'f','g' };

      System.out.println("Return Value : " + String.valueOf(d) );
      System.out.println("Return Value : " + String.valueOf(b) );
      System.out.println("Return Value : " + String.valueOf(l) );
      System.out.println("Return  Value  :  "  +  String.valueOf(arr)
);
   }
}
```

UPES
UNIVERSITY WITH A PURPOSE

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. Nonalphabetical characters, such as digits, are unaffected. Here are the simplest forms of these methods:

```
String toLowerCase( )
String toUpperCase( )
```

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

Changing the Case of Characters Within a String

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

Output:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

UPES
UNIVERSITY WITH A PURPOSE

Joining Strings

- JDK 8 adds a new method to **String** called **join( )**.
- It is used to concatenate two or more strings, separating each string with a delimiter, such as a space or a comma.
- It has two forms. Its first is shown here:

```
static String join(CharSequence delim, CharSequence . . . strs)
```

→ Here, *delim* specifies the delimiter used to separate the character sequences specified by *strs*. Because **String** implements the **CharSequence** interface, *strs* can be a list of strings.

UPES
UNIVERSITY WITH A PURPOSE

Joining Strings

```
// Demonstrate the join() method defined by String.
class StringJoinDemo {
public static void main(String args[]) {
String result = String.join(" ", "Alpha", "Beta", "Gamma");
System.out.println(result);
result = String.join(", ", "John", "ID#: 569", "E-mail: John@HerbSchildt.com");
System.out.println(result);
}
}
```

Additional String Methods

Method	Description
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
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.
boolean contains(CharSequence *str*)	Returns **true** if the invoking object contains the string specified by *str*. Returns **false** otherwise.
boolean contentEquals(CharSequence *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
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 19 for details on formatting.)
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 19 for details on formatting.)
boolean isEmpty( )	Returns **true** if the invoking string contains no characters and has a length of zero.
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 within the invoking string that is *num* code points beyond the starting index specified by *start*.
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*.
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*.

UPES
UNIVERSITY WITH A PURPOSE

Additional String Methods

Method	Description
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, but no trailing empty strings will be included.
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 implemented by **String**.

References

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

UPES
UNIVERSITY WITH A PURPOSE