AP Computer Science
Chapter 9 Notes (2)

Name: _____

Strings

`String` Methods:

The more frequently used `String` methods are summarized below. There are methods for returning the string's length, for getting the character at a specified position, for building substrings, for finding a specified character or substring in a string, for comparing strings alphabetically, and for converting strings to upper and lower case.

`length` and `charAt`

The `length` method returns the number of characters in the string. For example:

```
String s = "Internet";
int len = s.length();    // len gets the value 8
```

The `charAt` method returns the character at the specified position.

Character positions in strings are counted starting from 0.

```
int n     = s.length();
char ch   = s.charAt(pos);
String s2 = s.substring(fromPos);
String s2 = s.substring(fromPos, toPos);
String s2 = s.concat(str);
```

```
int result     = s.compareTo(s2);
int result     = s.compareToIgnoreCase(s2);
boolean match  = s.equals(s2);
boolean match  = s.equalsIgnoreCase(s2);
```

```
int k     = s.indexOf(ch);
int k     = s.indexOf(ch, fromPos);
int k     = s.indexOf(str);
int k     = s.indexOf(str, fromPos);
int k     = s.lastIndexOf(ch);
int k     = s.lastIndexOf(ch, fromPos);
int k     = s.lastIndexOf(str);
int k     = s.lastIndexOf(str, fromPos);
```

```
String s2      = s.trim();
String s2 = s.replace(oldChar, newChar);
String s2 = s.toUpperCase();
String s2 = s.toLowerCase();
```

This convention goes back to C, where elements of arrays are counted from 0. So the first character of a string is at position (or index) 0, and the last one is at position `s.length() - 1`. For example:

```
String s = "Internet";
char c1 = s.charAt(0);        // c1 gets the value 'I'
char c2 = s.charAt(7);        // c2 gets the value 't'
```

If you call `charAt(pos)` with `pos` less than 0 or `pos` greater than or equal to the string length, the method will throw a `StringIndexOutOfBoundsException`.

Always make sure that when you refer to the positions of characters in strings, they fall in the range from 0 to string length – 1.

Substrings

The `String` class has two (overloaded) `substring` methods. The first one, `substring(fromPos)`, returns the tail of the string starting from `fromPos`. For example:

```
String s = "Internet";
String s2 = s.substring(5);   // s2 gets the value "net"
```

The second one, `substring(fromPos, toPos)` returns the segment of the string from `fromPos` to `toPos - 1`. For example:

```
String s = "Internet";
String s2 = s.substring(0, 5);     // s2 gets the value "Inter"
String s3 = s.substring(2,6); // s3 gets the value "tern"
```

Note: the second parameter is the position of the character following the substring, and that character is <u>not</u> included into the returned substring. The length of the returned substring is always `toPos - fromPos`.

Concatenation

The `concat` method concatenates strings; it works exactly the same way as the string version of the + operator. For example:

```
String s1 = "Sun";
String s2 = "shine";
String s3 = s1.concat(s2);    // s3 gets the value "Sunshine"
String s4 = s1 + s2;          // s4 gets the value "Sunshine"
```

The += operator concatenates the operand on the right to the string on the left. For example:

```
String s = "2*2 ";
s += "= 4";     // s gets the value "2*2 = 4"
```

It may appear at first that the += operator violates the immutability of strings. This is no so. The += first forms a new string concatenating the right-hand operand to the original `s`. Then it changes the reference `s` to point to the new string. The original string is left alone if some other variable refers to it, or thrown away. So `s += s2` may be as inefficient as `s = s + s2`.

As was stated earlier, you can also concatenate characters and numbers to strings using the + and += operators, as long as the compiler can figure out that you are working with strings, not numbers. For example:

```
String s = "Year: ";
s += 1776;          // s gets the value "Year: 1776";
```

But if you write

```
String s = "Year:";
s += ' ' + 1776;    // space in single quotes
```

it won't work as expected because neither `' '` nor `1776` is a `String`. Instead of concatenating them it will first add 1776 to the Unicode cord for a space (32) and then append the sum to `s`. So `s` would get the value "`Year:1808`". On the other hand,

```
String s = "Year:";
s += " " + 1776;    // space in double quotes
```

does work, because the result of the intermediate operation is a `String`.

Finding characters and substrings

The `indexOf(Char c)` method returns the position of the first occurrence of the character `c` in the string. Recall that indices are counted from 0. If `c` is not found in the string, `indexOf` returns –1. For example:

```
String s = "Internet";
int pos1= s.indexOf('e');     // pos1 gets the value 3
int pos2 = s.indexOf('x');    // pos2 gets the value -1
```

You can also start searching from a position other than the beginning of the string by using another (overloaded) version of `indexOf`. It has a second parameter, the position from which to start searching. For example:

```
String s = "Internet";
int pos = s.indexOf('e', 4);  // pos gets the value 6
```

You can search backward starting from the end of the string or from any other specified position using one of the two `lastIndexOf` methods for characters. For example:

```
String s = "Internet";
int pos1 = s.lastIndexOf('e');          // pos1 gets the value 6
int pos2 = s.lastIndexOf('e', 4);       // pos2 gets the value 3
int pos3 = s.lastIndexOf('e', 2);       // pos3 gets the value -1
```

`String` has four similar methods that search for a specified substrings rather than a single character. For example:

```
String 2 = "Internet", s2 = "net";
int pos1 = s.indexOf("e");              // pos1 gets the value 3
int pos2 = s.indexOf("net");            // pos2 gets the value 5
int pos3 = s.indexOf(s2, 6);            // pos3 gets the value -1
int pos4 = s.lastIndexOf(s2);           // pos4 gets the value 5
int pos5 = s.lastIndexOf("net", 6);     // pos 5 get the value 5
```

Comparisons

You cannot use relational operators (==, !=, <, >, <=, >=) to compare strings.

Recall that relational operators `==` and `!=` when applied to objects compare the objects' references (that is, their addresses), not their values. Strings are no exception. The `String` class provides the `equals`, `equalsIgnoreCase`, and `compareTo` methods for comparing strings. `equals` and `equalsIgnoreCase` are `boolean` methods; they return `true` if the strings have the same length and the same characters (case-sensitive or case-blind, respectively), `false` otherwise. For example:

```
String s = "OK!";
boolean same1 = s.equals("Ok!");        // same1 is set to false
boolean same2 = s.equals(OK");          // same2 is set to false
boolean same3 = s.equalsIgnoreCase("Ok!");  // same3 is set to true
```

Occassionally the string in the comparison may not have been created yet. If you call its `equals` method (or any other method) you will get a `NullPointerException`. For example:

```
private String name;                // name is an instance variable
…
     boolean same = name.equals("Sunshine");
          // NullPointerException if name has not been initialized
```

To avoid errors of this kind you can write

```
boolean same = (name != null && name.equals("Sunshine"));
```

The below statement always works due to short-circuit evaluation. However, real Java pros may write

```
boolean same = "Sunshine".equals(name);
```

This always works, whether `name` is initialized or `null`, because you are not calling methods of an uninitialized object. The same applies to the `equalsIgnoreCase` method.

The `compareTo` method returns an integer that describes the result of a comparison. `s1.compareTo(s2)` returns a negative integer if `s1` lexicographically precedes `s2`, `0` if they are equal, and a positive integer if `s1` comes later than `s2`. (To remember the meaning of `compareTo`, you can mentally replace "compareTo" with a minus sign.) The comparison starts at the first character and proceeds until different characters are encountered in corresponding positions or until one of the strings ends. In the former case, `compareTo` returns the difference of the Unicode codes of the characters, so the string with the first "smaller" character (that is, the one with the smaller Unicode code) is deemed smaller; in the latter case `compareTo` returns the difference in lengths, so the shorter string is deemed smaller. This is called "lexicographic ordering," but it is not exactly the same as used in a dictionary because `compareTo` is case-sensitive, and uppercase letters in Unicode come before lowercase letters. For example:

```
String s = "ABC";
int result1 = s.compareTo("abc");
     // result1 is set to a negative number:
     //        "ABC" is "smaller" than "abc"

int result2 = s.compareTo("ABCD");
     // result2 is set to a negative number:
     //   "ABC" is "smaller" than "ABCD"
```

Naturally, there is also a `compareToIgnoreCase` method.

Conversions

Other useful String method calls include:

```
String s2 = s1.toUpperCase();
     // s2 is set to a string made up of the characters
     // in s1 with all letters converted to the upper case

String s2 = s1.toLowerCase();
     // same for lower case

String s2 = s1.replace(c1, c2);
     // s2 is set to a string that has the same characters
     // as s1, except all occurrences of c1 are replaced with c2

String s2 = s1.trim();
     // s2 is set to the same string as s1, but with the
     // "whitespace" characters (spaces, tabs, and newline
     // characters)trimmed from the beginning and end of the string
```

For example:

```
String s1 = " <u>String Methods</u>   ";

String s2 = s1.trim();   // s2 becomes "<u>String Methods</u>"
                         // s1 remains " <u>String Methods</u>   "

String s3 = s2.toUpperCase();
                         // s3 becomes "<U>STRING METHODS</U>"
                         // s2 remains "<u>String Methods</u>"

String s4 = s3.replace('U', 'B')
                         // s4 becomes "<B>STRING METHODS</B>"
                         // s3 remains "<U>STRING METHODS</U>"
```

None of these methods (nor any other `String` methods) change the `String` object for which they are called. Instead, they build and return a new string.

This is a potential source of tricky bugs. The names of these methods might imply that they change the string, and it is easy to call them but forget to put the result anywhere. For example:

```
String s1 = " <code>   ";
s1.trim();     // A useless call: s1 remains unchanged!
               // You probably meant s1 = s1.trim();
```