CS 1110, LAB 4: WRITING STATIC METHODS

Name: ______________________________ **Net-ID**: _______________

There is an online version of these instructions at

http://www.cs.cornell.edu/courses/cs1110/2012sp/labs/lab04.php

You may wish to use that version of the instructions.

The purpose of this lab is to give you practice with developing the bodies of methods. At the same time, this lab will give you practice with Strings. We begin with some information on Strings. We also introduce you to the equality comparison operator `==` and its counterpart, the function `equals()`. After this lab, you should study Section 5.2 of the text, beginning on page 175.

Requirements For This Lab. The very first thing that you should do in this lab is to download the file Methods.java from the course web page:

http://www.cs.cornell.edu/courses/cs1110/2012sp/labs/lab04/Methods.java

You will note that this is a class filled with static function *stubs*. A function stub is a function with nothing more than a single `return` statement that gives back a (wrong) answer. Stubs are used to ensure that the file compiles while you work on filling in the details. For each stub, you should are going to add several lines of code, as well as a proper return statement, in order to satisfy the specification.

As with the previous lab, this lab will involve three different components. First, you will need to write answers to written questions on these sheet. Second, you will need to modify the contents of `Methods.java` to fill in the function stubs. Finally, you will need to make a JUnit test case. When you are done, you should show *all three* to your lab instructor, who will record that you did it. You do not need to submit the actual paper, and you do not need to submit the computer files anywhere.

This lab is a bit more work than the previous labs. Hopefully you have the process down now so that you can finish the lab within your section; it is certainly doable. However, if you do not finish during the lab, you have **until the beginning of lab next week to finish it**. You should always do your best to finish during lab hours; remember that labs are graded on effort, not correctness.

---

1. Warm Up Exerises

String Objects. A String object (instance or folder) associates a number with each character in its list of characters. The number is called the index or position of the character. Type the following line into the Interactions pane of Dr. Java:

```
String s = "Java is fun.";
```

The String object `s` now contains the list of characters `"Java is fun."` The index of each character is shown below:

```
Index:      0  1  2  3  4  5  6  7  8  9  10  11
Character:  J  a  v  a     i  s     f  u  n    .
```

Note that the index of the first character is 0 (not 1) and that the period and each of the space characters between each of the words have their own index.

In the string `"I will study every day."`, what is the index of the character 'w'?

Of the last space character?

Integer Static Methods. You have already seen how to turn an integer into a String; you just concatenate (+) the number with the empty String "". Often, you find yourself wanting to do the reverse, however. If a String s contains only digits (not even blanks), then the function call

```
Integer.parseInt(s)
```

yields the integer represented by s. For example, `Integer.parseInt("345")` is 345.

What happens when you type `Integer.parseInt("text")` into DrJava?

Equality. The symbol == is used for equality testing. You know 2+3 == 5 has the value true. More importantly, if x and y are class variables, the test x == y is made on the names (on the folder tabs) of the object. Hence the following expression is always false because it is between two objects with different names:

```
new C(args) == new C(args)
```

Evaluate the following expressions in the Interactions pane and write down their values. In the last one, note that in each occurrence of `"ab"` Java creates a new object of class String (even though we do not use the keyword `new` in this case).

Expression	Value
`new String("ab") == new String("ab")`	
`"ab" == new String("ab")`	
`"ab" == "ab"`	

The class string has a function `equals(Object)`, which is an alternative to ==. This compares two String objects by the text inside of the double quotes, and not by the folder tab name. To see this, try the following in the interactions pane and write down the values:

Expression	Value
`new String("ab").equals(new String("ab"))`	
`"ab".equals(new String("ab"))`	
`"ab".equals("ab")`	

It is very important that you understand the distinction between == and function `equals`. Read about equality of strings on page 179 of the text, and equality testing on page 118.

2. The String API

For the rest of this lab, you will need to make use of functions that found in the String API:

`http://docs.oracle.com/javase/6/docs/api/java/lang/String.html`

For convenience, here is a list of the most useful String functions.

s.length()	Yields: length of `s`, that is, the number of characters in it. It can be 0. `"abc".length() is 3`
s.charAt(i)	Yields: character at index `i` of String `s`, which we might write as `s[i]`. The result is of type char.
s.substring(b,e)	Yields: the String `s[b..e-1]` (i.e. the chars `s[b]`, `s[b+1]`, ..., `s[e-1]` . `"abc".substring(1,3)` is `"bc"`
s.substring(b)	Yields: the String `s[b..]`, or `s[b..s.length()-1]`. `"abc".substring(1)` is `"bc"`
s.indexOf(s1)	Yields: index of the first char of the FIRST occurrence of String `s1` in `s` (-1 if `s1` does not occur in s). `"abbc".indexOf("b")` is 1
s.indexOf(c)	Yields: index of FIRST occurrence of char `c` in `s` (-1 if `c` does not occur in `s`). `"abbc".indexOf('b')` is 1
s.lastIndexOf(s1)	Yields: index of first char of the LAST occurrence of String `s1` in `s` (-1 if `s1` does not occur in `s`) . `"abbc".lastIndexOf("b")` is 2
s.trim()	Yields: a copy of `s` but with any preceding and ending whitespace removed. `" abbc ".trim()` is `"abbc"`
s.startsWith(s1)	Yields: "`s` begins with String `s1`" (i.e. true if `s` begins with `s1` and false otherwise).
s.endsWith(s1)	Yields: "`s` ends with String `s1`". `"abbc".endsWith("c")` is true
s.equals(s1)	Yields: true if `s` and `s1` contain the same sequence of characters (i.e. the same strings).
s.compareTo(s1)	Yields: negative, 0 , or positive, depending on whether `s` is less than, equal to, or greater than `s1`. The comparison is based on alphabetic ordering, as in the dictionary. `"abbc".compareTo("a")` is 3, `"abbc".compareTo("abbcdb")` is -2

---

3. Writing Time Functions

The file `Methods.java` contains specifications for a bunch of functions for you to write. The function bodies have *stub* return statements so that the class will compile. Write the bodies of as many of them as you can in this lab. You probably won't finish them all. We only ask that that you **finish three of them** during the lab; show them to your lab instructor at the end of the lab. How many of the others you do is up to you. The more you practice, the easier developing such programs will become.

The methods will change a time in a String into a different format. Time comes in four formats:

24-hour-string:	"<hours>:<minutes>" <hours> is in 0..23 and <minutes> is in 0..59. **Examples**: "4:20" "13:0" "23:59" "0:0"
AM-PM-string	"<hours>:<minutes>AM" or "<hours>:<minutes>PM" <hours> is in 0..11 and <minutes> is in 0..59. **Examples**: "4:20AM" "1:0PM" "11:59PM" "0:0AM"
24-hour-verbose:	**Example**: "1 hours and 20 minutes". **Example**: "23 hours and 59 minutes" Note that the answer is not grammatically correct. Also, there is exactly one blank between each of the pieces.
24-hour-correct:	Exactly like the 24-hour-verbose format except that it is grammatically correct. So, instead of "1 hours and 20 minutes" it reads "1 hour and 20 minutes" and instead of "0 hours and 1 minutes" it reads "0 hours and 1 minute".

3.1. **JUnit Testing.** In addition to working on the methods above, you should create a JUnit test class as usual. The purpose of the test class is to verify that the functions you choose to work on are correct. For this exercise is okay if you write one test procedure per function (in `Method.java`); it is easier to keep the tests separate that way.

In general recommend that you write your tests before starting on the next part. In particular, if you are just going to work on three functions, you should pick those three functions and then write the test for the functions. That way, the only thing that you have to do to debug a function is to press the "Test" button while you work on it.

Once you are done with at least three functions, you should show both `Methods.java` and your JUnit test to your lab instructor. If you do not have time to finish three in the allotted time, then show the completed lab to your instructor the next week. If you need help, ask for it; do not waste time.