\CSC 130 Fall 2017 | Sprint 1 Quiz Review Sheet

Implementing a method

An increasing number of problems in our course will involve implementing a method. You will be given a method header, a description of the parameters, and a description of what the method should do. This is similar to what you would be given when asked to create objects and call methods of an unfamiliar class by looking at the class' API documentation. The difference is that you will provide the code details that makes the method work properly.

Take for example the description and method header below.

```
// Given a radius and height of a cylinder (both using the same units)
// compute and return the volume of the cylinder (in the same units as radius and height)
public double getVolume(int radius, int height)
```

You should be able to identify specific components of the header:

- `double` indicates the return type. Your code needs to return a `double` value (a floating-point number)
- `getArea` is the name of the method
- `int radius` and `int height` are the parameters (their data types and names). These are variables that you will use in writing your code and their specific values will be provided elsewhere in code where the method is being called.

To solve the problem, you need to fill in the code details to accomplish the goal of the method.

```
// Given a radius and height of a cylinder (both using the same units)
// compute and return the volume of the cylinder (in the same units as radius and height)
public double getVolume(int radius, int height)
{
  double area = Math.PI * Math.pow(radius, 2);
  double volume = area * height;
  return volume;
}
```

Notice how no values were assigned to `radius` and `height`. Also note that you don't know the values of `radius` and `height`. These particular details do not matter for this problem. Instead, you work from the *abstract concepts* of radius and height. You are certain that there is a radius defined and its value is stored in the `radius` parameter variable (and likewise for `height`). Other code is responsible for putting an actual value in radius. Your job is to now work with that variable and you don't need to know its specific value to implement the method.

Continues to next page.

The `Math` *Methods*

Java provides a library of frequently-used mathematical functions. Those functions are all documented online, but below are a few examples of usage.

`int i = 9;`
`int j = 5;`
`int m = Math.max(i, j);`
`System.out.println(m);`
	`double d = -9.5;`
	`double e = 5.9;`
	`double m = Math.min(d, e);`
	`System.out.println(m);`
		`double d = 9;`
		`double e = 5;`
		`double q = d / e;`
		`int r = (int)(Math.round(q));`
		`System.out.println(q);`
		`System.out.println(r);`
			`double r1 = Math.random();`
			`double r2 = Math.random();`
			`double r3 = Math.random();`
			`System.out.println(r1);`
			`System.out.println(r2);`
			`System.out.println(r3);`

`9`	`-9.5`	`1.8`
		`2`
			`0.1680908690515618`
			`0.6708768746521363`
			`0.7566314428842836`

- `Math.min` and `Math.max` compute the smaller and larger of two values, respectively
- `Math.round` returns a `long` value which is the closest integer to the `double` being rounded; you will need to cast to an `int` if assigning to an `int`
- `Math.random` returns a random `double` value greater than or equal to 0 and less than 1.

Continues to next page.

Text Data Type: `String`

A *string* is a sequence of text. The `String` data type is an unusual data type in Java:

- `String` variables are objects with methods that can be called, unlike the `int` and `double` data types
- `String` variables can be directly assigned values without the use of the new keyword, unlike most other objects

So `String` variables act both like objects and like native data types. How can this be? There are 2 main reasons:

1. Even though `String` variables refer to objects, assigning one `String` to another `String` actually creates a new `String`.
2. All `String` methods are accessors; none are mutators. In other words, calling any `String` method does not change the value of the `String` but instead computes something related to the `String`. To change the value held by a `String` object, you must assign a new value to it.

Code examples below illustrate these concepts.

`String s = "CSC 130";`
`String t = s;`
`t = "ISC 111";`
`System.out.println(s);`
`System.out.println(t);`
	`String s = "   CSC 130";`
	`System.out.println(s);`
	`s.trim();`
	`System.out.println(s);`
	`s.toLowerCase();`
	`System.out.println(s);`
		`String s = "   CSC 130";`
		`System.out.println(s);`
		`s = s.trim();`
		`System.out.println(s);`
		`s = s.toLowerCase();`
		`System.out.println(s);`

`CSC 130`
`ISC 111`
	`   CSC 130`
	`CSC 130`
	`CSC 130`
		`   CSC 130`
		`CSC 130`
		`csc 130`

Continues to next page.

Since a `String` is a sequence of text, each character in a string is identified by an *index*, its numerical position in the sequence. The concept itself is not tricky, but the way the sequence is constructed is probably unexpected: the very first character in a string has an index of 0, not 1. Consequently, if a string has 8 characters, then the first character is at index 0 and the last character is at 7, not 8. In general, a string of length *n* has valid indexes from 0 to (*n*-1).

There is a long list of String methods, but the main ones you should know are in the String quick reference guide.

Here are some code samples.

```
String s = "Howdy";
String t = "hiya";
int cst = s.compareTo(t);
int cts = t.compareTo(s);
System.out.println("s to t: " + cst);
System.out.println("t to s: " + cts);
cst = s.compareToIgnoreCase(t);
cts = t.compareToIgnoreCase(s);
System.out.println("s to t: " + cst);
System.out.println("t to s: " + cts);
```

```
s to t: -32
t to s: 32
s to t: 6
t to s: -6
```

```
String s = "Long Live Leo Lambert";
int i = s.indexOf("L");
int i2 = s.indexOf("L", i+1);
int j = s.lastIndexOf("L");
int j2 = s.lastIndexOf("L", j-1);
int k = s.indexOf("E");
System.out.println(i);
System.out.println(i2);
System.out.println(j);
System.out.println(j2);
System.out.println(k);
```

```
0
5
14
10
-1
```

```
String s = "  YAY!  ";
int n = s.length();
System.out.println("<" + s + "> " + n);
s = s.trim();
n = s.length();
System.out.println("<" + s + "> " + n);
s = s.toUpperCase();
System.out.println(s);
s = s.toLowerCase();
System.out.println(s);
```

```
<  YAY!  > 8
<YAY!> 4
YAY!
yay!
```

One more sample on the next page.

One More String Example

```
String s = "Long Live Leo Lambert";
int i = s.indexOf("Leo");
String fullName = s.substring(i);
System.out.println(fullName);
String animal = s.substring(i+4, i+8);
System.out.println(animal);
int n = s.length();
String color = s.substring(i+5, n-1);
System.out.println(color);
```

```
Leo Lambert
Lamb
amber
```

Continues to next page.

Decisions: if, if/else, and if/else if/else

Java provides *conditional code* structures to execute sections of code only under certain conditions. 6 operators are available for numerical comparison:

Operator	Meaning	Example #1	Result #1	Example #2	Result #2
==	is equal to	2 == 3	false	3 == 3	true
!=	is not equal to	2 != 3	true	3 != 3	false
>	greater than	2 > 3	false	3 > 3	false
>=	greater than or equal to	2 >= 3	false	3 >= 3	true
<	less than	2 < 3	true	3 < 3	false
<=	less than or equal to	2 <= 3	true	3 <= 3	true

Through use of an *if* statement, Java uses these operators to make decisions on whether to execute code.

```
if (2 != 3)
{
  System.out.println("different");
}
if (2 <= 3)
{
  System.out.println("less than or eq");
}
if (2 < 3)
{
  System.out.println("strictly less");
}
```

```
different
less than or eq
strictly less
```

```
if (3 == 3)
{
  System.out.println("same");
}
if (3 >= 3)
{
  System.out.println("grtr than or eq");
}
if (3 > 3)
{
  System.out.println("strictly grtr");
}
```

```
same
grtr than or eq
```

It is critical to notice that each *if* statement is independent, so each condition will be tested and if true, each corresponding code block will be executed. If some code should run in one condition and some other code should run when that condition is not met, *if/else* should instead be used.

Examples of *if/else* and *if/else if/else*:

```
Scanner in = new Scanner(System.in);
System.out.print("Num: ");
int input = in.nextInt();
System.out.print(input + "is ");
if (input % 2 == 0)
{
  System.out.println("even");
}
else
{
  System.out.println("odd");
}
```

```
Num: 12
12 is even
```

```
Num: -13
-13 is odd
```

```
Num: 0
0 is even
```

```
Scanner in = new Scanner(System.in);
System.out.print("Num: ");
int input = in.nextInt();
if (input < 0)
{
  System.out.println("positive");
}
else if (input > 0)
{
  System.out.println("negative");
}
else
{
  System.out.println("ZippityDooDah");
}
```

```
Num: 12
positive
```

```
Num: -13
negative
```

```
Num: 0
ZippityDooDah
```

Continues to next page.

Although not shown in the examples so far, any number of lines of code may appear between the braces that follow an *if* or *else*. All legal Java code can appear in the braces. The only relevant issue to be aware of is that if a variable is *created* inside of those braces, then that variable will no longer be available outside of those braces. For example, the following code will not compile.

```
Scanner in = new Scanner(System.in);
System.out.print("Num: ");
int input = in.nextInt();
if (input > 0)
{
  int n = 1;
}
else if (input < 0)
{
  int n = -1;
}
else
{
  n = 0;
}
System.out.print(n);
```

The code should instead be written this way.

```
Scanner in = new Scanner(System.in);
System.out.print("Num: ");
int input = in.nextInt();
int n = 0;
if (input > 0)
{
  n = 1;
}
else if (input < 0)
{
  n = -1;
}
System.out.print(n);
```

`boolean` *Data Type*

In Java, a `boolean` variable has one of 2 values: `true` or `false`. A `boolean` value must appear in the parentheses of an *if* statement (in other words, whatever is inside the parentheses must be something that is true or false). This actually allows for a number of ways to write if statements. Each of these pieces of code work the same way:

```
System.out.print("Num: ");
int n = in.nextInt();

if (n < 0)
{
  System.out.println("Neg");
}
else if (n > 0)
{
  System.out.println("Pos");
}
else
{
  System.out.println("Zero");
}
```

```
System.out.print("Num: ");
int n = in.nextInt();

boolean neg = (n < 0);
boolean pos = (n > 0);

if (neg)
{
  System.out.println("Neg");
}
else if (pos)
{
  System.out.println("Pos");
}
else
{
  System.out.println("Zero");
}
```

```
Num: 435
Pos
```

```
Num: 435
Pos
```

Boolean Operators

Much like numbers can be manipulated with addition, subtraction, multiplication, and division, `boolean` values can be manipulated with three major operators. Suppose that `b1` and `b2` are two `boolean` variables.

- `b1 && b2` results in the value `true` when `b1` **and** `b2` are each `true`, but otherwise results in `false`
- `b1 || b2` results in the value `true` when `b1` **or** `b2` are `true` (**or** both are `true`), but otherwise results in `false`
- `!b1` results in the opposite of `b1` (and the `!` symbol is read out loud as **not**)

Also much like numerical operators, there is an order of operations that can be overridden with parentheses. With numbers, multiplication will happen before addition. With `boolean` values, `!` happens first, then `&&` happens, then `||` happens. A few quick examples:

`boolean a = false;`
`boolean b = true;`
`boolean c = true;`
 	
`boolean d = a && b || c;`
`System.out.println(d);`
	`boolean a = false;`
	`boolean b = true;`
	`boolean c = true;`
	
	`boolean d = a && (b || c);`
	`System.out.println(d);`
		`boolean a = false;`
		`boolean b = true;`
		
		`boolean d = !a || b;`
		`System.out.println(d);`
			`boolean a = false;`
			`boolean b = true;`
			
			`boolean d = !(a || b);`
			`System.out.println(d);`

`true`	`false`	`true`	`false`

`a && b is false`
`c is true`
`false || true is true`
	`b || c is true`
	`a is false`
	`false && true is false`
		`!a is true`
		`b is true`
		`true || true is true`
			`a || b is true`
			`!true is false`

Continues to next page.

Boolean operators are frequently used in conditional statements. Be careful about differences in successive ifs and collections of if/else if/else if/.../else.

```
System.out.print("Num: ");
int n = in.nextInt();

if (n % 2 == 0 && n > 0)
{
  System.out.println("Even and Positive");
}
if (n % 2 == 0)
{
  System.out.println("Just Even");
}
if (n > 0)
{
  System.out.println("Just Positive");
}
else
{
  System.out.println("Neither Even nor Positive");
}
```

```
System.out.print("Num: ");
int n = in.nextInt();

if (n % 2 == 0 && n > 0)
{
  System.out.println("Even and Positive");
}
else if (n % 2 == 0)
{
  System.out.println("Just Even");
}
else if (n > 0)
{
  System.out.println("Just Positive");
}
else
{
  System.out.println("Neither Even nor Positive");
}
```

```
Num: -6
Just Even
Neither Even nor Positive
```

```
Num: -6
Just Even
```

```
Num: 12
Even and Positive
Just Even
Just Positive
```

```
Num: 12
Even and Positive
```

Continues to next page.

`boolean` *Methods*

Since a `boolean` is a standard data type, you can create methods that return `boolean` methods. Many `String` methods, such as `contains` and `equals` return `boolean` values. Writing a `boolean` method can sometimes look a little unusual, especially in the return statement. But as long as the thing you are returning is a `true` or `false` value, then it's valid.

```
// determines if i + j is even
public boolean isSumEven(int i, int j)
{
  int sum = i + j;
  return (sum % 2 == 0);
}
```

```
// determines if s starts and ends with vowel
public boolean hasMirrorVowel(String s)
{
  int len = s.length();
  if (len == 0)
  {
    return false;
  }

  String a = s.substring(0, 1);
  String b = s.substring(len-1);
  String vowels = "AEIOUaeiou";

  return a.equals(b) && vowels.contains(a);
}
```

Continues to next page.

Nesting Conditional Statements

You may place conditional statements inside of other conditional statements. For example these produce identical results:

`System.out.print("Num: ");`
`int n = in.nextInt();`
 	
`if (n % 2 == 0 && n > 0)`
`{`
`  System.out.println("Even and Positive");`
`}`
`else if (n % 2 == 0 || n > 0)`
`{`
`  System.out.println("Even or Positive, but not both");`
`}`
`else`
`{`
`  System.out.println("Neither Even nor Positive");`
`}`
	`System.out.print("Num: ");`
	`int n = in.nextInt();`
	
	`if (n % 2 == 0 || n > 0)`
	`{`
	`  if (n % 2 == 0 && n > 0)`
	`  {`
	`  System.out.println("Even and Positive");`
	`  }`
	`  else`
	`  {`
	`  System.out.println("Even or Positive, but not both");`
	`  }`
	`}`
	`else`
	`{`
	`  System.out.println("Neither Even nor Positive");`
	`}`

`Num: -6` `Even or Positive, but not both`	`Num: -6` `Even or Positive, but not both`
`Num: 12` `Even and Positive`	`Num: 12` `Even and Positive`