Part I: Arrays Basics #2:

This discussion is related to class code set12A. You should have that code open as you read through this.

The first line of the program:

```
    int [ ] numbers = { 3, 5, 8, 1, 3, -2, 4, 11, 7, 6 };
```

declares and initializes the array. Building arrays like this must be done globally in one statement or it will not compile. This line of code builds an array with a length of 10. The field, **numbers.length** stores the value **10** which is the number of elements in the array or the size of the array.

The pattern we follow for much of the array work involves using a **for** loop to get to each value stored in the array. This is called traversing the array. Unless specified otherwise, we often begin with the element **[0]** and move to element **[length-1]**. However, this is a pattern and some requirements (as seen in the fourth example below) require starting and/or stopping at other elements of the array.

The first example in the code prints the array on a single line. Here is the function:

```
void printArray( )
{
    println("Values in the array:");
   for ( int i = 0 ; i < numbers.length ; i++ )
   {
      print( numbers[i] + "  " );
   }
   println( );
}
```

Let's trace the execution of the for loop:

```
for ( int i = 0 ; i < numbers.length ; i++ )
```

Value of i	Evaluation of test: i < numbers.length (*length's value is 10*)	Array element being visited	Value of array element being visited	What the visit does with the value
0	true	[0]	3	Print it
1	true	[1]	5	Print it
2	true	[2]	8	Print it
3	true	[3]	1	Print it
4	true	[4]	3	Print it
5	true	[5]	-2	Print it
6	true	[6]	4	Print it
7	true	[7]	11	Print it
8	true	[8]	7	Print it
9	true	[9]	6	Print it
10	false	🟥	🟥	🟥

This rather verbose way of tracing the execution of the **for** *loop shows how Processing uses the* **for loop variable, i** *to access or "visit" each element of the array. The word visit means that we do something with the value stored in the array. The "something" we do is specified in the problem we are solving. The specification here was to print each element.*

The second function in the code has the task of computing and returning the average back to the **draw( )** *function where it was called so it could be printed.*

```
float getAverage( )
{
   float sum = 0;
   for ( int i = 0 ; i < numbers.length ; i++ )
   {
      sum = sum + numbers[i] ;
   }
   return sum/numbers.length;
}
```

This function is not a **void** *function because it must return the average to* **draw( )** *for printing. The reason the function returns a* **float** *is because a fractional answer is often a better representation of an average. The average of the values* **1** *and* **2** *is best represented by the value* **1.5**. *If we use* **int** *values to do this we get a different result:*

2 / 1 → 1

*Remember the rules of integer division – the result must be i***int**.
Here is a similar tracing of the **for** *loop:*

for (int i = 0 ; i < numbers.length ; i++)

Value of i	Evaluation of test: i < numbers. length	Array element being visited	Value of array element being visited	What the visit does with the value	Value of local variable sum
🟥	🟥	🟥	🟥	🟥	0.0
0	true	[0]	3	Add it to sum	3.0
1	true	[1]	5	Add it to sum	8.0
2	true	[2]	8	Add it to sum	16.0
3	true	[3]	1	Add it to sum	17.0
4	true	[4]	3	Add it to sum	20.0
5	true	[5]	-2	Add it to sum	18.2
6	true	[6]	4	Add it to sum	22.0
7	true	[7]	11	Add it to sum	33.0
8	true	[8]	7	Add it to sum	40.0
9	true	[9]	6	Add it to sum	46.0
10	false	🟥	🟥	🟥	🟥

The third example prints values in the array that are greater than the average:

```
void  printValuesGreaterThanAverage( float average )
{
   println("Values greater than the average of " + average + ": ");
   for ( int i = 0 ; i < numbers.length ; i++ )
   {
     if ( numbers[i] > average)
    {
       print( numbers[i] + "  " );
    }
   }
   println( );
}
```

Here is a another trace of the for loop:

```
for ( int i = 0 ; i < numbers.length ; i++ )
```

Value of i	Evaluation of test: i < numbers. length	Array element being visited	Value of array element being visited	Evaluation of test: numbers[i] > average The average is 4.6	What this visit does with value
0	true	[0]	3	false	nothing
1	true	[1]	5	true	print it
2	true	[2]	8	true	print it
3	true	[3]	1	false	nothing
4	true	[4]	3	false	nothing
5	true	[5]	-2	false	nothing
6	true	[6]	4	false	nothing
7	true	[7]	11	true	print it
8	true	[8]	7	true	print it
9	true	[9]	6	true	print it
10	false	🟥	🟥	🟥	🟥

The fourth example uses a different pattern in the **for** *loop. The* **for** *loop must begin at element* **[1]** *and stop at element* **[numbers.length-1]**. *The reason is the task specified in the name of the function:*

printValuesGreaterThanBothNeighbors( )

To understand the task we have to define the term neighbor. For this task a neighbor is an the element that is either immediately before and after an array element. For element [3], the neighbors are elements [2] and [4]. Note that not all elements have two neighbors. Element [0] and and element [length-1] have only one neighbor(elements [1] and [length-2] repsectively). This function must print only those elements that have values greater than both neigbors. Given the above, lets think about this... In order to "qualify" for printing, an element must have two neighbors. Elements **[0]** *and* **[numbers.length]** *do not have two neighbors. If our code tries to visit the element before element* **[0]** *or after element* **[numbers.length]**, *the program will crash. So we have to alter the pattern of the* **for** *loop. Here is the function definition:*

```
void printValuesGreaterThanBothNeighbors( )
 {
   println("Values greater than both neighbors: ");
   for ( int i = 1 ; i < numbers.length-1 ; i++ )
   {
     if ( numbers[i] > numbers[i-1] &&
          numbers[i] > numbers[i+1])
    {
       print( numbers[i] + "  " );
    }
   }
   println( );
 }
```

For practice, we will let you trace the execution of the for loop in this code:

Here is the array again so you do not have to flip back and forth:

```
int [ ] numbers = { 3, 5, 8, 1, 3, -2, 4, 11, 7, 6 };
```

```
for ( int i = 1 ; i < numbers.length-1 ; i++ )
```

Value of i	Evaluation of test: i < numbers. length	Array element being visited	Value of array element being visited	Evaluation of test: numbers[i] > numbers[i-1] && numbers[i] > numbers[i+1]	What this visit does with value