CSci 1103 Fall 2018 Lab 7 – Methods

Lab 7 will include:

- two-dimensional Arrays
- methods
- constants

Please refer to the **Lab Procedures for the Semester** in the Lab 1 writeup as they pertain to all labs for this course.

At any point, if you have questions, please *ask a TA to help you*. That is what they are here to do.

Step 1. Starting Your Own 15-Puzzle Game.

In this lab, you will build a few components of a 15-Puzzle Game. The Canvas Lecture Examples section for Week 7 contains `Fifteen.java`, which includes the code for declaring, initializing and printing an array for a 15-Puzzle. You are encouraged to refer to `Fifteen.java` as you complete this lab as it will serve as a good example, and you may re-use some parts of it. *As always, if you use ideas that are not your own, be sure to credit the source with a comment inside your code.* Additional general information about the 15-Puzzle game can be found at: https://en.wikipedia.org/wiki/15_puzzle

Begin by creating a new directory for this lab within your home directory. Call it `lab7`. Within the `lab7` directory, create a new file called `MyFifteen.java`. Please be careful to differentiate between references to the Canvas code (from lecture) which is called `Fifteen.java` and the version you will be starting in this lab, called `MyFifteen.java`.

Starting with an empty `main()` method, create two *constants*—NROWS and NCOLS. These will be the dimensions of your puzzle. (In lecture, we assumed a 4X4 grid and the code presented in `Fifteen.java` also is written for a 4X4 grid.) The constants, `NROWS` and `NCOLS`, can be used to easily adjust your program to various size grids in the future. Constant declarations are created like variable declarations except that they are preceded by the keyword "`final`," and they are usually given their values at declaration. Unlike variables, constants can*not* be changed later in your program once they are set. By convention, constant identifiers are all upper case. This lets a reader of your program know that these values will not change throughout the program, and it distinguishes constants from variables.

Now, you can declare your puzzle array using the constants NROWS and NCOLS as the dimensions in place of the literal values 4 and 4 used in `Fifteen.java` and in lecture. If you were to decide to make a different size puzzle in the future (of course it would not be a 15-Puzzle anymore), it will be a simple as editing the constant declarations to reflect the new sizes. The code you write in `main()` for this step should look something like this:

```
final int NROWS = 4;
final int NCOLS = 4;
int[][] puzzle = new int[NROWS][NCOLS];
```

Compile this very short beginning of your program to verify that you have the constant and array declarations syntactically correct. Once the program compiles cleanly, have this step checked by a TA before moring on to Step 2.

Step 2. The Right "Method"ology.

The code to initialize the puzzle array to a completed puzzle appears in two ways in the `Fifteen.java`

file on canvas. You may borrow this code (*but, be sure to give credit for it with comments in your program*). Recall the array for a solved 15-Puzzle looks like this:

```
 1   2  3  4
 5   6  7  8
 9  10 11 12
 13 14 15  0
```

You will write the method:

```
static void initSolution(int[][] p)
```

to accept a two-dimensional array of `int`s and set each location in that array to the proper value for a solved 15-Puzzle. That is, the upper left corner should contain 1, the index next to it, should contain 2, and so on. Recall that that methods have their own "scope" meaning that identifiers used in `main()` will not be available in other methods. So, any variables from `main()` that are needed in a method must be explicitly declared within the method *signature* (the first line of the method) and *passed* (or communicated) to the method at the time of call.

When you look at the signature for `initSolution()` above, you can tell that this method will be passed a two-dimesioned array of `int` (and nothing else) because of the formal parameter declaration `int[][] p` within parenthesis. Also, `initSolution()` will not explicitly return anything (indicated by the keyword, `void`, in front of the method signature). Note that the name of the array, the formal parameter, `p`, is strictly internal to `initSolution()` so it does not need to agree with the name of the array passed in at the time of call (the actual argument)—and preferably, the name should *not* agree. In general, the names of arguments and formal parameters will not agree, and if they do agrees, it is often coincidental. The key is that what is declared in a method is constrained to that method. Therefore, identifiers declared in `main()` are available only in `main()` and identifiers declared in another method, such as `initSolution()`, are available only in the method in which they are declared. To share things between methods, values are explicitly passed (or communicated) through the formal parameters.

Your method `initSolution()` can be located either before or after `main()`, but it can*not* be inside `main()` nor can it be outside of a class (in this case My`Fifteen`).

Once you have written your method, call it from within `main()` like this:

```
initSolution(puzzle);
```

You will also need to write some code within `main()` to print out the contents of `puzzle`, so you can verify that `initSolution()` works. This code to print out the puzzle array can be borrowed from `Fifteen.java`, but again, credit the source.

When `MyFifteen.java` compiles and runs correctly using the method `initSolution()`, have a TA check this step.

Step 3. Methods, Methods.

In a manner similar to Step 2 above, write another method:

```
static void printPuzzle(int[][] p)
```

to accept a two-dimensional array of `int`s (just like `initSolution()`), and print the contents in a nice format. This step is primarily "packaging up" the printing code from the end of Step 2 into a method within your program.

Now, replace the code in `main()` to print the contents of the puzzle with a call to `printPuzzle()`.

At this point, your `main()` method should be very short—six lines or so. This is typical of a good Java program. The `main()` method should be relatively short and consist primarily of calls to methods.

Test your code by printing a `puzzle` array that has been initialized with `initSolution()`. Then, try modifying one or more locations in the `puzzle` array you just printed, and print the modified version to verify that `printPuzzle()` works correctly on other puzzle configurations.

Have this step checked by a TA.

Step 4. Sometimes its OK to compare.

What if you wanted to know if a given puzzle represented the correct solution? You would need to compare *each* index in the puzzle against a solved puzzle. In effect, you would want to determine whether a given puzzle is "equal" to the solved puzzle. You may already know that the builtin == operator will *not* accomplish this for us. The == operator simply checks to see if the items on either side are the *same* item, which is fine for numbers, but *not* fine for arrays. We are interested *not* in knowing if two arrays are the *same* array, but rather, we are interested in whether two arrays *contain* the *same information*, even though the arrays are distinct from eachother.

In this step, we will write a method:

```
static boolean puzzleEqual(int[][] a, int[][]b)
```

which will take in two, two-dimensional arrays as arguments and determine whether or not they contain the exact same information. `puzzleEqual()` will return either `true` or `false` depending whether *all* the corresponding elements between the two arrays are the same or not.
Note: this method has a *return value*. In some programming languages, this type of method is called a *function*, because it produces a value which becomes the method's *value* upon call. For example, given two puzzle arrays, `puzzle` and `anotherPuzzle`, the code below uses `puzzleEqual()`to determine whether the two arrays contain exactly the same information—even though the arrays are distinct. Effectively, the call to `puzzleEqual()` within the *if* predicate below, becomes a boolean value (true or false).

```
if (puzzleEqual(puzzle, anotherPuzzle))
  System.out.println("The puzzles match!");
else System.out.println("The puzzles are different");
```

Write this method by creating a nested loop (just like the printing loop in Step 3) that fully traverses both dimensions of one of the arrays. But, instead of printing in the body of the inner loop, check to see if:

```
puzzle[row][col] == anotherPuzzle[row][col]
```

where `row` and `col` are the variables used to control your nested loops.

Each element in the first array will be compared to the corresponding element in the second array. If the two compared values are *not* equal, then immediately `return false;` *Note: if the two compared values <u>are</u> equal, you do not need to do anything, just have the nested loops move onto the next comparison.* If the nested loops finish completely, then no unequal elements were found between the two arrays (otherwise, the method would have finished early by returning `false`). The last line of your method should be:

```
return true;
```

For example, given `puzzle`:

```
 1   2  3  4
 5   6  7  8
 9  10 11 12
13 14 15  0
```

and `anotherPuzzle`:

```
 1   2  3  4
 5   6  7  8
 9  10 11 12
13 14 15  0
```

`puzzleEqual(puzzle, anotherPuzzle)` will return `true`.

But, given `puzzle`:

```
 1   2  3  4
 5   6  7  8
 9  10 11 12
13 14 15  0
```

and `anotherPuzzle` (note that a couple of values have been moved):

```
15  1  3  4
 5  6  7  8
 9 10 11 12
13 14  2  0
```

`puzzleEqual(puzzle, anotherPuzzle)` will return `false`.

To test your `puzzleEqual()` method, you should create two arrays in `main()`, one of them, `puzzle` already exists; the other can be called whatever you like. Initialize both arrays using your method `initSolution()` from above. Then, in `main()`, you can use code like what is in the box above to check to see that the two arrays contain the same information.

Then, to test `puzzleEqual()`on arrays that are not the same, just change one or more of the values in one of the arrays and repeat the test.

Have this step checked by a TA.

That is all for this week. Have a great rest of the week as you prepare to apply what you have done here to implementing the Fifteen-Puzzle as a complete user game!