COMP1006/1406 – Winter 2023

Submit your `.java` files to Gradescope.

The assignment is out of 10 and is worth 10% of your final grade.

The objective of this assignment is to get you up to speed with basic Java syntax (control flow, branching, variables, methods, etc.) with an emphasis on using arrays.

This assignment will completely be graded for correctness.
Be sure that your code compiles and runs. Test your code!

In this assignment, you can use the following classes/methods: `Math`, `String`, `StringBuilder`, `Double` (or any primitive wrapper class), any helper class you create yourself, and static helper method you write yourself.
You can use `System.in` and `System.out` for your own testing.

You are **NOT** allowed to use `Arrays`, `Array`, `ArrayList` (or any JCF class), or `System.arraycopy()`.
Essentially, using any other class that solves your problems for you is forbidden.
Using any of these may result in a grade of zero.

You can use `Arrays.toString()` when testing your code.

Do NOT change the input types, output types or modifiers of the methods you are completing. If you change any of these, it will not compile when submitted.

There is a 48-hour grace period in which submissions will be accepted without penalty. If need be, you can submit this assignment up to Sunday, February 5th, at 11:59pm without any penalty. However, there will not be any office hours and no guarantees that questions will be answered on discord over the weekend. Be sure to start early and submit often.

Note that we will only test your code with **valid** inputs. If a function specification says an input value is a positive integer, for example, only positive integers will be used when testing.

Part One

1 Temperature Conversion [2.5 marks]

In the provided `Convert.java` file, complete the `convert(String, String)` method. The method takes a String and a char as input and outputs (returns) a double.

```
public static double convert(String temperature, char scale){...}
```

The input[1] String `temperature` has both a value and scale and will always be in the form `"x.yS"`, where x.y is a decimal number like 10.1, 0.0, or -15.3. It always has a number before the decimal point (it might be negative), the decimal point itself and at least one digit to the right of the decimal point. So, 10 and .3 are **not** valid. The last character in the this String will either be `C` (for Celsius) or `F` (for Fahrenheit). Examples of valid input are `"10.0C"`, `"-12.29F"`, and `"1010.7C"`. The second input, `scale`, is either the char `'C'` or `'F'`.

The method will take the input `temperature` and convert it to the specified `scale`, outputting (returning) the resulting temperature value.

Here are some examples.

```
Convert.convert("10.0C", 'F')    →  50.0
Convert.convert("10.2C", 'C')    →  10.2
Convert.convert("-40.0C", 'F')   →  -40.0
```

Note that you are using floating point numbers in this problem. Because of this, if you expect your result to be a number like `50.0` and you actually get something like `50.00000000000000020` or `49.99999999999999999787`, don't worry. Remember representational error! However, if you expect `1.0` and you get `1.01` then something went wrong. Our testing will consider two numbers like `1.00000000000000020` and `0.99999999999999999787` as being the same.

If your method is called and the output is less than absolute zero then instead of returning the number you computed, you will instead return the appropriate value of absolute zero (depending on the scale you are converting to). An example, of this special case, would be

```
Convert.convert("-500.0F", 'C')   →  -273.15
```

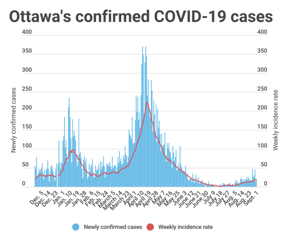
There is a `main()` method that can be used as a *basic* tester for your code. Note that you should add more test cases than the ones provided to ensure your methods are correct. When you submit, we will **not** be running this file as a program. Instead, we will use another program to call your function. But, your `Convert.java` file **must** compile without error to receive marks.

[1]When we test your code we will always use valid input values unless we specify otherwise.

Part Two

2 Peaks [7.5 marks]

In the past few years, plots of data like the one shown to the right have been in the news and all over the internet.



When analyzing such data it is sometimes useful to identify where the (local) maximum values are located. We'll call these maximal values *peaks* in the data. More formally, if we have a sequence of numbers $d_0, d_1, d_2, \ldots, d_{n-1}$, the peaks are identified as follows:

1. d_0 is a peak if $d_0 > d_1$,
2. d_{n-1} is a peak if $d_{n-2} < d_{n-1}$,
3. for any $1 \leq i \leq n-2$, d_i is a peak if $d_{i-1} < d_i$ and $d_i > d_{i+1}$

In the provided `Peaks.java` file, you will complete three static methods that relate to peaks in some data.

```
public static int numPeaks(int[] data)
public static int[] peaks(int[] data)
public static int[][] minmax(int[] data)
```

The `numPeaks(int[])` will count and return the number of peaks in the input data. The `peaks(int[])` method will find and return the locations of all the peaks in input data. The `minmax(int[])` will find the maximum and minimum values in the input data and output these values and all positions of where they occur. The output format must be as follows:

- the output array will have dimension 2 (that is, the length is 2)
- the first element will correspond to the min value and the second will correspond to the max value in the input
- each element in the output will be an array that has at least 2 values: the first is the value (either min or max) and the next will be the positions in the input array where this value is located

Here are some examples:

```
int[]   data1  = {1, 1, 4, 15, 3, 1};
Peaks.numPeaks(data1) -> 1
Peaks.peaks(data1) -> [3]
Peaks.minmax(data1) -> [ [1,0,1,5], [15,3] ]

int[]   data2  = {3, 1, 4, -2, 13, -2, 5};
Peaks.numPeaks(data2) -> 4
Peaks.peaks(data2) -> [0,2,4,6]
Peaks.minmax(data2) -> [  [-2,3,5]  , [13,4] ]

int[]   data3  = {1, 4, 4, 1};
Peaks.numPeaks(data2) -> 0
Peaks.peaks(data2) -> []
Peaks.minmax(data2) -> [ [1,0,3], [4,1,2] ]
```

Note: we are **not** considering what might be described as *plateaus* in the data. A plateau occurs when a local maximum value is repeated one or more times consecutively (like the value 4 in the last example above).

Note: all input to these methods will be an array with at least one number in it. That is, we will never pass an empty array or `null` as input to your methods.

Note: `peaks()` must always return an array. Even if there are no peaks, it must return an array object (and not `null`).

Aside: finding the *peaks* (which include the *plateaus*) in scientific data is very common. For example, analyzing data from various spectrometers in chemistry and physics labs often involves identifying the peaks in the data first. Tools like Matlab and Octave have built-in functions to find peaks similar to your `peaks()` method. In practise, it is more complicated though (you would be working with floating point numbers, you would have to deal with noise in the data, you would have more criteria for determining the peaks).

Submission Recap

A complete assignment will consist of two java files: `Convert.java` and `Peaks.java`.

Start early. Submit early. Submit often.