Homework 4

General comment about error handling: In most of the functions that you will have to write, the inputs of the functions can have all sorts of errors. However, at this stage in the course you can assume that the inputs are error-free, and there is no need to write error-checking code. Later in the course we will learn how to deal with input errors methodically. An important exception to this guideline is *empty inputs*: If a function expects to get an array, or a string, as an argument, it is possible that the array or the string are empty. Your method code must handle this particular edge case, *unless the method documentation says otherwise*.

1. Array operations

(30 points) For all practical purposes, a "string" and an "array of characters" are essentially the same thing. For example, consider the two declarations:

```
String str = "clearly";
char[] arr = {'c','l','e','a','r','l','y'};
```

The variable `str` points to a memory block holding a string object. The variable `arr` points to a memory block holding an array. The contents of these two different memory blocks are exactly the same – a sequence of characters. With that in mind, why does Java feature a special `String` type? That's because Strings are simpler to initialize (compared to arrays), and easy to process using cool method calls like `str.indexOf(char)` and `str.charAt(int)`. Arrays of characters don't have such built-in functions, so in this exercise we will write some

The array processing functions that we will write have exactly the same names as their equivalent string methods: `indexOf`, `charAt`, etc. But the signatures of these functions will be slightly different than the signatures of their corresponding string methods. In particular, when we call a *method* on some string object, say `str.charAt(3)`, we write the object's name before the method call. When we call the corresponding *function* on the equivalent array, say `charAt(arr,3)`, we pass the array as a regular argument. If you find this comment confusing, don't worry about it. We will have much more to say about the difference between methods and functions in the second half of the course.

Read the `ArrCharOps` class carefully, and complete its code. You will notice that the `compareTo` function has a complete API documentation that may look a bit cryptic. We will soon discuss the syntax and meaning of these API comments, but they are mostly self-explanatory.

Implementation notes

The `println(char[])` method: The code is given.

The `charAt(char[],int)` method: In this particular method you have to assume that the array is not empty, meaning that it contains at least one element.

The `lastIndexOf(char[])` method: Note that a for loop can also be written backwards: instead of iterating, say, from 0 up to 10, you can iterate from 10 down to 0.

The `concat(char[], char[])` method: One thing that you must do in this method, at some point, is creating a new array that will contain the concatenation of the two given arrays.

The `compareTo(String, String)` method: Unlike all the other methods in this class, this method operates on strings, and not on arrays. An equivalent method that operates on arrays of characters can easily developed, but we decided not to do it, since it complicates the method's testing. So, this will be a good opportunity to practice the syntax differences between array processing and string processing.

Note that in this method you have to assume that both strings are not empty.

2. Prime numbers

A *prime number* is a number > 1 which is divisible only by 1 and by itself. There is an infinite number of prime numbers, and here are the first few of them: 2, 3, 5, 7, 11, 13, 17, 19, ...

In lecture 4-2 we presented the "Sieve of Eratosthenes" algorithm for finding all the prime numbers between 2 and *n*. Here is an example of the program's execution for *n* = 30:

```
% java Primes 30

Prime numbers up to 30:
2
3
5
7
11
13
17
19
23
29
There are 10 primes between 2 and 30 (33% are primes)
```

Complete the code of the given `Primes` class.

Implementation notes

Before you start writing the code, read carefully the algorithm's explanation on lecture 4-2.

Although this algorithm can be implemented using `for` loops, it is easier to implement it using `while` loops.

3. MyString

The supplied `MyString` class features two string processing functions. Complete the code of these functions.

Implementation note

In the `contains(String, String)` method: Using `break` may simplify your code.

4. Bullshit detector

The ability to detect selected words in a given text comes to play in numerous AI, search, and real-life settings. In the Israeli media, for example, sentences that contain strings like ″אני בא ואומר″, ″ברמת העיקרון″, ″צריך לומר ביושר″, ″זה לא מובן מאליו″, normally indicate that the speaker has little to say, or is simply reciting known clichés. So, it's nice to have an automatic bullshit detector that can help flag fluff and hype in media contents.

The `KeywordsDetector` class is designed to perform this keywords detection task, in a more general way. Which keywords to detect is specified in the program's code. For example, the supplied version of `KeywordsDetector` detects and prints sentences that typically come up in bullshit business presentations. Here is an example of the program's execution:

```
% java KeywordsDetector
Our product will transform the market
We need to leverage our core competencies
This blockchain-based solution will disrupt the industry
The team showed great Synergy in the last project
Our new technology presents a significant paradigm shift
```

Complete the missing code of the `KeywordsDetector` class.

Implementation notes

The first thing to notice in the code of this class is that you can create and process arrays of `String` objects, just like creating and processing arrays of primitive types like `int` and `char`. For example, if `arr` is an array of strings, then `arr[2]` returns the third string in that array. If, for some reason, you want to access the first character in that string, you can use `arr[2].charAt(0)`. Or, assuming that the variable str is of type `String`, you can use `str = arr[2]` and then `str.charAt[0]`.

In the `detectAndPrint(String[], String[])` method: When comparing two strings (in this particular application), compare their lowercase versions. This will make sure that, for example, the string "`RUNI`" will match the string "`Runi`".