Deliverable

Functional Code is available at:

https://github.com/amartinezvilaclara/autocompleteJava

It contains two Java Classes and a Java Test Class, plus the libraries used (Junit 4.13 – beta 2 and Hamcrest core 1.3). The Main class corresponds to the functional running program, which shows on the console the result of the 4 examples given by the exercise.

There are no special compiling instructions, other than what is required to compile and run Java code and Junit.

The name of the unit tests can be used to understand the functionality of the class, as each one corresponds to a single feature.

Some coding decisions:

- I use a List instead of an array of strings as result, to avoid checking for NULL values if the result returns no suggested autocompletions.
- As the list of keywords is small, the cost of access and loop through the list of keywords is negligible. Thus the use of this Data Structure.
- Again, given the small scope of the exercise, the MVC model has been simplified, and all the keywords have been hardcoded into the autocomplete class.

Optional

What would you change if the list of keywords was large (several millions)?

Auto-complete is a "real time" operation and thus the latency between the input of the user and the return of the suggested words must be minimized as much as possible or be kept under a reasonable threshold (<1s) to do not affect the user experience.

https://github.com/Tendrl/documentation/wiki/Best-Practices-for-Response-Times-and-Latency

As the algorithm is made right now, the search has a linear cost, which scales badly. Therefore, I would use a Data Base to store and access the keywords, as they have been optimized to deal with millions of rows of data with a low latency.

If the use of a DB were not possible, I would try to reduce the linear cost of access by using different approaches. A bit of trial and error would be needed to select the best approach:

- using a different Data Structure (Trees have a lower access cost)
- splitting the data into several data structures (thus reducing the amount of data to be checked)
- etc.

Ultimately, scalability while keeping a low latency can be a difficult problem to solve, depending on the orders of magnitude for both the amount of data, and the response time.

What would you change if the requirements were to match any portion of the keywords (example: for string "pro", code would possibly return "re*pro*be")

I would have to loop through all the elements of the keyword list searching for keywords containing the given string:

```
s.toLowerCase();
for (int i = 0; i < keywords.size(); i++) {
    String keyword = keywords.get(i).toLowerCase();
    if(keyword.contains(s)) suggestions.add(keywords.get(i));
    if(suggestions.size() == MAXIMUM_SUGGESTIONS) break;
}
```

The loop could only be escaped earlier if the number of suggestions has reached the maximum allowed.