Object-Oriented Programming. Exercise 5.2

Unit 5. Collections and maps

Exercise 1. (project `prWordIndex`, package `prWordIndex`)

In this exercise we will practice on the use of collections and maps. We will use several types of maps together with lists and sets to create an application that allows us to make several lists with the words that appear in a text so that we know for each word of the text:

- How many times it appears.
- The lines in which it appears.
- The lines and the position (or positions) within each line in which it appears.

To do this, we are going to build three different indexes:

- **`CounterIndex`** will display each word in the text with its number of occurrences.
- **`LineIndex`** will display each word with all the lines in which it appears.
- **`PositionInLineIndex`** will display each word with all the lines it appears in, and within each line, the positions in which it appears.

For example, for text (accents have been intentionally removed):

> *Guerra tenia una jarra y Parra tenia una perra, pero la perra de Parra rompio la jarra de Guerra.*
>
> *Guerra pego con la porra a la perra de Parra. Oiga usted buen hombre de Parra! Por que ha pegado con la porra a la perra de Parra.*
>
> *Porque si la perra de Parra no hubiera roto la jarra de Guerra, Guerra no hubiera pegado con la porra a la perra de Parra.*

Using **`CounterIndex`** we would get the following output, which displays each word followed by the number of times it appears in the text:

```
a            3
buen         1
con          3
de           8
guerra       5
ha           1
hombre       1
hubiera      2
jarra        3
...
```

With **`LineIndex`** we would obtain the following output, which displays the words of the text along with the lines where they appear:

```
a          <2,3>
buen       <2>
con        <2,3>
de         <1,2,3>
guerra     <1,2,3>
ha         <2>
hombre     <2>
hubiera    <3>
jarra      <1,3>
...
```

Finally, with **PositionInLineIndex** we would obtain an index in which, for each word, the lines in which it appears are shown, and within each line, all the positions in which it appears:

```
a
              2 <6,24>
              3 <21>
buen
              2 <13>
con
              2 <3,21>
              3 <18>
de
              1 <13,18>
              2 <9,15,27>
              3 <5,12,24>
...
```

To separate the words within each line we will use the following delimiters. In the examples above, the delimiters string is `"[ .,:;\\-\\!\\¡\\¿\\?]+"` (The following delimiters could also be used `"[^a-zA-Z0-9áéíóúüÁÉÍÓÚÜ]+"`). The following diagram shows the interfaces and classes to develop within the **prWordIndex** package.

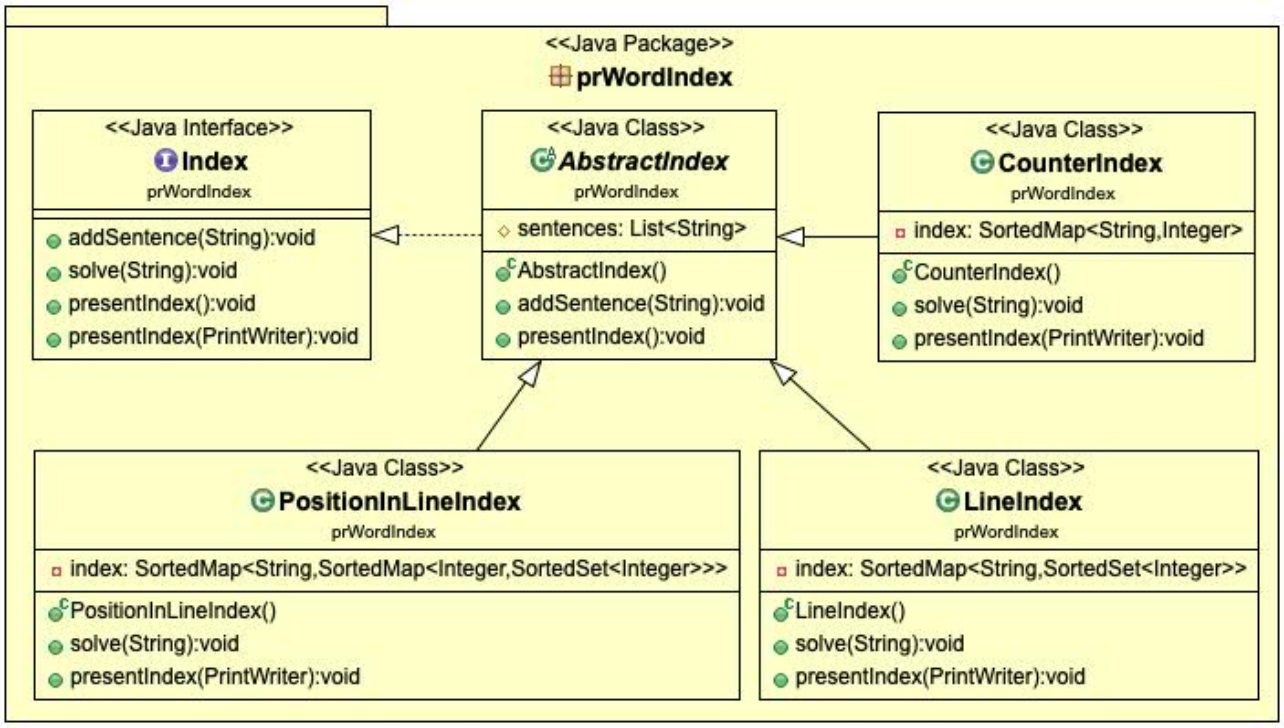

Figura 1: UML class diagram

Interface `Index`

Since, despite their differences, the three *indexes* have similar functionality, an interface **Index** will be defined, which will then be implemented by the three types of indexes. The interface specifies the following methods:

- `void addSentence(String sentence);`

  Adds a phrase or sentence of text to those already stored in the index. These sentences will form the text to be analyzed.

- `void solve(String delimiters);`

Constructs the index. It gets as argument the delimiters to be used to break works in the sentences. For example, the delimiters can be either `"[ .,:;\\-\\!\\¡\\¿\\?]+"` or `"[^a-zA-Z0-9áéíóúüÁÉÍÓÚÜ]+"`.

- `void presentIndex(PrintWriter pw);`

  Shows the result (in the format indicated above for each case) in `pw`.

- `void presentIndex();`

  Shows the result (in the format indicated above for each case) in the console.

Abstract class `AbstractIndex`

This class defines the common characteristics of the three indexes and **implements** the interface `Index`.

Sentences are stored in a *protected* list of `String` (`sentences`). Additionally:

- It defines a constructor that appropriately creates the structure.
- It implements the `addSentence(String)` method so that it adds a sentence or line of text to those already stored in `sentences`. These lines will form the text to be analyzed. The last line added will be the last line of the text.
- It implements the `presentIndex()` method in the `Index` interface so that the index is printed on the console. It uses the `presentIndex(PrintWriter)` method.

Class `CounterIndex`

The class `CounterIndex` **inherits** from the abstract class `AbstractIndex`, and will have, in addition to the variable `sentences` that inherits from `AbstractIndex`, a variable `index` where will store the index to be constructed using the text available in the sentence list `sentences` and the delimiters that are specified. The index to be built will be stored in a **map** in which the number of occurrences of a word is associated to each such word of the text (see the output in the case of the example above). Note that words must be kept lexicographically ordered.

To do this, the `CounterIndex` class will provide a constructor and implement the methods inherited from the `AbstractIndex`:

- It defines a constructor that properly initializes the structures that are necessary to develop the application. Initially there will be no text to operate on; both text and delimiters will be entered with appropriate methods.
- It redefines the `solve(String)` method to build the index, calculating the number of occurrences of each significant word in the `sentences` text and completing the `index` structure to keep this information.

  Note that:

  - You must first reset the index to empty (`clear()`).
  - Words are stored in lowercase.
  - To separate the words within each line of the text we will use an instance of the `Scanner` class with the provided delimiters.
  - The algorithm used must give a single pass through the sentences to complete the index.
- It redefines the `presentIndex(PrintWriter)` method to produce the output in the expected format.

We can test the operation of this class using the `MainIndex` application.

Class `LineIndex`

The `LineIndex` class also inherits from `AbstractIndex` and follows a pattern very similar to that of `CounterIndex`. The main differences with this are:

- The variable **`index`** will now be a map in which each word of the text is associated with a set with the numbers of lines where it appears (see the example above). The words will be ordered lexicographically and the line numbers from smallest to greatest.
- It redefines the `solve(String)` method to build the index as above.
- It redefines the `presentIndex(PrintWriter)` method to generate the listing as shown in the example.

We can test the operation of this class with the `MainIndex` application.

Class `PositionInLineIndex`

The `PositionInLineIndex` class also inherits from `AbstractIndex`, and follows a pattern very similar to that of `CounterIndex` and `LineIndex`. In this case **`index`** will be a map in which each word is associated with a second map in which a set is associated with each line number with the positions of the word in each line number. The words will be ordered lexicographically and the lines and positions from smallest to greatest.

We can test the operation of this class with the `MainIndex` application.