IF184301 Object Oriented Programming (E)

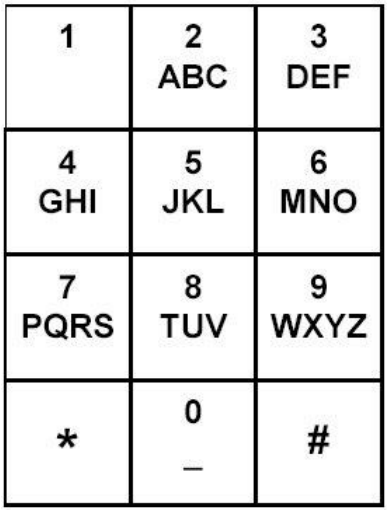
Quiz 2

Starting date:	18 November 2021
Deadline:	25 November 2021, 23:59 WIB. **Penalty: 0.15% of grade/minute of tardiness.**
Exam type:	Open, Group (of one/two/three students).
Send to:	MM Irfan Subakti <yifana@gmail.com> CC to Dicksen Alfersius Novian <dicksenan@gmail.com> with the subject: **IF184301_OOP(E)_Q2_ StudentID1_Name1_StudentID2_Name2**
File type & format:	A **zip file** containing the *source codes* (the *project* files), *Report.PDF* & *Declaration(s).PDF*. One *declaration* file per student. So, if a group has 3 members, then there will be 3 *declaration* files.
Filename format:	IF184301_OOP(E)_Q2_ StudentID1_Name1_StudentID2_Name2.ZIP

Introduction

The *predictive text* will be implemented. It's an application using the Java Collection classes. The Graphical User Interface (GUI) will also be used for empowering the application.

Recalling when we were entering text into a cell phone (*handphone*) without a full keypad, a compromise is made by putting more than one letter on a single button. A common layout can be seen below.



First row: 1 | 2 (ABC) | 3 (DEF)
Second row: 4 (GHI) | 5 (JKL) | 6 (MNO)
Third row: 7 (PQRS) | 8 (TUV) | 9 (WXYZ)
Fourth row: * | 0 (space) | #

In the standard system without predictive text, the user must press the appropriate button several times for a particular letter to be shown. Consider the word "hello". With this method, the user must press 4, 4, 3, 3, 5, 5, 5, then pause, then 5, 5, 5, 6, 6, 6.

Predictive text (*T9*) is a system that aims to reduce the number of button presses needed to enter text. The user presses each button only once and several matching words are presented by the predictive text. So, the word "hello" can be typed in 5 button presses "43556" without pauses, instead of 13 times pressing the button in the standard keypad system. The numeric string "43556" is referred to as a "signature" of the word "hello".

A given numeric signature can correspond to more than one word. Predictive text technology is possible by restricting available words to those in a dictionary. Entering the numeric signature "4663" produces the words "gone" and "home" in many dictionaries.

You will design and develop a predictive text system. For simplicity, assume that the user does not need punctuation or numerals. You have to limit your solutions to producing only `lower-case` words. You must use the dictionary, e.g., the file `words`, found on our website, along with this document. Use the `package/class/`method names given in the question.

[1] Prototypes and design [**25 points**]

This part deals with building a "prototype" for the predictive text problem, which is not expected to be efficient. However, it will be simple and allow us to compare it with the efficient implementation to be done in later parts.

Write the first two methods in a class named `PredictivePrototype`. The classes in this assignment (except part 4) should be placed in a package called `predictive`.

1. [**5 points**] Write the method (in the `PredictivePrototype` class):

```
public static String wordToSignature(String word)
```

The method takes a word and returns a numeric signature, e.g., "home" should return "4663". If the word has any non-alphabetic characters, replace them with a " " (`space`) in the resulting signature. Accumulate the result character-by-character.

You should do this using the `StringBuffer` class rather than `String`. Explain, in your comments, why this will be more efficient.

2. [**10 points**] Write another method in the `PredictivePrototype` class:

```
public static Set<String> signatureToWords(String signature)
```

It takes the given *numeric signature* and returns a *set of possible matching words* from the dictionary. The returned list must not have duplicates and each word should be *lower-case*. The method `signatureToWords(String signature)` will need to use the dictionary to find words that match the string signature and return all the matching words.

You must not store the dictionary in your Java program. Explain in the comments why this implementation is inefficient.

3. [**10 points**] Create command-line programs (classes with `main` methods) called `Words2SigProto` for the `wordToSignature` method and `Sigs2WordsProto` for the `signatureToWords` method.

Each program must accept a list of words and call the appropriate method to do the conversion.

Run:
```
Word2SigProto home hello world my name is
```

Output:
```
input : [home, hello, world, my, name, is]
output : 4663 43556 96753 69 6263 47
```

Run:
```
Sigs2WordsProto 4663 43556 96753 69 6263 47
```

Output:
```
4663 : [hood, ione, ioof, good, hond, inne, gond, hone, hoof,
gone, goof, home, gome]
43556 : [gekko, hello]
96753 : [world, yorke]
69 : [ow, nw, ox, mw, oy, mx, ny, oz, my, nz]
6263 : [mane, name, mand, nane, nand, oboe, mame]
47 : [ip, hp, iq, gp, hq, ir, gq, hr, is, gr, hs, gs]
```

Hints

- Use the `Scanner` class to read the dictionary line by line, assume there is only one word per line.
- When reading the dictionary, ignore lines with non-alphabetic characters. A useful helper method to accomplish this would be:

  ```
  private static boolean isValidWord(String word)
  ```

  in `PredictivePrototype`, which checks if a given word is valid.
- To create the command-line programs, you will need to use the `String[] args` array of the

  ```
  public static void main(String[] args)
  ```

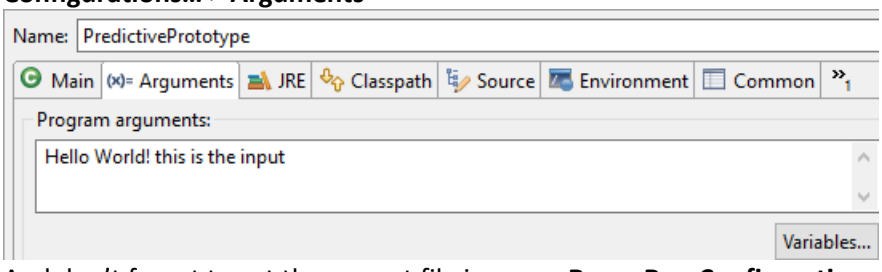
  method to access command-line input. For example, when executing

  ```
  Word2SigProto Hello World! this is the input
  ```

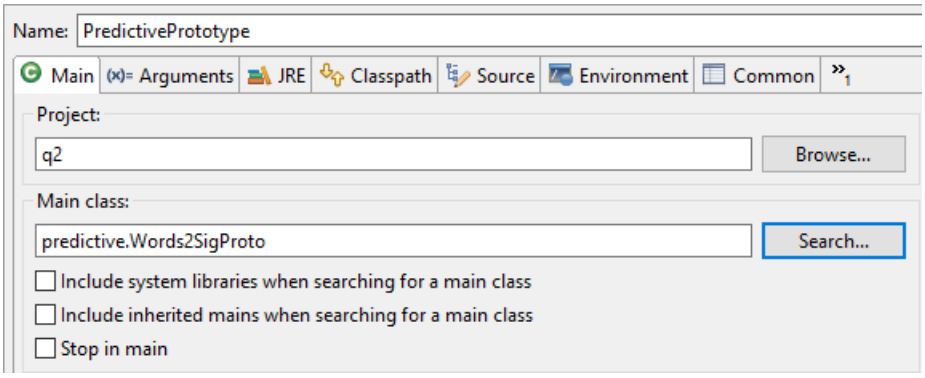
  the `String[] args` array will contain

  ```
  ["Hello", "World!", "this", "is", "the", "input"]
  ```

  Alternatively, in Eclipse you can write the argument in menu **Run > Run Configurations… > Arguments**



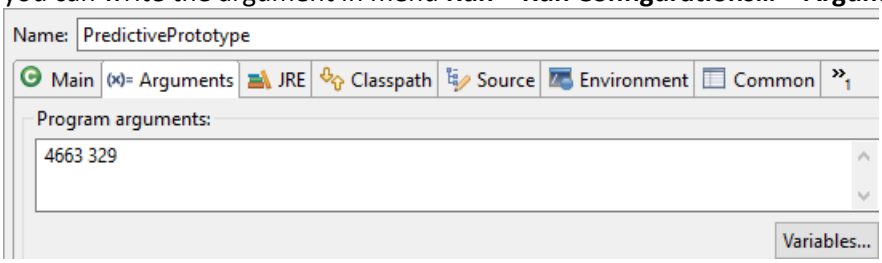
And don't forget to set the correct file in menu **Run > Run Configurations… > Main**



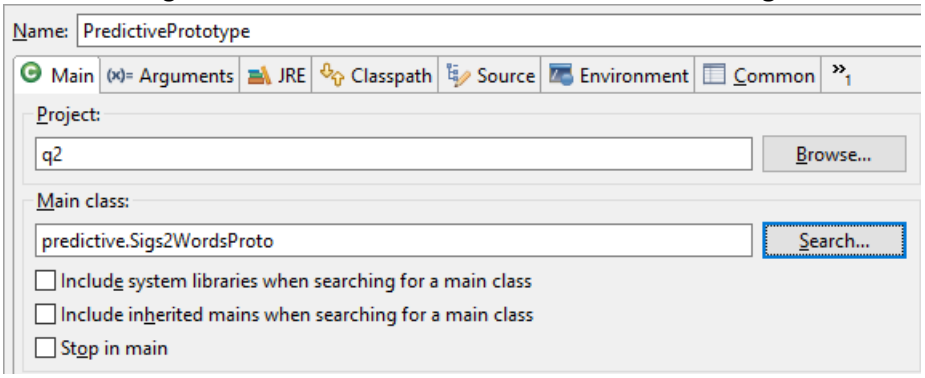
- You can ignore any words with non-alphabetic characters given in the input of `Sigs2WordsProto`.
- Format the output of `Sigs2WordsProto` as one line per signature, as there may be more than one word for a given numeric signature. E.g.,

```
D:/predictive/>java -cp .. predictive.Sigs2WordsProto 4663 329
4663 : good gone home hone hood hoof
329 : dax fax faz day fay daz
```

the actual output you get will depend on the dictionary used. Alternatively, in Eclipse you can write the argument in menu **Run > Run Configurations… > Arguments**



And don't forget to set the correct file in menu **Run > Run Configurations… > Main**



- In the above example, notice that the folder name is the same as the package name, the full class name is used and the a `-cp ..` option is in the `java` command. This will allow a class that is in the `package predictive` to run.
- The program `Words2SigsProto` can be tested by converting large amounts of text to signatures, the output can be used to test `Sigs2WordsProto` (and later, in timing comparisons).
  Try using news articles to start with.

[2] Storing and searching a dictionary [**25 points**]

Text input should be responsive, reading the entire dictionary from the disk each time a word is looked up will cause a noticeable *delay*. In this part of the assignment, you will read and store the dictionary in *memory* as a ***list of pairs***. As the list will be sorted and in memory, a faster look-up technique can be used.

1. [**15 points**] Create another class, named `DictionaryListImpl`. In its constructor, you should read the dictionary from a file and store it in an `ArrayList`. Each entry of the `ArrayList` must be a pair, i.e., the word that has been read in and the corresponding signature, so you will need to create a class named `WordSig` that pairs words and signatures (see the hints).

The `wordToSignature` method will be **the same** so you can re-use the code from the **first part**, i.e., [1] Prototypes and design – so, just only need to build the method/function: `signatureToWords`. The `signatureToWords` method must be **re-written** to use the dictionary stored in the `ArrayList<WordSig>`. The `ArrayList<WordSig>` must be stored in sorted order and the `signatureToWords` method must use ***binary search*** to perform the *look-ups*.

The result from your above method/function will be:

```
signatureToWords("4663") -> [hood, ione, ioof, good, hond, inne,
gond, hone, hoof, gone, goof, home, gome]
signatureToWords("43556") -> [hello, gekko]
signatureToWords("96753") -> [world, yorke]
```

2. [**5 points**] Based on the design of the `DictionaryListImpl` class, create and document the Java **interface** `Dictionary`. This will be used in later parts.

3. [**5 points**] Design and create the command-line program `Sigs2WordsList` for the `DictionaryListImpl` class. Compare the time taken to complete the execution of `Sigs2WordsList` and `Sigs2WordsProto` with the same large input(s). Is it possible to make the time difference between `Sigs2WordsList` and `Sigs2WordsProto` noticeable? Make a note of the data you use and your timing results.

Run:

```
Sigs2WordsList 4663 43556 96753 69 6263 47
```

Output:

```
4663 : [hood, ione, ioof, good, hond, inne, gond, hone, hoof,
gone, goof, home, gome]
43556 : [hello, gekko]
96753 : [world, yorke]
69 : [ow, nw, ox, mw, oy, ny, mx, oz, nz, my]
6263 : [name, mane, nane, mand, oboe, nand, mame]
47 : [ip, iq, hp, ir, hq, gp, hr, gq, is, hs, gr, gs]
```

Hints

- Create a class which pairs the numeric signatures with words, like this:

```
public class WordSig implements Comparable<WordSig> {
      private String words;
      private String signature;
      public WordSig (...) { ... }
      public int compareTo(WordSig ws) { ... }
      ...
}
```

- When you read the dictionary, you will need to create new `WordSig` objects.
- A list of `Comparable` objects can be sorted using the method `Collections.sort`.
- To automatically sort a list using the collections API, the objects `WordSig` stored in the list must implement the `Comparable` interface. That means they must have a `compareTo(...)` method. `compareTo` returns -1, 0 or 1 according to whether the current object is less than, equal to, or greater than the argument object, in the intended ordering.
- List elements must be sorted by numerical order by signature. Note that the alphabetic (String's `compareTo`) order is not the same as numerical order, you must ensure that the `WordSig` is in numerical order. Only sort the dictionary ***once***.
- You can search a sorted list using `Collections.binarySearch`. Note that binary search will return the index of the first match it finds. You must return all matching words. Scan above and below the found index to collect all matching words.
- The `time` command-line program on `Linux/Mac OS` machines will tell you how long a given command takes to complete. E.g.

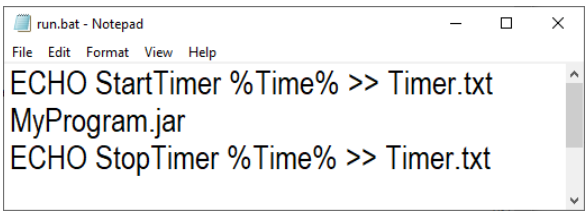
```
a@b:~/predictive/$ time java -cp .. predictive.Sigs2WordsList
<input> <output>

real 0m0.286s
user 0m0.260s
sys 0m0.010s
```

  Use the `real` elapsed time in all comparisons.

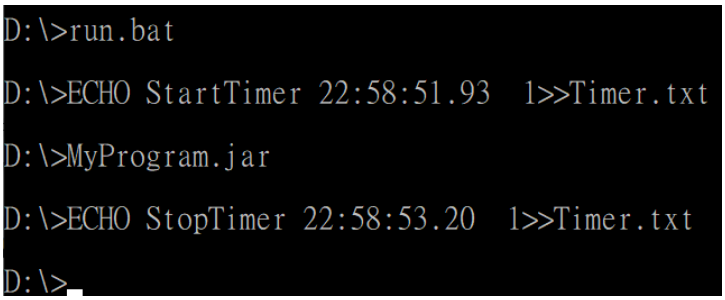
- For `Windows OS` machines, create a batch file, e.g., `run.bat`. Write down as follows.

```
ECHO StartTimer %Time% >> Timer.txt
<write down your command/program in here, e.g., MyProgram.jar>
ECHO StopTimer %Time% >> Timer.txt
```

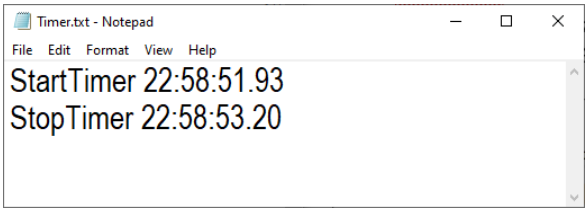
  For instance, we have `run.bat` as follows.



Once it's created, we can type `run` as below.



`Timer.txt` contents can be seen as follows.



From `Timer.txt`, then you can tell how long a given command takes to complete.

[3] More efficiency and prefix-matching [**25 points**]

This part involves creating two improved implementations of the `Dictionary` interface.

1. [**5 points**] Implement a new class `DictionaryMapImpl` that stores the dictionary using a *generic* multi-valued `Map`. In this context, a multi-valued map is a data structure that maps signatures to a collection of words. Using a `Map`, data can be retrieved quickly by signature, as in previously `DictionaryListImpl` but does not require scanning either side of the index as in the previous part. `DictionaryMapImpl` should also allow the efficient insertion of new words into the dictionary while still allowing fast look-up.

You must choose a `Map` implementation from the Java Collections API, explain how the map works and justify your choice. The constructor `DictionaryMapImpl` must populate the `Map`. Write the method `signatureToWords` that returns, in a `Set<String>`, only the matching whole words for the given signature. The character length of each returned word must be the same as the input signature.

The result from your above method/function will be:

```
signatureToWords("4663") -> [hood, ione, ioof, good, hond, inne,
gond, hone, hoof, gone, goof, home, gome]
signatureToWords("43556") -> [gekko, hello]
signatureToWords("96753") -> [world, yorke]
```

2. [**15 points**] Implement a new class `DictionaryTreeImpl` that now stores the dictionary in your tree implementation. It should be possible to search the tree-based implementation quickly (similar to `DictionaryListImpl`) and words can be inserted quickly (as in `DictionaryMapImpl`). `DictionaryTreeImpl` should support finding words when only the first part of the signature (a *prefix*) is known. This is so that the user can see the word they intend to type as they are typing. This tree implementation is quite similar to the actual implementations found in mobile phones.

The `DictionaryTreeImpl` class forms a recursive data structure, similar to, but more general than, the `Tree` class in our midterm exam weeks ago. This tree differs in that each node now has up to *eight branches*, one for each number (2-9) that is allowed in a signature. Each path of the tree (from the root to a node) represents a signature or part of a signature. Each node of the tree must store a collection of words that corresponds to the part of the signature represented by the path from the root to that node.

Write a constructor for the class `DictionaryTreeImpl` that takes a `String` path to the dictionary and populates the tree with words. Write the method `signatureToWords` that returns, in a `Set<String>`, the matching words (and prefixes of words) for the given signature. The character length of each of the returned words or prefixes must be the same as the input signature.

The result from your above method/function will be:

```
signatureToWords("4663") -> [inoe, inod, hood, inme, ioof, ione,
imme, good, inne, hond, inof, hooe, hone, gond, hoof, gooe, gnof,
home, gone, goof, honf, gome, gonf]
signatureToWords("43556") -> [gellm, helln, hellm, gekko, hello]
signatureToWords("96753") -> [workd, worle, world, workf, worke,
yorke]
```

3. [**5 points**] It should be possible to modify just one line in your `Sigs2WordsList` program so that it can work with any given implementation of the `Dictionary` interface. Create the programs `Sigs2WordsMap` and `Sigs2WordsTree`.

Compare the time taken to complete the execution of `Sigs2WordsMap` and `Sigs2WordsTree` with large inputs. Is it possible to make the time difference between `Sigs2WordsList` and `Sigs2WordsMap` or `Sigs2WordsTree` and `Sigs2WordsMap` noticeable? Again, make a note of the data you use and your timing results.

Run:
```
Sigs2WordsMap 4663 43556 96753 69 6263 47
```

Output:
```
4663 : [hood, ione, ioof, good, hond, inne, gond, hone, hoof,
gone, goof, home, gome]
43556 : [gekko, hello]
96753 : [world, yorke]
69 : [ow, nw, ox, mw, oy, mx, ny, oz, my, nz]
6263 : [mane, name, mand, nane, nand, oboe, mame]
47 : [ip, hp, iq, gp, hq, ir, gq, hr, is, gr, hs, gs]
```

Run:
```
Sigs2WordsTree 4663 43556 96753 69 6263 47
```

Output:
```
4663 : [inoe, inod, hood, inme, ioof, ione, imme, good, inne,
hond, inof, hooe, hone, gond, hoof, gooe, gnof, home, gone, goof,
honf, gome, gonf]
43556 : [gellm, helln, hellm, gekko, hello]
96753 : [workd, worle, world, workf, worke, yorke]
69 : [ow, ox, nw, oy, mw, nx, ny, oz, mx, my, nz, mz]
6263 : [name, mane, ocne, mand, nane, ocoe, namd, obne, nand,
oboe, mcme, mcne, mame, manf]
47 : [ip, iq, hp, ir, gp, hq, is, hr, gq, gr, hs, gs]
```

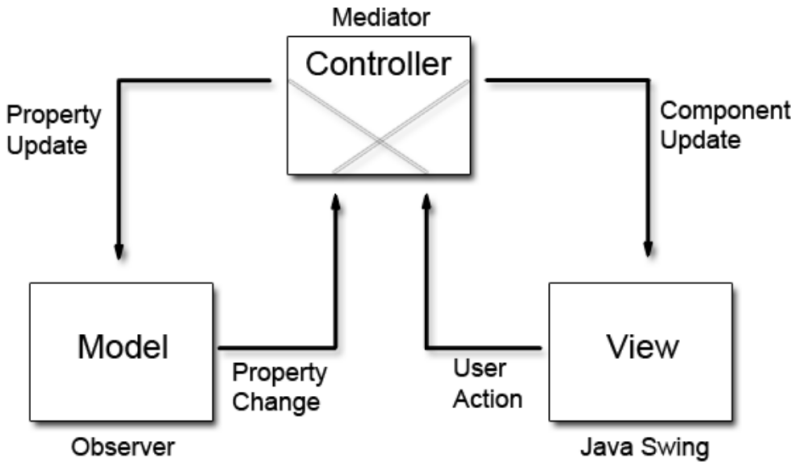
Hints

- Both of the classes in this part of the assignment must implement `Dictionary`. Do not use the `WordSig` class.
- When deciding what your `Map` will store in `DictionaryMapImpl`, keep in mind that one signature often corresponds to several words.
- When developing `DictionaryListImpl`, you may notice it was useful to create helper methods to add words to the data structure. Creating add helpers will simplify the constructors of both `DictionaryMapImpl` and `DictionaryTreeImpl`.
- Before starting `DictionaryTreeImpl`, sketch a tree-dictionary containing 2-3 words.
- Every node of `DictionaryTreeImpl` will have a collection of words and eight `DictionaryTreeImpl`s.
  You may use an array of `DictionaryTreeImpl` or just store several objects, as you prefer.
- The `signatureToWords` method of `DictionaryTreeImpl` can be implemented using a `while`-loop or recursion. When using recursion, you may find it useful to create a helper method that performs the recursion.
- The root node of `DictionaryTreeImpl` should not store any words.
- In `DictionaryTreeImpl` it is more memory efficient to store ***only*** whole words as read-in from the dictionary. You should do this and write a helper method to trim all words in a given list.

[4] Graphical user interface [**25 points**]

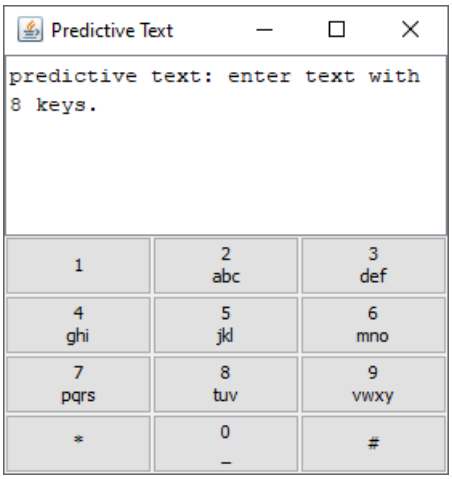
In this part, you will develop a *Graphical User Interface* (GUI) for the previously developed classes. You should create the following program in a new package: `predictivegui`.

1. [**25 points**] To ensure your GUI is reliable and does not interfere with the existing code, you must use the *Model View Controller* (MVC) method of GUI development we have learned in our lectures. MVC is a good discipline for building GUIs because it ensures that the user interface code is separate from the program code. This makes it possible to change the user interface without changing the underlying program.

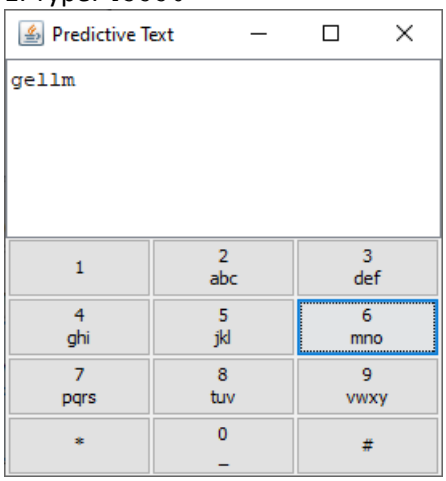


You should encapsulate the `Dictionary` object in the model. The model contains a constructor to load the dictionary and a get method to query the `Dictionary` with signatures. The model may store one dictionary object and a list of entered words but must not store signatures or spaces.

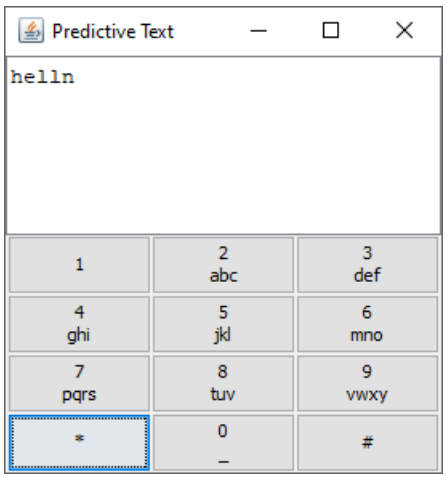
In this program, there is one main control mechanism, the *buttons*. The message appears in the *text window* which is the main view of the program, shown above the keypad. Pressing button "`2`", you will get one of "`a`", "`b`", or "`c`". It should not be possible to edit or type as you normally would with a full keypad in the text-field. To *cycle* through the words that match, the user clicks the button labelled "`*`". Pressing "`0`" completes the entry of the current word and creates a space, ready for the entry of the next word. Last one, "backspace" functionality can be done on the current word by pressing "`#`", i.e., deleting one character/letter from behind.



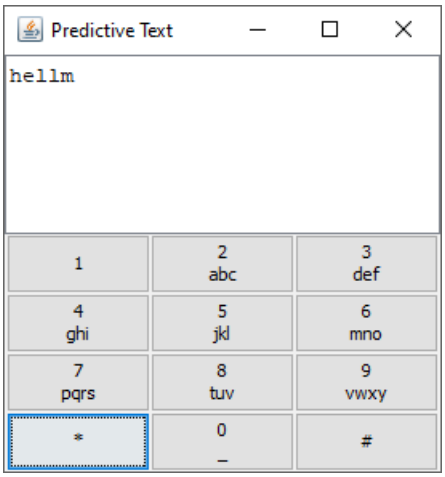
1. Type: `43556`



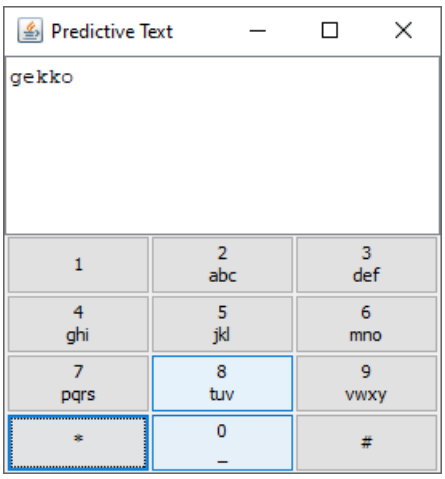
2. Press: `*`



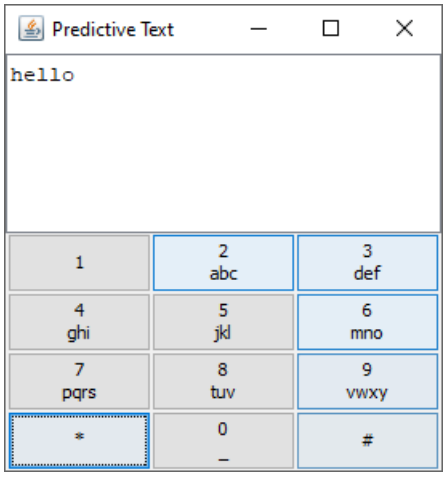
3. Press: `*`



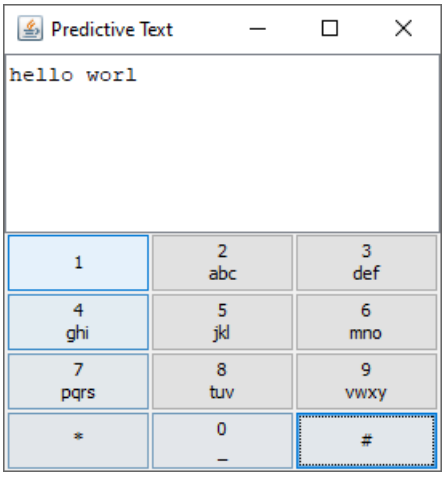
4. Press: `*`



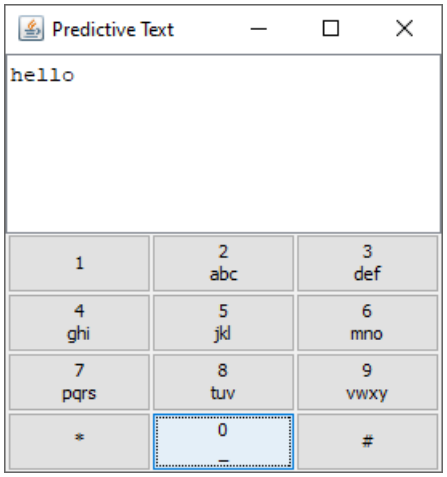
5. Press: `*`



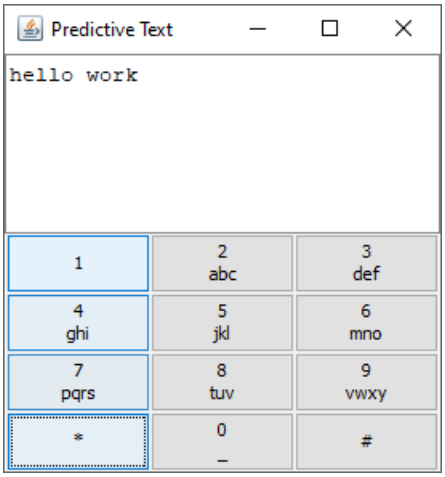
8. Press: `#`



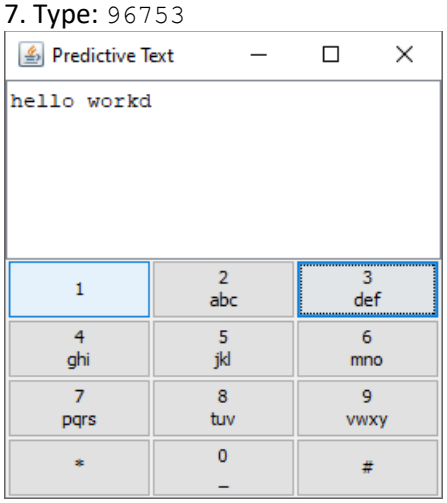
6. Press: `0`



9. Press: `*`



7. Type: `96753`



Hints

- Use what we have learned (and any material you may get from the internet).
- This part of the assignment can be developed using any implementation of the `Dictionary` interface. It should not be necessary to modify any of your existing classes to complete this part.
- You may want to implement more features such as number entry by keypad, punctuation, editing already typed words and adding words to the dictionary, etc., but make sure the basic functionality is working.

Declaration

To avoid plagiarism/cheating, every student needs to pledge and declare, then she/he must submit her/his **signed pledge and declaration** as in the following. Failing to do so will be resulted in getting a 0 (zero) grade. Attach the **scanned**/**photo** of your *declaration* in your report.

> "By the name of Allah (God) Almighty, herewith I pledge and truly declare that I have solved quiz 2 by myself, didn't do any cheating by any means, didn't do any plagiarism, and didn't accept anybody's help by any means. I am going to accept all of the consequences by any means if it has proven that I have been done any cheating and/or plagiarism."
>
> [Place, e.g., Surabaya], [date, e.g., 25 November 2021]
>
> `<Signed>`
>
> [Full name, e.g., Pujiati Sri Lestari]
> [StudentID, e.g., 05112040000xxx]

Final task

Make a report of this project, namely `Report.PDF`, about what you have been done, the output, the answers to the mentioned questions, pseudocode, whatever you think it's important.

To maintain fairness for your awarded grade, state clearly in your report, the job descriptions and its percentage (distribution) of works, along with the contribution(s) for each member in your group. E.g., Kelana Bagus 55% [*state Kelana's contribution(s) here*], and Pujiati Sri Lestari 45% [*state Pujiati's contribution(s) here*]. Include this distribution and contribution's section in your `Report.PDF`.

Then `Report.PDF` needed to be compressed (zipped). Rename the `ZIP` file to `IF184301_OOP(E)_Q2_StudentID1_Name1_StudentID2_Name2.ZIP`. All of the involved files in your project (the *source codes* and everything), as well as all of your *declaration* files needed to be added to the `ZIP` file.

Have a lovely day, guys! Good luck! 😊