Lab 2. Analysis of Algorithms

Theme. In this lab, you will:

- deriving growth function for an algorithm
- practise string manipulations
- practise using Java interfaces to implement a flexible object-oriented design (OOD)
- practise the use of polymorphism in OOP
- perform error handling using exceptions

Key concepts: growth functions, Big-O notation, Java interface, polymorphism, exceptions, arrays and collections, iteration, package

Required file(s): `lab2.zip`

1 Getting Started...

1. Download the archive `lab2.zip` from Blackboard. Extract the contents into your default `Eclipse` workspace for this module.
2. Start up `Eclipse`.
3. Create a new Java project named **conference_algorithmAnalysis** using the contents extracted from `lab2.zip`.

2 Your Tasks: Algorithm design and analysis

In a computer science conference, some participants cannot hear what the speakers say very well. To improve the hearing condition, the organisers decided to use a Java application to display each speaker's speech while they give their seminar. This system has a simple GUI front-end which displays the exact speech in text format of the current speaker as it is delivered. This GUI has a button for the participants to 'see' the next speech and another button for closing the application. The development of this Java application has begun, but it is not yet fully completed. Your task is to finish off the implementation of this application according to the following requirements.

Hint: The approximate locations where you are expected to add your Java code and relevant hints for accomplishing the tasks have been marked throughout the given Java programs. Look out for block comments that include a sequence of *four* exclamation marks, i.e.:

```
/* !!!!  ...  ...  */
```

1. Derive a growth function for *each* speaker style as modelled in each **class method** in the class `Speakers`:
   - Some speakers like to 'shout' (modelled by capitalising all letters in their speech).
   - Some speakers like to 'shout' out long words only (i.e. words with > 6 letters).
   - Some speakers like to emphasise the words that have occurred more than once in their speech using an exclamation mark.

Given a message to be uttered by a speaker, each class method modifies the output speech with the required quirk.

N.B.: You should aim at deriving a growth function for at least 2 speaker styles.

Hint: One way to derive a growth function for a computer program is to count the number of steps in the program. To observe how the time efficiency of the program changes as the size of problem grows, we execute the program several times, each time with a different number of inputs.
In the `main` method in `Speakers`, count the number of steps each invocation of a **class method** requires:

(a) Set a break point on the required **class method**.

If you are using `eclipse`, simply right-click on the left hand side scroll bar of the code editor and select "Toggle Breakpoint" at the required method call.

(b) Debug `Speakers` as a Java application.
If you are using `eclipse`, simply choose "Debug As".

The program execution will pause at the class method.

(c) Click on the "Step Into (F5)" icon to follow the execution of the class method step by step.

N.B.: Do not step into the execution of any line of code that are not defined in class `Speakers`. Simply *step over* them.

To simplify our step-counting task, we assume that the execution time of the Java library code that class `Speakers` uses is constant. Of course, in reality, such library code is likely to have a linear time complexity. For example, the time required to print the characters in a `String` object is dependent on the length of the underlying **string**. However, as the focus of our exercises is to compare the algorithms used in the methods in class `Spearkers`, by stepping over the lines of code that are not defined in class `Speakers`, we focus the step counting on our definition of `Speakers`, rather than the Java library code that `Speakers` uses.

(d) At each line in the executing **class method**, click on the "Step Over (F6)" icon to move to the next execution step.

(e) Tally the number of times you clicked on the "Step Over (F6)" icon.
Note down the number of execution steps at each iteration of the required class method. This will enable you to observe how the total number of steps changes when the size of input changes.
Complete the worksheet in the appendix.

(f) When the execution finishes, return to the Java perspective by selecting the option on the top right hand corner of the workbench.

To tally the steps required in other scenarios, e.g. worst case and best case, you may need to edit the contents of the messages defined in the named constant `Speaker.messages` before redoing the step counting exercise.

2. There should be a **realisation** relation[1] between the class `Dog` and the interface `Speaker`.
   Modify the `Dog` class header accordingly to reflect this relationship.

3. Complete the `speak` method for class `Dog`.

   A dog can speak like a human being(!). However, every now and then the old speaking habits creep into its speech. Thus, after a dog has uttered *five* words, it then gives out a "Woof!". For example, if a dog intends to say "*My name is Super Dog. Thank you for listening to me.*", you will hear "*My name is Super Dog. Woof! Thank you for listening to Woof! me.*".

4. There should also be a **realisation** relation between the class `Philosopher` and the interface `Speaker`.
   Modify the `Philosopher` class accordingly to reflect this relationship.

   Hint: When a class implements an interface, it inherits *all* abstract methods from the interface.

   Philosophers like to clear their throats when they make a speech. Every time they clear their throats, an "Ah-Hem!" is uttered. Hence, when a philosopher delivers a speech, "Ah-Hem!" is scattered throughout the speech in a *random* manner. For example, if a philosopher wants to say "*Today is a sunny day. We are going to look at how the global climate affects our lives.*", you may hear "*Today is a sunny day. We are going to look at how Ah-Hem! the global climate affects our lives.*" or "*Today Ah-Hem! is a sunny day. We Ah-Hem! are going to look at how the global climate Ah-Hem! affects our lives.*".

5. The constructor `Conference(String)` in the `Conference` class is responsible for creating `Conference` objects. This constructor reads seminar records from a plain text data file using a `java.util.Scanner` object. For each seminar record, a `Seminar` object is created accordingly. Each seminar record is expected to occupy one line in the data file and it has four attributes:

   - the title of the seminar
   - the content of the speech
   - the name of the speaker
   - the speaker type, represented as an integer
     (e.g. 0 means that it is a dog and 1 refers to a philosopher)

   Each attribute is separated by a **Tab** character (i.e. the character `'\t'` in Java).
   During the creation of a `Conference` object, if the input data file does not conform to the expected data format (e.g. a seminar record has two attributes only), the constructor should throw a `BadDataFormatException` exception. The exception should contain an appropriate error message.

   Add suitable error check(s) to this constructor.

[1] When a class implements an interface, there exists a *realisation* relation between them.

3 Testing

Now test your implementation to see if it meets the above requirements. You will find a plain text file named `seminars.txt` in your `seminars` folder. This file contains two seminar records. You may use it for testing your application.

Hint: To run a Java application within `Eclipse`:

1. Select the top level class of this Java application (i.e. `GUIConference.java`) within the `Package Explorer` tab.
2. Select from the pull-down menu: `Run` ➡ `Run Configurations...`
   A new window titled "Run Configurations" will then pop up.
3. On the left hand side of the "Run Configurations" window, there is a navigation menu with the item `Java Application`. If the name of the high level class (i.e. `GUIConference`) has not yet appeared as a sub-item underneath `Java Application`, double-click on the item `Java Application`. This will add the previously selected class to the list of runnable Java applications known by `eclipse`.
4. Select the required runnable Java application (i.e. `GUIConference`).
5. `GUIConference` requires the name of a seminar data file as its runtime argument. To specify the runtime argument:
   (a) On the right hand side of the "Run Configurations" window, select the tab titled `(x)= Arguments`.
   (b) This Java program takes one program argument, which is the name of the data files for seminar details.
   Select the `Arguments` tab. In the `Program arguments:` textarea, enter the desirable program argument, e.g.: `seminars.txt` or `badData.txt`.
   (c) Press the button "Apply", when finished entering the program argument(s).
6. Press the button "Run", when you are ready to run the Java application.

- When a dog gives a seminar, does its speech contain a "Woof!" after every *five* words?
- When a philosopher gives a seminar, does the speech contain "Ah-Hem!" at random points? Thus, if you re-run the conference and display the same speech by the philosopher again, does the throat-clearing sound now appear at different points in the speech?
- When inputting a data file (e.g. the data file `badData.txt`) which does not conform to the expected data format, does the application pop up a message window to alert the bad data format error?

4 Further Challenges

1. Each of the following **Big-O** notation expressions describe the time complexity of an algorithm.

   - $O(1)$
   - $O(5^n)$
   - $O(n \log n)$
   - $O(n^3)$

   (a) State the meaning of *each* of the above **Big-O** expressions.

   (b) List the above **Big-O** expressions in ascending order of time complexity, i.e. the most time efficient one on the left, and the least time efficient one on the right.

2. State the time complexity of *each* of the following **growth functions**:

   (a) $T(n) = 400 + 36n^2 + \log_2 n + 15n^2$

   (b) $T(n) = 25n^2 + \frac{7^n}{500} + 8 + \frac{1}{30}n^3 + 3n$

   (c) $T(n) = 20 \times 5$

3. Consider the following Java code for method `speak`, which generates a new `String` object by adding `"Quack!"` after every 10 words in the specified `String` array `words`:

```java
public String speak(String[] words) {

    String mySpeech = "";

    /* Adds 'Quack!' after every 10 words. */
    int i = 0;
    while (i < words.length) {
        for (int j = 0;
             (j < 10 && i < words.length);
             j++)
        {
            mySpeech += words[i] + ' ';
            i++;
        }
        mySpeech += "Quack! ";
    }

    return mySpeech;
}
```

   Making use of a suitable **Big-O** notation, state the time complexity of method `speak`. Justify your answer.

5 Further Programming Exercises

After you have finished all of the above tasks, if you would like to take on further challenges, do the following:

1. Design and implement a different speaker type that has similar capabilities to a `Dog` or a `Philosopher`, but speaks with a different verbal quirk.
2. Modify the application to enable a speech to be given by two speakers, with each speaker having different characteristics.

6 Appendix

Method `Speakers.capitalisedAll(String)`

Number of words in message (n)	Counted Steps
1	9
2	
3	
4	
5	
6	

How many times did the debugger execute each line of the following piece of Java code? Write out your answer in the red angled brackets (i.e. <>).

```
< >         String[] words = message.split(" ");
< >         String result = "";
< >         for(String word : words) {
< >             char firstLetter = word.charAt(0);
< >             result += Character.toUpperCase(firstLetter);
< >             result += word.substring(1);
< >             result += " ";
            }
< >         return result;
```

Growth Function for `Speakers.capitalisedAll(String)` is:

Method `Speakers.shoutLongWords(String)`

Average Case: a mixture of long and short words in the message

Number of words in message (n)	Counted Steps
1	
2	
3	
4	
5	
6	

How many times did the debugger execute each line of the following piece of Java code? Write out your answer in the red angled brackets (i.e. <>).

```
< >         String[] words = message.split(" ");
< >         String result = "";
< >         for(String word : words) {
< >             if(word.length() > 6) {
< >                 word = word.toUpperCase();
                }
< >             result += word;
< >             result += " ";
            }
< >         return result;
```

Growth Function for *average case* of `Speakers.shoutLongWords(String)` is:

Method `Speakers.shoutLongWords(String)`

Best Case: no long words in the message

Number of words in message (n)	Counted Steps
1	8
2	
3	
4	
5	
6	

How many times did the debugger execute each line of the following piece of Java code? Write out your answer in the red angled brackets (i.e. <>).

```
< >         String[] words = message.split(" ");
< >         String result = "";
< >         for(String word : words) {
< >             if(word.length() > 6) {
< >                 word = word.toUpperCase();
                }
< >             result += word;
< >             result += " ";
            }
< >         return result;
```

Growth Function for *best case* of `Speakers.shoutLongWords(String)` is:

Method `Speakers.shoutLongWords(String)`

Worst Case: all words in the message are long

Number of words in message (n)	Counted Steps
1	9
2	
3	
4	
5	
6	

How many times did the debugger execute each line of the following piece of Java code? Write out your answer in the red angled brackets (i.e. <>).

```
< >         String[] words = message.split(" ");
< >         String result = "";
< >         for(String word : words) {
< >             if(word.length() > 6) {
< >                 word = word.toUpperCase();
                }
< >             result += word;
< >             result += " ";
            }
< >         return result;
```

Growth Function for *worst case* of `Speakers.exclamationMarkOnRepeat(String)` is:

Method `Speakers.exclamationMarkOnRepeat(String)`

Worst Case: no repeat words in the message

Number of words in message (n)	Counted Steps
1	9
2	
3	
4	
5	
6	

Growth Function for *worst case* of `Speakers.exclamationMarkOnRepeat(String)` is:

Method `Speakers.exclamationMarkOnRepeat(String)`

Best Case: all words in the message are the same

Number of words in message (n)	Counted Steps
1	9
2	
3	
4	
5	
6	

Growth Function for `Speakers.exclamationMarkOnRepeat(String)` is:

Method `Speakers.exclamationMarkOnRepeatUsingSet(String)`

Average Case: a mixture of long and short words in the message

Number of words in message (n)	Counted Steps
1	10
2	
3	
4	
5	
6	

How many times did the debugger execute each line of the following piece of Java code?
Write out your answer in the red angled brackets (i.e. <>).

```
< >         String[] words = message.split(" ");
< >         Set<String> wordsUsed = new HashSet<>();
< >         String result = "";
< >         for(String word : words) {
< >             result += word;
< >             if(wordsUsed.contains(word)) {
< >                 result += "!";
                }  else {
< >                 wordsUsed.add(word);
                }
```

```
< >             result += " ";
            }
< >         return result;
```

Growth Function for `Speakers.exclamationMarkOnRepeatUsingSet(String)` is: