The learning objectives of this homework are

- to write a OO program from scratch: **no provided code** (almost),
- to gain experience using existing structures to build more complex ones and
- to improve your code design skills.

The design and implementation of the word generator program uses Java `ArrayList` and an `Association` class. We studied `ArrayList` in depth in class by doing our own implementation of it, and thus you know about the inner workings of this fundamental data structure and are aware of its performance for its method calls – we expect you to use `ArrayList` with the maturity of a developer that knows the various costs associated with its implementation. In terms of the `Association` implementation, you are required to implement a simple 2-tuple class `Pair`, a key-value association using generic notation.

**Problem Description**

This assignment comes from the world of narrow artificial intelligence. The idea is to read an input text, that is long and then use its properties to generate new, “machine generated” text. The results can make for interesting reading, but will not be *intelligent* enough to write a course term paper.

The method for generating the text uses simple probability to “train” our program to generate new words based on an input text. To do this, we read the text character by character and keep track of how often each three-character sequence appears. From this, we can compute the probability that a certain character will immediately follow three given characters.

For example, if the text is

`For the rain it raineth every day`

We have the three character sequence of letters `'rai'` appearing twice, both times immediately followed by an `'n'`. Additionally, we have the three character sequence of letters `'ain'` appearing twice, once followed by a space (`' '`), and once followed by an `'e'`.

So the probability that a space (`' '`) follows `'ain'` is 50%, and the probability that `'e'` follows `'ain'` is also 50%; the probability that any other characters follows `'ain'` is 0. Likewise, the probability that an `'n'` follows the character sequence `'rai'` is 100%, as all of the appearances of `'rai'` comes in the word `'rain'`.

The input text has to be processed and stored in a **structure** such that the probabilities are available for each unique three character sequence (**Hint: maybe collections of the patterns suffice**).

To machine generate the output text, two letters (for example, the first two in the input text, or two random characters) are picked to use as a beginning. Then subsequent characters are based on the preceding two characters and the probability information gathered from the input text.
When does the new text generation stop? The generation stops if either of the two following conditions is true:

1. A 3-character sequence for which there is no subsequent character is encountered. It will occur only if the last three characters of the input occur only in that location.

2. A total number of characters is reached. This maximum is specified as a constant, which is possibly overwritten by a parameter passed to the program.

This problem description uses a 3-character sequence as the seed to determine the next character. Provided the input was longer, a better text could be generated by using a 4, 5, or greater character sequence to determine the next character.

While you do not have to do this extension, you should realize that the integer 3 in the n-character sequence, \( n = 3 \), is a specific but arbitrary value. Once the program is complete and fully debugged it would be nice to increase this value down the road. Thus, a well-designed and robust program would use a constant, \( \text{SEED\_LENGTH} \), such that changing its value would automatically produce text that doesn’t have as many spelling mistakes! For your program you are required to make it work and test it with a 3-character sequence.

Design

You should think about the design of this program carefully before beginning any implementation. Combining the \texttt{ArrayList} and \texttt{Pair} will be required to create a new data structure: a table. A table stores and organizes the information to solve the word generation part effectively. The combination of \texttt{ArrayList} and \texttt{Pair} happens twice and at two levels. Indeed in the table for each row (which represents an unique key), the value is a frequency list, which is made of a second instance of an \texttt{ArrayList} of Pairs.

It is initially difficult to know how and where to approach the code. You need to decompose the problem in simple steps to code it reasonably easily and efficiently. You should not work on too many classes (i.e. three) at the same time when you are starting.

Pair

But first to get you started, why not implement \texttt{Pair}… We provide some guidance here: the comments of the class that reveal a specific design.

Open \texttt{Pair.java}, read through and complete while testing as you go.

Note - it implements \texttt{Map.Entry<K,V>} from the Java 7+ API, - the \texttt{hashCode} method is given and - a \texttt{main} exists.

Now, I encourage you to write a design document, which you should bring when you seek help to get feedback on your plan (more information below). Regardless, if you hesitate where to start please come and see us.

Table

Specifically you will need to implement a table, which constitutes a good data structure to solve the word generator problem. The primary functionality of your program is

1. to update the probabilities in a table, given a new quadruple of characters, and
2. to select a next character, given a triple of characters and the probabilities stored in the table.
A **Table** class stores in an **ArrayList** of **Pairs** mappings between 3-character sequences with possible subsequent characters. Each **Pair** should have a 3-character sequences (stored as a **String**) as its key, and a **FrequencyList** as its value. (As mentioned above the key length could be a constant so you can easily modify your program later to generate less gibberish text.)

**Frequency List**

Each frequency list should be an object of another class, **FrequencyList**, which you will define. **FrequencyList** is the second data structure you will implement (which you should not start coding before the class **Table** is under control, meaning you debugged its initial state).

The **FrequencyList** should keep track of each character that appeared after the given 3-character pair, along with the number of times it appeared. While there are many ways to implement the **FrequencyList** for now you should use another **ArrayList** of **Pairs**. (If you think a hashmap is better: you are right but we have not seen in class yet. Talk to me if you have that knowledge and that is your preference.)

In the case of the **FrequencyList** the **Pairs** have a key of a single character (which could be stored as a **String** for consistency with the pair key of the table) and a value which is the count (stored as an **Integer**) of the number of times that letter occurred after the 3-character sequence with which this **FrequencyList** is associated with. Think carefully about what methods the frequency list needs to support and which other instance variables might be useful.

The data structure design built from these two classes, the **Table** and the **FrequencyList**, has the benefit of having only as many entries as necessary for the given input text. You should have the table implemented before being concerned with the **FrequencyList**.

**Client class: WordGen**

Your main method for the program should be written in a client class, **WordGen**, which is responsible to get the input and display the output. Use a hard-coded input string until you complete the **Table** and the **FrequencyList** classes. **WordGen** uses the two data structure classes you are writing

1. to build the table,
2. to print the table built and finally
3. to print out a randomly-generated string based on the character sequence probabilities from the input text.

All I/O (input and output) happen from the **main** method of the **WordGen** class and not much more code belong there. If your code for the output is long it is because you are doing too much in the client and it is ineffective, i.e., bad design. You should not have to handle any frequency lists in **WordGen**: it is the responsibility of the table to deal with the frequency lists for example. Think about which important behaviors belong to which data structure object.

**Design Document:**

Read this document all the way through at least three times and take notes to help you writing a design document. I encourage you to first draw pictures/diagrams/sketches to illustrate how an example string is sequentially processed to create the table data. Approach the input process by hand in a **systematic manner**. For a small input, i.e. the short sentence we gave you, how is the table simply and progressively produced? Focusing on the distinct row first, later on the right hand side, which is the **FrequencyList**toString(). Once you own the pattern to process the input write elements important to the **Table** and **FrequencyList** classes in your design document. Specifically
include descriptions of the classes, instance variables and important instance methods,
briefly describe pseudo-code for the main functionality of your program (the items 1. and 2. listed
above in the Table section) and
describe simple data and tests you are planning to use to check the correctness of the program as you
gradually develop it.

For the different data representation (stored in the table and the frequency lists) think about where
uniqueness is fundamental and document that information.

Input/Output

Hard-coded

First develop and debug your program using as input a String constant (e.g., He'll keep calling me,
he'll keep calling me until I come over. He'll make me feel guilty. This is uh... This is
ridiculous, ok I'll go, I'll go, I'll go, I'll go. What - I'll go) until it works
properly. Print on the console the created table in a format that facilitates to check the correctness of your
program at storing the statistics built from the input.

It is important to consider bottom-up design. Your goal is to have the table working before the generator
part. However, you should not start with implementing the FrequencyList, but rather focus on the table
rows first. Work on having the keys for each row (that are unique) in the table processing the output fully.
Once you have every distinct rows work on the values, which are the FrequencyList, getting their own keys
before the proper count/statistics,
The milestone for this assignment will consist of the output of your table based on your hard-coded
String, and is due on Thursday, November 8th, at 10:00 P.M.. Your will post your table’s output in
forum your class’ Moodle page. Do not include any code in your forum post, only the print out of your table
– you’re encouraged to demo your milestone to your instructor during lab to get feedback. Use your own
preferred input String, not any of the provided ones, and it has to be longer than 150 characters long.
Include your input string at the top of your post
Once you have the output of the table in an easy to follow format, you should implement the machine
generated text corresponding to the table frequencies.

The generated text starts with a pair of letters. I also suggest that you initially use a hard-coded version of
this 2-character sequence. Later on it may be provided as a command-line argument or picked randomly pair
from the table key if none is provided, as described below.

As stated in the problem description, you should generate text until there is either no next character available
or until you have produced a maximum number of characters, which is a constant of the WordGen by default.

Program parameters (review)

After your program can build a table and generate text using hard-coded input and a default 2-character pair
to start the output you have to augment its capability by providing the user with controls using command-line
arguments. On jEdit a user may type

```
java WordGen arg1 arg2 arg3
```

Those arguments are the Strings parameters to the main method: String[] args. It is possible to omit all
or some of them and your program should still run using default values. Note that arg3 can only be used as
the third string in the sequence. If only one argument string is provided it is consider as arg1 and nothing
else.
• The first argument, \texttt{arg1}, is a filename from which to read the input text.
• The second one, \texttt{arg2}, is the two letters used to start the generated text.
• The third, \texttt{arg3}, is an integer used to stop the generated text.

When arguments are not provided (none are or the latter ones are missing), your program should use default values. Code to read from a file is provided commented in the starting code. Feel free to try the provided text file, e.g. \texttt{as\_you\_like\_it.txt} and other text of your choice. The file has to be in plain text, or format characters may cause issues.

Since your \texttt{readme} has to contain the tests you used, document the testing you do as you develop your program in a \texttt{readme}. Another “interesting” input to try out is the Java source code for one of the classes you have implemented.

\textbf{Notes}

• When debugging the table construction, keep in mind that you can use the built-in \texttt{toString} methods in \texttt{ArrayList} and \texttt{Pair} very easily, but they may not present the information in a convenient form. Write custom \texttt{toString} methods for your \texttt{FrequencyList} and \texttt{Table} classes.

• Treat spaces and punctuation just like any other characters. The word generator program should be case-sensitive.

\textbf{Grading}

Make sure for each class to include a general description at the top of the file, with authorship and date information.

Your program design is important in this homework, you should consider and reflect on the following. Document your thoughts about any in your \texttt{readme}.

• How efficient is your code?
  – Think about where it is best to add in an \texttt{ArrayList}.
  – Do not do the same thing twice: it is inefficient.
  – Avoid extra looping. Use a loop when needed, do not guess and make it work.

• How is the code spread among the three classes?
  – Each class has well-defined functionalities.
  – No class is accessing things that are too far away (in the vertical \texttt{has-a} composition design). It is a sign of a problem.

For program style good formatting, appropriate namings and complete Javadoc are expected.

The \texttt{readme} should include

1. your student information at the top,
2. sufficient examples to demonstrate how you tested your program. For some input and generated output—short and long ones—write up a paragraph with the observations you made about the generation correctness. For long examples feel free to truncate the output or to submit as separated files included in the submission.
3. one or two paragraphs commenting about your program design. You should describe the strong points, what aspects you are happy/proud of and which if any you think could be improved.
If you have not fully implemented the functionality of the word generator, list in your `readme` the parts that work (and how to test them) and the ones you attempted so as to receive partial credits.
Your grade will be based on the following weights:

- **First Output of Table due November 8th, at 10:00 p.m**: 10
- **Table**: 25
- **Generated text**: 25
- **Command line arguments**: 10
- **Program design/style including comments**: 15
- **readme with tests**: 15