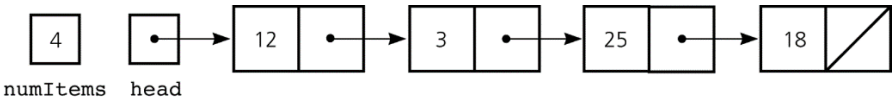
SYSC 2100, Summer 2017
Assignment 2: Implementing and Comparing Lists
Due: July 19, 2017

Problem 1 (4 marks)
Implement a reference-based linked list `ListReferenceBased` from the scratch without using the implementations in the Java Collections Framework. This linked list implementation should be capable of performing the following operations.

- `isEmpty`
- `size`
- `add`
- `remove`
- `get`
- `removeAll`



After successful implementation of the linked list, create the above linked list within a client program and demonstrate the following operations.

- Display the items in the linked list
  `4 Items in the linked list: 12, 3, 25, 18`
- Add Integer Object valued 13 at index 0 and display the items in the linked list
  `5 Items in the linked list: 13, 12, 3, 25, 18`
- Add Integer Object valued 17 at index 2 and display the items in the linked list
  `6 Items in the linked list: 13, 12, 17, 3, 25, 18`
- Remove Integer Object at index 4 and display the items in the linked list
  `5 Items in the linked list: 13, 12, 17, 3, 18`

Hints:

1. Refer slides on Linked list and the text book.

Submission Requirements: Submit your assignment (the source files) using ***cuLearn***. Your program should compile and run as is in the default lab environment, and the code should be well documented. Submit all files without using any archive or compression as separate files. The main program should be called **TestListReferenceBased.java**, if you need to define additional classes etc., you are free to name them according to your own needs. But the TA(s) should be able to run your application by entering **`java TestListReferenceBased`** on a command-line.

Marks for Problem 1 will be based on:

	25%	50%	75%	100%
Program (4 marks)	Compiles	Runs	Correct, Output	Correct implementation of Reference-based linked list

Problem 2 (16 marks)
One of the recurring themes in the course is that the same abstract data type can be implemented in different ways, using different underlying data structures. The choice of data structures and algorithms however does have an impact on the performance/efficiency of the implemented abstract data type. While we haven't formally introduced the notions of algorithm complexity yet, we already discussed how different choices for a list (linked structure or array) impacts the efficiency of specific operations. In this assignment, you are to experimentally confirm that indeed the choice of a specific implementation for a given abstract data type impacts the runtime performance, even when giving the same results.

We will assume that a string is implemented as a `List` of characters. Write a program that prompts the user for a file name and opens that text file. Example text files to be used as input are provided in the **cuLearn** (two books, one the English translation of Victor Hugo's Les Miserables, the other Charles Dickens' A Tale of Two Cities). You can also download other large text files from repositories such as Project Gutenberg (www.gutenberg.org). Then also prompt the user for a string A. Read in the text file string by string and determine whether each string in that text file contains your input string A as a substring. Count how often this is the case and display that number at the end. Also measure the time that elapsed to process the complete file and display this as well.

To show the difference between the various list implementations of the ADT `List` provided in the Java Collections Framework (`ArrayList` and `LinkedList`), repeat the task, the first time representing all strings as an `ArrayList` and the second time as a `LinkedList`. Substring matching is a surprisingly complex problem (when striving for efficiency), we may revisit this later once we introduced more ADTs. For the time being, use the following brute-force substring matching algorithm (it is okay if in your solution you do not declare `findBrute()` to be static). The code can also be downloaded as a textfile from the **cuLearn**.

```
/*
 * Returns the lowest index at which substring pattern begins in text (or
 * else -1).
 */
private static int findBrute(List<Character> text, List<Character> pattern) {
            int n = text.size();
            int m = pattern.size();
            for (int i = 0; i <= n - m; i++) { // try every starting index
                                               // within text
                  int k = 0; // k is index into pattern
                  while (k < m && text.get(i + k) == pattern.get(k))
                        // kth character of pattern matches
                        k++;
                  if (k == m) // if we reach the end of the pattern,
                        return i; // substring text[i..i+m-1] is a match
            }
            return -1; // search failed
}
```

Your program should produce output similar to the one shown below (using `Les Miserable` as input file and `Javert` as substring). Obviously the timing is hardware-dependent and may well differ on different computers and implementations. To not bias your expectations, I've xxx-ed out the values, your final solution should provide some actual measurement here… ☺. The counts though should be the same across all platforms and implementations.

```
Please enter the path for the input file: LesMis.txt
Enter the pattern to look for: Javert
Using ArrayLists: 457 matches, derived in xxx.x milliseconds.
Using LinkedLists: 457 matches, derived in xxx.x milliseconds.
```

Hints:

1. Make sure that you use the correct data types (`LinkedList` and `ArrayList`) for both the string being read in from the file and the string A, where appropriate.
2. To measure elapsed time, use `System.currentTimeMillis()` twice: once before you start processing the input file, once after. The difference between these two timestamps gives you the elapsed time in milliseconds and provides a rough approximation for the algorithm runtime.
3. We use strings in the common sense of the word: sequences of characters separated by whitespace. So you need to read in and parse the textfile accordingly.
4. When matching the substring, it has to be an exact, case-sensitive match. For example, upper and lower case characters are not matching each other.
5. Three helper methods named `openFile(), convertStringToList(),` and `readAndMatchDocument()` are provided. `openFile()` repeatedly prompt the user for filename until a file with such a name exists and can be opened. `convertStringToList()` converts a string to a list. `readAndMatchDocument()` iterates over all strings in input file to determine whether the input string is a substring in any of these strings.

Submission Requirements: Submit your assignment (the source files) using ***cuLearn***. Your program should compile and run as is in the default lab environment, and the code should be well documented. Submit all files without using any archive or compression as separate files. The main program should be called **CountSubstrings.java**, if you need to define additional classes etc., you are free to name them according to your own needs. But the TA(s) should be able to run your application by entering **`java CounSubstrings`** on a command-line.

Marks for Problem 2 will be based on:

	25%	50%	75%	100%
Program (10 marks)	Compiles	Runs	Correct count of test string.	Program shows difference in run times.
Following good coding style (2 marks)	Hard to decipher	Can follow flow of control	Easy to read, meaningful variable and function names	Good error handling such as gracefully catching invalid input (no stack traces).
Comments (2 marks)	Occasional	Lots, but extraneous	Few, but meaningful	Sufficient and high-quality comments, both in-line and method declaration
Completeness of your submission (2 marks)	Something in CULearn	Lots of effort to extract and run	Some effort to extract and run	Adhering to the submission requirements

---

The due date is based on the time of the ***cuLearn*** server and will be strictly enforced. If you are concerned about missing the deadline, here is a tip: multiple submissions are allowed. So you can always submit a (partial) solution early, and resubmit an improved solution later. This way, you will reduce the risk of running late, for whatever reason (slow computers/networks, unsynchronized clocks, failure of the Internet connection at home, etc.).

In ***cuLearn***, you can manage the submission until the deadline, taking it back, deleting/adding files, etc, and resubmitting it. The system also provides online feedback whether you submitted something for an assignment. It may take a while to learn the submission process, so I would encourage you to experiment with it early and contact the TA(s) in case you have problems, as only assignments properly and timely submitted using ***cuLearn*** will be marked and will earn you assignment credits.