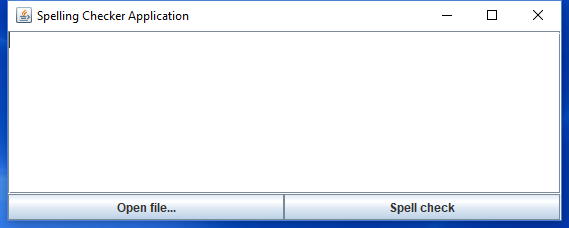
**Data Structures and Concurrency : Continuous Assessment 1**

**October 2018**

See folder CA 1 October 2018 on X:\lab\CathrynCasey\Data Structures & Concurrency.

CA1 is an IntelliJ project that consists of a spell checking application. In this application, you will be able to open a file, which will then be displayed in a text area. You will have the option to do a spell check on it.



The GUI has a text area and 2 buttons:

* Open file… allows you open a file. The file is then displayed in the text area
* Spell check – performs a spell check on the contents of the text area and outputs to the console the list of misspelt words.

To perform the spelling, it uses a dictionary of words: words\_alpha.txt from <https://github.com/dwyl/english-words>

Note: This file was not in sorted order. I have sorted it!!

The application has the following packages:

* data package with the dictionary – words\_alpha.txt
* src\gui package with classes:
  + TextEditorMain – this has the main method
  + TextEditorFrame – this is the GUI class
* src\spelling package:
  + with interface:
    - Dictionary
  + with classes
    - different implementations of Dictionary:
      * DictionaryArrayList
      * etc
    - AVLTree
    - SpellChecker

**Notes:**

**1.** You have been given AVLTree – see <https://en.wikipedia.org/wiki/AVL_tree>

The code is similar to our BinarySearchTree. But the insert method, as well as adding a new node, rebalances the tree, if necessary. The Node class has some extra properties:

* parent – a reference to the parent node
* balance – the balance factor of the node.

It has an add method to be consistent with the other Collection classes. The add method simply calls the insert method.

**2.** You will use TreeSet from the java collection classes. As mentioned in class, this is a binary search tree, but like AVLTree, it is self-balancing.

**3.** doSpellCheck method of SpellChecker class uses a regular expression to get an array of words from its text argument.

The regular expression is "[^A-Za-z0-9']+"

This parses text into words where the character pattern that separates words is any sequence of characters other than letters, numbers and apostrophe

For regular expressions see <https://javatutorial.net/basic-java-regular-expressions>

**4.** Profiling of doSpellCheck method

Here you will use VisualVM, as well as timing the method like we did in Lab 1 using System.*currentTimeMillis*();

To do the profiling you will need to run the code once for each different type of dictionary (ArrayList, LinkedList etc), i.e. commenting out but all of one of the dictionaries. As given, it uses the DictionaryArrayList.

**public class** TextEditorMain  
{   
 **public static void** main(String[] args)  
 {  
 Dictionary dictionary = **new** DictionaryArrayList(**"data/words\_alpha.txt"**);  
 *//Dictionary dictionary = new DictionaryLinkedList("data/words\_alpha.txt");  
 //Dictionary dictionary = new DictionaryAVLTree("data/words\_alpha.txt");  
 //Dictionary dictionary = new DictionaryTreeSet("data/words\_alpha.txt");*

To perform profiling, run the application and open a large text file and run spell check on it. You will need to use a relatively large file. The size will depend on the system that you are running your code on.

Files are available to download from <http://www.gutenberg.org/> or <http://www.gutenberg.org/browse/scores/top>

Download the [Plain Text UTF-8](http://www.gutenberg.org/files/11/11-0.txt) version. One of the smaller ones is “Alice's Adventures in Wonderland by Lewis Carroll” which is 170kB. You can save it to any directory that you like. In the application, you will be able to navigate to the file. The code uses FileChooser class for this.

(See the following for using JFileChooser: [https://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html](https://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html%20%20) )

Experiment with files of different sizes in order to get useful results from the VisualVM profiler. When experimenting, use the DictionaryLinkedList version of the dictionary, as this takes the longest time.

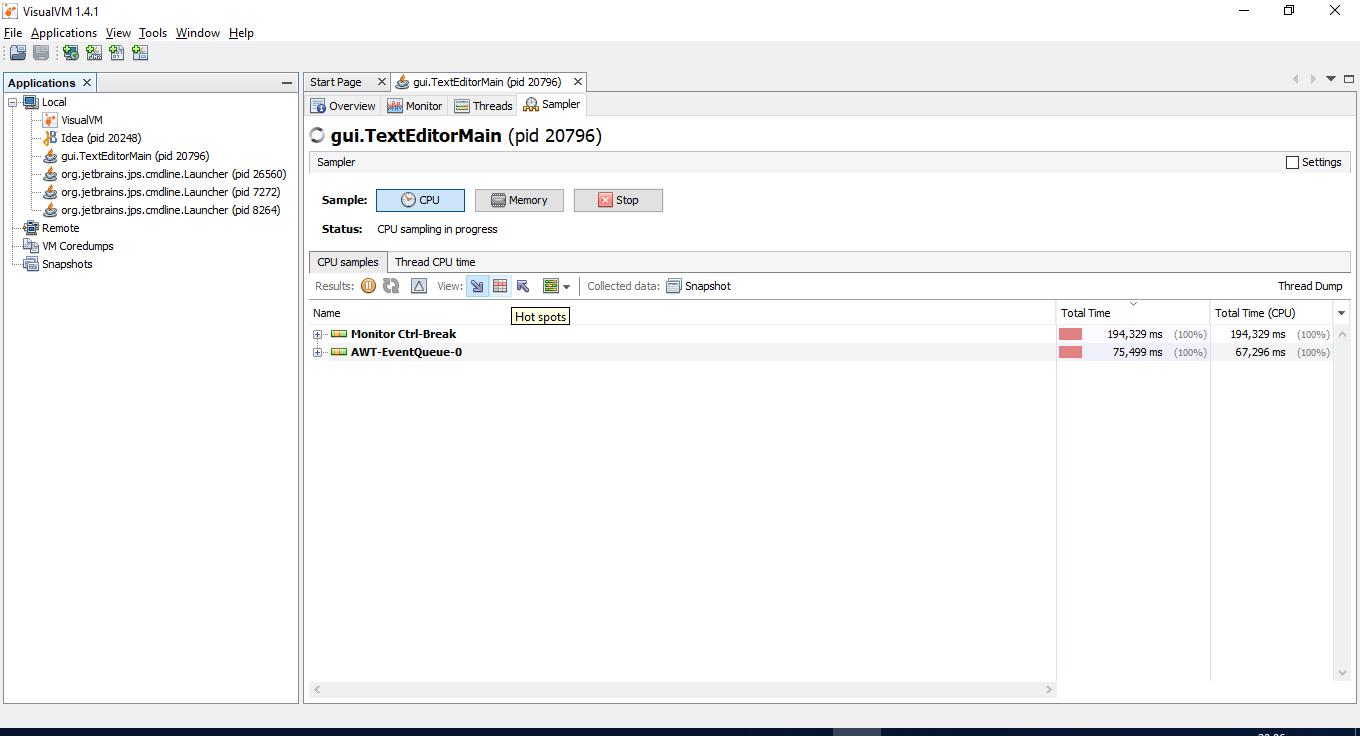
**Using VisualVM**

In IntelliJ, choose

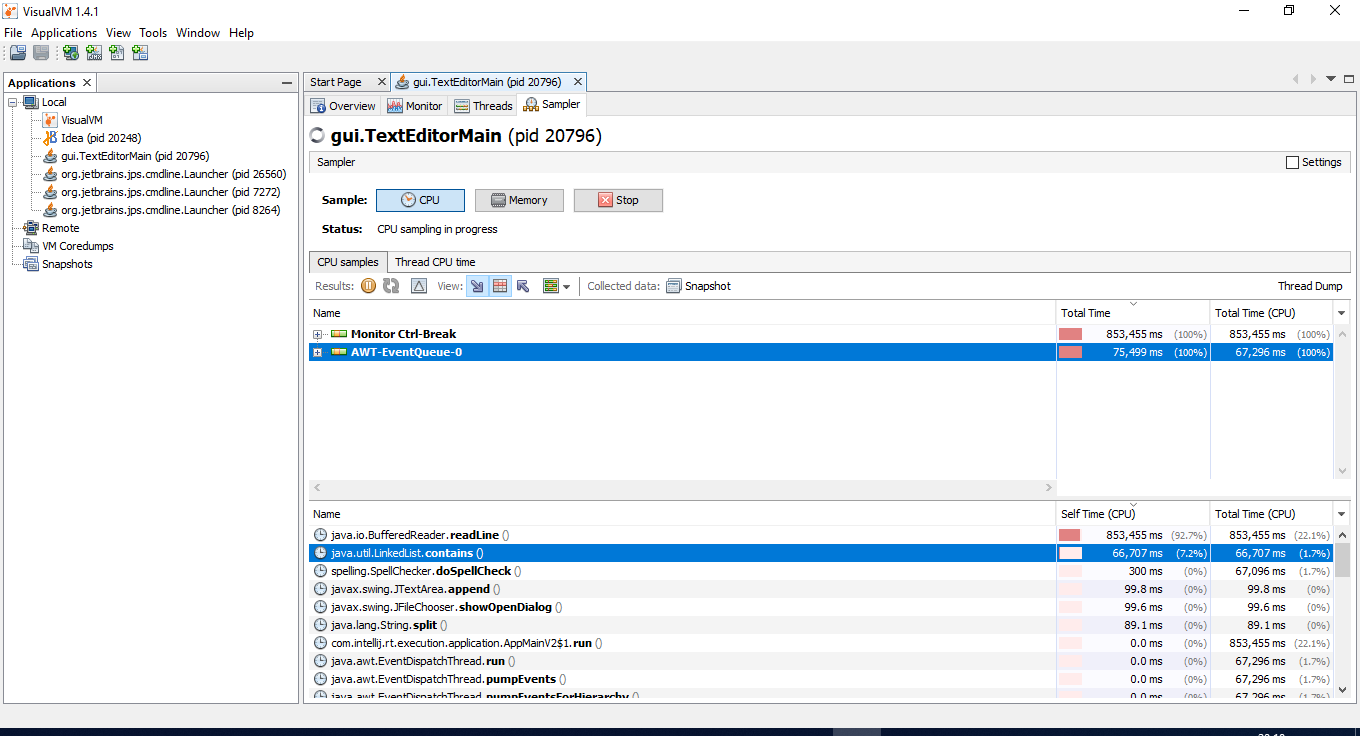
Run with VisualVM ‘TextEditor Main’

Choose Sample and then CPU as you did before – see “Lab3 – Profiling with IntelliJ”.

Then open the large file and do the spell check. After this has completed look at the profiler. In CPU samples, you need only consider AWT-EventQueue-0 thread. Look at hot spots - click here



Then look for occurrences of doSpellCheck (SpellChecker class) or isWord (Dictionary…. class) method or contains or relevant method (from one of the collection classes).



Right-click on the highlighted line (contains or other method). Choose “Find in Forward Calls” and then navigate to the sequence of calls for doSpellCheck to contains (or relevant) method call. Note the time for the contains (or relevant) method. This is the value you will insert in the table Time taken to do the spell checkin the CA report under the column labelled “Time from profiler”.

Note: if none of these methods appear for some of the dictionaries, you may presume the time taken is 0 i.e. it is less than a millisecond.