**Lab 5\_ 2**

**Profiling in IntelliJ**

“Profiling is a type of runtime analysis that operates on large amounts of runtime data and gives you a birds-eye view of what is happening inside a process. The collected data relates to various aspects of program operation, such as CPU usage, memory allocation, and interaction between the threads.”

We will look at CPU usage.

We will use IntelliJ’s built-in profiler which is **much** improved since the last version.

See <https://www.jetbrains.com/help/idea/profiler-intro.html#run-with-profiler>

and

<https://www.jetbrains.com/help/idea/tutorial-get-started-with-profiling.html>

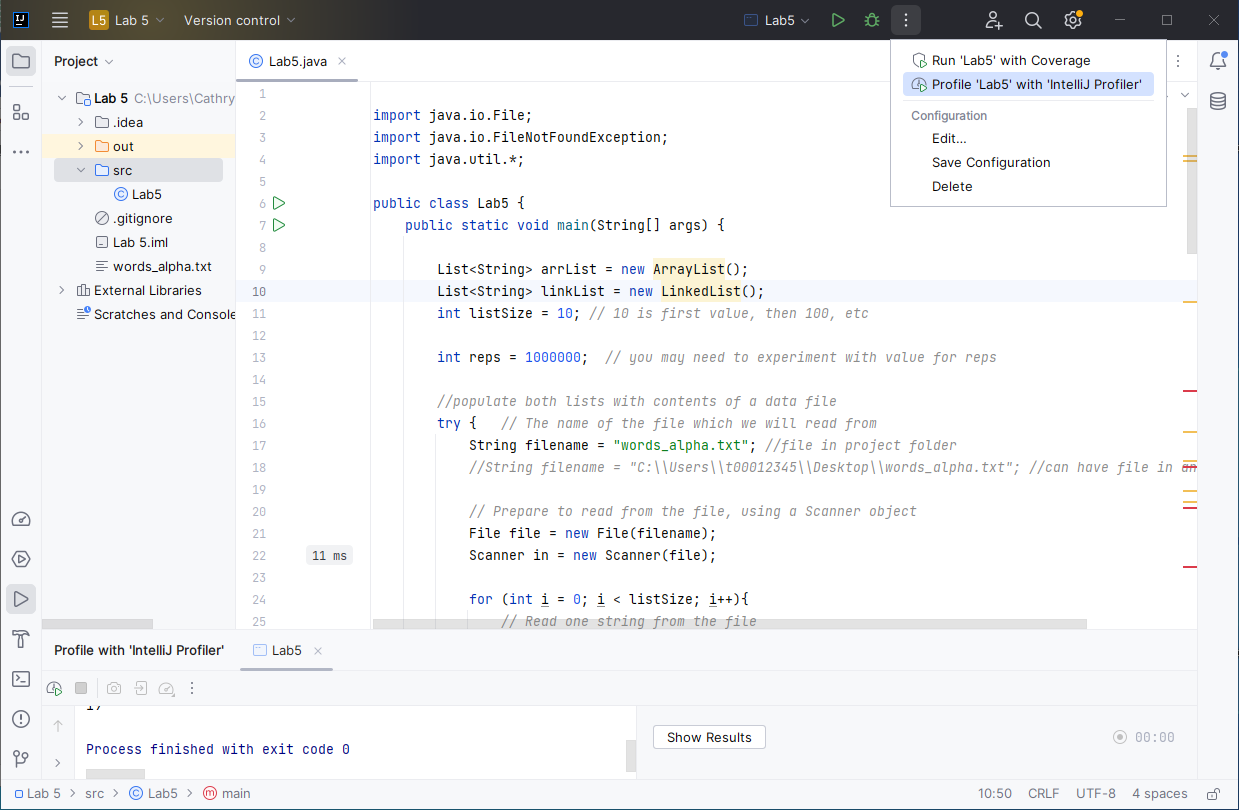
To test the profiling software, use Lab5.java. This program looks at the efficiency of get() method of ArrayList and LinkedList:

[E](https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/List.html) get​(int index)

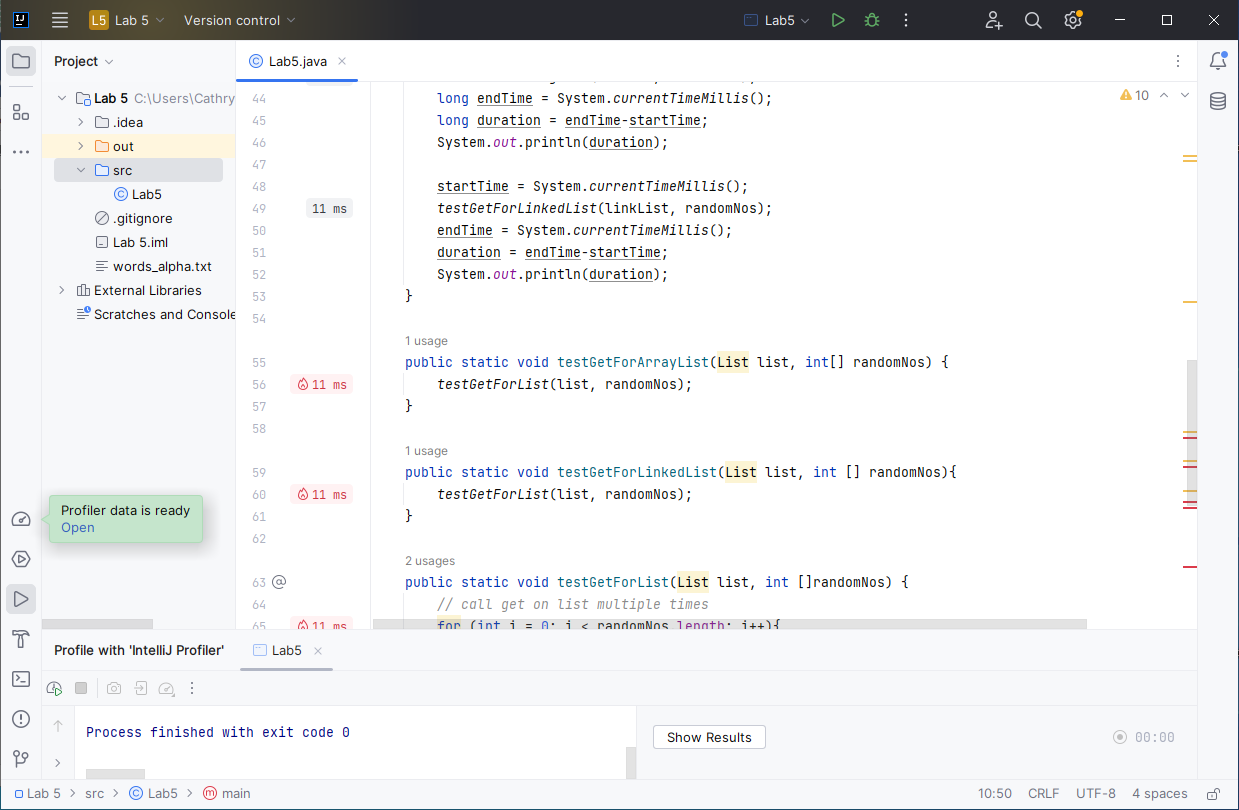
Returns the element at the specified position in this list.

Run the program in the usual way and check that it works. It should output the time for multiple “get()”s for ArrayList and LinkedList. This may be **slow** to run for LinkedList. The time for ArrayList may be very small if you run it for small number of iterations. You will need to experiment with the value of reps that you use, so that it will allow you to compare the different sizes of List implementation used (10, 100, 1000, .. 100,000 etc). Remember size of word\_alpha.txt file is 370099.

Then start profiling – choose Profile ‘Lab5’ with ‘IntelliJ Profiler’:



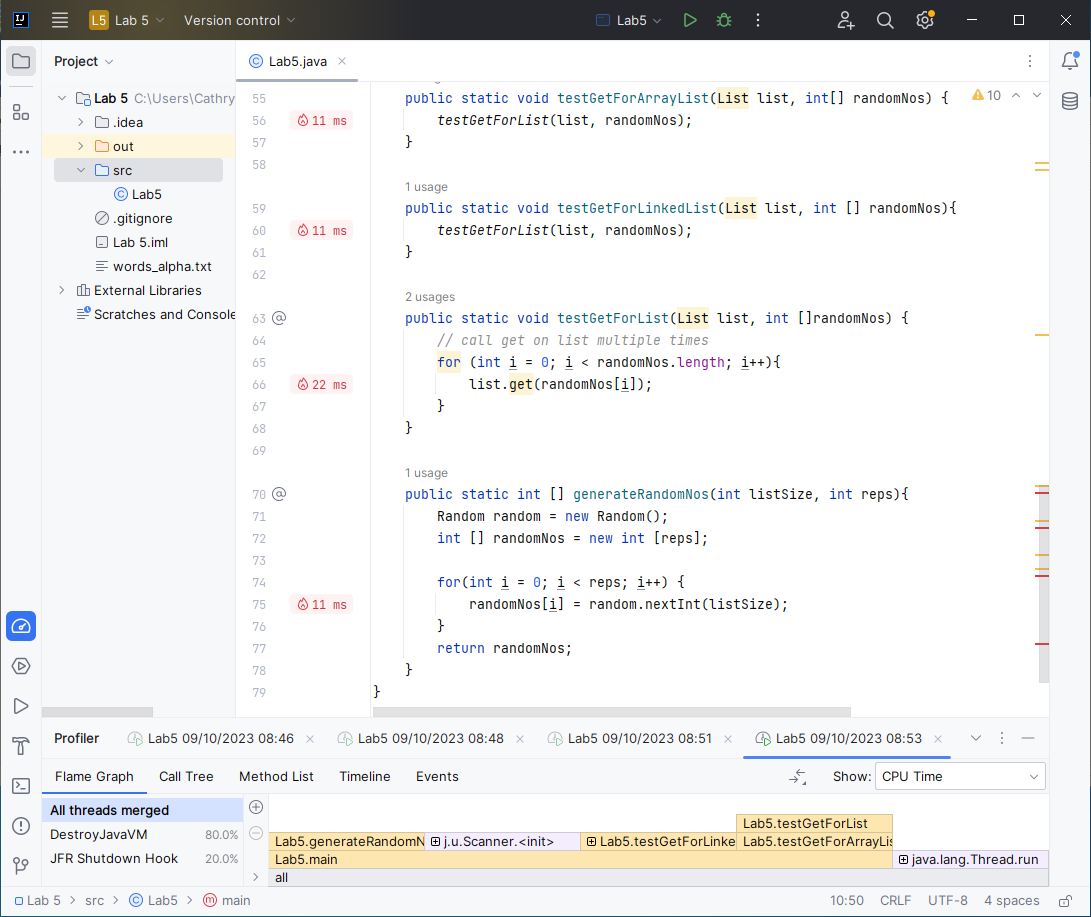
In the gutter (to the left of the code), you will see time taken for some of the code. You will see a flame symbol beside large values (all in red) to highlight the code that is using the most CPU:



At the bottom left of the screen, you will see a message ‘Profiler data is ready Open’ (in green).

Click ‘Open’ here and then you will have various options.

Look at the Flame Graph:



The yellow part of the stack is Java code, the blue is native method invocations – this of course depends on what colour you use in the Theme in:

* Settings->Appearance & Behaviour->Appearance.

You can hover over a method to see % of time and actual time (in ms) spent in methods.

Take a look also at Method List and the other options. Can you find any other interesting info?

Run your code with the same value for reps but different values for listSize (10, 100, etc)

If you have time, add code to compare set(), insert() and remove() methods.

Fill in the following table (same as ‘Lab 5\_1 Looking at Java’s ArrayList and LinkedList.docx’)

Reps 1000

ArrayList

|  |  |
| --- | --- |
| size | get() |
| 10  100  1000  10000  100000 | 0.3168 ms  1.5768 ms  5.3405 ms  3.7176 ms  n/a |

LinkedList

|  |  |
| --- | --- |
| size | get() |
| 10  100  1000  10000  100000 | 1.0226ms  1.9911 ms  273.6641 ms  34109.8979ms  n/a |

Here is graph created by me in Python using Matplotlib

A graph with a red line

Description automatically generated

Obtained with \_\_\_\_\_\_**11th Gen Intel(R) Core(TM) i7-1165G7**\_\_\_\_\_ processor, \_\_\_\_\_**2.80 GHz** \_\_\_\_ GHz, Java Version \_**17**\_\_, Windows 11 (or specify if not…)

How do they compare with the results you got in ‘Lab 5\_1 Looking at Java’s ArrayList and LinkedList.docx’?

What are the Big Oh values for

1. ArrayList get() method: Big O(1)
2. LinkedList get() method: Big O(n)

Are the timings consistent with the Big Oh values? \_\_Yes, ArrayList get() method remains stable reflecting its O(1) complexity. LinkedList get() method becomes very slow for larger sizes, reflecting its O(n) complexity. \_\_\_\_\_\_\_\_\_\_\_

Comment on results \_\_\_\_  
ArrayList behavior is quite consistent for the get() operation; its access time remains almost constant for list sizes. This will make ArrayList more suitable for situations when random access is needed frequently.  
  
The get() operation does rather poorly for LinkedList, especially as the list sizes get larger, since each access takes a little longer with traversal. This large increase in time for large list sizes makes LinkedList a very poor candidate for operations that call for a lot of random accesses; it does much better under other use scenarios, such as frequent insertions or deletions. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Note: You can also look at Memory to analyse the memory usage of the application.