Unit 3. Final exercise

Password hacker

Service and Process Programming

Arturo Bernal
Nacho Iborra

IES San Vicente

Index of contents

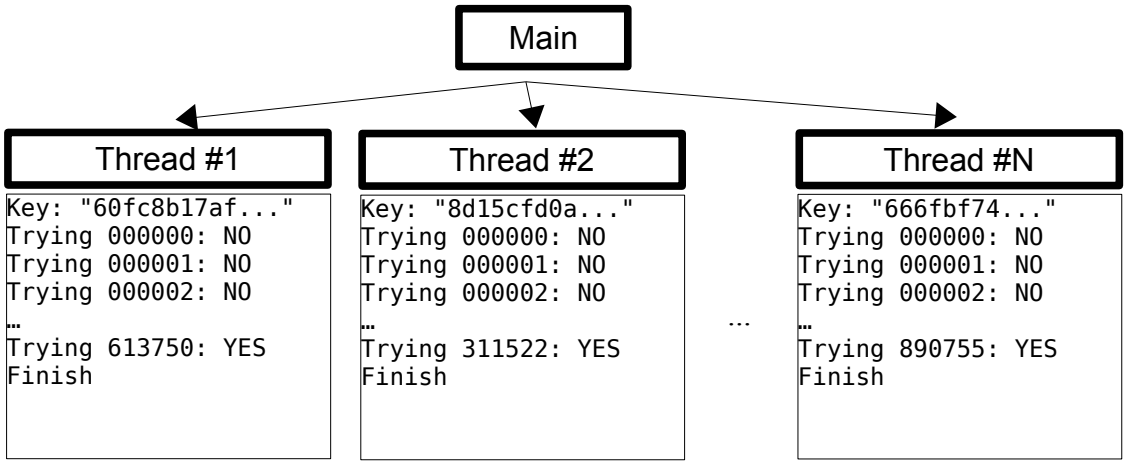
1. Introduction

In this final exercise we are going to implement an application that gets a list of passwords encrypted with an SHA algorithm, and launches a list of threads to try to decrypt each password by using brute force.

The application will read a text file containing N lines with all the passwords encrypted (one in each line):

```
60fc8b17af3847d62eb2d1533c09962adf2c92b94e8a323e6c18eccbbeacc249
8d15cfd0a7e4e8136f3eaac1157442a67c07b5893fa91f6219267192077f5174
0984be9625bf1755bff910be7c90b3aada3955c94b3f4e553d51aeddaa72622b
9886bf2e8d39af685e01bbdccadfb703972801754ac62dcc24eddd3d07b79a90
882df92801f27059a528e73e6c47898abc290974efef4ab806c94dc89834bb44
...
```

Then, it will create and launch a thread to break each password (N threads in total). We are going to assume that passwords consist of a sequence of **6 digits**, so threads only have to go from "000000" to "999999" and check if the SHA encryption of the current number is the one they are trying to discover.



So, the code of every thread will be the same: explore all numbers from 000000 until it gets to the number whose SHA key is the one it is looking for.

1.1. Libraries and code structures

1.1.1. Encrypting passwords with Apache Commons Codec

We are going to rely on **Apache Commons Codec** library (the one that you used in Unit 2 final exercise), and *DigestUtils.sha256Hex* method (the same method that you used in that final exercise) to encrypt passwords. So we only have to check if:

```
if (DigestUtils.sha256Hex(currentNumber).equals(key))
    … // Found!!
```

1.1.2. Formatting numbers with 6 integer digits

You may need to transform an integer into a 6-digit string. You can use the *DecimalFormat* class to help you with this:

```
int number = 23;
System.out.print(new DecimalFormat("000000").format(number));
// The last instruction prints "000023"
```

1.2. Introducing the projects

We are going to solve this problem in two different ways, by implementing two separate projects:

- A Java console application (not JavaFX) that gets the passwords file and creates and launches the threads by using *Callable* and/or *CompletableFuture* elements
- A JavaFX application that loads the passwords in a list and launches the threads by using the JavaFX concurrency framework.

Both projects will be Gradle projects, so that dependency management with Apache Commons Codec library will be easier to import/export. No code will be shared between these two projects, although some parts may be the same (you will need to duplicate this code in both projects)

In the following sections you will see how to create and configure each project, and what you are expected to implement in each one.

2. PasswordHackConsole

Create a Gradle project called **PasswordHackConsole**. The main class should be in **unit3.finalexercise** package.

First of all read all encrypted passwords from **passwords.txt** file. For every password create a CompletableFuture that tries to decrypt that password (If no password matches, return "NOT FOUND"). After each CompletableFuture finishes and returns the resulting password, process that result (using CompletableFuture's methods → .then...()) and store it in a **ConcurrentMap<String, String>** (**key** → encrypted password, **value** → decrypted password).

Run all these CompletableFuture tasks at the same time using **allOf** method. After all these tasks have finished, run another task that will print the results (stored in the Map), using a CompletableFuture method (not in the main thread).

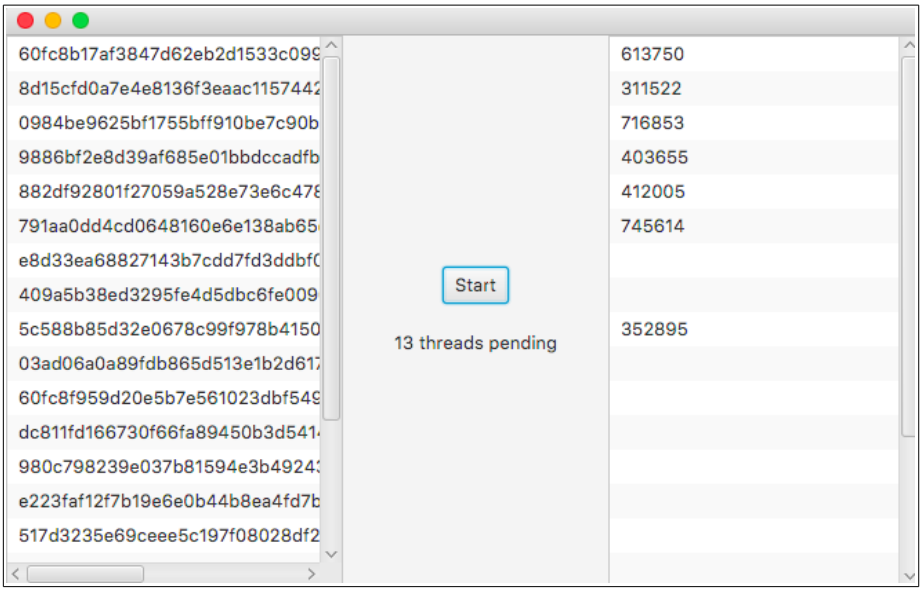
Create also a task inside an **ScheduledExecutorService**. This task will be repeated every second, and will print how many passwords have been decrypted (check the Map's size). When all passwords are decrypted, shut down this executor.

Output example:

```
0 decoded: 0%
0 decoded: 0%
3 decoded: 15%
4 decoded: 20%
5 decoded: 25%
8 decoded: 40%
9 decoded: 45%
12 decoded: 60%
13 decoded: 65%
14 decoded: 70%
15 decoded: 75%
19 decoded: 95%
100% decoded
8d15cfd0a7e4e8136f3eaac1157442a67c07b5893fa91f6219267192077f5174 = 311522
980c798239e037b81594e3b4924394030dc966d02c96effd5c750a703331225e = 608967
5c588b85d32e0678c99f978b415047cab1517df6dbc50c912ce02cdc4fb01bfd = 352895
517d3235e69ceee5c197f08028df2c7a2ec49e2d61e5825a17ee973ab977ed3d = 950814
409a5b38ed3295fe4d5dbc6fe00968d90ef27cdb40095bef66252fac99cee7dd = 875206
c3397a49989eeab189cfbf620946a6f300270200241736f3dccedf1a8daa63ad = 475833
e223faf12f7b19e6e0b44b8ea4fd7b55b6f632d5e95ff83d8ba66e39d9bf31d5 = 917123
03ad06a0a89fdb865d513e1b2d617095c3d729e6b019772769a804ecd8954829 = 983568
60fc8f959d20e5b7e561023dbf549b772b7ce732a9fe66622a8a896dddf05d2f = 623291
dc811fd166730f66fa89450b3d5414cc815ca0d9455b03a78d05b0f7894418da = 599171
12ac74f1c6f1110554fd06d93977b1c2c13e4b22224c0ab7f00471a63ab7ad37 = 921076
791aa0dd4cd0648160e6e138ab65c6d97629179ef1255521f4698d7c17b81b00 = 745614
0984be9625bf1755bff910be7c90b3aada3955c94b3f4e553d51aeddaa72622b = 716853
9886bf2e8d39af685e01bbdccadfb703972801754ac62dcc24eddd3d07b79a90 = 403655
882df92801f27059a528e73e6c47898abc290974efef4ab806c94dc89834bb44 = 412005
60fc8b17af3847d62eb2d1533c09962adf2c92b94e8a323e6c18eccbbeacc249 = 613750
e8d33ea68827143b7cdd7fd3ddbf0322195e6eed6513f6acda4443e8195e66db = 642429
779377df1caa90d556bcf1919d92bf48c6c784b4d9c7209d5f9376c0420ee924 = 449874
1a11fc5de8aa09c1f9feae81d8f883675cb8387137b6f94647535571e4b9be8f = 976385
666fbf7447bebd344215a3fff64ad33bbd95f4fa87cc2fc660de0f23af5479d7 = 890755
```

3. PasswordHackFX

In this section we are going to see the steps that you must follow to implement a JavaFX application to break the list of passwords provided. The general appearance of the application must be like this:



Use a *BorderPane* as general container, and place a *ListView* at the left and right sections. Then, use a *VBox* container and place the button and the label in the center section.

3.1. Setting up the project

Create a single Gradle project called **PasswordHackFX**, with a package called *passwordhackfx*. Then, create a *Main JavaFX class* called *PasswordHackFX,* and an FXML file with its corresponding controller.

Remember to define the main class in the *build.gradle* file, along with the dependency with the Apache Commons Codec library, as you did in previous project, or in Unit 2 final exercise.

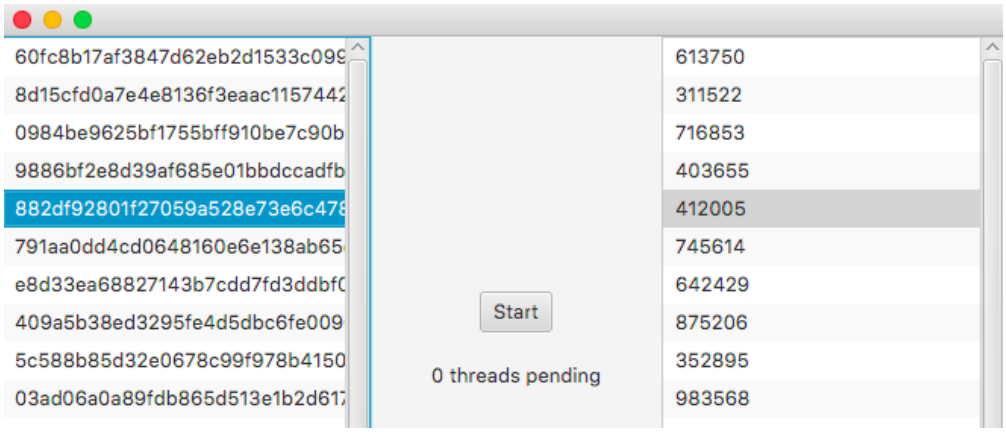
3.2. Application's behavior

When the application starts, the password file will be loaded into the left ListView, and as soon as we click on the *Start* button, all the process begins:

- Define an executor (*ThreadPoolExecutor*) to handle all the threads that will be created and launched. The pool size for this executor must be the same than the number of cores of your processor (or, at least, 2 threads if your processor has only one core). The methods
  - `Executors.newFixedThreadPool(...)` and
  - `Runtime.getRuntime().availableProcessors()`

  may be helpful.

- Create a thread for each key in the left list, and launch it in the executor. The code of the thread must solve the problem explained in the introduction: explore the numbers from 000000 to 999999 until it finds the desired SHA key. Then, add the number to the right list, in the same position than its encryption.
    - You must use *Platform.runLater* to access the right *ListView* from the thread and update the corresponding index with the number found.
- Besides, create a *Service* that periodically (every 100 ms, for instance), updates the label in the center of the application with the total number of threads that have not completed their task yet. There are some methods in the executor object that may be helpful for this, such as *getCompletedTaskCount*.
- If you click on any key from the left list, then its corresponding number will be automatically selected from the right list.



- Remember to shut down the executor after launching all the threads (call to `shutdown()` method)

4. Evaluation rules

This final exercise will be evaluated as follows:

- PasswordHackConsole application: **5 points**
  - Loading passwords from the text file: **0,5 points**
  - Defining a *CompletableFuture* for each password: **1 point**
  - Storing the results from every *CompletableFuture* into a *ConcurrentMap* object: **1 point**
  - Running another task to print the results once all the *CompletableFuture* objects have finished: **1 point**
  - Defining a *ScheduledExecutorService* to print the progress of the application until all the *CompletableFuture* tasks finish: **1 point**
  - Code cleanliness and documentation/comments: **0,5 points**
- PasswordHackFX application: **5 points**
  - JavaFX layout and loading passwords from the text file into the left list view: **0,5 points**
  - Defining a thread for each password: **1,5 points**
  - Defining an executor of the appropriate size to manage all the threads: **1 point**
  - Defining a service that updates the label with the number of threads pending: **1 points**
  - Retrieving the corresponding number when clicking on a key from the left list: **0,5 points**
  - Code cleanliness and documentation/comments: **0,5 points**

4.1. Avoid trivial failures

You must take into account this additional rule for your final mark:

- Every trivial error will be strongly punished, so that the final exercise will be considered wrong. For instance, if the application can't load the file at the beginning, or the list is not properly populated to test the threads on it. In other words, if we need to apply any change to the application to let us test the important concepts (thread management), you will fail this exercise.