Concurrent programming exam Lab 2021-12-22

Due No due date **Points** 35 **Questions** 1
Available Dec 22, 2021 at 5pm - Dec 22, 2021 at 8pm about 3 hours **Time Limit** None

Instructions

Concurrent Programming exam

Remember that the conditions for the practical exams (see the file about them in the theoretical Canvas) apply.

- Specifically, don't forget to keep sending in your Code Together links using the assignment in Canvas.
- Also remember what you need to do at the end of the exam (the details of solution submission).

You may download the files to start with **by clicking here** (https://exam.inf.elte.hu/exam20211222-kanji.zip).

- Keep the tester code unmodified except for the following.
  - You may temporarily comment out the yet unimplemented parts.
- You have to solve the exercises in order.
  - It is required for the test code to compile, run, and produce proper output before you may continue with the next exercise.

Random kanji-number generator

Summary

We want to create a flashcard pack to practice number translation into Japanese. The Japanese number system is different from the western number system in some respects. That is, not only Arabic numerals, but kanji symbols are also used to represent numbers, furthermore, numbers are grouped by 10,000. For example, we can express 130,500 as 13 (十三) × 10,000 (万) + 5 (五) × 100 (百) = 十三万五百.

Our application shall create files with comma separated values (csv), in which random generated numbers and their kanji representations (separated by comma) are enumerated (in random order). For example:

```
68702, 六万八千七百二
43959, 四万三千九百五十九
21149, 二万千百四十九
71456, 七万千四百五十六
65838, 六万五千八百三十八
1700, 千七百
657, 六百五十七
```

Tester

The `Tester.java` file implements some simple verifications that must pass (run without throwing and exception) for each solution and should also terminate. Parts of `Tester.java` can be commented out if there is no solution for that sub-task, and debug prints can also be moved around temporarily; however, other modifications are not allowed.

- For further testing, you should write test code into the `TaskN.java` files. You should only test your solution for numbers in the [0; 99,999,999] interval, as `KanjiLib.convert` cannot handle values outside that range.
- Each solution shall be in the `TaskN.generate` method, which generates `count` pieces of random numbers that are in the `from..to` interval.
- This method returns a list of strings, that has format `<number>, <kanji number>`.
  - The length of this list must be `count`, except for task 3, which implements an interrupt mechanism.
- Parameters to this method may not be checked, as they are assumed to be correct.
- **Method signatures must not be modified.**
- Your solution must **avoid race-conditions**.
- You should not inherit from `java.lang.Thread` in your solution.

Task 1 : `Task1.java` (10 points max.)

The solution for this task shall be in the method called `Task1.generate`. You shall use a shared list for collecting the generated `Strings` (already declared in the file). Your task is to start 10 threads, each of which has the following behaviour:

- In a loop:
  - Generate a random number in the [`from`, `to`] closed interval. `Math.random` or other **thread-safe** random number generated can be used.
  - Create the Japanese kanji version by calling `KanjiLib.convert`.
  - Create the string `<number>, <kanji number>` (e.g.: `123, 百二十三`), which will be added to the **shared list**.
  - Without interference (as an atomic operation):
    - Check whether the list has enough items (`count`), and if so, immediately stop.
    - Check whether the generated `String` is already in the list, and if not, place it into the end of the list.

Once all 10 threads finished their execution the method shall return the list of generated strings.

Task 2: `Task2.java` (10 points max.)

The solution for this task shall be in the method called `Task2.generate`. In this task, random number generation and kanji conversion will be separated into two distinct steps. One dedicated thread will generate exactly `count` pieces of unique random number and send these numbers to other threads that do only the conversion part. As a result, conversion threads need not check preconditions anymore, they can focus only on their designated task (kanji conversion).

The `generate` method shall work as follows:

- Start a thread (we shall call it **A**) that will:

- Generate `count` random numbers in a loop:
    - If the number is already present (in some **thread-confined** container), generate a new one.
    - If the number is unique, **send it** to the other threads (see below).
    - Ignore interruptions.
- When the loop ends, make sure that all the other threads will eventually stop (see below).

- Start another 10 thread (referred to as **B** threads), each of which will start a loop and:
    - Receive a random number from thread **A**.
    - Convert this number into the usual `String` and add this string to a **shared** data structure (shared among **B** threads). Since **A** already ensured uniqueness and will only send up to `count` numbers, there is **no need for further verifications**.
    - Ignore interruptions.
    - However, when **A** signals the end of work, break out of the loop, and stop (there is no other exit criterium).
- For the communication of threads (**A** and **B**), you shall apply one of the **producer-consumer patterns** that you learned and use some **bounded buffer** solution.
- The communication channel has capacity of 100 (use constant `CHANNEL_CAPACITY`).
- When all threads have stopped, return the generated list of `String`.
- End of communication (or end of work) is signaled via a designated special value (*poison pill*) that thread **A** sends to each thread **B** (one to each of them, a total of 10). You can use the `POISON_PILL` constant.

Task 3: `Task3.java` (15 points max. in total)

Your solution is expected to be in class `Task3`.

This time, instead of using a single static method, you shall create a class that is capable of not only random number generation but also gracefully stopping in the middle of work. The skeleton of this class is available in `Task3.java` and your task is to complete it. Additional private helper methods can be defined, if necessary (although it is not needed), however, removing methods or changing signatures is **not allowed**. For this task, you will need to declare private data member references to shared data structures (instead of local references).

Part (a) (7 points max.)

The constructor of class `Task3` works similarly to `generate` in **Task 2**. That is, it starts 10 + 1 threads which cooperate in generating numbers (strings) via a producer-consumer pattern. Therefore, it is advised to start by copy-pasting your previous solution.

Then, we have the following changes:

- The constructor only starts the threads, it does not wait for them to finish.

- Parts of the code shall be placed into private methods. For this, fill the skeleton methods provided in `Task3.java`.
- **Waiting** for threads to stop and returning the result list shall be implemented in method `Task3.get`. In this method, `InterruptedException` should be propagated instead of caught.
- Instead of synchronizing explicitly when accessing the shared result **list**, you shall use a **thread-safe data structure**.

Part (b) (3 points max.)

Implement the `Task3.interrupt` method which can stop the **whole** process of number generation before it finishes.

- For this, you have to modify the previous solution so when threads are interrupted, they eventually **stop**.
- For this sub-task, it is enough to make thread **A** interruptible and ensure that **A** signals end of work to all **B** threads as usual.

Part (c) (5 points max.)

The `Task3.getThreads` method shall return the list of **B** threads. When any of these threads is interrupted, the whole number generation process shall stop (as soon as possible). For this we need the following:

- Thread **A** shall not wait forever if the channel is at capacity. Instead, wait up to 100ms, and when time is up, the thread shall stop **immediately**. When this happens, sending `POISON_PILL` is pointless as probably thread **A** is the last running thread.
- The **B** threads shall be interruptible. If any of them is interrupted, it shall send `POISON_PILL` to others (in the **B** group) so that they finish. Note: it is okay to include itself when sending poising pills (see below why).
- Sending `POISON_PILL` shall have a **timeout** of 10ms.

This quiz is no longer available as the course has been concluded.

Attempt History

	Attempt	Time	Score
LATEST	Attempt 1	175 minutes	12 out of 35

Correct answers are hidden.

Score for this quiz: **12** out of 35
Submitted Dec 22, 2021 at 7:56pm
This attempt took 175 minutes.

Question 1

Not yet graded / 35 pts

Upload your solution here.

[lab exam.zip (https://canvas.elte.hu/files/1500371/download)](https://canvas.elte.hu/files/1500371/download)

Quiz Score: **12** out of 35

This quiz score has been manually adjusted by +12.0 points.