Assessed Exercise II
CS5002 Advanced Programming

Dr. Paul Henderson

February 2025

- **Deadline:** 16:30 on 17th March 2025
- **Contribution to final course mark:** 25%
- **Solo or Group:** Solo work
- **Anticipated duration:** 4 hours

Introduction

From the course Moodle page, download the files `CommandRunner.java` and `SlowCalculator.java`. The latter contains a class `SlowCalculator` implementing `Runnable`. Its constructor takes a number N as input and stores it. Method `run` performs a long, slow calculation on this number, then prints the result. You do *not* need to consider (or even understand) the mathematical details of what is being calculated!

In this exercise, you will write a class implementing the interface `CommandRunner`, that runs certain commands passed as strings. These are inputs from a user, requesting the calculation be performed for particular numbers. The inputs and outputs are read/written by a separate part of the program – you just have to use the passed string, and return a string.

As the calculation is slow, you will run it on background threads, so the user can continue to interact with the system while tasks are running. To take advantage of modern multi-core CPUs, the user may request several calculations be run in parallel. Each calculation will still run on one single thread – your goal is to let the user run multiple calculations in parallel, **not** to make the individual calculations faster.

Task

You will write a class called `Solution` that implements the `CommandRunner` interface. The `runCommand` method of this class will be passed a string, corresponding to a command the user entered. It should perform the relevant command (see below), and return the specified output string. You should **not** modify the interface `CommandRunner` in any way. Your `Solution` class should have a public constructor taking no parameters.

The following table lists the commands the user may enter (i.e. that may be passed as the `command` argument to your `Solution.runCommand` method), and the required behaviour of your `runCommand` method (note *N* and *M* are long integers chosen by the user):

`start` *N*	start calculating with input N, by calling `SlowCalculator.run` on a new thread; immediately return the message "`started` N"
`cancel` *N*	immediately cancel the calculation with input N that is currently running or pending with `after` (do nothing if it already completed or if it was never requested); when it has stopped (which should be within 0.1s) return message "`cancelled` N"
`running`	return a message indicating the total number of calculations currently running (i.e. excluding those already completed/cancelled), and their inputs N (in any order), in the form "`3 calculations running: 83476 1000 176544`". If no calculations are running, return the string "`no calculations running`".
`get` *N*	if the calculation for N is finished, return message "`result is` *M*" where *M* is the integer result; if the calculation is not yet finished, return message "`calculating`". If the calculation was started but already cancelled, return message "`cancelled`". If the calculation is scheduled with `after` but not yet started, return message "`waiting`".
`after` *N* *M*	schedule the calculation for *M* to start when that for *N* finishes (or is cancelled). Return the message "*M* `will start after` *N*" immediately (without waiting for either calculation). The calculation for *M* should not appear in `running` until it is actually running (i.e. *N* has completed). If *N* has already finished, *M* should start immediately. If a circular dependency would arise (i.e. two calculations waiting for each other, hence neither would start), then `after` should not schedule *M*, but instead return the message "`circular dependency` *N* `...` *M*" where `...` is replaced by the numbers of all calculations scheduled after M (or after another that is itself after M, recursively), and which N is itself scheduled after (these can be listed in any order)

`finish`	wait for all calculations previously requested by the user (including those scheduled with `after`) to finish, and then **after** they are all completed, return message "`finished`"
`abort`	immediately stop all running calculations (and discard any scheduled using `after`), and then when they are stopped (which should be within 0.1s) return message "`aborted`"

For any command not given above (including malformed commands, e.g. N is not an integer), the method should return message "`Invalid command`"

Note that the provided code for `SlowCalculator` prints the answer at the end of `run()`. You must **remove** this `print` statement, and replace it with a suitable mechanism to return the result so your `Solution` class can retrieve it. You should **not** modify the mathematical calculations performed by either `SlowCalculator.calculateNumFactors` or `SlowCalculator.isPrime`, but you will need to add support for interruption. You may change other parts of this class if you want (e.g. adding new fields or constructor parameters). You must **not** change the names of the existing methods.

Here is an example of what should happen when your program runs; the highlighted lines are commands passed to `runCommand`, and the remaining lines are the messages it returns:

```
start 10456060
started 10456060
running
1 calculations running: 10456060
get 10456060
result is 3
start 72345680
started 72345680
start 534912560
started 534912560
get 534912560
calculating
running
2 calculations running: 72345680 534912560
cancel 72345680
Cancelled 72345680
running
1 calculations running: 534912560
finish
finished
```

Notes

- We will not test on pathological inputs for which the behaviour is not defined, i.e. not specified above. Please do not ask questions about any special cases you may think of (e.g. starting a calculation that is already running). It does not matter what your program does in such cases
- Do ensure you return "`Invalid command`" (as mentioned above) if you receive a command that is not of the required form (e.g. N is missing or not an integer, or a command is misspelt)
- Your program should **not** limit the number of calculations/threads that can be run in parallel
- Your program should **not** print any output directly, nor read from the console
- Do **not** use any deprecated methods, such as `Thread.stop()` (such methods are typically deprecated because they are dangerous, and so it is bad practice to use them)
- Remember the goal of this exercise is **not** to make each individual calculation parallel, but to run several independent calculations for different N in parallel!

Submission

- You must implement a class `Solution` in file `Solution.java`, inheriting from `CommandRunner`. Do **not** put your code in a Java package (i.e. do not use the `package` keyword in any files).
- You will submit a single `.zip` file (**not** a tar, or a rar, or anything else), named `1234567.zip` where 1234567 is replaced by your 7-digit numeric student ID
- This `.zip` file shall contain no folders, (e.g. `src/`), but only the `.java` files containing your code, including `SlowCalculator.java` and `Solution.java`, and maybe others if you wish
- Do **not** include `CommandRunner.java` in your submission
- Do **not** include a `main` method anywhere
- Upload your `.zip` file to Moodle before the deadline!

Mark Scheme
Your submission will be marked out of 65 points, broken down as follows:

- Code is provided in the correct form and compiles successfully [5]
- `start`, `running`, `get` and `cancel` commands work as expected [20]

- `finish` and `abort` commands work as expected [15]
- `after` command works as expected [25]

If you do not follow the exercise specification precisely, you will lose marks (your submission will be processed by a computer program; it will assume you've followed the instructions).