CS 511: Homework Assignment 2
Due: 9 October, 11:55pm

1 Assignment Objectives

Get acquainted with the notion of

- Semaphores

2 Assignment Policies

Collaboration Policy. This homework may be done in individually or in pairs. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be reused. Violations will be penalized appropriately.

Late Policy. Late submissions are allowed with a penalty of 2 points per hour past the deadline.

3 Assignment

The aim of this assignment is to model a gym that has clients. Each client has a routine. A routine consists of a sequence of exercises. Each exercise consists of a gym apparatus a, corresponding weight and the time it takes to perform it. Clients compete for the use of the gym apparatus and the weight plates. Weight plates come in different sizes. Also, due to security regulations, there may be at most `GYM_SIZE` clients in the gym at any given time.

3.1 The Gym

A gym has numerous clients and apparatus. Below is a simple UML diagram of the class `Gym`. IMPORTANT: This diagram (and all subsequent ones) should be understood as a guide: you may need to add additional variables and methods. In particular, some classes will require that you declare various semaphores.

Gym

private static final int GYM-SIZE = 30;
private static final int GYM-REGISTERED-CLIENTS = 10000;
private Map<WeightPlateSize,int> noOfWeightPlates;
private Set<Integer> clients; // for generating fresh client ids
private ExecutorService executor;
// various semaphores - declaration omitted

public void run()

A gym has a capacity and a number of registered clients. It has a number of apparatus (these shall be semaphores that you will have to add to the data fields of the class) which make use of weight plates (the map `noOfWeightPlates`). There are numerous weight plates of each size; the possible sizes are:

```
1 public enum WeightPlateSize {
2     SMALL_3KG, MEDIUM_5KG, LARGE_10KG}
```

The total set of weight plates in the gym are: 75 of size 10kg, 90 of size 5kg and 110 of size 3kg; as mentioned, this information is recorded in the `noOfWeightPlates` variable of the class `Gym`. Finally, a gym has a list of clients. Clients are described below. The executor service is described towards the end of this document.

3.2 The Clients

A client has an id and a routine. Both of these will be created randomly, on the fly, as the simulation executes.

Client
private int id; private List<Exercise> routine;

public Client(int id);
public void addExercise(Exercise e);
public static Client generateRandom(int id, Map<WeightPlateSize,int> noOfWeightPlates);

A routine is simply a list of exercises.

3.3 The Exercises

Each gym *client* has a *routine* consisting of a sequence of *exercises*. An exercise consists of an apparatus type, a corresponding weight and a duration in milliseconds.

Exercise
private ApparatusType at; private Map<WeightPlateSize,int> weight; private int duration;
public Exercise(ApparatusType at, Map<WeightPlateSize,int> weight, int duration) public static Exercise generateRandom(Map<WeightPlateSize, Integer>);

There are various apparatus types, and for each type, various apparatus. More precisely, there are 5 of each of the following types of apparatus in the gym (for example there are 5 Leg Press machines, 5 Bar Bells, and so on):

```
public enum ApparatusType {
    LEGPRESSMACHINE, BARBELL, HACKSQUATMACHINE, LEGEXTENSIONMACHINE,
    LEGCURLMACHINE, LATPULLDOWNMACHINE, PECDECKMACHINE,
    CABLECROSSOVERMACHINE
}
```

An example of a routine might be:

```
[
 (LEGPRESSMACHINE,[(SMALL_3KG,0), (MEDIUM_5KG,2), (LARGE_10KG,0)]),
 (BARBELL,[(SMALL_3KG,2), (MEDIUM_5KG,2), (LARGE_10KG,2)])
]
```

The exercises must be performed in the order in which they appear in the routine. Note that a routine may include repeating the use of an apparatus, in that case it would simply appear repeated in the list.

In order to perform an exercise:

1. the required apparatus must be available and
2. the required weight plates must be available (weight plate specification must be respected as stated in the routine, for example, if a large weight plate is required it cannot be substituted by 2 medium sized ones).

Once a client has finished using an apparatus, she must unload all weight plates and release them for others to use (this includes consecutive uses of the same apparatus).

3.4 Solution

Model this scenario using semaphores. These semaphores must allow for a correct use of shared resources. The shared resources in this problem are:

- The apparatus: a client must wait for an apparatus to be free before using it.

- The weight plates: a client must wait for all the required weight plates to be available. You may use `acquire()` but **not** `acquire(int)` for acquiring weights.

The gym itself is also a shared resource. However, rather than using semaphores for upholding this requirement you are to use the maximum of `GYM_SIZE` allowed clients as the size of your thread pool (see Sec. 3.5).

Regarding the simulation of the gym, the gym should generate clients randomly and have them execute their corresponding exercise routines. Clients should be assigned unique ids and should have between 15 and 20 exercises in their routines. The duration of each exercise should be anything reasonable (in the sense that the simulation doesn't slow down too much). Also, the number of plates for each exercise should be between 0 and 10, making sure that at least one weight plate is required. The simulation should print out events in which a client starts exercising (including the apparatus number and the weight plates it is using) and then when it finishes.

3.4.1 List of Classes

Here is a list of the classes you have to define, all included in a package called `Assignment2`:

- `ApparatusType`: Contains the declaration given above and possibly methods for generating a random apparatus type.
- `WeightPlateSize`: Contains the declaration given above and possibly methods for generating a random weight plate size.
- `Client`: A client with her routine and other information indicated above. Also should include a method `generateRandom(int id)` that generates a random client with the given `id`.
- `Exercise`: An exercise as specified above. Also should include a method `generateRandom(Map<WeightPlateSize, Integer>)` that generates a random exercise using the maximum number of plates given in the argument.
- `Gym`: The simulation of the gym consists in randomly having clients come in to the gym. You may assume that there are a total of `GYM_REGISTERED_CLIENTS` registered clients. The `run` method of `Gym` should be in charge of the simulation itself. The threads corresponding to the clients should be spawned using a *pool thread* (See Sec. 3.5).
- `Assignment2`: Here is the code:

```
1 package Assignment2;
2
3 /** start the simulation */
4 public class Assignment2 {
5         public static void main(String[] args) {
6                 Thread thread = new Thread(new Gym());
7                 thread.start();
```

```
                try {
                        thread.join();
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
```

3.5 A Note on Pool Threads and the Executor Interface

The `Executor` interface[1] or its subinterface `ExecutorService` (which is the one we will be using in this assignment) represents an asynchronous execution mechanism which is capable of executing multiple tasks in the background. Such multiple tasks are referred to as *thread pools*. An `Executor` is normally used instead of explicitly creating threads for managing each of the tasks, i.e., rather than invoking `new Thread(new(RunnableTask())).start()` for each of a set of tasks, you might use:

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
 ...
```

Thread pools are useful when we need to limit the number of threads running in an application at the same time as there is a performance overhead associated with starting a new thread (each thread is also allocated some memory for its stack, etcetera). Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and is then executed.

There are many ways of managing thread pools. One easy way is to use the static method `newFixedThreadPool` of the class `Executors` (not to be confused with the `Executor` interface). This method, whose signature is

```
public static ExecutorService newFixedThreadPool(int nThreads)
```

creates a thread pool that reuses a fixed number of threads operating off a shared (unbounded) queue. At any point, at most `n` threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. The threads in the pool will exist until it is *explicitly* shutdown.

Here is a simple example of `ExecutorService`:

```
ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("I am an asynchronous task!");
    }
});

executorService.shutdown();
```

[1] Found in `java.util.concurrent`

Here is what is happening:

- An `ExecutorService` is created using the `newFixedThreadPool(int)` factory method. This creates a thread pool with 10 threads executing tasks.
- An anonymous class implementation of the `Runnable` interface is passed to the `execute()` method. This causes the `Runnable` to be executed by one of the threads in the `ExecutorService`.
- The `ExecutorService` is shut down, so the all threads finish running. To terminate the threads inside the ExecutorService you call its `shutdown()` method. The `ExecutorService` will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, it shuts down. All tasks submitted to the `ExecutorService` before `shutdown()` is called, are executed.

  If you want to shut down the ExecutorService immediately, you can call the `shutdownNow()` method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

4 Submission Instructions

Submit a zip file named `Assignment2_<Surname>.zip` (where `<Surname>` should be replaced by your surname) through Canvas containing the java source files.

Important: Please make sure that your assignment compiles and runs on the Linux lab before submitting.