A Concurrent Visit to the Museum

COMP40009 – Computing Practical 1

6th – 10th February 2023

Aims

- To practice writing concurrent programs in Java
- To gain experience with race conditions
- To practice working with *synchronization* primitives and *locks*
- To gain experience with *abstract* classes and *inheritance*

Problem Overview

- Your first task is to implement an abstract class *MuseumSite*, representing a generic site in a Museum. You will then need to implement the *Entrance*, *Exit*, and *ExhibitionRoom* classes to represent three different types of sites. For simplicity, we assume that each museum has only one entrance and one exit, while it can have multiple exhibition rooms. For each room, there is at least one path that allows a visitor to reach it from the entrance, and there is at least one path that allows a visitor to reach the exit from that room.
- For each class implementing *MuseumSite*, override the methods enter(), exit(), and hasAvailability() to reflect the specific behaviour of Entrance, Exit, and ExhibitionRoom.
- You will then need to implement the class *Turnstile*. Turnstiles have exactly one originSite and one destinationSite (i.e. they only go in one direction). When calling the method *passToNextRoom()*, check if destinationSite has availability. If so, call exit on originSite, call enter on destinationSite, and return the destination site; otherwise, return an empty Optional.
- The third task is to implement two simple Museum topologies (as depicted in Figure 1 on page 6 below), which will be part of the *MuseumSite* class.
- The fourth task is to add logic to the *Visitor* class which implements the *Runnable* interface, allowing visitors to move concurrently in the Museum. Visitors pick a *random* turnstile for their next move, and they wait for a *random* amount of time when the turnstile is not available.
- **The final task** is to edit the main method of the class Museum so that *numberOfVisitors* threads corresponding to running Visitors are created and started. Afterwards, the main thread has to wait for all these Visitor threads to complete their execution.

- For this exercise, you have been provided with a JUnit testsuite that you should run incrementally to help check the correctness of your solution. You can also write additional tests as you proceed through the various tasks below. **In addition to running the testsuite, run the main method of the Museum class and look for the output, as it may point out to additional concurrency issues**.

What to Do

Clone your exercise repository (remembering to replace *login* with your login) with:

```
> git clone
  https://gitlab.doc.ic.ac.uk/lab2223_spring/javathreads_login.git
```

You will notice that the skeleton files have been zipped and encrypted with a password. In order to extract them, you will need to use an application that supports 7z files[1]. On the lab machines, this has already been installed for you, so you can extract the archive by typing:

```
7z x skel.7z -p0093E56B
```

(*n.b., 0093E56B is the password*). Make sure that you extract the contents of the archive directly inside your repo, i.e. do not create an extra subfolder named "skel", because that would break the LabTS autotesting process. Once this is done, you can then safely delete the 7z archive. Remember to commit and push all the extracted files back into the repo.
If you list all the files in the newly checked out repository, you should see the following:

```
% ls -a java_threads
.  ..  .git  .gitignore  src  test
```

As in previous labs, these are:

- `.`, `..` – these refer to the current directory and its parent directory.
- `src` – the directory in which the Java source files for the exercise reside. The `src` directory contains one directory, `museumvisit` corresponding to a Java package.
- `test` – this directory contains a Java source file corresponding to a JUnit 4 test suite for the exercise. This is discussed further under **Testing**, below.
- `.git`, `.gitignore` – these contain the git repository information, and a list of filename patterns that should be ignored by git, respectively.

A note on Race Conditions and Deadlocks

The exercise will involve writing code to avoid race conditions. What is a *race condition*? When multiple threads try to read and write a shared variable concurrently, and these read and write operations overlap in execution, then the final outcome depends on the order in which the reads and writes take place, which is unpredictable. To deal with race conditions, mutual exclusive access has to be enforced. This allows only one thread at a time to access the critical section where the shared resources are accessed.

[1] `https://www.7-zip.org/download.html`

If two or more threads need to acquire exclusive access to two or more shared resources, they may incur in a circular dependency. For example, thread `t1` and `t2` need to acquire resources `r1` and `r2`. If they order in which the threads acquire exclusive access to the resources is not decided carefully, it may happen, for example, that `t1` acquires `r1`, `t2` acquires `r2` and then they are in a deadlock because `t1` needs `r2` (locked by `t2`) while `t2` needs `r1` (locked by `t1`).

To establish an order among different shared resources, you need to make them comparable. This usually require finding an attribute that uniquely correspond to one instance of the resource (e.g., name, id, an hashcode) and compare different resources on the base of that attribute (e.g., `r1.id` ¿ `r2.id`).

A final note: if a thread calls a method defined in another class (e.g., a Visitor calls a method of the Turnstile), it is exactly as if the thread is executing the instructions in that method, including acquiring or releasing locks.

The `MuseumSite` class

A museum is composed of different sites: the entrance, the exit, and a number of exhibition rooms. An exhibition room has limited capacity, i.e., at most a certain number of visitors can be in the room at any time. However, the entrance and exit sites let *any* number of visitors in and out, who are either waiting to enter the museum, or after completing their visit.

When the method enter() is called on a site, the occupancy state of the site is increased by 1, to account for the new visitor that entered. Similarly, when the method exit() is called, the occupancy is decreased by 1. *Entrance and exit sites have unlimited capacity*, so it is always possible to enter them.

In the `museumvisit` package you will find the abstract `MuseumSite` class representing a generic site of a museum. Each room has a current number of visitors (occupancy) `int`, and a List of Turnstiles, representing the ways in or out of the room.

Public constructors

- A constructor that accepts a `string` argument, representing the room *name*. It constructs an empty room with occupancy zero and an empty List of turnstiles.

As it is an abstract class, constructors cannot be instantiated directly.

Public methods

- `boolean hasAvailability()`, which returns true if a room has availability to host a visitor; false otherwise.
- `void enter()`, is called when a visitor enters the room (implementation depends on room characteristics).
- `void exit()`, is called when a visitor leaves the room. Occupancy of the room should never be negative (invariant).
- `List<Turnstile> getExitTurnstiles`, `String getName()` and `int getOccupancy()`, return the List of turnstiles, name and the occupancy of the room respectively.

The `ExhibitionRoom` class

Public constructors

- A constructor that accepts a `string` argument, representing the room *name* and an `int` capacity value. It constructs an empty room with the specified capacity which should always be greater than zero.

Public methods

- `int getCapacity()`, returns the capacity of the room.

The `Entrance` and `Exit` classes

Public constructors

- As museums have only one entrance and one exit, the skeleton contains default constructors that assign the names "Entrance" and "Exit" to the instances of the `Entrance` and `Exit` classes. This should be enough, but feel free to add additional constructors as you see fit.

Public methods

- `List<Turnstile> getExitTurnstiles` for the `Exit` class should return an empty list, as there are no further sites to visit beyond the exit.
- `boolean hasAvailability()` returns always true for entrance and exit as these sites do not have limited capacity.

The `Turnstile` class

To make sure no room is overcrowded, each room keeps track of its current occupancy, and the management installed turnstiles at all access points from a site to another, in order to control the flow of visitors. A turnstile connects exactly one origin room to exactly one destination room *unidirectionally* (and the two rooms, of course, have to be different). Each site in the museum has a set of turnstiles through which the visitors can move to the corresponding destination rooms. The only exception is the exit site: when the visitors reach the exit, the visit is complete and they cannot go anywhere; thus, *the exit site has no turnstiles* letting the visitors move to other sites.

When a visitor approaches a turnstile to access the corresponding destination site, the turnstile will: 1) check if there is availability at the destination site 2) if the destination site is not full, the visitor can leave the origin room and enter the destination room. In this case, the turnstile will:

- inform the origin room that one visitor is leaving by calling the method exit() on the origin room.
- inform the destination room that one visitor is entering by calling the method enter() on the destination room.

Public constructors

- A constructor that accepts two `MuseumSite` arguments, the first representing the origin room and the second one representing the destination room. The two rooms have to be different, and the turnstile is also responsible for updating the origin site by adding itself to the turnstile list.

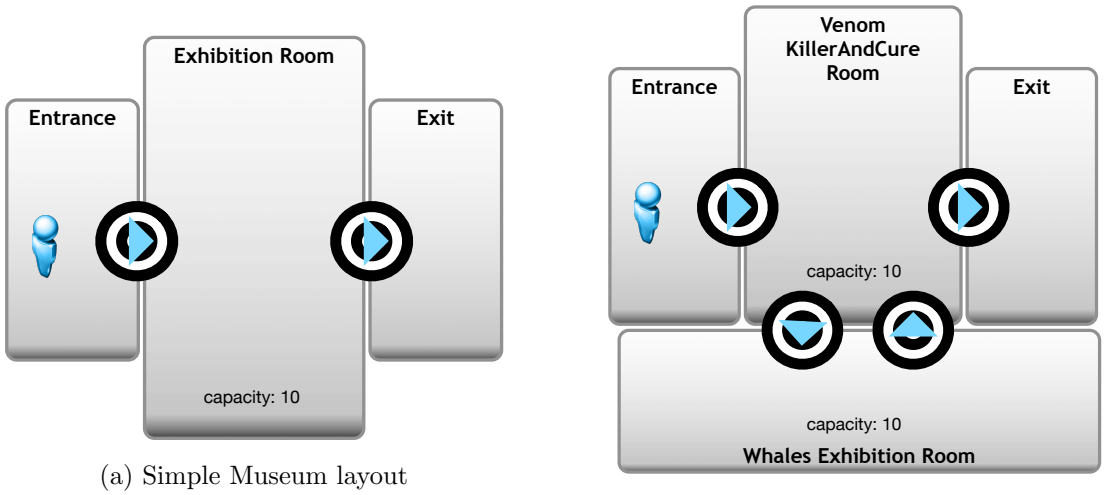
Public methods

- `Optional<MuseumSite> passToNextRoom()`, enters the destination site when there is availability, as described in detail above.
- `MuseumSite getOriginRoom()`, and `MuseumSite getDestinationRoom()`, returns the origin and destination rooms respectively.

The `Museum` class

You are now ready to introduce *Museum Topologies*. Figure 1a, depicts a very simple museum with a single exhibition. Figure 1b depicts a slightly more complex topology, where the central *Venom* room is leading both to another exhibition about *Whales*, as well as an exit.

- Complete the `Museum buildSimpleMuseum()` and `Museum buildLoopyMuseum()` methods accordingly.



(a) Simple Museum layout

(b) Loopy Museum layout

Figure 1: A figure with two subfigures

The `Visitor` class

Visitors can move independently within the museum. The class `Visitor` implements `Runnable`, so that a thread can be created for each visitor, allowing them to move independently from one another. Visitors know in which room they are at any time. When they enter a room, they spend some time visiting the exhibition in the room, then look for the available turnstiles to randomly pick the next room to visit. At the beginning, all the visitors are at the entrance site **and have to enter it**. When a visitor reaches the exit site, there are no more turnstiles to follow and thus the visit terminates.

Public constructors

- A constructor for this class has been provided in the skeleton. However, you are welcome to create additional constructors if you wish (or even change the implementation of the existing one), as long as you maintain the signature of the provided constructor intact. (*e.g. if you want to enforce that the visitor enters the Entrance site directly in the constructor*)
- Please note that you have also been provided with some code for randomising the movement of Visitors. You are free to change this code at will, as long as you maintain the concept of visitor randomisation (i.e. do not define hard-coded static paths).

Submission

As usual, use `git add`, `git commit` and `git push` to send your created **.java** files to the gitlab server.

Then, log into the gitlab server `https://gitlab.doc.ic.ac.uk`, and click through to your `javathreads_<login>` repository. If you view the `Commits` tab, you will see a list of the different versions of your work that you have pushed. Select the right version of your code and ensure it is working before submitting to CATe via LabTS.

Assessment

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```