CO661 Class 2 - Java concurrency
Locking individual resources and channels

This class continues our learning about shared-memory concurrency primitives (in Java) and structuring concurrent programs, as well as message-passing concurrency.

Task 1 - Locks associated to resources

Imagine a car rental agency where customers come in looking to rent a specific car. Only one driver can rent a particular car at once, i.e., we need to ensure that individual resources are uniquely allocated. The customers are willing to wait for the car if it is not available. This task will guide you to implement this program using data structures to associate locks to each resource (car).

Download `Class2.java` which provides partially-implemented classes for this situation. There are four classes, two of which you need to complete:

- `Renter` (`implements Runnable`) models a renter who has a particular car in mind, modelled by an integer (`interestedIn`); `run` tries to rent the car, then drives for a random distance between 1 and 100 km before returning the car;
- `Car` - Models a car and provides a `drive` method;
- `Lender` - **TODO: complete** – This class controls access to the cars, stored internally using a list of cars, each of which has an associated lock in a separate list. Use these locks to provide mutual exclusion access to each car in the implementations of `rentCar` and `returnCar`.
- `Class2` - **TODO: complete** – Provides the static `main` method which you need to complete to spawn 10 renters who each are interested in one of three possible cars (randomly allocated).

Consider whether you should make `rentCar` and `returnCar` into `synchronized` methods. If so why? If not, why?

Task 2 - Channel interaction

Rewrite your solution to the previous task using channels for the renters to communicate with the lender. Here is a rough guide/set of hints:

1. Download `Channel.java` and `Request.java` from Lecture 4 to provide synchronous bidirectional channels and requests which include the response channel endpoint;
2. Create a channel for requests where one end is given to the lender (when it's constructed) and the other end is passed to all the renters;
3. Create subclasses of `Request` for rental requests (which when sent will replace calling `rentCar` in the renter) and returning cars (which when sent will replace calling `returnCar` in the renter). Think about the data that needs to be sent with the request, and the data that needs to be returned with the response.
4. Change lender into a runnable thread with `run` that contains a message receiving loop that deals with requests and calls `rentCar` and `returnCar` as appropriate internally. To provide concurrent access, requests will need to be process in separate "sub server" threads. The easiest way to spawn a separate thread to deal with requests is to use an anonymous `Runnable` class, e.g.:

```
Thread subServer = new Thread(
   new Runnable() {
     @Override
     public void run() {
       // Your code here for handling received requests
     }
  });
subServer.start();
```

Recall the locks must be locked and unlocked by the same owner thread; if you use the above tactic you need to replaced locks with binary semaphores instead to allow `acquire` and `release` to be performed by separate sub-server threads.

5. Tie it all together.

Task 3 - Extension: Semaphore-based extension

Imagine that we want multiple cars of each kind to be available. Replace binary semaphores (or locks) in your previous solution with n-semaphores whose size corresponds to some number of available cars for each type.

Task 4 - Extension: Timeouts

(I don't expect you to do this task, but have a go if you have a chance and want to go further).

Adapt your solution to the previous task so that if cars of a particular kind are not available within a certain time limit (say 1s in this program) then the lender provides a different car to the renter. To do this you can use `tryAcquire`[1] e.g. `tryAcquire(1, TimeUnit.SECONDS)` instead of `acquire()`. What happens if no car is available?

[1] `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html`