Homework 4: Coffee Shop Simulation

CSYE 7215: Fall 2016

Overview:

In this project, you will write a simulation for a coffee shop. The simulation has five parameters:

- the number of customers (patrons) that wish to enter the coffee shop;
- the number of tables in the coffee shop (note: there can only be as many customers inside the coffee shop as there are tables at any one time - other customers must wait for a free table before going into the coffee shop and placing their order);
- the number of cooks in the kitchen that fill orders;
- the capacity of machines in the kitchen used for producing food; and
- a flag as to whether everyone has the same order or not.

Customers in the coffee shop place their orders (a list of food items) when they enter. Available cooks then handle these orders. Each cook handles one order at a time. A cook handles an order by using machines to cook the food items. There will be one machine for each kind of food item. Each machine produces food items in parallel (for different orders, or even the same order) up to their stated capacity.

Our particular coffee shop will only have three food items, each made by a single machine: a burger, made by machine Grill, which takes 500 ms to make; fries, made by machine Fryer, which take 350ms to make; and a coffee, made by machine CoffeeMaker2000, which takes 100 ms to make.

General Requirements

You must use the following classes and the methods specified for them (most of which is what you'll be writing). All of these are available in the skeleton code. You may implement any other methods you need for these classes. You can also implement additional private classes or new helper public classes as needed (just make sure they are in the same folder/package as the rest of the project files).

- **Simulation.java** - main() method is the simulation entry point for manual testing. Automated testing will call the runSimulation method directly, so make sure your program works when that is done (the current version of the main method shows you an example of how we'll call it).
- Food.java* - food items - don't change this!
- FoodType.java* - contains the three food types we have defined above.
- **Customer.java** - runs in a thread to implement a coffee shop patron.
- **Cook.java** - runs in a thread; makes orders provided by customers. Ensure that the order made by the cook is actually the one delivered to the customer.
- **Machine.java** - cooks particular food items, in parallel up to its capacity.

- SimulationEvent.java* - instances represent "interesting" events that occur during the simulation - don't change this, just use the events in the way described later in this description.
- **Validate.java** - defines method to validate the results of the simulation - this is something you can work on to help you test your own code. We recommend you define the validator to test your code.

** The starred classes are provided for you, and **must not be changed**; the remaining classes can be changed within the bounds given below.*

Restrictions. The focus of this project is to learn how to use waiting and notification in Java. For this reason, *you are not allowed to use any concurrent collections in the Java libraries*. Specifically, do not use anything contained in java.util.concurrent or the synchronized collections in the Collections class in java.util.collections. You may use the regular (i.e. non-synchronized) version of collections if you wish, however.

Particulars:

- **Simulation**
  - **main(String args[])** : This class is the entry point of the simulation, which is initiated by the main method.
    While the simulation runs, it will generate events, which are instances of the class **SimulationEvent**. Each event should be printed immediately, via its **toString()** method, and logged for later validation by the **Validate.validateSimulation** method. We've given you an events list to use to hold these events. When the simulation starts, it will generate a **SimulationEvent** via a call to **SimulationEvent.startSimulation()**. For this project, there will be two scenarios for orders; (1) if the randomOrders is set to false, then each Customer places the same order: two burgers, one fries, one coffee but (2) if the randomOrders is set to true, then each customer will place an order with a random number of burgers, fries, and coffee where each random number will be between 0 and 3. These items will be in a list sent into the Customer constructor. Once all **Customer**s have completed, the simulation terminates, shutting down the machines, and calling **interrupt** on each of the cooks telling them they can go home (so each of the Runnable Cook objects will be running in a thread and you can call interrupt() on that thread - you should set up the try/catch block in the Cook's run method so that it will catch the **InterruptedException** and invoke **Simulation.logEvent(SimulationEvent.cookEnding(this))** as a result). The last thing the simulation itself will do is generate the event **SimulationEvent.endSimulation()**.
- **Food**
  This class is provided for you. It represents a food item. You will create only three food items for your simulation (hamburger, fries, and coffee, as

described above) but your classes should treat food items generically; i.e., you should be able to easily change just the **Simulation** class if you want the simulation to have **Customer**s order different food items or different amounts, without changing any other class.

- **Customer implements Runnable** (needs to run as its own thread)
  - constructor **Customer(String name, List<Food> order, priority)** : takes the name of the customer and the food list it wishes to order; the format of the name is described below. You may extend this constructor with other parameters if you would find it useful, e.g., add the amount of time the customer requires to eat the ordered food. Each customer's order must be given a unique order number, which is used in generating relevant simulation events; it is easiest to generate this number in the constructor. All customer names should be of the form "Customer "+num where num is between 0 and the number of customers minus one.

  - **run()** : attempts to enter the coffee shop, places an order, waits for their order, eats the order (takes some time), leaves. Customers will generate the following events:
    Before entering the coffee shop: **SimulationEvent.customerStarting()**
    After entering the coffee shop:
    SimulationEvent.customerEnteredCoffeeShop()
    After placing order (but before it has been filled):
    SimulationEvent.customerPlacedOrder()
    After receiving order:
    SimulationEvent.customerReceivedOrder()
    Just before about to leave the coffee shop:
    SimulationEvent.customerLeavingCoffeeShop()

- **Cook implements Runnable** (needs to run as its own thread)
  - constructor **Cook(String name)** : the name is the name of the cook (described below). You may extend this constructor with other parameters if you wish. All cook names should be of the form "Cook "+num where num is between 0 and the number of cooks minus one.
  - **run()** : waits for orders from the coffee shop, processes the order by submitting each food item to an appropriate machine (remember about priorities). Once all machines have produced the desired food, the order is complete, and the Customer is notified. The cook can then go to process the next order. If during its execution the cook is interrupted (i.e., some other thread calls the **interrupt()** method on it, which could raise InterruptedException if the cook is blocking), then it terminates. Note that if the cook needs more than one item from a machine for an order, the cook can place all of those requests to the machine one after the other without waiting for the previous request to be filled. This means that a machine could (for

example) be making two orders of fries at the same time for the same order.
Cooks will generate the following events:
At startup: **SimulationEvent.cookStarting()**
Upon starting an order: **SimulationEvent.cookReceivedOrder()**
Upon submitted request to food machine:
SimulationEvent.cookStartedFood()
Upon receiving a completed food item:
SimulationEvent.cookFinishedFood()
Upon completing an order: **SimulationEvent.cookCompletedOrder()**
Just before terminating: **SimulationEvent.cookEnding()**

- **Machine**
  - constructor **Machine(String name, Food food, int capacity)** : the name of the machine, the Food it makes, and the capacity (how many Food items may be in process at once). The names of the machines are specified in the description of the Simulation class, above.
  - **makeFood()** : This method is called by a Cook in order to make the Machine's food item. You can extend this method however you like, e.g., you can have it take extra parameters or return something other than void. It should block if the machine is currently at full capacity. If not, the method should return, so the Cook making the call can proceed. You will need to implement some means to notify the calling Cook when the food item is finished. Machines will generate the following events:

    At startup: **SimulationEvent.machineStarting()**
    When beginning to make a food item:
    SimulationEvent.machineCookingFood()
    When done making a food item: **SimulationEvent.machineDoneFood()**
    When shut down, at the end of the simulation:
    SimulationEvent.machineEnding()

Hint: you will need some way for your Machine to cook food items in parallel, up to the capacity, but ensure that each item takes the required time. You might do this by having a Machine use threads internally to perform the "work" of cooking the Food. This approach will require some way of communicating a request by a Cook to make Food to an internal thread, and a way to communicate back to that Cook that the Food is done. In this simulation, the only thing that will actually be done during the "cooking" time is waiting the proper amount of time.

Note: You should model the time a food item is actually cooking using a statement of form Thread.sleep(n), where n is the cook time of the food. So, modeling the actual cooking of a pizza would be represented via the statement Thread.sleep(600). This means that simulations will run much “faster” than real time, since Thread.sleep(600) terminates in approximately 600 milliseconds rather than 600 seconds. ***This is a standard practice in simulation development!*** Simulation time

and real time are rarely the same. Sometimes simulations run much faster than the phenomena being modeled (as is the case here); other times they run much slower.

Getting Started
First, familiarize yourself with the above requirements in general. Next, download the skeleton code. After getting the source code, review the requirements while looking at the skeleton code. You will be writing helper methods, etc. so the skeleton files are just a starting point.

Testing
We have allowed a fair amount of freedom in the design of your classes for this project. Because of this freedom, and the nature of the submit server, constructing generic unit-level tests poses a problem. In particular, we've allowed you to modify the constructors of many objects (such as Machine and Customer), and unit tests would not work on modified objects since the pre-written tests wouldn't know things like what to pass into the constructor. Therefore, you might have to take on a fair bit more responsibility for your own tests as you work on this project. We've set up some preliminary public tests that will give you a sense of whether you are on the right track, but we will test your code with our own validators to check a variety of simulation characteristics. Our testing will include performing multiple test runs on the public scenarios as well as other scenarios.

In order to test your code with our own validator, we have asked you to place all simulation code into a method **Simulation.runSimulation()** which returns a **List** of **SimulationEvent** objects which we can then pass directly to things like the **Validate.validateSimulation()** method.

Submission
Every file you submit should have your name as the author. To enforce academic integrity, your code may be checked for similarity to other submissions, including some from the previous semesters. You should know that although this assignment is very similar to the ones assigned earlier, there are some subtle differences. Submit a .zip file containing your project files to Assignments in Blackboard.

Two phases:

The above description applies to two homeworks: Homework 4 and Homework 5. For Homework 4 (due next week) you need to develop only two things:
- A document that summarizes your approach to synchronization (max 1 page).
- Pre/post conditions, invariants and exceptions (similarly as for the previous homework). Put all of those into one text file; separate all of them by headings; use Java File names as headings.

For Homework 5 (due in 2 weeks) you will have to upload your source code with a build.xml file.