Project 2 – Ratsie's Simulation

Due: Fri. 2016-03-04 11:59:59 pm

Background. Ratsie's was a College Park institution that nurtured generations of Maryland students through the rigors of college life. (Some of you may have eaten there at some point!) The food served there was not what your nutritionist would recommend, but whenever you needed a dose of sodium and saturated fat, it was hard to beat. It closed in early 2015.

Goal. In this project, you will write a simulation for (a highly simplified) Ratsie's (-like restaurant). The simulation has five parameters: the number of customers wishing to enter Ratsie's; the number of tables in the restaurant; 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 customers' orders will be randomized.

Ratsie's customers 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 version of Ratsie's will only have four food items, each made by a single machine: an order of chicken wings, made by machine Fryer, which takes 350s (i.e. 350 seconds) to make; a pizza, made by machine Oven, which takes 600s to make; a toasted sub, made by machine GrillPress, which takes 200s to make; and soda, made by machine Fountain, which takes 15s to make. This information is summarized in the table below.

Machine	Food	Cook time (s)
Fryer	wings	350
Oven	pizza	600
Grill Press	subs	200
Fountain	soda	15

Getting started. The project skeleton can be downloaded as a zip file from the course website. Follow the directions given in Project 0 to create a project, named "p2", in Eclipse.

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.lang.Object`. You may use the regular (i.e. non-synchronized) version of collections if you wish, however.

In addition, you must use the following classes and the methods specified for them (most of which 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 may 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` – the `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; do not change this!
- `FoodType.java*` – contains the four food types we have defined above
- `Customer.java` – runs in a thread to implement a Ratsie's patron
- `Cook.java` – runs in a thread; makes orders provided by customers
- `Machine.java` – machines for cooking particular food items
- `SimulationEvent.java*` – instances represent "interesting" events that occur during the simulation. Do not change this; just use the events in the way described later in this assignment.
- `Validate.java` – defines method to validate the results of the simulation. You will use this method to check the results of testing 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.)

Specification in Detail.

- `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 have 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()`. The simulation consists of the given number of customers, cooks, and tables. It also involves exactly four machines, each with a given capacity.

or this project, there will be two scenarios for orders; (1) if `randomOrders` is set to false, then each Customer places the same order: one portion of wings, one pizza, one sub, and one soda but (2) if `randomOrders` is set to true, then each customer will place an order with a random number of portions of wings, pizzas, subs, and sodas, where each random number will be between 0 and 2. These items will be in a list sent into the `Customer` constructor. Once all Customers have completed, the simulation terminates, shutting down the machines, and calling interrupt on each of the cooks telling them they can go home. Because each of the Runnable `Cook` objects will be running in a thread, a call to `interrupt()` on that thread which will cause the Cook's `run()` method to throw an `InterruptedException.` We have put a try/catch block there for you which invokes `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. Objects in this class represent a kind of food item. You will create only four kinds food items for your simulation (wings, pizza, sub, and soda, as described above) but your classes should treat food items generically; that is, you should be able to easily change just the Simulation class if you want the simulation to have Customers 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)`: 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. 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 Ratsie's, places an order, waits for their order, eats the order, leaves. (Note that there can only be as many customers inside the Ratsie's as there are tables at any one time. Other customers must wait for a free table before going into Ratsie's and placing their order.) Customers will generate the following events:
    - Before entering Ratsie's: `SimulationEvent.customerStarting()`
    - After entering Ratsie's: `SimulationEvent.customerEnteredRatsies()`
    - Immediately *before* placing order: `SimulationEvent.customerPlacedOrder()`
    - After receiving order: `SimulationEvent.customerReceivedOrder()`
    - Just before leaving the Ratsie's: `SimulationEvent.customerLeavingRatsies()`

- `Cook` (implements `Runnable` – needs to run as its own thread)
  - Constructor `Cook(String name)`: the name is the name of the cook. You may extend this constructor with other parameters if you wish. All cook names must be unique and be of the form "Cook " + `num`, where `num` is between 0 and the number of cooks minus one.
  - `run()`: waits for orders from the restaurant, processes the order by submitting each food item to an appropriate machine. 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. The cook should not wait for each item to be completed before moving on to the next item. In particular, if the cook needs to cook multiple items, s/he can place all of those requests to the relevant machine(s), one after the other, without waiting for the previous request to be filled. (Note that this applies even if the requests are for the same machine.) For example, one machine could be cooking two batches of wings at the same time for the same order, while another could be making a pizza at the same time for this order as well. 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(MachineType machineType, Food food, int capacity)`: the type of the machine, the kind of food it makes, and the capacity (how many `Food` items may be in process at once). `MachineType` is an enumerated type that is declared in this class.
  - `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; for example, 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.

You may add helper classes and methods as you see fit to the skeleton code.

Testing. We have allowed a fair amount of freedom in the design of your classes for this project. In particular, we have allowed you to modify the constructors of many classes (such as `Machine` and `Customer`), and pre-written unit tests that access these classes would not work since the tests would not know the signatures of the constructors and methods. In our grading, we will instead use our own simulation-validation routine to check that the simulation output of your code is correct. These checks will be performed multiple times on multiple runs of your code in order to check for things like thread safety.

For these reasons, 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 the `Validate.validateSimulation()` method.

Your `validateSimulation()` method will not be graded, although you still must turn one in. (If you do not use the method in your programming, just turn in a method that does nothing. Please note, however, you are strongly advised not to rely only on hand-checking of simulation results.) You are encouraged to collaborate with others in this class in developing this method, and it is fine in this case if you and your collaborators turn in the same implementation of it. In this case, please put in a header comment in the `Validate` class

indicating who participated in the development of `validateSimulation()` method you use. ***All other code that you turn in (excepting the files we gave you) must be your own work.***

Sharing of tests for this project is also encouraged.

Submission. Submit a .zip file containing your project files to the CS submit (submit.cs.umd.edu). You may use the same approach outlined in Project 0 to create this from inside Eclipse.