It’s Pizza Time

1 Introduction

This assignment will have you using several of the default Java Collections classes to solve a simulation of pizza deliveries.

2 Problem Description

The small town of Suwanee has a prominent pizza market. The town government has been wrestling with some new legislation that would force all pizza shops to close at midnight. The government is worried that this new legislation would give an advantage to certain pizzerias. They have asked you to create a model of pizza deliveries and to demonstrate if this legislation would create an unfair advantage in the pizza market. The pizza market is dominated by three players, Mary Sue Pizza, Pizza Shack, and Cascade Pizza.

In the process of investigating the pizzerias in the town, you discover that Pizza Shack and Mary Sue have some questionable business practices.

Mary Sue schedules pizza orders based on the price of the order, apparently choosing to give their higher-paying customers priority.

Pizza Shack schedules orders based on both the price of the order and a customer’s ability to pay for the food that they’ve ordered. It is unclear how Pizza Shack determines whether a customer can pay for their pizza before the pizza is delivered, but their information is uncannily reliable. Additionally, Suwanee townspeople’s desire for pizza often exceeds their ability to pay, so the practice helps Pizza Shack cut the costs of wasted pizzas.

Cascade processes pizza orders in the same order they were received. This strategy is based on the owners’ strong beliefs that services should be provided equally to all. Unfortunately, Cascade has been having a hard time staying out of the red.

3 Solution Description

We provide the following classes:

- `Ingredient` an enum of Ingredients with their price.
- `Driver` The simulation driver. The driver handles generating customers and their orders based on the Ingredient enum. The driver has a `public final Random` object that you should use in your classes in order to get reproducible results. The Driver may generate orders that do not fit the menu because sometimes people order things that are not on the menu. Some more things about the driver:
  - Every Pizzeria has the same menu
  - Every Pizzeria receives the same n orders from n customers. 1 order per customer.
- `Pizzeria` A pizzeria interface. This describes the behavior of the methods a pizzeria must have.

You need to create the following classes

- `Order` A representation of a set of `Ingredients` a `Customer` may order. Order should extend `java.util.HashSet` with a type parameter of Ingredient.
- `Customer` A representation of a customer with a name, a randomly generated amount of money between $5 and $34, and an idea of an `Order` that they want. They can also answer whether they have enough money to pay for their pizza. Customers are also comparable based on the price of their orders, where **the Customer with more money should come first.**
- `AbstractPizzeria` A representation of an abstract pizza place. All of the pizza shops in town are run exactly the same way except for the strategy they use to schedule orders. This strategy will be supplied by the concrete subclasses. The Pizzeria interface explains what each method should do.
  - A Pizzeria is created with a menu which is a Set of Orders.
  - A Pizzeria uses a Queue to organize its Customers. Subclasses provide a specific type of Queue to impose an order on their waiting list.
  - A Pizzeria knows how many orders it receives, how many orders it attempts to deliver, how many orders are successfully delivered, and its total revenue.
- `CascadePizza` A normal pizza place. Delivers pizzas in the same order they were ordered in, regardless of price or the customer's ability to pay.
- `MarySuePizza` A cunning pizza place. Delivers orders based on their price. Expensive orders get delivered first.
- `PizzaShack` A sophisticated (and probably criminal) pizza place. Delivers orders based on both price and the customer's ability to pay.

An example run with a Random seed of 10 gives something like

```
$ java Driver

Delivered to 115 customers when...
Midnight!!!
Cascade Pizza
 - We delivered 4% of our orders! We delivered 38% of our attempted orders and made $259.00
Mary Sue Pizza
 - We delivered 3% of our orders! We delivered 28% of our attempted orders and made $864.00
Pizza Shack
 - We delivered 7% of our orders! We delivered 61% of our attempted orders and made $1,664.00
```

4 Tips

- This homework is intended to familiarize you with the Collections API. As much as possible, find methods in the Collections classes you're using to achieve what you want to do. The `Collections` static class could also be helpful.
- If you're unsure about what public methods one of your classes should have, you will find some of them in the driver.
- A queue is a data type that supports at least `add()` and `remove()` operations. A simple queue will return elements from `remove()` in the same order they were added in. Imagine a line to pay for something at a store: that is a queue. For example,

  ```
  q.add(1);
  q.add(3);
  q.add(2);
  q.remove(); //returns 1
  q.remove(); //returns 3
  q.remove(); //returns 2
  ```

- Java has no default queue implementation but it does provide a deque (read as 'deck') which is a more sophisticated queue. You can use a deque to simulate a simple queue.
- An even more sophisticated queue arranges the elements inside based on some ordering. This priority queue would give you elements in an order which may or may not be the order you added them in.

  ```
  //with a priority queue that gives lowest values first
  pq.add(2);
  pq.add(1);
  pq.add(3);
  pq.remove(); //returns 1
  pq.remove(); //returns 2
  pq.remove(); //returns 3
  ```

  Don't worry about the internals of Java's priority queue, you just need to give it a way to compare elements to make it do this.
- A set is an unordered collection of unique items. If you look in java's Set interface there is no way to get the ith member of a set. That kind of operation is undefined. How could you

do something with the members of a set?

5 Javadocs

We are going to have you do Javadocs for this assignment (and for all assignments here on out). Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful. The relevant tags that you need to have are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```java
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 13.31
 */
public class Dog {

   /**
    * Creates an awesome dog (NOT a dawg!)
    */
   public Dog(){
      ...
   }

   /**
    * This method takes in two ints and returns their sum
    * @param a first number
    * @param b second number
    * @return sum of a and b
    */
   public int add(int a, int b){
      ...
   }
}
```

6 Checkstyle

You must run checkstyle on your submission. The checkstyle cap for this assignment is **50** points. Review the Style Guide and download the Checkstyle jar and associated XML file. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count Checkstyle errors by piping the output of Checkstyle through `wc -l` and subtracting 2 for the two non-error lines printed above (which is how we will deduct points). For example:

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | wc -l
     2
```

Alternatively, if you are on Windows, you can use the following instead:

```
C:\> java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | findstr /v "Starting
    audit..." | findstr /v "Audit done" | find /c /v "hashcode()"
0
```

Food for thought: is there a one-liner like above that shows you only the number of errors? Hint: `man grep`.

The Java source files we provide contain no Checkstyle errors. For this assignment, there will be a maximum of **50** points lost due to Checkstyle errors (1 point per error). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

7 Turn-in Procedure

Submit all of the Java source files you modified and resources your program requires to run to T-Square. Do not submit any compiled bytecode (`.class` files), the Checkstyle jar file, or the `cs1331-checkstyle.xml` file. When you're ready, double-check that you have submitted and not just saved a draft.

Please remember to run your code with Checkstyle!

Verify the Success of Your Submission to T-Square
Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.

2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.

3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.

4. Recompile and test those exact files.

5. This helps guard against a few things.

(a) It helps insure that you turn in the correct files.

(b) It helps you realize if you omit a file or files. [1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

(c) Helps find last minute causes of files not compiling and/or running.

[1]Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight. Do not wait until the last minute!