Module 6: Building Client Classes that Use Your Gate Class

Introduction

A well-written Java class can be reused in many different client classes. In this assignment you will use your *Gate* class from the first assignment in Module 6 to model two different scenarios where a *Gate* object would be useful to manage snails in an escargatoire (which is a fancy name for a nursery of snails). Make sure to copy your working *Gate.java* file from the first assignment in Module 6 into your new project. You should not need to make any changes to this file—it will be reused by the two client files that you'll write based on the specification below.

Learning outcomes

When you have completed this exercise you will be able to

- Create client files that use the *Gate* class to model "real world" scenarios
- Instantiate *Gate* objects and create arrays of *Gate* objects
- Generate random integers and random *Boolean* values
- Call methods on objects

Resources

Along with this specification document, you are provided with an Android Studio project to download and use on your computer. This project contains Java files organized into the following two directories:

- ***app/src/main/java/mooc/vandy/java4android/gate/logic*** -- This directory contains several files you need to implement, as described in the "**What You Need to Do**" section below. It also contains several files whose class implementations are provided for you. In particular, the *Logic.process()* method in the *Logic.java* file creates a *FillTheCorral* object and 4 *Gate* objects and uses them to corral all the snails. Likewise, the *Logic.process()* method creates a *HerdManager* and two *Gate* objects and tests that the *HerdManager.simulateHerd()* method works properly to simulate the movement of snails in and out of the escargatoire.

- ***app/src/main/java/mooc/vandy/java4android/gate/ui*** -- This directory contains a class and an interface that are provided for you. The *MainActivity.java* file contains the Android Activity that defines UI for this app and calls the *Logic.progress()* method to test your class implementations. You don't need to know anything about the contents of this file. The *OutputInterface.java* file contains a Java interface called *OutputInterface* that defines methods (which are implemented by *MainActivity*) that the classes you write can use to print various messages to the UI. We therefore recommend you examine *OutputInterface* to learn what methods are available for use in your classes.

In addition to these files, there are also unit tests in the ***app/src*** directory. Running these unit tests will provide you feedback on the correctness of your class implementations. They are also the same unit tests used by the auto-grader.

If you choose to have your solution evaluated by your peers (which is optional and doesn't count towards your final grade on this assignment) they will need to download and compile your code. Your code should therefore be importable into Android Studio, should compile without error, and should then run correctly on an emulated Android device.

What You Will Need to Do

Turn in**: HerdManager.java**

You will create a *HerdManager.java* class that simulates the movement of snails in and out of the escargatoire. The following UML diagram shows the fields and method in this class that you are required to implement.

HerdManager
- mOut: OutputInterface **- mWestGate: Gate // IN gate** **- mEastGate: Gate // OUT gate** **- MAX_ITERATIONS: int = 10** **+ HERD: int = 24**
+ HerdManger(OutputInterface out, Gate gate1, Gate gate2) // constructor **+ simulateHerd(Random rand): void**

Notes on ***HerdManager****:*

- Create a *public static final int HERD* to indicate the size of your escargatoire, which should be set to the constant value 24 for this simulation.
- Create a public *HerdManager* constructor that is passed an *OutputInterface* and two *Gate* parameters: a *gate1* and a *gate2*. This constructor should store these parameters in fields called *mWestGate* and *mEastGate*, respectively. The constructor should also call the *Gate open()* method to set *mWestGate* to swing *IN* and *mEastGate* to swing *OUT*. The *OUT* gate (*mEastGate*) will allow the snails to leave the pen and go out pasture. Likewise, the *IN* gate (*mWestGate*) will allow snails to re-enter the pen from the pasture.
- Create a *simulateHerd()* method that accepts a *Random* object as an input parameter. Set a local variable inside of *simulateHerd()* equal to the size of the *HERD*. The method will run ten iterations of the simulation. In each iteration, use the *Random* object passed as a parameter to the method to randomly select one of the pen gates and move a random number of snails through that gate (*IN* or *OUT* depending on the gate's swing state), thereby changing the number of snails in the pen and out to pasture.

  You must be sure that neither of the numbers "in the pen" or "out to pasture" is ever negative and that the sum total of snails is always equal to the size of the *HERD*. If there are no snails currently out to pasture during some iteration do not *randomly* select a

gate, but instead move a random number of snails out of the pen and into the pasteur using your *OUT* gate (*mEastGate*).

The range of random numbers generated will change according to which gate you have selected and how many snails are currently available to go through that selected gate, but should always be greater than 0. Print out the necessary information for each iteration as shown in the sample run of *HerdManager.java* below. Note that the first output line has been included in your skeleton program. Your output format must match the sample exactly as shown below.

Sample output

The following output is obtained by using a seed value of '1234' for the random number generator, which determines its starting point for randomness. Different seed values would result in different output.

HerdManger.java Output:

```
East Gate: This gate is closed
West Gate: This gate is closed
There are currently 24 snails in the pen and 0 snails in the pasture
There are currently 3 snails in the pen and 21 snails in the pasture
There are currently 6 snails in the pen and 18 snails in the pasture
There are currently 15 snails in the pen and 9 snails in the pasture
There are currently 20 snails in the pen and 4 snails in the pasture
There are currently 2 snails in the pen and 22 snails in the pasture
There are currently 9 snails in the pen and 15 snails in the pasture
There are currently 12 snails in the pen and 12 snails in the pasture
There are currently 18 snails in the pen and 6 snails in the pasture
There are currently 23 snails in the pen and 1 snails in the pasture
There are currently 24 snails in the pen and 0 snails in the pasture
```

The following figures show various stages of operations during the simulation of the snail herding.

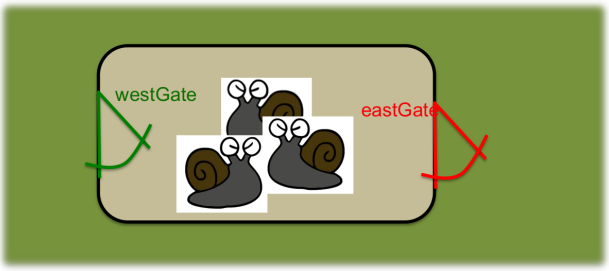

Figure 1: Must select eastGate

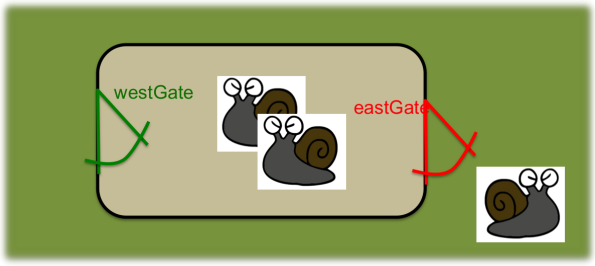

Figure 2: Randomly select a gate

Turn in**: FillTheCorral.java**

You will create a *FillTheCorral.java* class that that simulates moving snails from a pasture

into four different corrals. Below is the UML diagram for the *FillTheCorral*, whose methods are called by the *Logic.process()* test method in the *Logic.java* file.

FillTheCorral
- mOut: OutputInterface // set in constructor
+ FillTheCorral(OutputInterface out) // constructor **+ setCorralGates(Gate[] gate, Random selectDirection): void** **+ anyCorralAvailable(Gate[] corral): boolean** **+ corralSnails(Gate[] corral, Random rand): int**

Notes on ***FillTheCorral:***

- The *FillTheCorral()* constructor given to you in the provided skeleton simply saves a reference to the passed *OutputInterface* parameter so that you can print strings to the user as the tests progress.
- Create the *setCorralGates()* method that is passed an array of *Gate* objects and a *Random* object and sets the direction of each gate's swing. Recall that when *Gate* objects are created, they are originally set to CLOSED. Your method will randomly set each gate to swing *IN*, *OUT*, or leave it CLOSED.
- Create *anyCorralAvailable()* method that is passed an array of *Gate* objects and returns a value of true or false. Once the gates are randomly set, they must be checked to ensure that at least one is set to swing IN so that snails can enter at least one corral (returns true). This method returns a boolean value of false if all gates are set to OUT or CLOSED indicating the provided *process()* method of the *Logic* class must try to set them again. The do-while loop in this method will continuously call s*etCorralGates()* until a configuration with at least one IN gate is created.
- Create *corralSnails()* that is passed an array of *Gate* objects and a *Random* object and runs the simulation, as follows:
    - The method begins with 5 snails out to pasture
    - Generate a random number of snails *s*, not to exceed the number already out to pasture
    - Randomly select one of the four Gate objects *G*
    - Attempt to move *s* snails through gate *G* and adjust the number of snails out to Pasture.
    - For each iteration, use *mOut.println()* to display the attempted movement of snails. The simulation ends when all of the snails have been corralled.
    - Finally, this method prints and returns the number of attempts that were required to corral all the snails (see sample output below).
- Here's a summary of how the movement of snails changes the count of snails out to pasture
    - If the chosen *Gate G* is set to IN and allows for entry into the corral, the snails enter the corral and the number of snails out to pasture is reduced.
    - If the chosen *Gate G* is CLOSED, the snails do not enter and the number out to pasture is unchanged.
    - If the chosen *Gate G* is set to OUT and allows snails to exit the corral, the same number of snails that attempted to enter that corral actually exits the corral,

thereby increasing the number of snails out to pasture. You can assume that a corral will never run out of snails to send out to pasture.

- Use caution when calculating the pasture count. Recall that in the previous assign in *Module 6: Building Your Own Class*, you wrote the method *Gate.thru()* to return the positive value of *n* when *n* snails pass thru an IN gate. Does movement through an IN gate in the current assignment mean the number of snails in the pasture increases? Likewise, should the number of snails in the pasture decrease?

Sample Output

The following output is obtained by using a seed value of '1234' for the random number generator, which determines its starting point for randomness. Different seed values would result in different output.

FillTheCorral.java Example Output:

```
Initial gate setup:
Gate 0: This gate is open and swings to enter the pen only
Gate 1: This gate is open and swings to enter the pen only
Gate 2: This gate is open and swings to enter the pen only
Gate 3: This gate is open and swings to exit the pen only
4 are trying to move through corral 3
5 are trying to move through corral 1
1 are trying to move through corral 1
3 are trying to move through corral 0
It took 4 attempts to corral all of the snails.
```

The following figures show various stages of operations during the corralling of the snails. A green gate is set to IN and allows snails into a corral. A red gate is set to OUT and allows snails out to the pasture.

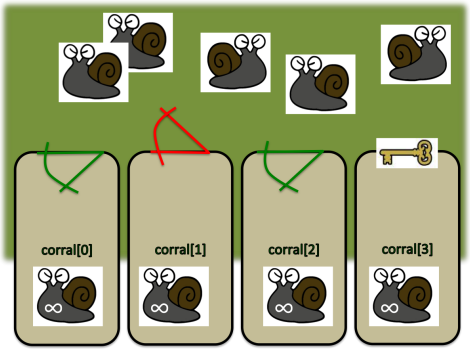

Figure 3: Example initial configuration

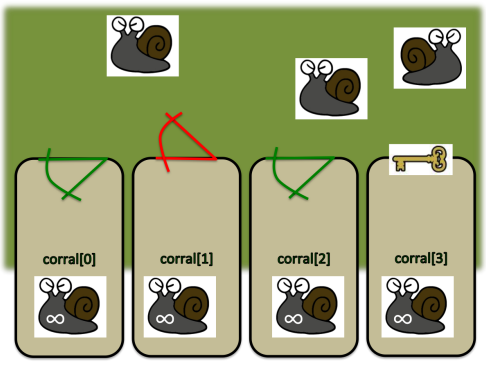

Figure 4: After corral[0], 2 snails

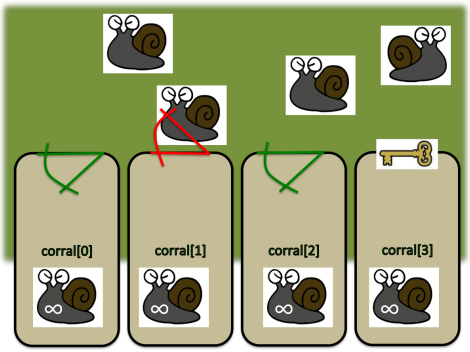

Figure 5: After corral[1], 1 snail; note 1 additional snail is sent out to pasture

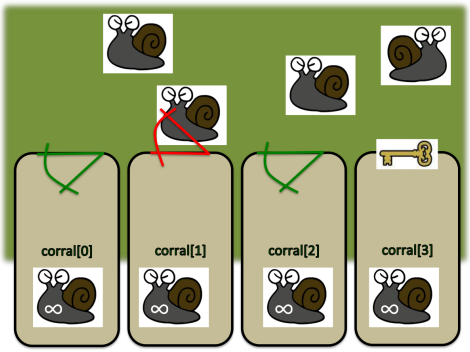

Figure 6: After corral[3], 3 snails; note no change

Source code aesthetics (commenting, indentation, spacing, identifier names):
If you choose to have your solution reviewed via the optional peer grading mechanism you'll be evaluated against a number of criteria. For example, you are required to properly indent your code and will lose points if you make significant indentation mistakes. No line of your code should be over 100 characters long (even better is limiting lines to 80 characters). You should use a consistent programming style, which should include the following:

- Meaningful variable and method names
- Consistent indenting
- Use of "white-space" and blank lines to make the code more readable
- Use of comments to explain pieces of complex code