Module 6: Building client files that use your 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 Will 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.process()* 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 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
+ HerdManager(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 to pasture. Likewise, the *IN* gate (*mWestGate*) will allow snails to reenter the pen from the pasture.
- Create a *simulateHerd()* method that accepts a *Random* object as an input parameter. Create a local variable inside of *simulateHerd()* that will keep track of how many snails are inside the pen and set it equal to the size of the *HERD* . The method will run ten iterations of the simulation. On each iteration, you will first choose a gate to move snails thru, and then choose how many snails to move through that gate. If all the snails are in the pen or all are in the pasture, pick the gate that will allow some of the snails to move (i.e., if all of the snails are in the pen then choose the OUT gate), otherwise use the *Random* object passed as a parameter to the method to randomly select one of the two pen gates. After you have chosen a gate, move a random number of snails through that

gate (*IN* or *OUT* depending on the chosen gate's swing state), thereby changing the number of snails in the pen and out to pasture. You should always move at least one snail and you cannot move more than are available to move through the chosen gate, so generate a random value in the correct range of values.

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*.

Print out the necessary information before the start of the simulation and for each of the 10 iterations as shown in the sample run of *HerdManager.java* below (that's 11 lines of output). Note that the first two output line displayed below are printed prior to calling the *simulateHerd()* method. Your output format must match the sample exactly as shown below. If your results do not match, try flipping the logic used to select the random gate.

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:

```
West Gate: This gate is open and swings to enter the pen only
East Gate: This gate is open and swings to exit the pen only
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
```

From *Logic.process()*

From *simulateHerd()*

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

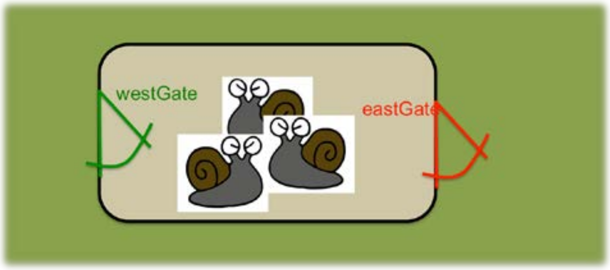

Figure 1: Must select eastGate

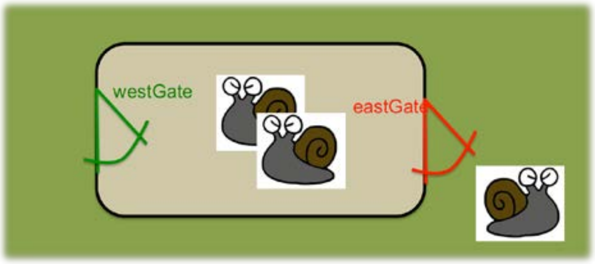

Figure 2: Randomly select a gate

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 rand) : 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. Your method will randomly set each gate to swing *IN*, be CLOSED, or swing *OUT*. (Recall that IN is 1, CLOSED is 0, and OUT is -1. Thus, you should generate a random integer in the range 0-2 and then subtract 1.) The status of each gate is printed to *mOut* as they are set – be sure to utilize the *toString()* method of the *Gate* objects (see sample output below). Be sure to set the *Gate* objects in the order they appear in the array.
- Create *anyCorralAvailable()* method that is passed an array of *Gate* objects and returns a value of true or false. This method returns a boolean value of true if at least one gate in the array is set to swing IN so that snails can enter at least one corral. This method returns a boolean value of false if all gates are set to OUT or CLOSED.
- 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.
  - Randomly select a Gate objects *G* from the array of Gate objects by generating a random index into the array*.*
  - Generate a random number of snails *s* to attempt to move. This number should be in the range 1 up to the number out to pasture. You must always move at least one snail, and you cannot attempt to move more than are out to pasture.
  - 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. See the sample output below.
  - The simulation ends when all of the snails have been corralled (i.e., none are left in the pasture).
  - 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 exit 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 assignment 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 pen 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
```

From *setCorralGates()*

```
4 are trying to move through corral 2
1 are trying to move through corral 3
1 are trying to move through corral 3
2 are trying to move through corral 1
1 are trying to move through corral 1
It took 5 attempts to corral all of the snails.
```

From *corralSnails()*

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.

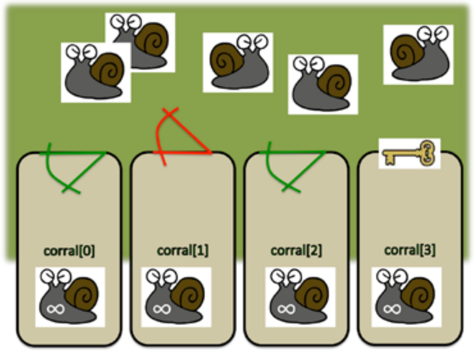

Figure 3: Example initial configuration

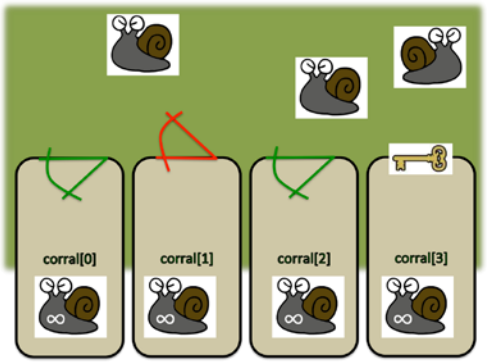

Figure 4: After corral[0], 2 snails

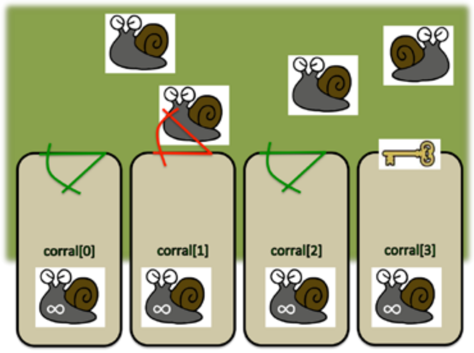

Figure 5: After corral[1], 1 snail; note 1 additional snail is sent out to pasture

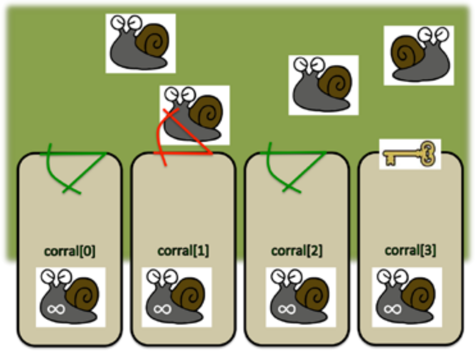

Figure 6: After corral[3], 3 snails; note no change

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 & 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

Submission:

See the assignment page in Coursera for instructions on using gradle to create the necessary zip file and submit it for evaluation by the auto-grader.