Module 6: Building client files that use your class

In this assignment you will see for yourself how a well-designed object can be used in two different client files.

Learning outcomes:

When you have completed this exercise you will be able to

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

Introduction:

A well-written class file can be used in many different client files. In this assignment you will use your Gate class to model two different scenarios where your object would be useful.

Resources:

Along with this specification file, you are provided with a project file to download to Android Studio. Copy your working Gate.class file into your new project. You should not need to make any changes to this file however it will be used by your two client files.

In order for evaluators to do their job 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**

In this exercise, you will build a client program that simulates the movement of an escargatoire of snails in and out of a pen. Create a public static final integer HERD to indicate the size of your escargatoire. Set the constant to 24 for this simulation.

Create two gates in your main method, an westGate and a eastGate. Build the setGates method that accepts as input parameters, two Gate objects. This method will set the eastGate to OUT, to allow snails to leave the pen and go to pasture and set the westGate to allow snails to re-enter the pen, leaving the pasture. *To make your program compatible with the autograder, pass the parameters in order `westGate, eastGate`.*

Create the simulateHerd method that accepts two Gate objects as input parameters. Set the number of snails in the pen equal to the size of the HERD. *To make your program* *compatible with the autograder, pass the parameters in order `westGate, eastGate`.* The method will run ten iterations of the simulation. In each iteration, randomly select one of the gates to the pen and move a random number of snails through that gate (in or out depending on the swing value), thus changing the number of snails in the pen and out to pasture. You must be sure that neither of these 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, during some iteration, there are no snails currently out to pasture, then you would not *randomly* select a gate, but would move a random number of snails through the eastGate and out to pasture. Also, 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 included in this specification. Note that the first output line has been included in your shell program. Your output format must match the sample exactly.

Sample output:

HerdManger.java Example Run 1:
```
East Gate: This gate is locked
West Gate: This gate is locked

East Gate: This gate is not locked and swings to exit the pen only.
West Gate: This gate is not locked and swings to enter the pen only.
There are currently 24 snails in the pen and 0 snails in the pasture

There are currently 8 snails in the pen and 16 snails in the pasture
There are currently 2 snails in the pen and 22 snails in the pasture
There are currently 20 snails in the pen and 4 snails in the pasture
There are currently 5 snails in the pen and 19 snails in the pasture
There are currently 1 snails in the pen and 23 snails in the pasture
There are currently 3 snails in the pen and 21 snails in the pasture
There are currently 9 snails in the pen and 15 snails in the pasture
There are currently 22 snails in the pen and 2 snails in the pasture
There are currently 24 snails in the pen and 0 snails in the pasture
There are currently 23 snails in the pen and 1 snails in the pasture
```

HerdManger.java Example Run 2:
```
East Gate: This gate is locked
West Gate: This gate is locked

East Gate: This gate is not locked and swings to exit the pen only.
West Gate: This gate is not locked and swings to enter the pen only.
There are currently 24 snails in the pen and 0 snails in the pasture

There are currently 4 snails in the pen and 20 snails in the pasture
There are currently 1 snails in the pen and 23 snails in the pasture
There are currently 19 snails in the pen and 5 snails in the pasture
There are currently 9 snails in the pen and 15 snails in the pasture
There are currently 5 snails in the pen and 19 snails in the pasture
There are currently 9 snails in the pen and 15 snails in the pasture
There are currently 13 snails in the pen and 11 snails in the pasture
There are currently 15 snails in the pen and 9 snails in the pasture
There are currently 12 snails in the pen and 12 snails in the pasture
There are currently 14 snails in the pen and 10 snails in the pasture
```

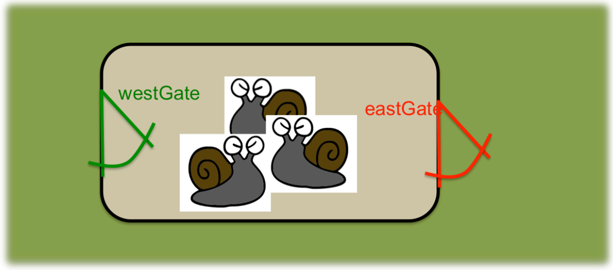

Figure 1: Must select eastGate

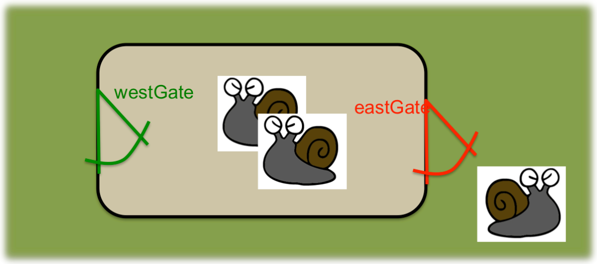

Figure 2: Randomly select a gate

Turn in: **FillTheCorral.java**

In this exercise, you will build a client program that simulates moving snails from a pasture into four different corrals. It's a bit of a game really. Following the best practices for writing programs, fill in the methods provided in the shell to accomplish the following. Using a Random object to determine swing direction or closed, each of the four corrals is set to swing IN to the corral, OUT of the corral, or remain locked. The program begins with 5 snails out to pasture and an infinite number of snails in each corral. A random number of snails, not to exceed the number already out to pasture, attempt to pass through a randomly chosen corral gate. If the gate allows for entry, the snails enter the corral and the number of snails out to pasture is reduced. If the gate is locked, they do not enter. If the gate is set to exit the corral however, the same number of snails that attempted to enter that corral actually exits the corral thus increasing the number of snails out to pasture. This continues until finally there are no snails out to pasture.
We must suspend reality a bit to run this game. For example, we must imagine there are an infinite number of snails in each coral waiting to get out to pasture. Note that when setting the gates randomly to IN, OUT or locked, be sure that at least one gate is set to IN so that the game can finally end. A natural way to do this with such a small number of corral gates is to set them, then test to be sure at least one is IN. If not, try setting all four corral gates again. Another algorithm would randomly set all four gates, then randomly select one of the gates to be set to IN, overriding its original setting. It is also conceivable that this game goes on infinitely, or that the ever increasing number of snails out to pasture exceeds the size of the largest int for your computer and the program crashes or begins delivering anomalous results. If this occurs, simply kill your program. Here is a brief description of the methods provided in the shell:

Method	Description
`main`	Create PrintStream for output; create Random object to generage ints and Bollean values; contains some logic; makes method calls
`setCorralGates`	Sets the corral gates as described; printout gate status when they are successfully set
`anyCorralAvailable`	Checks the setting of all four gates to be sure that at least one is set to IN
`corralSnails`	Runs the simulation, prints attempted movement of snails and the current status

Your output format must match the sample exactly.
Sample output:

FillTheCorral.java Example Run 1:
```
Initial gate setup:
Gate 0: This gate is not locked and swings to enter the pen only.
Gate 1: This gate is not locked and swings to exit the pen only.
Gate 2: This gate is locked
Gate 3: This gate is not locked and swings to exit the pen only.
5 snails are trying to move through locked coral 2.
There are currently 5 snails still in the pasture.
2 snails are trying to move through locked coral 2.
There are currently 5 snails still in the pasture.
4 snails are trying to move through entry coral 0.
There are currently 1 snails still in the pasture.
1 snails are trying to move through entry coral 0.
There are currently 0 snails still in the pasture.
It took 4 attempts to coral all of the snails.
```

FillTheCorral.java Example Run 2:
```
Initial gate setup:
Gate 0: This gate is not locked and swings to enter the pen only.
Gate 1: This gate is not locked and swings to exit the pen only.
Gate 2: This gate is not locked and swings to enter the pen only.
Gate 3: This gate is not locked and swings to exit the pen only.
1 snails are trying to move through exit coral 1.
There are currently 6 snails still in the pasture.
4 snails are trying to move through entry coral 0.
There are currently 2 snails still in the pasture.
1 snails are trying to move through exit coral 3.
There are currently 3 snails still in the pasture.
3 snails are trying to move through exit coral 1.
There are currently 6 snails still in the pasture.
6 snails are trying to move through entry coral 2.
```

```
There are currently 0 snails still in the pasture.
It took 5 attempts to coral all of the snails.
```

FillTheCorral.java Example Run 3 (partial printout):

```
…
66 snails are trying to move through entry coral 3.
There are currently 18 snails still in the pasture.
12 snails are trying to move through exit coral 1.
There are currently 30 snails still in the pasture.
30 snails are trying to move through entry coral 3.
There are currently 0 snails still in the pasture.
It took 330 attempts to coral all of the snails.
```

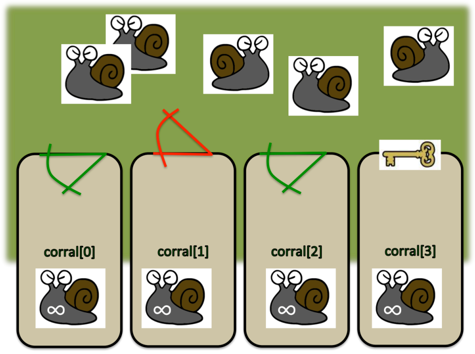

Figure 3: Example initial configuration

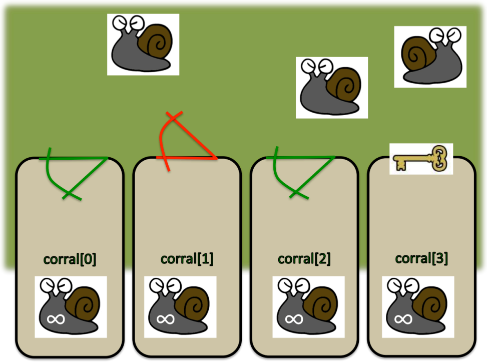

Figure 4: After corral[0], 2 snails

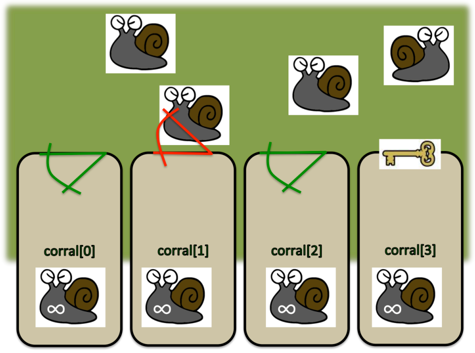

Figure 5: After corral[1], 1 snail; note 1 additional snail is sent out to pasture

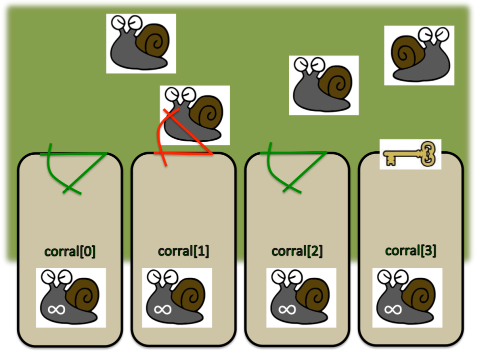

Figure 6: After corral[3], 3 snails; note no change

Source code aesthetics *(commenting, indentation, spacing, identifier names)*:
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. This 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