Module 6: Building your own class

In this assignment you will create a class file, Gate.java. You will use this file in the second assignment, creating two client files that use the class. Submitting this file independently (before working on the client files) will enable you to be sure that you have built the class correctly. This will make the task of creating the client files in the next assignment go much more smoothly.

Learning outcomes:

When you have completed this exercise you will be able to

- Read a UML diagram and build the corresponding class
- Create a class file that includes instance variables, constructors and methods necessary to interact with objects of the class

Introduction:

Recall that one of the design goals of building classes is to create classes that are reusable in different scenarios modeled by different client files. In this assignment you will create a class file with an eye towards using it in future assignments.

Resources:

Along with this specification file, you are provided with a project file to download to Android Studio.

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: **Gate.java**

Create a Gate class that can be used to represent a gate for a livestock pen. There will be two states for this gate, `locked`, which would be a boolean variable, and `swing`, which we will create as an `int`. The field `swing` will indicate which direction the gate is able to swing. To make our programs more readable, create two `public static final int`'s to indicate the swing direction.

- `OUT` = -1 to swing out, to let animals leave the pen or enclosed area
- `IN` = 1 to swing in, and let animals enter the pen or enclosed area

When a gate is first constructed, it should be locked and the swing direction should remain 0, the default value.

Create set methods for (i) the direction of `swing` as well as (ii) to open (`lock = false`) and (iii) close (`lock = true`) the gate. When setting the `swing` direction, return a boolean value to indicate if the swing was successfully set (`true`), or if an invalid swing direction was given (`false`, not set). When attempting to open the gate, the user must provide a swing direction. The `open()` method should return a boolean value as well, once again indicating if the setting of the swing direction was successful.

Provide get methods for the two fields as indicated in the table below.

Another behavior of the gate is that animals will go through it. Suppose *n* animals attempt to go through the gate. If the gate is set to the swing `OUT` position, the animals will leave the pen and the total number of animals in the pen will be decreased. If the gate is set to the swing `IN` position, animals will be entering the pen and the number of animals will be increased. If the gate is locked, there should be no change to the number of animals in the pen. Create a method that, given the input parameter *n*, will return *n*,-*n* or 0 depending on the position and locked status of the gate. *Note: Here you should be thinking like the author of the class—making your class as useful and general as possible. Instead of attempting to alter some total that is in or out of the facility that the gate is controlling, we are simply going to return the net change, to be used by the client as needed.*

Override the `toString()` method to match the output shown in these samples.

```
This gate is locked
```
//a gate that is set to locked

```
This gate is not locked and swings but the swing is not set properly
```
//a gate that is unlocked but the swing has not been set

```
This gate is not locked and swings to enter the pen only
```
//a gate that is unlocked and set to swing IN

```
This gate is not locked and swings to exit the pen only
```
// a gate that is unlocked and set to swing OUT

Requirements and method names for you class file are given in table form below. Name your fields and methods exactly as specified so that the included test program will work for your class.

Gate
<u>+IN: int= 1</u>
<u>+OUT: int= -1</u>
-swing: int
-locked: boolean

```
+setSwing(direction:int): boolean
+open(direction:int): boolean
+close()
+isLocked(): boolean
+getSwingDirection(): int
+thru(count:int): int
+toString(): String
```

A note on this table: *This is a simple version of a class diagram called a UML diagram or unified modeling language diagram. These types of diagrams are commonly used during the Software Engineering design process when programmers are first building classes and client files. They are programming language independent.*

- *an underlined variable name is* `static`
- *a variable name in all caps is* `final`
- *+ indicates the field or method is* `public`
- *- indicates the field or method is* `private`
- *the parameter list for a method is listed in () with descriptive name and type*
- *the return type is listed after the method*
- *if no constructors are listed, assume only the default is used*

Once you have built your class, you cannot run it because it is not executable code, but a class file that is used to instantiate objects in other files. You should test your Gate.java file using Logic.java This program will attempt to create a Gate object and invoke some of its methods. The relevant code is already created for you in the process method of Logic.java

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.

- Create helper method as needed *(not used in this exercise)*
- Meaningful variable & method names *(not critical for this exercise as most variable names are given in the spec)*
- Consistent indenting
- Use of "white-space" and blank lines to make the code more readable
- Use of comments to explain pieces of complex code

Peer Assessment:

You will be asked to evaluate five other student's projects as well as do a self-assessment of your own project. If you do not assess the work of others, then you will receive a 20% penalty for your solution.