Module 6: Building Your Own Class

Introduction

Recall that one design goal of building classes is to create components that can be reused in different scenarios modeled by different clients. In the first assignment in Module 6, therefore, you will create a Java class called *Gate* in a file called *Gate.java*. You will use this file in the second assignment in this module, creating two client files that use the *Gate* class. Submitting this file independently (i.e., before working on the client files) will ensure you've built the class correctly, thereby making the task of creating the client files in the next assignment go more smoothly.

In this assignment you'll need to understand portions of a *Unified Modeling Language* (UML) diagram (See https://en.wikipedia.org/wiki/Unified_Modeling_Language for more information about UML). UML is commonly used during an object-oriented software engineering process when programmers are first designing their classes. Although UML is an independent programming language, its notations map to Java as follows:

- *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*

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

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 a file 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* and *LogicInterface* classes are provided with this assignment to test the classes you implement. In particular, the *Logic.process()* method in the *Logic.java* file creates a *Gate*object and calls its methods to ensure they print the execpted output.

- ***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 Need to Do

Turn in: **Gate.java**

Requirements and method names for you *Gate* class are given in the UML diagram below. Name your fields and methods exactly as specified so that the included test program will work for your class.

Gate
+IN: int= 1 **+OUT: int= -1** **-swing: int** **-locked: boolean**
+setSwing(direction:int): boolean **+open(direction:int): boolean** **+close()** **+isLocked(): boolean** **+getSwingDirection(): int** **+thru(count:int): int** **+toString(): String**

Create a *Gate* class that can be used to represent a gate for a livestock pen containing sheep. There will be two fields for each instance of *Gate: mLocked*, which is a boolean variable, and *mSwing,* which is an *int.* The *mSwing* field indicates which direction the gate is able to swing. To make our programs more readable, create the following two *public static final int* values to indicate the swing direction.

- *OUT* = -1 to swing outward allowing the sheep to leave the pen or enclosed area

- *IN* = 1 to swing inward allowing the sheep to enter the pen or enclosed area

Note that each gate will only allow sheep to move in a single direction, either *IN,* or *OUT* of the pen. When an instance of *Gate* is first constructed, it should be locked and the *mSwing* direction should remain 0, the default value.

As shown in the UML diagram above, create a getter (*getSwingDirection)* and a setter (*setSwing)* for the the *mSwing* field**.** When setting the *mSwing* direction through the *setSwing* method , return a boolean value to indicate if the swing direction was valid (*true,* which means successfully set) or if an invalid swing direction was given (*false,* which means not successfully set).

When attempting to open the gate, the caller is required provide a swing direction. The *open()* method performs two tasks. It calls the *setSwing()* helper method to set the gate swing direction, and if this call fails. it returns false. Otherwise, it opens the gate by setting the "locked" state to false, and returns true to indicate success.

The primary purpose of the gate is to allow sheep to pass through the gate in either the *IN* or *OUT* direction. Suppose *n* sheep attempt to go through an instance of *Gate.* If this gate is set to the swing *OUT* position, the sheep will leave the pen and the total number of sheep in the pen will be decreased. If the gate is set to the swing IN position, sheep will be entering the pen and the number of penned sheep will be increased. If the gate is locked, there should be no change to the number of sheep in the pen.

Now you will create a method to control the movement of the sheep in and out of the pen. Name this method *thru()* and have it accept a single int parameter *count.* This method will return either count, -count, or 0 depending on swing position and locked status of the gate. You should make your class as useful and general as possible, e.g., 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 that a client can use as needed. For example, if the *thru()* method receives a count of 3, then if the gate swings *IN,* then the number of snails in the pen will increase by +3, so +3 is returned. If, the gate swings *OUT,* then snails will be leaving the pen so the pen count will be reduced by 3 and so -3 is returned. We leave it to you to determine the appropriate return value when a gate is locked.

Finally, you are required to override the *toString()* method to exactly match the output shown in the following samples (tip: use copy/paste to avoid spelling/typing mistakes).

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

After you have created your *Gate* class in the *Gate.java* file you should test it using the Junit tests in the supplied Android Studio. Running your JUnit test is simple: just right-click on the file *GateUnitTests.java* and choose Run 'GateUnitTests". These tests will called the *Logic.process()* method in the *Logic.java*, which will instantiate a *Gate* object and invoke some of its methods.

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, including the following:

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