CS 211, ALL SECTIONS

PROJECT 4

DUE SUNDAY, APRIL 14TH AT MIDNIGHT

The objective of this project is to simulate a smart home controller which works with smart switches. The project will make use of generics and enumerations while reinforcing skills involving working with interfaces.

OVERVIEW:

1. Implement the enumeration `SwitchState`, which describes the states which a switch can take (besides on and off, unknown and error states are also allowed).
2. Write the interfaces `DeviceInfo` and `Switch` which will be used to describe smart devices.
3. Implement the class `SmartDevice` which describes a basic, general smart device.
4. Write the generic interface `Listener` which is used to listen for signals from a smart device.
5. Implement the class `SmartSwitch`, which implements our smart switch.
6. Implement the class `Controller`, which manages a number of smart devices.
7. Students in the honors section will have an additional smart device to implement for full credit.
8. Download and use the tester module to ensure that your program is correct.
9. Prepare the assignment for submission and submit it.

The concept of smart devices is decades old, but it hadn't caught on much until recently. Now, however, Internet and cell phone technology has matured to the point that smart devices are common and practical. It's not at all uncommon see homes with smart lights or cameras or thermometers. Yes, there are issues which we will need to tackle in the near future (security comes to mind), but controlling a smart home is a task for the here and now. Interestingly, much of smart home control is done by way of cell phone apps, and many cell phone apps (think Android) are written in Java.

For this project, we will simulate a smart home system which uses smart switches. Like in previous projects, we will build from the ground up by making the smart device classes, and build up to a controller which manages the devices. The smart devices will be capable of signalling: when their state changes, they will send out signals to notify others of state changes. Our controller will be a listener which listens for those state changes and reports them.

We will incorporate two more language features into this project: generics and enumerations. We will use enumerations to describe switch state. If we just wanted on and off, we could use a `boolean`, but by using an enumeration we can permit additional states such as unknown or error states. We will use generics for signal listeners for signals sent by smart devices, because by using generics, we can define different types of listeners for different classes of devices.

RULES

1. This project is an individual effort; the Honor Code applies
2. You may import Java built-in libraries, although avoid doing so unless necessary.
3. You may add your own helper methods or data fields, but they should be private or protected. You may add additional constructors which are public.

SWITCHSTATE TASK:

```
public enum SwitchState
```

(*10pts*) This enumeration will contain the possible values which a switch can take, which will include the obvious `ON` and `OFF` values, but also `UNKNOWN` and `ERROR`. Additionally, the enumeration will provide the following public methods:

- A `value` method which takes no parameters and returns a `Boolean`. This method will return `true` or `false` if the element is `ON` or `OFF`, respectively, but `null` for either of the other values.
- A `toString` method which returns `"on"` for element `ON`, `"off"` for element OFF, `"unknown"` for element `UNKNOWN`, and `"error"` for element `ERROR`.
- A `flip` method which takes no parameters and returns a `SwitchState`. If the element is `ON` or `OFF`, then it returns `OFF` or `ON`, respectively (i.e. it returns the opposite state). If it is `UNKNOWN` or `ERROR` then it returns the same state.

DEVICEINFO TASK:

```
public interface DeviceInfo
```

(*5pts*) This interface will be a part of smart devices to get identifying information about the device. It has two methods, which allow us to get the device name and the device ID. The device ID is a number which will be used by the controller to identify a particular device, especially if more than one device has the same name.

The interface will have the following public methods:

- A `name` method which takes no parameters and returns a `String`.
- An `id` method which takes no parameters and returns an `int`.

Switch Task:
`public interface Switch`, which is a `DeviceInfo`

(*5pts*) This interface will be part of switch devices (light switches, etc). It contains methods related to setting, flipping, or querying the state of a switch. This interface will have the following public methods:

- A `getState` method, which takes no parameters and returns a `SwitchState`.
- A `flip` method, which takes no parameters and returns a `SwitchState`.
- A `change` method, which takes a `SwitchState` parameter but does not return anything.

SmartDevice Task:
`public class SmartDevice`, which is a `DeviceInfo`

(*10pts*) This class will be the basis for any smart device we write. It will not have much functionality for now, only enough to identify the device. We will add onto the functionality later in subclasses of this class. At the minimum, every smart device will have a name and an (integer) ID value to identify it.

Our `SmartDevice` will implement the following public methods:

- A constructor which takes a `String` parameter, which would represent the device name. The device's ID will be zero by default.
- An `id` getter method, which returns an `int`.
- A `setID` setter method, which takes an `int` parameter.
- A `name` getter method, which returns a `String`.
- A `toString` method, which returns a string in the form `"name ID"`. For example:

```
> SmartDevice d = new SmartDevice("device");
> d.setID(5);
> System.out.println(d.toString());
device 5
```

Listener Task:
`public interface Listener`, which is a generic interface with two type parameters

(*10pts*) Our smart devices will send out signals if something happens (i.e. their state changes, etc). In order to react to those signals, our controllers will be listeners which listen for signals. The smart devices do not need to know the details of our controllers to know that they are listeners which can accept signals. Thus, we will use a listener interface which can react to a signal sent out by our smart devices.

This interface is going to be a generic interface. By using generics, we can write re-usable code which is essentially the same except for the data type. We can send signals from different types of smart devices, and we can use it to send different types of signals. Instead of rewriting the interface for every type of device and every class of signals, we use a generic interface which allows us to treat these types as placeholders until we're ready to use them. Our listener will have two type parameters: the first one will represent the smart device category which is generating the signal, and the second will represent the type of signal which is being sent.

The interface will define one method:

- A `signal` method with two parameters, where the first parameter's type is the smart device category, and the second parameter's type is the signal type.

SmartSwitch Task:
`public class SmartSwitch`, which is a `SmartDevice` and a `Switch`

(*30pts*) First and foremost, the `SmartSwitch` device is a `Switch`. That means that it has a `SwitchState` state, meaning that it can be `ON` or `OFF` (or even be in an `UNKNOWN` or `ERROR` state). The state can be changed or flipped (from `ON` to `OFF` and back again), or queried. We can ask for the current state and change the state.

On top of the switching functionality, we will also include signalling functionality, which will send out a signal whenever the state changes. To do this, we will keep a list of `Listener<Switch,SwitchState>`s as a private member, and whenever a state change occurs, we will call the `signal` method for each of these listeners to inform them that the state has changed. When we call signal, the parameters would be `this` smart switch instance plus the new state of the switch. We will have methods to add or remove listeners from the list, so that our controller(s) would know whenever a smart switch's state has changed.

Our smart switch should implement the following:

- A constructor which takes a `String` parameter representing the device's name. Initially, the state of the switch should be `UNKNOWN`

- A `getState` method, which takes no parameters and returns a `SwitchState`. This method should return the current state of the switch.
- A `flip` method, which takes no parameters and returns a `SwitchState`. If the current state of the switch is `ON` or `OFF`, this method would toggle the state (flip it to the opposite one). If the state is something else then it would be unaffected. If calling this method results in the state being changed, then the class should signal all registered listeners by calling their `signal` method as described above.
- A `change` method, which takes a `SwitchState` parameter but does not return anything. This sets the new state of the switch to the passed-in value. If calling this method results in the state being changed, then the class should signal all registered listeners.
- An `addStateListener` method which takes a `Listener<Switch,SwitchState>` parameter but does not return anything. This method will add a new listener to the private list of listeners.
- A `removeStateListener` method which takes a `Listener<Switch,SwitchState>` parameter but does not return anything. This method will remove the listener from the private list of listeners.
- A `toString` method which returns a string in the format `"name ID: state"`. For example:

```
> SmartSwitch ss = new SmartSwitch("light");
> ss.setID(1);
> ss.change(SwitchState.OFF);
> System.out.println(ss.toString());
light 1: off
> ss.flip();
> System.out.println(ss.toString());
light 1: on
```

Tip: it may be useful to have a private `sendSignals` method to call the `signal` method of all registered listeners. This method could be called whenever the smart switch's state changes.

CONTROLLER TASK:

`public class Controller`, which is a `Listener<Switch,SwitchState>`

(*30pts*) Our controller keeps a list of smart devices, and will print reports about the existing smart devices, or allow us to add, remove, or retreive a particular smart device. Also, it serves as a listener, so we can use it to report changes whenever the state of a smart device has changed. The controller assigns each smart device a unique ID (the ID number grows every time a new device is added, so that the first device is device 0, the next is device 1, etc). We do not need to go back and "plug holes" in the ID numbers if devices are removed and others are added later. Here is an example of how the controller might work:

```
> Controller c = new Controller();
> SmartSwitch ss0 = new SmartSwitch("front door light");
> SmartSwitch ss1 = new SmartSwitch("kitchen light");
> SmartSwitch ss2 = new SmartSwitch("pool light");
> c.addDevice(ss0);
> c.addDevice(ss1);
> c.addDevice(ss2);
> c
front door light 0: unknown
kitchen light 1: unknown
pool light 2: unknown

> ss0.change(SwitchState.ON); // a state change will trigger the signal listener, which prints a message
front door light 0 changed state to on
> ss0.change(SwitchState.ON); // if there is no state change, then no signal
> ss1.flip(); // if there is no state change, then there is no signal
> ss1.change(SwitchState.ON);
kitchen light 1 changed state to on
> ss1.flip();
kitchen light 1 changed state to off
> ss2.change(SwitchState.OFF);
pool light 2 changed state to off
> c
front door light 0: on
kitchen light 1: off
pool light 2: off
```

This class will have the following public methods:

- A default constructor.
- An `addDevice` method which takes a `SmartDevice` parameter and returns an `int`. The method adds the smart device to a private list of devices and assigns it a unique ID number (this number should increment every time, so that the first device which is added gets ID 0, the second gets ID 1, and so on). Additionally, if the device is a `SmartSwitch`, then the controller should register itself as a listener of the device. This method should return the ID number which was assigned to the device.
- A `getDevice` method which takes an `int` parameter and returns a `SmartDevice` object. This method uses the ID number given as input and searches the private list of devices for a device which has that ID value, and returns the first one which is found. If there are none, then this method returns `null`
- A `getDeviceN` method which takes an `int` parameter and returns a `SmartDevice` object. Instead of searching by ID number, this method simply returns the `n`th smart device from the private list.
- A `numDevices` method which takes no parameters and returns an `int`. This method returns the total number of devices stored in this controller.
- A `removeDevice` method which takes a `SmartDevice` parameter and returns a `boolean`. This method would remove the given device from the private list of devices, returning true if it was removed and false if it was not in the list. Additionally, if the device is a `SmartSwitch`, the controller should remove itself as a listener of the device.
- A second `removeDevice` method which takes an `int` paramter. This version should behave identically to the previous one, except that the parameter refers to the ID number of the device rather than the device object.
- A `signal` method which takes a `Switch` parameter and a `SwitchState` parameter, and does not return anything. This method would be called whenever one of the smart switches signals a state change. Whenever it is called, it should print a message of the form `"NAME ID changed state to STATE"` (see above for an example).
- A `toString` method which returns the `toString` values of all stored smart devices, separated by newlines.

Honors Section:

If you are in the honors section, you must complete this part and it is worth 20 points of the project grade. If you are not in the honors section, you are welcome to attempt this but you do not need to complete it and it is not worth any points if you do.

`public interface Meter`

`public class SmartMeter` which is a `SmartDevice` and a `Meter`

The `Meter` interface has one method, `value`, which is a getter method returning an `int`. The `SmartMeter` is similar to a `SmartSwitch`, but it has an additional feature: it has an additional set of listeners for triggers. Whenever the meter reaches or goes above a certain value, it will signal a trigger. For example, if we set a trigger listener with a trigger threshold of 10, and our meter value changes from 6 to 20, then we would signal that particular trigger listener, passing it the value 10 (the value of the threshold). This would be used to set controls like "if the meter reading ever reaches at least 10, let me know."

The public methods in the `SmartMeter` class will be as follows:

- A constructor which takes a `String` parameter representing the device's name and an `int` parameter representing the meter value.
- A `value` getter method, which takes no parameters and returns an `Integer`. This method gets the current meter value.
- A `change` method, which takes an `int` parameter but does not return anything. This sets the new state of the meter to the passed-in value. If calling this method results in the value being changed, then the class should signal all registered state listeners (the second argument to the `signal` method would be the current value of the meter) plus any necessary trigger listeners (the second arguement to the `signal` method would be the value of the trigger threshold). The trigger listener is called whenever the value changes plus it is greater than or equal to the trigger value of a particular trigger listener.
- An `addStateListener` method which takes a `Listener<Meter,Integer>` parameter but does not return anything. This method will add a new listener to the private list of state listeners.
- A `removeStateListener` method which takes a `Listener<Meter,Integer>` parameter but does not return anything. This method will remove the listener from the private list of state listeners.
- An `addTriggerListener` method which takes a `Listener<Meter,Integer>` parameter and an `Integer` parameter but does not return anything. This method will add a new listener to the private list of trigger listeners, and assign it a trigger threshold based on the integer input parameter.
- A `removeTriggerListener` method which takes a `Listener<Meter,Integer>` parameter but does not return anything. This method will remove the listener from the private list of trigger listeners.
- A `toString` method which returns a string in the format `"name ID: value"`. For example:

```
> SmartMeter sm = new SmartMeter("thermostat");
> sm.setID(1);
> sm.change(5);
> System.out.println(sm.toString());
thermostat 1: 5
```

Finally, the `Controller` class should be updated so that it implements `Listener<Meter,Integer>`. Whenever a `SmartMeter` device is added to the controller, the controller should register itself as a listener of the device. In the controller's corresponding `signal` method (which is a different method from the earlier one because the data type of the `Listener` is different), it should print a message of the following form whenever a signal is received: `"NAME ID changed value to VALUE"`

Testing:

- https://mason.gmu.edu/~iavramo2/classes/cs211/junit-cs211.jar
- https://mason.gmu.edu/~iavramo2/classes/cs211/s19/P4tester.java

SUBMISSION:

Submission instructions are as follows.

1. Let *xxx* be your lab section number, and let *yyyyyyyy* be your GMU userid. Create the directory `xxx_yyyyyyyy_P4/`
2. Place your files in the directory you've just created.
3. Create the file `ID.txt` in the format shown below, containing your name, userid, G#, lecture section and lab section, and add it to the directory.

   Full Name: Donald Knuth
   userID: dknuth
   G#: 00123456
   Lecture section: 004
   Lab section: 213

4. compress the folder and its contents into a .zip file, and upload the file to Blackboard.