Assignment	Apply the Observer Pattern to a simple Counter object. Starter code is at: `https://classroom.github.com/a/ORLWyv8l` Short URL: **`https://goo.gl/obaqHh`**
What to Submit	Submit the completed project.

Introduction: Dependency Injection for Shared References

In software, objects usually *depend on* or *use* other objects. How does object A get a reference to some object S that A depends on? The S object may be needed by many other objects, or it may require some attributes to be set (its state), so we can't simple create a new S object inside of A.

A solution to this is *give* A a reference to the S object that A should use. This is called ***dependency injection***.

Here are 2 basic ways to perform dependency injection. In this example, GuessingGame is the S object (dependency) we want to share.

1. ***Consturctor Injection*** - "set" a reference to GuessingGame using a constructor of A:

```
public class A {
    private GuessingGame game;
    // Constructor receives a reference to the game
    public A(GuessingGame game) {
        this.game = game;
    }
    // the code for A can use the game
}
```

2. ***Setter Injection*** - The B class provides a setGuessingGame( ) method to set a reference to the game.

```
public class B {
    private GuessingGame game = null;
    public B( ) {
        // no guessing game yet
    }
    public void setGuessingGame(GuessingGame game) {
        this.game = game;
    }
    // the code for B can also use the game
```

JavaFX also uses *dependency injection,* but it does it by setting attributes directly (even private ones). In a JavaFX controller class you wrote code like this:

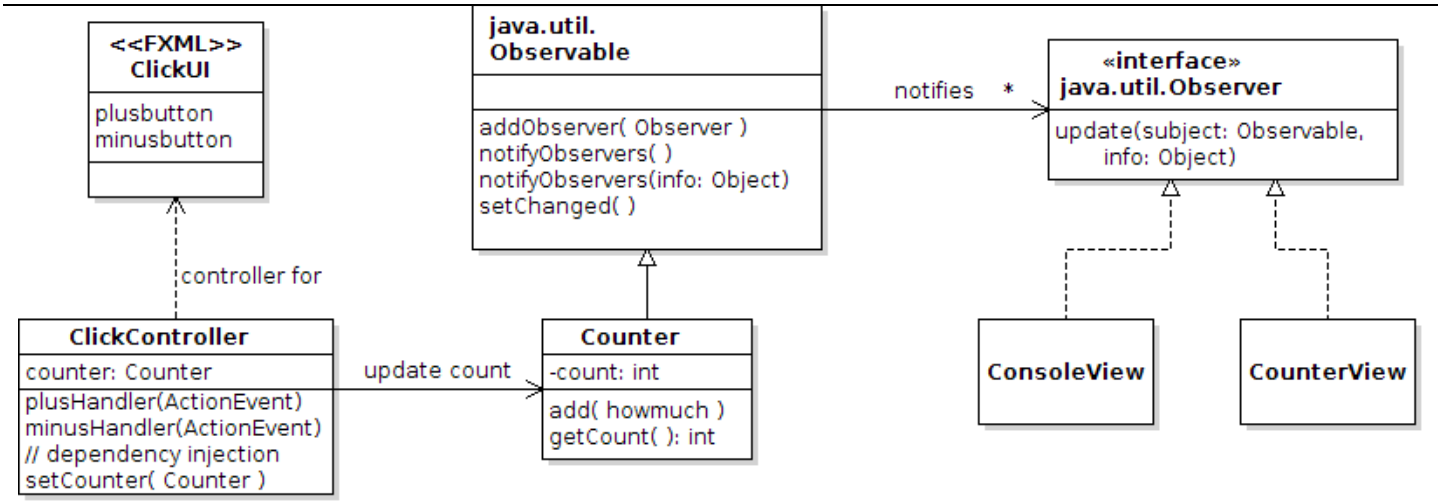
```
public class ConverterController {
    @FXML
    private TextField inputfield;
}
```

The `FXMLLoader` class creates the TextField object from an FXML file, and then *injects* a reference to the `inputfield` variable in the Controller. We don't have to write any "set" method!

You will use *dependency injection* to *set* a reference to a subject (Counter) into your UI controller objects.

Problem 1: Click Counter with Observers

Complete the ClickCounter application (starter code provided).



1. Make the **`Counter`** class be an Observable and notify observers when the counter changes.

```
package counter;
public class Counter extends java.util.Observable {
    private int count = 0;

    public int getCount() { return count; }

    public void add(int howmuch) {
        count += howmuch;
        //TODO notify observers.  See UML diagram for methods.
        // First you must invoke setChanged() .
    }
}
```

2. Complete the simple **`ConsoleView`** that prints the click count on the console whenever Counter notifies observers.

```
package counter;
public class ConsoleView implements java.util.Observer {
    private Counter counter;
    /**
     * A ConsoleView with reference to a counter (the subject).
     * @param counter the counter to display.
     */
    public ConsoleView(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void update(Observable subject, Object info) {
        //TODO If the info parameter is not null, then print it.
        //TODO Then print the current value of the counter,
        //     using the format "Count: 17".
    }
}
```

3. Write an **`ObserverTest`** class to test that the observer code works. Every time you enter a number, it should display the updated total count on the console.

This shows how to add an Observer to an Observable object.

```
public class ObserverTest {
    public static void main(String[] args) {
        final Scanner console = new Scanner(System.in);
        Counter counter = new Counter();  // The observable subject
        ConsoleView view = new ConsoleView(counter);
        counter.addObserver(view);

        while(true) {
            System.out.print("Count how many? ");
            int howmany = console.nextInt();
            if (howmany==0) System.exit(0);
            counter.add(howmany);
        }
    }
```

4. The ClickController has event handler methods for the + and - buttons, but they don't do anything. We want them to update the counter. So ClickController needs a *reference* to the counter.

4.1 In ClickController, add a **`counter`** attribute and a **`setCounter(Counter counter)`** method.

4.2 Modify the button handler methods to add +1 or -1 to the counter.

5. The **`Main`** class needs to create the Counter object and call clickContoller.setCounter(counter). To do this, we need a reference to the ClickController. Notice that we use the FXMLLoader to get a reference to the controller class. This is different from the code you wrote last week and the Eclipse auto-generated code. The starter code contains complete code. The interesting part is shown here:

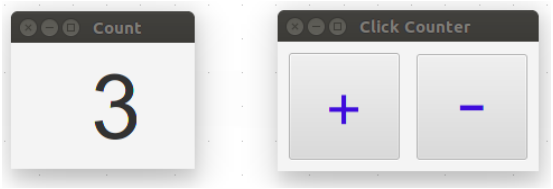
```
@Override
public void start(Stage primaryStage) {
    // Create the Counter object (the "model" part of our app)
    Counter counter = new Counter();

    URL url = getClass().getResource("ClickUI.fxml");
    if (url == null) {
           System.out.println("Couldn't find file: ClickUI.fxml");
           Platform.exit();
    }
    // Load the FXML file and get reference to the loader.
    FXMLLoader loader = new FXMLLoader(url);
    // Create the UI. This will instantiate the controller object, too.
    Parent root = loader.load();
    // Now we can get the controller object from the FXMLloader.
    // This is interesting -- we don't need to use a cast!
    ClickController controller = loader.getController();

    // Dependency Injection:
    // Set the Counter object we want the view to update.
    controller.setCounter(counter);

    // Build and show the scene
    Scene scene = new Scene(root);
     primaryStage.setScene(scene);
     primaryStage.show();
```

```
    //TODO Create a ConsoleView object and add it as Observer
    //     of the Counter.
```

6. In the Main class's start( ) method, create a **ConsoleView** and add it as observer of the Counter.

7. Run the Main class. When you click "+" or "-", it should show the count on the console.

8. Create a graphical display of the counter, named **CounterView**. It should show the counter value, and automatically update itself.



This view only contains one component, so we will write the UI in Java code instead of FXML. The starter code is in CounterView.java.

a) The initComponents() method creates all components and scene, and the run() method displays it.

b) It uses *Constructor Injection* to set a reference to the counter.

It is good practice to use separate methods for separate tasks. Don't create lots of components in the constructor, nor show the view from the constructor (which lots of tutorial code does).

TODO: Complete all the TODO items in code.

```
public class CounterView implements java.util.Observer {
    /** the Window to show scene on */
    private Stage stage;
    /** a counter to show value of */
    private Counter counter;
    /** the label that shows the counter value. */
    private Label label;
    /**
     * Initialize a CounterView, which shows value of a counter.
     * @param counter the Counter to show.
     */
    public CounterView(Counter counter) {
        this.counter = counter;
        initComponents();
    }

    private void initComponents() {
        stage = new Stage();
        // components and containers for our window
        HBox root = new HBox();
        //TODO Add some padding around the HBox (setPadding)
        root.setPadding( ? );
        //TODO Align components in the CENTER of HBox (setAlignment)
        root.setAlignment( ? );
        // The label that will show the counter value.
        label = new Label("   ");
        // set preferred width (setPrefWidth) only if necessary
        //label.setPrefWidth(144);
        //TODO Make the text BIG. Use setFont to set the font & size.
        //Be careful to import the correct Font class (not java.awt.Font).
        label.setFont( ? );
        //TODO Set the text alignment to CENTER (setAlignment)
```

```
        label.setAlignment(  ?   );
        // Add the label to the HBox.  You can all more components, too.
        root.getChildren().add(label);
        // Create a Scene using HBox as the root element
        Scene scene = new Scene(root);
        // show the scene on the stage
        stage.setScene(scene);
        stage.setTitle("Count");
        stage.sizeToScene();
    }

    /** Show the window and update the counter value. */
    public void run() {
        stage.show();
        //TODO show current counter value in the label
    }
    //TODO Write the Observer method to show counter value on the label
```

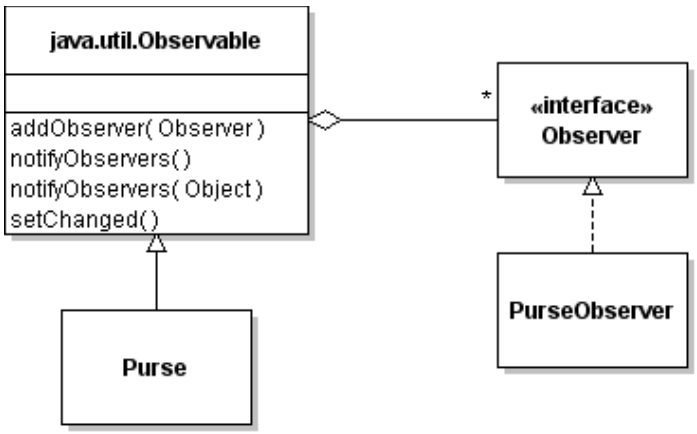
9. Now that you have **`CounterView`**, you need to create a **`CounterView`** object in **`Main`** and show it. In the start() method add code:

```
    CounterView view = new CounterView(counter);
    counter.addObserver(view);
    view.run();
```

10. Run the application. When you press + or -, the count should be updated in the view *and* on the console.

Create a Stage or Be a Stage?

The **`CounterView`** creates a Stage object to contain the Scene graph (the view). Another design is for CounterView to *extend* Stage. The benefit of extending Stage is that the Main class can modify the window size, position, etc., or close the window of CounterView. If Stage is an attribute of CounterView, then its encapsulated. If you make CounterView a subclass of Stage, then *don't* create a Stage object! It is the stage already. If you want to try with it without much recoding, write "stage = this" in initComponents().



Problem 2: Add Graphical Observers to Coin Purse

Modify the Coin Purse to support observers and create at least 2 graphical observers.

There are 2 ways to do this.
1. Use java.util.Observable and java.util.Observer, as in the previous problem.
2. JavaFX has *many* Observable types. One of them is ObservableList, which extends java.util.List. If you use ObservableList for the objects in the Purse, then you will have an easy way to create interesting views of the Purse, without injecting a Purse reference into each view.

2.1 *Purse Balance Observer:* Write an observer to display the balance in the Purse.

2.2 *Purse Status Observer*: Write an observer that displays "FULL", "EMPTY", or number of objects in the Purse (when not full or empty).

2.3 Modify the Main class to create the observers and add Observers to the Purse. Main should create a Purse and Observers, and add observers to the Purse.