SYSC 2004
Winter 2018
Lab 10

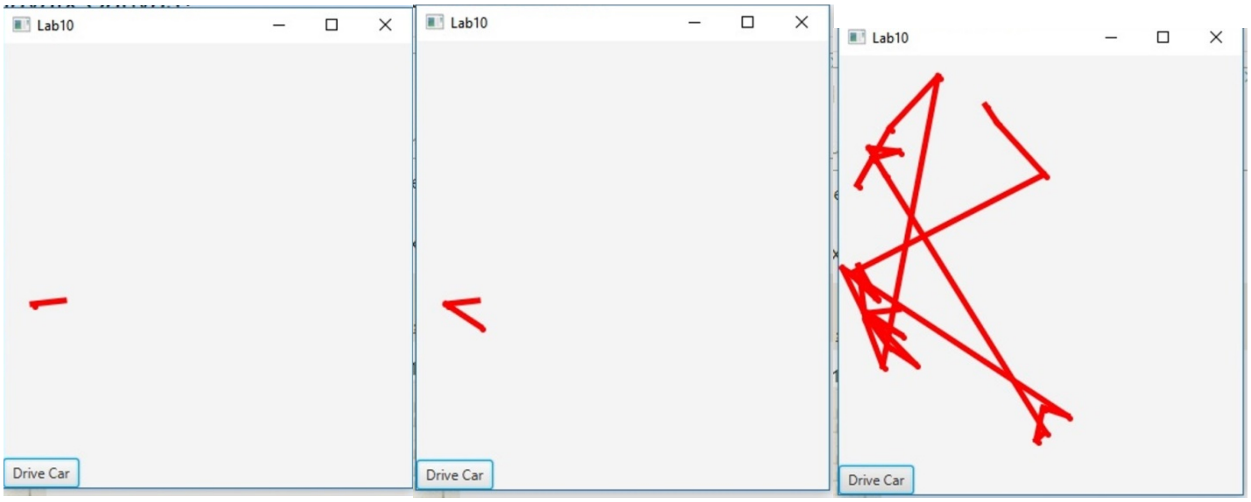
Objectives

- Continued Exercises in GUI, using the Canvas
- Use of Inner Classes
- Use of the Observer Pattern

Submission Requirements: (Exact names are required by the submit program)

- **Without a submission you will not get a mark:** Lab10.java

You are to create the most rudimentary car-tracking interface possible.



Canvas object in the CENTER

Button in the BOTTOM

The button will allow the simulation of a car moving. Each time a user clicks on the button, the car's position will be randomly updated, and its new position updated on the display as a rectangle and a line connecting the new position to the previous position.

The Scene is organized as a `BorderPane` (https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/BorderPane.html). Please go to this URL to understand this kind of layout. In the CENTER of the `BorderPane` object is a `Canvas` object; in the BOTTOM of the `BorderPane` object is a `Button` object. A `Canvas` is different from most other JavaFX Controls because it allows free-form drawing – points, lines, circles, rectangles and so on.

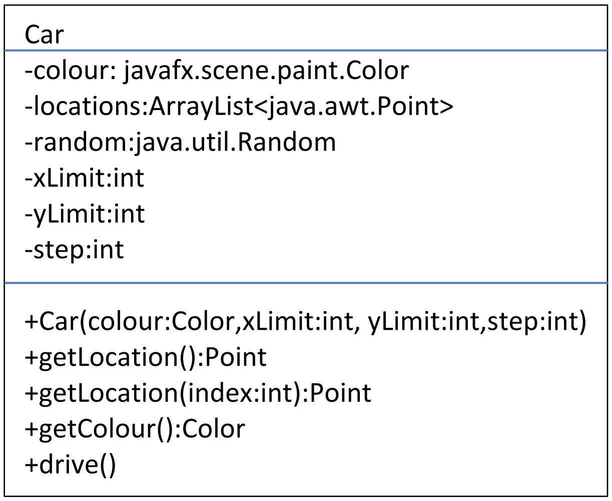
Part A – Preparing the Background Model

Behind the GUI is the actual object that is being visually represented on the GUI; in our case, a car. We will use the `Car` class to practice some old topics, including random number generation as well as defensive safe copies.

A `Car` object has two primary attributes: its `colour` and a history of all of its `locations`. A location is represented as an (x,y) coordinate, in turn, to be coded using the `Point` class. Ultimately, these (x,y) coordinates will be plotted on our GUI canvas and hence must stay within the size of this canvas. For this reason, the constructor accepts two arguments `xLimit` and `yLimit` that will be the maximum values of each dimension; zero is the minimum value.

A `Car` object moves about randomly. On each invocation of the `drive`() method, the `Car` object will move some random distance – in both the x and y direction - from its current location. In mathematics, we call this a translation. Again, ultimately, the new location must be within the bounds of the canvas, so the steps should be a fraction of the overall size of the canvas. For this reason, the constructor also accepts an argument `step` that will be the absolute maximum value of a translation. Translations can be either positive or negative (in both the x and y dimension).

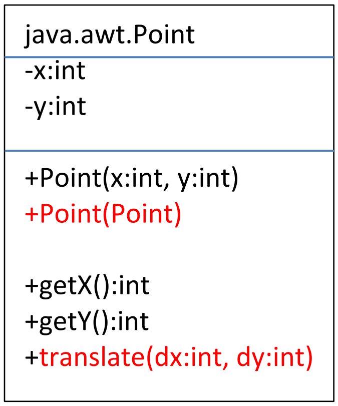
The `Car` object stores a record of all of its `locations`. Each new location is to be added to the front of the list; i.e. at index 0 of the list. Because the list makes use of the `ArrayList` class, adding a new location at index 0, automatically shuffles the previous location to index 1, and so on.



Car

-colour: javafx.scene.paint.Color
-locations:ArrayList<java.awt.Point>
-random:java.util.Random
-xLimit:int
-yLimit:int
-step:int

+Car(colour:Color,xLimit:int, yLimit:int,step:int)
+getLocation():Point
+getLocation(index:int):Point
+getColour():Color
+drive()



java.awt.Point

-x:int
-y:int

+Point(x:int, y:int)
+Point(Point)

+getX():int
+getY():int
+translate(dx:int, dy:int)

Detailed Instructions

1. Create a new project called **lab10**. Create the project as a JavaFX application (not as a Java application). For now, ignore the code that is automatically generated for you.
2. Create a new Java class called `Car`.
3. Declare all the instance variables.
4. Write the constructor to now initialize all these instance variables
    - An empty list of locations must be constructed.
    - A random number generator must be constructed.
    - An initial first location must be randomly generated
        - Generate a random number between 0 and `xLimit`, for the x coordinate

          Recall: `int value = random.nextInt(max);`

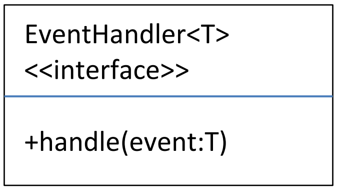
        - Generate a random number between 0 and `yLimit`, for the y coordinate
        - Create a `Point` object, using the x and y coordinates
        - Add the `Point` object to the list of `locations` (by default, at index 0).
5. Write the accessors
    - The no-argument version of `getLocation`() should return the current location (always at index 0)
    - The other version of `getLocation`() should return the (previous) location at the given index. If the index is not invalid, the method should throw an IndexOutOfBoundsException.
    - Both `getLocation`(…) methods must return defensive safe copies of the location to be returned. Notice that the `Point` class has a copy constructor that will be useful to you.
6. Write the `drive`() method
    - The purpose of this method is to translate the car from its current location (stored at index 0) by a random xdelta and a random ydelta. Its new location must be added to the stored list, so that it is now at index 0.
    - Movement can be in both directions, so you must generate a random number for the x direction, between –step and +step.
        - To generate a number between –max and max:

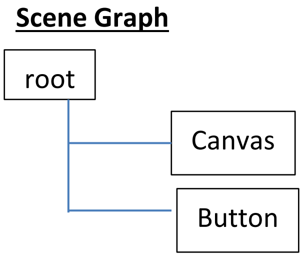
          ```
          int num = random.nextInt(2*max) – max;     //
          ```

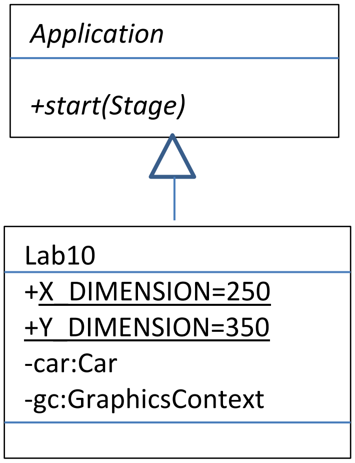
    - Generate a random number for the y direction, between –step and +step.
    - Construct a duplicate `Point` object of the current location. Use `Point’s` copy constructor.
    - Translate the duplicate `Point` object. Use `Point’s translate`() method.
    - Check that neither of the (x,y) coordinates exceed the limits (0 to x/yLimit). If any of them are out of bounds, simply throw this `Point` object away and construct a brand new random location, within the limits.
    - Add the new location to the stored list of locations, at index 0.
7. Test your implementation using the given JUnit test for the `Car` class.

Part B – Preparing the GUI

It is now time to build the GUI







Detailed Instructions

1. Go back to the **Lab10.java** application that was created for you when you created the project. You should already have code starting with:

   ```
   public class Lab10 extends Application { }
   ```

2. Declare the variables, shown in the UML diagram.
3. Within the start() method:
   - Instantiate the `car` object, passing in the static constants as the `xLimit` and `yLimit` arguments. Use `step` = 50.
   - Instantiate a `Canvas` object, passing in the static constants as the size of the canvas.

     ```
     Canvas canvas = new Canvas(X_DIMENSION, Y_DIMENSION);
     ```

   - Initialize the instance variable `gc` as follows:

     ```
     gc = canvas.getGraphicsContext2D();
     ```

     This variable is key to drawing shapes and lines on the canvas, later.
   - Instantiate the `root` of the given **Scene Graph** as an instance of the `BorderPane` class.

     ```
     BorderPane root = new BorderPane();
     ```

   - Add your `Canvas` object to the centre of the `root`.
     ```
     root.setCenter( canvas );
     ```
   - Instantiate the Button object (with no help from me) and then add it to the bottom of the root.

     ```
     root.setBottom( button );
     ```

- **Using an anonymous inner class**, add an event handler to the button that – for now – simply prints out a message saying that it has been clicked. Use the lecture notes to help you.

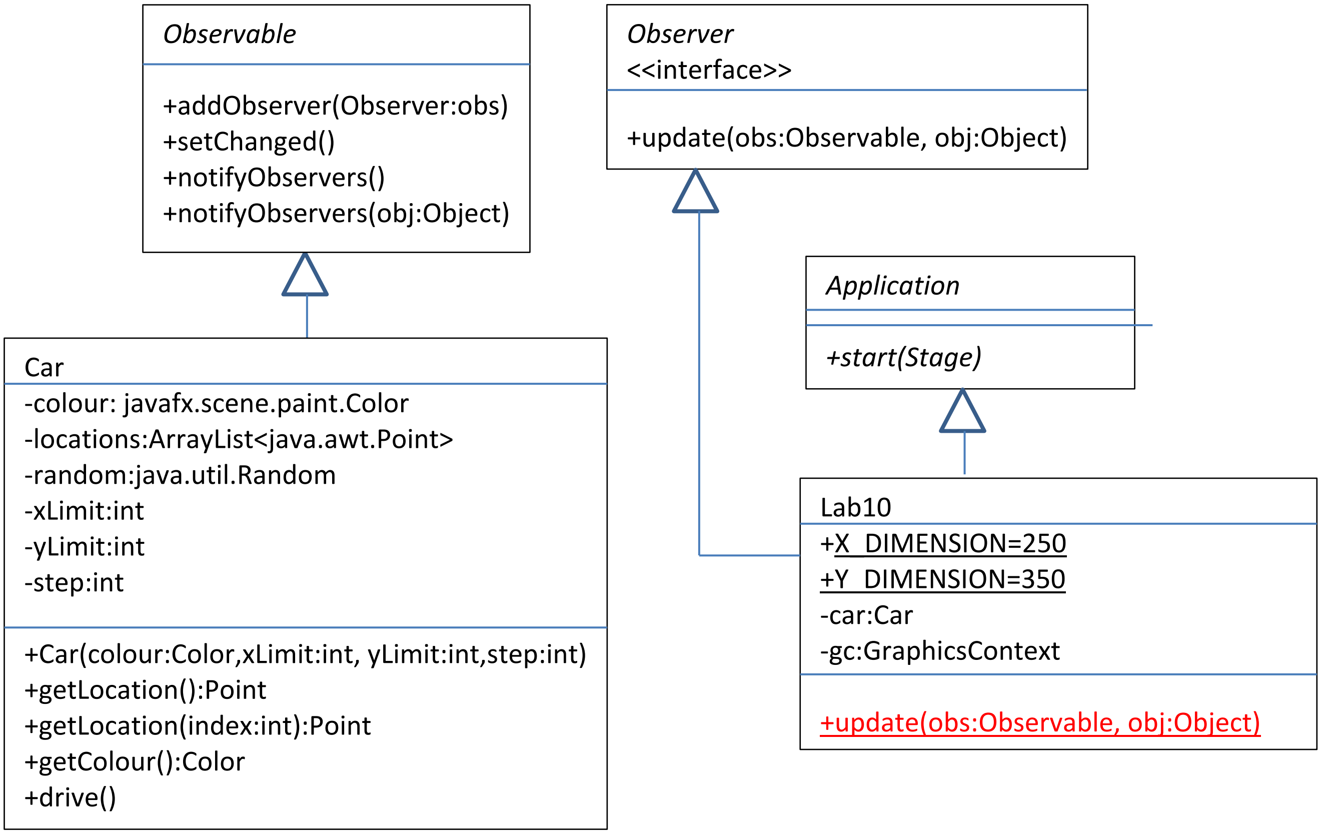
```
button.setOnAction ( …. );
```

- Change the code that instantiates the `Scene` object, so that it is initialized with a width = X_DIMENSION and a height = Y_DIMENSION+30.

4. Run your program. At this point, you should have a blank canvas and a button. When you click on the button, it should print a message. If you don't see the button right away, try re-sizing your window with your mouse.

Part C – Completing the GUI using the Observer Pattern

It is time to complete the behaviour of the application. When a user clicks on the button, the car's location should be updated and the new location be displayed on the canvas. To break this down further: the `eventHandler` of the `button` should invoke the `drive`() method of the `car` (which will update its own location). The missing link is then how to trigger an update to the `canvas`, when the `car` changes its `location`. The Observer Pattern shall be used.


Observable
+addObserver(Observer:obs)
+setChanged()
+notifyObservers()
+notifyObservers(obj:Object)
Observer
<<interface>>
+update(obs:Observable, obj:Object)
Application
+start(Stage)
Car
-colour: javafx.scene.paint.Color
-locations:ArrayList<java.awt.Point>
-random:java.util.Random
-xLimit:int
-yLimit:int
-step:int
+Car(colour:Color,xLimit:int, yLimit:int,step:int)
+getLocation():Point
+getLocation(index:int):Point
+getColour():Color
+drive()
Lab10
+X_DIMENSION=250
+Y_DIMENSION=350
-car:Car
-gc:GraphicsContext
+update(obs:Observable, obj:Object)

Detailed Instructions

1. In the `Car` class:
    - Have the `Car` class extend `Observable`.
    - Within the `drive`() method, after all the other code:
        - Invoke the `setChanged`() method, to flag the change in location
        - Invoke the `notifyObserverss`() method, to propagate the change to any registered observers.
2. In the Lab10 class
    - Have the **Lab10** class implement `Observer` (as well as extend `Application`!)
    - You will now have to implement the `update`(…) method.
        - Downcast the `Observerable` argument called `o` to a `Car` object
        - Get the colour of the `Car` object.
        - Get the current location of the `Car` object
        - Draw an oval on the canvas, at the current location

          ```
          gc.setFill (colour);
          gc.fillOval(x,y, 8,8);
          ```

        - Get the previous location of the `Car` object (if any)
        - Draw a line between the current location and the previous location.

          ```
          gc.setLineWidth(5);
          gc.setStroke (colour);
          gc.strokeLine( x1, y1, x2, y2);
          ```

    - Your `update`() method will never be invoked unless this object is subscribed. Within your `start`() method,

      ```
              car.addObserver(this);
      ```
3. Have fun with your program.