Assignment 5
COMP 86

In this assignment we will add animation using time ticks, some more interactive controls, and better layout to the simulation/game of Assignments 2, 3, and 4.

Time Ticks

Each vehicle already has a location and a way to draw itself in its current location. Add some instance variables for how it moves (speed, direction, how long since last move, whatever you need). Then, your program simply needs to update each Vehicle's location (using your information about speed, direction, etc.) on every clock tick (e.g., every second) and then redraw them all.

A good way to handle this kind of simulation is to think of each Vehicle as if it were an independent module. To advance the simulation, your program will call a tick() method for each Vehicle, at regular intervals. The tick method should simply compute a Vehicle's new position, based on its previous position and on its velocity and direction. Different subclasses of Vehicle can move differently, by overriding tick.

We will use a Timer object to keep the simulation running. It will send actionPerformed events to our process, and we will provide a callback routine to be called each time the callback is received. Your callback routine updates the Vehicle locations (by calling tick on each of the Vehicles) and then triggers a repaint of the window.

Controls

Controls will still apply to the simulation as a whole or to a category of vehicles, rather than to an individual Vehicle on the screen.

- Provide a command to stop or start animation of all vehicles or of a particular subclass of vehicles or to change their speed. (The overall animation should start running automatically when your program starts.)
- Provide a command to add a new Vehicle, with a way to input its starting position and other parameters. You could provide a scrolling list or a set of radio buttons for the user to choose the Vehicle subclass, and appropriate widgets for entering the other parameters.
- Include some controls that are only enabled under certain conditions. For example, perhaps changing speed doesn't make sense if the animation is halted. One good way to do this to use a method called setEnabled(boolean) that is included with all java widgets. For example, YOUR_BUTTON.setEnabled(false) will gray out a button and not allow the user to click it; and YOUR_BUTTON.setEnabled(true) will reactivate the button.

Layout

Make sure your window layout will still look reasonable and be usable when the user resizes the window across a range of sizes. Optionally, try to provide a more responsive window layout, so that your program can be used in any size from full screen desktop down to the size of a smartphone. If necessary, your code can rearrange the window layout explicitly when the window becomes very small or large, perhaps removing some optional features, moving them to pop up dialogue boxes, or using a tabbed dialogue (`CardLayout` in Java). You can also use the **Layout2** example for reference.

Program Design and Practices

(The rest of this still applies from previous assignments)

Your program design should exploit the features of object-oriented programming (encapsulation of code and data, support for abstract data types, polymorphism/overloading, inheritance).In particular, object-oriented programming provides us a good way to handle the various data needed in callback routines. You should use objects to encapsulate each interactive widget with the routines and data you need to use it or pointers to other objects containing such.

You should provide an object for each interactive widget or small group of widgets you create. That object should hold anything you need to remember about the widget from one callback to another, all the data pertinent only to the command for that widget or that you need to operate this control, (including, in most cases, a pointer to the model or other outside object needed to perform the actual action the user requested), and the widget's own listener callback routines.

If you have several widgets that share some behavior or properties, you should organize your objects into an appropriate inheritance hierarchy.

You will have other data that must be accessed by several widgets, particularly shared information about the state of the program or global information about the state of the user interface. Provide additional classes and objects for holding this kind of information.

Remember to trigger your drawing to repaint itself explicitly whenever one of your commands causes a change that should be reflected on the screen. And remember that the way to change the screen is first to change the data stored your classes and then to trigger the repaint.

You should follow these general Java programming practices:

- Make all instance variables of your classes **protected** or **private**. If you need access outside of the class, provide it with *set* and/or *get* methods; don't access protected the variables from outside the class (even though Java -- unfortunately -- allows us to).
- Avoid most global variables or widely-accessible public variables; pass the data you need explicitly.
- Put each class in a separate .java file.

Design Documentation

In addition to your program, submit documentation about the design of your system in these forms:

- An outline showing the inheritance hierarchy
- An outline showing the aggregation hierarchy (which objects contain or "own" which other objects)
- **New:** A list showing "uses" or collaboration relationships (which objects use which other objects to perform functions)
- **New:** The information hiding "secrets" of each of your classes (i.e., what design decisions are entirely encapsulated within that class).

Submit this documentation electronically in text form, along with brief instructions for how to compile and run your program. Include it as part of the `readme` file that you submit with your assignment.