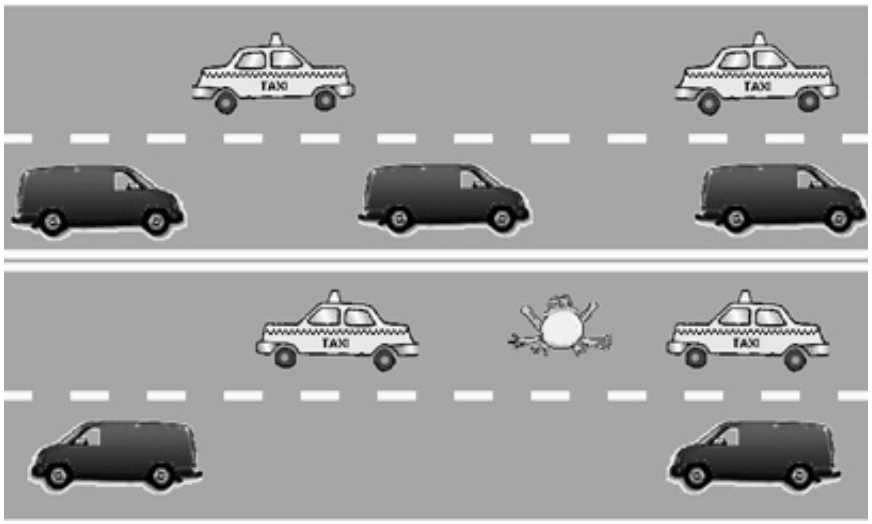
CS 134 Programming Exercise 5:
It's not Easy Being Green

Objective: To gain experience with loops and active objects.

The Scenario

For this lab, we would like you to write a program that plays the game Frogger. In this game, you control a frog that is trying to cross a busy 4-lane highway. Each lane has cars or trucks zooming by. The vehicles in a given lane all travel at the same speed, but vehicles in different lanes may travel at different speeds (and even in different directions if you would like). The user is in control of a frog. Clicking in front of the frog moves it forward one hop (one hop is the width of a lane of traffic), clicking behind moves it back, and similarly for clicking to the left and right of it. The goal is for the user to get the frog across the highway without it getting squished.

If the frog does get squished it should display an "OUCH!" message at the bottom of the screen. The user should be able to restart the frog from its original starting position by clicking the mouse once in the area below the lanes of the highway.



A demo version of this program can be found in the online version of this handout. You can play with it to see what we have in mind.

Design

A design for all of the classes used in this program (`Frogger`, `Frog`, `Vehicle`, and `Lane`) is due at the beginning of lab. While we expect a design for *all* aspects of the program, we urge you to write and debug the program as suggested in the **Getting Started** section later in this handout.

Preparing for this lab

Read this entire handout before doing anything else. In this handout we will begin by describing the classes you need to implement. After we describe the classes, we will outline one plan for proceeding with your implementation.

The Objects

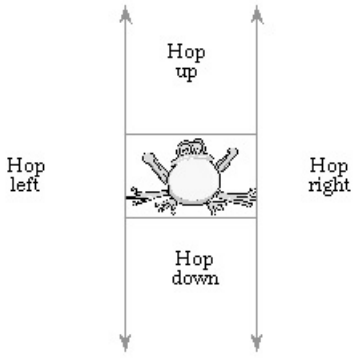
There are three important classes of objects involved in the Frogger game. There is the frog, there are the lanes of traffic (four of them), and there are the vehicles that go on the road. (There are also some graphical objects on the screen to represent the lane markings on the highway, but we won't discuss these in detail as they are easily constructed. In fact, we have provided in the starter folder the code that draws the highway markings.)

The Frog

The frog is a very important object. The frog will be displayed on the screen by creating a `VisibleImage` using an image file we will provide. The `Frog` class will need an instance variable to keep track of this `VisibleImage`.

The constructor for a `Frog` should create the `VisibleImage` and place it just below the lowest lane of the highway, approximately halfway from each end. Your constructor will require several parameters: the `Image` of the frog, the location where the frog should start, how wide a lane is (so the frog knows how far to hop), and the canvas.

The frog clearly needs to be able to hop in each of the four directions in response to a user click. We suggest writing a single method `hopToward` that takes a Location parameter (e.g., point). Depending on which side of the frog the point is on, the frog should move in the appropriate direction. To keep the testing required to determine how the frog should jump simple, we suggest you divide the space around the frog as shown in the diagram below.



Unfortunately, the other thing that happens to a frog is that it gets splattered on the road. A vehicle will be responsible for determining whether it has killed a frog. To do so, it will need to ask the frog if any part of the Frog's body overlaps the `VisibleImage` displayed to represent the vehicle. To make this possible, your `Frog` class should include the definition of an `overlaps` method that takes a `VisibleImage` as a parameter and returns a boolean.

If a vehicle hits the frog, it kills the frog by calling a `kill` method on the frog. This causes an "OUCH!" message to appear at the bottom of the screen.

Finally, through the miracles of medicine, we'd like our frog to be able to come back to life. Add a method `reincarnate` which moves the frog back to its starting point and `show`s the image. Of course, you should not reincarnate a frog unless it is dead. So, include a boolean instance variable that keeps track of the condition of the frog (alive or dead), and check this variable in `reincarnate`. As described above, the user will reincarnate the frog by clicking the mouse below the highway. The `Frog` class should include an `isAlive` accessor method that returns a boolean to enable the `onMousePress` method to determine whether the frog should hop or possibly be reincarnated.

The Vehicles

The constructor for a vehicle will need parameters specifying where the vehicle should be located (two `double`s or a single `Location` parameter), the `Image` used to display the vehicle on the screen, the velocity with which the vehicle should move, and the distance the vehicle should travel before disappearing. In order to check if the vehicle runs over the frog, you also need to pass the frog as a parameter to the `Vehicle` constructor.

`Vehicle` extends `ActiveObject`. This means you must define the method `public void run(){...}`. The while loop inside `run` should:

1. Save the current time. (Get the current time by calling `System.currentTimeMillis()`).
2. `pause` for (at least) 30 milliseconds.
3. Determine how long it actually paused for (e.g., subtract the time saved earlier from the current time) and move the appropriate distance. (Remember your equation from physics: `rate * time = distance`.) You need to do this to ensure smooth motion of your vehicles. With so many active objects, the computer cannot ensure that the length of pauses will be very precise.
4. Find out if the vehicle squished the frog and kill the frog with the `kill` method if it did.

You may assume that all vehicle velocities will be positive. That is, your highway may be a one-way street. You are free (for extra credit), however, to add the ability to handle vehicles with negative velocities so that you can have some lanes where traffic goes from left to right and others where traffic moves from right to left.

The Lane

The purpose of an object of the `Lane` class is to constantly generate the vehicles that travel in a particular lane on your highway. As such, the lane does not actually correspond to any drawing on the screen. Instead, a `Lane` will be an `ActiveObject` that creates `Vehicle`s.

The constructor for a `Lane` is quite simple. The lane was already drawn as part of the background in our window controller so we do not need to update the display. One thing we know is that all the traffic in a lane should drive at the same speed so that cars do not run into each other. The `Frogger` class should pick a random speed for each of the lanes and pass it in to the `Lane` constructor. We have found speeds in the range .033 to .133 pixels/millisecond to be good (though you may want to start with a slower set of speeds while you debug your program).

A lane's main responsibility is to periodically place a new car on the screen. It will do this inside the `run` method. In the while loop of the run method, the lane should generate a car, wait a while to allow a gap between cars, generate another car, and so on.

As you recall from above, the `Vehicle` constructor requires a lot of parameters: its starting location, its image, etc. So far all the lane knows is how fast the cars drive. Where will `Lane` get this other information? The car should be located at one end of the lane initially and should drive until it reaches the other end. If the lane knew where it was located, it could pass this information on to the car. Its location is relative to the entire highway. Our window controller can provide this information to the lane when it constructs it so that the lane can pass the information on to the vehicle.

What about the image? It turns out that the image can only be loaded from the window controller, so this information must also be passed down to the `Vehicle` by passing it to the `Lane` constructor. The `Lane` constructor remembers the image of the cars for its lane and passes this image to the `Vehicle` constructor. (Because of this limitation, it is simplest for all the cars in a lane to have the same image.)

Finally, the vehicle needs to know about the frog so it can tell if it hit the frog. Again, our window controller created the frog. It can pass the frog to the `Lane` in its constructor. The lane can remember the frog so that it can tell the vehicles about the frog in the `Vehicle` constructor call.

After generating one vehicle, the lane should pause for some time. The pause should be at least long enough so that there will be a one car-length gap between pairs of vehicles. The pause should never be so long that it leaves more than about four car lengths between vehicles.

For simplicity, you may use a fixed value for the pause time when you first write your program, but eventually the pause time must be selected randomly. That is, the gaps between successive cars should not be the same, but should be distributed randomly with the constraint that there is at least one car length and no more than 4 car lengths between successive cars in the same lane. To do this, your code will have to use the speed with which vehicles travel and their lengths in pixels to compute the minimum and maximum amount of time you should pause between generating new cars.

The Image Files

There is an image file for the frog. We provide 8 images of vehicles. They show 4 different types of vehicles. For each type of vehicle there is an image of that vehicle facing right and another facing left. The image files are included in the starter folder. You can see their names listed in the left column of the Eclipse window. It is fine if you only use a single image file so that all your vehicles look the same and move in the same direction. Using multiple pictures is a feature that we hope you add (it does make the display look better), but it is not required.

The frog is 83 pixels wide and 48 pixels tall. The widest vehicle is 139 pixels wide. The tallest vehicle is 66 pixels tall. This information should help you figure out how to place the vehicles and frog within the lanes. Remember to use constants effectively so that you would not need to change many values if we introduced a much taller vehicle for example.

Getting Started

See the online version of this handout for instructions on obtaining a copy of the starter folder.

After downloading the starter project archive called froggerStarter.tar.gz from the web page and saving it in your cs134 folder, the files for the project will then be extracted to a folder in your cs134 folder called froggerStarter.

To import the starter code into Eclipse, open the "File" menu and select "Import...". Select "Existing Project into Workspace" and click the "Next" button. Now, click the "Browse" button and navigate to where you stored the froggerStarter folder. Click "Finish." When you import the project, Eclipse will automatically use the project name that we have given it. You should not change the project name.

To double check everything is set up correctly, try to run the program:

- Open the Run menu.
- Select "Run..."
- Select "Java Applet" from the Configurations list.
- Click the New button at the bottom left.
- Make sure that the Name field at the top and the Project say Frogger.
- Make sure that the Applet Class field says Frogger.
- Click on the Parameters tab.
- Enter 600 for the width and 600 for the height.
- Click the Apply button and then the Run button.

The "starter" folder contains several files intended to hold Java code. The file `Frogger.java` should be used to write the extension of the `WindowController` that will serve as your "main program". The `Frog.java` file should be used to hold your code for the `Frog` class, `Lane.java` will hold the code for the `Lane` class, and `Vehicle.java` will hold the code for the `Vehicle` class. The `Frogger.java` file contains code that draws the highway background and markings for you. All the files contain skeletons of code which you will need to complete.

A listing of the contents of the `Frogger.java` file included in the starter folder is attached to this handout. The online version of this document contains links to copies of all the Java code included in the starter folder.

There are many ways of proceeding with this lab. Here is one suggested ordering:

1. Read the `begin` method in `Frogger.java` to understand how we've drawn the highway background for you.

2. Write the `Frog` class except for the `kill` and `reincarnate` methods.

3. Modify the `Frogger` class to create the frog on the screen, and write `onMousePress` to control the movements of the frog. Make sure that the frog hops around appropriately.

4. Write the `Vehicle` class.

5. Test the `Vehicle` class by having the `begin` method in the `Frogger` class put one car on the road. (*You don't want the code here in the end, but for now it lets you test the* ***`Vehicle`*** *class before you've written the* ***`Lane`*** *class.*) Move the frog into the road to see if it gets killed by the car. Add code to reincarnate the frog and test it.

6. Write the constructor of `Lane`. Move the code that creates a car from the `Frogger` class to the `run` method of the `Lane` class so that a lane generates a stream of cars on the lane. Make sure they don't bump into each other (they shouldn't as they are all going the same speed). Make sure that the frog gets killed if hit by any of the cars (though our cars will be hit-and-run – they don't stop!).

Advanced Features

You may notice that the sample version of Frogger we have provided (and depicted in the image at the beginning of this handout) has more features than we have required. Vehicles move in both directions and several different images are used to represent vehicles. You are encouraged, but definitely not required, to incorporate such extensions in your program for a small amount of extra credit. Only do so, however, after completing the construction of a program that meets the basic requirements.

Submitting Your Work

Before submitting your work, make sure that each of the .java files includes a comment containing your name. Also, before turning in your work, be sure to double check both its logical organization and your style of presentation. Make your code as clear as possible and include appropriate comments describing major sections of code and declarations. Use the Format command in the Source menu to make sure your indentation is consistent. Refer to the lab style sheet for more information about style.

Turn in your project the same as in past weeks. Use the Export command from the File menu. Check that the Frogger project is being exported to a new folder whose name includes your name and identifies the lab. Then quit Eclipse, connect to Cortland, and drag your folder to the appropriate dropoff folder on Cortland.

Monday labs are due Wednesday at 11 PM. Tuesday labs are due Thursday at 11 PM.

Table 1: Grading Guidelines

Value	Feature
	Design preparation (3 pts total)
1 pt.	instance variables & constants
1 pt.	constructors and parameters
1 pt.	methods and parameters
	Style (5 pts total)
2 pts.	Descriptive comments
1 pts.	Good names
1 pts.	Good use of constants
1 pt.	Appropriate formatting
	Program Quality (5 pts total)
1 pt.	Good use of boolean expressions
1 pt.	Not doing more work than necessary
1 pts.	Check if killed and reincarnate at appropriate times
2 pts.	Parameters used appropriately
	Correctness (7 pts total)
1 pt.	Car moves smoothly
1 pt.	Car disappears at end of lane
1 pt.	Car kills frog appropriately
1 pt.	Cars spaced appropriately in lane
1 pt.	Frog moves correctly on clicks
1 pt.	Frog says “ouch” when killed
1 pt.	Frog reincarnated properly

Frogger.java

```
import objectdraw.*;
import java.awt.*;

public class Frogger extends WindowController {

   // Constants defining the sizes of the background components.
   private static final double HIGHWAY_LENGTH = 700;
   private static final double LANE_WIDTH = 100;
   private static final int NUM_LANES = 4;
   private static final double HIGHWAY_WIDTH = LANE_WIDTH * NUM_LANES;
   private static final double LINE_WIDTH = LANE_WIDTH / 10;

   // Constants defining the locations of the background components
   private static final double HIGHWAY_LEFT = 50;
   private static final double HIGHWAY_RIGHT = HIGHWAY_LEFT + HIGHWAY_LENGTH;
   private static final double HIGHWAY_TOP = 100;
   private static final double HIGHWAY_BOTTOM = HIGHWAY_TOP + HIGHWAY_WIDTH;

   // Constants describing the lines on the highway
   private static final double LINE_SPACING = LINE_WIDTH / 2;
   private static final double DASH_LENGTH = LANE_WIDTH / 3;
   private static final double DASH_SPACING = DASH_LENGTH / 2;

   // This method currently just draws the highway.  You will have to add
   // instructions to create the frog and the Lane ActiveObjects.
   public void begin() {

      // Draw the background
      FilledRect highway = new FilledRect (HIGHWAY_LEFT, HIGHWAY_TOP,
                                           HIGHWAY_LENGTH, HIGHWAY_WIDTH, canvas);

      // Draw the lane dividers
      int whichLine = 1;
      while (whichLine < NUM_LANES) {
        if (whichLine == NUM_LANES / 2) {
           // The middle line is a no passing line
           drawNoPassingLine (HIGHWAY_TOP + (whichLine * LANE_WIDTH) -
                                                    (LINE_SPACING / 2 + LINE_WIDTH));
        }
        else {
            drawPassingLine (HIGHWAY_TOP + (whichLine * LANE_WIDTH) - (LINE_WIDTH / 2));
        }
        whichLine = whichLine + 1;
      }

      // ADD YOUR CODE TO CREATE THE FROG AND THE LANES

   }

   // Draws a pair of solid yellow lines to represent a no passing divider between lanes
```

```
   // Parameter:  y - the top of the top line
   //
   // YOU SHOULD NOT NEED TO MODIFY THIS METHOD
   //
    private void drawNoPassingLine (double y) {
         // Draw the solid dividing lines
      FilledRect topLine = new FilledRect (HIGHWAY_LEFT, y,
                                           HIGHWAY_LENGTH, LINE_WIDTH, canvas);
      topLine.setColor (Color.yellow);

      FilledRect bottomLine = new FilledRect (HIGHWAY_LEFT, y + LINE_WIDTH + LINE_SPACING,
                                              HIGHWAY_LENGTH, LINE_WIDTH, canvas);
      bottomLine.setColor (Color.yellow);
   }

   // Draws a dashed white line to represent a passing line dividing two lanes of traffic
   // Parameters:  y - the top of the line.
   //
   // YOU SHOULD NOT NEED TO MODIFY THIS METHOD
   //
   private void drawPassingLine (double y) {
      double x = HIGHWAY_LEFT;
      FilledRect dash;

      while (x < HIGHWAY_RIGHT) {
            // Draw the next dash.
            dash = new FilledRect (x, y, DASH_LENGTH, LINE_WIDTH, canvas);
            dash.setColor (Color.white);
            x = x + DASH_LENGTH + DASH_SPACING;
      }

   }

   // Note: Use onMousePress rather than onMouseClick to decide when to move the frog
   public void onMousePress(Location point) {

   }

}
```