CPS 112: Computational Thinking with Algorithms and Data Structures
Homework 2 (35 points)

Midway Checkpoint Due: 11:59pm, February 7, 2018 (Wednesday)
Late submissions accepted with penalty until 11:59pm February 9, 2018 (Friday)
Final Due Date: 11:59pm February 14, 2018 (Wednesday)
Late submissions accepted with penalty until 11:59pm February 16, 2018 (Friday)

If you run into trouble or have questions, arrange to meet with me or the tutor!

Before you submit:

- **Please review the homework guidelines available on Canvas**

- Read the assignment very carefully to ensure that you have followed all instructions and satisfied all requirements.

- Create a zip file named like this: *yourusernameHWX.zip* (with your username and replacing the *X* with the assignment number) that contains a directory named *yourusernameHWX*, which contains all of your .java files, your debug log if you have one, and your cover sheet.

- Make sure to compile and thoroughly test all of your programs before you submit your code. **Any file that does not compile will receive a 0!**

- Ensure that your code conforms to the style expectations set out in the homework guidelines.

    - Make sure your variable names are descriptive and follow standard capitalization conventions.

    - Put comments wherever necessary. Comments at the head of each file should include a description of the contents of the file **(no identifying information please!)**. Comments at the beginning of methods describe what the method does, what the parameters are, and what the return value is. Use comments elsewhere to help your reader follow the logical flow of your code.

    - *Program readability and elegance are as important as correctness.* After you've written your function, read and re-read it to eliminate any redundant/unused lines of code, and to make sure variable and function names are intuitive and relevant.

NOTE: This is a two week project! To help you pace your work, there is a midway checkpoint. Your final submission should contain the *entire project*. The whole thing will be graded, including the problems that appear before the midway checkpoint. **The final score for Parts 1-6 will be the average of your score at the checkpoint and your score for the final submission.** This is to allow you the chance to encounter difficulties and recover or simply to get credit for improving on your initial solutions to these problems.

This little guy is Mosca the fly.

Unfortunately Mosca is lost and he needs your help getting home. The problem is that there are hungry frogs like the one below, looking to eat little Mosca.

For this assignment you will develop a simple game called Fly Away Home. The basic idea behind the game is to safely move Mosca around a grid to the home square before he gets eaten. The user will control the movements of the fly using arrow keys and the frogs will move on their own based on the descriptions given below. I have provided you with much of the GUI implementation. Below are the pieces you need to implement.

There are many pieces to this project. I strongly urge you to write and test in small chunks as suggested in the remainder of the homework description. Before writing any code, look through the provided files to familiarize yourself with the code that is given to you and read over the entire homework description to get a quick idea of what you will need to implement. You should NOT modify *FlyWorldGUI.java* or *GridLocation.java*. I have provided comments in those files so you very roughly understand what they do, but there is nothing in there that needs to be modified.

I have provided you with two sample files for some basic testing, world0.txt and world1.txt but you will need to create other files for more testing, especially after the midway check. The `main` method is in *FlyWorldGUI.java* and it requires one command-line argument, a file. For example, you could run the code as java `FlyWorldGUI world0.txt`

Part 1: Initializing the Grid (2 pts)

The grid will be initialized from an input file. You have been given two example files, world0.*txt* and *world1.txt*. The first line in the file indicates the number of rows and the number of columns (in that order). The remainder of the file provides information regarding the layout of the game. A '-' indicates a normal square. An 'h' is the home square. An 'f' indicates the location of a frog. The 's' indicates the starting location of Mosca.

In *FlyWorld.java* is the `FlyWorld` class. The constructor takes a `String` parameter, which is the name of input file. The method should open the given file and read the first line to get the dimensions, which should be stored in the instance variables `numRows` and `numCols` and initialize the `world` variable to be an appropriately sized double array of type `GridLocation`. Next read the rest of the file.

1. For each location in the grid, create a new `GridLocation` based on its row/column position and add it to the `world` variable.
2. If the location is the start position, set the background color (there's a method for this in the `GridLocation` class) to `Color.GREEN`, and set the `start` instance variable.
3. If the location is the home square, set the background color to `Color.RED` and set the `goal` instance variable.

If you compile and run the game now, it should display the grid with a home square in red and the start square in green. Not very fun yet...

Part 2: Adding Mosca (2 pts)

In *Fly.java* is a `Fly` class. There are some missing pieces.

1. Fill in the top part of the constructor (marked by comments). The bottom part, which is already written, loads the fly's image.

Now update constructor in `FlyWorld` so that when it encounters the start location in the file, it creates a `Fly`, and assigns it to the `fly` instance variable. Note that the `Fly` constructor takes a `GridLocation` and a `FlyWorld` object. You should already have a `GridLocation` if Part 1 is working. The world should pass *itself* into the constructor.

If you compile and run the game, you should now also see the fly on the grid. Still not very fun!

Part 3: Adding Frogs (5 pts)

In *Frog.java* the start of a `Frog` class is provided. Fill in the constructor (see the `Fly` constructor for the image loading code).

Now update constructor in `FlyWorld` so that it

1. Initializes the `predators` instance variable.
2. When you encounter a frog in the file, create a `Frog` and add it to the `predators` list variable.

You should now see frogs on the screen when you run the program. Make sure you see all the frogs you should see based on the example files I gave you. Now let's get things moving...

Part 4: Moving Mosca (3 pts)

Let's get Mosca moving in the right direction

1. In *FlyWorld.java* fill in the `isValidLoc` method.
2. In *Fly.java* class fill in the `update` method. **Hint**: Use `FlyWorld` `isValidLoc` and `GridLocation` `removeFly`/`setFly` methods to help
3. In *FlyWorld.java* fill in the `moveFly` method. For now it only needs to call the `update` method for Mosca and check if Mosca is on the goal location. If it is, return `FlyWorldGUI.ATHOME`, otherwise return `FlyWorldGUI.NOACTION`. **Hint**: You can compare two `GridLocation` objects using the `equals` method

If everything is working right, you should now be able to move Mosca around, and if you reach the goal the game should end. Make sure nothing bad happens if you try to move past the edges.

Part 5: Moving Frogs (5 pts)

In the `Frog` class fill in

1. The `generateLegalMoves` method. This should return a list of all legal `GridLocations` in the `world` that a frog can move to. Additional details are given in the comments for the method.
2. The `update` method. It should
    1. Call `generateLegalMoves`
    2. Randomly choose one of the legal moves (if there are any)
    3. Set the frog's location to the updated position (if there is one)

Finally, in *FlyWorld.java* fill in the `movePredators` method. For now only worry about updating every frog's position. Now you should be able to see the frogs move when you play. You should check to see that frogs don't overlap or go off the screen. Is it fun yet?

Part 6: Eating the Fly (3 pts)

In `Frog`, fill in `eatsFly`, which determines whether the frog is in position to eat the fly or not. You will find `world.getFlyLocation()` useful in this method.

Now in *FlyWorld.java*,

1. Update `moveFly` so that after updating Mosca's location and checking if Mosca is home, go through all the `Frog`s in `predator` and see if any of them have eaten Mosca If a frog has eaten Mosca return `FlyWorldGUI.EATEN`. If no frogs eat Mosca, return `FlyWorldGUI.NOACTION`
2. Update `movePredators` so that after you call `update` for each frog, check if that frog can eat Mosca. If it can, return `true`

---Midway Checkpoint (due 11:59pm, February 7, 2018)---

Part 7: Code Re-use (4 pts)

In lab we discussed the commonalities `Fly` and `Frog` share.
In *Creature.java* implement an abstract class `Creature` based on the work you did in lab.
Remember that instead of a `setFly` or `setFrog` method there will be a `setCreature` method for `location` when you use the updated *GridLocation.java* provided for you.

Now make `Fly` and `Frog` both subclasses of `Creature`. You should remove the instance variables and methods from these classes that are now contained in `Creature`. You should also change their constructors so they use the superclass' constructor to initialize those variables. Ah, that's better! It always feels good to get rid of some code duplication. Remember that, while `Fly` and `Frog` both have `update` methods, they are abstractly different (`Fly`'s takes a direction from the user, but `Frog`'s doesn't). It probably makes sense to leave those methods in the individual subclasses.

Part 8: Prepare for Polymorphism (2 pts)

A fly's life is dangerous, and frogs are not the only things to watch out for. Soon you'll be implementing some more predators for Mosca to avoid. In order to do that, the program needs to be able to work with more than just `Frog`s!

In *Predator.java* implement an abstract class `Predator`. It should be a subclass of `Creature`. It should have a constructor that simply invokes `super`'s constructor. It should have two public abstract methods:

- `ArrayList<GridLocation> generateLegalMoves()`
- `boolean eatsFly()`

It should also have two concrete methods `update()` and `isPredator()`. The `update` method should work the exact same way the `update` method in `Frog` works since all predators will update their position the same way. The difference between `Frog`'s and other predators is that they will have different rules for moving around the board and that is handled by the abstract method `generateLegalMoves`.

Make `Frog` a subclass of `Predator` (instead of `Creature`) and remove the update method.

Now that we are rethinking the architecture of the program, you'll need to make some changes to *FlyWorld.java*. Some references to the `predators` instance variable need to be changed to now be type `Predator` and not `Frog`. This includes its declaration at the top of the file, its initialization in the constructor, and its use in the `moveFly` and `movePredators` methods.

Compile all your code and test it carefully. If you changed everything correctly, your code should work the way it did before. Make sure you are using the updated *GridLocation.java* file provided on Canvas

Important coding lesson:
Having made all these changes, you might see why we're always talking about carefully planning things out beforehand. Re-architecting a complicated system like this can be a tedious and error-prone process, which is why software developers try their hardest to avoid it!

Part 9: For the Birds (4 pts)

Implement a new class `Bird` (in *Bird.java*), which will be a subclass of `Predator`. A `Bird` behaves in the following way:

- Its image is *bird.jpg*.
- Birds can move to any unoccupied (by a predator) location on the grid.
- A `Bird` can only eat the fly if it is on the same square as the fly.

  You should also change the constructor in *FlyWorld.java* so that it creates a `Bird` and adds it to `predators` when it encounters a 'b' in the input file. Go ahead and make an example file that has both frogs and birds in it to test it all out.

Part 10: Along Came a Spider (5 pts)

Implement a new predator called `Spider`. A `Spider` behaves like this:

- Its image is *spider.jpg*.
- When a `Spider` moves, it always moves toward the fly. If the spider is in the same column or row, then it should move toward the fly along that column or row, unless that position is occupied by another predator, in which case it does not move. Otherwise the spider may randomly choose whether to move horizontally or vertically toward the fly (unless one or both of those positions are already occupied by another predator).
- A `Spider` can only eat the fly if it is on the same square as the fly.

Once again, alter the `FlyWorld` constructor so that when it encounters an 'a' (for arachnid) in the file you create a `Spider` and add it to `predators.`

---Final Submission (due 11:59pm, February 14, 2018)---

This is the end of the project! Your final submission of the *entire project* will be graded. Your score for the first 6 parts will be the average of your checkpoint submission and your final submission. Make sure you've tested everything thoroughly and fixed as many problems with your midway checkpoint submission as you can (feel free to come see me if you are stuck on something). **Go back through your code to look for opportunities to improve readability or elegance.**