P04 Carrot Patch

Overview

This assignment involves the development of a graphic application from the scratch. This will give you practice with implementing interfaces, and defining an hierarchy of classes including super classes, and subclasses. We'll also work on improving the organization and clarity of your code. Having this programming assignment as reference, you will also be able to develop your own graphical application using the processing library, from the scratch. We no longer need to use the Utility class provided for you in the p2core.jar file in P2. The CarrotPatch graphic application defines different graphic components which can be drawn to the display window. We distinguish:

- Carrots which are non interactive images. They do not react to the mouse events or to key pressed by the user.
- Interactive objects such as buttons and animals (rabbits and wolves). These graphic objects are GUI listeners objects which react to the mouse events or specific key pressed. They define common properties and common behaviors. But they act differently in the patch. Each animal/button has a specific behavior indeed. We are going to define and implement these similarities and differences in this assignment.

A screenshot of what the display window of this Carrot Patch application may look like after you have completed this assignment is shown in Fig.1. A demo of this application is provided soon in this **video**.

Learning Objectives

The goals of this assignment include:

- Experience organizing code in an object oriented fashion that takes advantage of inheritance relationships between your classes.
- Use of inheritance and interfaces to better organize your code in a more clear and concise manner. Students will also enjoy the power of polymorphism.
- Learn how to use `PApplet` class defined in the processing graphic library directly to develop a graphic application from the scratch, rather than through a provide .jar wrapper file.
- Gain experience with writing tests to assess the correctness of your program. All developed test methods will be run with graphic mode disabled and would assert the correctness of specific methods in your code.

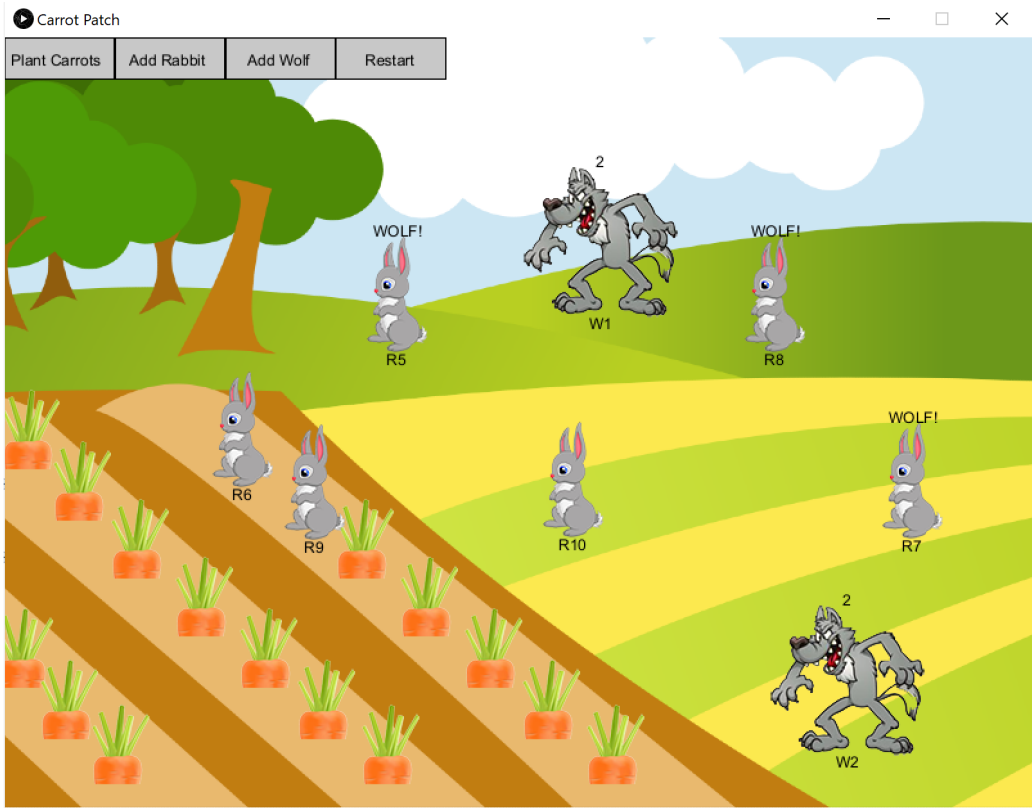

Figure 1: Carrot Patch Graphic User Interface

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P4 pre-assignment quiz is accessible through Canvas before having access to this specification by **11:59PM on Sunday 02/28/2021**. Access to the pre-assignment quiz will be unavailable passing its deadline.
20 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is otherwise correct. To become more confident of this, you should run additional tests of your own.
25 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.

Additional Assignment Requirements and Notes

- **The only import statements** that you may include in your project's classes are:

```
import java.io.File;
import java.util.ArrayList;
import java.util.Random;
import processing.core.PApplet;
import processing.core.PImage;
```

- The automated grading tests in gradescope are **NOT using the full processing library** when grading your code. These tests only know about the following fields and methods (referenced directly from this assignment). If you are using any other fields, methods, or classes from the processing library, this will cause problems for the automated grading tests. Such references must be replaced with references to one or more of the following:
  - `PImage`: the **two fields** int `width`, and int `height`.
  - `PApplet`: the **three fields** int `mouseX`, int `mouseY`, and boolean `mousePressed`.
  - **Five methods** from `PApplet`: PImage `loadImage(String)`, void `image(PImage,int,int)`, void `size(int,int)`, void `getSurface().setTitle(String)`, and void `main(String)`.
- You are NOT allowed to add any constant or variable not defined or provided in this write-up outside of any method.
- You are NOT allowed to add any public method (static or instance) not defined in this write-up.
- Overriding a method does not mean adding a new public method not defined in the write-up. Feel free to override any public behavior implemented in a superclass, if needed.
- You CAN define local variables that you may need to implement the methods defined in this program.
- You CAN define private static methods to help implement the different public methods defined in this write-up, if needed.
- Do NOT submit the following source files to gradescope: ListenerGUI.java, Button.java, and Carrots.java.
- You MUST adhere to the course's Academic Conduct Expectations and Advice.
- All implemented methods including the overridden methods MUST have their own javadoc-style method headers, with accordance to the CS300 Course Style Guide.

1 Getting Started

Start by creating a new Java Project in eclipse. You can call it `P04 Carrot Patch` for instance. You must ensure that your new project uses Java 11, by setting the "Use an execution environment JRE:" drop down setting to "JavaSE-11" within the new Java Project dialog box. Do NOT create a project-specific package; use the default package.

Download the images and processing core.jar

Next, download this P4Distributables.zip file, and extract the contents of this archive file directly into your P4 project folder. This should add:

- a **core.jar file** which is part of the processing graphical library. You are welcome (but not required) to read more about here. If this .jar file does not immediately appear in the Project Explorer, try right-clicking your project in the project folder and selecting "Refresh" to fix that. To use of the code within this jar file, you'll need to right-click on it within the Project Explorer and choose **"Add to Build Path"** from the "Build Path" menu.
- a **folder of images** which contains the four following files: background.png, carrot.png, rabbit.png, and wolf.png. The core.jar file .

The result of this operation on the Package Explorer is illustrated in Fig.2.

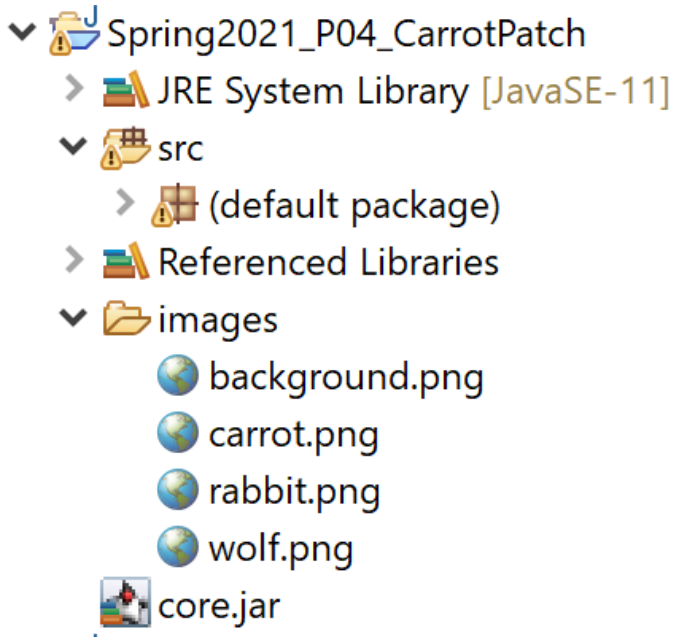

Figure 2: Eclipse Package Explorer

2 Create the CarrotPatch main class and download the supportive code

Create a new class called `CarrotPatch` which includes a main() method. Your `CarrotPatch` class MUST inherit from a class defined the core.jar file called `PApplet` (You'll need to import processing.core.PApplet for this to work). This makes your CarrotPatch class to BE of type PApplet. PApplet models the Processing window for a graphic application or program. It provides good ways to create a new window and draw graphics onto it. Note that in your whole graphic application, only ONE PApplet object must be used. It is an instance of this CarrotPatch class. All the visible and interactive objects including images which will be defined in this application will be drawn in this CarrotPatch PApplet object.

2.1 CarrotPatch's main() method

Now, define a main method to include only the following single statement.

```
PApplet.main("CarrotPatch"); // do not add any other statement to the main method
// The PApplet.main() method takes a String input parameter which represents
// the name of your PApplet class.
```

This should result in a program that opens a small window when it is run. We won't be adding anything more to this main method for the rest of this assignment.

2.2 Download the supportive code

For this assignment, we give you a supporting code that you will start working on it and extend it. Download each of the following files and make sure they are included in the same project directly inside the default package in the src folder. If they do not immediately appear in the Project Explorer, try right-clicking your project in the project folder and selecting "Refresh" to fix that.

- interface GUIListener (GUIListener.java)
- class Button (Button.java)
- class Carrots (Carrots.java)

2.3 Define the CarrotPatch data fields

Your `CarrotPatch` class must contain the following **instance** data fields at the top of the CarrotPatch class outside of any method.

- **backgroundImage**: private instance field of type **PImage**. It represents the background image of this application. (You'll need to import processing.core.PImage).
- **objects**: protected instance field of type $ArrayList < GUIListener >$. It stores all the interactive objects which will be defined in this CarrotPatch graphic application.

Do NOT initialize any of the above data fields where they are defined. They will be initialized in the `setup()` method.

2.4 Define the CarrotPatch methods

Your CarrotPatch class must contain and override the instance methods defined in the following text file. There are five callback methods from the PApplet class to override: `setup()`, `draw()`, `mousePressed`, `mouseReleased()`, and `keyPressed`. Note these methods are used in a similar fashion to the setup (called once at the beginning of our program), draw (called repeatedly while the program runs) mousePressed (called each time the mouse is pressed), and keyPressed (called each time a key is pressed) methods from P2 Memory Game. Notice that with respect to P2 implemented using procedural programming, this CarrotPatch class which extends PApplet is implemented based on the Object-oriented programming paradigm. All the callback methods are instance methods (versus static methods in P2). Some other PApplet methods are new such as settings() and some graphic setting features implemented in the private method `CarrotPatchSettings()` called in the setup() method).

In addition to the callback methods, the `CarrotPatch` class defines an instance method called `removeAll` which removes all carrots and animal objects already present in the patch. We are going to complete the implementation of the missing code (see TODO tags) as we go throughout this write-up.

2.5 Initialize the CarrotPatch instance fields

To start, initialize the `objects` list to an empty ArrayList in the `setup()` method. Then, load the background image in the setup() method using PApplet.loadImage() method as follows.

```
backgroundImage = this.loadImage("images" + File.separator + "background.png");
```

Then, draw it to the center of the screen at the beginning of the draw method using the using the PApplet.image() method as follows.

```
this.image(backgroundImage, this.width / 2, this.height / 2);
```

Recall that the `setup()` method is a callback method called only one time when the graphic application starts. Now, running this program will now result in a window showing the background image as depicted in Fig.3. Next, let's explore the other provided source files before we proceed.

Figure 3: Carrot Patch Background Screen

3 Explore the supportive code

3.1 GUIListener interface

Our P04 Carrot Patch program is a graphical application which interacts with the user via a Graphical User Interface (GUI). For instance, our program waits for a user input as one of the following events: press on the mouse button, release the mouse, drag an image, etc. Then, when such event is detected, a particular method is invoked.

Our application will contain different graphic objects (buttons and animals as shown above in Fig.1). These objects will be drawn repeatedly to the display window. They will listen to the mouse events and must react appropriately to them each time the mouse is pressed or released. To avoid redundancy and promote abstraction in our code, we defined the interface GUIListener (GUI refers to Graphic User Interface). You can notice that this interface contains the signatures of the following methods ONLY without any implementation details.

```
public interface GUIListener {

public void draw(); // draws this interactive object to the display window
public void mousePressed(); // called each time the mouse is Pressed
public void mouseReleased(); // called each time the mouse is Pressed
public boolean isMouseOver(); // checks whether the mouse is over this GUIListener object

}
```

Recall that an interface cannot be constructed. We can define a reference of type GUIListener. It can store the reference to any object instance of a class which implements this interface.

3.2 Button class

You can notice from the final display window for our CarrotPatch application shown in Fig.1, that we are going to add four buttons "Plant Carrots", "Add Rabbit", "Add Wolf", and "Restart". All these buttons will define a similar state (data fields) and share similar functionalities. Therefore, we created the class Button to serve as the base class (aka super class) for each of the specific Button classes that we will create later in this assignment. Each button will be drawn to display window, listen and react to the mouse events. Therefore, the class Button implements our GUIListener interface and overrides all its methods.

Note that the Button class contains the following fields.

```
private static final int WIDTH = 85; // Width of this Button
private static final int HEIGHT = 32; // Height of this Button
protected static CarrotPatch processing; // PApplet object where this button
                                         // will be displayed
private float[] position; // array storing x and y positions of this Button
                         // with respect to the display window
protected String label; // text/label that represents the name of this button
```

Notice carefully that the Button class defines one **protected** static field named processing of type `CarrotPatch`. It represents the graphic display window of this application where all the buttons will be drawn and will operate. It is protected. So, it is directly visible to all the subclasses of the class Button.

This class implements two constructors, a static setter method to set the processing field, and overrides the four methods defined in the GUIListener interface. Make sure to read through the class definition of the Button class and its implementation. To help you further understand the code for drawing a button, we note that we used the following processing methods defined in the PApplet class.

- rect(): This is the processing method for drawing a rectangle. It takes four parameters: 1) the x-position of the upper left corner, 2) the y-position of the upper left corner, 3) the x-position of the lower right corner, 4) the y-position of the lower right corner. These corner positions must be calculated so that the button appears centered around it's position field with the appropriate WIDTH and HEIGHT.

- fill(): This method is used to set the drawing color. When the mouse is over a button, its filling color is set to dark gray by calling fill(100), and when the mouse is not over this button, its filling color is set to light gray fill(200)

- text(): This processing method is used to draw a text to the display window. The PApplet.text() method takes three parameters: 1) the string of text to draw, 2) the x-position that this text should be centered around, and 3) the y-position that this text should be centered around.

Note that since our `CarrotPatch` class extends PApplet. All these public methods defined in

the PApplet class must be directly accessible through the Button.processing data field of type CarrotPatch.

3.3 Carrots class

The Carrots class models a list of the carrots planted in the carrot patch. This class will not be instantiable. It defines static data fields and implements static methods ONLY. The carrots will be visible objects in this patch (meaning instances of processing.core.PImage class). They do not interact with the mouse events. They are NOT GUIListener objects.

Notice that the Carrots class defines one static field of type CarrotPatch which represents the PApplet object where the carrots will be drawn. The Carrots class also defines a perfect size array of the potential positions of the carrots in the display window and a perfect-size array of the planted carrots. We invite you to read through the provided implementation of the class Carrots and its different methods. Do not hesitate to contact your instructor or any of our consultants or post a question on piazza if you have a question about any of the provided methods.

3.4 Set processing for the Carrots and Button classes

Now, let's set the processing CarrotPatch display window for the Carrots and the Button classes. In the `CarrotPatch.setup()` method, call the `Carrots.settings()` and the `Button.setProcessing()` methods and pass them a reference to the current CarrotPatch object (which is the reference **this**).

4 Create and add specific buttons

Now Create the four following classes. They MUST **extend** the `Button` base class. This makes them all of type `GUIListener` since the Button class implements that interface.

- PlantCarrotsButton extends Button
- AddRabbitButton extends Button
- AddWolfButton extends Button
- RestartButton extends Button

All these classes must define the following instance methods. No specific data fields are defined in these classes.

- One constructor which takes two float variables as input parameters. The first float represents the x-position of the button. The second float represents the y-position of the

button. Each of these constructors calls the constructor of the superclass Button(String, float, float). The String labels of the four specific buttons are as follows:

  - "Plant Carrots" for the PlantCarrotsButton,
  - "Add Rabbit" for the AddRabbitButton,
  - "Add Wolf" for the AddWolfButton,
  - "Restart" for the RestartButton,

- Only the public instance method mousePressed() must be overridden in each if the 4 specific button classes. At this stage, you can add a specific print statement which will be displayed to the console (not on the graphic display window) when the mouse is pressed and is over each specific button. For instance, this can be an implementation of PlantCarrotsButton.mousePressed() method. We are going to change it later.

```
if (isMouseOver()) {
  System.out.println("Plant Carrots Button Pressed");
}
```

Now, instantiate in your `CarrotPatch.setup()` method the following four buttons and add them to the `objects` ArrayList. Recall that the Button class implements the `GUIListener`.

- an instance of PlantCarrotsButton at position (43, 16),
- an instance of AddRabbitButton at position (129, 16),
- an instance of AddWolfButton at position (215, 16),
- an instance of RestartButton at position (301, 16); where position(x,y) represents the x- and y-coordinates where the related button will be placed in the screen display window.

Note that each reference of type Button or its subclasses can be safely cast to the type GUIListener and stored in the `objects` ArrayList.

4.1 Click on buttons

To use these buttons, we have to update first the implementation of the `CarrotPatch.draw()`, `CarrotPatch.mousePressed()`, and `CarrotPatch.mouseReleased()` methods as follows.

- In the `CarrotPatch.draw()` method, after drawing the background image, draw the planted carrots first by calling Carrots.draw() method. Then, traverse the contents of the ArrayList `objects` and draw each of the interactive objects stored there.
- In the `CarrotPatch.mousePressed()` method, traverse the ArrayList `objects` and call the mousePressed() method of the first clicked element ONLY.

- In the `CarrotPatch.mouseReleased()` method, traverse the ArrayList `objects` and call the mouseReleased() method of every element stored in the list.

Running the program will result in the display window comparable to the one provided in Fig.4 where the buttons have been clicked in this order: (1) "Plant Carrots", (2) "Add Rabbit", (3) "Add Wolf", and (4) "Restart". Notice that clicking on the different buttons will print the specific behavior implemented in the mousePressed of each button. Notice the power of polymorphism. Even though the arraylist objects stores elements of type GUIListener, each element performs differently with respect to the implementation of the mousePressed() method overridden in its most specific type.

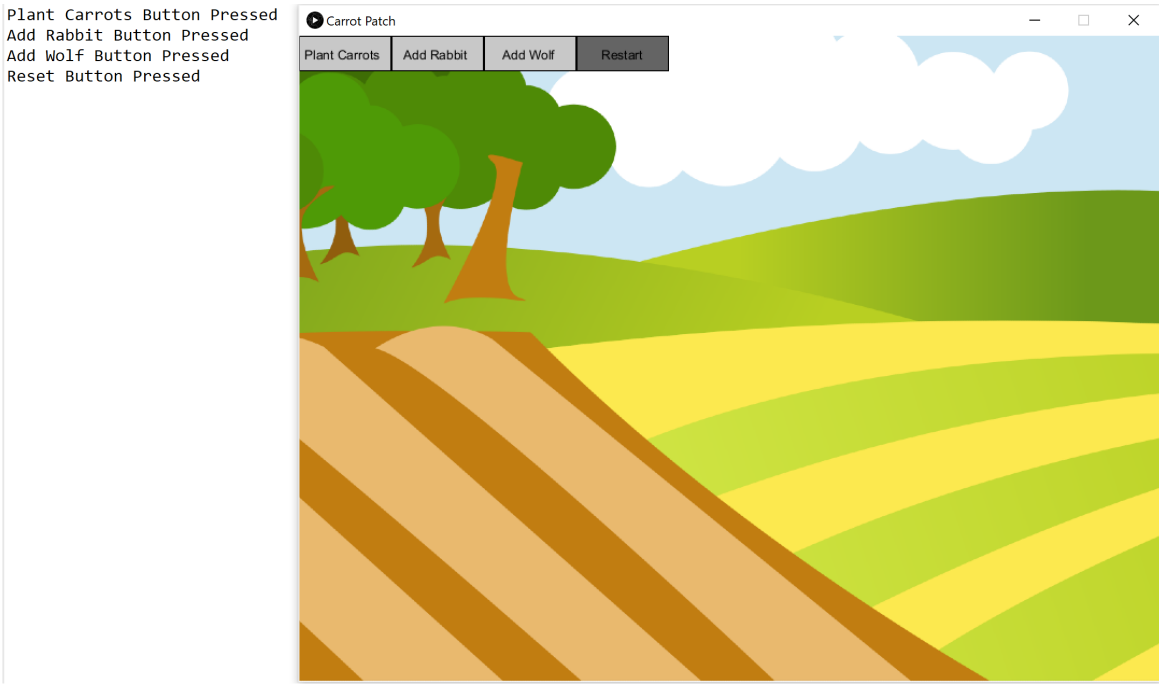

Figure 4: CarrotPatch with 4 buttons

4.2 Update the implementation of PlantCarrotsButton.mousePressed()

Now, you can implement the specific behavior of the PlantCarrotsButton when clicked. You can comment the print statement in the PlantCarrotsButton.mousePressed() method and instead call the Carrots.plant() method which is going to plant carrots in all the available spots within the Carrots.carrots array. Before checking whether this code will work or not, double check that the Carrots.draw() method call is added to the `CarrotPatch.draw()` method just before drawing the interactive objects stored in the objects arrayList.

This done, running your program and then clicking on the "Plant Carrots" button should result in a display window comparable to the one shown in Fig.5.

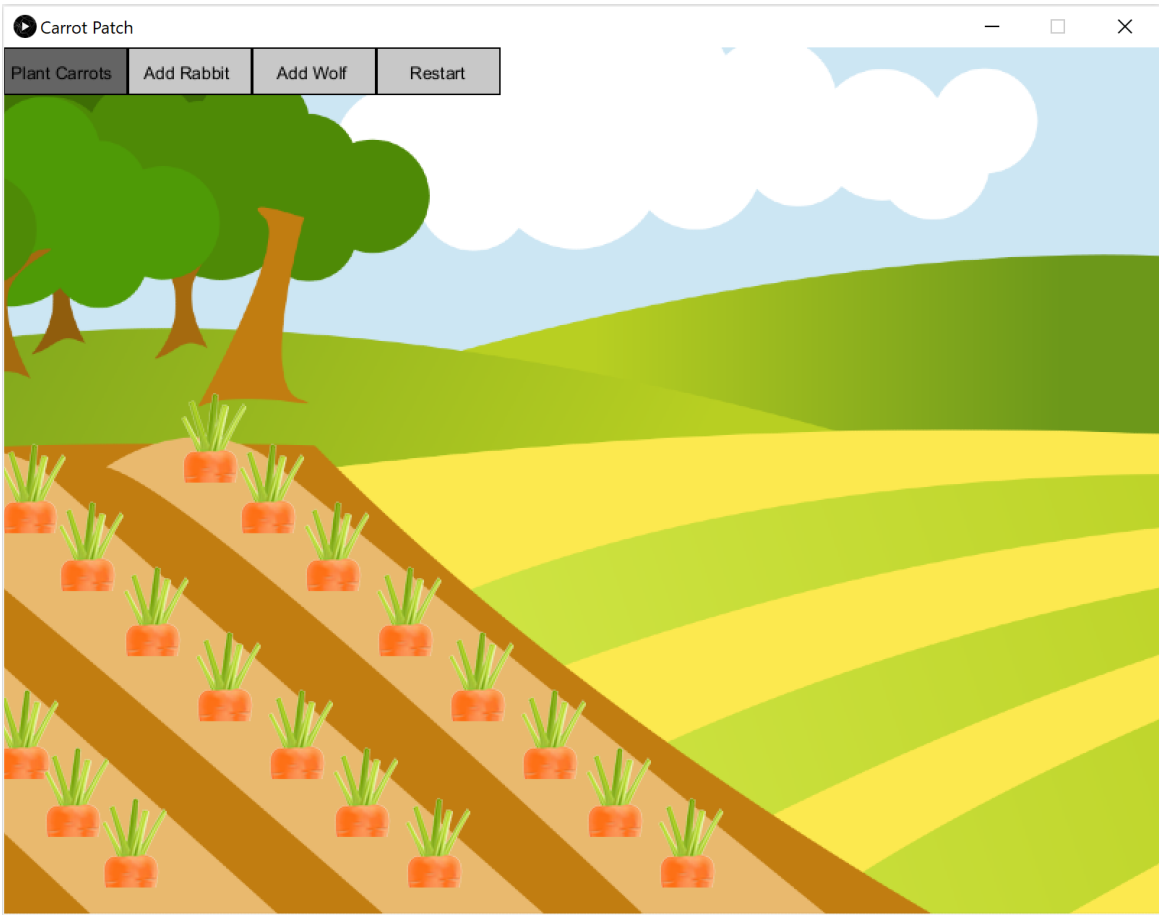

Figure 5: Plant Carrots Button Clicked

5 Create Animal base class

Now create a new class called `Animal`. This class MUST implement the `GUIListener` interface. It plays the role of a base class of any specific animal type such as Rabbit and Wolf which will be implemented in the next steps of this assignment. The class Animal defines the following data fields.

```
private static Random randGen = new Random(); // Generator of random numbers
protected static CarrotPatch processing; // PApplet object representing the display window
protected PImage image; // image of this
protected String label; // represents the name/identifier of this animal
private int x; // x-position of this animal in the display window
private int y; // y-position of this animal in the display window
private boolean isDragging; // indicates whether the animal is being dragged or not
private int oldMouseX; // old mouse x-position
private int oldMouseY; // old mouse-y position
```

It also defines the two following constructors (available on this supportive source file).

```
/**
 * Creates a new Animal object positioned at a given position of the display window
 *
 * @param processing  PApplet object that represents the display window
 * @param x           x-position of the image of this animal in the display window
 * @param y           y-position of the image of this animal in the display window
 * @param imageFileName filename of the animal image
 */
public Animal(int x, int y, String imageFileName) {

  // Set Animal drawing parameters
  this.x = x; // sets the x-position of this animal
  this.y = y; // sets the y-position of this animal
  this.image = processing.loadImage(imageFileName);
  isDragging = false; // initially the animal is not dragging
}

/**
 * Creates a new Animal object positioned at a random position of the display window
 *
 * @param processing  PApplet object that represents the display window
 * @param imageFileName filename of the animal image
 */
public Animal(String imageFileName) {
  this(randGen.nextInt(processing.width), Math.max(randGen.nextInt(processing.height), 200),
      imageFileName);
}
```

In addition, we provide you with the implementation of the GUIListener.draw() method and a few setter and getter methods in this text file. Read carefully through the implementation of the method Animal.draw(). Try to understand how the behavior of dragging an animal such that it follows the mouse moves when clicked and dragged has been implemented.

Override GUIListener abstract methods

The class Animal MUST override all the abstract methods defined in the `GUIListener`. Therefore, you MUST override on your own the following methods.

```
public boolean isMouseOver() {} // checks whether the mouse is over this animal
public void mousePressed() {} // starts dragging this animal by setting its isDragging
                  // field to true if the mouse is pressed and is over this animal
public void mouseReleased() {} // stops dragging this animal by setting its
                           // isDragging field to false
```

Animal class additional behaviors

Besides, the `Animal` class defines the following behaviors. Implement the two versions of the overloaded method `isClose()` with respect to their specification provided in their javadoc style method headers comments. Notice that a method called `distance` to compute the euclidean distance between the current animal and a given (x,y) position was provided for you. DO NOT make any change to the `action()` method. A default animal does not perform any action in this CarrotPatch application. This method must be overridden by the derived classes. Note that every animal has a continuous behavior. Therefore, call action() method from the Animal.draw() method.

```
 /**
 * Checks whether this animal is within a distance range with respect to a
 * given (x,y) position. It returns TRUE if this animal is located within a
 * range distance [0 .. range] around the provided position and FALSE otherwise.
 *
 * @param x a given x-position
 * @param y a given y-position
 * @param range     radius of the neighborhood range from the given (x,y) position
 * @return true if otherAnimal is close to this animal with respect to range
 */
public boolean isClose(int x, int y, int range) {
  // TODO implement this method

}

/**
 * Checks whether this animal is within a distance range with respect to anotherAnimal.
 * It returns TRUE if otherAnimal is located within a range distance [0 .. range]
 * around the current animal and FALSE otherwise.
 *
 * @param otherAnimal an animal to check if it is close to this animal
 * @param range     radius of the neighborhood range from the position of this Animal
 * @return true if otherAnimal is close to the current tiger
 */
public boolean isClose(Animal otherAnimal, int range) {
  // TODO implement this method

}

/**
 * Continuous behavior performed by this animal in the carrot patch
 */
public void action() {
  // This method should be overridden by a subclasse
}
```

Set the processing field of the class Animal in the CarrotPatch.setup() method

Notice also that all the animal objects share the same static field `processing` of type `CarrotPatch` which stores the reference to the PApplet display window of this application. Make sure to call the `Animal.setProcessing()` method in the `CarrotPatch.setup()` method to set the processing field of all animals to the current CarrotPatch object `this`.

6 Create and implement Rabbit class

Now, let's create the class `Rabbit`. This class represents the Rabbit animal. It MUST extend the Animal class. Rabbits can be added to randomly positions to the CarrotPatch application. They can move in the patch. They watch out for wolves and when they are clicked they make one hop towards a carrot. They also eat carrots. Every created Rabbit has their own label in the format *R#*. The first created Rabbit is labelled R1, the second R2, the third R3, and so on.

6.1 Define Rabbit's data fields, constructor, and getter methods

Here is a list of the data fields, and a provided implementation of the constructor of the class Rabbit. Note that the constructor of the class Rabbit creates a new Rabbit located at a Random position of the display window and initializes its instance fields.

```
private static final String RABBIT = "images"+ File.separator + "rabbit.png";
private static final String TYPE = "R"; // A String that represents the rabbit type

private static int hopStep = 70; // one hop step
private static int scanRange = 175; // scan range to watch out for threats
private static int nextID = 1; // class variable that represents the identifier
                               // of the next rabbit to be created
private final int ID; // positive number that represents the order of this rabbit

/**
 * Creates a new rabbit object located at a random position of the display window
 */
public Rabbit() {
  // Set rabbit drawing parameters
  super(RABBIT);

  // Set rabbit identification fields
  ID = nextID;
  this.label = TYPE + ID; // String that identifies the current rabbit
  nextID++;
}
```

```
// getter of Rabbit.scanRange static field
public static int getScanRange() {}

// getter of Rabbit.hopStep static field
public static int getHopStep() {}
```

6.2 Add new Rabbits to the carrot patch

Before implementing the specific behaviors related to Rabbit objects, let's try to add new Rabbits to our CarrotPatch. To do so,

1. Update the `AddRabbitButton.mousePressed()` method such that a new instance of the Rabbit class is created and added to the `CarrotPatch.objects` arraylist each time the mouse is pressed and is over a AddRabbitButton object. Notice that the `objects` array list is a protected instance field accessible directly using the Button.processing static field visible to the AddRabbitButton class.

2. Update the `CarrotPatch.keyPressed()` method such that a new Rabbit is added to the CarrotPatch's objects array list each time the R-key is pressed.

Now, if you run your program, a new Rabbit will be added to the carrot patch graphic display window at random positions each time the R-key is pressed or the "Add Rabbit" button is clicked. The Fig.6 shows an example of this application run at this level.

Note also that even though you did not make any further change to the program, the added rabbit objects respond to the mouse events. These behaviors have been already implemented in the Animal class and Rabbit IS-An Animal. If the rabbits do not follow the mouse moves when they are dragged, review your implementation of the Animal.isMouseOver(), Animal.mousePressed(), and Animal.mouseReleased() methods.

7 Create and implement Wolf Class

Now create a new class called `Wolf` which extends the `Animal` class. This class models wolves objects in the CarrotPatch. Wolves are looking for rabbits in the patch to eat them.

7.1 Define the Wolf's data fields

The class `Wolf` defines the following data fields.

```
// path to the wolf image file
private static final String WOLF = "images" + File.separator + "wolf.png";
```

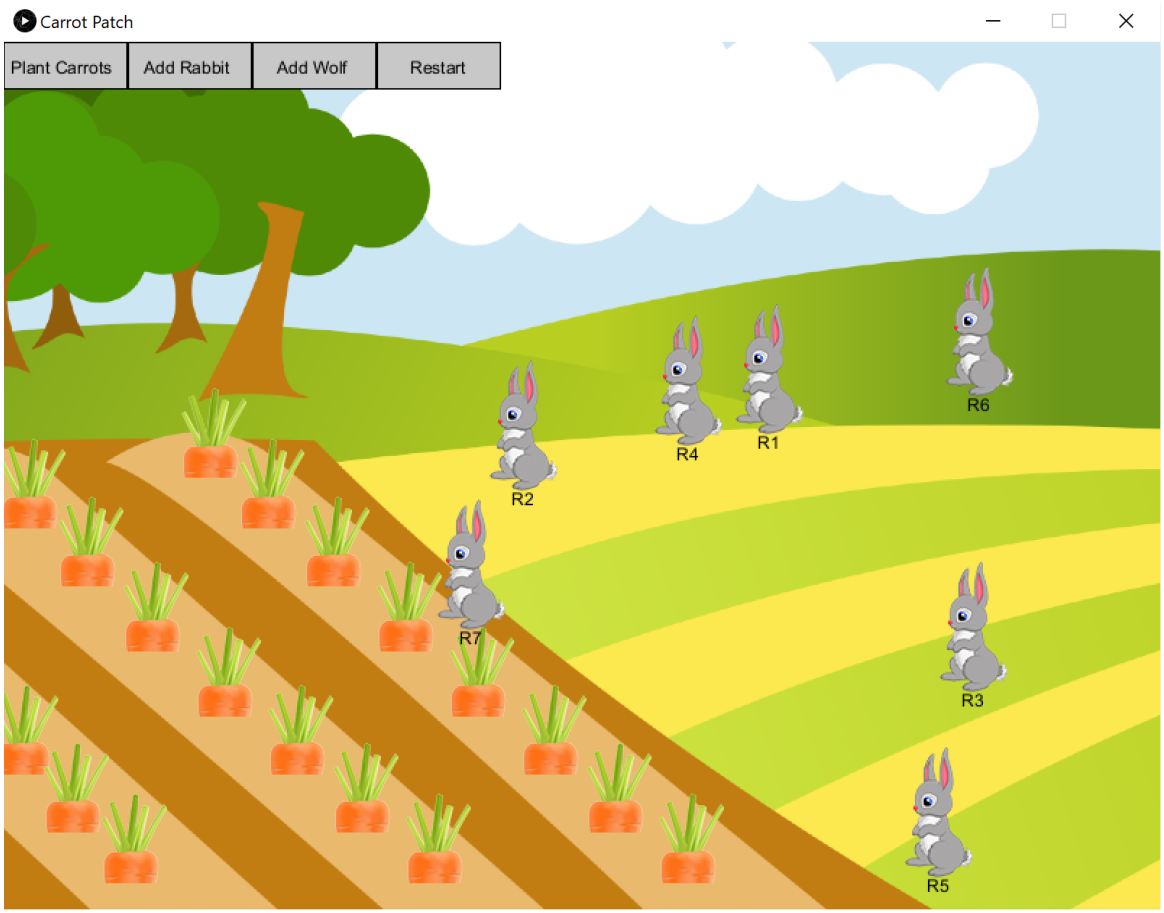

Figure 6: Rabbits in the Carrot Patch

```
private static int scanRange = 120; // scanning range to look for a rabbit
                                    // in the neighborhood
private static int nextID = 1; // identifier of the next Wolf to be created
private static final String TYPE = "W"; // A String that represents the Wolf type
private final int ID; // positive number that represents the order of this Wolf

private int rabbitEatenCount; // Number of rabbits that the current Wolf has eaten so far
```

7.2 Implement the constructor and getter methods of the Wolf class

Your `Wolf` class MUST define only one no-argument constructor which must create a new wolf and initialize its instance fields. Wolves are created at the same fashion of creating new rabbits. A new wolf is labelled $W\#$. The first created walf is named W1, the second is named W2, the third is named W3, and so on. Wolves will be created at random positions within the display window.

```
/**
 * Creates a new Wolf object at a random position of the display window
 *
 * @param processing CarrotPatch object which represents the display window
 */
public Wolf() {}
```

In addition, implement the following getter methods.

```
public int getRabbitEatenCount(){} // gets rabbitEatenCount instance field
public static int getScanRange(){} // gets the scanRange of a Wolf
```

7.3 Add new Wolves to the Carrot Patch

To add new wolves and display them to the CarrotPatch, update the `CarrotPatch.keyPressed()` and the `AddWolfButton.mousePressed()` methods as follows.

1. Update the `AddWolfButton.mousePressed()` method to instantiate a new Wolf object and add it to the `CarrotPatch.objects` arraylist each time an AddWolfButton object is clicked.

2. Update the `CarrotPatch.keyPressed()` method such that a new Wolf is created and added to the CarrotPatch's objects array list each time the W-key is pressed.

Running the program and clicking on "Plant Carrots", "Add Rabbit", "Add Wolf" buttons, or pressing the R-key or W-key will result in an interactive interface comparable to the one shown in Fig.7. Notice that the animals (whether wolves or rabbits) drawn in the graphic display window can be dragged with response to the mouse moves. If two or more animals overlap, and the user tries to drag them, only the oldest animal added to the patch will be dragged.

7.4 Remove animals from the Carrot Patch

Now, implement removing an animal from the Carrot Patch when the mouse is over it and the "D-Key" is pressed. To do so, update the implementation of the `CarrotPatch.keyPressed()` method to implement this behavior. Traverse the `objects` list and remove the **first** animal whose `isMouseOver()` returns true. Keep in mind that the objects list contains GUIListener references of different specific types (buttons and animals). Make sure to consider only instances of the Animal class in your search.

After implementing this behavior appropriately, run your program and check whether a wolf or a rabbit is removed from the CarrotPatch each time the D-key is pressed while the mouse is over it.

Figure 7: Rabbits and Wolves in the Carrot Patch

8 Implement RestartButton.mousePressed() behavior

Go to the CarrotPatch class and try to implement the `CarrotPatch.removeAll()` method with respect to its specification and the provided hints. Then, update the implementation of the `RestartButton.mousePressed()` behavior. When the mouse is pressed and is over the RestartButton button, all the planted carrots and all the animals must be removed from the CarrotPatch. Next, run your program and check whether this functionality works as expected.

9 Develop unit tests

Now that Rabbits and wolves can be added to the CarrotPatch, we can implement a few test methods to assess the correctness of the Animal.isClose() methods. Create a new class called `CarrotPatchTester` and add it to your project. This class MUST extend the `CarrotPatch` class and override its `setup()` method.

You can download the starter code for the `CarrotPatchTester` here. We provide you with an example showing how to define test scenarios for test1isCloseMethod(). You can make calls to your test methods in the overridden setup() method. Feel free to reuse the provided

implementation verbatim in your `CarrotPatchTester` class. You can also inspire by these examples and define your own scenarios for that test method. You MUST implement the `test2isCloseMethod()` and `test3isCloseMethod()` test methods.

In the next sections, we are going to implement the specific actions performed by Rabbit and Wolf objects in the CarrotPatch.

10 Implement the Rabbit specific behaviors in the CarrotPatch

In the Carrot Patch, our rabbit objects implement the following two behaviors.

- Each time a rabbit is clicked, it will make ONE hop step towards the first available carrot in the carrot patch. If it reaches the carrot, it will eat it.
- Each rabbit is going to watch for wolves in their scan range neighborhood. If a rabbit detects at least one close wolf, it displays a warning message on the graphic display window.

10.1 Rabbit hops toward a carrot when clicked

To implement this behavior, add the following method to your Rabbit class.

```
/**
 * Gets the first carrot in the patch. If the carrot is in the Rabbit
 * hopStep range, the rabbit eats it. It sets its position to the (x,y)
 * position of the carrot and the carrot will be removed from the Carrot Patch.
 * Otherwise, the rabbit moves one hopStep towards that carrot. If no carrot
 * found (meaning Carrots.getFirstCarrot() returns false),
 * the rabbit does nothing.
 */
public void hopTowardsCarrot() {
  // get the first carrot
  int[] carrot = Carrots.getFirstCarrot();
  if (carrot != null) {
    // TODO complete the implementation of this method

  }
}
```

To help you implement this method, we recommend that you review the `Carrots.getFirstCarrot()` and `Carrots.remove(int)` methods. Besides, we provide you in the following with a hint on how to calculate the new position (newX, newY) of an object located at (oldX, oldY) position when moving towards a destination located at (destinationX, destinationY) position with a moveDistance. We denote the euclidian distance between destination and object before moving

by `d`. In our case, if a rabbit is going to make one hop towards a carrot as destination, hopStep is the moveDistance, and newX, newY will be its new (x,y) position after performing the ONE hop.

$$newX = oldX + \frac{moveDistance * (destinationX - oldX)}{d}$$

$$newY = oldY + \frac{moveDistance * (destinationY - oldY)}{d}$$

$$d = (int)Math.sqrt(dx * dx + dy * dy)$$

where $dx = destinationX - oldX$ and $dy = destinationY - oldY$

Override the mousePressed() method in the Rabbit class

Once your `hopTowardsCarrot()` is implemented, override the the `mousePressed()` method in the Rabbit class. This will be a **partial** overriding as follows.

```
@Override
public void mousePressed() {
  // TODO
  // call the mousePressed defined in the Animal super class
  // call hopTowardsCarrot() method
}
```

Now, run your program, plant carrots, add at least one rabbit, then try clicking on one rabbit and check whether it will make one hop towards the first carrot after each mouse click. If the rabbit reaches the carrot, it will eat it. It takes its position, and the carrot is removed from the patch. If you keep clicking on the rabbit (simple clicks without dragging it), the rabbit will hop towards the next carrots, and eats them one by one.

10.2 Override the Rabbit.action() behavior

We define the following action of a rabbit in the carrot patch. A rabbit will continuously scan the area around it watching for a threat (i.e. if there is at least one wolf located nearby). If so, it will react by displaying an alert message "WOLF!". We provide you in the following with the implementation of the Rabbit.action() method. To enable it, you must implement the `watchOutForWolf()` behavior.

```
/**
 * This method watches out for wolves. Checks if there is a wolf
 * in the Rabbit.scanRange of this Rabbit.
 *
 * @return true if the current rabbit is close to at least one wolf
 */
public boolean watchOutForWolf() {
  // TODO complete the implementation of this method

  // Traverse the processing.objects arraylist checking
  // whether there is a wolf which is close by Rabbit.scanRange
  // of this rabbit.

  return false;
}

/**
 * Watches out for a wolf and display a Warning message "WOLF!"
 * if there is any wolf in its neighborhood.
 */
@Override
public void action() {
  if (watchOutForWolf()) {
    // this.setScaredImage();
    processing.fill(0); // specify font color: black
    processing.text("WOLF!", this.getX(), this.getY() - this.image.height / 2 - 6);
  }
}
```

Note that you do not need to call Rabbit.action() method. It is already called in Animal.draw() method. Polymorphism is MAGICAL!

To help you assess the correctness of your Rabbit.watchOutForWolf() method, we provide you with the implementation of these test methods. You can use them verbatim in your CarrotPatchTester class and call them from the CarrotPatchTester.setup() method.

11 Implement the Wolf action in the CarrotPatch

In our carrot patch, we have HUNGRY wolves. Each wolf is looking for rabbits in its scan range to eat it. To implement this behavior, follow the following steps.

11.1 Implement eatRabbit() method

Add the following method `eatRabbit()` to the `Wolf` class and implement it.

```
/**
 * Moves to the position of the specified rabbit passed as input, and eats it.
 * The eaten rabbit will be removed from the patch and he number of eaten
 * rabbits by this wolf is incremented by one.
 * @param rabbit rabbit to eat by this wolf
 */
public void eatRabbit(Rabbit rabbit) {
  // if the mouse is over the current Wolf, release it so the Wolf can move
  // ahead to the position of rabbit and eat it.
  if (isMouseOver())
    this.mouseReleased();
  // TODO
  // 1. set the position of the current Wolf to the position of the rabbit
  // 2. remove the rabbit from the patch
  // 3. increment the number of eaten rabbits by one
}
```

To help you assess the correctness of your Wolf.eatRabbit() method, we provide you with the implementation of this test method. You can use it verbatim in your CarrotPatchTester class and call them from the CarrotPatchTester.setup() method. Feel also free to define further test scenarios.

11.2 Override action() method

Next, override the `Wolf.action()` method as follows.

```
/**
 * Defines the action of this wolf in the carrot patch. This wolf lookup for
 * rabbits in its neighborhood (Wolf.scanRange) and eats the first found rabbit
 * only. This method also displays the number of rabbit eaten by this wolf if any.
 */
@Override
public void action() {
  // TODO
  //Traverse processing.objects arraylist, search for the first rabbit which
  // is close to this wolf with respect to Wolf.scanRange, and eats it.
  // If no rabbit is found in the neighborhood, nothing will be done.

  if (rabbitEatenCount > 0)
    displayrabbitEatenCount(); // display rabbitEatenCount
}
```

```
/**
 * Displays the number of eaten rabbits if any on the top of the Wolf image
 */
public void displayrabbitEatenCount() {
  processing.fill(0); // specify font color: black
  // display rabbitEatenCount on the top of the Wolf's image
  processing.text(rabbitEatenCount, this.getX(), this.getY() - this.image.height / 2 - 6);
}
```

12 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through gradescope.com. The only NINE files that you must submit include: `CarrotPatch.java`, `Animal.java`, `Rabbit.java`, `Wolf.java`, `PlantCarrotsButton.java`, `AddRabbitButton.java`, `AddWolfButton.java`, `ResetButton.java`, and `CarrotPatchTester.java`. Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline. Finally, the third portion of your grade for your submission will be determined by humans looking for organization, clarity, commenting, and adherence to the CS300 Course Style Guide.

Extra Challenges

Here are some suggestions for interesting ways to extend this CarrotPatch graphic application, after you have completed, backed up, and submitted the graded portion of this assignment. **No extra credit will be awarded for implementing these features**, but they should provide you with some valuable practice and experience. DO NOT submit such extensions via gradescope.

1. **Suggestion 1** – Try updating the way how a rabbit hops towards a carrot in the patch. Instead of moving forward the first non null position in the Carrots.carrots array, the rabbit can move forward the closest carrot to its current position.

2. **Suggestion 2** – Try expanding the possible actions performed by a wolf or a rabbit in the carrot patch. Use your imagination!

3. **Suggestion 3** – Try creating new animal classes with different data fields and different behaviors. For instance, you can define birds which tweet and jump on the trees or fly in the sky. Use your imagination!