EE 422C HW 5 Critter Simulator [Part II]

1 Instructions

1.1 Team Policy

This project is *optionally* a pair-programming project. **Read the full assignment instructions on Canvas before you continue reading, and again before you start coding.**

1.2 What to do

In this assignment, you'll add a more modern controller and view module based on the **JavaFX** framework to your *Critter* simulation engine (Assignment 4). We'll make the *Critter* model a little more interesting (just a little) so that *Critter* subclasses can be more interesting; if time permits, we'll let you show the class your fancy GUI (just for fun and bragging rights).

A large part of your grade will be based on the visual presentation of your JavaFX-based interface. Also note that we will not be imposing specific requirements on how you implement your interface. Rather, the TA will be working off a checklist of required features and will score your project partly on whether you can accomplish everything on that list (in some fashion, we won't tell you what to do), and how attractive and effective (i.e., easy to use) your interface is.

2 Requirements

2.1 Model Components

`look()`

Basic Functionality

The model remains largely unchanged for this project. Given that the model was the dominant focus of the last project, that seems reasonable. However, you do need to introduce one new piece of functionality. You must implement the protected `Critter.look(int direction, boolean steps)` method. This method examines the location identified by the *Critter*'s current coordinates and moving one or two positions (for `steps = false` or `true` respectively) in the indicated direction (recall direction 0 corresponds to moving along the x axis, 1 corresponds to moving diagonally along both the x and y axes, etc. (refer to Part 1 for documentation)). If the location is unoccupied, then `look()` returns `null`. If the location is occupied, then `look()` returns the `toString()` result for the Critter in that location. In either case, the *Critter* invoking `look()` will pay the `Params.LOOK_ENERGY_COST` energy cost.

In `doTimeStep()`

When implementing `look()` you must respect the simulation rule that all Critters move simultaneously during their `doTimeStep()`. So, if a *Critter* invokes `look()` during `doTimeStep()`, then the result of calling `look()` is based on the old position of the *Critter* (before it moved this time step, if it has) and based on the old positions of all the other *Critters* (before they move this time step).

In `fight()`

Conversely (and consistent with Part 1), if a *Critter* invokes `look()` during its `fight()` method, then the result of calling `look()` is based on the most current up-to-date information. Any *walk/run* effect is processed immediately in the arbitrary order that you process them (any sequence of `fight()` actions is fine – this is the same requirement as Part 1). A *Critter* can *look* as often as it wishes in a time step.

When to Remove Dead Critters

One source of confusion might be to decide when to remove dead *Critters* from their location. If *Critters* die during `doTimeStep()`, remove them after `doTimeStep()`. If they die during fighting, remove them right away. If there are some corner cases for these rules, implement anything reasonable, and document your behavior in your **README** file.

2.2 View Component

The view component is completely rewritten for this project, and you may design this component (almost) anyway you wish. These are the requirements:

- The view component must be triggered by the `Critter.displayWorld()` static method.
- The view component must display Critters graphically using a **JavaFX** class such as *GridPane*. For this purpose, you may access objects in your *Main* class statically.
- The view object can be scaled however you see fit. The quality and flexibility of your scaling will be rated by the TAs. A well-scaled view will permit large worlds (`Params.WORLD_WIDTH` and `Params.WORLD_HEIGHT` larger than 100) to be displayed on reasonably sized screens (i.e., laptop computer screens).
- Each Critter in the simulation can select how it is viewed by overriding the following methods
    - `viewShape()` – returns a *CritterShape* value, see the `Critter.CritterShape` enumeration for possible values
    - `viewOutlineColor()` – returns a JavaFX `Color` value (see the JavaFX Canvas tutorial for demonstrations of possible colors)
    - `viewFillColor()` – returns a JavaFX `Color` value

The view method must correctly draw each *Critter* based on these values, the shape must be outlined using the outline color and the shape must be filled using the fill color. Note that by default, the two colors are the same.

- In addition to rating your view based on the quality of its scaling implementation, the TAs will give an overall “attractiveness and quality” rating as part of your score.

2.3 Controller Component

The controller will also be largely rewritten; however, you may be able to salvage and reuse parts of your controller. The controller commands themselves will remain the same. Your controller must be a JavaFX graphical user interface rather than a text based interface. Users will enter commands by pushing buttons rather than typing text. When evaluating your controller, the TAs will look for the following.

- Do you have the ability for the end user to create *Critters*? Can the user add new *Critters* to the simulation at any time (i.e., creating new Critters should not be limited to before the first time step – when you implement animation, you can and should disable *Critter* creation while *animation* is running, see below)? You must not pre-program the acceptable *Critter* types.
- Does the user have the ability to perform time steps? Can the user perform multiple time steps with a single button push? Is the view updated correctly after performing time steps – note that if the user asks to perform 100 time steps, then the view should be updated only after all 100 time steps have completed, not updated after each time step. Users should be able to (with one click) step the simulation by 1 time step, or by 100 time steps, or by 1000 time steps (configured and selected by the user). If the user can select other values for the number of time steps, that’s even better.
- Does the user have the ability to invoke their `runStats()` method? Do you have a panel where the results of `runStats()` is continuously being displayed (updated whenever the view is updated)? Can the user select which *Critter* class(es) have their `runStats()` methods updated? You may simply display all the Critters' stats all the time, for a small point penalty. By default (if the particular critter has no `runStats()` defined) is the `Critter.runStats()` base class method invoked each time the view is updated? Note that `runStats()` now returns a *String*, instead of being *void*.
- Does the user have the ability to set the random variable seed?
- For all of the above items, the less typing the user has to do to activate the required functionality the better.
- Is there an easy and obvious way to terminate the program (this requirement is probably trivial with a JavaFX GUI, but a quit button is a nice touch). For this assignment, you may use *`System.exit(0)`*.
- Users should be able to add *Critters* as long as the class files are available in the package folder. Do not hard code the Critter names in your files. *TestCritters* should also be addable.
- Finally, is the controller properly separated from the model? You should still use the same Critter functions as before (`worldTimeStep`, `createCritter`, `runStats`, etc). Recall that in the MVC architecture, we really want to keep each component as well separated as is practical.

2.4 Critter Subclasses

You must also update your *Critter* subclasses so that at least one *Critter* class that you write invokes the `look` method. You don’t have to invoke the `look` method in any particular way (you can call it from your `doTimeStep` or from your `fight` method), and you don’t have to invoke the `look` method every time that method is called, but there must be some circumstance under which your Critter uses the `look` method.

Project teams of two developers (i.e., working with a partner) **must** update two Critter classes so that one *Critter* class calls `look` from `fight` and will not call `look` from inside its `doTimeStep` function; the other *Critter* class calls `look` from its `doTimeStep` function but will not call `look` from `fight`.

All *Critter* subclasses that you write must override the newly required `viewShape` method and override one or more of the `viewColor`, `viewOutlineColor` and `viewFillColor` methods (you don't have to override all three). You may not use external image files to make icons for your `Critters`.

Your solution should be able to **add unknown Critter** files found in the same directory (which will also happen to be in the same package). This can be done in one of two ways – the user has to type in a valid *Critter*'s name into a text box, or you have to search for and find all valid *Critter* classes in the same directory and package as Main, and then display these as choices in a pull-down menu. There should of course be a *separate text box* for the number of these *Critters* that you want to add. The pull-down menu is harder to implement, but obviously *more elegant*. You **may not hard-code** *Critter* names into your *.java* file.

2.5 Animation

The simulation is well suited for animation. First, allow the end-user to select an animation speed. In each animation "frame", the world could perform *1 time step* **by default**. If the user increased the animation speed, the world could perform 2, 5, 10, 20, 50 or 100 time steps per frame (you can pick your own options if you'd like, these "speedup factors" are just to give you some ideas). Another animation option is to keep the 1 time-step/frame, but speed up the rate of refresh of the view window.

Once the user has set their animation speed, the user should be able to start animating by pushing a button. Pushing this button should disable all other controls in your controller except the "stop animation" button. While the animation is running, the controller invokes the requested number of time steps each animation frame, then calls the view to update the graphical Canvas for the world, and also calls the selected `runStats` method to display the currently selected stats. The simulation continues repeating this behavior every animation frame until the stop button is pressed.

2.6 Changes to `displayWorld()` and `runStats()`

For this assignment, `runStats` returns a String, and is not a void method. You may change your `displayWorld` to accept an Object as a parameter. The parameter can be the pane (such as a *GridPane* object) on which you draw your world. You can cast the Object parameter to the correct type within your new `displayWorld` method.

3 Extra Credit

This assignment has the unique opportunity of implementing additional features for extra credit.

Please finish implementing and testing all the above requirements before attempting to add anything.

Your grade is based primarily on those requirements and additional features are no excuse for not completing them. With that said, you are allowed to implement any additional feature within reason (i.e., it should be related to *Critters* and not any arbitrary piece of software). Here are some great ideas you can add to your project.

- Improve graphics
    - Better general GUI color scheme
    - Change the window background color or make it an image
    - Change the grid background color or make it an image
    - Randomly select image/color when a new world is created
    - Procedurally generate a tile background for the world. Aim to make it look as cohesive as possible.
    - Increase *Critter* customization. Additional shapes, more color options, add patterns, etc.
    - Runtime dynamic/scaling components. For example, the world scales with the window.

  - Add a different view. Vanishing point, Isometric (or any other axonometric view), or full 3D with a fixed camera (amazing if you manage to do this).
  - Zooming. Instead of having to always look at the whole world allow the user to view a smaller part of it.
  - Follow the *Critter*. When zoomed in, let the user select a Critter to follow.
  - Unique animation. Instead of only stepping the world also create actual animations for the *Critters*. For example, a circle that bounces or a triangle that spins.
- Increase functionality
  - Allow changing settings found in `Params.java` at runtime.
  - Adding Critters from a menu rather than typing their names
  - Unique `runStats`. Like creating a graph showing *Critter* population over time.
- Other
  - Add sounds or background music

This list is obviously not exhaustive but will hopefully give you enough ideas. We **encourage** everyone to **add an extra feature**. Please note that if a feature modifies functionality found in the main requirements the original functionality must still work as intended. For example, even if you can set parameters at runtime `Params.java` must still work.

The actual point values of specific features will be determined *after grading is done* and will **take everyone's individual features** into account.

4 Submission and Grading

- Turn in all the files that are required to test your project: `Critter.java`, `Main.java`, your own *Critters*, and any other files *you* created.
- Submit a **README.pdf** file with the following:
  - A description of your code and graphics, and it might include diagrams.
  - Any feature in your project implementation that you think is unusually good or did not meet the standard. Briefly describe any problem that you had and could not solve and *how you tried everything you can to fix it*. We will take your effort into account when grading.
- Submit a team plan with each of your roles, and your Git URL, if you worked as a team.
- For grading, each team will sit with the TA and demonstrate your code. For this purpose, the code that you turn in will be downloaded into the TA's computer; it will not be graded on your own machine. Both students have to be present for the checkout.
- Before the deadline, one of you should submit a zip file with all your solution files. This file should contain `Critter.java`, `Main.java`, your own *Critters*, and any other files *you* created. Zip your source folder and other files together, and rename this file **Project5_eid_eid2.zip**. Omit "_eid2" if you are working alone.

To make the zip file, make a folder named **Project5_eid1_eid2**. Put the files in there as per the diagram below. Then invoke the Linux/MacOS command (or do the equivalent in Windows):

```
$ zip -r  Project5_eid1_eid2.zip Project5_eid1_eid2
```

Just to be sure, move your zip file to a different location and unzip it. Make sure that the structure of the final zip file is as follows, when unzipped.

```
Project5_eid1_eid2/ (folder created by zip)
    src/
        assignment5/
            Main.java
            Critter.java
            Critter1.java
            Critter2.java
            ...
```

5 FAQ

Q1. Why is there starter code? Are we not using our previous code?

The starter code provides the method and enum declarations defined above. Essentially you need to **merge the new starter code with your existing code**. Do not implement the requirements using different method/enum headers.

The starter code will become available to you when you create a repo.

Q2. Are we allowed to use SceneBuilder/FXML?

Yes.

Q3. Are we allowed to Swing/AWT?

Swing/AWT is **NOT** allowed because it's an entirely different framework from JavaFX.

Q4: How can we find all the Critter subclasses at run-time?

Since the JVM only loads classes as they are needed, the only way to do this is to look at the files inside the working directory, isolate the .class files, and then isolate the classes that are critters. This has to be done manually. That is, there is no pre-existing Java method that does this for you. Use the instance method `list()` in the File class to get a list of all files in a directory, then use the instance method `isAssignableFrom`() in the *Class* class to check if the found classes are *Critters*.

Q5: How are we supposed to implement other CritterShapes?

JavaFX by default only has shapes for circles and rectangles. You have to manually implement the other shapes using **Polygon**. You have to implement all of them even if your *Critters* only use one or two of them.

Q6: How can we implement animation without everything breaking?

Using timers and schedulers will probably cause instabilities because they will be on a separate thread from your main GUI, which will be on the JavaFX thread. To avoid such instabilities, use an **AnimationTimer**. For even finer control over your animations, extend *AnimationTimer* and override the `handle`() method with your own implementation. This is one of the hardest parts of the assignment.

Q7: How can we output text to the GUI instead of the console?

You can **redirect** your outputs to any source using the `System.setOut(PrintStream out)` method. The easiest way would be to redirect it to a *PrintStream* made of a *ByteArrayOutputStream*. Remember to manually refresh the part of the GUI that is displaying the text.

Q8: What does scaling mean? Scaling of the windows or scaling of the critter world?

The latter. The size of the window is of little issue, and it can be fixed as long as it works on most reasonable resolutions. The Critter world however, should be **scalable**. For example your program should be able to handle worlds as small as 4x4 and worlds as large as 100x100 equally well. For full credit, your *Critters* should be visible at a reasonable size for different world grid parameters (such as 4x4 and 100x100). For an even better GUI and extra credit, make your grid dynamically scalable with *click-and-drag* window resizing resulting in resizing of the *Critters* and their grid.

Q9: Must both partners be present for grading?

Yes.

6 Submission Checklist

- ☐ Did you complete a header for **all** your files, with both your names and UT EID's?
- ☐ Did you mark the slip days used?
- ☐ Did you do all the work by yourself or with your partner?
- ☐ Did you zip all the files required to test your project into a zip file?
- ☐ Did you include your own *Critters*, after testing them in your system?
- ☐ Did you download your zipped file into a fresh folder, make sure that your directory structure is exactly what we asked for, and run it again to make sure everything is working?
- ☐ Is your package statement correct in all the files?
- ☐ Did you preserve the directory structure?
- ☐ Did you include a **README** pdf document and a **team plan** pdf?
- ☐ Did you go through this document and the Piazza posts (especially the ***faq*** tagged posts) just before submission?

Good luck and have fun!