

EE 422C HW 5

Critter Simulator (Part 2)

Due: See Canvas

This project is optionally a pair-programming project. Read the pairing instructions on Canvas before you start.

Here you'll add a more modern controller and view module based on the JavaFX framework to your Critter simulation engine. 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 run a simulation contest (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.

Model components

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.

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).

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.

One source of confusion might be to decide when to remove dead critters from their location. Remove dead critters that have died during `doTimeStep` after `doTimeStep`. After fighting, remove dead critters right away. If there are some corner cases for these rules, implement anything reasonable, and document your behavior in your README file.

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 (checklist items)

- The view component must be triggered by the `Critter.displayWorld` static method.
- The view component must display critters graphically using a JavaFX Canvas object. Other types of JavaFX objects are also acceptable. For this purpose, you may access objects in your `Main` class statically.
- The Canvas 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 critter worlds (`Params.world_width` and `Params.world_height` larger than 100) to be displayed on reasonable 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` valueThe 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.

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., making new critters should not be limited to before the first time step – if 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.

- Do you have the ability for the user 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.
- Do you have the ability for the user 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? By default is the `Critter.runStats` base class method invoked each time the view is updated?
- 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).
- Finally, Is the controller properly separated from the model? You should still use the same `Critter` functions as before (`doWorldTimeStep`, `makeCritter`, `runStats`, etc). Recall that in the MVC architecture, we really want to keep each component as well separated as practical.

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 circumstances 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, and one critter class calls `look` from its `doTimeStep` function but will not call `look` from inside `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).

Your solution should be able to add unknown `Critter` files found in the same directory and 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.

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).

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.

Submission and grading

- Turn in all the files that are required to test your project. Turn in your `Params.java`, but do not add or remove parameters from it.
- 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 usually good, or did not meet the standard. Briefly describe any problem that you had and could not solve.
- 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.
- Use the same ZIP file structure as part 1, but change the package to `assignment5`, and the ZIP file's name appropriately.

Before 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, move it to the Linux server, make sure that your directory structure is exactly what we asked for, and run it again to make sure everything is working?
- ☐ Does your code work correctly on Eclipse with Java 8 as well as on the ECE Linux server?

- ☐ 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!