EE 422C HW4
Critter Simulator (Part 1)

125 Points
Due: Thursday 3/22/18 at 11:59pm

1. Objectives:
We have several objectives for this project.

- You will work with an inheritance hierarchy that has an abstract base class. The abstract base class will have public, private and protected components, concrete methods and abstract methods, and both static and non-static elements – a little bit of everything. You'll make concrete subclasses of this class and write "object-oriented" code that operates on instances of the subclasses in a polymorphic fashion.
- We'll introduce you to the concept of the Model-View-Controller (MVC) software architecture. Our model will be a simple simulation. The controller in part 1 will be a text-based controller with very rudimentary commands entered from the keyboard (technically, commands will be read from System.in, which of course may not be a keyboard). The views for part 1 will similarly be very rudimentary and will consist of a text representation of the simulated world sent to System.out. During part 1, most of your effort will go into the model itself (i.e., writing the simulator). In part 2, you'll build a more interesting and useful view and controller component.

2. Summary:
Imagine a 2-D rectangular grid of fixed length and width. Each grid point can be described with a pair of co-ordinates (x, y). Imagine now that some of these grid points are populated by Critters (i.e. animals or Algae plants). As time progresses in steps, the Critters can (i) move around the world (ii) fight other Critters when they find themselves on the same grid location (iii) eat Algae (iv) reproduce and (v) die when they run out of energy. You will write a simulation model for this world in Java, where we specify the rules for the above five activities.

Here is how the simulation model runs:
(i) The program is started up through a `main()` by the user.
(ii) The user is provided a prompt where he/she enters text commands. The first command might be to add a specified number of Critters of a specific type to the world model.
(iii) The user can now (or at any time) use the `show` command to print a view of the world to the console.
(iv) The user can issue the `step` command to step through time a fixed number of times. The world autonomously evolves as time passes, because of the activities listed in line 27.
(v) The user can use the `quit` command to finish the simulation.

3. Instructions:

You may work in teams of two for this project. Each team should make only one submission to Canvas. All of the project source files MUST have the names and UTEIDs of both students in the header at the top of the file. There will be no exceptions to this policy on team projects. Collaborating on the project and failing to follow these instructions and will be treated as a violation of academic honesty.

You may form your own team by finding a partner, or you may work on your own. Please see the Canvas assignment page for instructions on how to form your team, and the deadline for doing so.

You must write a simulator that supports the functionality for `Critter` described below. Your simulator will be controlled with a text-based interface that accepts a few simple commands and produces a rudimentary representation of the world. All of your classes must be included in a java package called "assignment4". You must create a class `Main` inside this package, and the `main()` function for your simulator (i.e., the controller) must be inside the `Main` class.

You must complete the `Critter` abstract class. There are several functions required in `Critter` – some are static, some are protected and some are private. Please review both the `Critter.java` file and the description below. You must implement all of the methods defined in this class. You may not delete or change any of the fields or methods already defined for `Critter`. You may add additional methods or fields to `Critter` only if you make those new methods or fields private.

Note that the `Critter` class has one inner class called `TestCritter`. The `TestCritter` class is used to (1) implement the `Algae` critter, which is the primary source of food within our simulated world, and (2) to test your projects during grading. You must ensure that the setter functions in the `TestCritter` class work correctly with your implementation of the Critter class and the simulation that you build. You must also implement the other methods in `TestCritter` correctly for the grading to work. We might discover more methods that we need for grading, and we will tell you that later. You are free to add any other methods that you like in `TestCritter` to help your testing. We will not be calling those methods in our grading, of course, but they should not result in compile errors when we run your code.

As you implement the functionality for your `Critter` model, you may find that you want to create additional classes. All of your classes must be in the `assignment4` package. You must implement all of the functionality described below. However, we recommend that you build this project in stages. Suggestions are provided within the descriptions below of the form **[STAGE 1], [STAGE 2]** or **[STAGE 3]**. You may, of course, implement the functionality in any order that you wish; however, please keep in mind that our grading process will assume that you worked on the stages in order (i.e., that you completed all the **STAGE1** functionality before implementing **STAGE2**).

In addition to implementing the model, view, and controller for basic `Critters` such as `Craig` and `Algae` (two critters that are included in your project kit), you must implement at least two distinct additional `Critter` classes per team member (i.e. four for a team of two). Each `Critter` class must behave differently when modeled. Each `Critter` class must be in its own .java file. At the top of the java file, you must include a paragraph description in the comments that explains how this `Critter` class behaves in the world. The description should be sufficient for the teaching assistant to easily determine how each `Critter` class you create is different from every other `Critter` class.

4. Model components:

The model consists primarily of the `Critter` class, and subclasses of `Critter`. A `Critter` is a simulated life form that lives in a 2-dimensional world. `Critters` have `(x,y)` coordinates in an integer grid to describe their position in the world, and an energy value that represents the critter's relative health. These values are represented with private fields in the `Critter` class. When a `Critter's` energy drops to zero (or below) the critter dies and is removed from the simulation. You are provided with a `Critter.java` file that describes the minimum required functionality for your `Critter`. Please refer to the file for details regarding our expectations for your solution. You are also provided with a `Craig.java` file that implements a subclass of `Critter`. You should not modify this file. Your implementation of `Critter` should work with the `Craig.java` file provided to you.

5. Constant List:

There are a number of constants defined in the `Params` class. These constants are `static` and `final` variables that identify parameters for the simulation. You must use these parameter variables when implementing the simulation. The parameter values that your program is tested with may be different than the values provided to you. The parameters in this file include:

- `world_width` – horizontal size of the world (integer units), typical values are 100-1000. We promise not to use values larger than 10^5 in our testing. Will never be smaller than 10.
- `world_height` – vertical size of the world. Same range expectations and restrictions as `world_width`.

The coordinates in our world run from 0 (left edge) to `world_width – 1` (right edge) in the x dimension and from 0 (top edge) to `world_height – 1` (bottom edge) in the y dimension. This coordinate system was chosen to match the way most graphics libraries work.

The simulated world is a 2-dimensional projection of a torus. That means that the right-hand edge of the world is considered to be adjacent to the left-hand edge. Or, if you prefer, that the world "wraps around" in both the horizontal and vertical dimensions. When `Critters` move, if a `Critter` moves off the top of the world, you should relocate that `Critter` to the bottom, and similarly for the four edges of the world.

The model understands eight directions – up, down, left, right and the four diagonals. These directions are numbered such that the values roughly approximate the radians around a circle – i.e., as direction increases in value, we move counter-clockwise in angle. The 0 direction is straight right (increasing x, no change in y). The 1 direction is diagonally up and to the right (y will decrease in value, x will increase). The 2 direction is straight up (decreasing y, no change in x), and so forth. <u>We will not test your program with negative directions or with directions larger than 7</u>.

- `start_energy` – the amount of energy assigned to a `Critter` when the critter is created at the start of the simulation. Note that this value is not the same as the amount of energy a Critter will have when it is "born" as the offspring of another `Critter`. See below for details about reproducing Critters during a simulation run.
- `walk_energy_cost` – the amount of energy required to move one grid position in any one of the eight directions in one time step
- `run_energy_cost` – the amount of energy required to move two grid positions in any one of the eight directions in one time step
- `rest_energy_cost` – the amount of energy required per time step in addition to any other energy expended by the Critter in that time step, i.e., the energy spent just standing still.
- `min_reproduce_energy` – the minimum amount of energy that a Critter must have if it will reproduce. See reproduce below.
- `photosynthesis_energy_amount` and `refresh_algae_count` are specific to the `Algae` class. See the discussion of `Algae` below.

You may alter this `Params` class file during your testing, as we will eventually replace it with our own.

6. **Critter collection**: **[STAGE1]**

You must create and maintain a collection (e.g., `List`, or `Set`) of `Critters`. In this collection you should store a reference to all the `Critter` instances that are currently alive and being simulated. You can store your critter collection as a `static` data component of the `Critter` class, or you can create a separate `CritterWorld` class that stores the critter collection (and perhaps will store other information about the state of the critter environment). Note that it does not make sense within the MVC architecture for the critter collection (which is part of the model) to be stored within the `Main` class (which is the controller).

The controller will populate this collection by invoking the static `Critter.makeCritter()` function.

- `public static void makeCritter(String critter_class)` – create and initialize a Critter and install the critter into the collection and prepare the critter for simulation. The critter's initial position must be uniformly random

within the world, and the initial energy must be set to the value of the `Params.start_energy` constant.
If the random location selected for the critter is already occupied, the critter should be placed into that position anyway. The encounter between the two critters now located in the same position will be resolved in the next time step (provided both critters are still in the same position at the end of that time step, see below).
The type of critter is given by the argument `critter_class`. If `critter_class` does not exist or if `critter_class` is not a concrete subclass of `Critter`, then this function must throw an "`InvalidCritterException`".
To implement this function you will need to use the `Class.forName()` static method and the `newInstance` non-`static` method for the class `Class`.

7. Time Steps: [STAGE1 except as noted below]

Our simulation consists of a sequence of time steps. During each time step, the state of all `Critters` in the simulation is updated, new critters may be added, and critters may be removed (births and deaths). All of the core functionality of the simulator is associated with time steps. The `Critter` class has two methods for handling time steps. The public static `worldTimeStep` function simulates one time step for every `Critter` in the critter collection (i.e., for the entire world). The abstract `doTimeStep` function simulates the actions taken (if any) by a single critter as it goes about its life in the simulation. Note that subclasses of Critter will override the `doTimeStep` function so that each type of critter can behave in different ways (some will walk, some will run, some will stand still, etc).

During a `worldTimeStep` you must accomplish all of the following tasks:

- Invoke the `doTimeStep` method on every living critter in the critter collection. The phrase "living" critter is used here for completeness. Hopefully all the dead critters are removed from your collection when they die.
- Some critters will implement their `doTimeStep` function by (in addition to other actions) walking or running. All of these critters must be moved to a new position (see the description of the walk and run methods below). Once all critters have moved in the time step, if two or more critters are occupying the same (x,y) coordinates in the world (i.e., are in the same position) you must resolve the encounter between that pair of critters. At the end of that resolution, only one critter will be permitted in any position. See encounter resolution below. If more than two critters are in the same position, then you must resolve the encounters pairwise, but you may do so in an arbitrary sequence. For example, if A, B and C are all critters in the same position, then you may first resolve the encounter between A and B. If B remains alive and in the same position, then you may then resolve the encounter between B and C (and so on, if there are more than three critters).
- **[STAGE 2]** Some critters will implement their `doTimeStep` function by (in addition to other actions) spawning offspring (i.e., calling the reproduce method, described below). Once all critters have had their `doTimeStep` function called,

their movements applied, and all encounters resolved, then all new `Critters` are added to the critter collection. Note that if a new critter is located in the same position as an existing critter, you will not simulate an encounter. Any encounter will take place in the next time step (assuming the two critters remain in the same position).

- Once all of the critters have been updated, with their `doTimeStep` functions invoked, their movement and encounters resolved and any offspring created, you must cull the dead critters from the critter collection. Any critter whose energy has dropped to zero or below during this time step is dead and should no longer be part of the critter collection. Don't forget to apply the `Params.rest_energy_cost` to all critters before deciding if they are dead.

8. Walking and Running Critters: [run is a STAGE2 function, walk is STAGE1

During each time step, a critter may choose to invoke the walk or run function. These functions are nearly identical, with the only difference being that walk will move a critter one position in one of the eight directions, while run will move a critter two positions in the specified direction. Note that while running, the critter must move in a straight line (no zig-zags). Note also that a running critter will probably be charged more than twice as much energy as a walking critter. The walk method must deduct `Params.walk_energy_cost` from the critter that invokes it, and the run method must deduce `Params.run_energy_cost` from the critter that invokes it. Since these methods are so similar, you might want to minimize your code by sharing stuff between these two. There will also be `look` functions added later that can further reuse your code.

There are two critter methods that can call the walk and run methods. Most critters will invoke the movement method directly from their `doTimeStep` function (the `Craig` critter has this implementation). When invoked from this method, you must update the energy for the `Critter` and calculate its new position. Recall that you will not check for encounters until after all critters have moved. That means that two critters may temporarily be located in the same position (`Critter` A moves on top of `Critter` B, but then `Critter` B moves out of that position during the same time step) and/or that two critters may move "through" each other (`Critter` A is directly to the left of `Critter` B, `Critter` A moves one position to the right, `Critter` B moves one position to the left). In neither of these situations will you simulate an encounter.

[STAGE 3] Note that critters cannot move twice from within the same `doTimeStep` function. If a `Critter` subclass calls walk and/or run two (or more) times within a single time step, you must deduct the appropriate energy cost from the critter for walking/running, but you must not actually alter the critter's position. `Critters` can die in this fashion.

[STAGE 3] `Critters` may also invoke `walk` or `run` from the `fight()` method. You will call `fight` when you are resolving an encounter (see below). A critter that does not want to fight can attempt to walk (or run) away. If a critter invokes walk or run from inside its `fight` method, you must charge the appropriate energy cost (whether you permit the critter to move or not). Then you will move the critter only if both of the following conditions apply.

1. The critter must not have attempted to move yet this time step. If the critter has previously invoked either its walk or run method this time step, then it will not move in fight (you'll still penalize the critter with the movement cost, however).
2. The critter must not be moving into a position that is occupied by another critter.

Only if both of those conditions apply will you move the critter. In this case, the encounter is resolved and no fight will take place between the critters in the encounter (see below). Note that if both critters attempt to move while resolving the encounter, and both critters attempt to move into the same position, you should move only one of the two critters (you can arbitrarily move one, "first" and then the second critter will not be able to move since that position is occupied).

9. Encounters Between Critters: [STAGE 2]

When two critters occupy the same position, an encounter must take place. Once all encounters are resolved, only a single critter can remain in any one position in the simulation world. Recall that your simulator must detect and resolve encounters only after every critter has had its `doTimeStep` method invoked (i.e., after every critter has had the opportunity to move). When you are resolving an encounter between critters A and B, you should proceed as follows:

1. Invoke the `A.fight(B.toString())` method to determine how A wants to respond. Note that A may try to run away. Note that A may die trying to run away (if it's very low on energy). If the fight method returns true, then A wishes to attempt to kill B.
2. Invoke the `B.fight(A.toString())` method to determine how B wants to respond. B may also try to run away. B may also die trying (both objects could die!). If fight returns true then B wishes to attempt to kill B.
3. After both fight methods have been invoked, if A and B are both still alive, and both still in the same position, then you must generate two random numbers (dice rolls, see below).
    a. If A elected to fight, then A rolls a number between 0 and `A.energy`. If A did not decide to fight, then A rolls 0
    b. If B elected to fight, then B rolls a number between 0 and `B.energy`. If B did not decide to fight, then B rolls 0

    The critter that rolls the higher number wins and survives the encounter. If both critters roll the same number, then arbitrarily select a winner (e.g., A wins).
4. If a critter loses a fight, then ½ of that loser's energy is awarded to the winner of the fight. The loser is dead and must be removed from the critter collection before the end of this world time step.

[STAGE 3] Recall that if there are three or more critters in the same position, then the encounters are resolved in an arbitrary sequence. If while resolving the encounter between A and B, both critters die or move out of the position, then you must not simulate an encounter between A or B and any other critters in that position. For example, if A, B and C are in the same position, and you simulate the encounter between A and B, and both critters run away and move into new positions, then C will not encounter anything this time step. On the other hand, if A and B fight, and B wins (and gains energy from A), then C will encounter (the newly strengthened) B critter.

10. **Rolling Dice:**

Critter provides a static function for generating uniformly-distributed random integers within a specified range. The name of this function is `Critter.getRandomInt` and you must use this function for generating any random numbers used in your simulation. This rule applies to subclasses of `Critter` as well. For example, `Craig` calls `Critter.getRandomInt` as part of its `doTimeStep` function. Generating random numbers using any other method is disallowed for this project (We're worried that you might have trouble making your simulation repeatable if we don't constrain how random numbers are produced, so we're putting this restriction in the hopes that it will make your lives easier in the long run).

11. Reproducing Critters: [STAGE 2]

Concrete subclasses of `Critter` may invoke the `reproduce` function. They can call this function from either their `doTimeStep` function or from their fight function. In order to call reproduce, the critter must first create a new `Critter` object (a new instance of a concrete subclass of `Critter`) and pass a reference to this object to the reproduce method. When that happens you must:

- Confirm that the "parent" critter has energy at least as large as `Params.min_reproduce_energy`. If not, then your reproduce function should return immediately. Naturally, the parent must not be dead (e.g., did not lose a fight in the previous time step), but you should have removed any such critters from the critter collection and/or set their energy to zero anyway.
- Assign the child energy equal to ½ of the parent's energy (rounding fractions down). Reassign the parent so that it has ½ of its energy (rounding fraction up).
- Assign the child a position indicated by the parent's current position and the specified direction. The child will always be created in a position immediately adjacent to the parent. If that position is occupied, put the child there anyway. The child will not "encounter" any other critters this time step.

New "child" critters created during a time step are not added to the critter collection until the end of the time step. They cannot prevent critter from walking (e.g., a critter wants to walk away from an encounter, that critter cannot move into a position that's already occupied by regular critter, but can move into a position occupied by a "newborn" critter), and the new children cannot encounter any other critters this time step. All new children will begin their existence within the simulated world in the next world time step. Note that the parent's reduction in energy happens immediately, however.

12. **The Algae and TestCritter Subclasses**:**[STAGE 2]**

`Algae` is a special critter type that can "cheat" – it can photosynthesize and is permitted to spontaneously appear within the simulated world. Essentially, Algae acts as the food supply for the other critters in the simulation. The Algae class is partially implemented for you. The current implementation is based on the inner class `Critter.TestCritter` which has three "setter" methods defined. As you implement your `Critter` class, you must ensure that these setter methods continue to work. For example, if you create an external data structure to represent the world "grid" (e.g., a two-dimensional array of `Critters`), then the `setX_coord` and `setY_coord` functions must update that external data structure correctly. Also, if the `setEnergy` setter is used to make the critter's energy go to zero (or become negative), then you must "kill" the critter and remove it from the critter collection.

New `Algae` must be added to the world every time step. At the end of the time step, after all other activity has been simulated (all movements and encounters), use a loop to create `Params.refresh_algae_count` new `Algae`. Each new `Algae` will have `Params.start_energy` energy and will be assigned a random position. If the `Algae's` random position places the `Algae` in the same location as another critter, that is OK. Newly created critters can be "on top of" other critters in the time step where they are created, by the end of the next time step, however, the critters must move apart, or they must fight (even `Algae` will fight if placed into the same location).

13. View Component: [STAGE 1]

The view (and controller) for this phase of the project is extremely rudimentary. We won't even bother pulling the "view" from the `Critter` class. Instead, your view consists of implementing the public static `displayWorld` method. This function must print a 2D grid to `System.out`. Each row in this grid represents one horizontal row in the simulated world. Thus, there will be `world_height` such rows. Each row will have `world_width` characters printed in it. If a position in the world is occupied then you will print the `toString()` result for that critter in the corresponding row/column in your output. If a position is not occupied, then you'll print a single space.

You must also print a border around your text representation of the world. You must start and end each row with a vertical bar "|" character, and you must include a row of dash "-" characters at the top and at the bottom of your diagram. Finally, the corners of your diagram must have "+" characters. So, a small 5x5 world might look like this:

```
+-----+
| @  C|
|     |
|   @ |
| @   |
|C @  |
+-----+
```

Note that this world has 4 `Algae` critters and two `Craig` critters. Yeah, it's pretty lame, but we'll look into building better graphics in phase 2 of the project.

14. Controller Component:

The controller for this phase is almost as rudimentary as the view, and is entirely text based. You must use a Scanner object created in main() for reading from the keyboard. Only one Scanner object connected to the keyboard may be created in the whole program. The controller must provide the end user with a prompt, "critters> ". In response to this prompt, the controller will accept a line of input (tabs and spaces do not matter, but newline characters do, a newline marks the end of line). The following commands are supported. All commands are case sensitive.

- quit – **[STAGE 1]** terminates the program
- show – **[STAGE1]** invoke the Critter.displayWorld() method
- step [<count>] – **[STAGE1]** The <count> is optional (count is **[STAGE2]**). If <count> is included, then <count> will be an integer. There are no square brackets in this command, this notation is used simply to indicate that the <count> is optional. For example, "step 10000" is a legal command, as is "step". In response to this command, the program must perform the specified number of world time steps. If no count is provided, then only one world time step is performed.
- seed <number> -- **[STAGE2]** invoke the Critter.setSeed method using the number provided as the new random number seed. This method is provided so that you can force your simulation to repeat the same sequence of random numbers during testing.
- make <class_name> [<count>] – **[STAGE3, for stages 1 and 2, edit your main function so that 100 Algae and 25 Craig critters are always placed into the world when it starts, for STAGE3, the world should start empty]** as before, the <count> argument is optional. The command "make" must be provided verbatim. The <class_name> argument will be a string and must be the name of a concrete subclass of Critter. When this command is executed, the controller will invoke the Critter.makeCritter static method. The <class_name> string will be provided as an argument to makeCritter. If no count is provided, then makeCritter will be called exactly once. If a count is provided, then makeCritter will be called inside a loop the specified number of times. For example "make Craig 25" will cause Critter.makeCritter("Craig"); to be invoked 25 times.
  - Note: The String passed in to the command and to MakeCritter is the unqualified name of the Critter. Our starter code extracts the package name, and you should prepend it to the class name as necessary.

- stats <class_name> -- **[STAGE3]** Similar to make, <class_name> must be a string and will be the name of a concrete subclass of Critter. In response to this command, the controller will
  1. Invoke the Critter.getInstances(<class_name>) which must return a java.util.List<Critter> of all the instances of the specified class (including instances of subclasses) currently in the critter collection – you must write Crittter.getInstances, by the way, we didn't provide that for you.

2. Invoke the static `runStats()` method for the specified class. For example, if <class_name> were `Craig`, then your controller will invoke `Craig.runStats()` and will invoke this function with a list of all of the `Craig` critters currently in the critter list. See the note about converting unqualified names to qualified.

After processing the command, prompt the user for the next command. Naturally, if the command is “`quit`”, then the program simply exits.

15. Exceptions and Errors: [STAGE3]

If any exception occurs for any reason while parsing or executing a command, your controller must print one of the following error messages and continue executing.

- If a command is entered which does not match the list of commands above, then your program must print: “invalid command: “ and then print the line of text entered. For example, if I entered the command “exit now”, which is not a valid command, your controller must print the error “invalid command: exit now” on a single line.
- If an exception occurs during the execution of a command (e.g., `InvalidCritterException`, or an exception while parsing an integer), then your program must print, “error processing: “ and then print the line of text entered. For example, if the command, “`make Craig 10-`“ would result in a parsing exception because of the malformed 10- and must produce the output, “`error processing: make Craig 10-`“
- Note that any extraneous text or parsing error on the command line is treated as if an exception occurred (whether one actually occurred or not). So, you treat “`make Craig blah`” the same way you treat “`make Craig 10 blah`”

16. Code Style:

You should have `Javadoc` style comments for all public, protected, and private methods in your code that you have written or modified. There is no need to add `Javadoc` comments to methods that already have such comments. Use good style, and provide comments, braces, blank lines, and good variable names throughout your code.
Convert your comments to Javadoc html files (see Eclipse documentation), and submit these HTML files in a `docs` folder along with the rest of your submission. We want single page html files for each class – if that is not possible, contact us. In any case, this part's format is somewhat flexible, as we will be grading these by eye. Don't convert the html files to PDF before submission.

17. Grading:

We will be using a combination of `JUNIT` testing and running your main for grading. We will also be inspecting your code by eye. `We` will be using a Linux server for our scripts, but might switch to Eclipse, particularly in case of problems encountered with Linux. It is your responsibility to see that your code works in both environments. We will explain later how to run our JUNIT tests on the Linux server environment.

18. Presubmission Testing:

We have provided two test case files. Please follow the instructions on how to download them to Eclipse and run them.

19. Submission:

- Check in your files regularly into Git. We expect at least 4 substantial check-ins from each team member.
- Each team should also provide a document `team_plan.pdf` describing the work done by each of you. This document must include your Git repository URL. Use the starter files provided on Canvas.
- Each team should also provide a `README.pdf` document describing your code structure.
    - Did you create any new classes, and if so, what fields and methods are in it?
    - What is the data structure that you used to hold your Critters?
    - Be prepared to have a paper copy of this document during the recitation section of the week the assignment is due.
- Name your critter source files `Critter1.java`, `Critter2.java` etc., and include header comments with descriptions. Your `toString()` for these critters should be `1`, `2` etc. I know this is not imaginative, but we need it for our grader.
- Before submission, make sure that your main is cleaned up, so that it produces no output to the console, and the Critter world is empty.
- Do not submit My`Critter1.java, MyCritter6.java` etc. that we supply to you.

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 (maybe initially called `Archive.zip`) `Project4_EID1_EID2.zip`. Omit `_EID2` if you are working alone.

To make the zip file, make a folder named `Project4_EID1_EID2.` Put the files in there as per the diagram below. The invoke the Linux/MacOS command (or do the equivalent in Windows):

```
zip –r  Project4_EID1_EID2.zip Project4_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:

```
  Project4_EID1_EID2/ (folder that is created by zip)
      README.pdf
      team_plan.pdf
      <other non-code files>
      docs/
      src/
          assignment4/
              Main.java
              Critter.java
              Critter1.java
              Critter2.java
              ...
```

Good luck and have fun!

20. FAQ:
See the separate document on Canvas.

21. Before submission checklist:

- ☐ Did you complete a header for *all* your files, with both your names and UT EID's?
- ☐ Did you do all the work by yourself or with your partner?
- ☐ Did you zip all your new or changed files into a zip file? Did you remember not to include the unchanged files that we provided?
- ☐ Did you remove or comment out all the features that you added for testing that violate the rules of submission?
- ☐ 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? This is not optional.
- ☐ 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 PDF document describing what each of you did on this project?
- ☐ Did you include a PDF document with your code structure?
- ☐ Did you include Javadoc files?