README.md

Assignment 4 "Critters"

Components

This lab consists of several components, including components given as part of the assignment, and components that we created to fulfill the assignment requirements. We added substantially to most of the components given to us, and created non-trivial components in order to complete the assignment. Many of the additions to the given components were to meet specific assignment requirements, namely that most additions must be private.

The components given to us were:

- Algae.java
- Craig.java
- Critter.java
- Main.java
- Params.java

The components we created were:

- CritterWorld.java
- InvalidCritterException.java

In the next sections we will briefly describe each component and the major data structures and methods inside of it, excluding small details such as getters and setters.

Components Given

Algae and Craig

Algae.java and Craig.java represent critters within the program. They are explicitly not to be modified, and will be used for testing. Our game must add Algae at each world time step, and must support the addition of Craigs.

Critter.java

Critter.java presented the largest challenge. It is supposed to act as an abstract class to implement additional specific critters. The assignment required that we add no methods or variables to the class that were not private to the class. The class also contained an inner class that provided a few getters and setters that must work with our implementation.

We attempted to make the Critter class self-contained in the sense that it contained functions and variables that the critter would need to access. There were a few instances where our model could not access information about the world state, due to it being private inside Critter. This led to a few back doors inside TestCritter to access private world information that was required to be kept inside Critter. For instance, each critter maintains a list of the world population, a list of the offspring created in the world, and other methods that needed to be accessed by the model.

The highlights that were added to Critter include the following:

- `boolean hasMoved`

A flag to determine if the Critter has already moved in this world time step. A Critter is only allowed to move once, and moving twice should still deduct energy, but not move the Critter.

- `private static void resetHasMoved()`

  This method is implemented here in order to be able to access the `hasMoved` flag.

- `private void updateMap(List<Integer> coords)`

  This method takes x and y coordinates and updates them in a virtual map of the world state inside our controller CritterWorld.

- `private void removeFromMap(List<Integer> coords)`

  This method serves a similar purpose, removing a Critter from the virtual map in the controller. It is here for the same reason, namely a Critter's coordinates are kept private.

The highlights that had to be implemented in Critter but were given as starter code include the following:

- `public void walk (int direction)`

  This function determines the steps needed when a Critter wanted to walk.

- `public void run (int direction)`

  This function determines the steps needed when a Critter wanted to walk. This function basically just implements walk twice.

- `public void reproduce (Critter offspring, int direction)`

  This function is passed in a Critter and implements the requirements for a Critter to reproduce. It changes the energies as required, and queues the offspring to be added later.

- `public static void makeCritter(String critter_class_name)`

  This function creates and initializes a Critter subclass. The parameter must be the unqualified name of a concrete subclass of Critter. If not, and InvalidCritterException must be thrown.

- `public static List<Critter> getInstances(String critter_class_name)`

  This function returns a list of Critters of the specified sub-class in the parameter. The function uses reflection to compare the parameter against the available Critter sub-classes in the package, and returns a list with the specified critters if successful. If the parameter does not match any of the avaible Critter sub-classes, then the function throws an InvalidCritterException.

- `public static void runStats(List<Critter> critters)`

  This function returns statistics about the critters specified in a list. The function can either be overridden by sub-classes of Critter such that the sub-classes can display their own statistics of interest. However, if the function is not overridden by a sub-class of Critter, then the Critter class will display the default statistics for the list of critters as provided by the instructors of the course.

- `public static void worldTimeStep()`

  This method ended up being pretty complicated to deal with the JUnit tests. The tests try to directly set the Critter's coordinates instead of calling the implemented methods. We had to set up several loops and checks outside of any of the actual world time stepping just to ensure the coordinates were always in sync.

- `private ArrayList<Integer> parseDirection(int direction)`

  This method had to determine the direction a critter wanted to move, converting from an integer from 0 to 7 into a movement on the map.

Main.java

Main has to create the controller, and implement a parser. The parser makes up most of the class, and is relatively self-explanatory. A variety of commands had to be supported, and error messages had to be displayed when unsupported commands were input. The commands supported were:

- `quit`
- `show`
- `step`
- `help`
- `make`
- `step xx`

  where `xx` is an integer
- `stats xx`

  where `xx` is a Critter
- `seed`
- `show`

Params.java

Params.java simply holds some parameters for constructing the world. This should be part of the model, but is a separate given class in this project to help automate testing.

Components Created

CritterWorld.java

CritterWorld.java was created to serve as the model for the world, or the environment. Some parameters that are aspects of the model environment are private to Critter.java and must be implemented there, due to the assignment requirements. Therefore CritterWorld implements several data structures and methods that must work with the Critter class. To facilitate the model's ability to maintain and update information about the world, it extends the TestCritter class (the inner class of Critter.)

The highlighted data structures of our model are:

- `protected static ArrayList<ArrayList<ArrayList<Critter>>> virtual_map`

  This data structure allows the model to keep track of where all the critters are at any time. The outer two ArrayLists form a 2d grid, and the innermost ArrayList allows for more than one Critter to occupy a spot at the same time. Simple ArrayList methods allow determining information about a specific grid location to be implemented very simply with the innermost ArrayList. This data structure is dynamically updated every time a Critter moves, runs, or reproduces.
- `protected static Queue<Critter> new_critters`

  This data structure holds offspring that result from reproduction until the end of each time step, when they are added in one by one.
- `protected static Queue<ArrayList<Integer>> conflicts`

  This data structure is used to check and resolve conflicts. Anytime a Critter changes position on the virtual map, there is a check on the number of critters in that position. If it is more than one (the Critter that just moved there) then the coordinates are put into an ArrayList of length two, and the potential conflict is logged in this Queue.

The highlighted methods are:

- `public static void resolveConflicts()`

  This method is run at the end of each world time step. Keeping in mind that the conflicts logged in the Queue are only potential conflicts and that Critters may have moved, this method empties out the conflicts list one by one. For each potential conflict, it is verified that the conflict exists, then lets the Critters try to run away. Then it checks that the Critters are still alive, and that the conflict persists. In this case, they proceed to fight by rolling "die". The process continues until the conflicts Queue is empty.

- `public static List<String> getClassList(String myPackage)`

  This method uses the JVM class-loader in order to obtain the file path of the source directory. The method scan all files for .class extensions and returns a list of all of the Classes available in the directory. This proved to be very useful in order to execute functions dependent on reflection.

InvalidCritterException.java

This class allows us create an exception to specify that an invalid Critter sub-class was attempted to be utilized or constructed when the class does not exist in the src/ directory of the project.