The Race for Artificial Life

Assignment 10
Due Monday, 20 November
CSC 062: Fall, 2017

Objectives

- To gain experience with threads and race conditions.
- To use `synchronized` and locks in Java.

Description

Computers are often used to simulate real world systems. Some simulations are constructed as part of computer games. Others are created for the more serious goal of predicting or understanding the behavior of the simulated system. For example, computer programs are used extensively to simulate the behavior of the atmosphere in hopes of predicting the weather.

The field of *artificial life* `https://en.wikipedia.org/wiki/Artificial_life` uses computer simulations to explore the behavior of living systems. For this assignment, we will have you construct a simple example of such a "living" simulation.

The scenario and model we will be using come from the text "Turtles, Termites and Traffic Jams" by Mitchel Resnick. In his book, Resnick uses simple computer simulations to illustrate the idea that the members of a large population may appear to exhibit cooperative or coordinated behavior when they are actually functioning quite independently and following very simple rules that dictate their individual behaviors.

One of the examples in the text involves the simulation of termites moving wood chips about in a simulated world. We would like you to construct a simulation similar to Resnick's termite program.

The world to be simulated starts out with a large number of wood chips and a somewhat smaller number of termites randomly spread about. The termites then begin to wander about completely aimlessly. At each step, a termite may move up, left, down, right, or diagonally regardless of the direction of its previous steps. If a termite happens to bump into a wood chip, it picks it up. It carries the wood chip around as it wanders until it bumps into another wood chip. At that point, it gets the urge to put the chip it is carrying down, but it doesn't do so immediately. Instead, it continues to wander around until it is no longer standing on top of another wood chip. Then, it puts its own chip down on this empty spot and wanders off randomly to repeat this process.

The surprising thing about this simulation is that although the action of the individual termites is random, the result of their combined work appears to be coordinated. As time progresses, the termites manage to gather all the wood chips into larger and larger piles.

Your program to simulate this world will be quite simple. The termites and wood chips will be represented on the screen by small (6x6) filled squares. You wood chips should be black, while the termites are red when they are not carrying a wood chip and green when they are carrying one.

The world in which the termites and wood chips reside will be divided into a grid of rows and columns. To keep track of the chips, we will use a two dimensional array of Cells. Each cell keeps track of its row and column and, if it contains a wood chip, a Rectangle corresponding to the position and size of the wood chip. If it does not contain a wood chip then the instance variable for the rectangle should have the value null.

We will keep track of the termites by putting them into an ArrayList as they are generated. They will be self-animated. To accomplish this, you will define the Termite class to extend Thread.* When a Termite is created, it will be added to the ArrayList and then displayed on the screen the next time the paint method in the TermiteWorld class executes. Once started, the termite's run method will randomly move it around the screen and also move any wood chips it finds following the rules described above. To do this, each termite, when created, will have to be passed the array used to keep track of the wood chips.

We will discuss later in this document how race conditions can get in the way of this simulation.

*We won't bother to use the ForkJoin Framework as we won't be dynamically creating or destroying threads.

Classes

There are three classes involved in this simulation. Two of them, `TermiteWorld` and `Cell` are given to you, while we provide a skeleton for the third, `Termite`.

TermiteWorld

`TermiteWorld` controls the simulation. Technically it extends `JPanel`. The main method of the class creates a `JFrame` (window), creates a `TermiteWorld` object, and then adds that (as a panel) to the `JFrame`.[†] The constructor for `TermiteWorld` fills in the two-dimensional array of `Cell`s, creates the termites, and then starts the termites. Only a fraction of the cells contain wood chips. A cell with a wood chip is created by passing `true` as a parameter in the constructor, while `false` indicates it is empty. The wood chips and termites are distributed randomly over the array. The termites are also added to an `ArrayList` in order to make it easier to draw them on the screen in the `paint` method.

The paint method draws the cells with wood chips in black and then adds the termites. Termites not carrying anything are shown in red, while those carrying chips are shown in green.

This class is provided for you, though you will have to modify it when you handle the issues with race conditions.

Cell

The `Cell` class represents a 6 by 6 square on the screen. It has instance variables for the row and column and the contents, which will be either a rectangle or null. Methods are provided to determine if the cell has a chip (`isEmpty`) and to return the cell contents (`getContents`) whether it is empty or not. There are also two methods that are used to change the contents of a cell. Method `takeChip` removes the rectangle in the cell and replaces it by `null`. It also returns that rectangle. If the cell was empty at the time of the call an exception will be thrown. The method `putChip` replaces a `null` value of contents by the rectangle passed as a parameter. It will throw an exception if there already is a chip in the cell.

The class is provided for you, though you will have to modify it when you handle the issues with race conditions.

Termite

The `Termite` class extends `Thread` and represents a termite that wanders over the screen. While the termite knows what row and column it is in, there is no rectangle associated with it as an instance variable. Instead the `paint` method in `TermiteWorld` will create the rectangle on the fly as it is painting the screen. This should make life a bit easier for you.

The `move` method is responsible for moving the termite one cell in any direction (including diagonally). You can simulate a random move by picking random integers between -1 and 1. Add one of these random number to the row number and another to the termite's column number. Note that this approach allows both diagonal moves and the possibility of occasionally not moving at all. You will have to deal with the possibility that a random move will place a termite outside the boundaries of the simulated world. You can deal with this in one of two ways. You can make the world "wrap around" so that if a termite walks off the right edge it appears on the left edge (and similarly for the top and bottom). To do this, use the % operator when adding to row and column numbers. Alternately, you can treat the boundaries as obstacles through which the termites can not walk. To do this, you will need to add if statements that test each move and reject it if it would require walking through a boundary.

We have provided code at the end of the `move` method in the start-up code to trigger a repaint of the screen after the termite moves and then to pause a little bit so that the animation is slow enough for you to see.

We have also provided methods `gotOne` – indicating whether the termite is carrying a wood chip, as well as `getRow` and `getCol`. These three methods are used in the paint method in `TermiteWorld`, so please do not change them.

[†]It turns out that `JPanel`s are better for animations than `JFrame`s alone, as there is less flickering.

Your main work will be to write the `run` method so that the termite can pick up and move wood chips. Here are the rules:

1. The termite should make moves until it finds itself on a cell with a wood chip.
2. It should then take the chip.
3. It should then move until it finds a space with another chip.
4. It should then move until it finds a space that is empty, and drop the chip there.
5. Having succeeded in getting rid of the chip it moves until it finds an empty space and then starts over again at 1.

This program is very different from the others you have constructed in this class. It does not involve any user interaction.

There's Trouble, Yes Trouble, Right Here in Termite City!

If you've followed the instructions above, you should have a lovely simulation running. At the beginning the chips are distributed randomly, but after some time (perhaps hours) you will be able to see that there are piles of chips forming, which was the goal of the simulation.

However, if you are very unlucky, you may see a message in the console indicating that you had a null pointer exception. Most of you, however, won't see that message, which is too bad, because this program definitely has a race condition.

Imagine that two termites are on the lookout for wood chips and both run into the same wood chip at roughly the same time. Each will notice the wood chip there, but one will actually grab the wood chip first and replace it with null. When the second one grabs it, they will find only null there, and this will result in the thread crashing. That is, an exception will be thrown when executing `takeChip` is executed. A similar problem can arise if two termites attempt to drop a chip in the same (initially empty) cell.

You can make this problem much more likely if you insert the following code:

```
try{
    sleep (30);
}catch (InterruptedException exc) {
    System.out.println("sleep interrupted!");
}
```

just before you execute the `takeChip` method on the cell containing the wood chip.

Aside from the null pointer exceptions, another sign of the problem is that you will start seeing lots of dead termites on the screen as the thread moving them will have crashed.

Part 2 of this assignment is now to fix this problem by eliminating the race condition where two termites could be changing the contents of a cell at the same time.

You can fix the problem in `Cell` by making all of the methods synchronized. That is a good beginning, but it won't solve all of the problems. Now you have to worry about the gap between when a termite has noticed a wood chip and when it picks it up. I suggest you create synchronized blocks (using the cell being examined as the lock) to surround code that checks to see if the cell has a chip and then picks it up. Similarly you should create a synchronized block that encompasses the code that checks the cell is empty, and then drops the wood chip. Note that you might have to do some rewriting of your code in order to fit in these synchronized blocks correctly.

Finally, there is another possible problem with the `paint` method in `TermiteWorld`. When drawing the termites, we first check if the cell is non-empty, and if so, draw the filled rectangle. It is possible that between the checking and grabbing the rectangle, a termite might have grabbed the contents. Please guard against that by putting a synchronized block around those statements.

Having done this, you should no longer have any race conditions (and hence no runtime exceptions) disrupting your artificial world of termites and wood chips.

Extra Credit

There are a number of ways you could extend this project. As described, we intend the interface to the basic program to be simple (i.e. non-existent). You could easily add some sort of controls to specify the density of wood chips and termites before the simulation began, and perhaps add a slider to regulate how fast the termites move (i.e., change the pause time between moves).

Another form of extension would be to make the termite behavior a bit more complex. For example, it might seem more realistic to assume termites can somehow sense wood chips if they get close enough to them. What happens if you change your termites so that if they can see a chip within a few spaces straight ahead, they go right for it? Obviously there are many such "senses" you could add and experiment with.

Turn in

This week you have an extra day to work on the project. Turn in your entire project using the submit script as usual, but turn it in by midnight Monday.

Grading

You will be graded based on the following criteria:

Criterion	points
Termites move correctly on the screen	3
Termites pick up wood chips	3
Termites drop wood chip properly	3
Correct use of synchronized blocks	4
Quality of code	4
Appropriate comments (including JavaDoc)	3
Extra credit: controls	up to 2
Extra credit: more complex behaviour	up to 4