CSci 1933 Lab 2

September 22, 2015

1. Introduction

The purpose of this lab is to have you begin to write some basic Java code, and to introduce you to JUnit.

2. Unit Testing

- Many times, developers, companies, or other interests want to have some way to test and see if a program functions correctly. It is frequently infeasible to exhaustively test or verify a program to guarantee with mathematical precision that it will produce the correct results in all circumstances, but testing can be used to gain a degree of confidence in it.
- Frequently, tests are used to check methods against a variety of inputs for which the correct output has been determined by some other means (interpreted from a problem specification, computed by hand, etc.), and see if the method responds correctly.
- In this course, we will frequently use unit tests to see if your homework solutions and lab assignments function properly. Additionally, in later assignments, you are likely to write some unit tests yourself.

3. Game of Life

- In today's lab, you will implement the rule set for a simulation called Conway's Game of Life. We encourage you to look at the linked article to get a feel for the history and nature of this game.
- The basic idea of the game of life is to simulate the behavior of communities of living organisms. It consists of a grid (mathematically, an infinite one) made up of cells. Each cell is either /dead/ or /alive/. The simulation proceeds by moving through generations. In each generation, cells are made alive or dead by the following rules:
    - If a cell is alive, and has more than 3 living neighbors, it dies (as if it were overcrowded).
    - If a cell is alive, and has 0 or 1 living neighbors, it dies (as if it were lonely).
    - If a cell is dead, and has exactly 3 living neighbors, it comes to life.
    - In all other cases, there is no change — the cell stays in its current state, either dead or alive.

4. Your task

Your task is to implement the rule set for the game of life.
To complete this lab, do the following:

- In IntelliJ, create a new Java project, called Lab2.
- Add JUnit to your project. To do this, do the following:
    - Select “Project structure” from the “File” menu.
    - Go to the "Libraries" group, click the little green plus (look up), and choose "From Maven...".
    - Search for "junit" -- you're looking for something like "junit:junit:4.8.1" (latest stable option), and click OK.
- Download the following file, and import them into your project:
    - LifeRulesTest.java — Import this into your src/ directory.
- In your src/ directory, create a new class called LifeRules.
- In this class, create a public method, getNextState, which takes two parameters, a boolean currentState and an integer numNeighbors, and returns a boolean. Put in its body a single statement which returns the value of the currentState variable. You sh2284nguy2284nguy2284nguy2284nguy2284nguy2284nguy2284ould have the following:

  ```
  public boolean getNextState(boolean currentState, int numNeighbors) {
      return currentState;
  }
  ```

  The currentState parameter is the current state of a cell; true if the cell is alive, and false if it is dead. numNeighbors is the number of living neighbors the cell has. The return value is true if the cell is to be alive in the next generation, and false otherwise.
  The code that is currently in the method body is not enough to implement the rules of life, but that's OK — its purpose is to allow the code to compile so that we can run the tests and watch them fail before actually implementing the correct logic.
- Run the JUnit tests. Right-click on your src folder and choose "Run 'All Tests'". You'll see the JUnit results. Some of the tests should pass, and some should fail. Some tests pass because this default stay-the-same code happens to have the correct behavior for 3 of the test cases.
- Write code so that this method correctly returns the value of the cell in the next generation. For reference, the rules are repeated here:
    - If a cell is alive (that is, currentState is true), and has more than 3 living neighbors (that is, numNeighbors is greater than 3), it dies (return false).
    - If a cell is alive, and has 0 or 1 living neighbors, it dies.
    - If a cell is dead, and has exactly 3 living neighbors, it comes to life (returns true).

    - In all other cases, there is no change — the cell stays in its current state, either dead or alive.
- Once you have this code written, run the unit tests. When your code is correct, they should all pass.
- After completing the work above, you should implement an alternative rule set for Life. The HighLife variation of of Life just like the normal rules above, except that both 3 and 6 living neighbors will bring a dead cell to life.
    - Look at LifeRulesTest.java. It defines a variety of tests in testSomething methods. Within these methods, it uses a couple of helper methods:
        i. assertTrue takes a boolean value, and causes the test to immediately fail if the value is false.
        ii. assertFalse takes a boolean value, and causes the test to immediately fail if the value is true.

      Tests run by going through the code. If any assertion fails, then the test is immediately over with a result of "failure". If an error occurs, then the test is finished with a result of "error". If the test runs completely with no failures or errors, then the test passes.
    - Modify the LifeRulesTest.java code to test for the HighLife rules rather than the Conway rules. You'll need to add code to test that 6 living neighbors bring a cell to life, and remove code that tests that they leave it dead.
    - Run the unit tests. Some of them should fail again.
    - Modify your logic in LifeRules.java to implement the HighLife rules, and verify that the tests pass.

5. Configure and test Gradle

The next task is to make sure Gradle is properly installed and configured in your account. Gradle is a tool that we use for compiling and running Java programs from the command line you can read more about it on its main website. You will use Gradle to compile your program in Assignment 1 so it's critical to make sure it's working properly. Open terminal, type in `gradle{space}-v` and hit enter. if it prints out something similar to the following then it's correctly configured.

```
------------------------------------------------------------
Gradle 1.4
------------------------------------------------------------

Gradle build time: Monday, September 9, 2013 8:44:25 PM UTC
Groovy: 1.8.6
Ant: Apache Ant(TM) version 1.9.3 compiled on April 8 2014
```

Ivy: non official version
JVM: 1.7.0_79 (Oracle Corporation 24.79-b02)
OS: Linux 3.13.0-63-generic amd64

(If you want to install Gradle on your own computer, please follow the instruction here.)
You first need the build.gradle build file that Gradle uses to build the program. Download the file from Moodle site and save it somewhere where you know where it is, such as your desktop.

Next, open a new terminal window. Go into the directory containing the build.gradle. Once in this directory, type `gradle run`. Your session will look something like this:

```
AllenLins-MacBook-Pro:test allenlin$ gradle run
:run
Hello Gradle!

BUILD SUCCESSFUL

Total time: 14.666 secs
This build could be faster, please consider using the Gradle Daemon:
https://docs.gradle.org/2.7/userguide/gradle_daemon.html
```