Programming Project: GridWorld

Due dates:

Part I: June 28 at the beginning of class (hardcopy)

Part II: Jun 5 at the beginning of class (electronic submission)

Deliverables

In this project you will design and implement a class and application program for efficiently keeping track of objects (in this case "people") on a grid.

Part I: This is the design phase. You will write a document describing your design (organization of data and methods of access and modification) and argue that it meets the requirements of the project. You will submit a hardcopy in class. It does not need to include any Java, but an experienced programmer should be able understand how to translate it into a real program.

See below for the specs your design must meet and exactly what is expected for this part.

Part II: This is the implementation phase. you will write a class called `GridWorld` which implements the `GridWorldI` interface (given and which may not be modified).

You will also implement a simple interactive application program which exercises the class. This program will be called `GridWorldSim`

There is almost a 1-to-1 correspondence between user commands in the application program and methods in the class.

Your submission must include the files `GridWorld.java` and `GridWorldSim.java`. If you design other "helper" classes, you must also include them of course.

Problem Description

In the following we describe the behavior of the application program. A subsequent section gives the corresponding specification of the `GridWorld` class you will implement.

The program will keep track of locations of "people" (rerpresented by integers) in a "grid world." When a "World" is created, it is initialized by the following:

- `N`: the number of people initially in the world. The individuals are identified by integers $0..N-1$.
- `nrows`: the number of rows in the grid.
- `ncols`: the number of columns in the grid.

We refer to a particular entry in the grid as a *district* and refer to it by a row-column pair (i, j). Indices **always begin at 0** (for people, rows and columns).

When your program starts it will take these parameters (as specified on the command line) and, depending on the mode of the program, will either assign each person to a random district (this is the default mode) or assign all N people to district $(0, 0)$ (details on how these modes are specified appears later in this handout).

Once the program has set up the grid, it enters an interactive loop which supports four commands (**Note the runtime requirements!**):

- `members i j`: this prints out a list of all of the people currently living in district (i, j).

  Runtime requirement: If we use $N_{i,j}$ to indicate the number of people living in district (i, j) then we state the *runtime requirement* as $O(N_{i,j})$ – i.e., the operation must be **linear** in the specified district (*not* in N – the number of people in the world).

  For example, even though the world might have millions of people certain districts might be very sparsely populated with only a few people.

  The order in which the members are printed doesn't matter – **do not worry about sorting or anything like that!**

- `population i j`: this prints the **number of people** living in district (i, j).

  Runtime requirement: $O(1)$.

- `population`: prints N, the **number of currently living people** in the entire grid/world.

  Runtime requirement: $O(1)$.

- `move x i j`: this moves person x from wherever they currently are to district (i, j).

  Runtime requirement: $O(1)$.

- `whereis x`: this reports the district (i.e., the (i, j) pair) where person x is currently living.

  Runtime Reqirement: $O(1)$.

- `kill x`: eliminates person x. If there is no such person, an appropriate message is printed.

  Runtime Reqirement: $O(1)$.

- `create i j`: creates a new person, assigns a unique integer ID to the person and puts him/her in district (i, j). It reports to the user the ID of the new person.

  Runtime Reqirement: $O(1)$. (technically, this is an "amortized" requirement because of possible `ArrayList` resizing).

  Detail: You must not be wasteful in assigning IDs. Some examples:

  – Suppose you start with 10 people 0..9 and the user creates a new person. In this case, it makes sense to assign the ID 10 to the new person.

  – Now suppose that you start with people 0..9 and the user kills person 3, kills person 6 and then creates a new person. Since IDs 3 and 6 are now both available, you should assign one of those instead of 10. It **does not** have to be the minimum available ID – e.g., in the previous example, 6 is just fine even though 3 might also be available.

  Why the above requirement? As you may have guessed, since many of the operations have $O(1)$ time requirements, you will have to use IDs as indices into one or more array (or `ArrayList`). We want to avoid wasting space in such arrays by having lots of unused IDs and continually adding new IDs at the end resulting in possible resizing.

- `quit`: terminates the program.

Part I: Solution Design

In this part you will design a solution to the problem, but will not do any implementation. You should think of this as a preliminary design document.

The document will include the following:

(a) An informal description of what it means for an operation to take constant time including examples (not necessarily relating to this project) illustrating constant time and non-constant time operations.

(b) Solution description. This should include "boxes and arrows" diagrams (like we've used in lecture many times) clearly illustrating how you intend to organize the data as well as describe (along with appropriate pseudo-code) how you will accomplish each operation in the required time. A boxes-and-arrows diagram should make it clear how various data is accessed and modified. The pseudo-code can include English, but must be very clear and described in a step-by-step fashion – **not paragraph form!**. You will need to make a clear argument that your design meets the runtime requirements. Though you will not be writing real code at this point, it will probably help if you use variable names and class names to refer to your main data structures the types of individual data items.

Suggestion: you should probably have a class which encapsulates the entire world (independently of the command parser) and for each user command, there should be a corresponding method defined on this class.

Your writeup should **not** include any information on parsing user commands. What we want to see is what happens for each command once the command is determined.

Hint: doubly-linked lists might be handy!

Suggestion: Start by designing a straigntforward, functionally correct solution that may not meet the runtime requirements. Analyze the runtime of individual operations in this solution and *then* start looking for modifications/alternatives so the runtime requirements can be met.

Part II: Implementation

Now you will complete your implementation. This includes the implementation of the Java class `GridWorld` and the application program `GridWorldSim`.

Your code should be well documented and obey standards of good coding style.

Behavior of Application Program

Your executable/application will be called `GridWorldSim`. A synopsis of how your program will run from the command line is as follows:

```
java GridWorldSim [-norand] [-ncols <numcols>] [-nrows <numrows>] [-n <N>]
```

(If you are unfamiliar with command line arguments and/or running programs from the command line, see FAQ at the end of this document).

All four flags are optional and can appear in any order. If a flag does not appear, then the corresponding configuration parameter takes on its default value.

If the program is run without any command line arguments, it uses the following defaults: `nrows=5`; `ncols=5`, `N=10`. Additionally, the program runs initially assigns each person to a random district *unless* the `-norand` flag is specified.

Once the program begins, it enters the interpreter loop executing the commands described above. A sample session is below.

```
% java GridWorldSim
> members 3 4
    people: 3 7
> whereis 7
    district (3,4)
> move 3 0 0
    ok
> whereis 3
    district (0,0)
> members 3 4
    people: 7
> members 5 5
    error: value out of range
> whereis 10
    error: value out of range
> kill 7
    7 is dead
> kill 7
    no such person
> members 3 4
    people:
> population 3 4
    0
> create 1 1
    new person 7 created
> kill 8
> population
    total grid population:  9
> quit
    goodbye...
%
```

The GridWorld class

Your application program **must** work by exercising a class called `GridWorld` which you must write and which does most of the real work.

Your `GridWorld` class must implement a Java **interface** called `GridWorldI`. The interface appears at the end of this document and will be posted.

You may **not** modify this interface in any way!

When testing your submission, we will not only run your interactive application program; we will also directly exercise your implementation of the `GridWorld` class.

What you may use

Almost all of your data structures must be done "from scratch". You may use the `ArrayList` class and the `String` and `Scanner` classes (for parsing user input), but that is about it. (Any of the Collections classes besides `ArrayList` is off limits – e.g., `HashMap`, etc.)

Grading

Programs will be graded both on their correctness (as evidenced by our executing them), on the readability of the source code and evidence of good programming practices.

Programs that fail to compile will receive no credit. Otherwise, partial credit may be given. Therefore, if you cannot complete the entire assignment, do try to get at least some part(s) to work correctly so as to maximize the credit you earn.

Part I will be submitted in hardcopy form in class.

Part II will be submitted via Blackboard.

GridWorld Java Interface

```
import java.util.ArrayList;


// GridWorld interface
public interface GridWorldI {

        // initializes the data structures with the given arguments.  World
        //  presumably initially created with a default constructor.
        // if rand==true, each person is assigned to a random grid location
        // if rand==false, each person is assigned to grid location (0,0)
        // returns true if operations succeeds
        //   false otherwise (i.e., non-positive number of rows or columns;
        //   negative number of people).
        public boolean init(boolean rand, int nrows, int ncols, int numPeople);

        //returns the number of rows in the grid
        public int numRows();

        //returns the number of colummns in the grid
        public int numCols();

        //returns an ArrayList containing the member identifiers living in
        //the district (row, col) passed in input
        public ArrayList<Integer> members(int row, int col);

        //returns the current population count (which may change as people
        // are created and deleted)
        public int population();

        //returns the population in the district (row,col) passed in input
        public int population(int row, int col);

        //moves person with ID personID to district (newRow,newCol)
        //returns true if operation succeeds, false if operation fails
        //  (e.g., person with personID does not exist, either because person
        //  was deleted before or because it was never created or because
        //  personID is negative,
        //(another example of when it should return false: newRow or newCol is
        //  out of range)
        public boolean move(int personID, int newRow, int newCol);

        //returns the row where the person with ID personID lives
        //returns -1 if personID does not exist
        public int row(int personID);

        //returns the column where the person with ID personID lives
        //returns -1 if personID does not exist
        public int column(int personID);
        //deletes the person with ID personID.
        //returns true if operation succeeds, false if operation fails
        //(e.g., person with ID personID does not exist, either because it
        //was deleted previously or because it was never created)
        public boolean kill(int personID);

        //creates a new person and places that person in district (row,col).
        //returns that person’s ID
        //if one or more previously used IDs are available (because of pervious kill
        // operations), one of those must be used.
        public int create(int row, int col);

}
```

FAQ: Command Line Args, etc.

==

Q: How do I run a Java program from the UNIX shell -- i.e., outside of
and IDE?

A: Follow these steps:

(1) navigate to the directory containing your Java source files
(2) run javac ("Java Compile") on your Java files, for example
(from the shell prompt):

```
$ javac MyJavaClass.java
```

This will produce a .class file

(3a) You can then run your program like this:

```
$ java MyJavaClass
```

(3b) if your program takes command line arguments, you just specify
them on the same line. For example, if I want to pass in arguments
foo and bar (as strings) I would say:

```
$ java MyJavaClass foo bar
```

(cmd-line arguments below...)

==

Q: What are command line arguments and how to I access them?

A: Any java application has a main method which takes an array of
strings (usually called args) as its only argument. This array gives
a way to pass information to the program from the time it starts --
these are called command line arguments. For example, I might specify
the name of a file I want the program to process on the command line.
The program would access this name in args[0].

==

Q: Ok, but how do I set the cmd-line arguments when running the program?

A: If you are running the program from the UNIX shell (i.e., not inside
an IDE), see above discussion on that topic.

If you are running in Eclipse, from within your project, click
Run->Run Configuration. From the popup window, click the "Arguments" tab
and enter the cmd line arguments in the "Program arguments" window.