CS61B, Fall 2012 **Project #2: Spatial Database** **P. N. Hilfinger**

Due: 9 November 2012

1 Background

For this second project, we consider the problem of organizing a collection of data about objects in space. Imagine that you are given a large collection of things—people, say—each of which has some location at any given time. We might then imagine making *queries* about this collection, such as "Where is so-and-so?" or "Who is currently within 100 yards of location such-and-such?" or "What pairs of people are within 50 yards of each other?"

Any one of the sample queries above could be answered by simply searching all objects or (in the last case) all pairs of objects. However, if we want to our system to handle large collections of data, it would be nice to narrow down the set of objects or pairs we must consider. In the case of geographical data, one way to do this is a data structure known as a *quadtree,* described in §6.3 of *Data Structures (Into Java)* (for three-dimensional data, there is an analogous structure known as an *octtree.*) This is a recursive structure that divides the set of data into four quadrants by position, and then repeatedly subdivides the quadrants as necessary to get down to subdivisions that contain just one (or at least some small number) of items. With this, it is relatively easy to answer "who is within distance d of point x."

In this project, we'll consider building a library class for just such a structure and using it in a small application. Your solution will consist not just of a main program that runs the application, but also of a specific library data structure that can be used by other main programs. That is, you'll be fulfilling an *Application Programming Interface (API).* We'll be testing *both* your main program and API implementation separately. Therefore, if you violate the boundary between these two—making your application depend on some detail of your data structure other than the public interface, or making your data structure assume certain things about the application that uses it—your program will fail our tests (it might not even compile).

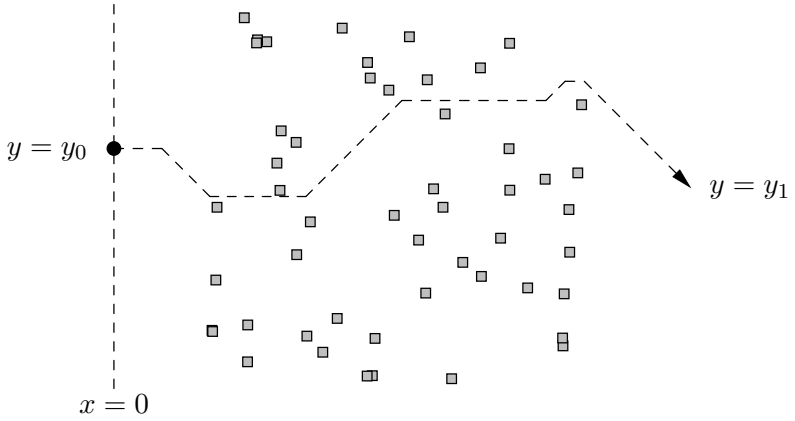

Figure 1: Path of a sample vehicle.

The application, called `tracker`, simulates a vehicle moving through a large field of fixed observation posts. At intervals, some of the observation posts broadcast a strong radar pulse that makes the vehicle detectable to posts in the vehicle's vicinity. These posts record the position of the vehicle and the time at which they see it. Meanwhile, the vehicle (whose pilot really doesn't like being tracked) changes course when it detects one of these signals, turning 45° to the right or left according to a pseudo-random chooser that we provide.

2 Running the program

The main program you will write is called `simulate.Track`. You invoke it from a command line like this (things in square braces are optional):

`java tracker.Main [ --debug=`*N* `] [` *INPUTFILE* `[` *OUTPUTFILE* `] ]`

This runs the program taking input from the file *INPUTFILE,* or, if this is defaulted, from the standard input, writing the results to OUTPUTFILE, or if that is defaulted, to the standard output. The `--debug` switch, if present, turns on any diagnostic printing you want to do (N indicates what level of output you want; interpret it as you choose). We will test that your program does not blow up if we use this option, but will ignore the output and won't dictate what N means (except that $N \leq 0$ means no output. We suggest that if you want to put in statements to print things while debugging your program, you arrange for the printing to happen *only* when this switch is present. Your program is wrong if it produces extraneous debugging output with the switch absent.

3 Input

Input files for `Track` have the following contents, all floating-point numbers:

$T_{\texttt{max}}$ S D y v_x
x_1 y_1 Δt_1
x_2 y_2 Δt_2
...
x_n y_n Δt_n

This input is in *free format:* each input number is delimited by an arbitrary amount of whitespace (including blanks, tabs, and newlines.) We suggest using the `Scanner` class to read it. Although we have shown the input broken into lines at particular points, line breaks are freely interchangeable with other whitespace, so that each number could appear on a separate line or all the input could be on a single long line.

The input file may contain comments, which start with a '`#`' and continue to the end of their line. Consider the fact that therefore whenever you read a number, you might encounter a comment instead. First, you'll need to figure out how to get your Scanner to skip these comments. Second, you will now have a situation in which, if you're not careful, you'll have many repetitions of the code to skip comments sprinkled throughout your program. How should you deal with this?

The inputs have the following meanings:

- $T_{\max}$ is the length of the simulation (which starts at $t = 0$.)
- S is a seed value that your program is to feed to the supplied random-number generator in the skeleton.
- D is the maximum distance at which an observation post can detect the vehicle, when it is illuminated by a radar pulse.
- y is the initial y-position of the vehicle (the initial x is 0.)
- $v_x > 0$ is the initial velocity in the x direction (the initial y velocity is always 0.)
- x_k and y_k are the coordinates of the post k.
- $\Delta t_k \geq 0$ is the interval at which post k sends out a radar pulse that makes the vehicle visible to other posts. If $\Delta t_k = 0$, the post never sends out a pulse. Each post sends out its first signal at time $t = \Delta t_k$.

4 Output

Your program should print out a report containing the simulation parameters ($T_{\max}$, S, D, y, v_x); the positions of the vehicle reported by each observation post, and the vehicle's position at $t = T_{\max}$ using the format shown in this example:

```
Simulation parameters:
   Total simulated time: 100.0
   Random seed: 42
   Maximum detection radius: 5.0
   Initial vertical location: 50.0
   Initial horizontal velocity of vehicle: 10.0

Reports:
   Post #1 at (15.0, 52.0):
      (15.0, 50.0)@1.5
      (16.4, 51.4)@2.0
   Post #17 at (17.0, 53.0):
      (16.4, 51.4)@2.0
   ...
   Post #50 at (85.0, 80.0)
      (88.0, 79.0)@95.0

Final position: (91.5, 75.5)@100.0
```

Each observation post reports the positions and times at which it sees the vehicle in order of increasing time. Include only those posts that have at least one sighting. List the posts themselves in the order they were input (using "Post #k" for the post at (x_k, y_k).

5 The API

The template files for this project include an abstract class `util.Set2D`, an implementation of it contained in `util.QuadTree`, and a utility class, `util.Reporter`, to help control debugging output. The only things in the package `util` that your other files (`tracker/Main.java` and any other classes you write) are allowed to use are the public methods and constructors defined in these classes. You may not expand this set with additional public or protected members (only private or package private ones). Do *not* try to circumvent this restriction, since we will be testing your main program and your `util.QuadTree` class separately using our own implementations, and they will fail if you violate the interface in any way. Both `util` and `tracker` also contain testing classes. Be sure *not* to make your implementation depend on these files either. Finally, keep all your `.java` files in the two packages `tracker` and `util`, so that we can easily test these two independently.

6 Your Task

The directory `~cs61b/code/proj2` contains skeleton files for this project. They are also available via Subversion (under URL `$STAFFREPOS/proj2`).

Please read *General Guidelines for Programming Projects* (see the main lab web page). To submit your result, use the usual procedures. Before doing so, you can use `pretest proj2` on the instructional machines to do a simple sanity check. You will turn in nothing on paper.

Be sure to include tests of your program (yes, that is part of the grade), by extending the `tracker.UnitTest` and `util.UnitTest` methods in the skeleton. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time.

The input to your program will come from fallible humans. Therefore, part of the problem is dealing gracefully with errors. When the user makes a syntax error, your program should not simply halt and catch fire, but should give some sort of message including the word "error" (in either case) on its first line and then try to get back to a usable state. However, for this project, we are not going to be fussy. As long as you detect and report the *first* error, your program will be judged to be correct, and any output after the first error message will be ignored. As for Project 1, your choice of recovery is less important than being sure that your program *does* recover gracefully. Your project should exit with status code 1 (which happens when you call `System.exit(1)`, if the command-line arguments are wrong. Do so also if the user made any error entering commands to the program, but instead of exiting immediately on such errors, remember that they happened and generate a status code of 1 when your program eventually does exit. Error-free execution, on the other hand, should result in a status code of 0, as is conventional.

Pay attention to style (as will the graders). Comment all new methods you introduce *helpfully,* so that the comment alone allows a reader to understand what any given call will do. Keep the program readable. Avoid masses of redundant code, and take every opportunity to identify and factor out well-defined methods.

Our testing of your projects (but not our grading!) will be automated. The testing program will be finicky, so be sure that:

- Your main function is in a class called `tracker.`. Your quadtree implementation is in class `util.QuadTree` and implements `util.Set2D`. The skeleton is already set up this way.
- Again, don't modify the API.
- The main procedures in classes `tracker.UnitTest` and `util.UnitTest` should run your tests. We have set up the skeleton file to show you how you test even package-private methods in your JUnit tests.
- Observe the exit-code conventions.

We will eventually be providing our own version of the main program and our own implementation of the API. You can use these to test the two parts of your program in isolation.

7 Advice

As before, don't make the problem any harder than it already is. If you find handling the command syntax to be difficult, you are probably making life difficult for yourself unnecessarily: talk to us about it. For writing files, there are `java.io.FileWriter` and `java.io.PrintWriter`.

It's important to have *something* working as soon as possible. You'll prevent really serious trouble by doing so. I suggest the following order to getting things working:

1. Write some initial test cases.
2. Handle the mechanics of processing the command-line parameters and getting the input and output files connected to the standard input and standard output.
3. Get the program to read and properly echo the simulation parameters.
4. Figure out how you are going to represent an observation post (I suggest introducing a class for this purpose).
5. Write the code to read in all the positions of observation posts and to create whatever data structures you use to store them.
6. Figure out how you are going to keep track of the next time at which some observation post sends out a pulse.
7. Implement the vehicle simulation, pretending for the moment that *all* posts see the vehicle each time. Get the output format for their reports working.
8. Now restrict the posts that see the vehicle to those that are within range. You'll be able to use the staff solution for the `util` package for this, or just use the supplied `SimpleSet2D` class.
9. Now figure out (and document with comments) how you are going to represent a quadtree, which you will use for finding points.

10. Implement the insertion of points into your quadtree.

11. Implement the iterator that lets you sequence through all points in your quadtree.

12. Now implement finding all particles within a given distance of a given point. To find all particles within a distance d of (x, y), first find all particles whose coordinates are in the range $(x \pm d, y \pm d)$, which gives you a superset of what you want. Next, check all of these points to see which fall within distance d.

As you go through this process, keep adding JUnit tests of the features or methods you add, using them to test your program.

Please don't move code out of the `tracker` or `util` packages in order to avoid dealing with packages. We will be testing that your `tracker` package works with our `util` package. In order for this to work, you must leave `track.java` pretty much as it is (basically just a call to `tracker.Main.main`).

7.1 Using staff versions

We will maintain two JAR (Java Archive Repository) files containing the staff's versions of the packages `tracker` and `util`. You can use these to supply one half of the assignment while you work on the other. We are not supplying source, of course, so when you encounter an error that occurs in one of the staff's classes, you will have to step upwards in the call stack to find where your part most recently called the staff's. Don't overuse the staff packages; you should rely principally on your own unit testing. The staff package, however, can serve as a "sanity check" that you haven't seriously misunderstood the assignment.

The staff version of the `util` package will be in `~cs61b/lib/proj2-util.jar` and the `tracker` will be in `~cs61b/lib/proj2-tracker.jar`, when they become available. To use either of these, you must include the appropriate JAR file in your "class path" *before* the current directory. If you don't know how to do this, by all means ask!