

Name: \_\_\_\_\_ Section: \_\_\_\_\_ CM: \_\_\_\_\_

# CSSE 220—Object-Oriented Software Development

## Final Exam – Part 2, February 19, 2018

**Allowed Resources for Part 2.** Open book, open notes, and open computer. Limited network access. You may use the network only to access your own files, the course Moodle and Piazza sites (but obviously don't post on Piazza) and web pages, the textbook's site, Oracle's Java website, and Logan Library's online books.

**Instructions.** *You must disable Microsoft Lync, IM, email, and other such communication programs before beginning part 2 of the exam. Any communication with anyone other than the instructor or a TA during the exam may result in a failing grade for the course.*

You must actually get these problems working on your computer. Almost all of the credit for the problems will be for code that actually works. There are several different small methods to write, so you can get a lot of partial credit by getting some of them to work. If you get every part working, comments are not required. If you do not get a method to work, comments may help me to understand enough so I can give you (a small amount of) partial credit. **NOTE: CODE THAT DOES NOT COMPILE WILL NOT BE ELIGIBLE FOR PARTIAL CREDIT!**

**Begin part 2 by checking out the project named *Final-201820* from your course SVN repository.** (Ask for help immediately if you are unable to do this.)

When you have finished a problem, and more frequently if you wish, **submit your code by committing it to your SVN repository. We will check commit logs, so you must be careful not to commit anything after the end of the exam.** For grading, we will ensure that the included JUnit tests have not been changed.

*Part 2 is included in this document.* **Do not use non-approved websites like search engines (Google) or any website other than those listed above.** Be sure to turn in these instructions, with your name written above, to your exam proctor. You should not exit the examination room with these instructions.

## Part 2—Computer Part

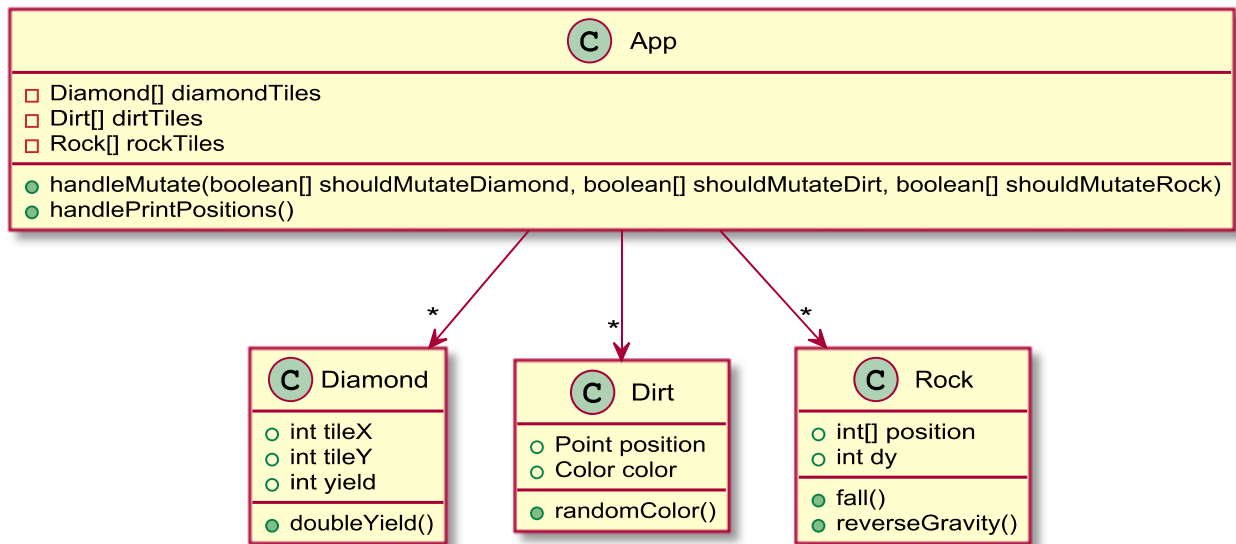
---

### Problem Descriptions

**Part A: 3 Linked List** (18 points) Implement the code for the 3 unimplemented methods in `SinglyLinkedList.java` – each problem is worth 6 points. Instructions are included in the comments of each method. Unit tests are included in `SinglyLinkedListTest.java`.

**Part B: Recursion** (6 points) Implement the code for the unimplemented method in `RecursionProblems.java`. Instructions are included in the comments of the file and corresponding unit tests are included in `RecursionProblemsTest.java`.

**Part C: HashMap** (6 points) Implement the code for the unimplemented method in `HashMapProblem.java`. Instructions are included in the comments of the file and corresponding unit tests are included in `HashMapProblemTest.java`.



A student's initial approach to Part D.

### Part D: Refactoring (15 points)

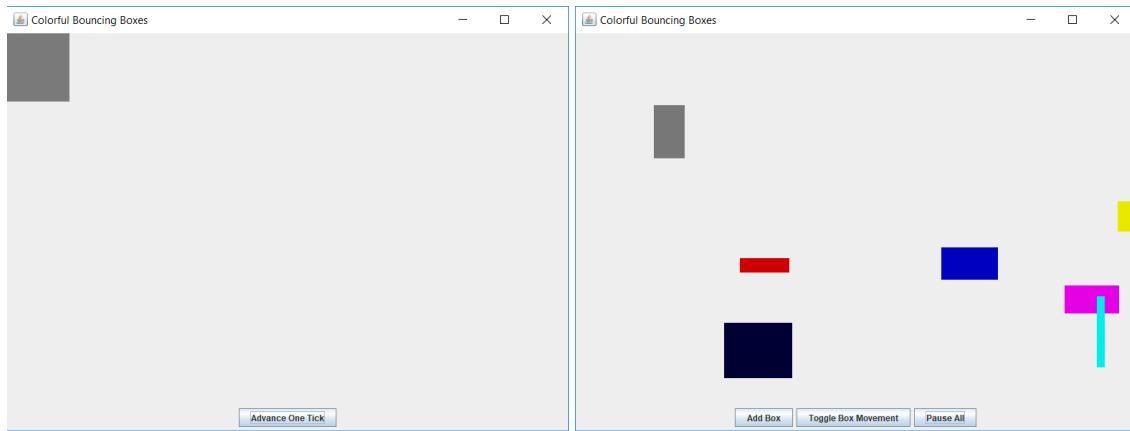
In a game similar to the one you worked on for the term project, levels are divided into tiles:

- Dirt does nothing interesting
- Diamond tiles yield diamonds when broken, which can be collected for points.
- Rocks fall when the dirt underneath them disappears.

During gameplay, the game randomly *mutates* tiles in interesting ways. Each tile knows how to mutate itself:

- When a diamond tile is mutated, its diamond yield doubles when it is broken.
- Dirt changes colors when mutated.
- Rocks reverse their gravity, so when they "fall", they rise **up** through any open space instead.

You are given an excerpt from the student's code. The student's code is functional, but the student's code is **poorly designed**. Your job is use **inheritance and/or interfaces** to remove as much code duplication as you can.



(Left) Starting code with a single dynamic box which changes color and position.

(Right) Multiple dynamic boxes added which also change color and position.

### Part E: Dynamic Boxes (20 points)

In this problem we have given you code, including a `DynamicBox` class (position and color change over time), that draws a single `DynamicBox` in a random location on the screen. Add or modify code to do the following:

#### Stage 1 (5 Points)

At the moment there is a single button you press which will advance the state of the simulation. You should replace this button press mechanism with a `Timer` (`javax.swing.Timer`) which updates based on the constant `DELAY` specified in `DynamicBoxMain.java`. (3 points)

Before or after making the `Timer` refactor above, you may notice that the box (`DynamicBox`) on the screen does not appear to be updating properly. As the code starts, the state of the box is getting updated, but you will not see the updated position unless you drag the window to re-size it. Think about what needs to happen to display the updated `Component` and modify or add to the code at an appropriate location so that it will display properly without requiring the window to be re-sized. (2 points)

#### Stage 2 (5 Points) Add buttons to adding boxes

Create the "Add box" button. Make it so that each time you click the "Add Box" button a new box will be created and added in a random location on the screen, while leaving all previously created boxes present. (This will require making changes in `DynamicBoxComponent`.) (Adding buttons 1 point. "Add box" adds box when clicked 2 points. Boxes added and update and move/change color correctly 2 points.)

**Note: You can receive partial credit (3/5) for Stage 3 and 4 even if you do not get stage 2 working, however, it will be faster to do the entire problem if you complete stage 2 first.**

### Stage 3 (5 Points) Toggle Box Movement Button

Create the "Toggle Box Movement" button. Make it so that each time you click the "Toggle Box Movement" button ALL of the boxes currently on the screen switch from moving to stopped, or from stopped to moving. While the movement may stop, the color should still change as normal. You should NOT change the boxes' velocity to do this! You will need to add to and modify some of the provided classes to do this. (Adding buttons 1 point. Toggle functionality as specified 4 points. If stage 2 is incomplete but this stage works correctly 3/5 points)

**Note: If this is done properly, after adding a few boxes and pressing "Toggle Box Movement" if you add more boxes the new boxes will be moving and the old boxes will still be stopped. If you press the "Toggle Box Movement" button again, the new boxes should stop and the old boxes should starting moving again.**

### Stage 4 (5 Points) Pause ALL Button (Movement and Color)

Create the "Pause ALL" button. Make it so that the first time you click the "Pause ALL" button the simulation freezes entirely (**movement and color!**), while preserving the state of all boxes. Upon a second press, the simulation should continue EXACTLY as it was before the first press. The same alternating effects should occur as the button is pressed repeatedly. You should not have to change the properties of any boxes to accomplish this! To do this, you should make the Timer itself stop so that ticks will not advance at all. (Adding buttons 1 point. Pause/UnPause functionality as specified 4 points. If stage 2 is incomplete but this stage works correctly 3/5 points)

**Hint: The Timer class has a stop() and restart() method which can be used after it has been started.**