Project 1: Rare Element Simulator

DUE: Sunday, Feb 14th at 11:59pm ***(Extra Credit for Early Submission!)***

Setup

- Download the `p1.zip` and unzip it. This will create a folder `section-yourGMUUserName-p1`.
- Rename the folder replacing `section` with the `DL1`, `DL2`, `003`, etc. based on the lecture section you are in.
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address.
- After renaming, your folder should be named something like: `DL1-krusselc-p1`.
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name the zip `section-username-p1.zip` (no other type of archive) following the same rules for `section` and `username` as described above.
    - The submitted file should look something like this:

```
DL1-krusselc-p1.zip --> DL1-krusselc-p1 --> JavaFile1.java
                                            JavaFile2.java
                                            JavaFile3.java
                                            ...
```

- Submit to blackboard. DOWNLOAD AND VERIFY WHAT YOU HAVE UPLOADED IS THE RIGHT THING. ***Submitting the wrong files will result in a 0 on the assignment!***

Basic Procedures

You must:

- Have code that compiles with the command: **`javac *.java`** in your user directory **without errors or warnings**.
- Have code that runs with the command: **`java Simulation`**

You may:

- Add additional methods and variables, however these methods **must be private**.
- Add additional nested, local, or anonymous classes, but any nested classes must be private.

You may NOT:

- Add additional *public* methods, variables, or classes. You may have public methods in private nested classes.
- Use any built in Java Collections Framework classes in your program (e.g. no **`LinkedList`**, **`ArrayList`**, etc.).
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- Create any arrays anywhere in your program except in **`DynamicArray.java`**. Using arrays, in any way, outside of **`DynamicArray.java`** will result in a 0 on this assignment (the goal is to build and use dynamic array lists!). If you use arrays when testing in main() in any other class, remember to delete that test code.
- Make your program part of a package.
- Alter provided classes or methods that are complete (e.g. **`Display`**, **`Element`**, **`Empty`**, **`Metal`**, and most of **`Simulation`**).
- Alter any method signatures defined in this document of the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Change any code in any file below the "DO NOT EDIT BELOW THIS LINE" line. You still must add comments to any methods and instance variables that do not have comments already.
- Add any additional libraries/packages which require downloading from the internet.

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Where Should I Start?

Students coming from other universities and colleges may not have been exposed to large projects of this type, and the size of this document may seem overwhelming regardless. To help you get a handle on large projects, we are including this ordered checklist. Starting on Project 2, you're going to have to identify these steps yourself, but the general procedure is: (1) read, understand, document, and plan, (2) get the code compiling as a "code skeleton", and (3) make incremental improvements, testing as you go.

(1) Read, Understand, Document, and Plan

- ☐ Read the textbook chapter on Dynamic Array Lists. This, for some reason, has a lot of information you may want.
- ☐ Read this document, in full. I know it's long, but Page 1 is a reference, Page 2 is this checklist, Page 3 is motivation and overview, and Page 7 is about testing. So that's really 3 pages of actual description.
- ☐ Read the ENTIRE code base and comment ALL the code with JavaDocs. After this you'll know where everything is and you'll know how all the parts fit together. If you leave this to the end you'll be *very* unhappy.
- ☐ Run the JavaDoc checker and code style checker. It would be *very* tedious to only run them at the end…

(2) Code Skeleton #1 (already done!) + (3) Incremental Improvements #1

- ☐ Build a Dynamic Array: Complete the core data structure for the project. Tips:
  - In CS310, supporting data structures can be built and tested independently, and will get you lots of points!
  - In P1, you have a "code skeleton" provided for your dynamic array. Pay attention to how this looks, as you'll be making one later. Code skeletons allow you to compile (and potentially run) code *before* you have completed an entire class. **`DynamicArray`** should compile when you download it because it is a full skeleton!
  - Work methods in an order that allows you to test them (not always the order written). For example, you can't test **`remove()`** until you write **`add()`**, so write **`add()`** first.
  - Test every method before writing the next one (you can edit **`main()`** to add your own tests).
- ☐ Run the **`DynamicArray`** class main method, then add more tests. Finish testing and debugging before you continue.
- ☐ Run the code style checker and, if you added and helper methods, run the JavaDoc checker. Fix any problems.

(3) Code Skeleton #2 (your turn!)

- ☐ Create a "code skeleton" for the rest of the project so that the entire code base compiles. This does NOT mean you write all the methods! You just put in the required method *signatures* and use *dummy return statements*. Do not continue until you have code compiling without errors or warnings using: **`javac *.java`**
- ☐ Complete the JavaDocs for any new methods, and run the code style checker and the JavaDoc checker.

(4) Incremental Improvements #2

- ☐ Create a grid and connect it to the display. I.e. in **`Simulation`** complete everything except **`resize()`**. Run the simulator. Test and debug adding metal and removing it with the empty void element.
- ☐ Make **`Sand`** so that it works like **`Empty`** or **`Metal`**. Choose a different color and weight! Test in the simulator and a main method like the one shown in Metal.
- ☐ Make **`Sand`** fall down. Test in the simulator *and* by writing tests (you can make your own **`main()`** in Sand!).
- ☐ Make **`Water`** so that it works like **`Sand`**. Test.
- ☐ Make **`Water`** so that it falls down. Test, and make sure **`Sand`** falls through **`Water`**!
- ☐ Allow the simulation area to be expanded/shrunk. I.e. write **`resize()`** in **`Simulation`**. There are many parts to this method, do one part at a time, write helper code, and *test as you go*.
- ☐ Write **`pushUp()`** in **`Element`**, and make **`Sand`** and **`Water`** push-able (write **`push()`**). Test with resizing.
- ☐ You *might* be done! Try a lot of things and if something looks wrong… debug, debug, debug.
- ☐ Complete the JavaDocs for any new methods, and run the code style checker and the JavaDoc checker.
- ☐ Try creating and adding a new element! (See: "GUI:Advanced" section.)

TL;DR: Use the order above to write your project or you'll get very muddled.

Overview: Rare Elements

Scientists at your university are studying some new, rare elements they've discovered. In the past, their students came into the lab to play with them, but due to certain circumstances (pandemic? radioactive spill? alien abductions in the lab? you decide...) the students aren't allowed to come into the lab anymore.

The scientists have asked you (and your professor) to develop a simulator that can be used by their students to experiment with these rare elements. Your professor is an expert at creating amazing GUI visualizations and "simulation/game loops", but wants your help in "modeling" the rare elements. Additionally, they would like you to provide helper data structures to support the visualization.

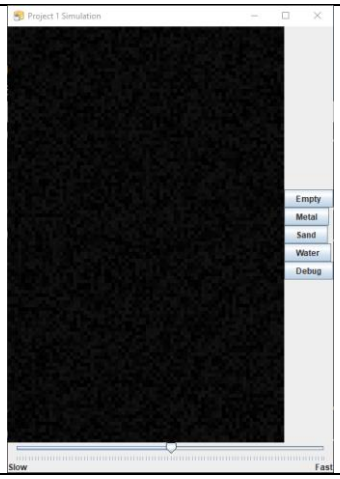
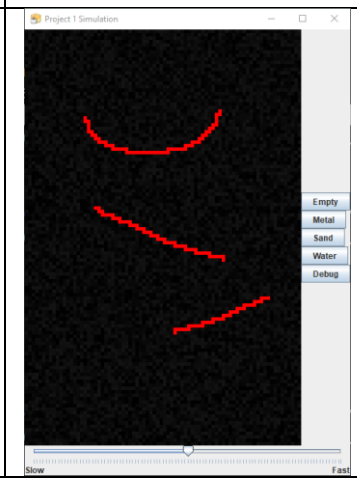
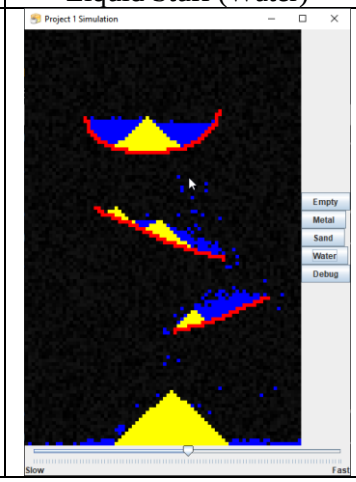
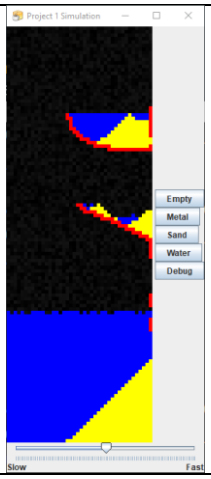
To get you started, let's introduce you to the rare elements you will be working with:

1. **Empty Void** – The emptiness of this cannot be emphasized enough. It is so empty that all other elements pass right through it and it can be infinitely compressed/squished. Oddly, Empty Void has a dark, shiny look to it.
2. **Rare Floating Metal** – This is a unique metal that, when placed in Empty Void, just sits there. It doesn't fly up or sink down, it just hangs exactly where it was placed. This metal can be compressed can crumple into itself if pushed around.
3. **Crystalline Grainy Stuff**, often called "Sand" by the scientists – This stuff gets everywhere and can fall through the tiniest cracks. It's rather heavy and sinks in Sloshy Liquid Stuff, but it's not heavy enough to push Rare Floating Metal around!
4. **Sloshy Liquid Stuff**, the scientists call this "Water" – This stuff sprays out and spreads out, so it fills in gaps pretty well. It's a lot like normal water, but don't drink it.

TL;DR: You're "helping" to simulate elements (such as metal, sand, and water) in an empty void.

Scientist's Rare Element Simulator

Below are some pictures of the simulator you are creating. Your professor has already written the Graphical User Interface (GUI) and clicking the buttons will (eventually) allow you to place the chosen element at the location and simulate what it would do. Below is what the simulator looks like initially, then after adding some Rare Floating Metal, and then after adding some Crystalline Grainy Stuff while placing some Sloshy Liquid Stuff.

Just Empty Void	Rare Floating Metal in Empty Void	Crystalline Grainy Stuff (Sand) and adding Sloshy Liquid Stuff (Water)	Simulation Area Resized (and elements pushed)
			

TL;DR: Cool GUI.

How Can You Help?

Before you start coding, make sure you have a solid understanding of what the end program will produce and a good understanding of what code has and hasn't been written. Again, you are "working with your professor" for this project, so there is a lot of code already written. You are just writing the "support code" for the simulator, *not* the simulator itself.

TL;DR: You're not writing the simulator. That's been done for you. You're writing the "helper" code.

Code Overview

Code that's ***completely*** written for you:

- **`Display`** (**`Display.java`**) – This is the GUI code. There are some notes here about writing GUIs as a programmer and it is fully JavaDoc-ed for your easy viewing pleasure. You may add code here if you decide to do the optional "GUI:Advanced" section of this project.
- **`Empty`** (**`Empty.java`**) – This represents a single atom of Empty Void[1]. Empty Void appears to shimmer and change colors when viewed with the human eye and the scientists have asked this to be replicated. It doesn't fall when placed and it "compresses" when pushed around (when we say an element "compresses" it means to do what Empty Void does when pushed). This element is super light too. Your professor has written this for you so that you can see how to get random numbers with Math.random(), and model your other elements off of it.
- **`Metal`** (**`Metal.java`**) – This represents an atom of Rare Floating Metal. It's very heavy, but doesn't fall down or get pushed around by other elements. This is another example from your professor for how to make an element.

Code that's ***partially*** written for you:

- **`Element`** (**`Element.java`**) – This abstract class is the parent of all elements (Empty, Metal, Sand, and Water) and there are some abstract methods that elements implement. Here is some basic information about elements:
  - Elements have a color (see **`getColor()`**) which is used for displaying the element in the simulator.
  - Elements have a weight (see **`getWeight()`**) which is used for comparing elements when falling or pushing (see **`compareTo()`**).
  - Elements have different behaviors when they fall or are pushed (**`fall()`** and **`push()`**).
  - There are two helper methods for **`push()`** called **`pushLeft()`** – written – and **`pushUp()`** – not written.

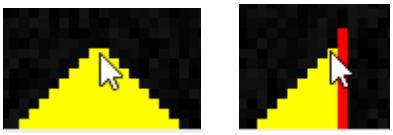
  Incidentally, there is an example here of how to document an overridden method without rewriting or copy-and-pasting using the {@inheritdoc} tag.
- **`Simulation`** (**`Simulation.java`**) – This is the core simulator! This has a 2D grid built off of dynamic array lists (see **`grid`** instance variable) and a GUI display (see **`display`** instance variable). The simulation/game loop has already been written by your professor, you just need to finish some things up:
  - You need to create and initialize the 2D grid, and initialize the display (see **`Simulation()`**).
  - You need to respond to a user placing an element into the grid (see **`locationClicked()`**).
  - You need to tell the display what colors to make each part of the image (see **`updateDisplay()`**).
  - You need to handle resizing the grid if the user expands/shrinks the simulation space (see **`resize()`**).

Code ***not*** written for you:

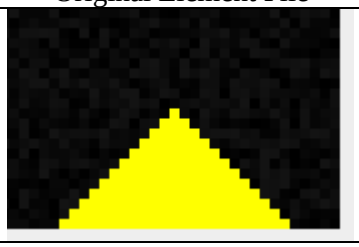
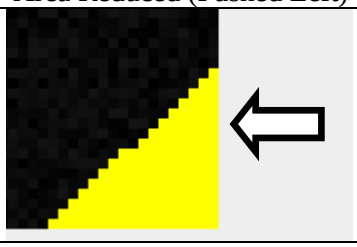
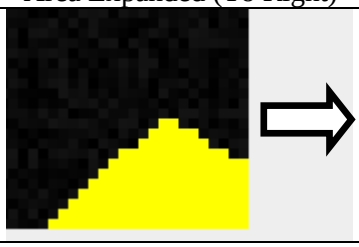
- **`Sand`** (**`Sand.java`**) – You need to simulate what Crystalline Grainy Stuff does when it is placed in the world. Here's what you need to know about this element:
  - Crystalline Grainy Stuff *is an* element. Just like Empty Void and Rare Floating Metal.
  - This element shouldn't be the same color as any other element in the simulator. I've used yellow in my solution, but you can use another color if you want.
  - This element is heavier than Empty Void but not as heavy as Rare Floating Metal. This means it will fall down when placed in the void and push the void out of the way. Do not hard code the relationship between elements, use getWeight() to determine the relative weight of elements.
  - When this element "falls" it does the following: If possible, i.e. if it is heavier than the element under it, it falls straight down. If it can't fall straight down, then it might fall diagonal left or diagonal right, but only if it is heavier than the element in that direction. If it could fall either way, it randomly chooses between

[1] Yes, the scientists assure us that this type of Empty Void has atoms.

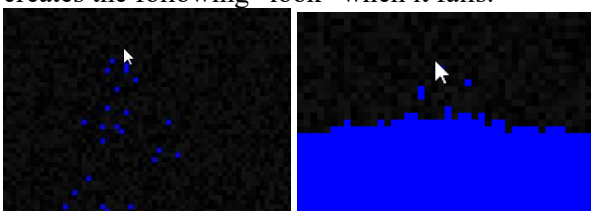
the two. This creates the following "look" when it falls:



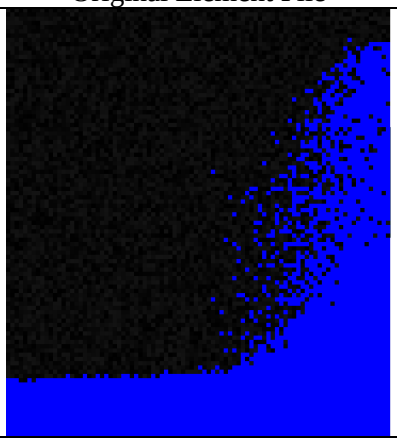
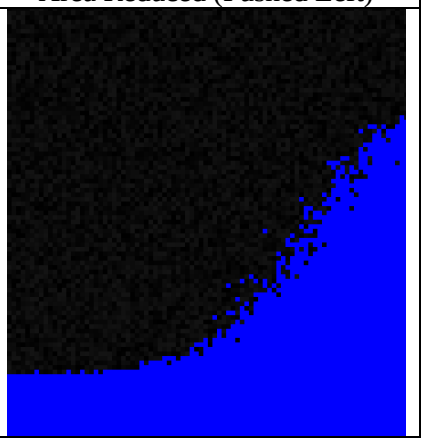
- When this element is "pushed" it tries to push elements *upward*. If that doesn't work, it tries to push to the *left*. If that doesn't work, it compresses itself (see the Empty Void element). Use the helper methods in Element, do not reinvent the wheel. If you follow these directions, this element will tend to push up against the edge of the area when the simulation area is compressed, and will spread out again (by falling) when it is expanded:

Original Element Pile	Area Reduced (Pushed Left)	Area Expanded (To Right)
		

- **`Water` (`Water.java`)** – You need to simulate what Sloshy Liquid Stuff does when it is placed in the world. Here's what you need to know about this element:
  - Sloshy Liquid Stuff *is an* element. Just like the other elements.
  - This element shouldn't be the same color as any other element in the simulator. I've used blue in my solution, but you can use another color if you want.
  - This element is heavier than Empty Void but not as lighter than Crystalline Grainy Stuff. This means it will fall down when placed in the void and push the void out of the way, but Crystalline Grainy Stuff should "sink" when dropped into it. Again, do not hard code the relationship between elements, use getWeight() to determine the relative weight of elements.
  - When this element "falls" it does the following: It chooses a random direction (left, right, or down) and falls that way if it is heavier than that element. If the chosen direction isn't possible, it gives up. This creates the following "look" when it falls:



  - When this element is "pushed" it randomly chooses between try pushing left or up *first*. If its chosen direction doesn't work, it tries the other way. This means that this element will take a little while to "flatten out" if you drop a lot in the same spot, but it will also create a type of "wave" when pushed from the side quickly:

Original Element Pile	Area Reduced (Pushed Left)
	

- **`DynamicArray`** (in **`DynamicArray.java`**) – Your 2D grid of elements is based on a dynamic array class. So you need to make one of those. Use your textbook and class discussions to help you out doing this, but remember: if you are copy-and-pasting something (or manually typing something you are currently reading), then you are "doing it wrong". Here's how to "do it right" so that you'll learn what you're supposed to learn:
    - Read the textbook and enjoy the lecture.
    - Put everything away and code a dynamic array class *on your own*. No notes, no textbook, no anything else but you and your code.
    - If you get very stuck and really want to look at your nodes. Put your code away (completely close the IDE or even your computer if possible). Only when you can avoid the temptation, go read/watch/listen to try get some more ideas or clarify your thoughts. Once you have ideas, put everything else away again and open your code.
    - Doesn't that make it "harder"? Yep, but have you ever been singing with a song you like and turned the music off, only to realize you weren't actually singing well and didn't really know the lyrics? Same thing happens when you "code along", you don't actually know how to do it but you feel like you do ☹

Every method you are required to implement in the DynamicArray class has already been put into the code. Dummy return statements have been used to make sure the code compiles (this is called a "code skeleton"). Similarly, much of the rest of the code base has been outlined for you. Do not alter any method signatures written in the provided code or add any additional public methods (though you may add additional *private* methods, fields, and classes). Additional comments are in the code describing what each method should do, what they should return, and any other requirements (such as Big-O, code reuse, etc.).

TL;DR: There is a lot of information above about how to write the classes… you'll need to read this completely.

GUI:Advanced?

You'll want to be 100% done with the project to try this part; it is a bonus challenge for those interested.

Once you've finished adding the required elements, try creating another element that works completely differently and integrate it into the simulator. There are several places in **`Display`** tagged with **`[GUI:Advanced]`** to help you.

Very important warning: If you break anything while doing this, yes, you will lose points, so don't break anything! Testing/grading will be done with a FRESH COPY of Display, so don't go changing (breaking) other classes willy-nilly.

If you do something cool (cool = not just a copy of an existing element, but something creative), and you want to share it with the TAs or the class (for a tiny amount of extra credit), then:

1. Before you submit your project, make a PRIVATE post on Piazza with the title in the format: "GUI:Advanced [Your Name]". In that post, show three things (in this order):
    - One of the following sentences at the top (in bold, please):
        - "I authorize the professors to make this post PUBLC if chosen by the UTAs."
        - "Please keep this post PRIVATE, I want to remain anonymous and would like you to make a public post for me (without my name) if my code is chosen."
    - A video (link) or a series of images (linked or posted directly on Piazza) showing off the cool feature(s) of your element.
    - The "cool" part of your element's implementation code (1-2 methods *in a code block*, do not upload the entire element's class).
2. Change **`newElementPost()`** in **`Simulation`** to return the full URL of the PRIVATE post.

Depending on how many submissions we have, the UTAs will be picking the top X "cool elements" to show off to the class. If your code is chosen, you'll get +1pt extra credit. If you've already submitted your project and want to do this part afterwards, that's cool too, but you won't be eligible for the extra credit. Only code where newElementPost() is a valid link to the post will be considered.

TL;DR: Tiny extra credit opportunity if you get finished and want to try out modifying the GUI.

Where Do I Start and How Do I Test?

Different parts of the project depend on other parts of the project. Documentation, testing, and debugging should be done *throughout* the entire project. There is a *detailed, ordered checklist* on the second page of this document, but this section gives you an overview.

Order of Attack

What depends on what?

- The simulator uses your **`DynamicArray`** class to build a 2D grid, so build the dynamic array list first.
- You won't be able to test the simulator until the code compiles, so make a complete code skeleton for the remaining classes and methods you need to write.
- If you want to test your implementation of **`Sand`** and **`Water`**, you need the simulator to run, so work on the simulator methods first (but don't worry about resizing, this only happens when the window size changes).
- Work on **`Sand`** and **`Water`** that can fall. Now you can test it with the simulator.
- Add resizing (without pushing), and then try out pushing.
- Try out the "GUI:Advanced" section.

Incremental Testing

This assignment has a lot of moving parts, so we are providing you will a few (very basic) self-checks to get you started testing. If you develop your code in the order we suggest in the next section, you'll be able to run a little main method in the core data structure to double check your progress half way through. You can edit this main method as much as you like to do additional testing on your own, and you can add main methods to other classes to test those as well (but don't edit the main method in **`Simulation`**). A sample main method is sketched out in Metal showing how to test elements independently of the simulator.

Below is sample output from running the main method in DynamicArray.java:

```
> javac DynamicArray.java
> java DynamicArray
Yay 1
Yay 2
Yay 3
Yay 4
Printing values: 5 10
```

There is an ***outline*** for how to test a single element in Metal using a main method tester. If you complete it, you can use:

```
> java Metal
Yay 1
```

TL;DR: Definitely checkout (and use) the main in DynamicArray and Metal.

Simulation Testing

Once you are done writing the helper code, the simulator can be used! **`Simulation`** can be run from the command line as follows:

```
> javac *.java
> java Simulation
```

There is a "debug" button in the simulator (it's a toggle). If you toggle this button on, then after *every* action the user takes (clicking, resizing) there will be a 5 second pause (see **`run()`** in **`Simulation`**). This might help with debugging pushing and falling. There is also a speed control at the bottom of the simulator if you want to slow down the simulator just a little.

TL;DR: Don't just write code without testing. This section describes how to write your project WHILE testing.