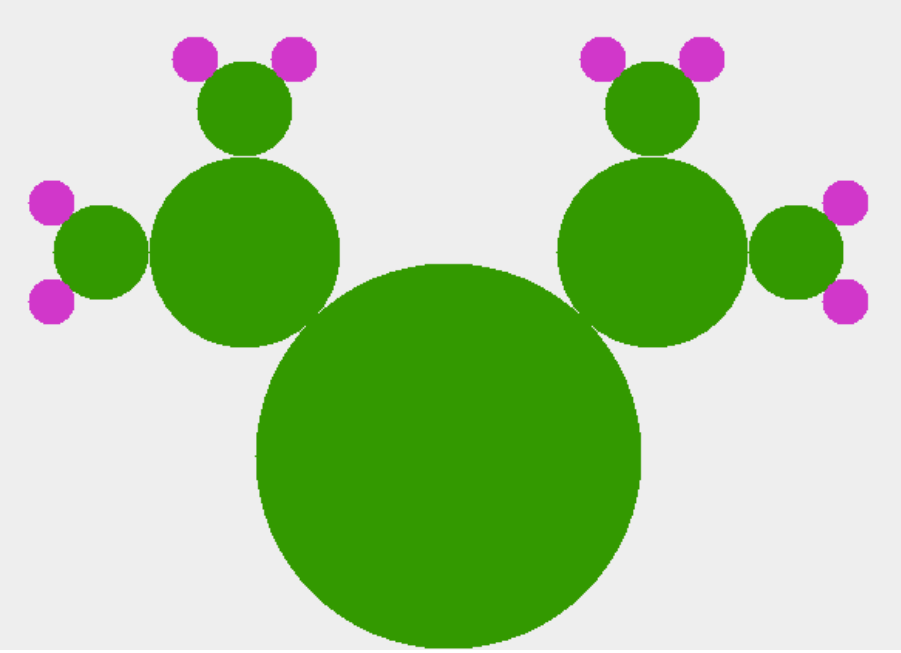
Project 6: Prickly Pear Cactus Fractal

1 Task

1.1 Scenario

Your goal is to create a program that will draw this type of fractal, which I'm calling a Prickly Pear Fractal because it resembles the prickly pear cactus or opuntia, native to the area where I grew up. This cactus is edible[1]. Here's an example of the drawn output:



1.2 The Fractal

Each of the children is drawn 45° from the parent's orientation. In the example above, the base of the cactus, the first "parent," is assumed to have an orientation that is "straight up"; its two children are at -45° left and right of that orientation. Each of *those* children's children are again 45° from their parent's orientation.

Recursion depth controls how many shapes will be drawn. In the example above, a recursion depth of *four* is shown. The color of the last level of shapes drawn is different; this will represent the "pear" portion, while the rest of the plant is the basic cactus color.

Regardless of the recursion depth set, stop drawing (and recursion) when the shapes are too small to draw; there's no point in attempting to draw shapes of zero size.

[1] The green "pads" or "paddles" can be eaten like a vegetable; in Spanish this part of the cactus is the *nopal*. The purplish fruit is said to taste like bubble gum crossed with watermelon; this is the *tuna*. You'll find recipes for preparing both parts of the cactus, e.g., here.

1.3 The User Interface

Create an attractive, usable GUI that allows the user to set these options:

- The recursion depth (range 2 to 10)
- The ratio of child radius to parent radius (range 40 to 70)
- The color of the "cactus" portion
  - Use a color picker dialog to make this easy; set a default color like the one in the example
  - Close to that widget, show a *color sample* of the color they have selected
- The color of the "pear" portion (same as described above for the "cactus" portion)
- A button that causes the fractal to be drawn using the requested settings

Use appropriate types of controls for the settings in question (simple text fields are not the answer; they make validation difficult and shouldn't be used for this project). If the user closes the settings dialog, the application should exit. The dialog should not be resizable.

1.4 The Display

Use a JPanel on a JFrame; draw the fractal on that. If the user closes the settings dialog, the application should exit. The commands to draw on the JPanel will be familiar to you, as you've used these Graphics methods before, when drawing on a DrawingPanel.

2 Code Organization

2.1 Classes

- Design one class responsible for **gathering user data** via the settings dialog.
- Design one class responsible for **generating fractal data** and storing it in an ArrayList (use the implementation you wrote in a previous project).
- Create one class responsible for **drawing the fractal** using the fractal data in the ArrayList (this must be a separate class from the GUI). But to make this more universal, have the fractal elements *draw themselves* when asked; a parameter to the draw method will be useful in having the drawing object communicate its dimensions.
- Main: since this is a full-featured application, you'll need a Main class separate from the others.
- Other helper classes are allowed.

2.2 Interactions

You'll need to design proper communication between the classes.

- Create a **Subject** interface that includes the expected methods (those for registering, removing, and notifying observers) plus a setData method that accepts all UI-generated data. The class that generates the fractal data must implement this interface.
- Create an **Observer** interface that includes expected methods, in this case a single method that signals updates to observers. The class that will display the fractal results must implement this interface.
- In the constructors of the GUI class and display class, accept as a parameter a reference to the fractal generation object.

- Use appropriate signaling:
  - When the **GUI** has new data for the fractal generator object, it should call the fractal generator's setData method and pass that data as arguments.
  - When the **fractal generator** is ready to generate new fractal pieces, it should notify its observers that updates are available.
  - When the **display** is ready to receive its new data, it should call the fractal generator's getData method. When the **fractal generator** receives this request, it should then (and only then) generate the fractal pieces and return them (a "pull" model).
- Main's job is then simple; it creates the fractal generator, GUI, and display objects, passing parameters as expected.

3 Design Documentation

This project requires a bit more documentation than have previous projects:

- Create an initial UML Class Diagram, including interfaces.
- Create a UML Object Diagram; that will help you visualize the data and connections between objects.
- Design an algorithm for determining where to draw the next layer of recursive fractal objects, the "children" of the current object, or "parent." Design this in whatever form you find helpful, e.g., diagrams and doodles, pseudocode, etc. This will take some time and thought.
- Sketch out a design for the user interface, showing widgets and layout. Draw in a tool, if you'd like (e.g., Word, PowerPoint, etc.) or on paper.
- While you don't need to draw it, do make use of the UML Sequence Diagram we'll cover in class.

4 Code Implementation

4.1 Additional Building Blocks You'll Need

- GUI dialog created in Java Swing
- Graphics drawn on a JPanel (which lives on a JFrame)
- Observer Design Pattern
- Recursion

4.2 Style

Follow the Course Style Guide. You'll lose points on every assignment if you fail to do so.

5 Testing

Take a break from the usual JUnit unit tests. The proof for this project will be visible by manipulating the user interface and seeing the results properly drawn.

6 Submitting Your Work

Follow the course's Submission Guide when submitting your project.

7 Hints

You can spend a lot of time tinkering and tweaking this code (aka you'll *waste a lot of time*). Take this advice to heart:

7.1 Remember the Basics

- Build your fractal-element generation code incrementally, verifying at each step that the results are exactly as you expect. Failure to do this creates A Huge Mess that you'll have to unravel.
- Remember the standard mathematical practice of carrying around the maximum precision for as long as possible, then round at the last moment; that is good advice, here.

7.2 Understand Coordinate Systems and Measurements

- In the end, circles must be drawn from the upper left-hand corner of their bounding box. But that's a detail that you can put off until the last minute. Instead, focus on the *center* in earlier thinking.
- Standard math focuses on the four-quadrant cartesian coordinate system, with (0,0) at the center. But most computer graphics programming does *not* use that system, but instead sets the origin to the upper left-hand corner. As you move right, X gets bigger; as you move down, Y gets bigger. Ponder that last detail and the implications of having two distinct systems.
- Unit-circle trigonometry assumes as radius of one. We'll start there but will quickly need to move outside of that confinement.
- In geometry and trigonometry, sometimes we talk about angles, sometimes radians. They aren't identical; make sure you're clear on the difference. I recommend focusing on the latter throughout.

7.3 Recommendations

- Build incrementally. Check your work at each step.
- I recommend you stay in Cartesian land (not Graphics land) until later in the process (see below).
- Refresh yourself on unit circle trigonometry[2] and conversing in radians; consider using radians everywhere in your project[3]. This refresher should allow you to calculate the point at which the parent and child circles are tangent; it is also helpful in figuring out the child's orientation. At the start, assume the parent circle is centered at (0,0) for ease of calculation.
- Once you feel confident about your unit-circle results, expand those to cover non-unit circles. This will help you calculate the centers of child circles. This should be sufficient to generate the fractal elements.
- When the elements want to draw themselves, they must first handle the Cartesian-to-Graphics shift; that's a simple mathematical one, but an important one. When the display object calls the fractal element's draw method, have it pass in the width and height of the display, so the FE can place itself properly (or find some other way to make sure the drawing knows about the display dimensions).
- The last thing elements must handle is to calculate the upper-left hand corner of their bounding boxes, which is again a simple thing to do given you know the coordinates of their centers and their radii.

[2] Do an online search for "unit circle trigonometry," finding resources to refresh your knowledge of the subject. Videos like this one might be helpful.

[3] Research Math.toRadians(), if you feel compelled to convert between them.

7.4 Overall Approach

It's tempting, and not unrealistic, to first build the program in a monolithic fashion. For example, you could write code that gathers setting information from the console/keyboard, generates the data, and draws the fractal on a DrawingPanel. But then this code will need to be split apart to fit into the model we'll design; that's not a small feat, requiring some mental paradigm shifts. Another approach would be to start with the model and make sure your signaling mechanisms are working properly; then write code in stages and check carefully that the state is right at each step.

8 Extra Credit

Allow the user to control the angle of the children, so that it's not always a 45° angle. Limit users to some reasonable range in which the drawing will look good.

9 Grading Matrix and Points Values

Area	Value	Evaluation
Design docs	10%	Did you submit initial design documents, and were they in reasonable shape to begin coding? Did you submit final design documents, and were they a good representation of the final version of the project?
Fractal drawn/colored properly	10%	Was the fractal drawn correctly, given the settings? Did subsequent drawings look good? Were screen sizes appropriate and friendly?
GUI attractive and useful	10%	Were appropriate widgets used? Was the layout attractive? Was space used wisely? Was the GUI intuitive to use?
GUI class implementation	10%	Was the class set up as requested? Was it successfully coded using good practices?
Fractal generation class	20%	Was the class set up as requested? Was it successfully coded using good practices?
Drawing class implementation	15%	Was the class set up as requested? Was it successfully coded using good practices?
Observer/Subject implementation	15%	Was the Observer model properly and successfully used? Were all the handshakes properly coded and used?
Style/internal documentation	10%	Did you use JavaDoc notation, and use it properly? Were other elements of style (including the Style Guide) followed?
Extra credit: user-controlled angle	3%	Is the user able to control the angle at which the child orientation branches off the parent orientation?
Total	**103%**	

Code that does not compile or run won't be graded