Chapter 8 Labs

Getting Started

Follow these instructions from the Chapter 4 Labs to get started.

Sudoku Solver

This exercise explores the use of recursion to solve sudoku problems. Sudoku is a Japanese logic-based number game that involves filling in a 9x9 grid of numbers according to specific rules. If you are unfamiliar with the game, please read up about it here.

Although there are many different algorithms for solving sudoku puzzles, backtracking is by far the most popular method. The idea of backtracking is simple. The first step is to grab the first empty square and determine all possibilities. Next, fill in the square with the first possibility and repeat the process with the next empty square. Eventually, the puzzle will either be solved (no empty squares) or a conflict will arise where an empty square has zero possibilities. In the latter case, the algorithm backtracks to the previous square and fills in the next possibility. If there are no possibilities left, backtrack again.

Your task is to implement a sudoku solver that uses the backtracking method. You should write your code in the `Solver.java` file in the `sudoku` package. I suggest that you write a `solve()` method that takes the puzzle (an `int[][]`) as the single input and returns a `boolean` that indicates whether the puzzle was solved. Additionally, it may help to make a `getPossibilities()` method that takes the puzzle and a row-column index and returns all possible numbers.

The file already has a main method with a test sudoku puzzle already loaded. In the puzzle array, zeroes are used to represent empty cells. To check your solution, the solved possible should be

```
735629148
928741365
164385279
819273654
347956821
256814793
671498532
482537916
593162487
```

Drawing in Java

The final three exercises all involve recursive drawings. To help you focus on developing the recursive algorithms, I've attempted to abstract drawing using turtles (no, not the animal). In the world of computer programming, turtles have been a staple of computer graphics since their inclusion in the Logo programming language released in 1967.

The abstraction is pretty simple to understand. Turtles are entities the draw and move on the screen in response to commands. A turtle's state consists of an (x,y) location on the drawing plane as well as a heading. The most basic command is forward; this moves the turtle forward a fixed distance in the direction of its heading. This functionality is supplemented by turning commands. This allows the user to change the heading of the turtle by a specific number of degrees. As the turtle moves, it leaves behind a trail (a line) on the screen. By commanding the turtle to move and turn in specific sequences, it can draw almost anything on the screen.

This basic functionality is baked into the `Turtle` class. However, before we create a turtle, we must create a `Screen` object for the turtle to draw on:

```
Screen screen = new Screen(750, 750);
```

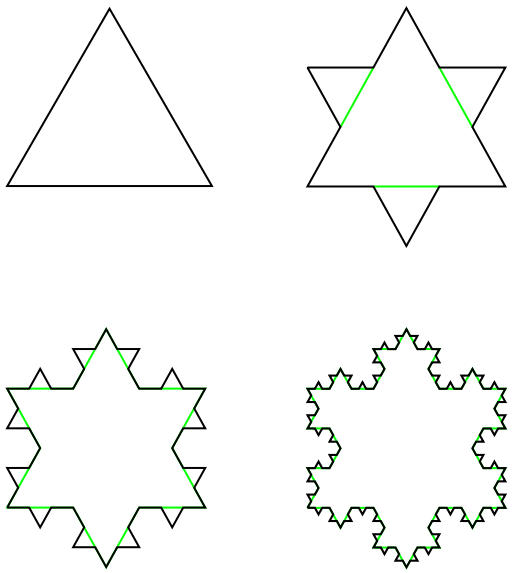
This line creates a `screen` object with a 750x750 drawing surface. Now we can create our turtle:

```
Turtle turtle = new Turtle(screen.getCanvas())
```

Now we're ready to draw on the screen! The methods are fairly self-explanatory; the most important are `forward()`, `backward()`, `moveTo()`, `left()`, and `right()`. Additionally, the coordinate system is a regular Cartesian plane with the origin at the center of the screen. The turtle's starting location also points right.

Koch Snowflake

Your first task is to draw this iconic fractal. The construction is simple; we begin with a single equilateral triangle. At each iteration, each side is replaced with four new ones each a third the length of the original. Here are a few steps of the process:



As you can see, the figure quickly becomes quite complex after a small number of iterations. For this task, you need to implement a recursive method that draws one side of the Koch Snowflake. Although this process is repeated endlessly to make the real fractal, we will limit the number of iterations to n=5.

Algorithmic Botany

In this exercise, we'll explore algorithmic botany, an interdisciplinary subfield of computer science that includes plant visualization. Specifically, your task is to write a recursive procedure that produces a plant image. The technique we'll use is similar to that of the previous exercise, just a little more complex. The Koch snowflake from before was generated by edge replacement; at each iteration, every edge was replaced by a more intricate pattern of edges and angles. To help describe this next pattern, we're going to formalize the process a little bit. We'll represent a forward move by "F", a left turn by +, and a right turn by -. Using this system, we can describe the snowflake iteration process by the replacement F→F+F--F+F where all angles are 60.

For our plant, we will also need two more commands. "[" corresponds to `Turtle.pushState()` and "]" corresponds to `Turtle.popState()`. Now, here's the replacement: F→F[+F]F[-F]F.