LAB 2 (part A)

Create a project named **`lab2`** for your work, or import the zip file from moodle. If you create your own, don't use the default package (rather, create at least one to contain your work).

Individual framework code, a complete framework project (exported as a .zip file), and this document are available at in the course moodle site.

1. Zipping arrays

Write a static method **`zip`** (in a class named **`Zip`**) that takes two **`int`** arrays and returns a single **`int`** array with the two arguments "zipped" together.

Zipping two arrays means: take the first element of the first array, add the first element of the second array, then add the second element of the first array, and the second element of the second array, and so forth and so on while each input has elements.

The following isn't valid Java, but is intended as example of what **`zip()`** should do.

```
zip( [1,2,3] , [11,12,13] )   -> [1,11,2,12,3,13]

zip( [1,2,3,4] , [11,12] )    -> [1,11,2,12,3,4]

zip( [1,2,3] , null )         -> [1,2,3]
```

One or both arguments may be **`null`**, which should be treated as an empty array. If both are **`null`** return a zero-length array.

Don't use **`ArrayList`**s for this problem, or declare more than the one array that **`zip()`** returns.

Do test your method thoroughly and submit your tests along with your solution. You can, although don't need to, make use of the **`ZipTester`** class. It includes methods to print arrays and get an array from the keyboard.

2. Drawing dots and lines – dream catchers

The **`DotsAndLines`** class in the framework code, along with the helper class **`MouseListenerDrawer`** that it uses, are basically the same as those discussed in lecture. Remember, you can ignore the **`MouseListenerDrawer`** class at this point; we'll be looking at drawing and graphics specifically later in the class.

`DotsAndLines` works currently; if you run it, you can click in the window that appears to create a dot. Repeated clicks make additional dots and lines are drawn between consecutive dots. Make

sure you understand how the algorithm to draw lines between adjacent points works (in the **`paintComponent()`** method).

In these next two problems, you'll be adding functionality to **`DotsAndLines`**. Keep the original **`DotsAndLines`** class around and, for each of the problems, work on a renamed copy.

You'll be changing the methods **`mousePressed()`** and **`paintComponent()`**. Recall, both are called automatically by Java in response to something the user does (clicking the mouse button in the window, for instance). Write helper methods as necessary to keep your code clean.

Problem: drawing a dream catcher.

Create a class **`ClosedPattern`** using a copy of **`DotsAndLines`**. In this program, a user clicks to create dots and lines as usual. However, when a user clicks on the first dot that was created, the figure becomes "closed" and the behavior of the program changes. After the figure has been closed:

a. Don't let the user make any more dots.
b. Draw a line between first and last dots as if they were adjacent. This will "close" the figure.
c. Draw a different colored line between all *other* pairs of dots. That is, "adjacent" dots should have a black line drawn between them, just as was done before the figure was closed, but "non-adjacent" dots should have a line of a different color drawn between them.

The Java class Color has predefined colors that you can use.

Don't make any additional arrays or **`ArrayList`**s to solve this. Make sure that you don't draw lines twice (e.g., both from point 1 to point 2 and from point 2 to point 1)!

3. A dot game

Create a **`DotGame`** class by making a copy of **`DotsAndLines`**. This program lets a user play the simple game.

The user clicks to add dots and lines as with **`DotsAndLines`**. And, as with **`ClosedPattern`**, when the user clicks on the first dot that was created, the program should stop making new dots and should "close" the figure as in step (b) above: a line should be drawn between the first and last dots.

At this point, the program enters a "toggling" mode. Whenever the user clicks on a dot, the program should to *toggle* all of the *possible* lines between that dot and every other dot. "Toggling" means that if a line is drawn, it should be erased, and if it is not drawn it should be drawn. See the example below with a 4-dot figure.

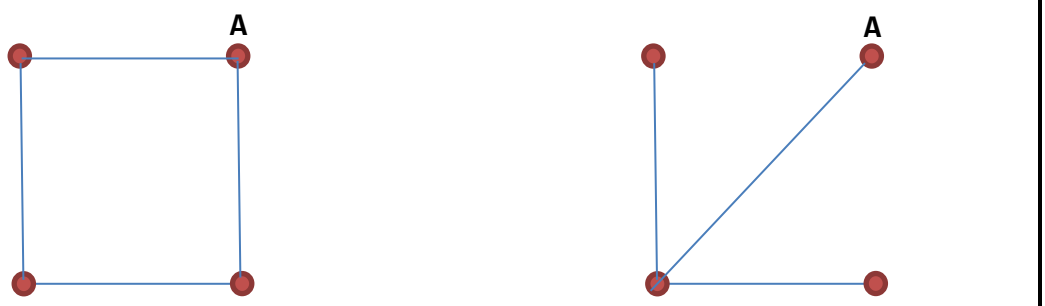

Figure starts as on left. After clicking on point A, all lines are "toggled", resulting in figure on right.

Use the following strategy: create two **`ArrayList`**s of **`Line`** objects. The first list should contain lines that are being drawn, while the second sequence contains lines that are hidden. The union of the two lists is all of the possible lines. In order to toggle whether a line is drawn or not, you need to move it from one **`ArrayList`** to the other. Think carefully about this, though: you can't remove from an **`ArrayList`** while you are looping over it! You'll need to keep temporary lists while you are setting up to move. Using helper methods will make your program easier to write.

You can build these two **`ArrayList`**s as the user is adding dots – this makes **`paintComponent()`** really simple but complicates **`mousePressed()`** -- or you can do it once the figure becomes closed.

A **`Line`** object is two points, similar to the classes you wrote in Lab 1. You'll want to add some methods to the **`Line`** class to make your **`DotGame`** class easier to write.

The game is "won" when either all the lines are drawn or none of the lines are drawn. It's easy to win with a figure of 4 dots (in the right figure above, clicking on the lower left dot will remove all possible lines). I'm not sure about figures with more than 4 dots, though!

It would be nice to indicate to the user that they have won, somehow. This is optional; you might look at the Java class **`Graphics`** to see what drawing methods it contains. Or, the **`MouseListenerDrawer.erase()`** method shows you how to fill the whole window with a single color.