**CS 245 Preassignment 1**

Xform

due before September 24

# 1 Overview

Say you have a point $x$, $y$. There are lot of things you can do to that point to move it around space. For example, sliding or translating the point somewhere else can be done by adding some numeric offsets to the points' coordinates:

$$\begin{aligned} x' &= x + \Delta x \\ y' &= y + \Delta y \end{aligned}$$

Scaling the point farther or closer to the origin can be done by multiplying the points' coordinates by some scale factors:

$$\begin{aligned} x' &= x \times \text{scaleX} \\ y' &= y \times \text{scaleY} \end{aligned}$$

Rotating the point around the origin by some number of degrees can be done with help from the trigonometric functions:

$$\begin{aligned} x' &= \cos(\text{angle}) * x - \sin(\text{angle}) * y \\ y' &= \sin(\text{angle}) * x + \cos(\text{angle}) * y \end{aligned}$$

Rotating the point around some point other than origin can be done by combining these ideas. First, we need to make the pivot point the origin, which we can do by subtracting off the pivot point from $x$, $y$. Then we rotate. Then we add the pivot point offsets back on.

These operations have significance beyond manipulating numbers: they are how we move digital things around in our digital worlds. Visit `https://www.desmos.com/calculator/7jz4rbdyqn` to see what happens when we apply these transforms to a series of points.

These four movement operations, amongst others, are what we call *transforms*. There are many meaningful transforms that we might apply to points, and modeling them in code is simpler if we create a generalization of them. We'll do that in this homework. You'll create a `Transform interface`, these four specializations, and a `Point` class that gets transformed.

# 2 Requirements

1. Write all code in package `pre1`.

2. Write a `Point` class, which encapsulates the notion of a 2-D position. It has a default constructor that creates a `Point` at the origin. It has a constructor that accepts `double` x and y parameters. It has getters for the x and y components of the `Point`. It has a method `transform` that accepts a `Transform` parameter and returns the new, transformed `Point`. Method `transform` does not change the invoking object.

3. Write an `interface` named `Transform`. It imposes on its implementers the method `apply`, which takes a `Point` as a parameter and returns the transformed one.

4. Write an implementation of `Transform` named `Translate`. It has a constructor that accepts two `double`s for the shift in x- and y-coordinates that this transform should apply. Its `apply` method returns a new `Point` representing the parameter `Point` shifted by the offsets.

5. Write an implementation of `Transform` named `Scale`. It has a constructor that accepts two `double`s for the x and y scale factors that this transform should apply. Its `apply` method returns a new `Point` representing the parameter `Point` scaled around the origin by the factors.

6. Write an implementation of `Transform` named `Rotate`. It has a constructor that accepts one `double` for the rotation in degrees around the origin that this transform should apply. Its `apply` method returns a new `Point` representing the parameter `Point` rotated around the origin by the specified number of degrees.

7. Write an implementation of `Transform` named `RotateAbout`. It has a constructor that accepts a pivot `Point` and one `double` for the rotation in degrees around the pivot that this transform should apply. Its `apply` method returns a new `Point` representing the parameter `Point` rotated around the pivot by the specified number of degrees. You need do no arithmetic in your implementation; your other classes already do that.

The elegance of using an `interface` for transforms is that we can very uniformly apply a sequence of transformations to a point:

```
Point p = new Point(1, 2);
Transform[] xforms = {new Translate(4, 6), new Scale(1, -1), new Rotate(45)};
for (Transform xform : xforms) {
  p = p.transform(xform);
}
```

## 3 Submission

This is a preassignment, meaning the assignment is less involved than a regular assignment and is automatically graded by the SpecChecker. Humans *will* inspect your work to check for plagiarism.

Speaking of plagiarism, please do your own work. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. (If you find violators of this rule, report them for an extra participation point.) Write your own code.

Put the SpecChecker in your Build Path. Run it as a Java Application (not a JUnit Test) and fix problems until all tests pass. Upload the resulting ZIP file to the W drive.

## 4 Extra

Students who write and post on Piazza a Swing-powered, visual tester of this code will get two extra participation points.

## 5 Files

- `http://www.twodee.org/teaching/cs245/2013C/homework/xform/speccheck_pre1.jar`