Programming Abstractions – Homework III

Dr. Ritwik Banerjee
Computer Science, Stony Brook University

Development Environment: It is highly recommended that you use IntelliJ IDEA. You can, if you really want, use a different IDE, but if things go wrong there, you may be on your own.

Programming Language: Just like the Java portion of your previous assignment, all Java code must be JDK 1.8 compliant.[1]

---

1. Groups and Symmetries

The first part of this assignment requires you to invest quite a bit into thinking about abstraction *before* you start coding. It is based on a mathematical structure called **group**. Before we get to the code, let us define this concept:

A nonempty set of elements G forms a **group** if in G there is a defined binary operation (which we will denote by $\cdot$ in this document), such that

1. $x, y \in G$ implies that $x \cdot y \in G$. This property is called *closure*, and the set of elements is said to be *closed under the operation*.
2. $a, b, c \in G$ implies that $a \cdot (b \cdot c) = (a \cdot b) \cdot c$. In other words, the binary operation is associative.
3. There exists an element $e \in G$ such that $a \cdot e = e \cdot a = a$ for all elements $a \in G$. This special element e is called the *identity* element of the group.
4. For every $a \in G$, there exists an element b such that $a \cdot b = b \cdot a = e$. That is, every element has an *inverse*. Often, we simply denote it by a^{-1}.

Much like programming, mathematics also relies on abstraction. Groups have become fundamentally important in modern mathematics because they distill the basic structural rules found in almost every important mathematical structure. Some are very obvious, such as the set of all integers with addition as the binary operation. Note that the same set is NOT a group with multiplication as the operation (no inverse)! However, as soon as we consider a bigger set, namely, the set of all real numbers, even with multiplication we have a valid group. These examples serve to show that you should not simply think about the set of elements, but instead, carefully consider the binary operation together with the set. It is also important to note that the binary operation may not always be commutative. That is, it is not always the case that $a \cdot b = b \cdot a$.

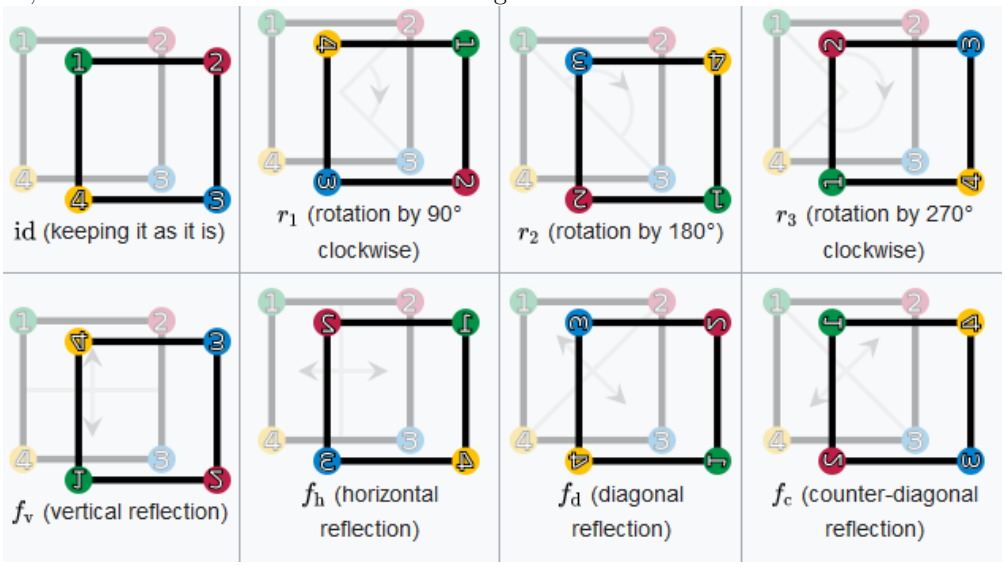
For a basic understanding of implementing simple groups, there is some Java code already given to you. The most important is the interface called `Group`. It is extensively documented, and students are expected to pay attention to the details provided there. Next, there is an implementation of the most obvious group we can think of: the group of all integers under addition. This is provided to you as `ZPlus`[2].

Finite groups: Based on everything you have seen up to this point, you may think that groups are just a fancy way of stating the basic properties of numbers. But that is not at all true! To start with, a group need not be infinite. In fact, you will now be implementing a few finite groups.

[1] You may have a higher version of Java installed, but the "language level" must be set to Java 8. This can be easily done in IntelliJ IDEA by going to "Project Structur" and selecting the appropriate "Project language level". This is a very important requirement, since Java 9 (and beyond) has additional language features that will not compile with a Java 8 compiler.

[2] The name may seem strange, but $\mathbb{Z}$ is the standard mathematical symbol to represent the set of integers. And since addition is the binary operation for this group, we are calling the class `ZPlus`.

Figure 1: The eight symmetries of a square: the identity operation that leaves everything as it is, three rotation operations (around its center by 90°, 180°, and 270°), two reflections around the horizontal and vertical lines, and two reflections around the two diagonals.



Non-commutative groups: As noted earlier, the binary operation of a group need not be commutative. That is, $a \cdot b$ is not always equal to $b \cdot a$. This may not be intuitive if you only think of numeric operations. But they make a lot of sense when we enter the world of geometry. In fact, one of the biggest applications of *group theory* is in fields like chemistry and physics, where the structural symmetry of molecules and particles is studied using this mathematical concept. So much so, that many consider the study of groups to be the "science of symmetry".

For this assignment, we will look at one simple example: the symmetry of squares. But first, another definition: two shapes are said to be **congruent**, if they have the same shape and size. Formally, two shapes are congruent if one can be changed into the other by using a combination of:

(i) rotations (around a fixed point),
(ii) reflections (around a line that serves as the axis of the reflection), and/or
(iii) translations (a transformation that moves every point in the same direction by the same distance).

Clearly, any shape in the 2-dimensional x-y plane is congruent to itself. Some shapes, however, are congruent to themselves in more than one way! Any such "extra" congruence is called a **symmetry**. A square has eight symmetries, as shown in Fig. 1. Similarly, an equilateral triangle has six symmetries (three rotations around its center by 0°, 120°, and 240°, and three reflections around the three perpendicular bisectors).

We are now ready to dive into some actual programming!

1. Let G be the set $\{\pm 1\}$, under the standard multiplication of real numbers. Your first task is to implement this group in Java, with the name `FiniteGroupOfOrderTwo`. When thinking about implementing this, note that `Group` is a parameterized interface. In the implemented example, `ZPlus`, the parameter was obvious, because we already know the data type for "integers" (`Integer`, of course). (20)

   (a) *Choice of data type for the parameter:* Here, the valid data that forms the set of elements, consists of only two values. So the very first thing to do in this question is to correctly choose the data type of the generic parameter. In your implementation, this parameter must be named `PlusOrMinusOne`.

   (b) Your data type must have an operation (in Java, this will be a method) called `toString()`, which is available to instances of this data type and returns only the numeric value as a string (i.e., "1" or "-1").

   (c) Finally, you must ensure that the following driver method (given to you in the `SimpleAlgebraTest` class) works with your code:

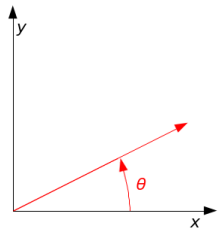


Figure 2: Counter-clockwise rotation through angle θ: the vector is initially aligned with the x-axis, and after the rotation, shown by the red arrow.

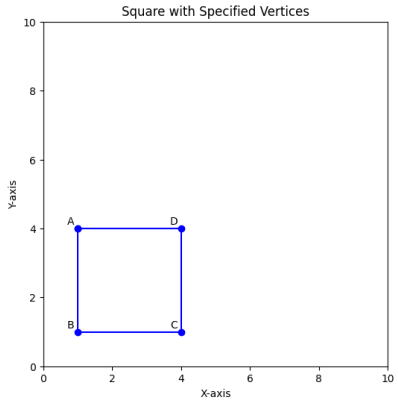


Figure 3: A square with vertices named A at (1,4), B at (1,1), C at (4,1), and D at (4,4).

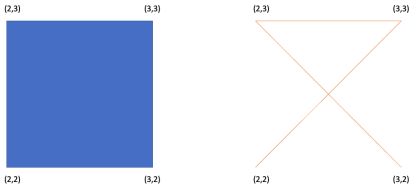


Figure 4: The square (left) initialized by passing arguments $(3,3)$, $(2,3)$, $(2,2)$, and $(3,2)$ (in this order) is, indeed, a valid square. Without such a convention being followed, we could end up with non-polygonal open curves (right). If the constructor is called with four points that do not form a valid square, your constructor must throw an `IllegalArgumentException`.

```
public static void main(String... args) {
    FiniteGroupOfOrderTwo g      = new FiniteGroupOfOrderTwo();
    PlusOrMinusOne[]      values = PlusOrMinusOne.values();
    System.out.printf("g.identity() = %s%n", g.identity());
    for (PlusOrMinusOne u : values) {
        for (PlusOrMinusOne v : values) {
            PlusOrMinusOne e = g.binaryOperation(u, v);
            System.out.printf("%s * %s = %s%n", u.toString(), v.toString(), e.toString());
            System.out.printf("inverseOf(%s) = %s%n", e.toString(), g.inverseOf(e).toString());
        }
    }
}
```

2. There is an interface called `Shape` provided to you. In this question, we will consider a `Square`. Your task in this question is to complete the implementation of the `Square` class, consistent with the requirements of the `Shape` interface. Most of the implementation is straight-forward. Implementing rotation (in the `rotateBy(int degrees)` method), however, requires some mathematics! (20)

   Formally, rotation in the 2-dimensional Euclidean space is defined by a 2×2 matrix. To rotate all the points in the x-y plane counterclockwise by an angle θ (in radians), with respect to the positive x axis about the origin, a point (x, y) is transformed by

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

   Without the matrix notation used in linear algebra, this simply means that such a rotation transforms the point (x, y) to the point $(x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$. Visually, this rotation is shown in Fig. 2.

   To rotate a shape using this formula, you need to ensure that the center of the shape is the origin $(0, 0)$. It is a part of this assignment to figure out how to rotate a shape that has its center somewhere else.

   Your task is to complete the implementation of the `Square` class. An empty constructor is already given. This constructor accepts four `Point` objects as its parameters. It is important to note that whether these points form a valid square may depend on the order in which these points are arranged.

   For this assignment, you can assume that the order in which the input arguments are provided to the constructor, will follow the order specified in the documentation of `Shape#toString()`. For example, the square shown in Fig. 3 is constructed by providing the points D, A, B, and C *in that order*. If the constructor is given the same four locations in any other order, it should simply throw an exception as specified in the code's documentation. *This is really important!* A mistake in ordering the vertices could yield the non-polygonal open curve shown in Fig. 4 instead of the intended square.

You should ensure that the following driver method in the `Square` class works with your code (pay attention to the documentation, which explains what must be printed for the test cases used in this driver method, and what exception must be thrown). You are encouraged to use this as a template and add more tests for (i) the creation of squares, (ii) throwing exceptions as described, (iii) printing a square, (iv) printing a rotated square, (v) printing a translated square, and (vi) calculating the center point.

```
public static void main(String... args) {
    Point  a = new Point("A", 1, 4);
    Point  b = new Point("B", 1, 1);
    Point  c = new Point("C", 4, 1);
    Point  d = new Point("D", 4, 4);

    Point p = new Point("P", 0.3, 0.3);

    Square sq1 = new Square(a, b, c, d); // throws an IllegalArgumentException
    Square sq2 = new Square(d, a, b, c); // forms a square
    Square sq3 = new Square(p, p, p, p); // forms a "trivial" square
                                         // this is a limiting case, but still valid

    // prints: [(D, 4.0, 4.0); (A, 1.0, 4.0); (B, 1.0, 1.0); (C, 4.0, 1.0)]
    System.out.println(sq2);

    // prints: [(C, 4.0, 4.0); (D, 1.0, 4.0); (A, 1.0, 1.0); (B, 4.0, 1.0)]
    // note that the names denote which point has moved where
    System.out.println(sq2.rotateBy(90));

    // you should similarly add tests for the translateBy(x, y) method
}
```

3. Now, we will use the squares to add the concepts of their symmetry. (20)

   Take a look at the `GeometryTest` class' `main(String[])` method. This is provided to you as a driver method outlining some tests for your code. Here, you will see a class called `SquareSymmetries` being mentioned. You will also see two methods being used: `areSymmetric`, and `symmetriesOf`. Carefully consider the `Symmetries` interface implemented by these two classes, and come up with the correct signatures for these methods in the implementations. In particular, you need to ask

   > *If the definition in the interface (i.e., the supertype) specifies returning a type `T`, can the method implementation in the class (which is its subtype) return a subtype of `T`? In other words, does Java allow covariant return types?*

   As shown in Fig. 1, a square has eight symmetries (including the identity transformation). Your task in this question is to implement the symmetries of `Square` in the class `SquareSymmetries`.

2. Higher order functions

Code for this second part of the assignment must be written in a file named `HigherOrderUtils.java`. You may also have to consult some of the official Java documentation and/or the reference text on Functional Programming in Java. The remaining answers need not be written as a single method chain. *Be very careful with the parameterized types in this section, and pay attention to the warnings issued by your IDE.*

4. First, write a static nested interface in `HigherOrderUtils` called `NamedBiFunction`. This interface must extend the interface `java.util.Function.BiFunction`. The interface should just have one additional method declaration: `String name();`, i.e., a class implementing this interface must provide a "name" for every instance of that class. Then, create `public static NamedBiFunction` instances as follows: (20)
   (a) `add`, with the name "plus", to perform addition of two `Double`s.
   (b) `subtract`, with the name "minus", to subtract the second `Double` from the first.
   (c) `multiply`, with the name "mult", to perform multiplication of two `Double`s.

(d) `divide`, with the name "div", to divide the first `Double` by the second. This operation should throw a `java.lang.ArithmeticException` if there is a division by zero being attempted.

5. Write a method called zip, defined as follows: (20)

```
/**
 * Applies a given list of bifunctions -- functions that take two arguments of a certain type
 * and produce a single instance of that type -- to a list of arguments of that type. The
 * functions are applied in an iterative manner, and the result of each function is stored in
 * the list in an iterative manner as well, to be used by the next bifunction in the next
 * iteration. For example, given
 * List<Double> args = Arrays.asList(-0.5, 2d, 3d, 0d, 4d), and
 * List<NamedBiFunction<Double, Double, Double>> bfs = Arrays.asList(add, multiply, add, divide),
 * <code>zip(args, bfs)</code> will proceed as follows:
 * - the result of add(-0.5, 2.0) is stored in index 1 to yield args = [-0.5, 1.5, 3.0, 0.0, 4.0]
 * - the result of multiply(1.5, 3.0) is stored in index 2 to yield args = [-0.5, 1.5, 4.5, 0.0, 4.0]
 * - the result of add(4.5, 0.0) is stored in index 3 to yield args = [-0.5, 1.5, 4.5, 4.5, 4.0]
 * - the result of divide(4.5, 4.0) is stored in index 4 to yield args = [-0.5, 1.5, 4.5, 4.5, 1.125]
 *
 * @param args the arguments over which <code>bifunctions</code> will be applied.
 * @param bifunctions the list of bifunctions that will be applied on <code>args</code>.
 * @param <T> the type parameter of the arguments (e.g., Integer, Double)
 * @return the item in the last index of <code>args</code>, which has the final result
 * of all the bifunctions being applied in sequence.
 *
 * @throws IllegalArgumentException if the number of bifunction elements and the number of argument
 * elements do not match up as required.
 */
public static <T> T zip(List<T> args, List<BiFunction<T, T, T>> bifunctions);
```

Now, is this method following the principles of functional programming? If yes, your implementation should also be true to the principles of the functional programming paradigm. Otherwise, you need not abide by them when implementing `zip`. Next, notice that the documentation mentioned bifunctions, and the method signature uses `BiFunctions`, but the example provided in the documentation uses `NamedBiFunctions`. Your implementation should be able to handle both! Specifically, the following code should work with your implementation:

```
public static void main(String... args) {
  List<Double> numbers = Arrays.asList(-0.5, 2d, 3d, 0d, 4d); // documentation example
  List<NamedBiFunction<Double, Double, Double>> operations = Arrays.asList(add,multiply,add,divide);
  Double d = zip(numbers, operations); // expected correct value: 1.125
  // different use case, not with NamedBiFunction objects
  List<String> strings = Arrays.asList("a", "n", "t");
  // note the syntax of this lambda expression
  BiFunction<String, String, String> concat = (s, t) -> s + t;
  String s = zip(strings, Arrays.asList(concat, concat)); // expected correct value: "ant"
}
```

Note: In order to have the above code compile and run correctly, you will need to make a very specific change to the signature of the `zip` method.

- Please keep in mind these points in the syllabus regarding homework assignments.
- You may have additional methods in your classes even if such methods are not required by the interface. Such additional methods, however, must not be `public`.
- Any x or y value for a point must be rounded to two decimal places if it is returned by a public method (such returned values will be printed by test codes, so you must take care that printed values you see in any driver method are always rounded to two decimal places).
- Rotations use trigonometric functions, and may return floating-point values that do not exactly match the correct value (e.g., you may have 2.99999... instead of 3). This is a known issue with floats, so we have to work around it. For this assignment, you may treat any difference of < 0.001 to be irrelevant. For example, if you rotate a shape by some degree and expect it to land at (0,1) but your code results

in (0, 0.999...), you should treat that as the correct answer. In any string representation of this shape, however, remember to round it to two decimal places and display 1 (1.0 or 1.00 are also acceptable).

- Any interface code given to you must not be changed!
- **What to submit?** The complete codebase (including classes and interfaces that were already given to you) as a single `.zip` file. Do NOT include anything other than `.java` files in your submission! *Deviations from the expected submission format carries varying amounts of score penalty (depending on the amount of deviation)*:
  - Java code is not JDK 1.8 compliant, i.e., some other Java version is required to compile your code. *Penalty: 8 points.*
  - Codebase does not have the specified structure in the .zip file. *Penalty: 5 points.*
  - Submission includes the entire project including files other than Java files (e.g., compiled .class files, classpath information, or other settings local to your project in your computer). *Penalty: 5 points.*

Due by November 29 (Friday), 11:59 pm.