CSE 216 – Homework II

Dr. Ritwik Banerjee
Computer Science, Stony Brook University

In this assignment, you will be using Java (make sure it is JDK 1.8 compliant) for programming. Moreover, 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`[1].

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.

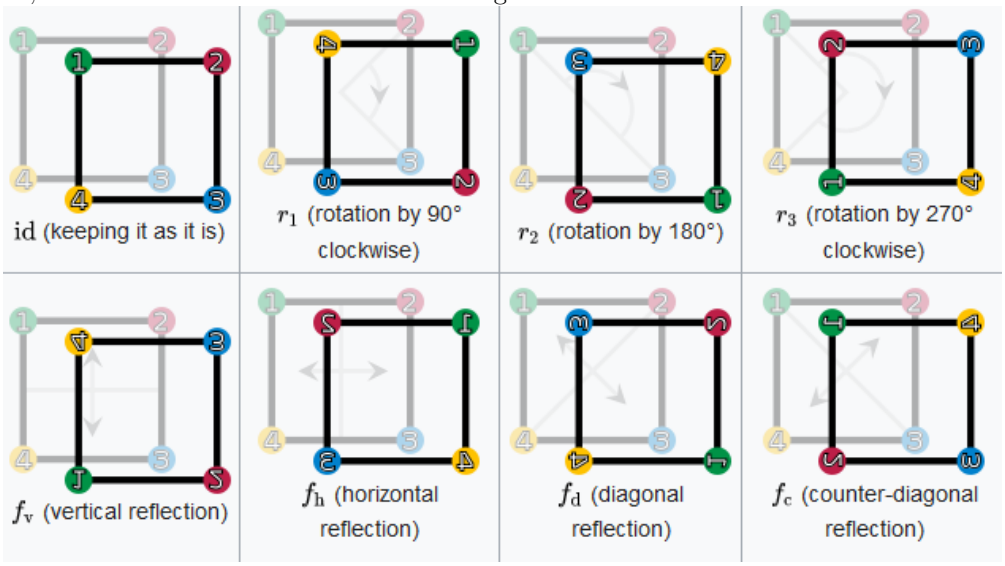
Non-commutative groups

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 two very simple examples – an equilateral triangle and a square – and their symmetries.

[1] 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, I decided to call 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.



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).

With this background, 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`. (20)

   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). But here, the valid data that forms the set of elements, consists of only two values. So think about what should be the data type of the generic parameter. In your implementation, this parameter class must be named `PlusOrMinusOne`. You should also ensure that the following driver method (given to you in the `ArithmeticTest` class) works with your code:

```
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());
        }
    }
}
```

   In the above code, `toString()` must return only the numeric value (i.e., "1" or "-1").

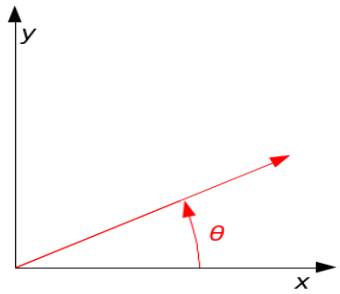


Figure 2: Counterclockwise rotation through angle θ: the vector is initially aligned with the x-axis, and after the rotation, shown by the red arrow.

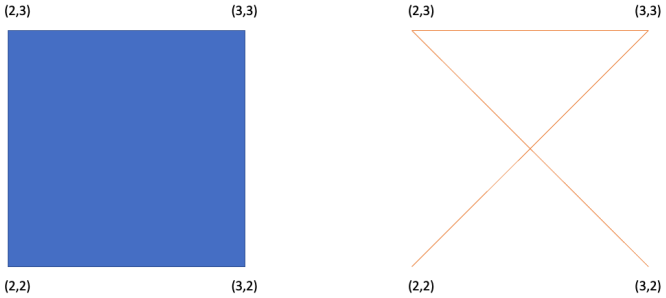


Figure 3: Counterclockwise ordering, as required by this assignment, yields the square (left) while a different ordering could yield a non-polygonal open curve (right). The ordering required by the `Square` class implementation in this case is $(2, 2)$, $(3, 2)$, $(3, 3)$, $(2, 3)$.

2. In the `geometry` package, you will notice an interface called `Shape`. Complete the implementation of the `EqTriangle` class, consistent with the requirements of this 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.

3. Similarly, complete the implementation of the `Square` class. (20)

   Here, the order in which the corner vertices are considered may determine if they form a valid shape. This was not a concern for triangles, but it does matter for a square! For example, a mistake in ordering the vertices could yield the right-side non-polygonal open curve shown in Fig. 3.

   You may have realized that this is not a concern only for squares, but for any polygon with more than three sides. You can, of course, write new code for every single polygonal shape separately. But isn't it better to write a single piece of code that orders a collection of vertices in the counterclockwise manner (as defined in the documentation of `Square#isMember()`)? Further, consider this scenario: what if in future, you require a different way of ordering the vertices?[2]

   For this, take a look at `Collections#sort()` provided by Java. This is an overloaded method. Your task is to figure out which of the overloaded methods you need, and then write a class called `Counterclockwise`, which can be used by this `Collections#sort()` method to order a collection of vertices. Keep in mind that your code may be tested with criteria different from counterclockwise ordering (as mentioned in the footnote below). It may also be tested separately with a list of vertices (i.e., just to test if your ordering works ... independent of the `Square` class).

   [**Hint:** Your `Counterclockwise` class needs to implement an interface, so that you can provide your own *customized* way of comparing vertices. Keep in mind that most sorting algorithms are *comparison sorts*, so the final ordering of a collection of vertices gets decided by the way two vertices are compared.]

[2] This is not a far-fetched concern. Imagine you are a software engineer, and are developing an application for a client. Initially, the client states that they need counterclockwise ordering, but later, they come and say that they need clockwise ordering (or some other ordering).

4. Now, we will use these two shapes and add the concepts of their symmetry and one important fact: the set consisting of an equilateral triangle and its symmetries forms a group under the six rotation and reflection operations. (20)

   Take a look at the `GeometryTest` class' `main(String[])` method. This is provided to you as an outline for testing your code. Here, you will see a class being mentioned, called `TriangleSymmetries`. You will also see two methods being used: `areSymmetric`, and `symmetriesOf`. Carefully consider the `Symmetries` interface implemented by this class, and come up with the correct signatures for these methods in the `TriangleSymmetries` implementation. 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?*

5. Similarly, implement the class `SquareSymmetries`. (20)

In the last two questions, you may have additional methods in your classes even if such methods are not required by the `Symmetries` interface.

- Please keep in mind these homework-related points mentioned in the syllabus.
- **What to submit?** The complete codebase (including classes and interfaces that were already given to you) as a single `.zip` file. Your zip file, once extracted, must contain two folders: `arithmetic`, and `geometry`. Your solution to the first question is expected to be in the `arithmetic` package, while the rest of your code must be in the `geometry` package.
  Deviations from the expected submission format carries varying degrees of score penalty (depending on the amount of deviation).

Submission Deadline: Apr 11 (Monday), 11:59 pm