Implementing an Immutable Data Type

Summary: Using immutability, we will construct a *reliable* class for handling matrices. We will also discuss a basic suite of tests for this new class.

1 Background

Our goal will be to implement an "immutable matrix ADT". Although some of that terminology should be familiar, some may not be. There is an optional appendix later in this document, that discusses mutability in a bit more detail than the textbook. The class we will construct will support computing basic matrix operations like addition and subtraction.

At first, a class for handling matrices might seem a little boring. However, matrices are an extremely valuable mathematical tool. Many algorithms in mathematics and scientific computing are built in terms of matrix operations. Matrices serve as a common language in which many mathematical problems many be expressed. The idea being if we have many problems that can be solved in the same way (i.e., a bunch of matrix operations), then we need only devise a single program (one that processes matrices) in order to solve all problems that are stated in that way... This is nice because we can optimize the piece of software that works with matrices, and see speedups for every problem that is expressed as matrices. Anyway, matrices are very common - computing matrix operations is pretty much all supercomputers do. Thus, implementing a matrix is useful - in fact, libraries that provide linear algebra operations (e.g., LINPACK) are among the most widely used in scientific and engineering research. Finding the fastest algorithms to compute matrix operations is very active area of research. People have built entire careers out of designing fast matrix processing algorithms!

This document is separated into four sections: Background, Requirements, Testing, and Submission. You have almost finished reading the Background section already. In Requirements, we will discuss what is expected of you in this homework. In Testing, we give some sample testing code for the matrix class, discuss how it works, and give some general pointers on how tests should be designed. Lastly, Submission discusses how your source code should be submitted on BlackBoard. At the very end of the document, there is an Appendix with optional reading on mutability. If you are confident with mutability, you may skip it.

2 Requirements

For this assignment, you will implement the provided matrix interface (see Matrix.java). This interface file defines not only which methods you must implement, but gives documentation that describes how they should work. Already provided for you is a base file to modify (ser222_01_02_hw02_base.java). The base contains an currently empty matrix class as well as some simple testing code. In order for it to be a reliable type, we will implement it as an immutable type. Creating this ADT will involve creating 9 methods:

- MyMatrix(int[][] matrix) - a constructor. [4 points]
- public int getElement(int y, int x) - see interface. [2 points]
- public int getRows() - see interface. [1 points]
- public int getColumns() - see interface. [1 points]
- public MyMatrix scale(int scalar) - see interface. [3 points]
- public MyMatrix plus(Matrix other) - see interface. [3 points]

- public MyMatrix minus(Matrix other) - see interface. [3 points]
- public MyMatrix multiply(Matrix other) - see interface. [5 points]
- boolean equals(Object other) - see interface. [5 points] (Hint: see Point2D from earlier.)
- String toString() - see interface. [5 points].

Be aware that some of the methods throw exceptions - this should be implemented.

3 Testing

Whenever you build a piece of software, you want to have some level of certitude that that piece of software does what you expect. This means testing! For this initial homework, you are provided with a set of simple tests to check if your program is functioning correctly. In the future, you may need to create your own tests. The following code is included in base file:

```
int[][] data1 = new int[0][0];
int[][] data2 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int[][] data3 = {{1, 4, 7}, {2, 5, 8}, {3, 6, 9}};
int[][] data4 = {{1, 4, 7}, {2, 5, 8}, {3, 6, 9}};

Matrix m1 = new ser222_01_02_hw02_base(data1);
Matrix m2 = new ser222_01_02_hw02_base(data2);
Matrix m3 = new ser222_01_02_hw02_base(data3);
Matrix m4 = new ser222_01_02_hw02_base(data4);

System.out.println("m1 --> Rows: " + m1.getRows() + " Cols: " + m1.getColumns());
System.out.println("m2 --> Rows: " + m2.getRows() + " Cols: " + m2.getColumns());
System.out.println("m3 --> Rows: " + m3.getRows() + " Cols: " + m3.getColumns());

//check for reference issues
System.out.println("m2 -->\n" + m2);
data2[1][1] = 101;
System.out.println("m2 -->\n" + m2);

//test equals
System.out.println("m2==null: " + m2.equals(null));                 //false
System.out.println("m3==\"MATRIX\": " + m2.equals("MATRIX"));       //false
System.out.println("m2==m1: " + m2.equals(m1));                     //false
System.out.println("m2==m2: " + m2.equals(m2));                     //true
System.out.println("m2==m3: " + m2.equals(m3));                     //false
System.out.println("m3==m4: " + m3.equals(m4));                     //true

//test operations (valid)
System.out.println("2 * m2:\n" + m2.scale(2));
System.out.println("m2 + m3:\n" + m2.plus(m3));
System.out.println("m2 - m3:\n" + m2.minus(m3));

//not tested... multiply(). you know what to do.

//test operations (invalid)
//System.out.println("m1 + m2" + m1.plus(m2));
//System.out.println("m1 - m2" + m1.minus(m2));
```

The results from running this code should be:

```
m1 --> Rows: 0 Cols: 0
m2 --> Rows: 3 Cols: 3
m3 --> Rows: 3 Cols: 3
m2 -->
1 2 3
4 5 6
7 8 9

m2 -->
1 2 3
4 5 6
7 8 9

m2==null: false
m3=="MATRIX": false
m2==m1: false
m2==m2: true
m2==m3: false
m3==m4: true
2 * m2:
2 4 6
8 10 12
14 16 18

m2 + m3:
2 6 10
6 10 14
10 14 18

m2 - m3:
0 -2 -4
2 0 -2
4 2 0
```

Let's consider this code and its output. The first few lines of the code just create matrix data using 2D arrays. These are just setting up for our new class, we aren't actually using it yet. The next step sees us creating a Matrix object for each array (m1, m2, m3). Next, we get to the actual testing. First, we display the size (rows and columns) for each matrix. The values we get should match the dimension of the 2D arrays that were created (0x0, 3x3, 3x3).

The next check is to see if we have correctly implemented our matrix as an immutable class. We display the contents of m2, make change in data2 (that was used to create m2), and then display the contents of m2 again. Recall that an object that is immutable should not change state - good thing that's what we see in the output. The middle value does not change from 5 to 101. If it was not an immutable object, then changing the value inside of data2 would have the (potentially unintended) side effect of changing what is stored in m2. (Remember that arrays, like data2, are passed as references.)

Next, we check if equals has been implemented properly. Based on what's being compared, equals should return true or false. If we compare a matrix to a null variable, a string, or a matrix containing different values, then it should return false. If we compare it to itself, or a matrix containing the same data, then it should return true.

Having verified that the class is immutable and supports comparison, we check if it supports the operations that should be implemented. For this homework, you are being given the output of these operations, and so can check you result versus ours. In the future, this might not be the case. You may have to run an algorithm by hand to determine what output should be expected. Note that the process of running it by hand will help you to understand the data structure (or algorithm) and may make it easier to implement. In addition to testing matrix operations that should work, we should also test operations that do not work.

In this class, that would be when trying to add or subtract matrix with different size. For now, this code is commented, since running it should throw an exception.

Some basic principles: 1) test every method that is implemented. This gives us a basic level of code coverage (the amount of code used by tests) over a significant fraction of the code base. It would be even better if we had test cases that used each possible line of code in our program but that can become very difficult. 2) Test "edge cases". That is, potentially special values that may invoke uncommon parts of the code base. For example, one of the first tests is to create a 0x0 matrix. While this matrix might not be practical, its parameters are unique and may cause issues. Typically one aims to test where an input changes significantly. For example, if the input was an integer, we might test some large position number, 1, 0, -1, and some large negative number. Note that we don't really need to test several large numbers. Chances are, if one works, then the others will as well. We want to minimize our testing work by looking inputs as their character changes (e.g., goes from positive to negative in this example).

4 Submission

The submission for this programming assignment has only one part: source code submission.

Writeup: For this assignment, no write up is required.

Source Code: Please rename and submit your code as "LastNameMatrix.java" (e.g. "AcunaMatrix.java"). The class inside of your file will also need to be renamed from ser222_01_02_hw02_base. The class must be in the default package. There is no need to turn in the interface file (Matrix.java).

5 Appendix: Reference Types and Mutability

In this appendix, we review the idea of reference types in Java, and how mutability is impacted by it. Recall the Point2D example from our discussion of abstract data types. Consider what would happen if we ran the following code that used Point2D:

```
public class PointTester  {
    public static void doSomething(Point2D p) {  //a 'user' of the point
        p.scale(5);
    }
    public static void main(String [ ] args) {   //the 'owner' of the point
        Point2D point = new Point2D(10, 10);

        System.out.println(point.toString());
        doSomething(point);
        System.out.println(point.toString());
    }
}
```

If we were to run this, we would get:

```
X: 10 Y: 10
X: 50 Y: 50
```

As you probably suspected, even though the call to scale happens inside of the doSomething() method instead of in the main, when main prints out the contents of the point for the second time, the x and y values have changed. This happens because classes are what is called a ***reference type***. For primitive types, such as int or double, when variables are passed around as parameters, their value is copied. Changing a int parameter in a method won't change it's value in the original caller. However, objects are not copied. Instead, Java passes a copy of the *address* where the object is stored in system memory. Since an address is passed, the method will know here to go and access the information (the *state*) that defines the class. In our example, when doSomething() is called , it is passed (via a parameter) the address of the Point2D that the main created. When the call to scale() is made on p inside of doSomething(), it changes the x and y values inside of the Point2D object that main created. Thus, when doSomething() completes and main

displays the x/y values, it shows them as being five times larger. (Note that in addition to objects, arrays are also treated as reference types.) Thus we can see that the Point2D class is ***mutable,*** i.e. it's contents can change. The opposite of this is an ***immutable*** class. At first, having a mutable class seems completely reasonable - how else would implement something like scaling the point? Why would you even want to?

Let's start with the second question: Say we're using the Point2D class in a game, and doSomething() is a method that displays a zoomed in version of the screen. If we're implementing zoom, so everything appears larger, it makes sense it would scale the point to be larger. What happens if we run doSomething() to zoom in, but doSomething() lacks the code (e.g., p.scale(1/5)) to undo the zoom operation? Well, in that case doSomething() would finish without any issue, but then over in main, we would be left with the enlarged point. Not good! This happened because main allowed doSomething() to "mess around" with the information inside the point. Consider another potential issue, this time in a scientific calculation. If you have a massive matrix storing the simulation data for the analysis of a cancerous protein, you don't want to accidentally change the data while analyzing it - that would ruin the results! Our goal should be to design reliable a data type (class) that does not allow that to occur... an immutable one.

The basic idea of a ***immutable*** class isn't that complicated: a class without any mutators, or shared references. But then how do we implement scale? Instead of having methods like void scale(double factor) that change the contents of the object, we will have methods like Point2D scale(double factor) that return a new copy of the point. Now, it is true that using immutable types can make your code must more safe (less error prone), but should remember that it will typically be slower than mutable implementation. (FYI, if you go on to study programming languages, you will encounter a programming language called LISP where all data is immutable.) It can be relatively straightforward to write an immutable type, the following needs to happen:

- Mutators: These must be removed - there should be no way for someone to change the values inside of the class after it has been created. (When you go to complete the homework, you will find this has already been set up for you in the interface file.)
- Constructors: We must make sure that no "reference types" (e.g., another object, or an array) leak into the object via the constructor. If this happens, our class will contain as instance data, some other object which is not immutable AND which may be accessible by another part of the program. Then, our object as a whole won't be immutable either. The solution is to manually make a copy of any objects or arrays that are passed into the class we are building. The copy will then be unique to us, and will be effectively immutable (assuming we also make sure to leave it alone).

Consider the task making Point2D immutable. There are three methods we would need to change: setX, setY, and scale. These are only methods that can change the state of the point. Previously setX(int X) would change the value of the x member variable. When refactoring this object to immutable, we rewrite setX() so it returns a Point2D (instead of void), where the new point is built from the new x value being set and the existing y value. The same thing happens for setY(). scale() is a little different. There, we must change it to return a new point, but both x and y will change since they are both scaled. Below is shown the immutable Point2D class:

```
public class Point2DImmutable {
    int x, y; //can even make "final"

    public Point2D(int x, int y) {setX(x); setY(y); }
    Point2D setX(int newX) { return new Point2D(newX, getY()) }
    int getX() {return x; }
    Point2D setY(int newY) { return new Point2D(getX(), newY) }
    int getY() {return y; }
    Point2D scale(double fac) {return new Point2D(getX() * fac, getY() * fac);}
    String toString() { return "X: " + getX() + ", Y: " + getY;}
    //note: equals is omitted because it doesn't change.
}
```

Compare this with our previous implementation and notice how the methods have changed.