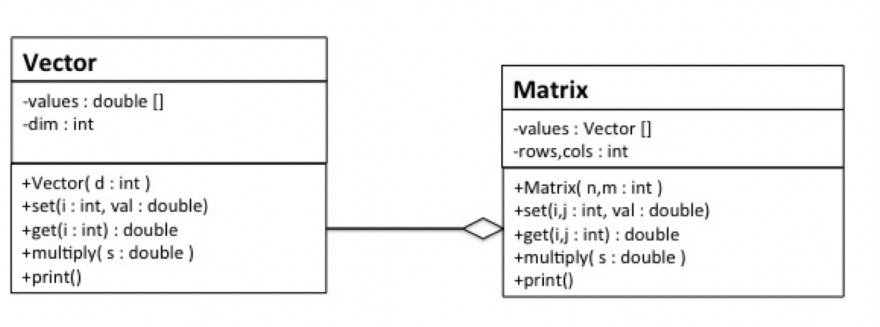
Lab Session 4: Implementing different classes of a design that uses inheritance

1.Introduction

In this lab we implement the design of Seminar 4 consisting of the Vector, Matrix, AudioBuffer, Frame, BWFrame and ColorFrame classes, as a way of using inheritance and code reuse. It's the first lab session which hasn't handled us an initial code and we'd have to create it from scratch. We take into account the benefits of reusing the classes of Vector and Matrix.Let's start.

2.Vectors and Matrix



First we create the classes Vector and Matrix that are related by composition.

Which Vector Class had to accomplish this functions :

1. The constructor having as parameter the dimension of the vector.
2. Getters and setters.
3. An operation that sets all the values of the vector to zero. This operation can be called by the constructor, after creating the array.
4. An operation that multiplies all of the vector values by a scalar.
5. A print operation.

For the Matrix Class, similar functions:

1. The constructor with both dimensions n,m as parameters.
2. Getters and setters.
3. An operation to set all its elements to 0, The operation can call the vector operation that you created.
4. Multiply by a scalar each of its elements.
5. A print operation.

Therefore following the patterns established and the diagram given:

```java
public class Vector{

    private double[] values;
    private int dim;

    public Vector(int dim){
        this.dim = dim;
        values = new double[dim];
    }

    public void set(int i, double val){
        values[i] = val;
    }

    public double get(int i){
        return values[i];
    }

    public void multiply(double s){
        for(int i = 0; i<values.length; i++){
            values[i] *= s;
        }
    }
```

```java
public void zero(){
    for(int i = 0; i<dim; i++){
        set(i, val: 0);
    }
}

//prints in different lines
public void print(){
    for(int i = 0; i<values.length; i++){
        if (i == 0){
            System.out.println("[" + values[i] +  ",");
        }
        else if (i == (values.length -1)){
            System.out.println(values[i] + "]");
        }
        else{
            System.out.println(values[i] + ",");
        }
    }
}
```

```java
public class Matrix {

    private Vector[] values;
    protected int rows;
    protected int cols;

    public Matrix(int n, int m){
        rows = n;
        cols = m;
        values = new Vector[n];
        for(int i = 0; i<n; i++){
            values[i] = new Vector(m);
        }
    }

    public void set(int i, int j, double val){
        values[i].set(j, val);
    }

    public double get(int i, int j){
        return values[i].get(j);
    }
```

```java
public void multiply(double s){
    for(int i = 0; i<rows; i++){
        values[i].multiply(s);
    }
}

public void zero(){
    for(int i = 0; i<rows; i++){
        for(int j = 0; j<cols; j++){
            set(i, j, val: 0);
        }
    }
```

Once our Classes are done, we try to test them in the main….

```
Vector v = new Vector(dim: 3);
v.set(i: 0, val: 1);
v.set(i: 1, val: 2);
v.set(i: 2, val: 3);
v.printVector();
System.out.println(x: "----");
v.zero();
v.printVector();
System.out.println(x: "----");
```

```
Matrix m = new Matrix(n: 2, m: 2);
m.set(i: 0, j: 0, val: 0);
m.set(i: 0, j: 1, val: 0);
m.set(i: 1, j: 0, val: 0);
m.set(i: 1, j: 1, val: 1);
m.print();
System.out.println(x: "----");
m.zero();
m.print();

System.out.println(x: "----");
```

And the results are accurate.

```
[1.0, 2.0, 3.0]
----
[0.0, 0.0, 0.0]
----
```

```
[0.0, 0.0]
[0.0, 1.0]
----
[0.0, 0.0]
[0.0, 0.0]
```

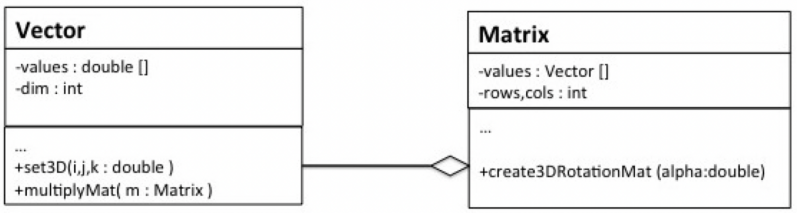
We assume the coded classes are done correctly and follow to the next step.

2.1. Application and testing using Rotation Matrices

In this section we're going to rotate a matrix using the formula:

$$R_z(90°) = \begin{bmatrix} \cos 90° & -\sin 90° & 0 \\ \sin 90° & \cos 90° & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If we multiply any column vector with this, we'll get the vector rotated by the angle it was created. In order to achieve this, we'll add new methods to both our main classes (Vector and Matrix).



So again following the diagram and the new given methods, we amplify our classes like…

For Vector new methods:

set3D(): function that creates a vector of 3 positions (array of 3).

```
public void set3D(double i, double j, double k){
    set(i: 0, i);
    set(i: 1, j);
    set(i: 2, k);
}
```

multiplyMat(): multiply an input matrix by our matrix, so this function has to run our matrix of 3x3 and multiply each element by the input element, when it's done this has to set the new values of the matrix.

```
public void multiplyMat(Matrix m){
    if(m.rows == this.dim){
        Vector v = new Vector(dim: 3);
        for(int i = 0; i<3; i++){
            double sum = 0;
            for(int j = 0; j<3; j++){
                double m1 = m.get(i, j);
                double v1 = get(j);
                sum += v1*m1;

            }
            v.set(i, sum);
        }
        for (int i = 0; i < 3; i++){
            set(i, v.get(i));
        }
    } else {
        System.out.println(x: "Dimensions don't match. Cannot multiply")
    }

}
```

Computing this new methods in the main:

```
Vector v2 = new Vector(dim: 3);
v2.set3D(i: 1, j: 0, k: 0);
System.out.println(x: "\nVector: ");
v2.printVector();
v2.multiplyMat(m2);
System.out.println(x: "\nMatrix*vector: ");
v2.printVector();
```

```
Vector:
[1.0, 0.0, 0.0]
```

```
Matrix*vector:
[0.0, 1.0, 0.0]
```

For Matrix new method, we'll use auxiliary math operations calling math library:

create3DRotationMat(): we set our matrix to be empty using function zero() just to facilitate the process, following the formula given in the report(rotationmat) we will run our input matrix value per value setting each element like the given formula. So for this function the input element will be $\frac{\pi}{2}$ (math.pi/2) , we know $cos(\frac{\pi}{2}) = 0 \ and \ sin(\frac{\pi}{2}) = 1$, therefore we compute this method like…

$$R_z(90°) = \begin{bmatrix} \cos 90° & -\sin 90° & 0 \\ \sin 90° & \cos 90° & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
public void create3DRotationMat(double alpha){
    zero();
    set(i: 0,j: 0, Math.round(Math.cos(alpha)));
    set(i: 0,j: 1, (-1)*Math.sin(alpha));
    set(i: 1,j: 0, Math.sin(alpha));
    set(i: 1,j: 1, Math.round(Math.cos(alpha)));
    set(i: 2,j: 2,val: 1);

}
```

so we try this in the main, as an accurate result…

```
Matrix m2 = new Matrix(n: 3, m: 3);
m2.create3DRotationMat(Math.PI/2);
System.out.println(x: "Matrix: ");
m2.print();
```

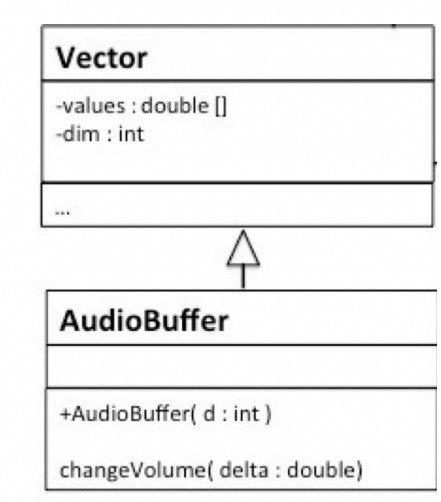
```
Matrix:
[0.0, -1.0, 0.0]
[1.0, 0.0, 0.0]
[0.0, 0.0, 1.0]
```

3. AudioBuffer Class

Vector class has a child class that inherits from them, AudioBuffer class, has to implement this functions:

1.A constructor.

2.changeVolume: this operation only multiplies the vector values by some positive real number greater than one to increase it, or smaller than one to decrease it. The multiply by scalar operation from Vector must be used.



With this information, we recreate this like…

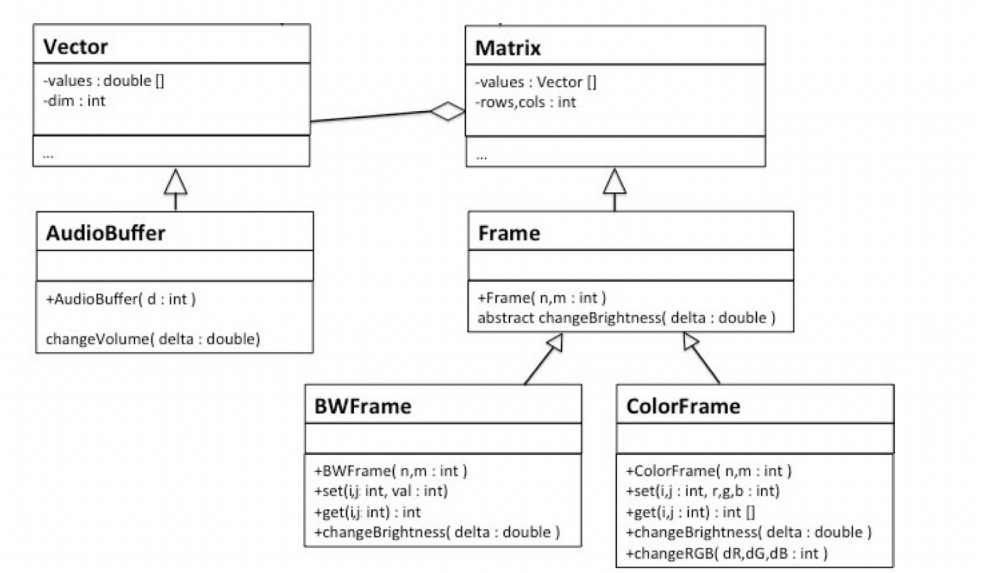
```
public class AudioBuffer extends Vector{

    public AudioBuffer(int dim){
        super(dim);
    }

    public void changeVolume(int num){
        super.multiply(num);
    }
```

Notice we use keywords **super()** and **extends()** to refer to attributes or the parent class itself.

4. Frame, BWFrame and ColorFrame classes



Lastly, we create the rest of the classes, Frame, BW Frame and ColorFrame, their parent class in this case is Matrix. So the requirements for this classes:

- Define and implement all the constructors.
- We want to define specific setters and getters for BWFrame and Color- Frame. Look at the design figure to see their signature. Remember that if a method has the same name and different signature we call that over- loading.
- changeBrightness: will increase or decrease the brightness of the image frame by a delta percentage double value. It can be and must be defined to color and black and white frames: for this purpose we will declare it as abstract. Remember that when a method is redefined with the same signature we call that overriding. As changeVolume this method will use the operation multiply by a scalar to change the pixel values.
- changeRGB (exclusive of color frame): has three parameters to increase or decrease the R,G,B components of the image. The operation can only be applied to color frames.

For Frame Class, inherited from Matrix, we use ***extends***() and ***super***() again .

```
public abstract class Frame extends Matrix {

    public Frame(int n, int m){
        super(n,m);
    }

    abstract void changeBrightness(double delta);
}
```

Notice we use <<*abstract*>> to define ***changeBrightness()*** as a method of a class that doesn't have any implementation or definition, we just declared it.

Then BWFrame class which inherits from Frame:

```java
public class BWFrame extends Frame {

    public BWFrame(int n, int m){
        super(n, m);
    }

    @Override
    public void changeBrightness(double delta){
        super.multiply(delta);
    }

    public void set(int i, int j, int val){
        super.set(i, j, val);
    }

    public int getBW(int i, int j){
        return (int)super.get(i, j);
    }


}
```

Referring to the methods, ***changeBrightness***() is a NON abstract method and we override to redefine it because there's the same method with the same signature located in their parent class.

For the ColorFrame class, it's also a child from Frame.

```java
public class ColorFrame extends Frame{

    public ColorFrame(int n, int m){
        super(n,m);
    }

    private int[] valToRGB(double rgb){
        int[] ret = new int[3];
        ret[0] =  ((int) rgb >> 16 ) & 255;
        ret[1] = ((int) rgb >> 8 ) & 255;
        ret[2] = ((int) rgb) & 255;
        return ret;

    }

    private double RGBToVal(int r, int g, int b){
        double ret = (r<<16) | (g<<8) | b;
        return ret;
    }

    public void set(int n, int m, int r, int g, int b){
        double val = RGBToVal(r, g, b);
        super.set(n, m, val);
    }

    public int[] getRGB(int i, int j){
        double val = super.get(i, j);
        return valToRGB(val);
    }
```

```java
@Override
public void changeBrightness(double delta) {
    for(int i = 0; i<super.rows; i++){
        for(int j = 0; j<super.cols; j++){
            int[] rgb = getRGB(i, j);
            int r = (int) (rgb[0]*delta);
            int g = (int) (rgb[1]*delta);
            int b = (int) (rgb[2]*delta);
            Double newval = RGBToVal(r, g, b);
            if((0<=r && r<=255) && (0<=g && g<=255) && (0<=b && g<=255)){
                set(i,j,newval);
            }
        }
    }
}

public void changeRGB(int dr, int dg, int db){
    for(int i = 0; i<super.rows; i++){
        for(int j = 0; j<super.cols; j++){
            int[] rgb = getRGB(i, j);
            int r = (int) (rgb[0]*dr);
            int g = (int) (rgb[1]*db);
            int b = (int) (rgb[2]*db);
            Double newval = RGBToVal(r, g, b);
            if((0<=r && r<=255) && (0<=g && g<=255) && (0<=b && g<=255)){
                set(i,j,newval);
            }
        }
```

Following the guide of the auxiliary functions ***ValToRGB***() and ***RGBToVal***(), that are more complex functions, we can use them to create the other ones, ***set***() , ***get***() , ***changeBrightness***() and ***changeRGB***(). The first function valToRGB is given a double value and returns an array with three elements, representing the corresponding R,G,B components. The second function RGBToVal is given the three R,G,B components and returns its corresponding double representation.

The method ***changeBrightness()***, have to run over all elements (using loop like for) to extract de R,G,B components, so while is processing this it has to multiply each element with our argument delta, when this is done the new elements RGB have to be store in the same place, so we use ***set***() but before this we have to assure our RGB components are in below 255 and above 0 (range established)

The method ***changeRGB() ,*** same procedure as ***changeBrightness***().

5.Conclusion

In closing in this lab, we've again seen types of relations, specifically composition relation and how to code them in VS code as their behavior between classes. Similar to lab 3 some parts were trivial, like the creation of the classes even so we apply new libraries like math, again really important keywords for inheritance (***super***() and ***extend***()) , the declaration <<*abstract*>>, a very new thing @*override* word learned their significant in last lectures of the subject.

In addition, this lab wasn't that difficult as the previous ones since there wasn't a given code, nonetheless there were functions difficult to compute like ***ChangeBrightness***() and ***ChangeRGB***() in *ColorFrame* class, they weren't that trivial as other functions .

Moreover, notice that we created many getters and print methods that are not explained in the report because they are trivial, but they helped us to make sure all the structure was correct in the Test main. We also had some trouble understanding running matrices due to the fact it has been a long time since we code matrices, but once we knew what we were working with, the process was mechanical and easier .

Nonetheless we managed to finish the lab and also their execution was successful. We found the seminar 4 more understandable and a little bit easier than the previous, so that means we're getting better at this, and also thanks to this lab more subjects were clarified.