Lab exercise 3: Implementing different classes of a design that uses inheritance
21414-Object Oriented Programming

November 14, 2016

1 Introduction

The aim of this lab session is to implement the design of Seminar 3 consisting of the Vector, Matrix, AudioBuffer, Frame, BWFrame and ColorFrame classes. For the moment we will leave for a later lab session the implementation of the File class hierarchy.

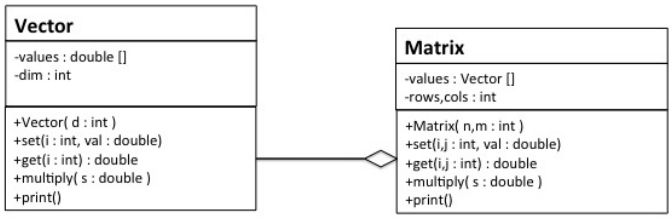
The session is mandatory and you have to deliver the source code of the java project and a document describing the implementation.

In this session we want to emphasize the use of inheritance and code reuse. For this purpose we will design first a Vector and Matrix class that will be of general usage. Secondly we will reuse the implemented classes to then implement an AudioBuffer and a Frame class hierarchy. It's important to take into account that Vector and Matrices could be reused for any other purposes, like for example a physics library, that's why we keep in the Vector and Matrix classes everything that relates to these mathematical structures and nothing more, so that they can be reused.

First of all create a project that will not include Graphics, thus remember to leave checked the create main class option.

2 Vectors and Matrices

We will then create two new classes Vector and Matrix. The steps to create a new class file are available in previous lab sessions guidelines. We show in the figure below the design based on the solution to the second seminar. Notice that Matrix is related to Vector by a composition relation: a matrix has vectors. This is translated in the design by the class Matrix having an array of Vectors. The composition relation will be explained in detail in theory class. This relation already appeared in some examples like the World has Agents relation and the Restaurant has Clients and Waiters. Composition contrasts with the inheritance relation defined as an is-a relation: a client is-a person and a waiter is-a person. Take a look at the design that you will have to implement:



The operations included in the seminar had the aim of practicing parameters and how the signature of methods can be defined and determine the implementation. In this lab session we are only going to implement the operations of vectors and matrices that follow. For the Vector class you will need to implement:

- The constructor having as parameter the dimension of the vector
- element getter and setters
- An operation that sets all the values of the vector to zero. This operation can be called by the constructor, after creating the array
- An operation that multiplies all of the vector values by a scalar
- A print operation

Once you are done you can test the Vector class in the main. Write in the main class the following code and run it to see everything works correctly:

```
Vector v = new Vector(3);
v.set(0,1);
v.set(1,2);
v.set(2,3);
v.print();
v.zero();
v.print();
```

Now let's create the Matrix class file. For the Matrix class you will need to implement very similar operations:

- The constructor with both dimensions n,m as parameters
- element getter and setters
- An operation to set all its elements to 0, The operation can call the vector operation that you created
- multiply by a scalar each of its elements
- A print operation

Once you are done you can test the Matrix class in the main. Write in the main class the following code and run it to see everything works correctly. You can see that the code creates a 2x2 instance of the matrix class and sets the identity matrix.

```
Matrix m = new Matrix(2,2);
m.set(0,0,1);
m.set(0,1,0);
m.set(1,0,0);
m.set(1,1,1);
m.print();
m.zero();
m.print();
```

2.1 Application and testing using Rotation Matrices

A 3D rotation matrix (3x3) of the z axis is a matrix of the form:

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

When multiplied by a column vector it results in the same vector rotated by the angle with which it was created:

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

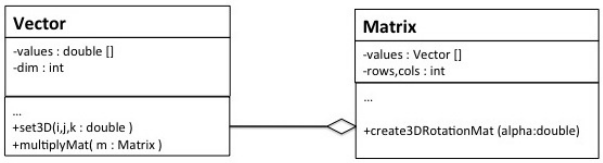
Look carefully at the code bellow which implements the previous matrix operation:

```
Vector v = new Vector(3);
v.set3D(1,0,0);

Matrix m = new Matrix(3,3);
m.create3DRotationZ( Math.PI / 2 );

v.print()
v.matrixMultiply( m );
v.print()
```

We will now add the supplementary functions to Vector and Matrix to be able to execute and test the previous code.



Remember how to call and access math operations and PI constant in java:

```
double alpha = Math.PI / 2.0;
double cosAlpha = Math.cos(alpha);
```

This example just makes it clear that Math.sin and Math.cos work using radian units. Remember $alpha_{degrees} = alpha_{radians}.\frac{180}{PI}$, thus $\frac{PI}{2}\frac{180}{PI} = 90$.
You will need to add the following operations:

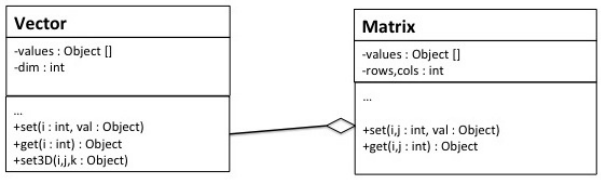
- a setter for 3d vectors as indicated by the operation set3d which receives the three components of the vector and sets the three positions of the array with them.
- a 3d rotation matrix creation operation for the z axis.
- a multiplication operation of a vector by a matrix. Remember to test that dimensions match. The following code should give an error:

```
Vector v = new Vector(2);
Matrix m = new Matrix(3,3);
m.create3DRotationZ( Math.PI / 2 );
v.matrixMultiply( m );
```

After adding the new operations, execute and test that the previous code is correct and does what it has to do, rotates the vector 90 degrees in the z axis.

2.2 A general Vector class

We have talked in class that we could implement a general Vector class using the Object class from which all classes inherit by default. We will now modify the code that we have written substituting the double array of vector by an Object array. We will also have to change the getter and setters signatures.



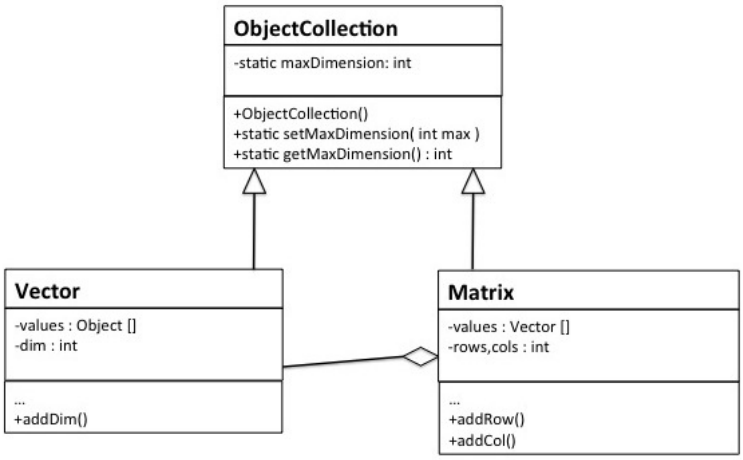
2.3 Changing dimensions: discussion

We will change the design and implementation to be able to change the size of a Vector or Matrix after its creation.

Remember that when an array is created the only way to change its capacity is to delete and create it again. This is because n array has contiguous memory positions.

Usually when we want to add elements after the creation is better to use another structure : a list for example (remember that we used a linked list in the first lab session). The problem with using a list to implement matrices is that they spend too much memory. For this reason we are going to use an alternative implementation design using a maxCapacity attribute. In the creation of a Vector or a Matrix we are going to create the array of size maxCapacity. Then if we want to add dimensions, rows or columns the space will already be created.

We will consider that the maxCapacity attribute is not related to the Vector or Matrix itself but to a more general class ObjectCollection, from which Vector and Matrix will inherit. In this sense a Vector is-a ObjectCollection and a Matrix as well.



From the previous design you will have to add one more class ObjectCollection: create the class file. Add the static maxDimension attribute to it and its static getter and setter. Now make Vector and Matrix inherit from it, remember this is done with the extends command:

```
public class Vector extends ObjectCollection {
...
}
```

Then add methods to Vector and Matrix so that we can add dimensions, rows and columns.

- adding a dimension to a vector

- adding a column to the matrix
- adding a row to the matrix

After that use execute in the main class a little testing:

```
Vector v = new Vector(3);
v.set3D(1,2,3);
v.print();
v.addDim();
v.print();
```

2.4 A different implementation

This section requires no implementation, only a design and a short discussion in the document:

- Comment in a few lines an alternative design of the classes matrix and vectors. In this alternative design Matrix contains a two dimensional array of values and the Vector Class inherits from Matrix.

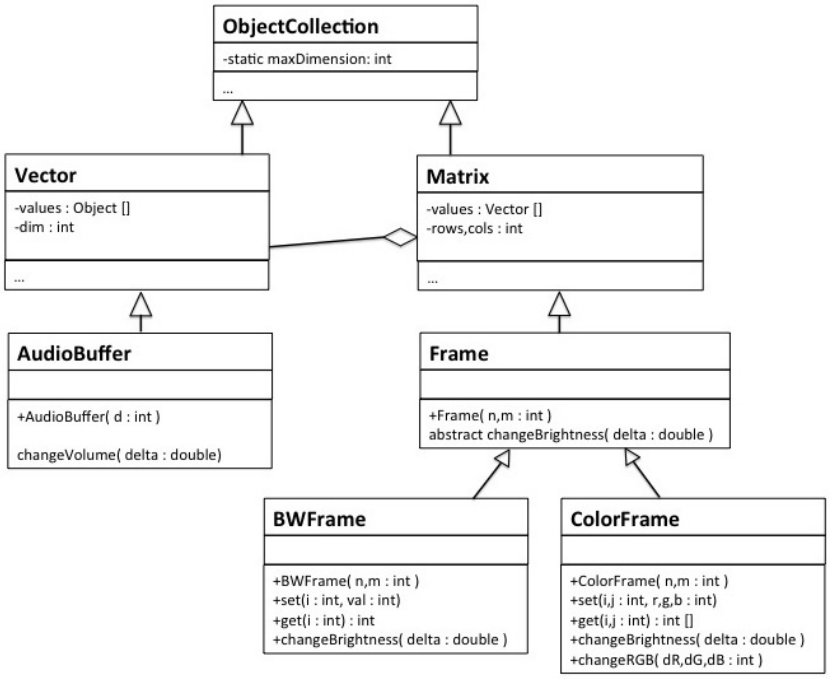
3 AudioBuffer class

Let's now create the AudioBuffer class that inherits from Vector as inspired by the solution to the seminar. Remember that the inheritance relation is indicated with the "extends":

```
public class AudioBuffer extends Vector {
...
}
```

We will only have to define and implement the following functions:

- A constructor
- 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.



4 Frame, BWFrame and ColorFrame classes

We will now create the classes Frame, BWFrame and ColorFrame following the design of last figure. The Frame class gathers the common operations of Black and White and Color frames. In section 2.2 we modified the design and the code to make the Vector class a general class that can contain Objects of any class, because all classes defined inherit by default from Object.

Frames add operations to deal with the intensity of the pixels and also with color R,G,B components:

- Define and implement all the constructors
- We want to define specific setters and getters for BWFrame and ColorFrame. Look at the design figure to see their signature. Remember that if a method has the same name and different signature we call that overloading.
- 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.

The R,G,B components of a color pixel are stored in a single double value that is big enough because it has 8 bytes: in fact with a float it will be enough. Remember each component is only one byte from 0 to 255.

You will need to use two auxiliary functions to be able to translate a double value into its three R,G,B components (and vice-versa). We provide the code for this translation:

```
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;
}
```

The first function valToRGB is given a double value and returns the 3 position array of its corresponding R,G,B components. RGBToVal is given the 3 R,G,B components and returns its corresponding double.

You will need to use these two functions for implementing the getter and setter values of the ColorFrame class. It will be your decision where to implement these two auxiliary functions and where to put them in the design hierarchy.

Here is a guide of how to use them in the setter of ColorFrame that we have overloaded:

```
public void set(int i, int j, int r, int g, int b) {
          double ret = RGBToVal(r,g,b);
          set(i,j,ret);
}
```

Observe that we are calling the setter of matrix as we are in a class that inherits from it.