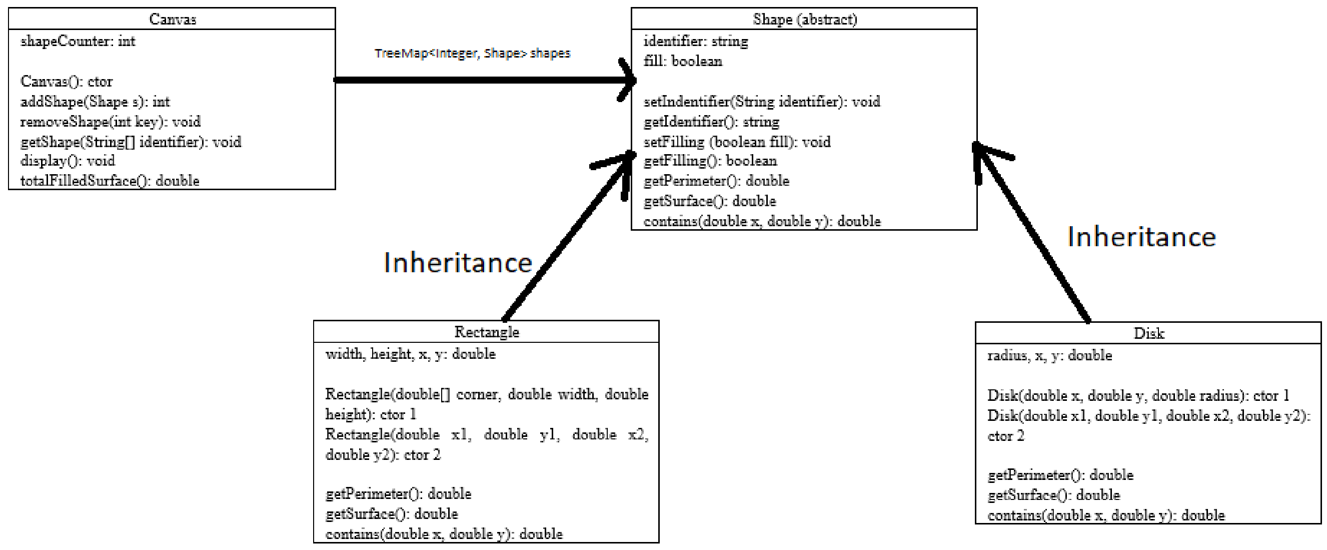
IT Tools BE Report – Shapes + Matrices

Student: TRAN Gia Quoc Bao, group G4-c Date: 20/11/2019

Shapes

UML diagram with classes and corresponding constructors & methods contained inside (their detailed descriptions are put in the Java code as comments):

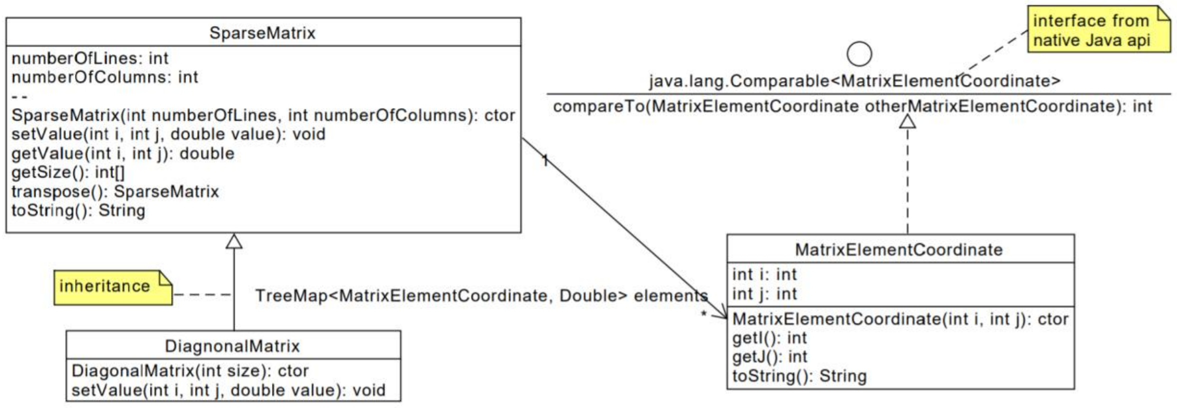

Canvas
shapeCounter: int
Canvas(): ctor
addShape(Shape s): int
removeShape(int key): void
getShape(String[] identifier): void
display(): void
totalFilledSurface(): double
TreeMap<Integer, Shape> shapes
Shape (abstract)
identifier: string
fill: boolean
setIndentifier(String identifier): void
getIdentifier(): string
setFilling (boolean fill): void
getFilling(): boolean
getPerimeter(): double
getSurface(): double
contains(double x, double y): double
Inheritance
Inheritance
Rectangle
width, height, x, y: double
Rectangle(double[] corner, double width, double height): ctor 1
Rectangle(double x1, double y1, double x2, double y2): ctor 2
getPerimeter(): double
getSurface(): double
contains(double x, double y): double
Disk
radius, x, y: double
Disk(double x, double y, double radius): ctor 1
Disk(double x1, double y1, double x2, double y2): ctor 2
getPerimeter(): double
getSurface(): double
contains(double x, double y): double

Some necessary explanations:

- I declared variable types to be private for the sake of code security and surround them with methods. To get access to these variables or other characteristics (perimeters, surfaces) we use the "get" methods.
- To keep track of the number of shapes, I created the shapeCounter variable which will +1 whenever we add a new shape. So for the map, we start from 0 and then 1 2 3 for a total of 4 shapes.
- For disks, there are 2 constructors: 3 inputs for 1 point + radius and 4 inputs for 2 points. For rectangles, there are also 2 constructors: 3 inputs for 1 point (double[x,y]) which is the bottom left corner + 2 sides and 4 inputs for 2 points. We can also delete shapes.
- Each shape can be indentified (in this case by the 1, 2, 3, 4 that I set to them or can be by any string in general if you want to call them Rect1, Disk2,…). They can also the filled (as "true").
- We can check if a point lies inside a shape or not using contains("point's coordinates"), it is overridden for each type of shapes, unlike setFill() and setIdentifier().
- The getShape() is to print just the shapes you specify using identifiers. Now I let it print the shape but it can easily be modified to print the surfaces, perimeter,… that you want.
- display(): to print out the whole map with keys and respective shape's surface area. It uses a FOR loop which is to loop through all the keys and add the new values to the string "output".
- totalFilledSurface(): this will return the total surface area of all the shapes that are declared as "filled". It uses a FOR loop and the condition to check if the shape is filled or not. If true it will add the surface.

Matrices

UML diagram with classes and corresponding constructors & methods contained inside (their detailed descriptions are put in the Java code as comments):


SparseMatrix
numberOfLines: int
numberOfColumns: int
--
SparseMatrix(int numberOfLines, int numberOfColumns): ctor
setValue(int i, int j, double value): void
getValue(int i, int j): double
getSize(): int[]
transpose(): SparseMatrix
toString(): String
interface from native Java api
java.lang.Comparable<MatrixElementCoordinate>
compareTo(MatrixElementCoordinate otherMatrixElementCoordinate): int
1
inheritance
TreeMap<MatrixElementCoordinate, Double> elements
*
MatrixElementCoordinate
int i: int
int j: int
MatrixElementCoordinate(int i, int j): ctor
getI(): int
getJ(): int
toString(): String
DiagonalMatrix
DiagonalMatrix(int size): ctor
setValue(int i, int j, double value): void

Some necessary explanations:

- I declared variable types to be private for the sake of code security and surround them with methods. To get access to these variables (attributes) we use the methods getI(), getJ(), getSize().
- I put in the Exceptions by using a new class containing them and implement them in other classes.
- For a sparse matrix, it would need to be one with fewer non zero values than zero values. That is why I put in the variable "elementCounter": it will be created as 0 when we construct a new matrix (just like the number of rows and columns) and whenever we succeed in entering a non zero value it is increased by 1 (just to count). We cannot add more element if this exceeds half the maximum possible number of elements (condition in line 39, class SparseMatrix).
- For the setValue() it has certain conditions to avoid entering outside of the size and entering zeros, and whenever we succeed in inputing an element it will let us know.
- I also wrote the displayMatrix() which will print all elements in a sorted order, which is useful to check later the results of transposition.
- Here in order for the Java native Comparable (which will sort the objects – in this case the coordinates, in an order we define) to work, we need to override the compareTo() method. I defined my order as follow: we display all elements in the same row, and then we move on and do this for the next row. For this I made the compareTo() return 1 if bigger row is entered, and -1 if smaller. For the case of 0 which means we enter an element into an existing row, it will move on the compare the columns.
- I developed the transpose() which reverses the coordinates and puts everything in a new matrix.
- The DiagonalMatrix class inherits from SparseMatrix with an additional condition in the constructor as well as other methods.
- Here, in order to transpose a diagonal matrix, because it inherits from the class of sparse matrices, we can just create the transposed matrix with the category "SparseMatrix" and the transpose() works fine. But if you want to create the transposed matrix with the category "DiagonalMatrix", you can just use the code I overrode in the class DiagonalMatrix, which I put in the comment. It would return exactly the same matrix anyway.

Thank you for reading