**ליאור סברי - 313137937**

**בן ביטון – 203330378**

**Reason for warning in DrawPanel line 307 –**

// Sadly SuppressWarnings doesnt work because of a bug and it isn't possible

// to ignore the warning. The warning happens because we read an object

// from a file which obviously can't be known beforehand and then try

// to cast it to a ArrayList<MyShape>, even if we check with instanceof

// the warning will presist, so the only option we have is to leave it as is.

//@SuppressWarnings("unchecked")

shapes = (ArrayList<MyShape>)ois.readObject();

**Explanation for the program –**

The program has been a bit improved, here is what has been added:

* It has the option to draw with a pen in the combo box by using small filled ovals.
* If you would like to draw more than the maximum limit you can press the Unlimited Drawing checkbox pressing it again will erase all drawings above the maximum limit.
* There is a Redo button to go with the Undo button.
* In the color’s combo box, there is a last option “Make color” which enables the three sliders on the East side with them you can control the color with RGB that ranges between 0-255.
* Above the 3 sliders there is another small JPanel which changes its color according to the current used color.
* The top left section there is a menu bar when pressing “File” you’re given an option to save and load your current drawing with serializable files only in the format that it is given as an example.

**Class MyShape – MyShape.java**

// MyShape.java

import java.awt.Graphics;

import java.awt.Color;

import java.io.\*;

/\*\*

\* This abstract class implements Serializable so the it's variables can be

\* serialized or deserialized, and implements the state and behaviour of all shapes.

\* @author Lior Sabri, Ben Biton

\*/

public abstract class MyShape implements Serializable

{

private int x1; // x coordinate of first point

private int y1; // y coordinate of first point

private int x2; // x coordinate of second point

private int y2; // y coordinate of second point

private Color myColor; // Color of the shape

/\*\*

\* MyShape default constructor, initializes a new MyShape with default values

\* by calling the argument cconstructor.

\*/

public MyShape()

{

this(0, 0, 0, 0, Color.BLACK);

}

/\*\*

\* MyShape argument constructor, initializes a new MyShape with recieved values

\* values sent to this function will be checked if they are legal if not they

\* will recieve the value 0.

\* @param x1 first x coordinate

\* @param y1 first y coordinate

\* @param x2 second x coordinate

\* @param y2 second y coordinate

\* @param myColor chosen color to draw the shape

\*/

public MyShape(int x1, int y1, int x2, int y2, Color myColor)

{

this.x1 = (x1 >= 0 ? x1 : 0);

this.y1 = (y1 >= 0 ? y1 : 0);

this.x2 = (x2 >= 0 ? x2 : 0);

this.y2 = (y2 >= 0 ? y2 : 0);

this.myColor = myColor;

}

/\*\*

\* Set the x-coordinate of the first point and check if it's a legal value

\* incase it isn't set the value with 0.

\* @param x1 first x coordinate

\*/

public void setX1(int x1)

{

this.x1 = (x1 >= 0 ? x1 : 0);

}

/\*\*

\* Set the y-coordinate of the first point and check if it's a legal value

\* incase it isn't set the value with 0.

\* @param y1 first y coordinate

\*/

public void setY1(int y1)

{

this.y1 = (y1 >= 0 ? y1 : 0);

}

/\*\*

\* Set the x-coordinate of the second point and check if it's a legal value

\* incase it isn't set the value with 0.

\* @param x2 first x coordinate

\*/

public void setX2(int x2)

{

this.x2 = (x2 >= 0 ? x2 : 0);

}

/\*\*

\* Set the y-coordinate of the second point and check if it's a legal value

\* incase it isn't set the value with 0.

\* @param y2 first y coordinate

\*/

public void setY2(int y2)

{

this.y2 = (y2 >= 0 ? y2 : 0);

}

/\*\*

\* Set the color

\* @param myColor new color

\*/

public void setColor(Color myColor)

{

this.myColor = myColor;

}

/\*\*

\* Get the x-coordinate of the first point.

\* @return the value of the first x-coordinate.

\*/

public int getX1()

{

return x1;

}

/\*\*

\* Get the y-coordinate of the first point.

\* @return the value of the first y-coordinate.

\*/

public int getY1()

{

return y1;

}

/\*\*

\* Get the x-coordinate of the second point.

\* @return the value of the second x-coordinate.

\*/

public int getX2()

{

return x2;

}

/\*\*

\* Get the y-coordinate of the second point.

\* @return the value of the second y-coordinate.

\*/

public int getY2()

{

return y2;

}

/\*\*

\* Get the color.

\* @return the color.

\*/

public Color getColor()

{

return myColor;

}

/\*\*

\* This abstract method is made for polymorphic reason each time a draw method

\* will be called it will draw the matching shape.

\* @param g the graphic panel

\*/

public abstract void draw(Graphics g);

} // end class MyShape

**Class MyLine – MyLine.java**

// MyLine.java

// Declaration of class MyLine.

import java.awt.Color;

import java.awt.Graphics;

/\*\*

\* This class implements the state and behaviour of a line.

\* It extends from MyShape

\* @author Lior Sabri, Ben Biton

\*/

public class MyLine extends MyShape

{

/\*\*

\* MyLine default constructor, calls default super constructor and

\* initializes variables with default values

\*/

public MyLine()

{

}

/\*\*

\* MyLine argument constructor, calls super argument constructor and

\* initializes a new MyLine with recieved values.

\* will recieve the value 0.

\* @param x1 first x coordinate

\* @param y1 first y coordinate

\* @param x2 second x coordinate

\* @param y2 second y coordinate

\* @param myColor chosen color to draw the line

\*/

public MyLine(int x1, int y1, int x2, int y2, Color myColor)

{

super(x1, y1, x2, y2, myColor);

}

/\*\*

\* Draws a line in the specified color.

\* @param g the graphic panel

\*/

public void draw(Graphics g)

{

g.setColor(getColor());

g.drawLine(getX1(), getY1(), getX2(), getY2());

}

} // end class MyLine

**Class MyBoundedShape – MyBoundedShape.java**

import java.awt.Color;

/\*\*

\* This abstract class implements the state and behaviour of all bounded shapes.

\* it extends from MyShape

\* @author Lior Sabri, Ben Biton

\*/

public abstract class MyBoundedShape extends MyShape

{

private boolean isFilled; // If the shape is filled or not

/\*\*

\* MyBoundedShape default constructor, calls default super constructor and

\* initializes variables with default values

\*/

public MyBoundedShape()

{

}

/\*\*

\* MyBoundedShape argument constructor, calls super argument constructor and

\* initializes a new MyRectangle with recieved values.

\* will recieve the value 0.

\* @param x1 first x coordinate

\* @param y1 first y coordinate

\* @param x2 second x coordinate

\* @param y2 second y coordinate

\* @param myColor chosen color to draw the shape

\* @param isFilled If the shape is filled or not

\*/

public MyBoundedShape(int x1, int y1, int x2, int y2, Color myColor,

boolean isFilled)

{

super(x1, y1, x2, y2, myColor);

this.isFilled = isFilled;

}

/\*\*

\* Sets whether this shape is filled or not.

\* @param isFilled If the shape is filled or not

\*/

public void setFilled(boolean isFilled)

{

this.isFilled = isFilled;

}

/\*\*

\* Determines whether this shape is filled.

\* @return If the shape is filled or not

\*/

public boolean getIsFilled()

{

return isFilled;

}

/\*\*

\* Get upper left x coordinate

\* @return math operation for upper left x coordinate

\*/

public int getUpperLeftX()

{

return Math.min(getX1(), getX2());

}

/\*\*

\* Get upper left y coordinate

\* @return math operation for upper left y coordinate

\*/

public int getUpperLeftY()

{

return Math.min(getY1(), getY2());

}

/\*\*

\* Get shape width

\* @return math operation for width

\*/

public int getWidth()

{

return Math.abs(getX2() - getX1());

}

/\*\*

\* Get shape height

\* @return math operation for height

\*/

public int getHeight()

{

return Math.abs(getY2() - getY1());

}

}

**Class MyOval – MyOval.java**

// MyOval.java

// Declaration of class MyOval.

import java.awt.Color;

import java.awt.Graphics;

/\*\*

\* This class implements the state and behaviour of an oval.

\* It extends from MyBoundedShape

\* @author Lior Sabri, Ben Biton

\*/

public class MyOval extends MyBoundedShape

{

/\*\*

\* MyOval default constructor, calls default super constructor and

\* initializes variables with default values

\*/

public MyOval()

{

}

/\*\*

\* MyOval argument constructor, calls super argument constructor and

\* initializes a new MyOval with recieved values.

\* will recieve the value 0.

\* @param x1 first x coordinate

\* @param y1 first y coordinate

\* @param x2 second x coordinate

\* @param y2 second y coordinate

\* @param myColor chosen color to draw the shape

\* @param isFilled If the shape is filled or not

\*/

public MyOval(int x1, int y1, int x2, int y2, Color myColor, boolean isFilled)

{

super(x1, y1, x2, y2, myColor, isFilled);

}

/\*\*

\* Draws an oval in the specified color.

\* @param g the graphic panel

\*/

public void draw(Graphics g)

{

g.setColor(getColor());

if (getIsFilled())

{

g.fillOval(getUpperLeftX(), getUpperLeftY(), getWidth(), getHeight());

}

else

{

g.drawOval(getUpperLeftX(), getUpperLeftY(), getWidth(), getHeight());

}

}

} // end class MyOval

**Class MyRectangle – MyRectangle.java**

// MyRectangle.java

// Declaration of class MyRectangle.

import java.awt.Color;

import java.awt.Graphics;

/\*\*

\* This class implements the state and behaviour of a rectangle.

\* It extends from MyBoundedShape

\* @author Lior Sabri, Ben Biton

\*/

public class MyRectangle extends MyBoundedShape

{

/\*\*

\* MyRectangle default constructor, calls default super constructor and

\* initializes variables with default values

\*/

public MyRectangle()

{

}

/\*\*

\* MyRectangle argument constructor, calls super argument constructor and

\* initializes a new MyRectangle with recieved values.

\* will recieve the value 0.

\* @param x1 first x coordinate

\* @param y1 first y coordinate

\* @param x2 second x coordinate

\* @param y2 second y coordinate

\* @param myColor chosen color to draw the shape

\* @param isFilled If the shape is filled or not

\*/

public MyRectangle(int x1, int y1, int x2, int y2, Color myColor, boolean isFilled)

{

super(x1, y1, x2, y2, myColor, isFilled);

}

/\*\*

\* Draws a rectangle in the specified color.

\* @param g the graphic panel

\*/

public void draw(Graphics g)

{

g.setColor(getColor());

if (getIsFilled())

{

g.fillRect(getUpperLeftX(), getUpperLeftY(), getWidth(), getHeight());

}

else

{

g.drawRect(getUpperLeftX(), getUpperLeftY(), getWidth(), getHeight());

}

}

} // end class MyRectangle

**Class DrawPanel – DrawPanel.java**

// DrawPanel.java

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import java.util.\*;

import java.io.\*;

/\*\*

\* DrawPanel class extends from JPanel.

\* <br> It uses classes MyLine, MyOval and MyRectangle to draw shapes.

\* <br> The class will recieve data from mouse, buttons, combobox and checkbox and

\* will draw shapes and save them with Polymorphism into an ArrayList of MyShape.

\* <br> See JPanel class: {@link javax.swing.JPanel}.

\* <br> See ArrayList class: {@link java.util.ArrayList}&lt;{@link MyShape}&gt;}.

\* <br> See all shapes class: {@link MyLine}, {@link MyOval}, {@link MyRectangle}.

\* @author Lior Sabri, Ben Biton

\*/

public class DrawPanel extends JPanel

{

private ArrayList<MyShape> shapes;// ArrayList containing all the shapes through polymorphism.

private static final int MAX\_NUMBER\_OF\_SHAPES = 100; // Size of the array.

private int shapesCounter; // Counts the number of shapes on the panel also used as an index.

private enum ShapeType {LINE, OVAL, RECTANGLE, PEN} // enum containing all shapes.

private MyShape currentDrawingShape; // The shape we currently draw.

private Color currentDrawingColor; // The current color of the shape.

private boolean isDrawingFilled; // Whether the current shape is filled or not.

private boolean isDrawingUnlimited; // Decides if we can draw more than max number of shapes.

private ShapeType currentDrawingShapeType; // What shape we are currently drawing.

private JFrame parentWindow; // JFrame sent from DrawFrame class.

private JLabel statusLabel; // Label to show mouse coordinates and shape counter.

/\*\*

\* DrawPanel constructor, creates a panel where the shapes are drawn and

\* initializes the class variables while also creating a mouse handler.

\* <br> See MouseHandler class: {@link javax.swing.plaf.basic.BasicTreeUI.MouseHandler}.

\* @param parentWindow Holds the built JFrame.

\* @param statusLabel Holds the JLabel which we will add mouse coordinates and shapesCounter.

\*/

public DrawPanel(JFrame parentWindow, JLabel statusLabel)

{

shapes = new ArrayList<MyShape>(MAX\_NUMBER\_OF\_SHAPES);

currentDrawingShapeType = ShapeType.LINE;

this.statusLabel = statusLabel;

this.parentWindow = parentWindow;

setBackground(Color.WHITE);

MouseHandler mouseHandler = new MouseHandler();

addMouseListener(mouseHandler);

addMouseMotionListener(mouseHandler);

}

/\*\*

\* Creates a shape according to the currentDrawingShapeType with switch case.

\* @param x1 Holds the first x-axis.

\* @param y1 Holds the first y-axis.

\* @param x2 Holds the second x-axis.

\* @param y2 Holds the second y-axis.

\* @return A new Shape will be sent back or null if some thing went wrong

\* although this should never happen.

\*/

private MyShape createShape(int x1, int y1, int x2, int y2)

{

final int PEN\_SIZE = 5;

switch(currentDrawingShapeType)

{

case LINE:

return new MyLine(x1, y1, x2, y2, currentDrawingColor);

case OVAL:

return new MyOval(x1, y1, x2, y2, currentDrawingColor, isDrawingFilled);

case RECTANGLE:

return new MyRectangle(x1, y1, x2, y2, currentDrawingColor, isDrawingFilled);

case PEN:

return new MyOval(x1, y1, x2 + PEN\_SIZE, y2 + PEN\_SIZE, currentDrawingColor, true);

default: // this should never happen

return null;

}

}

/\*\*

\* This method will lower the shape counter by 1 and repaint if there is at

\* least one shape drawn.

\* <br> See the repaint method: {@link java.awt.Component#repaint}.

\*/

public void undoShape()

{

if (shapesCounter > 0)

{

shapesCounter--;

repaint();

}

}

/\*\*

\* This method will check if there is a shape availabe to redo if not an

\* exception will be thrown the if is a fitting way to show we are looking

\* for a MyShapes variable which is null and getting IndexOutOfBoundsException.

\* <br> If it is possible to redo a shape the shapesCounter will enlarge by 1

\* and repaint.

\* <br> See the repaint method: {@link java.awt.Component#repaint}.

\*/

public void redoShape()

{

try

{

if (shapes.get(shapesCounter) != null)

{

shapesCounter++;

repaint();

}

}

catch (IndexOutOfBoundsException e)

{

}

}

/\*\*

\* If there are shapes drawn the method will prompt a dialog window to

\* re-assure the operation if 'Yes' will be the input the ArrayList will clear

\* the shapesCounter will be set to 0 and then it will call the repaint method.

\*/

public void clearDrawing()

{

if (shapesCounter == 0)

{

return;

}

int input = JOptionPane.showConfirmDialog(parentWindow,

"Are you sure you want to erase all?", "Clear Warning",

JOptionPane.YES\_NO\_OPTION);

if (input != JOptionPane.YES\_OPTION)

{

return;

}

shapes.clear();

shapesCounter = 0;

repaint();

}

/\*\*

\* The method will set the current drawing color.

\* @param currentDrawingColor New color to set to the current drawing.

\*/

public void setDrawingColor(Color currentDrawingColor)

{

this.currentDrawingColor = currentDrawingColor;

}

/\*\*

\* The method will set the current drawing shape type.

\* <br> Using values() we get and array of the following enum and then we save

\* the ShapeType in the matching index of the array.

\* @param shapeIndex Index which will be used to find the ShapeType.

\*/

public void setDrawingShape(int shapeIndex)

{

currentDrawingShapeType = ShapeType.values()[shapeIndex];

}

/\*\*

\* The method will set if the current drawing is filled or not.

\* @param isDrawingFilled whether the shape is filled or not

\*/

public void setDrawingFilled(boolean isDrawingFilled)

{

this.isDrawingFilled = isDrawingFilled;

}

/\*\*

\* The method will set if the the drawing is limited if cancelled it will

\* check if there more shapes drawn than the MAX\_NUMBER\_OF\_SHAPES if so

\* it will set back the shapesCounter to that limit and remove every shape

\* above that index from the ArrayList.

\* <br> See remove method: {@link java.util.ArrayList}&lt;{@link MyShape}&gt;#remove}

\* @param isDrawingUnlimited whether the ArrayList size is limited or not

\*/

public void setUnlimitDrawings(boolean isDrawingUnlimited)

{

this.isDrawingUnlimited = isDrawingUnlimited;

if (!isDrawingUnlimited && shapesCounter > MAX\_NUMBER\_OF\_SHAPES)

{

for (int i = shapesCounter - 1; i >= MAX\_NUMBER\_OF\_SHAPES; i--)

{

shapes.remove(i);

}

shapesCounter = MAX\_NUMBER\_OF\_SHAPES;

repaint();

}

}

/\*\*

\* The method saves the current drawing into a .ser file if the users input

\* for a file name is valid.

\* <br> See FileOutputStream class: {@link java.io.FileOutputStream}.

\* <br> See ObjectOutputStream class: {@link java.io.ObjectOutputStream}.

\* <br> See writeObject method: {@link java.io.ObjectOutputStream#writeObject}.

\*/

public void saveDrawing()

{

String userInput = JOptionPane.showInputDialog(parentWindow,

"Please type in the following format to save your file:", "FileName.ser");

if (userInput == null)

{

return;

}

if (!userInput.matches("\\w+\\.ser"))

{

return;

}

FileOutputStream fos = null;

ObjectOutputStream oos = null;

try

{

// Create a new file.

fos = new FileOutputStream(userInput);

oos = new ObjectOutputStream(fos);

// Write object to a file

oos.writeObject(shapes);

}

catch (FileNotFoundException e)

{

e.printStackTrace();

}

catch (IOException e)

{

e.printStackTrace();

}

finally // Close outputstreams if they exist

{

try

{

if (oos != null)

{

oos.close();

}

if (fos != null)

{

fos.close();

}

}

catch (IOException e)

{

e.printStackTrace();

}

}

}

/\*\*

\* The method loads the current drawing of a .ser file if the users input for

\* a file name is valid. If loaded successfuly the shapesCounter will be set

\* to The ArrayList size and then repaint.

\* <br> See FileOutputStream class: {@link java.io.FileInputStream}.

\* <br> See ObjectOutputStream class: {@link java.io.ObjectInputStream}.

\* <br> See writeObject method: {@link java.io.ObjectInputStream#readObject}.

\* <br> See size method: {@link java.util.ArrayList}&lt;{@link MyShape}&gt;#size}.

\*/

public void loadDrawing()

{

if (shapesCounter != 0)

{

int userInput = JOptionPane.showConfirmDialog(parentWindow,

"You still have things drawn loading a file will erase them,"

+ " would you like to go on?", "Load Warning",

JOptionPane.YES\_NO\_OPTION);

if (userInput != JOptionPane.YES\_OPTION)

{

return;

}

}

String userInput = JOptionPane.showInputDialog(parentWindow,

"Please type in the following format to load your file:", "FileName.ser");

if (userInput == null || !userInput.matches("\\w+\\.ser"))

{

return;

}

FileInputStream fis = null;

ObjectInputStream ois = null;

try

{

// Open file.

fis = new FileInputStream(userInput);

ois = new ObjectInputStream(fis);

// Read object from a file.

// Sadly SuppressWarnings doesnt work because of a bug and it isn't possible

// to ignore the warning. The warning happens because we read an object

// from a file which obviously can't be known beforehand and then try

// to cast it to a ArrayList<MyShape>, even if we check with instanceof

// the warning will presist, so the only option we have is to leave it as is.

//@SuppressWarnings("unchecked")

shapes = (ArrayList<MyShape>)ois.readObject();

}

catch (FileNotFoundException e)

{

e.printStackTrace();

}

catch (IOException e)

{

e.printStackTrace();

}

catch (ClassNotFoundException e)

{

e.printStackTrace();

}

finally // Close inputstreams if they exist

{

try

{

if (ois != null)

{

ois.close();

}

if (fis != null)

{

fis.close();

}

}

catch (IOException e)

{

e.printStackTrace();

}

}

shapesCounter = shapes.size();

repaint();

}

/\*\*

\* This Class extends from MouseAdapter and implements MouseMotionListener.

\* <br> The class will handle the following mouse events: mouseMoved, mousePressed,

\* mouseReleased, mouseDragged and mouseExited. Each of the events will handle

\* drawing or setting the label.

\* <br> See MouseAdapter class: {@link java.awt.event.MouseAdapter}.

\* <br> See MouseMotionListener interface: {@link java.awt.event}.

\*/

private class MouseHandler extends MouseAdapter implements MouseMotionListener

{

/\*\*

\* When the mouse will move the mouse coordinates shapes drawn label will

\* update.

\* <br> See mouseMoved method: {@link java.awt.event.MouseMotionListener#mouseMoved}.

\* @param event Holds the current event.

\*/

@Override

public void mouseMoved(MouseEvent event)

{

statusLabel.setText(String.format(

"Mouse Coordinates: (%d, %d), Shapes Drawn: %d",

event.getX(), event.getY(), shapesCounter));

}

/\*\*

\* When the mouse will be pressed only with the left button a new drawing

\* will start according to previous settings.

\* <br> If max drawing limit has been reached and the unlimit check box

\* isn't selected than a message will prompt.

\* <br> See mousePressed method: {@link java.awt.event.MouseAdapter#mousePressed}.

\* <br> See method used to create new shape: {@link #createShape}.

\* @param event Holds the current event.

\*/

@Override

public void mousePressed(MouseEvent event)

{

if (event.getButton() != MouseEvent.BUTTON1)

{

return;

}

int x = event.getX();

int y = event.getY();

// If the Pen will be chosen in the combobox.

if (currentDrawingShapeType == ShapeType.PEN &&

(shapesCounter < MAX\_NUMBER\_OF\_SHAPES || isDrawingUnlimited))

{

shapes.add(shapesCounter, createShape(x, y, x, y));

shapesCounter++;

repaint();

return;

}

// If any other shape than Pen will be chosen in the combobox.

if (shapesCounter < MAX\_NUMBER\_OF\_SHAPES || isDrawingUnlimited)

{

currentDrawingShape = createShape(x, y, x, y);

}

else // If the Shapes drawn have reached the limit

{

JOptionPane.showMessageDialog(parentWindow, String.format(

"Can't exceed the current number of shapes: %d",

MAX\_NUMBER\_OF\_SHAPES), "Capacity Overload",

JOptionPane.WARNING\_MESSAGE);

}

}

/\*\*

\* When the mouse will be released only with the left button the new drawing

\* will set.

\* <br> If the drawn shape will be invisible it will be released to the

\* garbage collecter.

\* <br> See mouseReleased method: {@link java.awt.event.MouseAdapter#mouseReleased}.

\* @param event Holds the current event.

\*/

@Override

public void mouseReleased(MouseEvent event)

{

if (event.getButton() != MouseEvent.BUTTON1)

{

return;

}

if (currentDrawingShapeType == ShapeType.PEN)

{

return;

}

currentDrawingShape.setX2(event.getX());

currentDrawingShape.setY2(event.getY());

int x1 = currentDrawingShape.getX1();

int y1 = currentDrawingShape.getY1();

int x2 = currentDrawingShape.getX2();

int y2 = currentDrawingShape.getY2();

boolean isDot = ((x1 == x2) && (y1 == y2));

boolean isInvisibleLine = ((x1 == x2) || (y1 == y2)) && isDrawingFilled && (currentDrawingShapeType != ShapeType.LINE);

// If the shape isn't invisible add it to the arraylist

if (!isDot && !isInvisibleLine)

{

shapes.add(shapesCounter, currentDrawingShape);

shapesCounter++;

}

currentDrawingShape = null;

}

/\*\*

\* When the mouse will be dragged and it's a Pen new shape will be drawn,

\* if it isn't a Pen the x2,y2 will always update and repaint to give a

\* filling as the shape follows our mouse.

\* <br> See mouseDragged method: {@link java.awt.event.MouseMotionListener#mouseDragged}.

\* @param event Holds the current event.

\*/

@Override

public void mouseDragged(MouseEvent event)

{

mouseMoved(event);

int x = event.getX();

int y = event.getY();

if (currentDrawingShapeType == ShapeType.PEN &&

(shapesCounter < MAX\_NUMBER\_OF\_SHAPES || isDrawingUnlimited))

{

shapes.add(shapesCounter, createShape(x, y, x, y));

shapesCounter++;

repaint();

return;

}

if (currentDrawingShape != null)

{

currentDrawingShape.setX2(x);

currentDrawingShape.setY2(y);

repaint();

}

}

/\*\*

\* When the mouse will exit the DrawPanel it will set the status label.

\* <br> See mouseExited method: {@link java.awt.event.MouseAdapter#mouseExited}.

\* @param event Holds the current event.

\*/

@Override

public void mouseExited(MouseEvent event)

{

statusLabel.setText(String.format(

"Mouse outside panel, Shapes Drawn: %d", shapesCounter));

}

}

/\*\*

\* The method will draw every shape in the array list till the shapeCounter

\* and will also draw the current shape that's initiated by a mouse pressed

\* event if the currentDrawingShape isn't null.

\* <br> See Graphics class: {@link java.awt.Graphics}.

\* @param g The graphic panel.

\*/

public void paintComponent(Graphics g)

{

super.paintComponent(g);

// Draw the arraylist of shapes.

for (int i = 0; i < shapesCounter; i++)

{

shapes.get(i).draw(g);

}

// With this we draw the shape as it's occuring.

if (currentDrawingShape != null)

{

currentDrawingShape.draw(g);

}

}

} // end class DrawPanel

**Class DrawFrame – DrawFrame.java**

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import javax.swing.event.\*;

/\*\*

\* The class extends from JFrame {@link javax.swing.JFrame} and implements

\* ItemListener {@link java.awt.event}, ActionListener {@link java.awt.event},

\* ChangeListener {@link javax.swing.event}.

\* <br> This Class implements the DrawFrame main window (GUI arrangement).

\* <br> It creates 4 different panels in the fame.

\* <br> The panels created are DrawingPanel {@link DrawPanel} to hold all drawings

\* made, selectionPanel which holds all JButtons, JComboBox, JCheckBox and

\* lastly a sliderPanel which holds another panel that shows the current used

\* color and all JSlider with matching JLabels for them.

\* @author Lior Sabri, Ben Biton

\*/

public class DrawFrame extends JFrame

implements ItemListener, ActionListener, ChangeListener

{

private JButton[] buttons; // Holds all three buttons need: Undo, Redo, Clear.

private JComboBox<String> colorsBox; // Combobox which shows available colors.

private JComboBox<String> shapesBox; // Combobox which show available shapes.

private JCheckBox filledBox; // When checked or unchecked the shape will be filled accordingly.

private JCheckBox unlimitDrawingsBox; // This gives the option to unlimit the amount of shapes to be drawn.

private JSlider[] rgbSlider; // Shows red, green, blue sliders to control the color.

private JPanel showCurrentColorPanel; // Displays the current color being used.

private DrawPanel graphicsPanel; // Holds all drawings made by the user.

private JLabel statusLabel; // A label to show mouse coordinates and how many shapes were drawn.

private JLabel[] rgbLabel; // Red, green, blue labels for each rgb slider.

// Array of Strings for the rgbLabels.

private static final String[] rgbString = {"Red", "Green", "Blue"};

// Color arrray to hold colors for the colorBox.

private static final Color[] allColors = {Color.BLACK, Color.BLUE, Color.CYAN,

Color.DARK\_GRAY, Color.GRAY, Color.GREEN, Color.LIGHT\_GRAY, Color.MAGENTA,

Color.ORANGE, Color.PINK, Color.RED, Color.WHITE, Color.YELLOW};

// Array of Colors for the rgbLabel.

private static final Color[] rgbColor = {allColors[10], allColors[5],

allColors[1]};

/\*\*

\* DrawFrame Constructor - creates all panels and file bar and adds them all

\* to the JFrame.

\* <br> see all methods used: {@link #createFileMenu}, {@link #createSelectionPanel},

\* {@link #createSliderPanel}, {@link #createGraphicsPanel}.

\*/

public DrawFrame()

{

// Create JLabel for mouse coordinates.

statusLabel = new JLabel("Mouse outside panel, Shapes Drawn: 0");

add(statusLabel, BorderLayout.SOUTH);

// Create set and add all GUI components

createFileMenu();

createSelectionPanel();

createSliderPanel();

createGraphicsPanel();

}

/\*\*

\* The method initializes all JButtons, JComboBox and JCheckBox needed,

\* and add thems to the selectionPanel in a GridLayout which will finally

\* add the JPanel to the JFrame.

\* <br> See the class GridLayout: {@link java.awt.GridLayout}.

\*/

private void createSelectionPanel()

{

// Creating names for the buttons and combobox.

final String[] buttonsNames = {"Undo", "Redo", "Clear"};

final String[] allColorsString = {"Black", "Blue", "Cyan",

"Dark Gray", "Gray", "Green", "Light Gray", "Magenta", "Orange", "Pink",

"Red", "White", "Yellow", "Make Color"};

final String[] shapesToDraw = {"Line", "Oval", "Rectangle", "Pen"};

JPanel selectionPanel = new JPanel();

// Setting a GridLayout to stretch all components for a better look.

selectionPanel.setLayout(new GridLayout(1, 7));

buttons = new JButton[buttonsNames.length];

filledBox = new JCheckBox("Filled");

unlimitDrawingsBox = new JCheckBox("Unlimited Drawing");

colorsBox = new JComboBox<>(allColorsString);

shapesBox = new JComboBox<>(shapesToDraw);

colorsBox.setMaximumRowCount(8);

shapesBox.setMaximumRowCount(4);

// This loop initializes each button and adds them to the selectionPanel

for (int i = 0; i < buttons.length; i++)

{

buttons[i] = new JButton(buttonsNames[i]);

buttons[i].addActionListener(this);

selectionPanel.add(buttons[i]);

}

buttons[0].setMnemonic('U');

buttons[1].setMnemonic('R');

buttons[2].setMnemonic('C');

colorsBox.addItemListener(this);

shapesBox.addItemListener(this);

filledBox.addItemListener(this);

unlimitDrawingsBox.addItemListener(this);

selectionPanel.add(colorsBox);

selectionPanel.add(shapesBox);

selectionPanel.add(filledBox);

selectionPanel.add(unlimitDrawingsBox);

add(selectionPanel, BorderLayout.NORTH);

}

/\*\*

\* The method sets the background color for the showCurrentColorPanel and

\* calls the setDrawingColor method in the DrawingPanel class.

\* <br> See setDrawingColor method: {@link DrawPanel#setDrawingColor}.

\* @param newColor recieved from combobox/sliders.

\*/

private void passColor(Color newColor)

{

showCurrentColorPanel.setBackground(newColor);

graphicsPanel.setDrawingColor(newColor);

}

/\*\*

\* The method initializes all JSlider's, JLabel's and the JPanel needed,

\* and add thems to the sliderPanel in a GridLayout which will finally

\* add the JPanel to the JFrame.

\* <br> It will also set the text and color for the rgbLabels.

\* <br> See the class GridLayout: {@link java.awt.GridLayout}.

\* <br> See the setForeground method to set rgbLabels color:

\* {@link javax.swing.JComponent#setForeground}.

\* <br> See other methods used: {@link #initializeSlider}, {@link #setColorLabelText},

\* {@link #setAllSlidersEnabled}.

\*/

private void createSliderPanel()

{

JPanel sliderPanel = new JPanel();

// Setting a GridLayout to stretch all components for a better look.

sliderPanel.setLayout(new GridLayout(8, 1));

JLabel showColorLabel = new JLabel("Current Color:", SwingConstants.CENTER);

showCurrentColorPanel = new JPanel();

showCurrentColorPanel.setBackground(Color.BLACK);

sliderPanel.add(showColorLabel);

sliderPanel.add(showCurrentColorPanel);

rgbSlider = new JSlider[3];

rgbLabel = new JLabel[3];

// Initialize sliders and labels set color for labels and set the visibility

// of the sliders to false.

initializeSlider();

setColorLabelText();

setAllSlidersEnabled(false);

// Sets the forground for each label and adds them to the sliderPanel

for (int i = 0; i < rgbSlider.length; i++)

{

rgbLabel[i].setForeground(rgbColor[i]);

sliderPanel.add(rgbLabel[i]);

sliderPanel.add(rgbSlider[i]);

}

add(sliderPanel, BorderLayout.EAST);

}

/\*\*

\* Initialize the sliders and labels and add a change listener to the sliders

\*/

private void initializeSlider()

{

for (int i = 0; i < rgbSlider.length; i++)

{

rgbSlider[i] = new JSlider(0, 255);

rgbLabel[i] = new JLabel("");

rgbSlider[i].addChangeListener(this);

}

}

/\*\*

\* This method sets visibility of all sliders.

\* <br> See the addChangeListener method: {@link java.awt.Component#setEnabled}.

\* <br> See setDrawingColor method: {@link DrawPanel#setDrawingColor}.

\* @param isEnabled Enable or disable the sliders.

\*/

private void setAllSlidersEnabled(boolean isEnabled)

{

for (int i = 0; i < rgbSlider.length; i++)

{

rgbSlider[i].setEnabled(isEnabled);

}

if (isEnabled)

{

graphicsPanel.setDrawingColor(new Color (rgbSlider[0].getValue(),

rgbSlider[1].getValue(), rgbSlider[2].getValue()));

}

}

/\*\*

\* This method sets the text for the rgbLabels.

\* <br> See the setText method: {@link javax.swing.JLabel#setText}.

\*/

private void setColorLabelText()

{

for (int i = 0; i < rgbLabel.length; i++)

{

rgbLabel[i].setText(String.format("%s %d", rgbString[i],

rgbSlider[i].getValue()));

}

}

/\*\*

\* The method initializes the JMenu, JMenu bar and two JMenuItem's.

\* The menu items are save and load which will be used to save and load the

\* drawing. There is a mnemonic to each of them.

\* <br> See the setMnemonic method: {@link javax.swing.AbstractButton#setMnemonic}.

\*/

private void createFileMenu()

{

JMenuBar fileBar = new JMenuBar();

JMenu fileMenu = new JMenu("File");

JMenuItem saveItem = new JMenuItem("Save");

JMenuItem loadItem = new JMenuItem("Load");

// Setting Mnemonics which will let us be able to access these items with

// alt + the fitting key.

fileMenu.setMnemonic('F');

saveItem.setMnemonic('S');

loadItem.setMnemonic('L');

fileMenu.add(saveItem);

fileMenu.add(loadItem);

add(fileMenu);

fileBar.add(fileMenu);

setJMenuBar(fileBar);

saveItem.addActionListener(new ActionListener()

{

/\*\*

\* Invokes the saveItem JMenuItem when pressed and calls a fitting

\* method.

\* <br> See saveDrawing method: {@link DrawPanel#saveDrawing}.

\* @param event Holds the current event.

\*/

@Override

public void actionPerformed(ActionEvent event)

{

graphicsPanel.saveDrawing();

}

});

loadItem.addActionListener(new ActionListener()

{

/\*\*

\* Invokes the loadItem JMenuItem when pressed and calls a fitting

\* method.

\* <br> See loadDrawing method: {@link DrawPanel#loadDrawing}.

\* @param event Holds the current event.

\*/

@Override

public void actionPerformed(ActionEvent event)

{

graphicsPanel.loadDrawing();

}

});

}

/\*\*

\* This methods calls the DrawPanel constructor to enable drawing on the

\* graphicsPanel and then adds it to the JFrame.

\* <br> See DrawPanel class: {@link DrawPanel}.

\*/

private void createGraphicsPanel()

{

graphicsPanel = new DrawPanel(this, statusLabel);

add(graphicsPanel);

}

/\*\*

\* Simply invokes the corresponding button in the button array and calls

\* a fitting method from the class DrawPanel.

\* <br> It is possible to undo, redo and clear a shape with the right actions.

\* <br> See DrawPanel button methods: {@link DrawPanel#undoShape},

\* {@link DrawPanel#redoShape}, {@link DrawPanel#clearDrawing}.

\* @param event Holds the current event.

\*/

@Override

public void actionPerformed(ActionEvent event)

{

Object source = event.getSource();

if (source == buttons[0])

{

graphicsPanel.undoShape();

return;

}

if (source == buttons[1])

{

graphicsPanel.redoShape();

return;

}

else

{

graphicsPanel.clearDrawing();

}

}

/\*\*

\* Simply invokes the corresponding item from the JComboBox and JCheckBox

\* and call a fitting method from the class DrawPanel.

\* <br> It is possible to set the color, fill, shape type and drawing limit with

\* the right actions.

\* @param event Holds the current event.

\*/

@Override

public void itemStateChanged(ItemEvent event)

{

Object source = event.getSource();

if (source == colorsBox)

{

// 13 equals the index of MakeColor in the combobox

if (colorsBox.getSelectedIndex() != 13)

{

// sends the fitting color to the index to the DrawPanel class

passColor(allColors[colorsBox.getSelectedIndex()]);

// disables visability to all sliders

setAllSlidersEnabled(false);

}

else

{

// enables visability to all sliders because MakeColor was chosen

// in the colorBox

setAllSlidersEnabled(true);

}

return;

}

if (source == shapesBox)

{

graphicsPanel.setDrawingShape(shapesBox.getSelectedIndex());

return;

}

if (source == filledBox)

{

graphicsPanel.setDrawingFilled(filledBox.isSelected());

return;

}

else

{

graphicsPanel.setUnlimitDrawings(unlimitDrawingsBox.isSelected());

}

}

/\*\*

\* Simply invokes the corresponding slider, set the rgbLabels text and call

\* the passColor method if the colorBox is set to MakeColor

\* @param event Holds the current event.

\*/

@Override

public void stateChanged(ChangeEvent event)

{

// 13 equals the index of MakeColor in the combobox

if(colorsBox.getSelectedIndex() == 13)

{

setColorLabelText();

passColor(new Color (rgbSlider[0].getValue(), rgbSlider[1].getValue(),

rgbSlider[2].getValue()));

}

}

}

**Class TestDraw – TestDraw.java**

// TestDraw.java

import javax.swing.\*;

import java.awt.\*;

/\*\*

\* This Class contains the main method.

\* The claass that holds the main method invokes the DrawFrame constructor to

\* start the program then sets the title of the frame and it's minimum size to

\* prevent the layout from not being fully displayed.

\* See the setMinimumSize method: {@link java.awt.Window#setMinimumSize}.

\* @author Lior Sabri, Ben Biton

\*/

public class TestDraw

{

/\*\*

\* static main method.

\* Invokes DrawFrame constructor. and sets title, minimum size, close operation

\* and visibility.

\* @param args command line arguments

\* see class {@link DrawFrame}.

\*/

public static void main(String[] args)

{

DrawFrame draw;

draw = new DrawFrame();

draw.setTitle("Shape Drawings");

draw.setMinimumSize(new Dimension(1000, 600));

draw.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

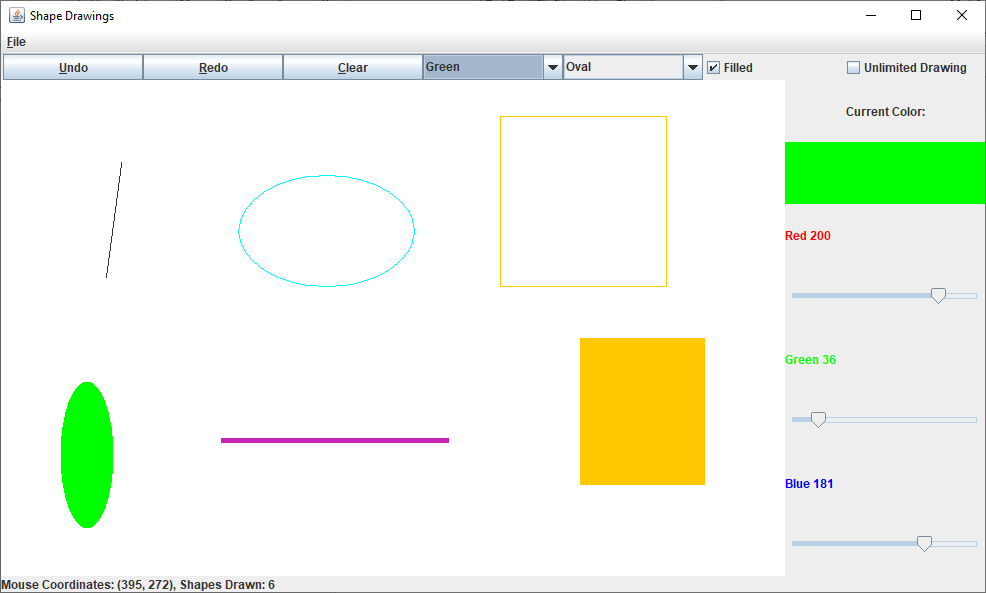
draw.setVisible(true);

} // end main(String[] args)

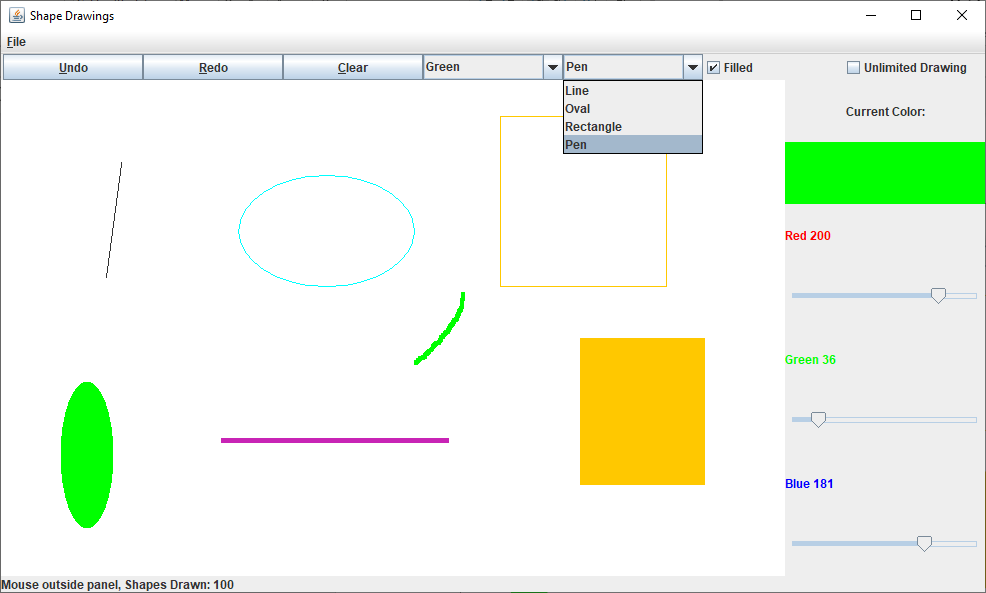
} // end class TestDraw

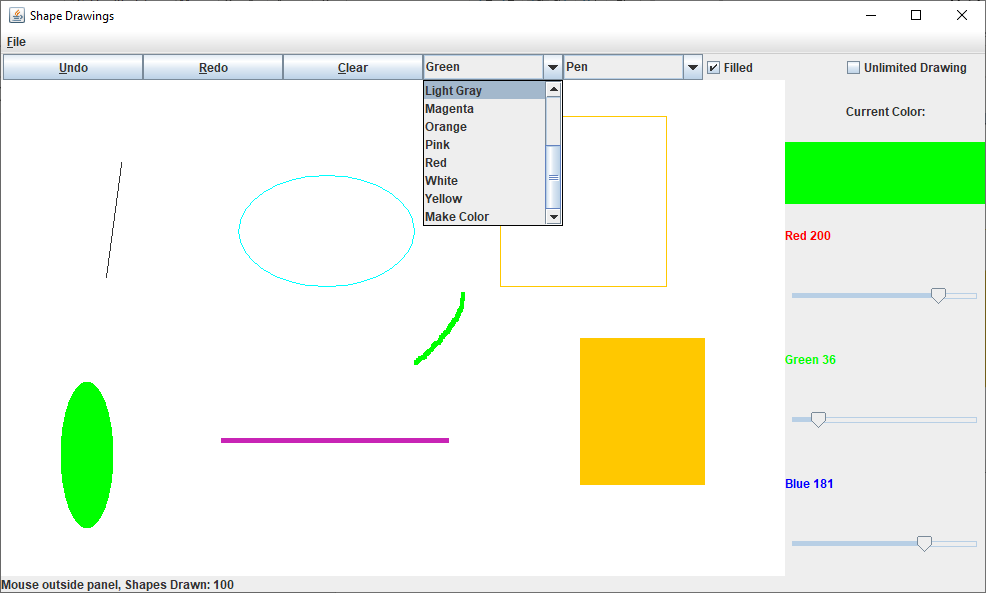
**Print Screen**

**Drawing:**

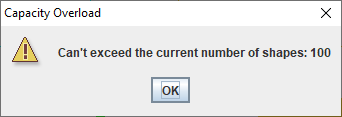


**Showing Drawing type options (notice there is an added type):**

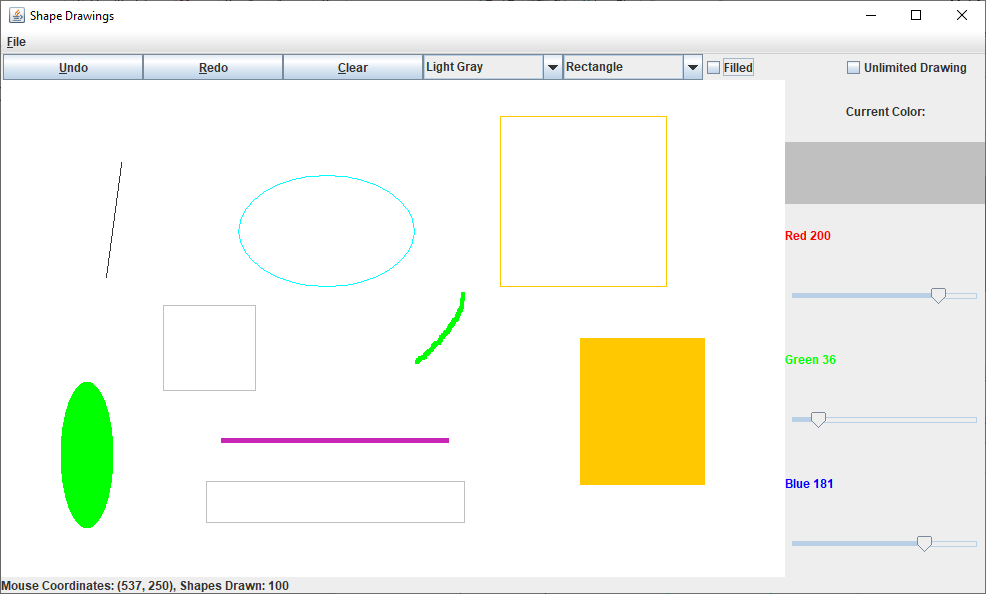


**Show color type options (notice there’s an added option Make Color which will enable the sliders to control the current color):**

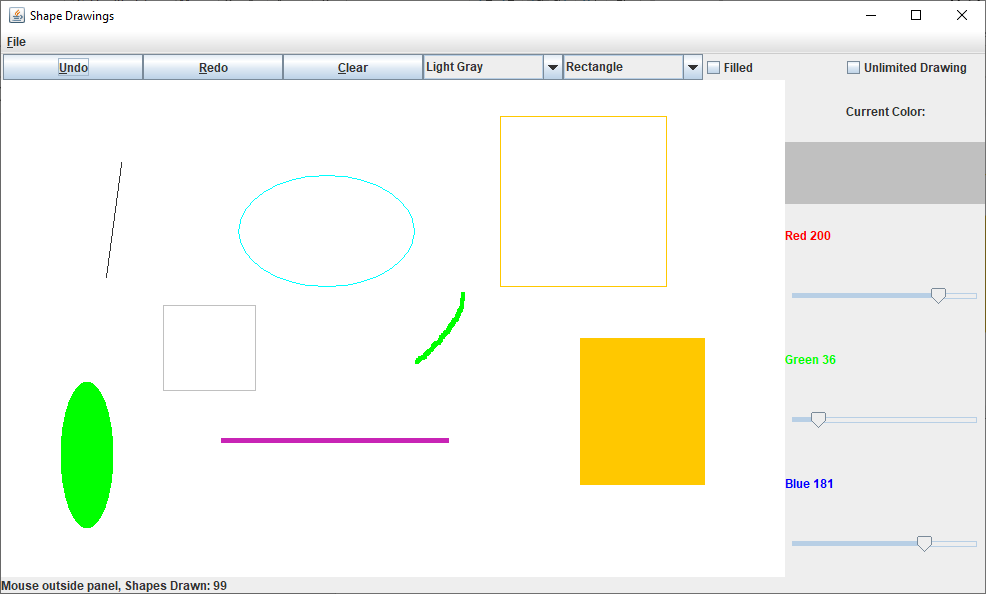
**Max capactiy message:**



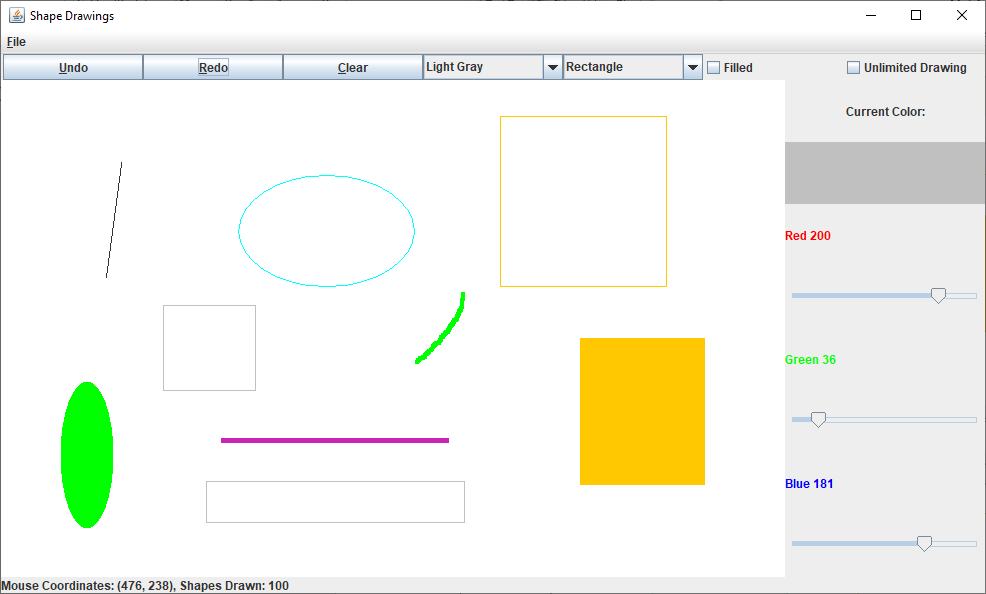
**Before undo:**



**Undo:**



**Redo:**



**Added option to save and load file:**

