

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/306918533>

Final Report: Java Programming Language. A Simple project to Draw Paint (Java language)

Technical Report · January 2016

CITATIONS

0

READS

61,209

2 authors, including:



[Mohamed Hazber](#)

University of Hail

17 PUBLICATIONS 104 CITATIONS

SEE PROFILE

Final Report: Java Programming Language

A Simple project to Draw Paint (Java language)

Mohamed A. G. Hazber

School of Computer Science and Technology

Under Supervision of Dr. ChangeWei Zhang

School of Electronic Information and Communications

Huazhong University of Science and Technology

Wuhan, China

moh_hazbar@yahoo.co.uk

ABSTRACT

Writing graphics applications in Java using Swing can be quite a daunting experience which requires understanding of some large libraries, and fairly advanced aspects of Java. In a graphical system, a windowing toolkit is usually responsible for providing a framework to make it relatively painless for a graphical user interface (GUI) to render the right bits to the screen at the right time. Both the AWT (abstract windowing toolkit) and Swing provide such a framework.

In this report, we designed and developed a simple painter project used to enable a user to draw any shape and any integrated graphic with any color using FreeHand (move the mouse using your hand to draw any shape and specify the coordinate in JPanel). Several tools such as Undo and Redo process, Clear JPanel, Set Background Color & set Foreground Color, Save paint (Panel) to file (*. JPG; *. GIF; *.*), and Open paint from image file are considered. The purpose of this project is to give you practice with graphical user interface programming in Java. This project implemented using the components from Java's awt and swing library in the Java programming language (NetBeans IDE7.2.1). As the final result of our project is enabling you to use FreeHand to draw as an easy way to draw the Circle, Line, Rectangle, Square, and Oval, and integrated graphics such as a car, a street, a football stadium, traffic signals and others.

Keywords: NetBeans IDE 7.2.1, AWT, Swing, GUI.

Contents

ABSTRACT.....	1
1. Introduction	3
1.1 Overview	3
1.2 Object-oriented Programming.....	3
1.3 The Basic GUI (graphical user interface) Application	4
2. Graphics and Painting.....	6
2.1 Overview of the Java 2D API Concepts	6
2.2 Coordinates	7
2.3 Colors	8
2.4 Shapes	9
2.5 Graphics2D	11
3. Painting in AWT and Swing.....	12
3.1 Evolution of the Swing Paint System.....	13
3.2 Painting in AWT	13
4. Design and Development our project	16
4.1 Program Ability (Objectives):.....	17
4.2 System Framework.....	17
4.3 Components.....	18
4.4 Program Structure and Results.....	19
4.4.1 Preview (System Interface).....	19
4.4.2 Undo and Redo	21
4.4.3 SetColor	22
4.4.4 setBackColor.....	23
4.4.5 Save to File	24
4.4.6 Open Image from File.....	25
5. Conclusion.....	25
References.....	26

1. Introduction

1.1 Overview

Java is a general-purpose, concurrent, class-based, object-oriented computer programming language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another. Java applications are typically compiled to byte code (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture. Java is, as of 2012, one of the most popular programming languages in use, particularly for client-server web applications, with a reported 10 million users [1][2]. Java was originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

Java [3] can be used to write applications and applets. A Java application is similar to any other high-level language program: It can only be compiled and then run on the same machine. An applet is compiled on one machine, stored on a server in binary, and can be sent to another machine over the Internet to be interpreted by a Java-aware browser. Java comes with a large library of ready-made classes and objects. The key difference between Java 1.0 and 1.1 was in this library. Similarly, Java 2.0 has a very much larger library for handling user interfaces (Swing by name) but only small changes to the core of the language.

1.2 Object-oriented Programming

Java supports object-oriented programming techniques that are based on a hierarchy of classes and well-defined and cooperating objects [4].

Classes: A class is a structure that defines the data and the methods to work on that data. When you write programs in Java, all program data is wrapped in a class, whether it is a class you write or a class you use from the Java API libraries. Classes in the Java API libraries define a set of objects that share a common structure and behavior.

Objects: An instance is a synonym for object. A newly created instance has data members and methods as defined by the class for that instance.

Well-Defined Boundaries and Cooperation: Class definitions must allow objects to cooperate during execution.

Inheritance and Polymorphism: One object-oriented concept that helps objects work together is inheritance. Inheritance defines relationships among classes in an object-oriented language. The relationship is one of parent to child where the child or extending class inherits all the attributes (methods and data) of the parent class. In Java, all classes descend from `java.lang.Object` and inherit its methods. Figure 1 shows the class hierarchy as it descends from `java.lang.Object` for the classes in the user interface example above. The `java.lang.Object` methods are also shown because they are inherited and implemented by all of its subclasses, which is every class in the Java API libraries. `java.lang.Object` defines the core set of behaviors that all classes have in common.

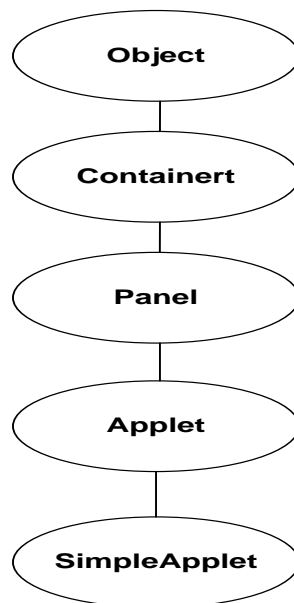


Figure 1. Object Hierarchy

1.3 The Basic GUI (graphical user interface) Application

There are two basic types of GUI program in Java [5]: stand-alone applications and (online) applets. An applet is a program that runs in a rectangular area on a Web page. Applets are generally small programs, meant to do fairly simple things, although there is nothing to stop them from being very complex. A GUI program offers a much richer type of user interface, where the user uses a mouse and keyboard to interact with GUI components such as windows, menus, buttons, check boxes, text input boxes, scroll bars, and so on.

JFrame and JPanel: In a Java GUI program, each GUI component in the interface is represented by an object in the program. One of the most fundamental types of component is the window. Windows have many behaviors. They can be opened and closed. They can be resized.

They have “titles” that are displayed in the title bar above the window. And most important, they can contain other GUI components such as buttons and menus. Java, of course, has a built-in class to represent windows. There are actually several different types of window, but the most common type is represented by the `JFrame` class (which is included in the package `javax.swing`).

A **JFrame** is an independent window that can, for example, act as the main window of an application. One of the most important things to understand is that a `JFrame` object comes with many of the behaviors of windows already programmed in. In particular, we have one `JFrame` “InterfaceForm” that contains all the components, which enables the user to work on our system easily and draw any shape in `Panel` and the ability to be opened and closed.

JPanel is another of the fundamental classes in Swing [6]. The basic `JPanel` is, again, just a blank rectangle. There are two ways to make a useful `JPanel` : The first is to add other components to the panel; the second is to draw something in the panel. Both of these techniques are illustrated in our sample project. In fact, you will find `JPanel` in the program: `content`, `display` `Panel`, which is used as a drawing surface.

Components and Layout: Another way of using a `JPanel` is as a container to hold other components. In our project, we used NetBeans IDE 7.2.1 to create all components in `JFrame` and `JPanel`.

Events and Listeners: The structure of containers and components sets up the physical appearance of a GUI, but it doesn’t say anything about how the GUI behaves. That is, what can the user do to the GUI and how will it respond? GUIs are largely event-driven; that is, the program waits for events that are generated by the user’s actions (or by some other cause). When an event occurs, the program responds by executing an event-handling method. In order to program the behavior of a GUI, you have to write event-handling methods to respond to the events that you are interested in. The most common technique for handling events in Java is to use event listeners. A listener is an object that includes one or more event-handling methods. When an event is detected by another object, such as a button or menu, the listener object is notified and it responds by running the appropriate event-handling method. An event is detected or generated by an object. Another object, the listener, has the responsibility of responding to the event. The event itself is actually represented by a third object, which carries information about the type of event, when it occurred, and so on. This division of responsibilities makes it easier to

organize large programs. As an example, consider the Undo or Redo button in our sample program. When the user clicks the button, an event is generated.

2. Graphics and Painting

Everything you see on a computer screen has to be drawn there, even the text. The (online) Java API includes a range of classes and methods that are devoted to drawing. In this section, I'll look at some of the most basic of these that helping us to achieve our project.

The physical structure of a GUI is built of components. The term component refers to a visual element in a GUI, including buttons, menus, text-input boxes, scroll bars, check boxes, and so on. In Java, GUI components are represented by objects belonging to subclasses of the class `java.awt.Component`. In a graphical system, a windowing toolkit is usually responsible for providing a framework to make it relatively painless for a graphical user interface (GUI) to render the right bits to the screen at the right time. Both the AWT (abstract windowing toolkit) and Swing provide such a framework. But the APIs that implement it are not well understood by some developers -- a problem that has led to programs not performing as well as they could.

In order to use graphics in Java programs, there are a number of libraries we need to import. For the sake of what will be covered in these notes, you need the following statements:

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.Graphics;
import java.awt.geom.*
import java.awt.Color;
import javax.swing.JLabel;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.ImageIcon;
import java.awt.BorderLayout;
import java.awt.event.*;
import java.awt.image.BufferedImage;
import java.awt.image.MemoryImageSource;
import java.awt.image.PixelGrabber;
import java.io.File;
import java.io.IOException;
import javax.imageio.*;
```

2.1 Overview of the Java 2D API Concepts

The Java 2D™ API [7] provides two-dimensional graphics, text, and imaging capabilities for Java™ programs through extensions to the Abstract Windowing Toolkit (AWT). This comprehensive rendering package supports line art, text, and images in a flexible, full-featured

framework for developing richer user interfaces, sophisticated drawing programs, and image editors. Java 2D objects exist on a plane called user coordinate space, or just user space. When objects are rendered on a screen or a printer, user space coordinates are transformed to device space coordinates. The following links are useful to start learning about the Java 2D API:

- **Graphics class** [8]
- `Graphics2D` class [9]

The Java 2D API provides following capabilities:

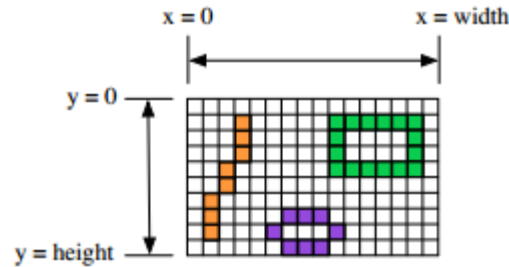
- A uniform rendering model for display devices and printers
- A wide range of geometric primitives, such as curves, rectangles, and ellipses, as well as a mechanism for rendering virtually any geometric shape
- Mechanisms for performing hit detection on shapes, text, and images
- A compositing model that provides control over how overlapping objects are rendered
- Enhanced color support that facilitates color management
- Support for printing complex documents
- Control of the quality of the rendering through the use of rendering hints

2.2 Coordinates

The Java 2D™ API maintains two coordinate spaces:

- User space – The space in which graphics primitives are specified
- Device space – The coordinate system of an output device such as a screen, window, or a printer.

The screen of a computer is a grid of little squares called pixels. The color of each pixel can be set individually, and drawing on the screen just means setting the colors of individual pixels. When the default transformation from user space to device space is used, the origin of user space is the upper-left corner of the component's drawing area. The x coordinate increases to the right and the y coordinate increases downward, as shown in the following figure. The top-left corner of a window is 0,0. All coordinates are specified using integers, which is usually sufficient. However, some cases require floating point or even double precision which are also supported.



A graphics context draws in a rectangle made up of pixels. A position in the rectangle is specified by a pair of integer coordinates, (x,y). The upper left corner has coordinates (0,0). The x coordinate increases from left to right, and the y coordinate increases from top to bottom. The illustration shows a 16-by-10 pixel component (with very large pixels). A small line, rectangle, and oval are shown as they would be drawn by coloring individual pixels. (Note that, properly speaking, the coordinates don't belong to the pixels but to the grid lines between them.)

For any component, you can find out the size of the rectangle that it occupies by calling the instance methods `getWidth()` and `getHeight()`, which return the number of pixels in the horizontal and vertical directions, respectively.

For example, you can use a `paintComponent()` method that looks like:

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    int width = getWidth(); // Find out the width of this component.
    int height = getHeight(); // Find out its height.
    ... // Draw the content of the component.
}
```

Of course, your drawing commands will have to take the size into account. That is, they will have to use (x,y) coordinates that are calculated based on the actual height and width of the component.

2.3 Colors

We will probably want to use some color when you draw. Java is designed to work with the **RGB color system**. An RGB color is specified by three numbers that give the level of red, green, and blue, respectively, in the color. A color in Java is an object of the class, `java.awt.Color`. You can construct a new color by specifying its red, blue, and green components.

For example, `Color myColor = new Color(r,g,b);`

Java defines a `Color` class; instances of this class represent various colors.

At the simplest level, we could pick one of 13 predefined colors from Java's virtual box of crayons:

- **Color:** `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`.
- **RGB:** Of course, there are many more colors we might want. We can specify other colors using the **RGB model**, which specifies a color with a red value, a green value, and a blue value (each called **channels**). For example, the red value is how much red we want in our color. In different graphics system, the individual color values might be represented in different ways, but in Java, we use 8 bits to each color value. This gives us $256 (2^8)$ choices for each of the R, G, and B values, and choices range from 0 to 255.

So, we'd specify a color as follows:

Component	Minimum Value	Maximum Value
Red Value	0 (no red)	255 (all red)
Green Value	0 (no green)	255 (all green)
Blue Value	0 (no blue)	255 (all blue)

So, a color is represented as an ordered triple. (0, 0, 0) is the absence of color, or black. (255, 255, 255) means all colors are at their maximum value, which corresponds to white.

We can create `Color` objects using RGB values:

```
Color colorName = new Color( red value , green value , blue value );
```

2.4 Shapes

The `Graphics` class includes a large number of instance methods for drawing various shapes, such as lines, rectangles, and ovals. The shapes are specified using the (x,y) coordinate system described above. They are drawn in the current drawing color of the graphics context. The current drawing color is set to the foreground color of the component when the graphics context is created, but it can be changed at any time using the `setColor()` method. Here is a list of some of

the most important drawing methods. With all these commands, any drawing that is done outside the boundaries of the component is ignored. Note that all these methods are in the Graphics class, so they all must be called through an object of type Graphics.

- **drawString(String str, int x, int y):** Draws the text given by the string str. The string is drawn using the current color and font of the graphics context. x specifies the position of the left end of the string. y is the y-coordinate of the baseline of the string. The baseline is a horizontal line on which the characters rest. Some parts of the characters, such as the tail on a y or g, extend below the baseline.
- **drawLine(int x1, int y1, int x2, int y2):** Draws a line from the point (x1,y1) to the point (x2,y2).
- **drawRect(int x, int y, int width, int height):** Draws the outline of a rectangle. The upper left corner is at (x,y), and the width and height of the rectangle are as specified. If width equals height, then the rectangle is a square. If the width or the height is negative, then nothing is drawn.
- **drawOval(int x, int y, int width, int height):** Draws the outline of an oval. The oval is one that just fits inside the rectangle specified by x, y, width, and height. If width equals height, the oval is a circle.
- **drawRoundRect(int x, int y, int width, int height, int xdiam, int ydiam):** Draws the outline of a rectangle with rounded corners. The basic rectangle is specified by x, y, width, and height, but the corners are rounded. The degree of rounding is given by xdiam and ydiam. The corners are arcs of an ellipse with horizontal diameter xdiam and vertical diameter ydiam. A typical value for xdiam and ydiam is 16, but the value used should really depend on how big the rectangle is.
- **draw3DRect(int x, int y, int width, int height, boolean raised):** Draws the outline of a rectangle that is supposed to have a three-dimensional effect, as if it is raised from the screen or pushed into the screen. The basic rectangle is specified by x, y, width, and height.
- **drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** Draws part of the oval that just fits inside the rectangle specified by x, y, width, and height.
- **fillRect(int x, int y, int width, int height):** Draws a filled-in rectangle. This fills in the interior of the rectangle that would be drawn by drawRect(x,y,width,height).

- **fillOval(int x, int y, int width, int height):** Draws a filled-in oval.
- **fillRoundRect(int x, int y, int width, int height, int xdiam, int ydiam) :** Draws a filled-in rounded rectangle.
- **fill3DRect(int x, int y, int width, int height, boolean raised):** Draws a filled-in three-dimensional rectangle.
- **fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** Draw a filled-in arc. This looks like a wedge of pie, whose crust is the arc that would be drawn by the drawArc method.

2.5 Graphics2D

This Graphics2D class extends the Graphics class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout. This is the fundamental class for rendering 2-dimensional shapes, text and images on the Java(tm) platform. All drawing in Java is done through an object of type Graphics. The Graphics class provides basic commands for such things as drawing shapes and text and for selecting a drawing color. These commands are adequate in many cases, but they fall far short of what's needed in a serious computer graphics program. Java has another class, Graphics2D, that provides a larger set of drawing operations. Graphics2D is a sub-class of Graphics, so all the methods from the Graphics class are also available in a Graphics2D. The paintComponent() method of a JComponent gives you a graphics context of type Graphics that you can use for drawing on the component. In fact, the graphics context actually belongs to the sub-class Graphics2D (in Java version 1.2 and later), and can be type-cast to gain access to the advanced Graphics2D drawing methods:

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2;
    g2 = (Graphics2D)g;
    .
    . // Draw on the component using g2.
    .
}
```

In our example:

```
public void draw(Graphics2D ga){
    ga.setColor(color);
    if (!IsFillColor)
        ga.drawArc(getX1(), getY1(), getWidth(), getHeight(), 0, 360);
    else
        ga.fillArc(getX1(), getY1(), getWidth(), getHeight(), 0, 360);
}
```

Drawing in Graphics2D is based on shapes, which are objects that implement an interface named Shape. Shape classes include *Line2D*, *Rectangle2D*, *Ellipse2D*, *Arc2D*, and *GeneralPath*, among others; all these classes are defined in the package *java.awt.geom*. Graphics2D has methods *draw(Shape)* and *fill(Shape)* for drawing the outline of a shape and for filling its interior. Advanced capabilities include: lines that are more than one pixel thick, dotted and dashed lines, filling a shape with a texture (that is, with a repeated image), filling a shape with a gradient, and so-called “anti-aliased” drawing (which cuts down on the jagged appearance along a slanted line or curve). In the Graphics class, coordinates are specified as integers and are based on pixels. The shapes that are used with Graphics2D use real numbers for coordinates, and they are not necessarily bound to pixels. In fact, we can change the coordinate system and use any coordinates that are convenient to our application. In computer graphics terms, you can apply a “transformation” to the coordinate system. The transformation can be any combination of translation, scaling, and rotation.

3. Painting in AWT and Swing

In a graphical system [10], a windowing toolkit is usually responsible for providing a framework to make it relatively painless for a graphical user interface (GUI) to render the right bits to the screen at the right time. Both the AWT (abstract windowing toolkit) and Swing provide such a framework. But the APIs that implement it are not well understood by some developers -- a problem that has led to programs not performing as well as they could.

This section explains the AWT and Swing paint mechanisms in detail. Its purpose is to help developers write correct and efficient GUI painting code.

3.1 Evolution of the Swing Paint System

When the original AWT API was developed for JDK 1.0, only heavyweight components existed ("heavyweight" means that the component has its own opaque native window). This allowed the AWT to rely heavily on the paint subsystem in each native platform. This scheme took care of details such as damage detection, clip calculation, and z-ordering. With the introduction of lightweight components in JDK 1.1 (a "lightweight" component is one that reuses the native window of its closest heavyweight ancestor), the AWT needed to implement the paint processing for lightweight components in the shared Java code. Consequently, there are subtle differences in how painting works for heavyweight and lightweight components.

After JDK 1.1, when the Swing toolkit was released, it introduced its own spin on painting components. For the most part, the Swing painting mechanism resembles and relies on the AWT's. But it also introduces some differences in the mechanism, as well as new APIs that make it easier for applications to customize how painting works.

3.2 Painting in AWT

To understand how AWT's painting API works helps to know what triggers a paint operation in a windowing environment. In AWT, there are two kinds of painting operations: *system-triggered painting*, and *application-triggered painting*.

System-triggered Painting: In a system-triggered painting operation, the system requests a component to render its contents, usually for one of the following reasons:

- The component is first made visible on the screen.
- The component is resized.
- The component has damage that needs to be repaired. (For example, something that previously obscured the component has moved, and a previously obscured portion of the component has become exposed).

App-triggered Painting: In an application-triggered painting operation, the component decides it needs to update its contents because its internal state has changed. (For example, a button detects that a mouse button has been pressed and determines that it needs to paint a "depressed" button visual).

The Paint Method: Regardless of how a paint request is triggered, the AWT uses a "callback" mechanism for painting, and this mechanism is the same for both heavyweight and lightweight components. This means that a program should place the component's rendering code inside a

particular overridden method, and the toolkit will invoke this method when it's time to paint. The method to be overridden is in `java.awt.Component`:

```
public void paint(Graphics g)
```

When AWT invokes this method, the `Graphics` object parameter is pre-configured with the appropriate state for drawing on this particular component:

- The `Graphics` object's color is set to the component's foreground property.
- The `Graphics` object's font is set to the component's font property.
- The `Graphics` object's translation is set such that the coordinate (0,0) represents the upper left corner of the component.
- The `Graphics` object's clip rectangle is set to the area of the component that is in need of repainting.

Programs must use this `Graphics` object (or one derived from it) to render output. They are free to change the state of the `Graphics` object as necessary.

Here is a simple example of a paint callback which renders a filled circle in the bounds of a component:

```
public void paint(Graphics g) {  
    // Dynamically calculate size information  
    Dimension size = getSize();  
    // diameter  
    int d = Math.min(size.width, size.height);  
    int x = (size.width - d)/2;  
    int y = (size.height - d)/2;  
    // draw circle (color already set to foreground)  
    g.fillOval(x, y, d, d);  
    g.setColor(Color.black); g.drawOval(x, y, d, d);  
}
```

Developers who are new to AWT might want to take a peek at the PaintDemo example, which provides a runnable program example of how to use the paint callback in an AWT program.

In general, programs should avoid placing rendering code at any point where it might be invoked outside the scope of the paint callback. Why? Because such code may be invoked at times when it is not appropriate to paint -- for instance, before the component is visible or has access to a valid Graphics object. It is not recommended that programs invoke paint() directly. To enable app-triggered painting, the AWT provides the following java.awt.Component methods to allow programs to asynchronously request a paint operation:

```
public void repaint()  
public void repaint(long tm)  
public void repaint(int x, int y, int width, int height)  
public void repaint(long tm, int x, int y, int width, int height)
```

The following code shows a simple example of a mouse listener that uses repaint() to trigger updates on a theoretical button component when the mouse is pressed and released

```
MouseListener l = new MouseAdapter() {  
    public void mousePressed(MouseEvent e) {  
        MyButton b = (MyButton)e.getSource();  
        b.setSelected(true);  
        b.repaint(); }  
        public void mouseReleased(MouseEvent e) {  
            MyButton b = (MyButton)e.getSource();  
            b.setSelected(false);  
            b.repaint(); } };
```

paint() vs. update(): Why do we make a distinction between "system-triggered" and. "app-triggered" painting? Because AWT treats each of these cases slightly differently for heavyweight

components (the lightweight case will be discussed later), which is unfortunately a source of great confusion. For heavyweight components, these two types of painting happen in the two distinct ways, depending on whether a painting operation is system-triggered or app-triggered.

System-triggered painting: This is how a system-triggered painting operation takes place:

The **AWT** determines that either part or all of a component needs to be painted.

The **AWT** causes the event dispatching thread to invoke **paint()** on the component.

App-triggered painting: An app-triggered painting operation takes place as follows:

- The program determines that either part or all of a component needs to be repainted in response to some internal state change.
- The program invokes `repaint()` on the component, which registers an asynchronous request to the AWT that this component needs to be repainted.

"Smart" Painting: While the AWT attempts to make the process of rendering components as efficient as possible, a component's `paint()` implementation itself can have a significant impact on overall performance. Two key areas that can affect this process are:

- Using the clip region to narrow the scope of what is rendered.
- Using internal knowledge of the layout to narrow the scope of what children are painted (lightweights only).

If our component is simple -- for example, if it's a pushbutton -- then it's not worth the effort to factor the rendering in order to only paint the portion that intersects the clip rectangle; it's preferable to just paint the entire component and let the graphics clip appropriately. However, if you've created a component that renders complex output, like a text component, then it's critical that your code use the clip information to narrow the amount of rendering.

4. Design and Development our project

In this section we describe how our project designed and implementation. This is a simple Painter project using the JAVA language. The program uses 10 classes to build the entire structure. Please compile using the JAVA SDK, and run the `Main_DrawPaintProject.java` as the main program to call the main form interface `InterfaceForm.java`.

4.1 Program Ability (Objectives):

- Draw Circle, Line, Rectangle, Square, and Oval using FreeHand (move the mouse using your hand to draw any shape and specify the coordinate in JPanel).
- Undo and Redo process.
- Clear JPanel
- Set Background Color & set Foreground Color.
- Save paint (Panel) to file (*. JPG; *. GIF; *.*)
- Open paint from file
- The system enables you to use FreeHand to draw (move the mouse using your hand to draw any shape and specify the coordinate in JPanel) as an easy way to draw the integrated paint, for example, a car , a street , a football stadium , traffic signals and others.

4.2 System Framework

The main objective of the proposed framework to describe how designed and developed our project to draw paint such as shapes and what are the required classes that are used to to build the entire structure. This project is used to avoid limiting manual work, reducing cost and time, and improving the efficiency for building the shape. The proposed framework to painter is shown in Figure 2 that includes the following stages:

- **Interface Model:** Which operation do you want to do that includes as the follows:
 - 1- The **input system** is user chosen operation what is the process that wants to be painted by the user, whether draw a shape on the panel or operations on the drawing located.
 - 2- **Buttons paint:** What are the available operations that you can to do such as Line, Circle, Square, Rectangle, Oval, and so on.
 - 3- **Menu Color:** Button and radio group are used to setColor for shape or setBackground for the panel.

- **Point.java**
- **DrawLine.java**
- **DrawCircle.java**
- **DrawRectangle.java**
- **DrawSquare.java**
- **DrawOval.java**
- **DrawString**

More detail about the collection components shown in Figure 3.

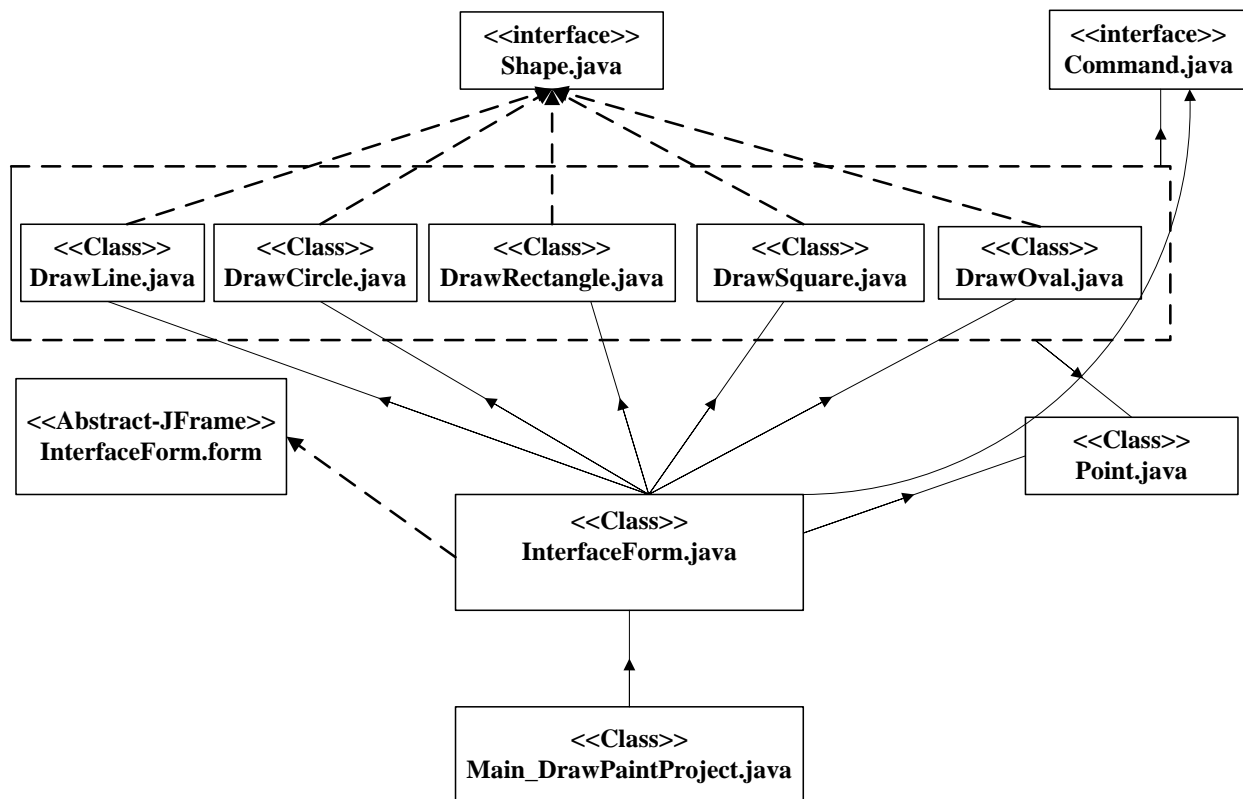


Figure 3. Collection Framework

4.4 Program Structure and Results

This is the public methods of each class which collaboratively used between classes

4.4.1 Preview (System Interface)

Figure 4 shows the main interface of our project and what abilities available (tools) are in our program for drawing, coloring, and save and open image file. Also explain how you can draw an integrated paint from several geometric shapes and different colors, for example, a car, a street, a

football stadium, traffic signals and others.

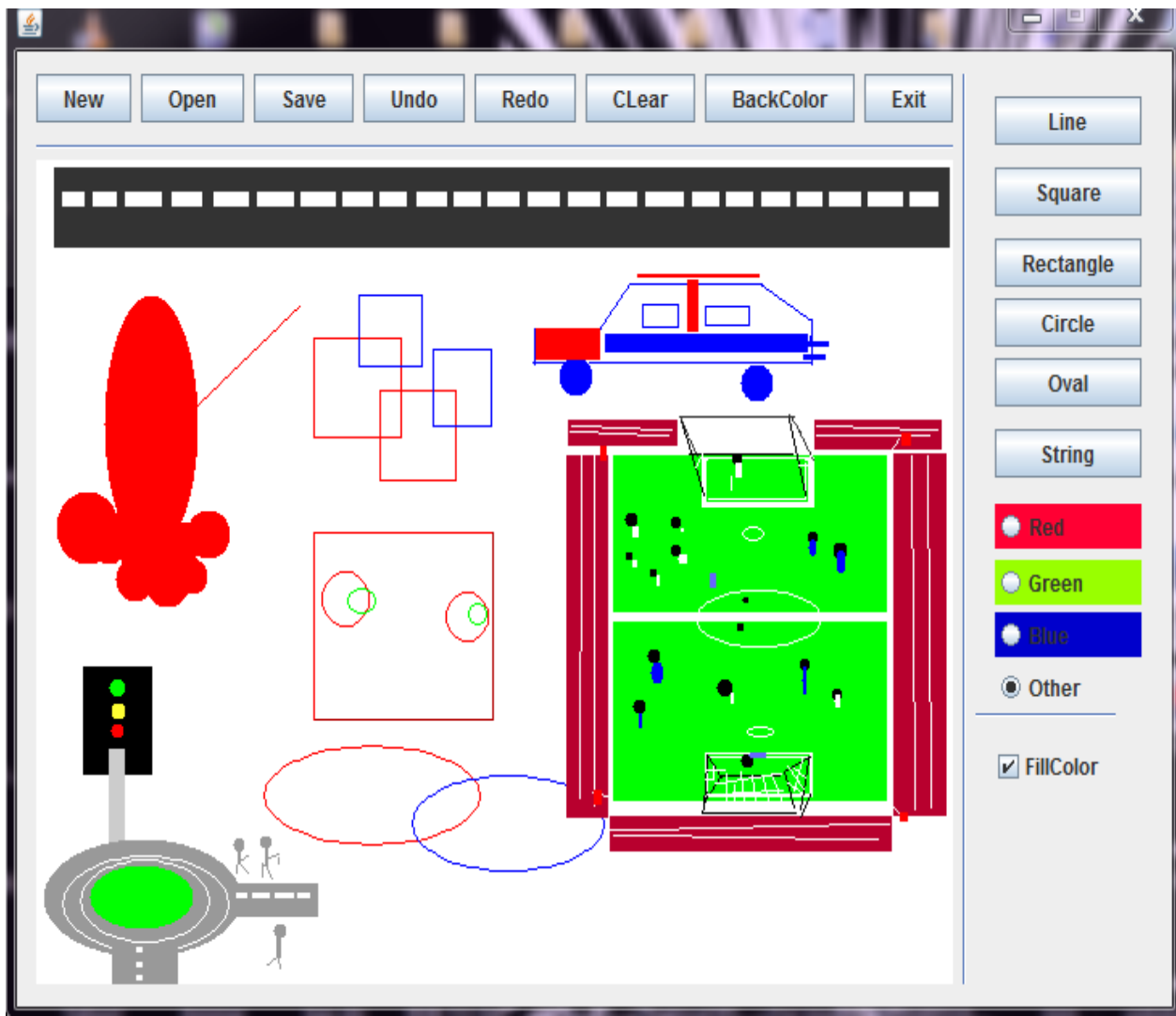


Figure 4. Main interface of our project

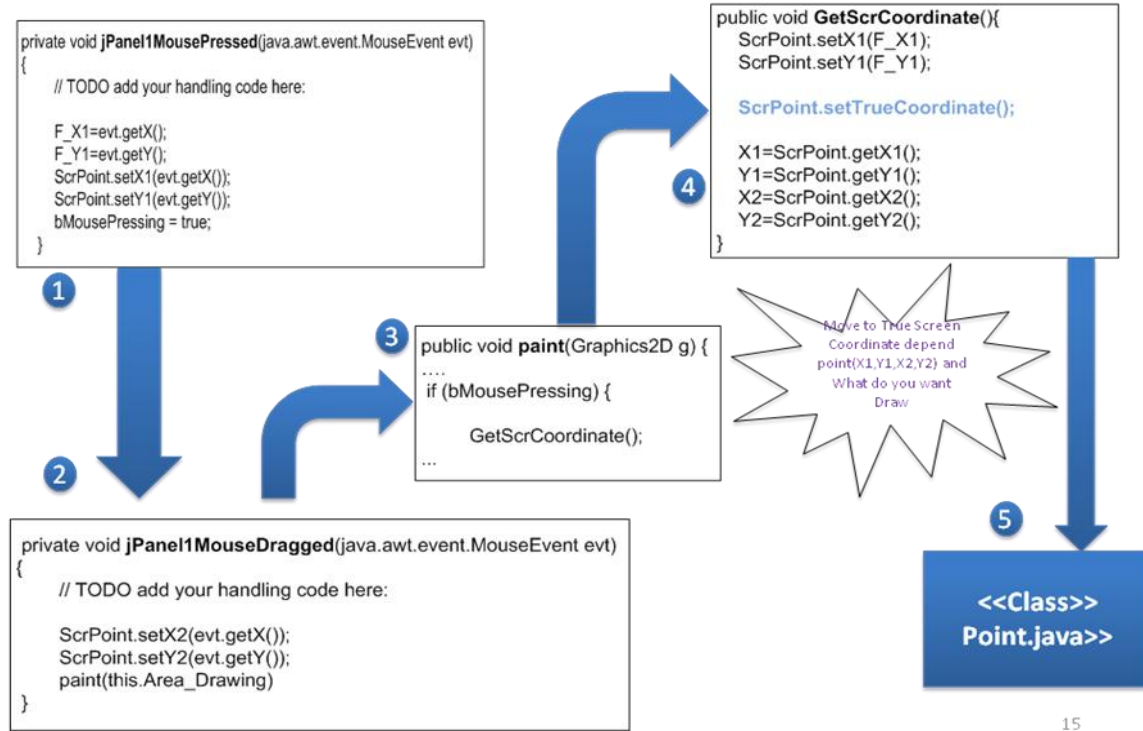
<<Interface>> Shape

```
public interface Shape {  
    public void draw(Graphics2D g);  
}
```

<<Interface>> Command

```
public interface Command {  
    public final static int LINE = 2;  
    public final static int CIRCLE = 4;  
    public final static int RECTANGLE = 8;  
    public final static int SQUARE=12;  
    public final static int Oval=16;  
}
```

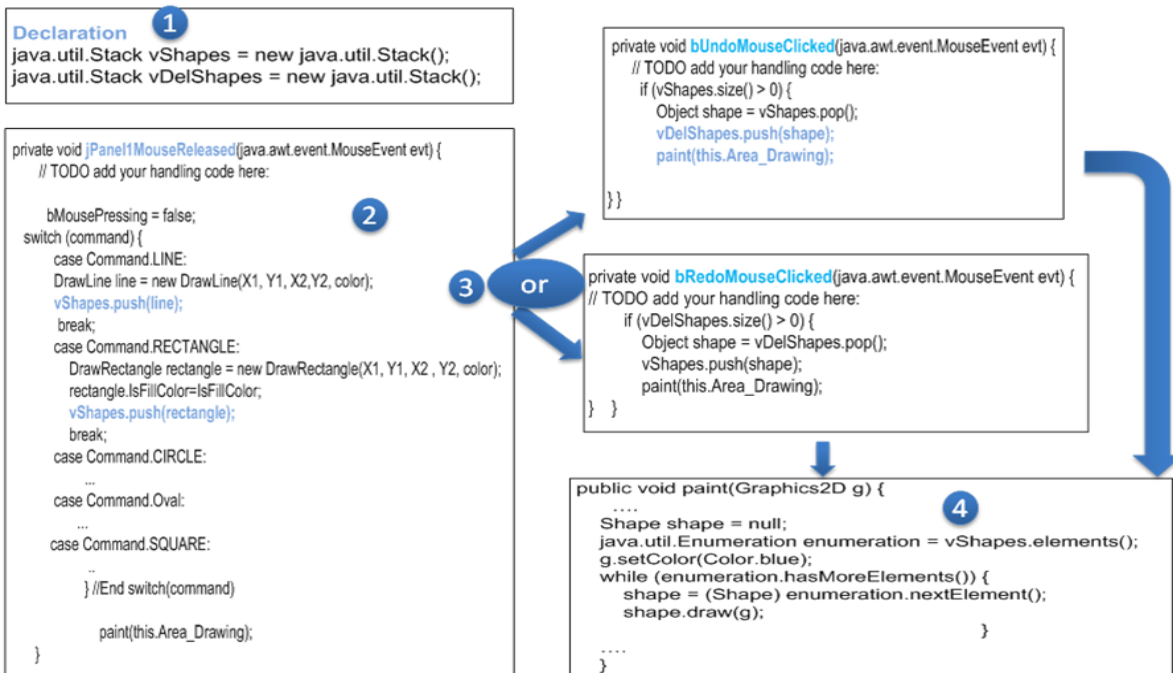
Get Screen Coordinate-Point(X1,Y1,X2,Y2)



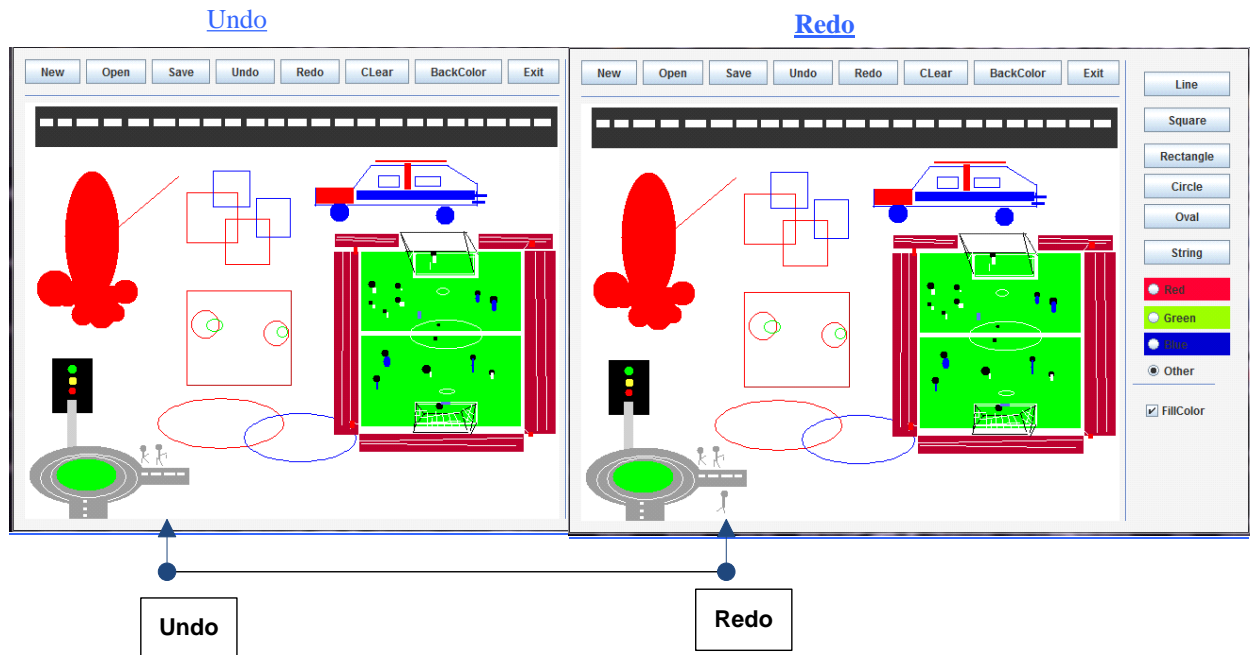
15

4.4.2 Undo and Redo

Undo and Redo



23

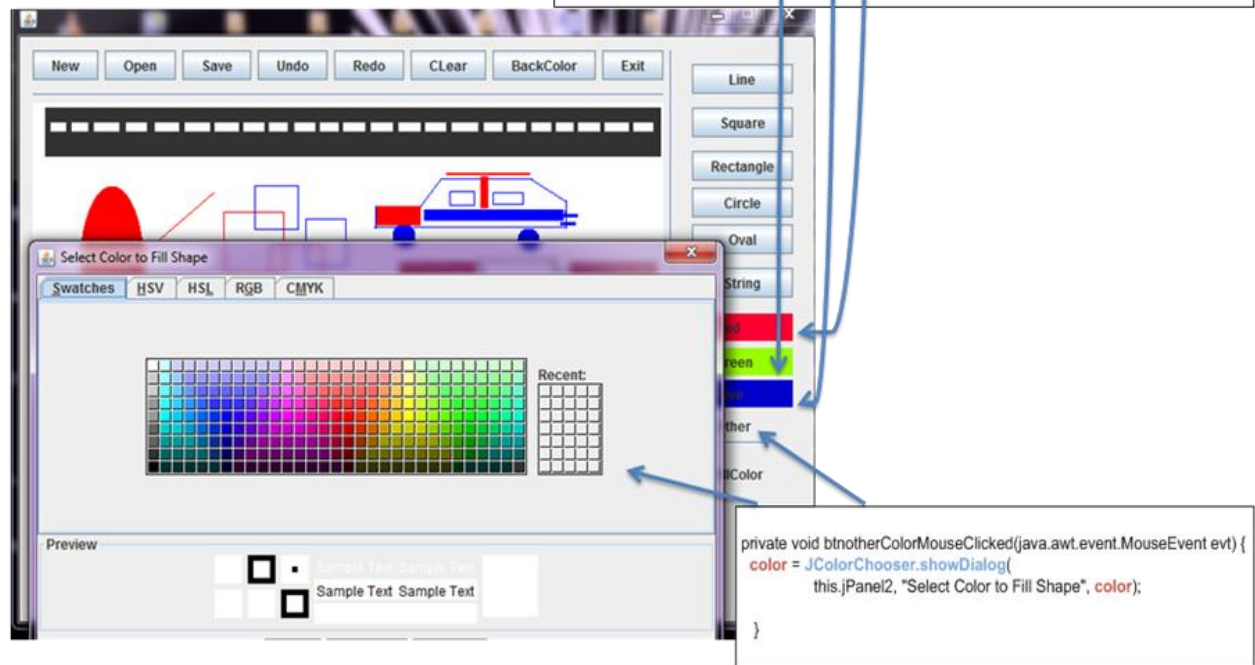


4.4.3 SetColor

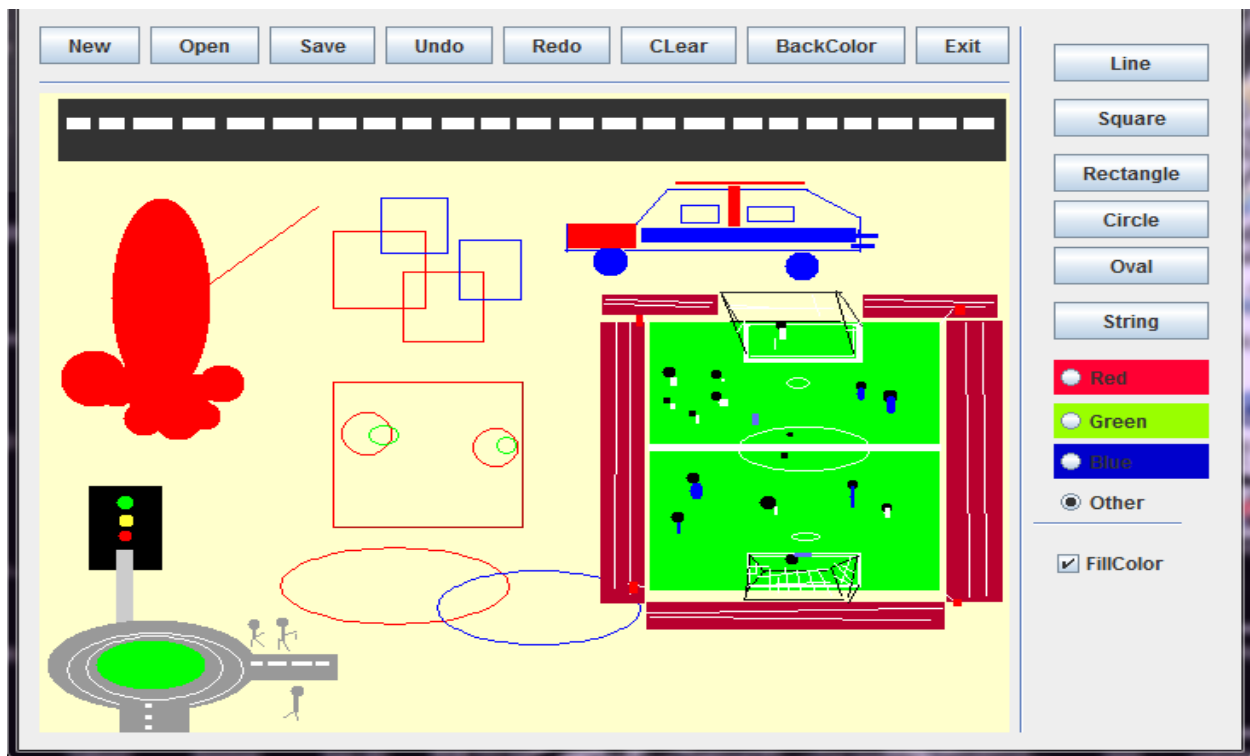
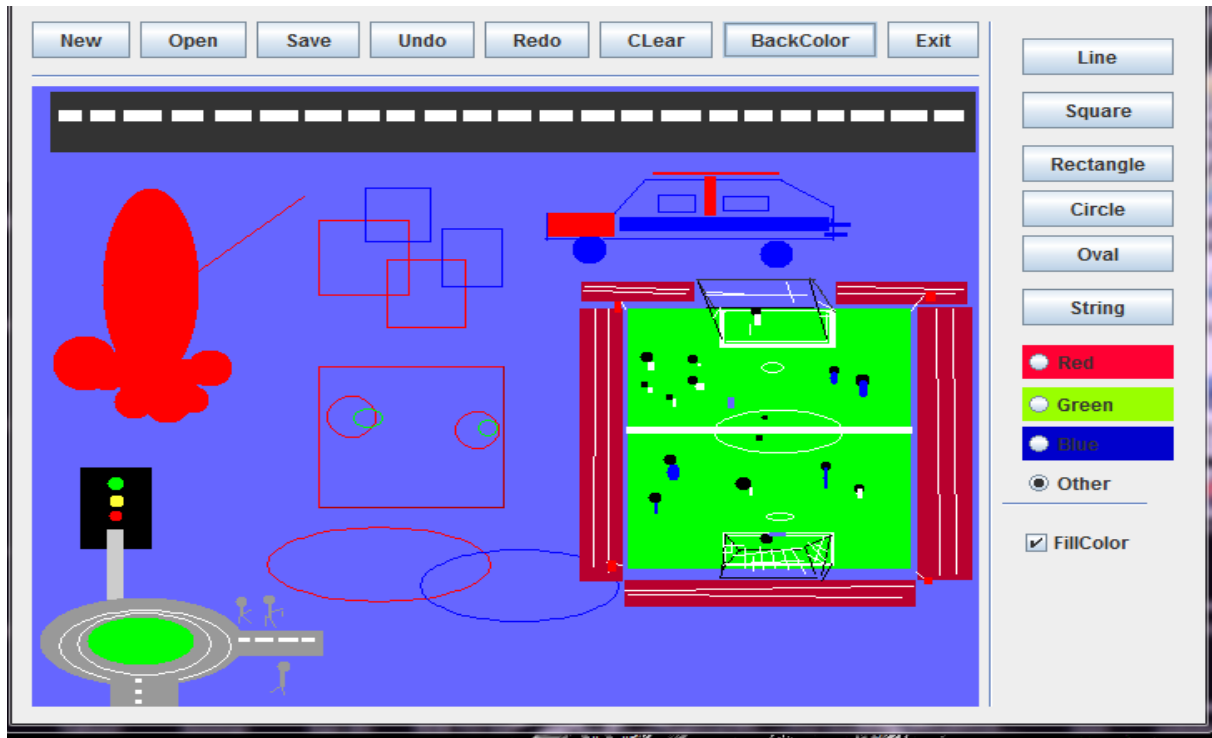
Set Color

Declaration
 public Color color=Color.RED;

```
private void btnRedFocusGained(java.awt.event.FocusEvent evt) {
    color=Color.RED;
}
private void btnGreenFocusGained(java.awt.event.FocusEvent evt) {
    color=Color.GREEN;
}
private void btnBlueFocusGained(java.awt.event.FocusEvent evt) {
    color=Color.BLUE;
}
```



4.4.4 setBackColor



4.4.5 Save to File

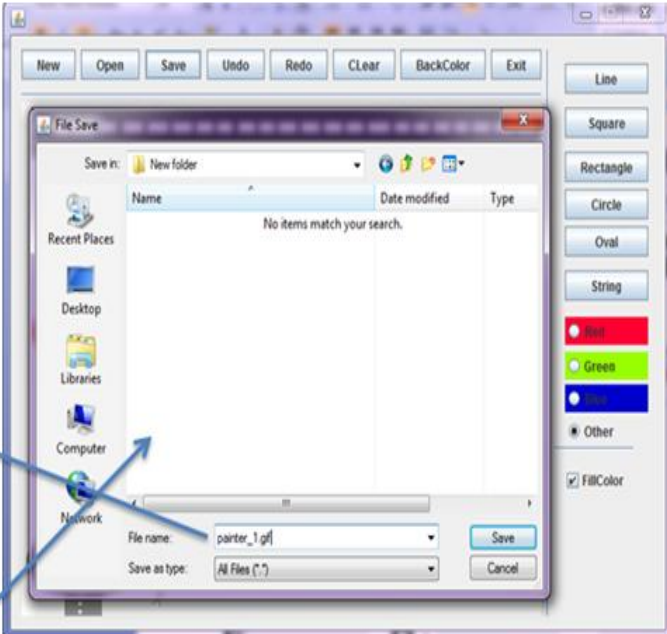
1

```
private void btnSaveMouseClicked(java.awt.event.MouseEvent evt) {  
    // TODO add your handling code here:  
    BufferedImage image = new BufferedImage(this.jPanel1.getWidth(),  
jPanel1.getHeight(), BufferedImage.TYPE_INT_RGB);  
    Graphics2D graphics2D = image.createGraphics();  
    String filename=writefile();  
    if (!filename.isEmpty()){  
  
        paint(graphics2D);  
        try{  
            if(filename.indexOf(".")!=-1){  
                filename+=".png";  
            }  
            // ImageIO.write(image,"png", new File("C:/Example.png"));  
            ImageIO.write(image,"png", new File(filename));  
            // ImageIO.write(image,"jpg", new File("C:/Example.jpg"));  
        }  
        catch(Exception ex){  
            ex.printStackTrace();  
        }  
    }  
}
```

2

```
String writefile(){  
    FileDialog fd=new FileDialog(new Frame(),"File  
Save",FileDialog.SAVE);  
    //fd.setFile("*.png;*.gif;*.jpg");  
    fd.show();  
    String fullpath=fd.getDirectory()+fd.getFile();  
    fd.dispose();  
    return fullpath;  
}
```

3



4

4.4.6 Open Image from File

1

```
private void btnOpenMouseClicked(java.awt.event.MouseEvent evt) {  
    // TODO add your handling code here:  
    //otherwaytoopenfile();  
  
    try{  
        String Filename = openfile();  
  
        if(Filename!=""){  
            this.ClearPanel();  
            this.setCommand(0);// for Draw Image in Jpanel  
            this.forImage=true;  
            image = ImageIO.read(new File(Filename));  
            paint(Area_Drawing);  
            this.forImage=true;  
        }  
    }catch(Exception ex){}
```

2

```
String openfile(){  
    String fullpath="";  
    FileDialog fd=new FileDialog(new Frame(),"File Select");  
    // fd.setFile("**.jpg;*.png;*.gif");  
    fd.show();  
    if (fd.getFile()!=null )  
        fullpath=fd.getDirectory()+fd.getFile();  
    fd.dispose();  
    return fullpath;  
}
```

3

4

5. Conclusion

This report presents an introduction to Java and how Java is used to build graphics and what are the tools that can be used to develop graphics and drawing required shapes.

This was an introduction to the main goal of our report that presented that is design and development a simple Painter project used to draw any shape (Circle, Line, Rectangle, Square, and Oval using FreeHand, Undo and Redo process, Clear JPanel, Set Background Color & set

Foreground Color, Save paint (Panel) to file (*. JPG; *. GIF; *.*), and Open paint from image file are considered. The system enables you to use Free Hand to draw (move the mouse using your hand to draw any shape and specify the coordinate in JPanel) as an easy way to draw the integrated paint, for example, a car , a street , a football stadium , traffic signals and others.

References

- [1] "Programming Language Popularity". 2009. Retrieved 2009-01-16.
- [2] "TIOBE Programming Community Index". 2009. Retrieved 2009-05-06
- [3] <http://www.csci.csusb.edu/dick/samples/java.html>, 2011, Thu Aug 25 21:04:36 PDT 2011
- [4] Monica Pawlan , Essentials of the Java Programming Language A Hands-On Guide
- [5] David J. Eck , Introduction to Programming Using Java , Version 6.0, June 2011 (Version 6.0.2, with minor corrections, May 2013)
- [6] Mads Rosendahl, Introduction to graphics programming in Java, February 13, 2009
- [7] 2D Graphics (The Java™ Tutorials) - Oracle Documentation,
<http://docs.oracle.com/javase/tutorial/2d/overview/index.html>, Copyright © 1995, 2013 Oracle
- [8] <http://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html> , 2013
- [9] <http://docs.oracle.com/javase/7/docs/api/java/awt/Graphics2D.html>, 2013
- [10] Oracle Technology, <http://www.oracle.com/technetwork/java/painting-140037.html#callback>