Government College of Engineering, Jalgaon
(An Autonomous Institute of Government of Maharashtra)

Name :	**PRN :**
Subject : CO310U (Application programming Lab)	**Sem :** V(Odd)
Class : T.Y. B.Tech	**Academic Year :** 2024-25
Date of Performance :	**Date of Completion :**

Practical No : 17

Aim: Write a program using a graphics method to draw an object ; provide direction buttons and move the object in the direction specified by the user through the button.

Required Software: OpenJDK version "1.8.0_131"
OpenJDK Runtime Environment (build 1.8.0_131-8u131-b11-2ubuntu1.16.04.3-b11)
OpenJDK 64-Bit Server VM (build 25.131-b11, mixed mode)

Java Compiler Version - JAVAC 1.8.0_131
Theory:

Graphics

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top-left corner and is 0,0.Coordinates are specified in pixels. All output to a window takes place through a graphics context. A*graphics context* is encapsulated by the **Graphics** class and is obtained in two ways:

• It is passed to an applet when one of its various methods, such as **paint( )** or **update( )**,

is called.

• It is returned by the getGraphics( ) method of Component.

The Graphics class defines a number of drawing functions. Each shape can be drawn edge only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped.
Let's look at several of the drawing methods.

<u>Drawing Lines</u>

Lines are drawn by means of the drawLine( ) method, shown here:
void drawLine(int startX, int startY, int endX, int endY)
drawLine( ) displays a line in the current drawing color that begins at startX,startY and ends at endX,endY.
The following applet draws several lines:

// Draw lines

```
import java.awt.*;
import java.applet.*;
/*
<applet code="Lines" width=300 height=200>
</applet>
*/
public class Lines extends Applet {
public void paint(Graphics g) {
g.drawLine(0, 0, 100, 100);
g.drawLine(0, 100, 100, 0);
g.drawLine(40, 25, 250, 180);
g.drawLine(75, 90, 400, 400);
g.drawLine(20, 150, 400, 40);
g.drawLine(5, 290, 80, 19);
}
}
```

Drawing Rectangles

The drawRect( ) and fillRect( ) methods display an outlined and filled rectangle, respectively.

They are shown here:

void drawRect(int top, int left, int width, int height)

void fillRect(int top, int left, int width, int height)

The upper-left corner of the rectangle is at top,left. The dimensions of the rectangle are specified

by width and height.

To draw a rounded rectangle, use drawRoundRect( ) or fillRoundRect( ), both shown here: void drawRoundRect(int top, int left, int width, int height, int xDiam, int yDiam) void fillRoundRect(int top, int left, int width, int height, int xDiam, int yDiam). A Rounded rectangle has rounded corners. The upper-left corner of the rectangle is at top,left. The dimensions of the rectangle are specified by width and height. The diameter of the rounding arc along the X axis is specified by xDiam. The diameter of the rounding arc along the Y axis

is specified by *yDiam.*

The following applet draws several rectangles:

```
 // Draw rectangles

import java.awt.*;

import java.applet.*;
```

```
/*

<applet code="Rectangles" width=300
height=200> </applet>

*/

public class Rectangles extends Applet

{

  public void paint(Graphics g)

{

  g.drawRect(10, 10, 60, 50);

  g.fillRect(100, 10, 60, 50);

 g.drawRoundRect(190, 10, 60, 50, 15,15);

 g.fillRoundRect(70, 90, 140, 100, 30, 40); }

}
```

Sample output from this program is shown here:

Drawing Ellipses and Circles

To draw an ellipse, use drawOval( ). To fill an ellipse, use fillOval( ). These methods are shown here:

void drawOval(int top, int left, int width, int height)

void fillOval(int top, int left, int width, int height)

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by top,left and whose width and height are specified by width and height. To draw a circle, specify a square as the bounding rectangle.

The following program draws several ellipses:

```
// Draw Ellipses

import java.awt.*;

import java.applet.*;

/*
```

```
<applet code="Ellipses" width=300 height=200>
        </applet>
*/
public class Ellipses extends Applet {
public void paint(Graphics g) {
g.drawOval(10, 10, 50, 50);
g.fillOval(100, 10, 75, 50);
g.drawOval(190, 10, 90, 30);
g.fillOval(70, 90, 140, 100);
}
}
```

Drawing Arcs

Arcs can be drawn with drawArc( ) and fillArc( ), shown here:

void drawArc(int top, int left, int width, int height, int startAngle, int sweepAngle) void fillArc(int top, int left, int width, int height, int startAngle, int sweepAngle)

The arc is bounded by the rectangle whose upper-left corner is specified by top,left and whose width and height are specified by width and height. The arc is drawn from startAngle through the angular distance specified by sweepAngle. Angles are specified in degrees. Zero degrees are on the horizontal, at the three o'clock position. The arc is drawn counterclockwise if sweepAngle is positive, and clockwise if sweepAngle is negative. Therefore, to draw an arc from twelve o'clock to six o'clock, the start angle would be 90 and the sweep angle 180. The following applet **draws several arcs:**

```
// Draw Arcs
import java.awt.*;
import java.applet.*;
/*
<applet code="Arcs" width=300 height=200>
</applet>
*/
public class Arcs extends Applet {
public void paint(Graphics g) {
g.drawArc(10, 40, 70, 70, 0, 75);
g.fillArc(100, 40, 70, 70, 0, 75);
g.drawArc(10, 100, 70, 80, 0, 175);
g.fillArc(100, 100, 70, 90, 0, 270);
g.drawArc(200, 80, 80, 80, 0, 180);
}
}
```

Working with Color

Java supports color in a portable, device-independent fashion. The AWT color system allows you to specify any color you want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet. Thus, your code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is encapsulated by the **Color** class.

You can also create your own colors, using one of the color constructors. Three commonly used forms are shown here:

Color(int *red*, int *green*, int *blue*)

Color(int rgbValue)

Color(float red, float green, float blue)

The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

```
new Color(255, 100, 100); // light red
```

The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);

Color darkRed = new Color(newRed);
```

The final constructor, Color(float, float, float), takes three float values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue.

Once you have created a color, you can use it to set the foreground and/or background color by using the setForeground( ) and setBackground( ) methods described in Chapter 21. You can also select it as the current drawing color.

Color Methods

The Color class defines several methods that help manipulate colors. They are examined here. Using Hue, Saturation, and Brightness .The hue-saturation-brightness (HSB) color model is an alternative to red-green-blue (RGB) for specifying particular colors. Figuratively, *hue* is a wheel of color. The hue is specified with a number between 0.0 and 1.0 (the colors are approximately red, orange, yellow, green, blue, indigo, and violet). *Saturation* is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. *Brightness* values also range from 0.0 to 1.0, where 1 is bright white and 0 is black. **Color** supplies two methods that let you convert between RGB and HSB. They are shown here:

static int HSBtoRGB(float hue, float saturation, float brightness)

static float[ ] RGBtoHSB(int red, int green, int blue, float values[ ])

HSBtoRGB( ) returns a packed RGB value compatible with the Color(int) constructor. RGBtoHSB( ) returns a float array of HSB values corresponding to RGB integers. If values is not null, then this array is given the HSB values and returned. Otherwise, a new array is created and the HSB values are returned in it. In either case, the array contains the hue at index 0, saturation at index 1, and brightness at index 2.

getRed( ), getGreen( ), getBlue( )

You can obtain the red, green, and blue components of a color independently using getRed( ), getGreen( ), and getBlue( ), shown here:

int getRed( )

int getGreen( )

int getBlue( )

Each of these methods returns the RGB color component found in the invoking Color object in the lower 8 bits of an integer.

getRGB( )

To obtain a packed, RGB representation of a color, use getRGB( ), shown here: int getRGB( )

The return value is organized as described earlier.

Setting the Current Graphics Color

By default, graphics objects are drawn in the current foreground color. You can change this color by calling the Graphics method setColor( ):

void setColor(Color newColor)

Here, newColor specifies the new drawing color.

You can obtain the current color by calling getColor( ), shown here:

Color getColor( )

Setting the Paint Mode

The paint mode determines how objects are drawn in a window. By default, new output to

a window overwrites any preexisting contents. However, it is possible to have new objects XORed onto the window by using setXORMode( ), as follows:

void setXORMode(Color xorColor)

Here, xorColor specifies the color that will be XORed to the window when an object is drawn.

The advantage of XOR mode is that the new object is always guaranteed to be visible no matter

what color the object is drawn over. To return to overwrite mode, call **setPaintMode( )**, shown here:

void setPaintMode( )

In general, you will want to use overwrite mode for normal output, and XOR mode for special purposes. For example, the following program displays cross hairs that track the mouse pointer. The cross hairs are XORed onto the window and are always visible, no matter what the underlying color is.

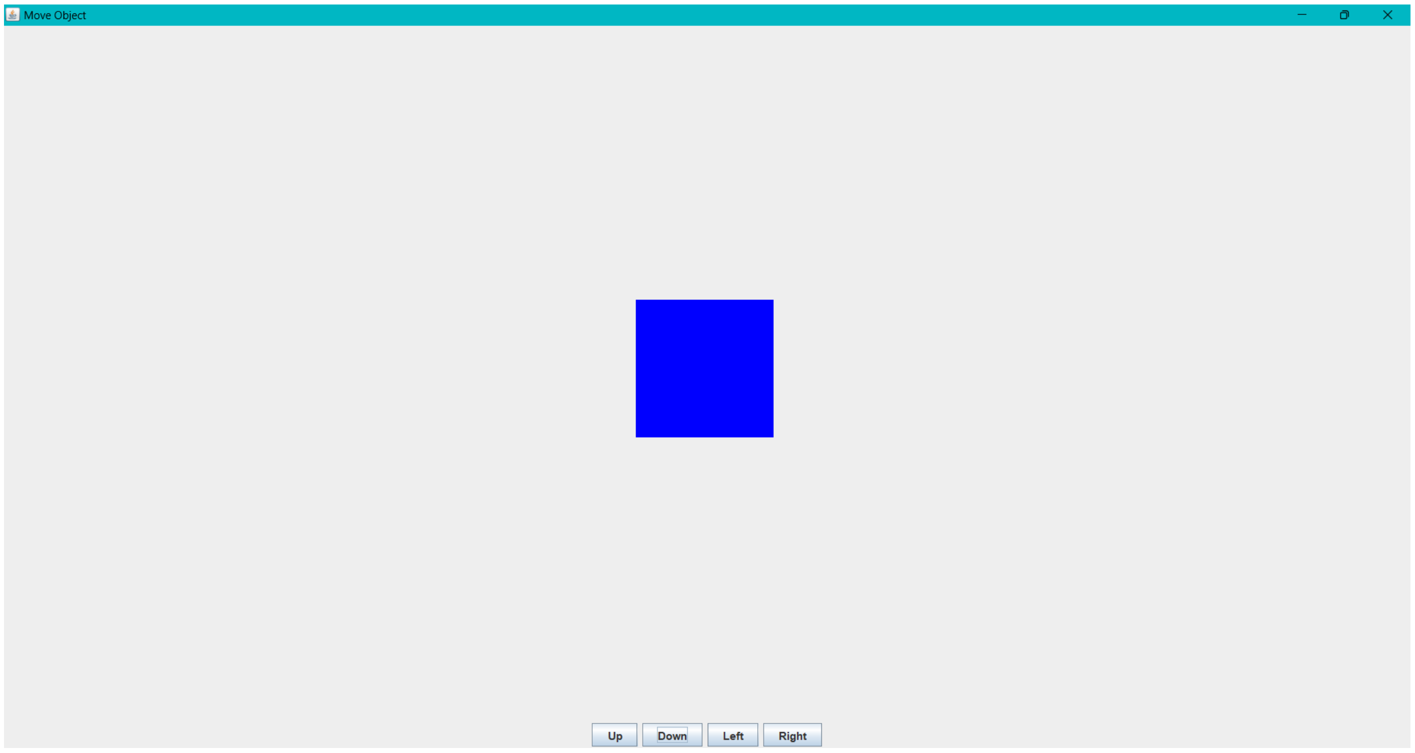
Conclusion:

Name & sign of Teacher

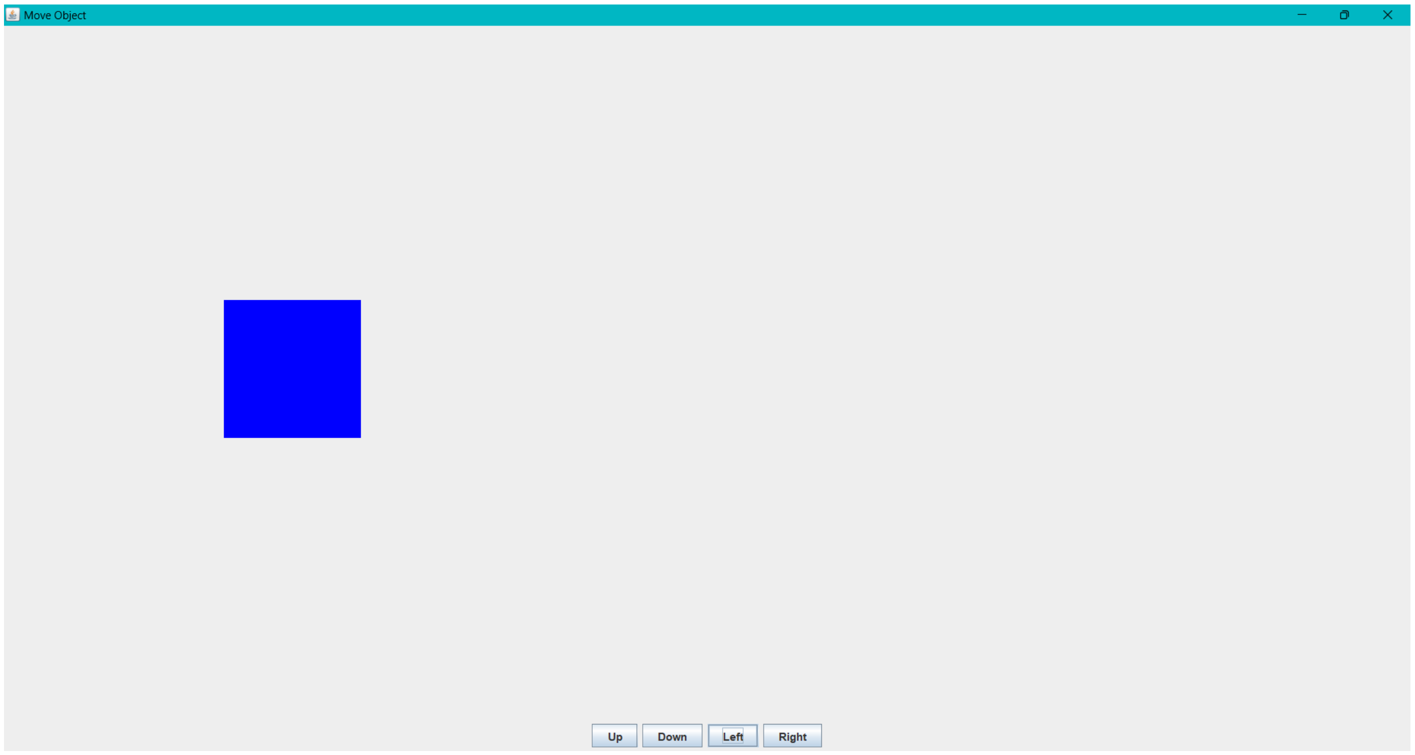
Program :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MoveObject extends JPanel implements ActionListener {
    private int x = 150, y = 150;          // Initial coordinates of the object
    private final int width = 150;         // Width of the square (150)
    private final int height = 150;        // Height of the square (150)
    private final int moveDistance = 30;   // Increased distance to move (30)

    // Constructor for setting up the GUI components
    public MoveObject() {
        // Creating buttons for each direction
        JButton upButton = new JButton("Up");
        JButton downButton = new JButton("Down");
        JButton leftButton = new JButton("Left");
        JButton rightButton = new JButton("Right");

        // Adding ActionListeners for each button
        upButton.addActionListener(this);
        downButton.addActionListener(this);
        leftButton.addActionListener(this);
        rightButton.addActionListener(this);

        // Adding buttons to a JPanel for layout organization
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(upButton);
        buttonPanel.add(downButton);
        buttonPanel.add(leftButton);
        buttonPanel.add(rightButton);

        // Setting up the main JFrame
        JFrame frame = new JFrame("Move Object");
        frame.setSize(600, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(this, BorderLayout.CENTER);  // Adding main drawing panel
        frame.add(buttonPanel, BorderLayout.SOUTH);  // Adding buttons at the bottom
        frame.setVisible(true);
    }

    // Overriding paintComponent to draw the object
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLUE);
        g.fillRect(x, y, width, height);  // Drawing the object as a scaled-up rectangle
    }

    // Handling button click actions
    @Override
    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();

        // Adjusting coordinates based on button clicked
        switch (command) {
            case "Up" -> y = Math.max(y - moveDistance, 0);
            case "Down" -> y = Math.min(y + moveDistance, getHeight() - height);
            case "Left" -> x = Math.max(x - moveDistance, 0);
            case "Right" -> x = Math.min(x + moveDistance, getWidth() - width);
        }

        // Repainting the panel to update the object's position
        repaint();
    }

    // Main method to run the program
    public static void main(String[] args) {
        SwingUtilities.invokeLater(MoveObject::new);
    }
}
```

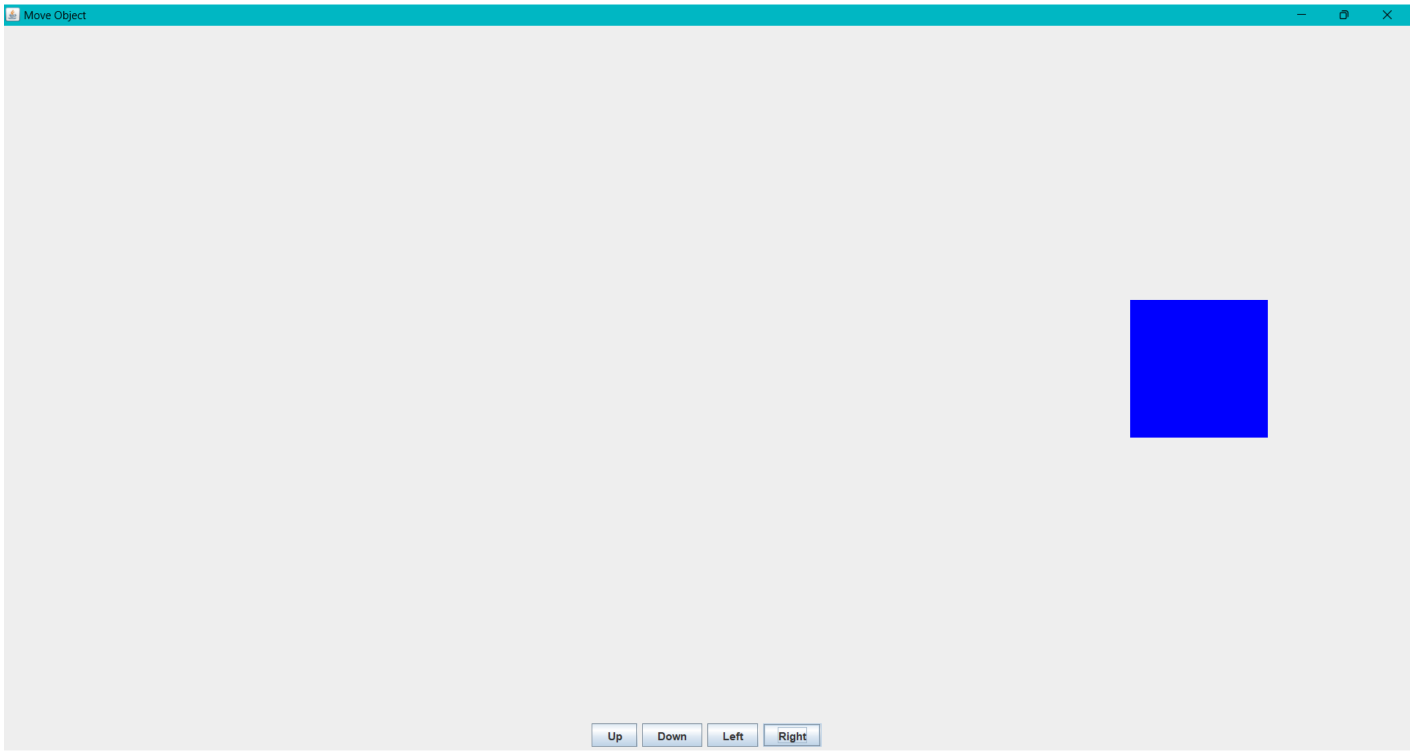
Output :



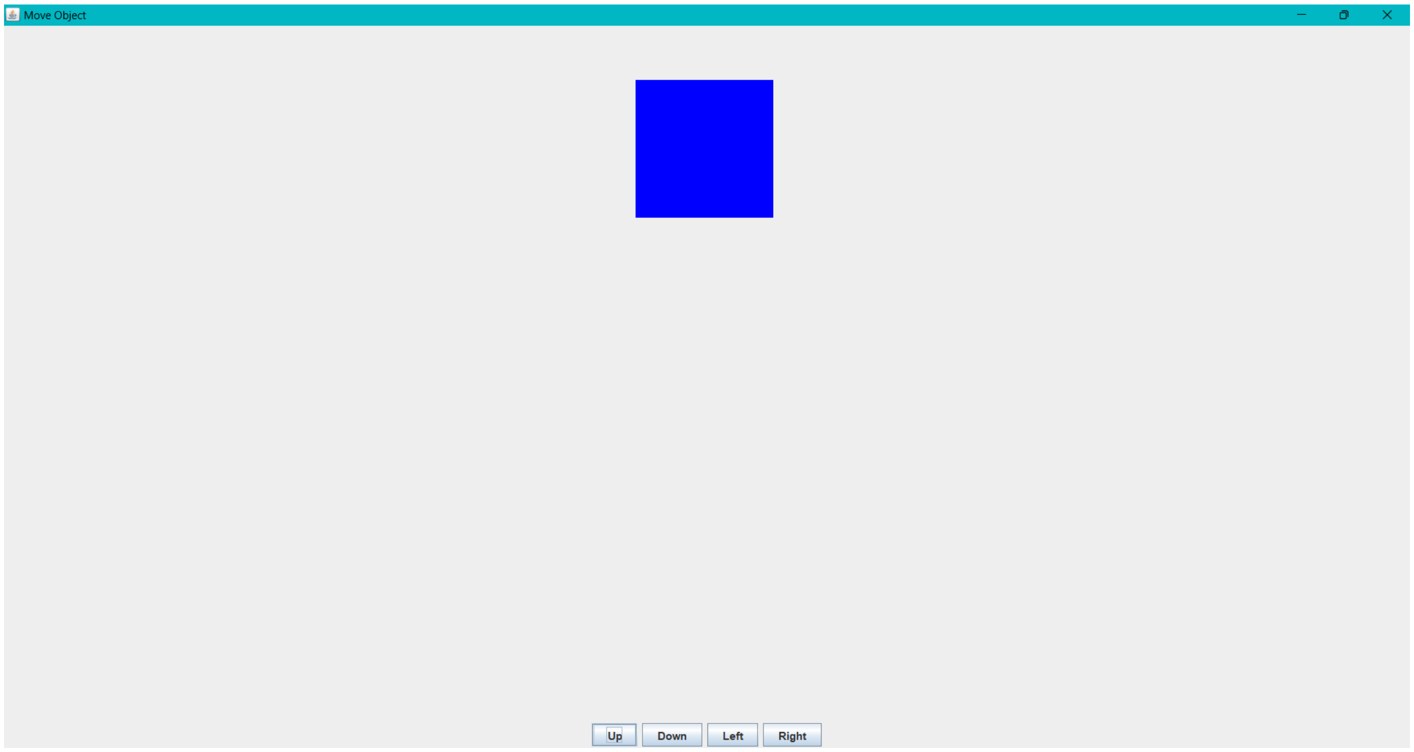
On Left click :



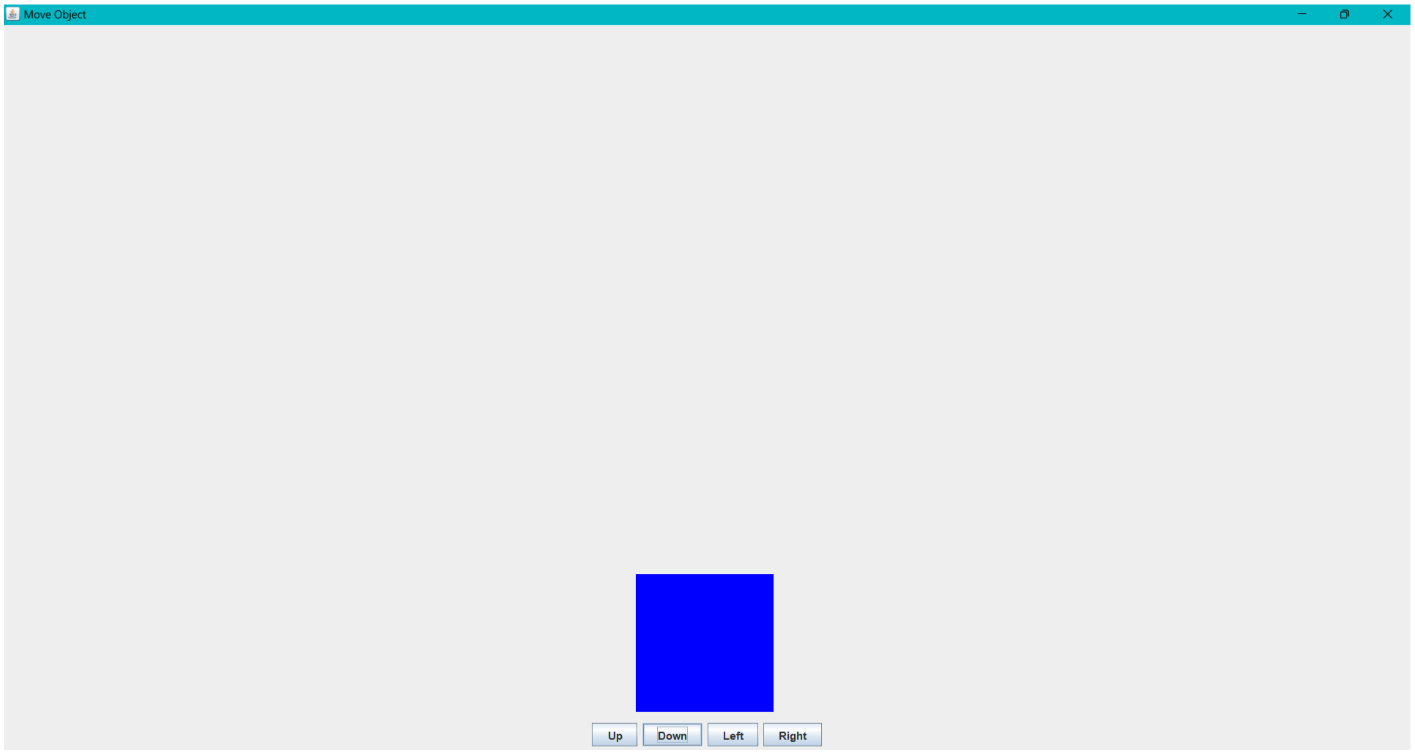
On Right click :



On Up click :



On Down click :


Move Object
Up
Down
Left
Right