



# CHAPTER 11

## Swing II

Shin-Jie Lee (李信杰)

Associate Professor

Computer and Network Center

Department of Computer Science and Information Engineering

National Cheng Kung University



# Window Listeners

- ❑ Clicking the close-window button on a **JFrame** fires a *window event*
  - Window events are objects of the class **WindowEvent**
- ❑ The **setWindowListener** method can register a window listener for a window event
  - A window listener can be programmed to respond to this type of event
  - A window listener is any class that satisfies the **WindowListener** interface



# Methods in the WindowListener Interface (Part 1 of 2)

## Methods in the WindowListener Interface

---

The WindowListener interface and the WindowEvent class are in the package `java.awt.event`.

```
public void windowOpened(WindowEvent e)
```

Invoked when a window has been opened.

```
public void windowClosing(WindowEvent e)
```

Invoked when a window is in the process of being closed. Clicking the close-window button causes an invocation of this method.

```
public void windowClosed(WindowEvent e)
```

Invoked when a window has been closed.

(continued)



# Methods in the WindowListener Interface (Part 2 of 2)

## Methods in the WindowListener Interface

---

```
public void windowIconified(WindowEvent e)
```

Invoked when a window is iconified. When you click the minimize button in a JFrame, it is iconified.

```
public void windowDeiconified(WindowEvent e)
```

Invoked when a window is deiconified. When you activate a minimized window, it is deiconified.

```
public void windowActivated(WindowEvent e)
```

Invoked when a window is activated. When you click in a window, it becomes the activated window. Other actions can also activate a window.

```
public void windowDeactivated(WindowEvent e)
```

Invoked when a window is deactivated. When a window is activated, all other windows are deactivated. Other actions can also deactivate a window.



# A Window Listener (Part 1 of 8)

## A Window Listener

---

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import java.awt.BorderLayout;
4  import java.awt.FlowLayout;
5  import java.awt.Color;
6  import javax.swing.JLabel;
7  import javax.swing.JButton;
8  import java.awt.event.ActionListener;
9  import java.awt.event.ActionEvent;
10 import java.awt.event.WindowListener;
11 import java.awt.event.WindowEvent;
```

(continued)



# A Window Listener (Part 2 of 8)

## A Window Listener

```
12 public class WindowListenerDemo extends JFrame
13 {
14     public static final int WIDTH = 300; //for main window
15     public static final int HEIGHT = 200; //for main window
16     public static final int SMALL_WIDTH = 200; //for confirm window
17     public static final int SMALL_HEIGHT = 100; //for confirm window

18     private class CheckOnExit implements WindowListener
19     {
20         public void windowOpened(WindowEvent e)
21         {}

22         public void windowClosing(WindowEvent e)
23         {
24             ConfirmWindow checkers = new ConfirmWindow();
25             checkers.setVisible(true);
26         }
27     }
28 }
```

*This WindowListener  
class is an inner class.*

(continued)



# A Window Listener (Part 3 of 8)

## A Window Listener

---

```
27     public void windowClosed(WindowEvent e)
28     {}

29     public void windowIconified(WindowEvent e)
30     {}

31     public void windowDeiconified(WindowEvent e)
32     {}

33     public void windowActivated(WindowEvent e)
34     {}
```

*A window listener must  
define all the method  
headings in the  
**WindowListener** interface,  
even if some are trivial  
implementations.*

(continued)



# A Window Listener (Part 4 of 8)

## A Window Listener

---

```
35     public void windowDeactivated(WindowEvent e)
36     {}
37 } //End of inner class CheckOnExit

38 private class ConfirmWindow extends JFrame implements ActionListener
39 {
40     public ConfirmWindow()
41     {
42         setSize(SMALL_WIDTH, SMALL_HEIGHT);
43         getContentPane().setBackground(Color.YELLOW);
44         setLayout(new BorderLayout());

45         JLabel confirmLabel = new JLabel(
46             "Are you sure you want to exit?");
47         add(confirmLabel, BorderLayout.CENTER);
```

*Another inner class.*

(continued)





# A Window Listener (Part 5 of 8)

## A Window Listener

---

```
48      JPanel buttonPanel = new JPanel();
49      buttonPanel.setBackground(Color.ORANGE);
50      buttonPanel.setLayout(new FlowLayout());

51      JButton exitButton = new JButton("Yes");
52      exitButton.addActionListener(this);
53      buttonPanel.add(exitButton);

54      JButton cancelButton = new JButton("No");
55      cancelButton.addActionListener(this);
56      buttonPanel.add(cancelButton);

57      add(buttonPanel, BorderLayout.SOUTH);
58  }
```

(continued)



# A Window Listener (Part 6 of 8)

## A Window Listener

---

```
59     public void actionPerformed(ActionEvent e)
60     {
61         String actionCommand = e.getActionCommand();

62         if (actionCommand.equals("Yes"))
63             System.exit(0);
64         else if (actionCommand.equals("No"))
65             dispose();//Destroys only the ConfirmWindow.
66         else
67             System.out.println("Unexpected Error in Confirm Window.");
68     }
69 } //End of inner class ConfirmWindow
```

(continued)




# A Window Listener (Part 7 of 8)

## A Window Listener

```
70
71     public static void main(String[] args)
72     {
73         WindowListenerDemo demoWindow = new WindowListenerDemo();
74         demoWindow.setVisible(true);
75     }
76
77     public WindowListenerDemo()
78     {
79         setSize(WIDTH, HEIGHT);
80         setTitle("Window Listener Demonstration");
81
82         setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
83         addWindowListener(new CheckOnExit());
84
85         getContentPane().setBackground(Color.LIGHT_GRAY);
86         JLabel aLabel = new JLabel("I like to be sure you are sincere.");
87         add(aLabel);
88     }
89 }
```

*Even if you have a window listener, you normally must still invoke setDefaultCloseOperation.*



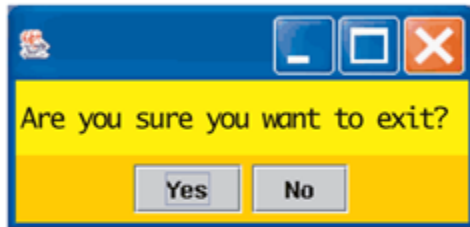
(continued)



# A Window Listener (Part 8 of 8)

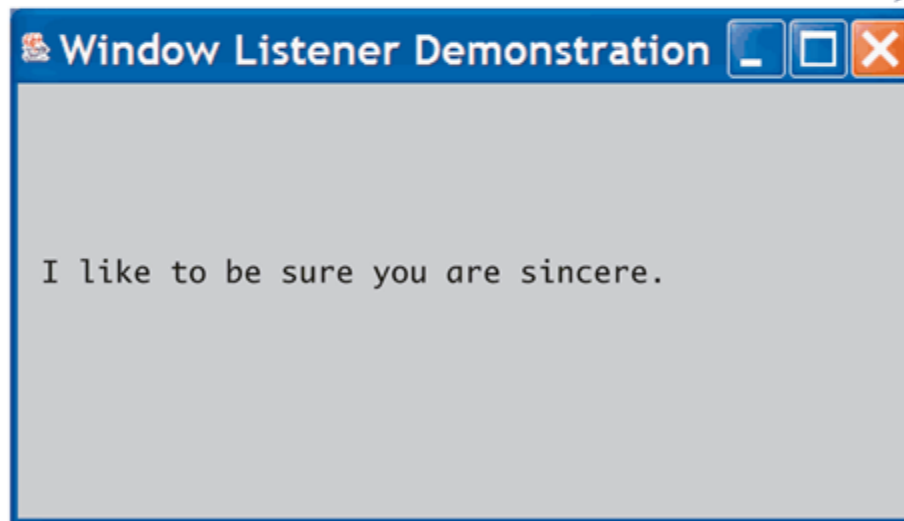
## A Window Listener

### RESULTING GUI



*This window is an object of the class `ConfirmWindow`.*

*When you click this close-window button, the second window appears.*



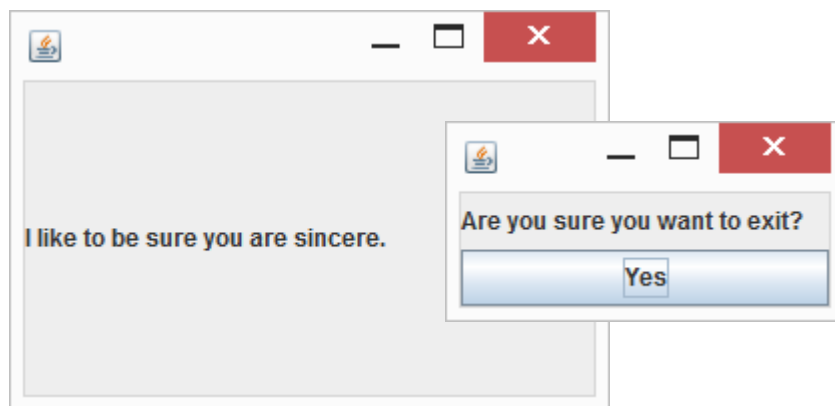


# Lab

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class WindowListenerDemo extends JFrame {
    public static void main(String[] args) {
        WindowListenerDemo demoWindow = new WindowListenerDemo();
        demoWindow.setVisible(true);
    }
    public WindowListenerDemo() {
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        addWindowListener(new CheckOnExit());
        JLabel aLabel = new JLabel("I like to be sure you are sincere.");
        add(aLabel);
    }
    private class CheckOnExit implements WindowListener {
        public void windowClosing(WindowEvent e) {
            ConfirmWindow checkers = new ConfirmWindow();
            checkers.setVisible(true);
        }
        public void windowOpened(WindowEvent e) {}
        public void windowClosed(WindowEvent e) {}
        public void windowIconified(WindowEvent e) {}
        public void windowDeiconified(WindowEvent e) {}
        public void windowActivated(WindowEvent e) {}
        public void windowDeactivated(WindowEvent e) {}
    }
    private class ConfirmWindow extends JFrame implements ActionListener {
        public ConfirmWindow() {
            setSize(200, 100);
            setLayout(new BorderLayout());
            JLabel confirmLabel = new JLabel("Are you sure you want to exit?");
            add(confirmLabel, BorderLayout.CENTER);
            JButton exitButton = new JButton("Yes");
            exitButton.addActionListener(this);
            add(exitButton, BorderLayout.SOUTH);
        }
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    }
}
```



# Lab





# Icons

- ❑ **JLabels**, **JButtons**, and **JMenuItems** can have icons
  - An *icon* is just a small picture (usually)
  - It is not required to be small
- ❑ An icon is an object of the **ImageIcon** class
  - It is based on a digital picture file such as **.gif**, **.jpg**, or **.tiff**



# Icons

- ❑ The class **ImageIcon** is used to convert a picture file to a Swing icon

```
ImageIcon dukeIcon = new  
    ImageIcon( "duke_waving.gif" );
```

- The picture file must be in the same directory as the class in which this code appears, unless a complete or relative path name is given
- Note that the name of the picture file is given as a string





# Icons

- ❑ An icon can be added to a label using the **setIcon** method as follows:

```
JLabel dukeLabel = new JLabel("Mood check");  
dukeLabel.setIcon(dukeIcon);
```

- ❑ Instead, an icon can be given as an argument to the **JLabel** constructor:

```
JLabel dukeLabel = new JLabel(dukeIcon);
```

- ❑ Text can be added to the label as well using the **setText** method:

```
dukeLabel.setText("Mood check");
```



# Lab

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

public class IconDemo extends JFrame implements ActionListener{

    public static void main(String[] args) {
        IconDemo frame = new IconDemo();
        frame.setVisible(true);
    }

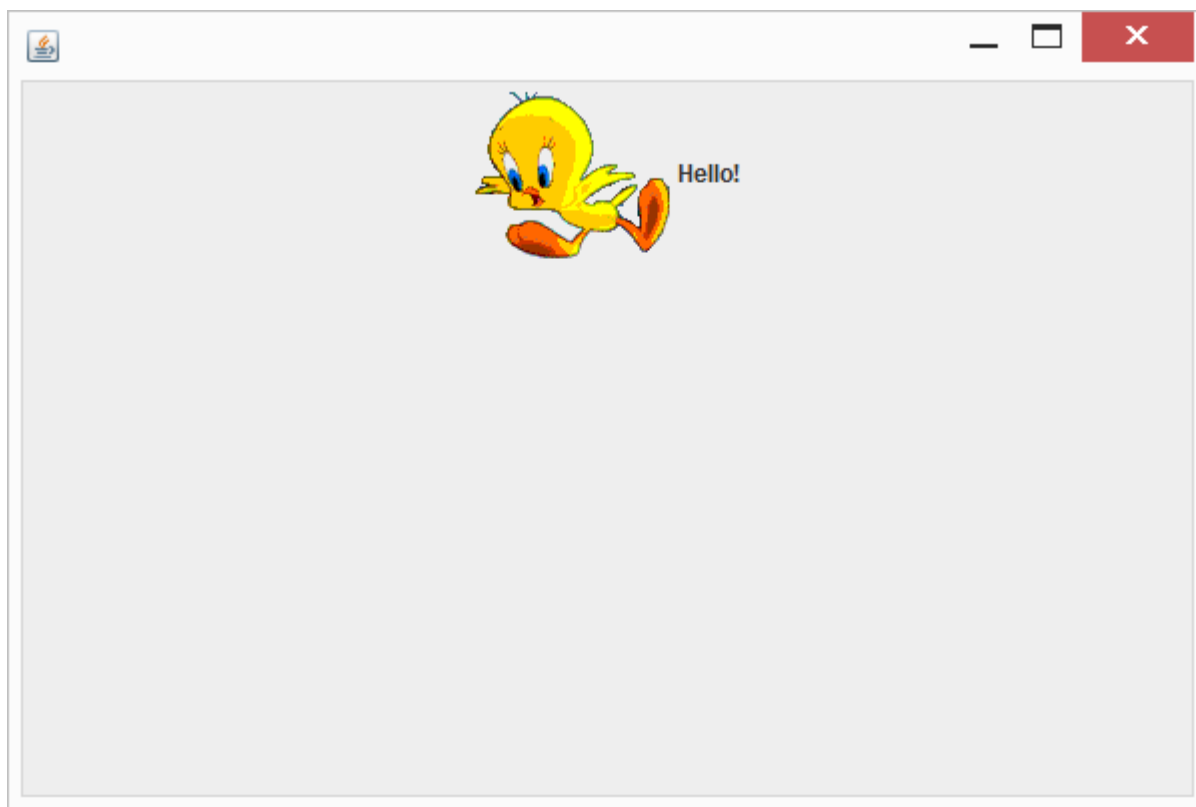
    public IconDemo(){
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        try {
            ImageIcon icon = new ImageIcon(new URL("http://www.smes.tyc.edu.tw/~learn/images/1015.gif"));
            JLabel lb = new JLabel(icon);
            lb.setText("Hello!");
            add(lb);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void actionPerformed(ActionEvent e){
    }
}
```



# Lab





# Icons

- ❑ Icons and text may be added to **JButtons** and **JMenuItems** in the same way as they are added to a **JLabel**

```
JButton happyButton = new  
                                JButton( "Happy" );  
ImageIcon happyIcon = new  
                                ImageIcon( "smiley.gif" );  
happyButton.setIcon(happyIcon);
```



# Icons

- ❑ Button or menu items can be created with just an icon by giving the **ImageIcon** object as an argument to the **JButton** or **JMenuItem** constructor

```
ImageIcon happyIcon = new  
                                ImageIcon("smiley.gif");  
JButton smileButton = new JButton(happyIcon);  
JMenuItem happyChoice = new  
                                JMenuItem(happyIcon);
```

- A button or menu item created without text should use the **setActionCommand** method to explicitly set the action command, since there is no string



# Lab

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

public class IconDemo extends JFrame implements ActionListener{
    public static void main(String[] args) {
        IconDemo frame = new IconDemo();
        frame.setVisible(true);
    }

    public IconDemo(){
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        try {
            ImageIcon icon = new ImageIcon(new URL("http://www.smes.tyc.edu.tw/~learn/images/1015.gif"));
            JLabel lb = new JLabel(icon);
            lb.setText("Hello!");
            add(lb);

            ImageIcon icon2 = new ImageIcon(new URL("http://icons.iconarchive.com/icons/tatice/cristal-intense/32/ok-icon.png"));
            JButton btn = new JButton(icon2);
            btn.setText("Ok!");
            add(btn);

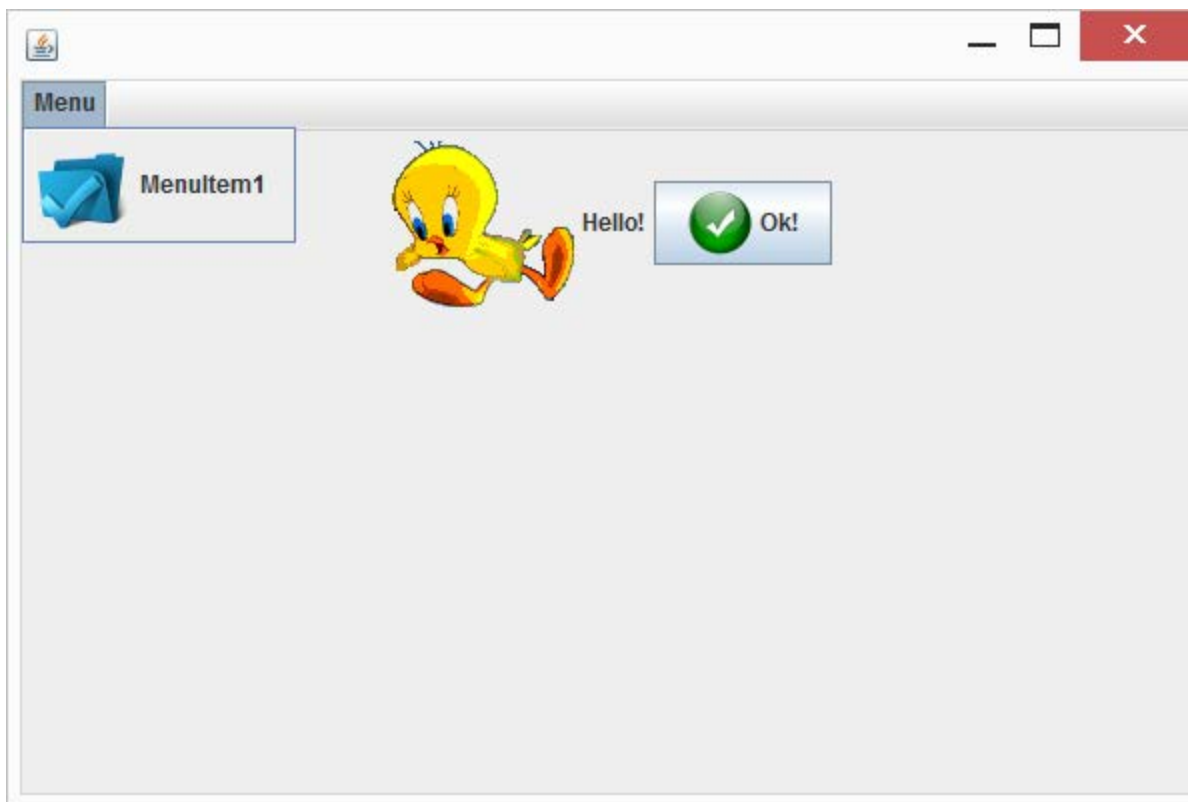
            ImageIcon icon3 = new ImageIcon(new URL("http://icons.iconarchive.com/icons/toma4025/rumax/48/folder-ok-icon.png"));
            JMenu diner = new JMenu("Menu");
            JMenuItem item1 = new JMenuItem("MenuItem1");
            item1.setIcon(icon3);
            diner.add(item1);
            JMenuBar bar = new JMenuBar();
            bar.add(diner);

            setJMenuBar (bar);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void actionPerformed(ActionEvent e){}
}
```



# Lab





# Scroll Bars

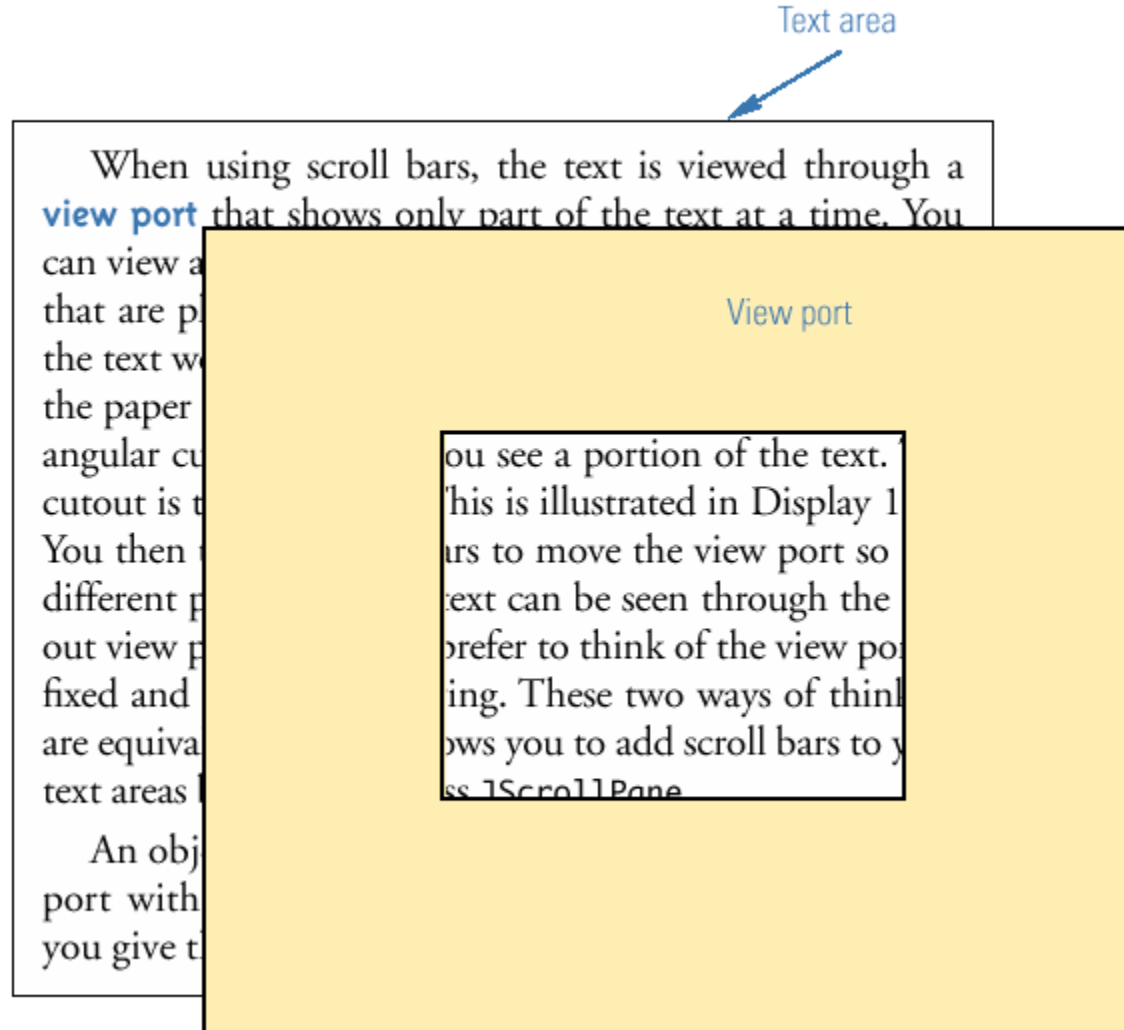
- ❑ When using scroll bars, the text is viewed through a *view port* that shows only part of the text at a time
  - A different part of the text may be viewed by using the scroll bars placed along the side and bottom of the view port
- ❑ Scroll bars can be added to text areas using the **JScrollPane** class
  - The **JScrollPane** class is in the **javax.swing** package
  - An object of the class **JScrollPane** is like a view port with scroll bars





# View Port for a Text Area

## View Port for a Text Area





# Scroll Bars

- ❑ When a **JScrollPane** is created, the text area to be viewed is given as an argument

```
JTextArea memoDisplay = new  
    JTextArea(15, 30);  
JScrollPane scrolledText = new  
    JScrollPane(memoDisplay);
```

- ❑ The **JScrollPane** can then be added to a container, such as a **JPanel** or **JFrame**

```
textPanel.add(scrolledText);
```



# Scroll Bars

- ❑ The scroll bar policies can be set as follows:

```
scrolledText.setHorizontalScrollBarPolicy(  
    JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);  
scrolledText.setVerticalScrollBarPolicy(  
    JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
```

- ❑ If invocations of these methods are omitted, then the scroll bars will be visible only when needed
  - If all the text fits in the view port, then no scroll bars will be visible
  - If enough text is added, the scroll bars will appear automatically



# Lab

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

public class ScrollBarTest extends JFrame implements ActionListener{

    public static void main(String[] args) {
        ScrollBarTest frame = new ScrollBarTest();
        frame.setVisible(true);
    }

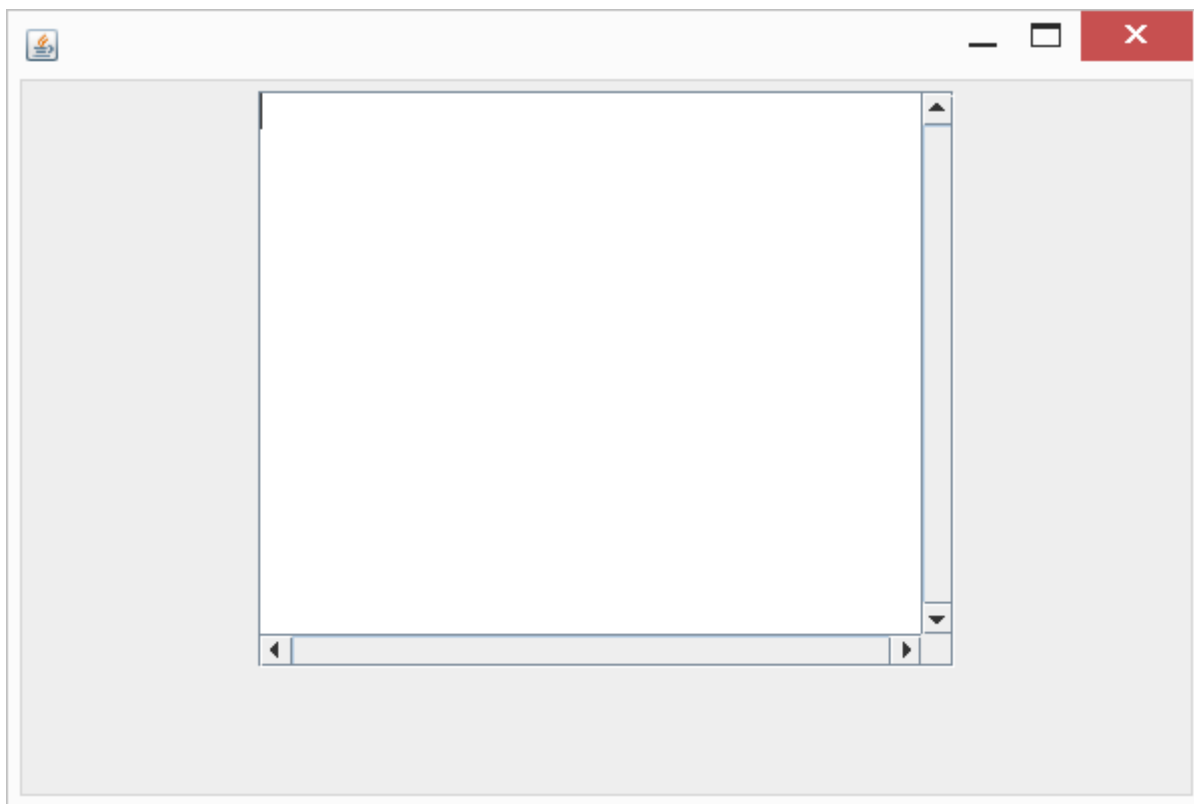
    public ScrollBarTest(){
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        JTextArea memoDisplay = new JTextArea(15, 30);
        JScrollPane scrolledText = new JScrollPane(memoDisplay);
        scrolledText.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
        scrolledText.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
        add(scrolledText);
    }

    public void actionPerformed(ActionEvent e){
    }
}
```



# Lab





# Components with Changing Visibility

---

- ☐ A GUI can have components that change from visible to invisible and back again
- ☐ A component can be made invisible without making the entire GUI invisible



# Labels with Changing Visibility (Part 1 of 6)

## Labels with Changing Visibility

---

```
1  import javax.swing.JFrame;
2  import javax.swing.ImageIcon;
3  import javax.swing.JPanel;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import java.awt.BorderLayout;
7  import java.awt.FlowLayout;
8  import java.awt.Color;
9  import java.awt.event.ActionListener;
10 import java.awt.event.ActionEvent;

11 public class VisibilityDemo extends JFrame
12     implements ActionListener
13 {
14     public static final int WIDTH = 300;
15     public static final int HEIGHT = 200;
```

(continued)



# Labels with Changing Visibility (Part 2 of 6)

## Labels with Changing Visibility

---

```
16     private JLabel wavingLabel;
17     private JLabel standingLabel;
18     public static void main(String[] args)
19     {
20         VisibilityDemo demoGui = new VisibilityDemo();
21         demoGui.setVisible(true);
22     }

23     public VisibilityDemo()
24     {
25         setSize(WIDTH, HEIGHT);
26         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27         setTitle("Visibility Demonstration");

28         setLayout(new BorderLayout());
```

(continued)





# Labels with Changing Visibility (Part 3 of 6)

## Labels with Changing Visibility

---

```
29    JPanel picturePanel = new JPanel();
30    picturePanel.setBackground(Color.WHITE);
31    picturePanel.setLayout(new FlowLayout());
```

```
32    ImageIcon dukeStandingIcon =
33        new ImageIcon("duke_standing.gif");
34    standingLabel = new JLabel(dukeStandingIcon);
35    standingLabel.setVisible(true);
36    picturePanel.add(standingLabel);
```

```
37    ImageIcon dukeWavingIcon = new ImageIcon("duke_waving.gif");
38    wavingLabel = new JLabel(dukeWavingIcon);
39    wavingLabel.setVisible(false);
40    picturePanel.add(wavingLabel);
```

(continued)



# Labels with Changing Visibility (Part 4 of 6)

## Labels with Changing Visibility

---

```
41      add(picturePanel, BorderLayout.CENTER);

42      JPanel buttonPanel = new JPanel();
43      buttonPanel.setBackground(Color.LIGHT_GRAY);
44      buttonPanel.setLayout(new FlowLayout());

45      JButton waveButton = new JButton("Wave");
46      waveButton.addActionListener(this);
47      buttonPanel.add(waveButton);

48      JButton stopButton = new JButton("Stop");
49      stopButton.addActionListener(this);
50      buttonPanel.add(stopButton);
```

(continued)



# Labels with Changing Visibility (Part 5 of 6)

## Labels with Changing Visibility

---

```
51         add(buttonPanel, BorderLayout.SOUTH);
52     }
53     public void actionPerformed(ActionEvent e)
54     {
55         String actionCommand = e.getActionCommand();

56         if (actionCommand.equals("Wave"))
57         {
58             wavingLabel.setVisible(true);
59             standingLabel.setVisible(false);
60         }
61         else if (actionCommand.equals("Stop"))
62         {
63             standingLabel.setVisible(true);
64             wavingLabel.setVisible(false);
65         }
66         else
67             System.out.println("Unanticipated error.");
68     }
69 }
```

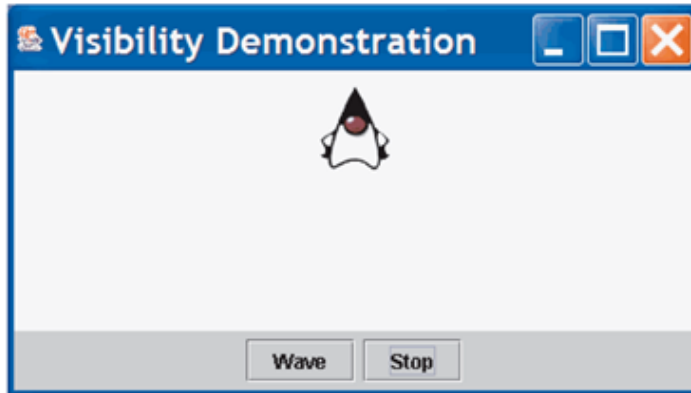
(continued)



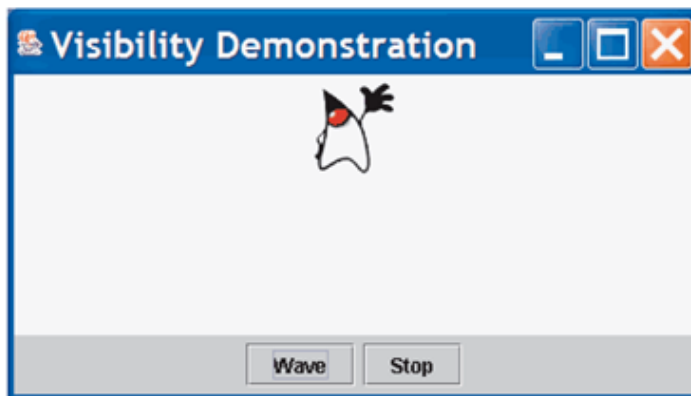
# Labels with Changing Visibility (Part 6 of 6)

## Labels with Changing Visibility

**RESULTING GUI** (After clicking Stop button)



**RESULTING GUI** (After clicking Wave button)





# Lab

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

public class ScrollBarTest extends JFrame implements ActionListener{

    public static void main(String[] args) {
        ScrollBarTest frame = new ScrollBarTest();
        frame.setVisible(true);
    }

    public ScrollBarTest(){
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        JTextArea memoDisplay = new JTextArea(15, 30);
        JScrollPane scrolledText = new JScrollPane(memoDisplay);
        scrolledText.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
        scrolledText.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
        add(scrolledText);
        scrolledText.setVisible(false);
    }

    public void actionPerformed(ActionEvent e){
    }
}
```



# Coordinate System for Graphics Objects

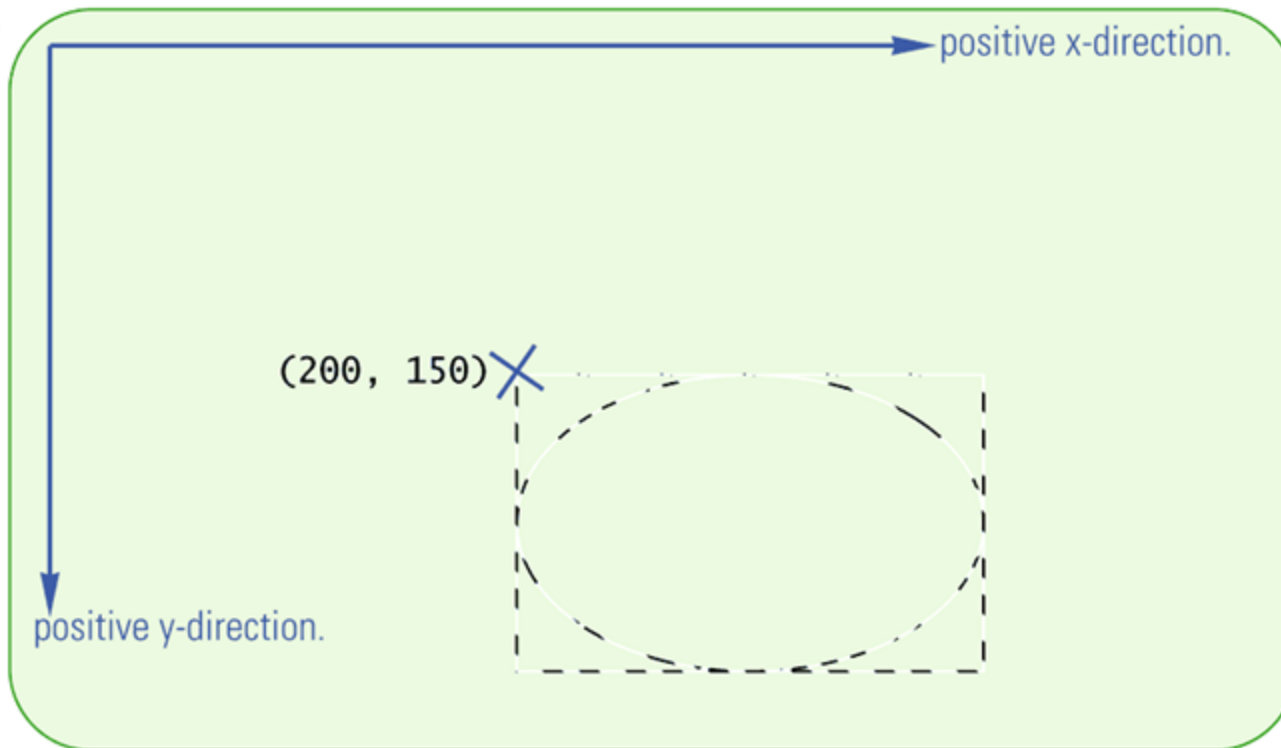
- ❑ When drawing objects on the screen, Java uses a coordinate system where the origin point (0,0) is at the upper-left corner of the screen area used for drawing
  - The x-coordinate (horizontal) is positive and increasing to the right
  - The y- coordinate(vertical) is positive and increasing down
  - All coordinates are normally positive
  - Units and sizes are in pixels
  - The area used for drawing is typically a **JFrame** or **JPanel**



# Screen Coordinate System

## Screen Coordinate System

$(0, 0)$





# The Method `paint` and the Class `Graphics`

- ❑ Almost all Swing and Swing-related components and containers have a method called `paint`
- ❑ The method `paint` draws the component or container on the screen
  - It is already defined, and is called automatically when the figure is displayed on the screen
  - However, it must be redefined in order to draw geometric figures like circles and boxes
  - When redefined, always include the following:  
`super.paint(g);`





# The Method `paint` and the Class `Graphics`

- ❑ Every container and component that can be drawn on the screen has an associated `Graphics` object
  - The `Graphics` class is an abstract class found in the `java.awt` package
  - The method `paint` has a parameter `g` of type `Graphics`



# Drawing a Very Simple Face (part 1 of 5)

## Drawing a Very Simple Face

---

```
1  import javax.swing.JFrame;
2  import java.awt.Graphics;
3  import java.awt.Color;

4  public class Face extends JFrame
5  {
6      public static final int WINDOW_WIDTH = 400;
7      public static final int WINDOW_HEIGHT = 400;

8      public static final int FACE_DIAMETER = 200;
9      public static final int X_FACE = 100;
10     public static final int Y_FACE = 100;
```

(continued)



# Drawing a Very Simple Face (part 2 of 5)

## Drawing a Very Simple Face

---

```
11     public static final int EYE_WIDTH = 20;
12     public static final int X_RIGHT_EYE = X_FACE + 55;
13     public static final int Y_RIGHT_EYE = Y_FACE + 60;
14     public static final int X_LEFT_EYE = X_FACE + 130;
15     public static final int Y_LEFT_EYE = Y_FACE + 60;

16     public static final int MOUTH_WIDTH = 100;
17     public static final int X_MOUTH = X_FACE + 50;
18     public static final int Y_MOUTH = Y_FACE + 150;
```

(continued)



# Drawing a Very Simple Face (part 3 of 5)

## Drawing a Very Simple Face

---

```
19     public static void main(String[] args)
20     {
21         Face drawing = new Face();
22         drawing.setVisible(true);
23     }

24     public Face()
25     {
26         super("First Graphics Demo");
27         setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
28         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29         getContentPane().setBackground(Color.white);
30     }
```

(continued)



# Drawing a Very Simple Face (part 4 of 5)

## Drawing a Very Simple Face

---

```
31 public void paint(Graphics g)
32 {
33     super.paint(g);
34     g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
35     //Draw Eyes:
36     g.drawLine(X_RIGHT_EYE, Y_RIGHT_EYE,
37               X_RIGHT_EYE + EYE_WIDTH, Y_RIGHT_EYE);
38     g.drawLine(X_LEFT_EYE, Y_LEFT_EYE,
39               X_LEFT_EYE + EYE_WIDTH, Y_LEFT_EYE);
40     //Draw Mouth:
41     g.drawLine(X_MOUTH, Y_MOUTH, X_MOUTH + MOUTH_WIDTH, Y_MOUTH);
42 }
43 }
```

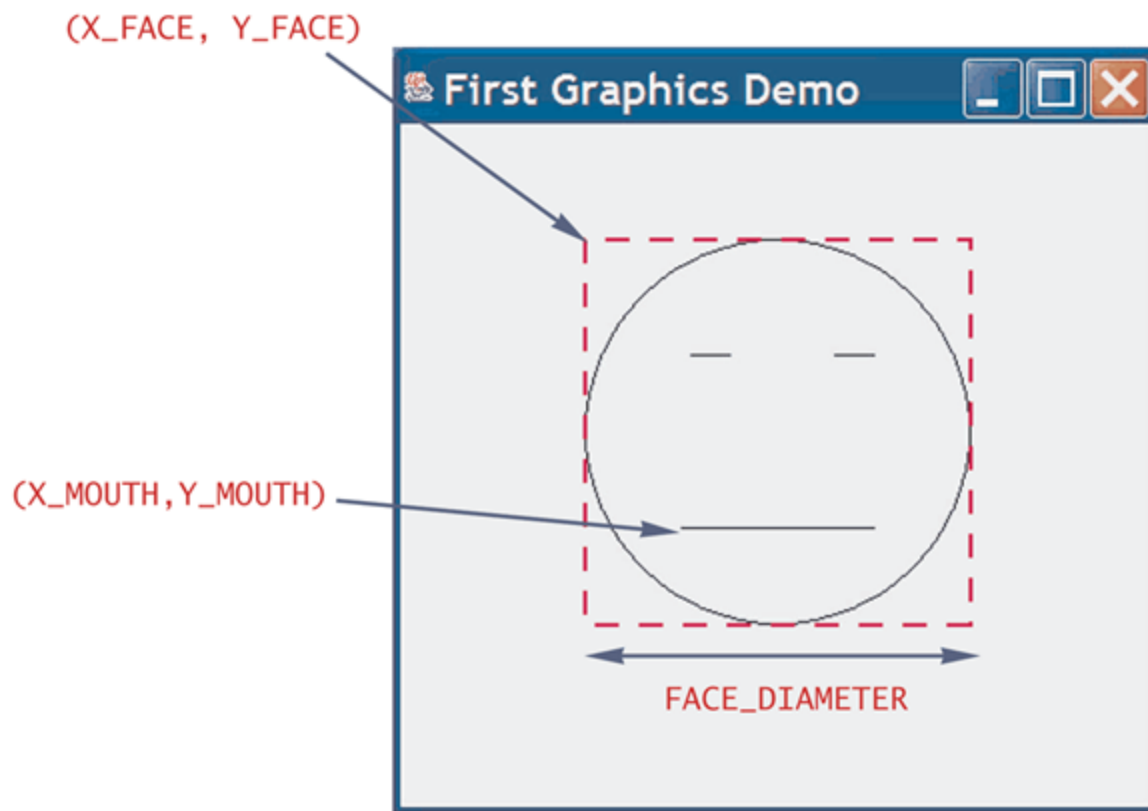
(continued)



# Drawing a Very Simple Face (part 5 of 5)

## Drawing a Very Simple Face

### RESULTING GUI



The red box is not shown on the screen. It is there to help you understand the relationship between the **paint** method code and the resulting drawing.



# Lab

```
import java.awt.FlowLayout;
import java.awt.Graphics;
import javax.swing.JFrame;

public class PaintTest extends JFrame{

    public static void main(String[] args) {
        PaintTest frame = new PaintTest();
        frame.setVisible(true);
    }

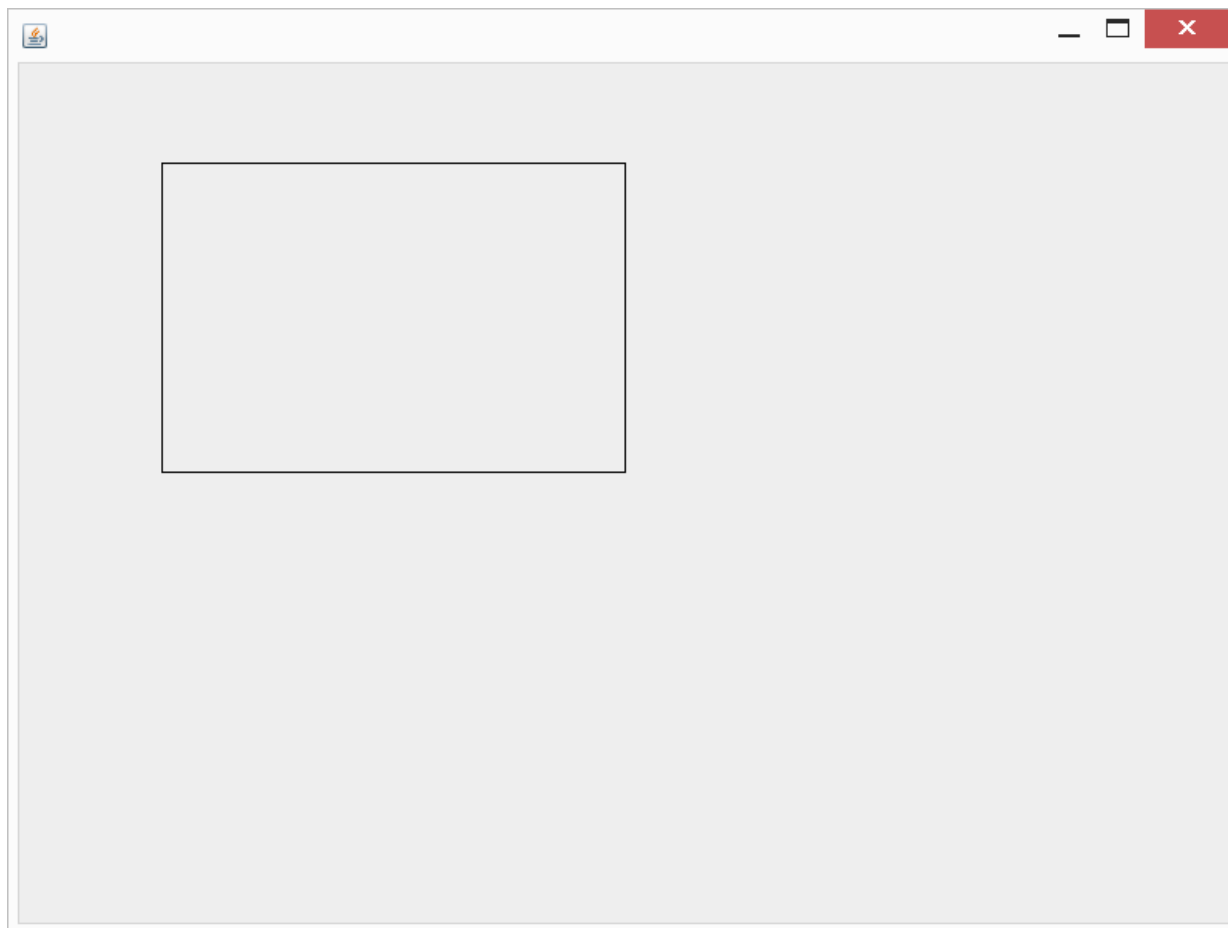
    public PaintTest(){
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
    }

    public void paint(Graphics g){
        super.paint(g);
        g.drawRect(100, 100, 300, 200);
    }

}
```



# Lab







# Some Methods in the Class Graphics (part 1 of 4)

## Some Methods in the Class Graphics

---

Graphics is an abstract class in the `java.awt` package.

Although many of these methods are abstract, we always use them with objects of a concrete descendent class of Graphics, even though we usually do not know the name of that concrete class.

```
public abstract void drawLine(int x1, int y1, int x2, int y2)
```

Draws a line between points (x1, y1) and (x2, y2).

```
public abstract void drawRect(int x, int y,  
                             int width, int height)
```

Draws the outline of the specified rectangle. (x, y) is the location of the upper-left corner of the rectangle.

```
public abstract void fillRect(int x, int y,  
                             int width, int height)
```

Fills the specified rectangle. (x, y) is the location of the upper-left corner of the rectangle.

(continued)



# Some Methods in the Class Graphics (part 2 of 4)

## Some Methods in the Class Graphics

---

```
public void draw3DRect(int x, int y, int width,  
                      int height, boolean raised)
```

Draws the outline of the specified rectangle. (x, y) is the location of the upper-left corner. The rectangle is highlighted to look like it has thickness. If raised is true, the highlight makes the rectangle appear to stand out from the background. If raised is false, the highlight makes the rectangle appear to be sunken into the background.

```
public void fill3DRect(int x, int y, int width,  
                     int height, boolean raised)
```

Fills the rectangle specified by

```
draw3DRect(x, y, width, height, raised)
```

(continued)



# Some Methods in the Class Graphics

## (part 3 of 4)

### Some Methods in the Class Graphics

---

```
public abstract void drawRoundRect(int x, int y,  
                                   int width, int height, int arcWidth, int arcHeight)
```

Draws the outline of the specified round-cornered rectangle. (x, y) is the location of the upper-left corner of the enclosing regular rectangle. arcWidth and arcHeight specify the shape of the round corners. See the text for details.

```
public abstract void fillRoundRect(int x, int y,  
                                   int width, int height, int arcWidth, int arcHeight)
```

Fills the rounded rectangle specified by

```
drawRoundRect(x, y, width, height, arcWidth, arcHeight)
```

```
public abstract void drawOval(int x, int y,  
                              int width, int height)
```

Draws the outline of the oval with the smallest enclosing rectangle that has the specified width and height. The (imagined) rectangle has its upper-left corner located at (x, y).

(continued)



# Some Methods in the Class Graphics

## (part 4 of 4)

### Some Methods in the Class Graphics

---

```
public abstract void fillOval(int x, int y,  
                             int width, int height)
```

Fills the oval specified by

```
drawOval(x, y, width, height)
```

```
public abstract void drawArc(int x, int y,  
                             int width, int height,  
                             int startAngle, int arcSweep)
```

Draws part of an oval that just fits into an invisible rectangle described by the first four arguments. The portion of the oval drawn is given by the last two arguments. See the text for details.

```
public abstract void fillArc(int x, int y,  
                             int width, int height,  
                             int startAngle, int arcSweep)
```

Fills the partial oval specified by

```
drawArc(x, y, width, height, startAngle, arcSweep)
```



# Drawing Ovals

- ❑ An oval is drawn by the method **drawOval**
  - The arguments specify the location, width, and height of the smallest rectangle that can enclose the oval

```
g.drawOval(100, 50, 300, 200);
```

- ❑ A circle is a special case of an oval in which the width and height of the rectangle are equal

```
g.drawOval(X_FACE, Y_FACE,  
           FACE_DIAMETER, FACE_DIAMETER);
```



# Lab

```
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Graphics;
import javax.swing.JFrame;

public class PaintTest extends JFrame{

    public static void main(String[] args) {
        PaintTest frame = new PaintTest();
        frame.setVisible(true);
    }

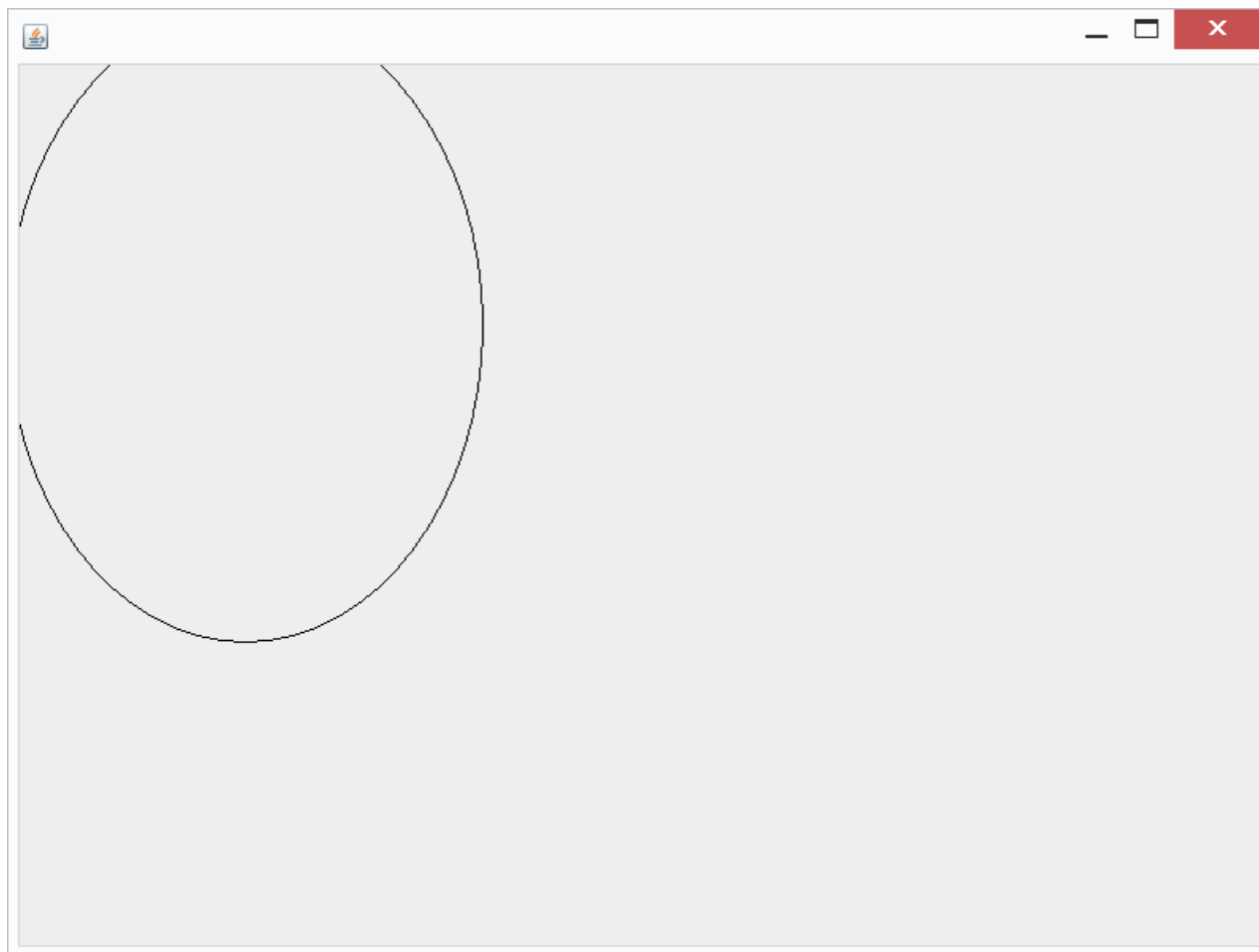
    public PaintTest(){
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
    }

    public void paint(Graphics g){
        super.paint(g);
        g.drawOval(0, 0, 300, 400);
    }

}
```



# Lab





# Drawing Arcs

- ❑ Arcs are described by giving an oval, and then specifying a portion of it to be used for the arc
  - The following statement draws the smile on the happy face:

```
g.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH,  
          MOUTH_HEIGHT, MOUTH_START_ANGLE,  
          MOUTH_ARC_SWEEP);
```
  - The arguments **MOUTH\_WIDTH** and **MOUTH\_HEIGHT** determine the size of the bounding box, while the arguments **X\_MOUTH** and **Y\_MOUTH** determine its location
  - The last two arguments specify the portion made visible

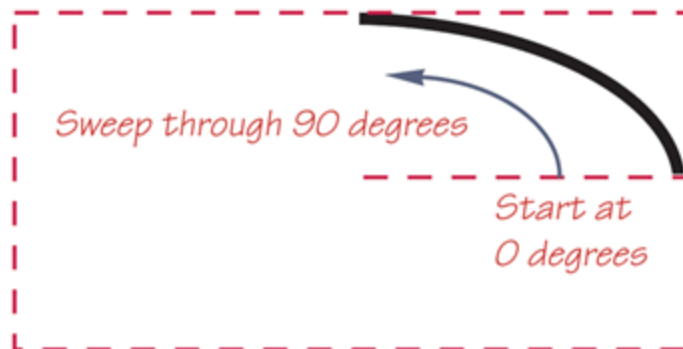




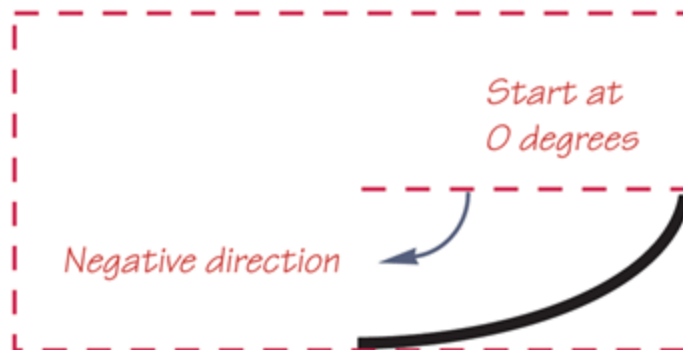
# Specifying an Arc (Part 1 of 2)

## Specifying an Arc

```
g.drawArc(x, y, width, height, 0, 90);
```



```
g.drawArc(x, y, width, height, 0, -90);
```



(continued)



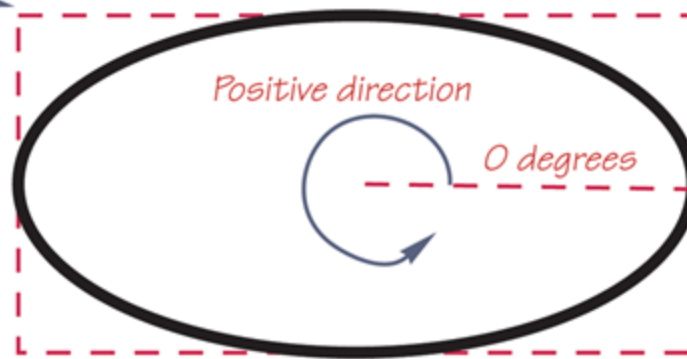
# Specifying an Arc (Part 2 of 2)

## Specifying an Arc

```
g.drawArc(x, y, width, height, 0, 360);
```

(x, y)

width

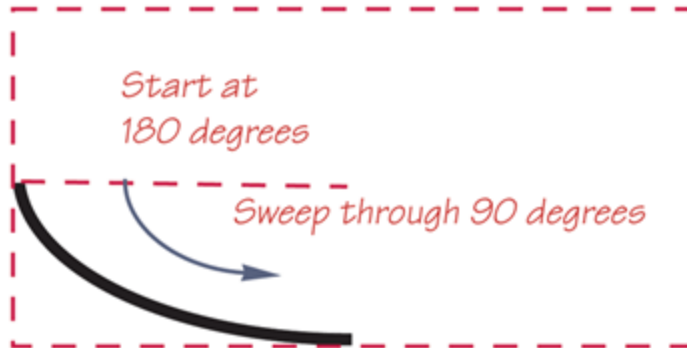


height

```
g.drawArc(x, y, width, height, 180, 90);
```

Start at  
180 degrees

Sweep through 90 degrees



(continued)



# Lab

```
import java.awt.FlowLayout;
import java.awt.Graphics;
import javax.swing.JFrame;

public class PaintTest extends JFrame{

    public static void main(String[] args) {
        PaintTest frame = new PaintTest();
        frame.setVisible(true);
    }

    public PaintTest(){
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
    }

    public void paint(Graphics g){
        super.paint(g);
        g.drawArc(100, 100, 500, 300, 180, 180);
    }

}
```



# Lab





# Rounded Rectangles

- ❑ A *rounded rectangle* is a rectangle whose corners have been replaced by arcs so that the corners are rounded

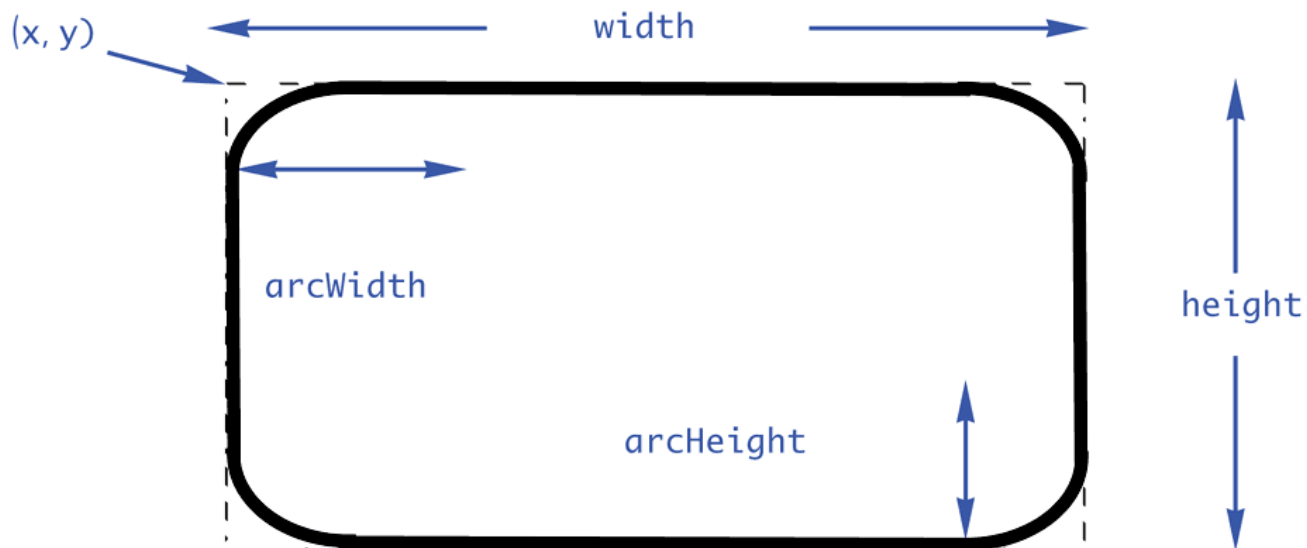
```
g.drawRoundRect(x, y, width, height,  
                arcWidth, arcHeight)
```

- The arguments **x**, **y**, **width**, and **height** determine a regular rectangle in the usual way
- The last two arguments **arcWidth**, and **arcHeight**, specify the arcs that will be used for the corners
- Each corner is replaced with an quarter of an oval that is **arcWidth** pixels wide and **arcHeight** pixels high
- When **arcWidth** and **arcHeight** are equal, the corners will be arcs of circles



# A Rounded Rectangle

`g.drawRoundRect(x, y, width, height, arcWidth, arcHeight);`  
produces:





# Lab

```
import java.awt.FlowLayout;
import java.awt.Graphics;
import javax.swing.JFrame;

public class PaintTest extends JFrame{

    public static void main(String[] args) {
        PaintTest frame = new PaintTest();
        frame.setVisible(true);
    }

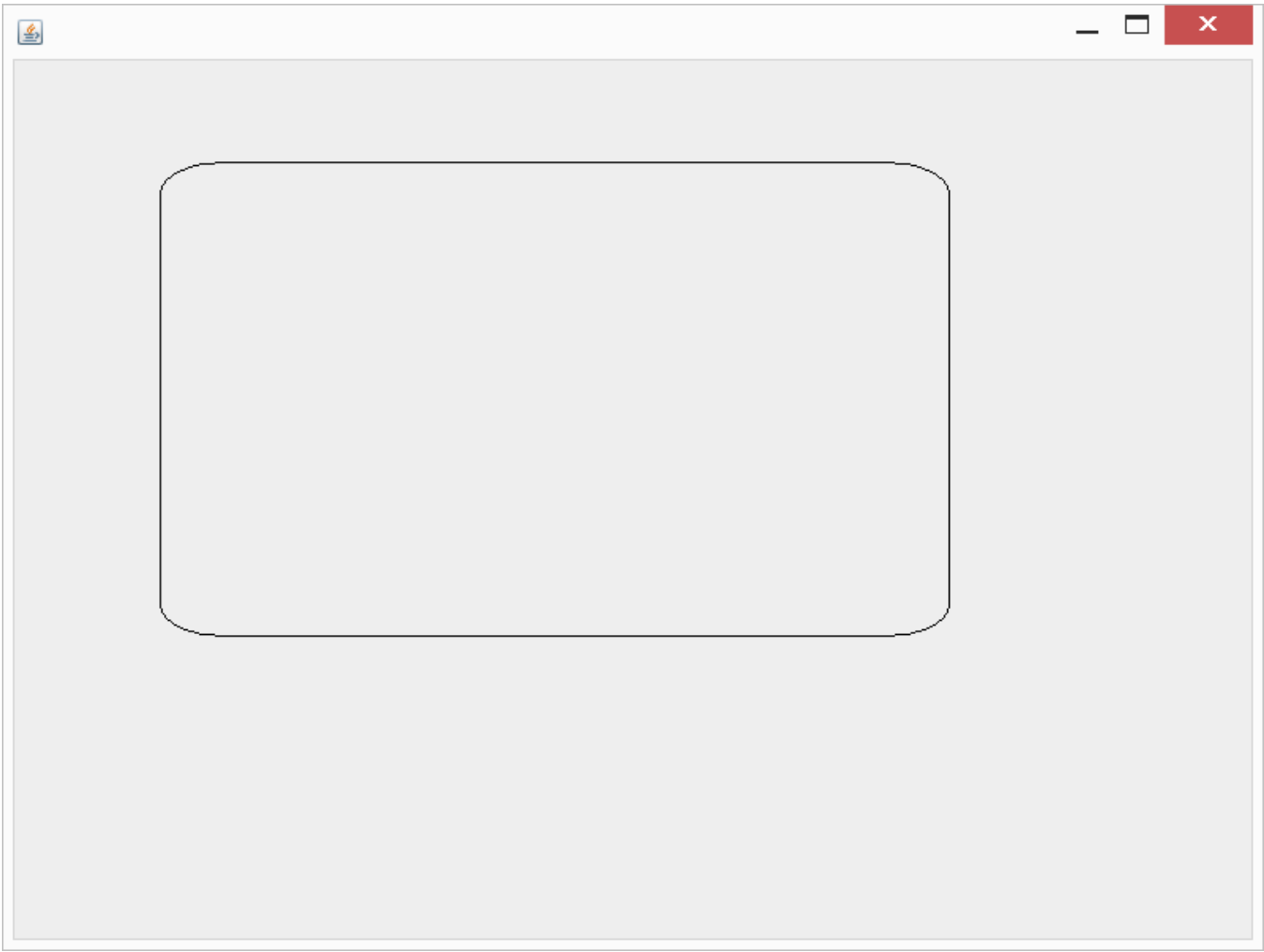
    public PaintTest(){
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
    }

    public void paint(Graphics g){
        super.paint(g);
        g.drawRoundRect(100, 100, 500, 300, 80, 40);
    }

}
```



# Lab







# Lab

```
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Graphics;
import javax.swing.JFrame;

public class PaintTest extends JFrame{

    public static void main(String[] args) {
        PaintTest frame = new PaintTest();
        frame.setVisible(true);
    }

    public PaintTest(){
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(null);

        MyPanel mypanel = new MyPanel();
        mypanel.setLocation(100, 100);
        mypanel.setSize(300,300);
        mypanel.setBackground(Color.YELLOW);
        add(mypanel);
    }
}
```



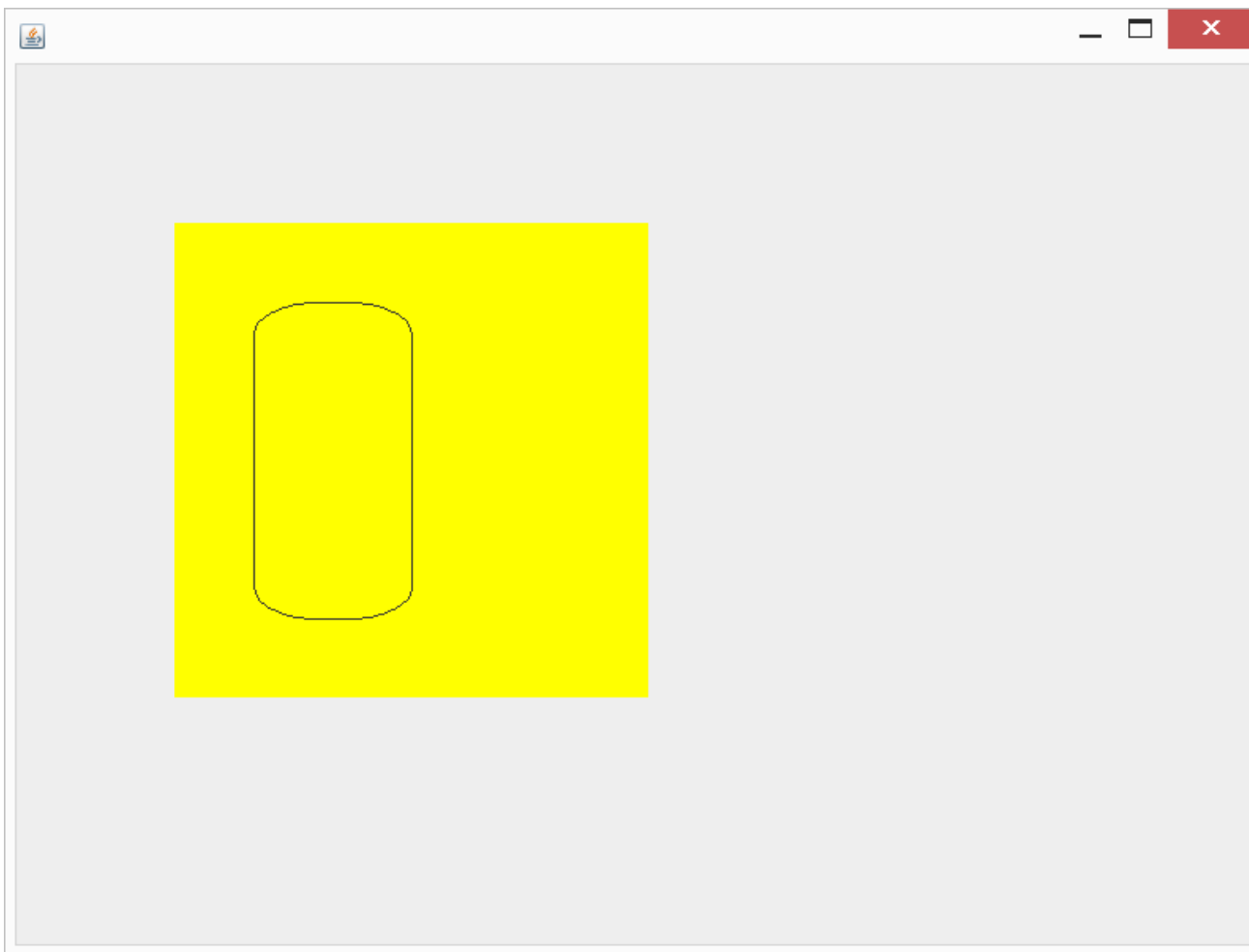
# Lab

---

```
import java.awt.Graphics;  
import javax.swing.JPanel;  
  
public class MyPanel extends JPanel{  
  
    public void paint(Graphics g){  
        super.paint(g);  
        g.drawRoundRect(50, 50, 100, 200, 80, 40);  
    }  
}
```



# Lab





# Action Drawings and `repaint`

- ❑ The `repaint` method should be invoked when the graphics content of a window is changed
  - The `repaint` method takes care of some overhead, and then invokes the method `paint`, which redraws the screen
  - Although the `repaint` method must be explicitly invoked, it is already defined
  - The `paint` method, in contrast, must often be defined, but is not explicitly invoked



# Lab

```
import java.awt.Graphics;
import javax.swing.JPanel;

public class MyPanel extends JPanel{

    private int state = 0;

    public void paint(Graphics g){
        super.paint(g);

        if(state ==0){
            g.drawRoundRect(50, 50, 100, 200, 80, 40);
        }else if(state==1){
            g.drawOval(50, 50, 10, 20);
        }
    }

    public void setState(int state){
        this.state = state;
    }
    public int getState(){
        return this.state;
    }
}
```



# Lab

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

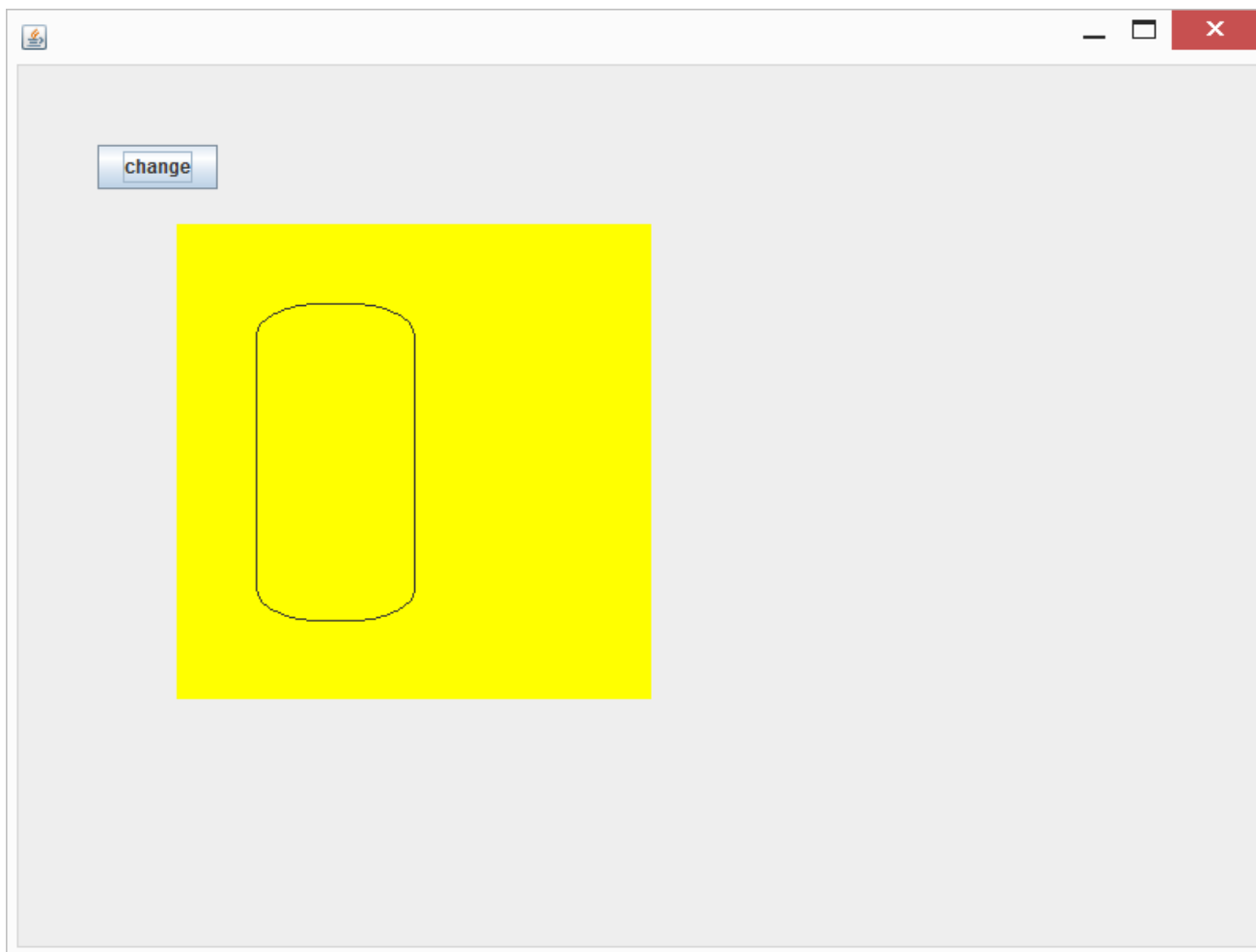
public class PaintTest extends JFrame implements ActionListener{
    MyPanel mypanel;
    public static void main(String[] args) {
        PaintTest frame = new PaintTest();
        frame.setVisible(true);
    }
    public PaintTest(){
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(null);

        mypanel = new MyPanel();
        mypanel.setLocation(100, 100);
        mypanel.setSize(300,300);
        mypanel.setBackground(Color.YELLOW);
        add(mypanel);

        JButton btn = new JButton("change");
        btn.addActionListener(this);
        btn.setLocation(50,50);
        btn.setSize(btn.getPreferredSize());
        add(btn);
    }
    public void actionPerformed(ActionEvent e) {
        if(mypanel.getState() == 0){
            mypanel.setState(1);
        }else{
            mypanel.setState(0);
        }
        mypanel.repaint();
    }
}
```



# Lab





# Specifying a Drawing Color

- ❑ Using the method **drawLine** inside the **paint** method is similar to drawing with a pen that can change colors
    - The method **setColor** will change the color of the pen
    - The color specified can be changed later on with another invocation of **setColor** so that a single drawing can have multiple colors
- `g.setColor(Color.BLUE)`**





# The drawString Method

- ❑ The method **drawString** is similar to the drawing methods in the **Graphics** class
  - However, it displays text instead of a drawing
  - If no font is specified, a default font is used

**`g.drawString(theText, X_START, Y_Start);`**



# Fonts

- ❑ A font is an object of the **Font** class
  - The **Font** class is found in the **java.awt** package
- ❑ The constructor for the **Font** class creates a font in a given style and size

```
Font fontObject = new Font("SansSerif",  
                             Font.PLAIN, POINT_SIZE);
```
- ❑ A program can set the font for the **drawString** method within the **paint** method

```
g.setFont(fontObject);
```



# Font Types

---

- ❑ Any font currently available on a system can be used in Java
  - However, Java guarantees that at least three fonts will be available:  
**"Monospaced"**, **"SansSerif"**, and **"Serif"**



# Font Styles

❑ Fonts can be given style modifiers, such as bold or italic

- Multiple styles can be specified by connecting them with the `|` symbol (called the bitwise OR symbol)

```
new Font("Serif",  
        Font.BOLD|Font.ITALIC, POINT_SIZE);
```

❑ The size of a font is called its *point size*

- Character sizes are specified in units known as *points*
- One point is 1/72 of an inch



# Lab

```
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import javax.swing.JFrame;

public class DrawStringTest extends JFrame{

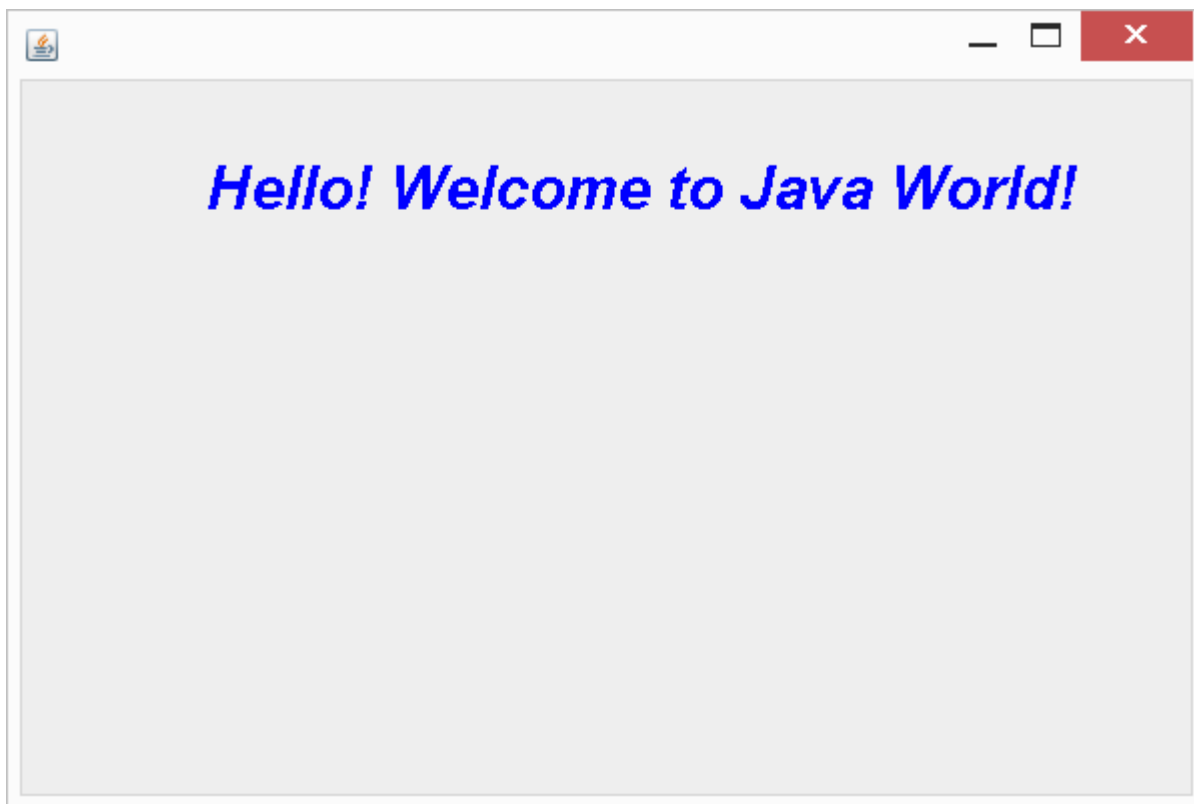
    public static void main(String[] args) {
        DrawStringTest frame = new DrawStringTest();
        frame.setVisible(true);
    }

    public DrawStringTest(){
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void paint(Graphics g){
        super.paint(g);
        g.setColor(Color.BLUE);
        Font f = new Font("Arial Bold",Font.BOLD/Font.ITALIC,30);
        g.setFont(f);
        g.drawString("Hello! Welcome to Java World!", 100, 100);
    }
}
```



# Lab





# Reference

- ❑ “Absolute Java”. Walter Savitch and Kenrick Mock. Addison-Wesley; 5 edition. 2012
- ❑ “Java How to Program”. Paul Deitel and Harvey Deitel. Prentice Hall; 9 edition. 2011.
- ❑ “A Programmers Guide To Java SCJP Certification: A Comprehensive Primer 3rd Edition”. Khalid Mughal, Rolf Rasmussen. Addison-Wesley Professional. 2008