Helwan University

Faculty of Computer Science

Programming Language 2

Object oriented design using JAVA

Dr. Ayman Ezzat

Email: ayman@fcih.net

Web: www.fcih.net/ayman

Unit 7: Event driven programming

Outline

1. Introduction
2. The `Swing` event system
3. Component listeners
4. Graphics
5. Some examples

1. Introduction

- **Subject of the unit is “event-driven programming”**
  - To create this link between something happening in the graphical user interface – **an event** – and the **execution of code**.
  - Event-driven programs are programs that respond to
    - **Events initiated by the user** :
      - a button being clicked;
      - a text field being modified;
      - a menu being accessed;
      - a list element being selected.
    - **Events initiated through software coding**, using for example the **Timer** class, which may specify that certain events are to take place at certain times.
- **Listeners**:
  - A major part of event-driven programming is the creation of objects called listeners that are attached to components.
  - As their name suggests they 'listen' for events happening to 'their' components. These listeners then 'fire' off the desired code.
  - Listeners can be 'told' to look at particular components and to look only at particular events. For example, a listener may be interested only in a mouse entering a particular part of the screen and not in it being clicked.

2. The `Swing` event system

Overview of event handling

- To describe how events are handled, two small items need to be introduced:

  (1) An event source is an object that gives rise to an event.

  - Typical event sources are buttons and menu lists.

  (2) An event listener is an object that responds to an event.

  - An event listener is **an object of a class** that **implements** a specific type of **listener interface**.
  - Inside the listener you put the code that triggers off the processing that you want to happen when an event occurs.

- The source of the event and the place where the event is dealt with are **separate**.
  - **Why separate?**
    This nicely separates the interface (the visual components) from the implementation (the code that carries out the processing). If, at a later date, either the graphical user interface or the processing code has to change then these changes can be localized.

2. The Swing event system

3 steps to set up Event Handling for a GUI Component:

1. **Create the GUI component** on the graphical user interface.

2. **Define the event listener class** that represents the event handler.
   - This class should `implement` an appropriate listener interface.

3. **Create an object of the event listener class**, and **register this object** with the GUI component
   - `addActionListener` method

```
public class myFrame extends JFrame{
    private JTextField textField1; //create a text box
    ...
    // JFrame constructor
    public myFrame(){
        myHandler handler = new myHandler();
        textField1.addActionListener( handler );
        ...
        // private inner class for event handling
        private class myHandler implements ActionListener{
            public void actionPerformed( ActionEvent event ){
                // do something here
            }
}}
```

This is called an **inner class:**
a class defined inside another class

2. The Swing event system

Example 2:

when the button is clicked, the

number on the right is incremented.


```
public class Main {
    public static void main(String[] args) {
        MyFrame world = new MyFrame("Button Watcher");
        world.setVisible(true);
    }
}
```

```
import java.awt.event.*;
import javax.swing.*;
public class MyFrame extends JFrame {
    private int buttonClicks;
    private JButton button;
    private JLabel output;
    public MyFrame(String s) {
        super(s);
        setSize(300, 300);
        buttonClicks = 0;
        button = new JButton("Press Me");
        output = new JLabel("Starting");
        getContentPane().add(button, "West");
        getContentPane().add(output, "East");
        /* Create an instance of ButtonWatcher and
        register this object listener with the button. */
        button.addActionListener(new ButtonWatcher());
    }

    /* Define the listener class by implementing
    the interface ActionListener. */
    private class ButtonWatcher implements ActionListener {
        /* this code is executed when the button is clicked. */
        public void actionPerformed(ActionEvent a) {
            buttonClicks++;
            output.setText("" + buttonClicks);
            if (buttonClicks == 10) {
                buttonClicks = 0;
            }}}
}
```

2. The `Swing` event system

In Example 2…

- There are some final points worth making about the above code.
  (1) We need to **import**
    - **javax.swing** for the visual object classes
    - **java.awt.event** for the event classes and interfaces;
  (2) The actionPerformed method has a single ActionEvent argument a,
    - However, it is not used in this example.
    - This argument contains subsidiary information about the event, which can be used for more complicated processing. This is discussed in the next example!

2. The Swing event system

Example 3:

this example shows the use of the **event argument.** Notice the **getsSource** method


```
public class Main {
    public static void main(String[] args) {
        MySecondFrame world2 = new MySecondFrame("Two Buttons");
        world2.setVisible(true);

    }
}
```

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class MySecondFrame extends JFrame {
    private int buttonClicks;
    private JButton buttonUp, buttonDown;
    private JLabel label;
    public MySecondFrame(String s) {
        super(s);
        setSize(300, 300);
        buttonClicks = 0;
        buttonUp = new JButton("Press Me Up");
        buttonDown = new JButton("Press Me Down");
        label = new JLabel("Starting");
        buttonUp.addActionListener(new ButtonWatcher());
        buttonDown.addActionListener(new ButtonWatcher());
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(buttonUp);
        getContentPane().add(buttonDown);
        getContentPane().add(label);
    }
    private class ButtonWatcher implements ActionListener {
        public void actionPerformed(ActionEvent a) {
            Object buttonPressed = a.getSource();
            if (buttonPressed.equals(buttonUp))
                buttonClicks++;
            if (buttonPressed.equals(buttonDown))
                buttonClicks--;
            label.setText(buttonClicks + "");
        }}
}
```

2. The `Swing` event system

In Example 3…

- there are two buttons.
  - The first button increments the integer displayed in the label and the second decrements the integer.
  - Both buttons have the same listener (ButtonWatcher) attached,
  - so when actionPerformed is executed there **is a need to determine the source** of the event. This is done by invoking the method **getSource**.
  - The **getSource method** returns an object and, using the equals method of the Object class, the specific button pressed can be identified and the appropriate code executed.

Outline

1. Introduction
2. The `Swing` event system
3. Component listeners
4. Graphics
5. Some examples

3. Component listeners

- There are **many listener interfaces** being more complex than the **ActionListener**.
  - *Although we will not be able to cover every aspect of all of the events and component combinations, you will have sufficient experience of them to be able to use the Java reference material for the rest.*

- The **names of the listeners** are formed by the ***name of the event*** *(see list below)* + the word **`listener`**
  - so we have *`ComponentListener`*, *`KeyListener`* and so on.
- The **add** and **remove** methods simply have the word **`add`** **(or `remove`) prefixing** the **interface name**
  - such as *`addComponentListener`* *and* *`removeKeyListener`*.

Table 1 Events generated by GUI objects

Type of event	Associated components
Action events	buttons, lists, menu items and text fields
Adjustment events	scroll bars
Item events	check boxes, choices and lists
Text events	text components such as text fields and text areas
Component events	visual components: for example, being resized or hidden
Container events	containers such as frames: for example, when a component is added or removed
Focus events	components coming into focus or going out of focus
Key events	keys being pressed or released
Mouse events	actions such as clicking a mouse or moving a mouse
Window events	windows: for example, opening or closing a window

3. Component listeners

Mouse listeners

- In Java there are two interfaces that can be used for mouse events:
  - **MouseMotionListener**: This defines two events concerned with dragging and moving the mouse (*we will not focus on this one*)
  - **MouseListener:** This defines **five methods** that are used for monitoring:
    - **mousClicked**: when a mouse button is clicked

```
public abstract void mouseClicked(MouseEvent m)
```

    - **mouseEntered**: when a mouse enters a component

```
public abstract void mouseEntered(MouseEvent m)
```

    - **mouseExited**: when a mouse exits a component

```
public abstract void mouseExited(MouseEvent m)
```

    - **mousePressed**: when a mouse is pressed

```
public abstract void mousePressed(MouseEvent m)
```

    - **mouseReleased**: when a mouse button is released

```
public abstract void mouseReleased(MouseEvent m)
```

  - Any listener that wants to react to any of the above events **must implement** the **MouseListener interface**, which contains the definition of the above five methods.

3. Component listeners

Mouse listeners – Example

- The code defines a **window** that contains **two labels** and a **panel**,
  - with the panel being coloured yellow (this is achieved by means of the method setBackground, which uses the static variable Color.yellow).
- When the **mouse is clicked** within the panel the labels display the x- and y-coordinates of the point within the panel that the mouse was clicked in.
  - The methods **getX** and **getY** from the class **MouseEvent** are used to retrieve these values from the MouseEvent argument.

```
public class Main {
    public static void main(String[] args) {
        MouseFrame world3 = new MouseFrame("Mouse Watcher");
        world3.setVisible(true);
    }
}
```

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;
public class MouseFrame extends JFrame {
    private JLabel yCoordLabel;
    private JLabel xCoordLabel;
    private JPanel c;
    public MouseFrame(String title) {
        super(title);
        setSize(300, 300);
        yCoordLabel = new JLabel("");
        xCoordLabel = new JLabel("");
        c = new JPanel();
        c.setBackground(Color.yellow);
        getContentPane().setLayout(new GridLayout(3, 1));
        getContentPane().add(yCoordLabel);
        getContentPane().add(xCoordLabel);
        getContentPane().add(c);
        c.addMouseListener(new MouseWatcher());
    }
    private class MouseWatcher implements MouseListener {
        public void mouseClicked(MouseEvent e) {
            int xCoordinate = e.getX();
            int yCoordinate = e.getY();
            yCoordLabel.setText(yCoordinate + "");
            xCoordLabel.setText(xCoordinate + "");
        }
        public void mouseEntered(MouseEvent e) {}
        public void mouseReleased(MouseEvent e) {}
        public void mousePressed(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
    }
}
```

3. Component listeners

Adapter classes

- These are classes that are **analogues of the interfaces**, which **implement the methods** asked for by the interfaces by means of providing the **empty methods**.
- The programmer who wants to react to a small number of events can **inherit** from these adapter classes (**instead of implementing an interface**)
- An example of an adapter class is `MouseAdapter`. This class provides empty bodies for the five methods detailed above.
- We **use** adapters in the **same way as listeners** by defining inner classes, except now we **extend** rather than **implement**.

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;
public class NewMouser extends JFrame {
    private JLabel yCoordLabel, xCoordLabel;
    private JPanel c;
    public NewMouser(String title) {
        super(title);
        setSize(300, 300);
        yCoordLabel = new JLabel("");
        xCoordLabel = new JLabel("");
        c = new JPanel();
        c.setBackground(Color.yellow);
        getContentPane().setLayout(new GridLayout(3, 1));
        getContentPane().add(yCoordLabel);
        getContentPane().add(xCoordLabel);
        getContentPane().add(c);
        c.addMouseListener(new MouseEventer());
    }
// inner class
    private class MouseEventer extends MouseAdapter {

        public void mouseClicked(MouseEvent e) {
            int xCoordinate = e.getX();
            int yCoordinate = e.getY();
            yCoordLabel.setText(yCoordinate + "");
            xCoordLabel.setText(xCoordinate + "");
        }
    }// end of inner class
}
```

ArrayList for storing mouse pressed locations

```
private class mouselisten extends MouseAdapter
{
    public void mousePressed(MouseEvent e)
    {
        currentx=e.getX();
        currenty=e.getY();
        allovals.add(new Point (currentx,currenty));
        repaint();
    }
}
```

Mouse motion

```
private class mymousemove extends MouseMotionAdapter
{
    public void mouseDragged(MouseEvent evt) {
     Graphics g = getGraphics();
     g.drawLine(10, 10, evt.getX(), evt.getY());
     currentx=evt.getX();
     currenty=evt.getY();
     g.setColor(Color.blue);
     g.drawLine(currentx,currenty,++currentx, ++currenty);

    }
}
```

3. Component listeners

Window listeners

- In previous examples, clicking on the close window icon does not end the program, it simply closes the window.
- Two solutions (use either (1) or (2)):

  (1) use the method **`setCloseOperation`**

```
MyFrame world = new MyFrame("Button Watcher");
world.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

  (2) Link the clicking of that icon with program code using the **`WindowAdapter`** class

  - `WindowAdapter` is the adapter class for the WindowListener interface
  - `WindowAdapter` contains seven methods including **`windowOpening`**, **`windowClosing`** and **`windowClosed`**.
  - We will use the **`windowClosing`** method to link the clicking of the close window icon with our program and get the program to stop when the window is closed.

3. Component listeners

Window listeners - Example

(1) Define the following class

```
import java.awt.event.*; import javax.swing.JOptionPane;
public class CloseAndQuit extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        JOptionPane.showMessageDialog(null, "Now Exiting...!");
        System.exit(0);
    }
}
```

(2) Add the following line anywhere in the constructor of the our frame class in any previous example (after the call to super, of course).

```
addWindowListener(new CloseAndQuit());
```

– By putting the class CloseAndQuit into the same package as NewMouser we will see that when we click the close icon on the window, the whole program stops.

Outline

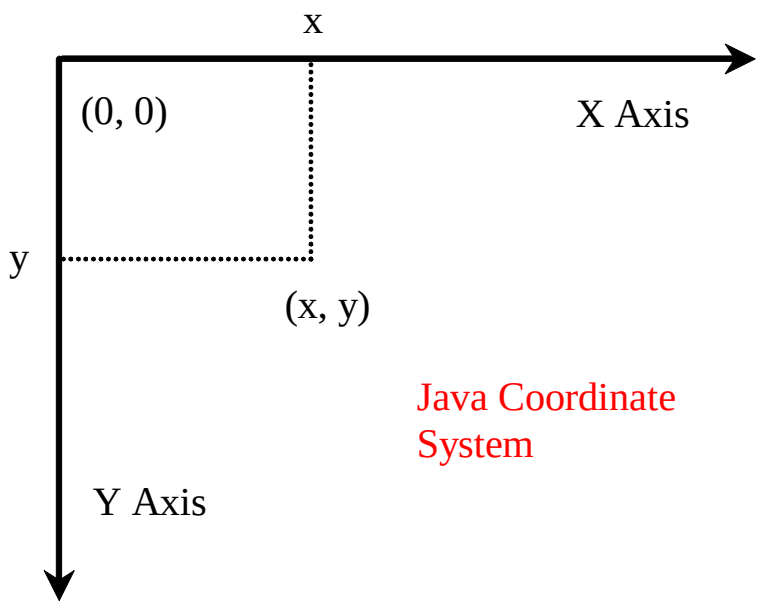
1. Introduction
2. The `Swing` event system
3. Component listeners
4. Graphics
5. Some examples

4. Graphics

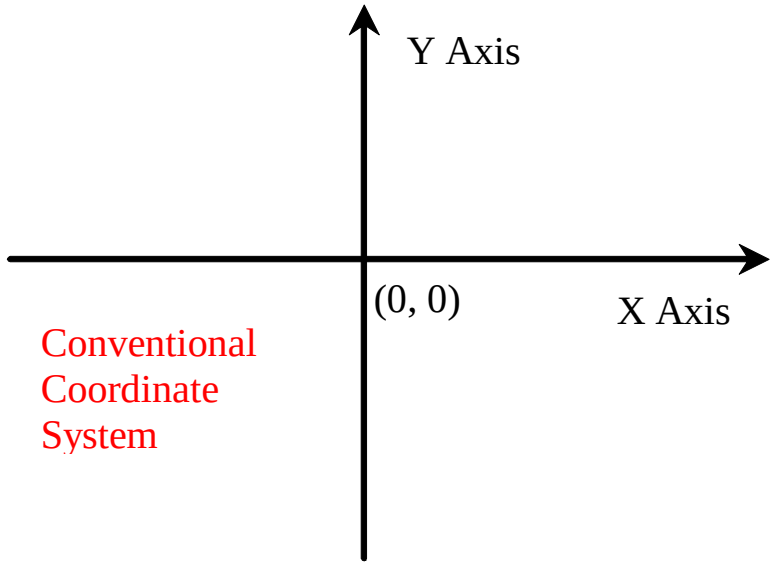
Drawing methods

- **The Graphics class**:
  - There is a special class known as Graphics that enables drawing to take place.
  - The Graphics class is an abstract class that provides a graphical context for drawing
- **The Paint method**
  - The drawing process is achieved by means of the programmer providing a concrete implementation of the paint method.
    - This takes a Graphics object as its argument and carries out drawing actions on that object, which will be displayed on a screen.
  - The paint method is **invoked whenever**:
    (1) a GUI object is created
    (2) some external action that requires redrawing (such as a window being moved) occurs.
    (3) It is called inside a program using such method as **repaint()**.

Java Coordinate System

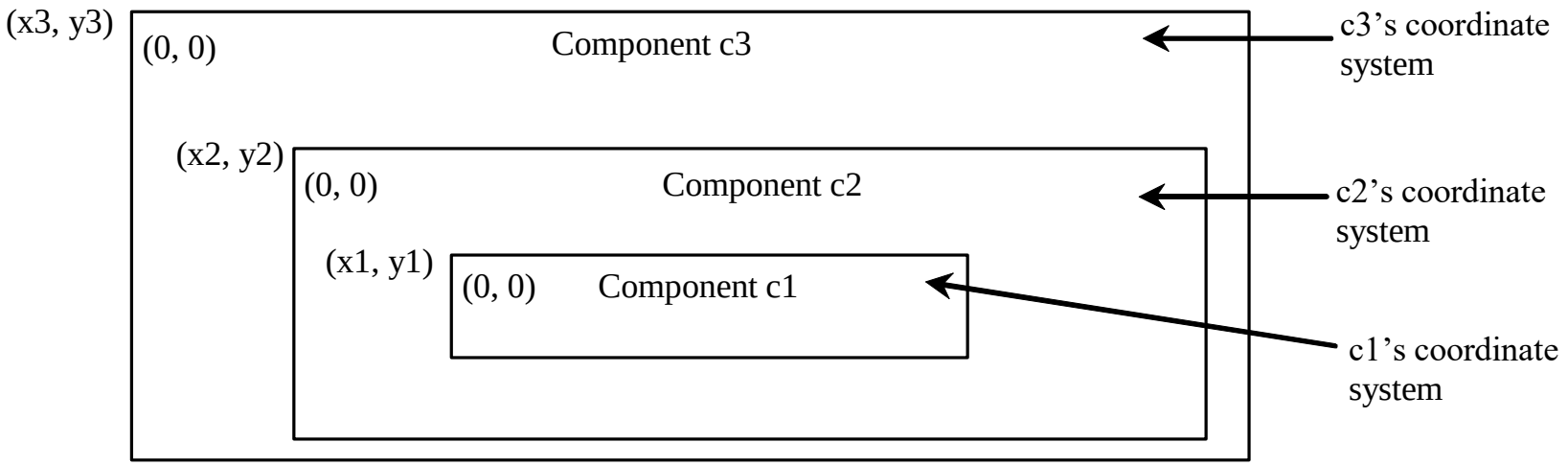


Java Coordinate System



Conventional Coordinate System

Each GUI Component Has its Own Coordinate System



The Graphics Class

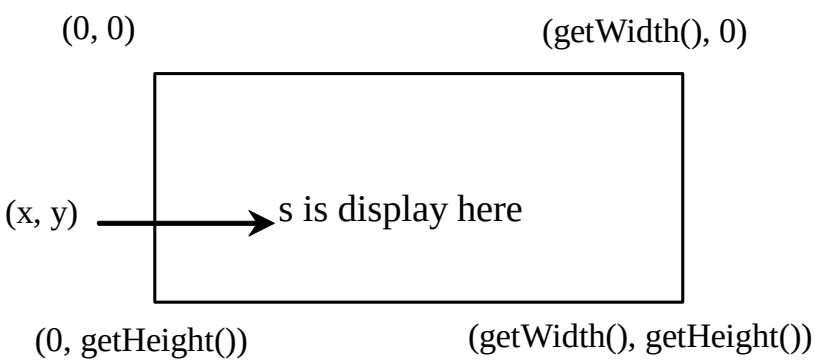
You can draw strings, lines, rectangles, ovals, arcs, polygons, and polylines, using the methods in the Graphics class.

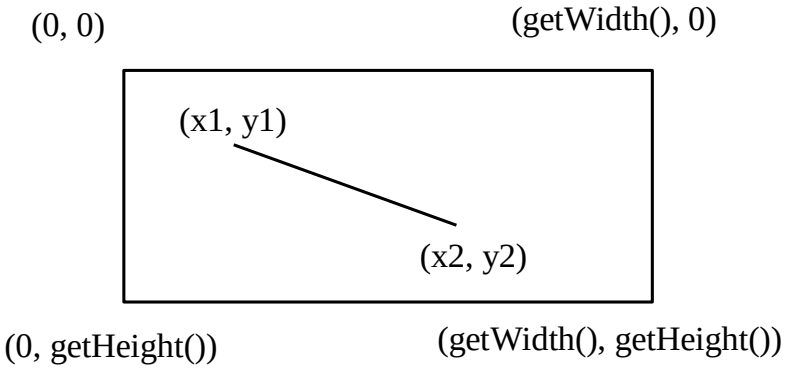
java.awt.Graphics	
+setColor(color: Color): void	Sets a new color for subsequent drawings.
+setFont(font: Font): void	Sets a new font for subsequent drwings.
+drawString(s: String, x: int, y: int): void	Draws a string starting at point (x, y).
+drawLine(x1: int, y1: int, x2: int, y2: int): void	Draws a line from (x1, y1) to (x2, y2).
+drawRect(x: int, y: int, w: int, h: int): void	Draws a rectangle with specified upper-left corner point at (x, y) and width w and height h.
+fillRect(x: int, y: int, w: int, h: int): void	Draws a filled rectangle with specified upper-left corner point at (x, y) and width w and height h.
+drawRoundRect(x: int, y: int, w: int, h: int, aw: int, ah: int): void	Draws a round-cornered rectangle with specified arc width aw and arc height ah.
+fillRoundRect(x: int, y: int, w: int, h: int, aw: int, ah: int): void	Draws a filled round-cornered rectangle with specified arc width aw and arc height ah.
+draw3DRect(x: int, y: int, w: int, h: int, raised: boolean): void	Draws a 3-D rectangle raised above the surface or sunk into the surface.
+fill3DRect(x: int, y: int, w: int, h: int, raised: boolean): void	Draws a filled 3-D rectangle raised above the surface or sunk into the surface.
+drawOval(x: int, y: int, w: int, h: int): void	Draws an oval bounded by the rectangle specified by the parameters x, y, w, and h.
+fillOval(x: int, y: int, w: int, h: int): void	Draws a filled oval bounded by the rectangle specified by the parameters x, y, w, and h.
+drawArc(x: int, y: int, w: int, h: int, startAngle: int, arcAngle: int): void	Draws an arc conceived as part of an oval bounded by the rectangle specified by the parameters x, y, w, and h.
+fillArc(x: int, y: int, w: int, h: int, startAngle: int, arcAngle: int): void	Draws a filled arc conceived as part of an oval bounded by the rectangle specified by the parameters x, y, w, and h.
+drawPolygon(xPoints: int[], yPoints: int[], nPoints: int): void	Draws a closed polygon defined by arrays of x and y coordinates. Each pair of (x[i], y[i]) coordinates is a point.
+fillPolygon(xPoints: int[], yPoints: int[], nPoints: int): void	Draws a filled polygon defined by arrays of x and y coordinates. Each pair of (x[i], y[i]) coordinates is a point.
+drawPolygon(g: Polygon): void	Draws a closed polygon defined by a Polygon object.
+fillPolygon(g: Polygon): void	Draws a filled polygon defined by a Polygon object.
+drawPolyline(xPoints: int[], yPoints: int[], nPoints: int): void	Draws a polyline defined by arrays of x and y coordinates. Each pair of (x[i], y[i]) coordinates is a point.

Drawing Geometric Figures

- Drawing Strings
- Drawing Lines
- Drawing Rectangles
- Drawing Ovals
- Drawing Arcs
- Drawing Polygons

Drawing Strings , line





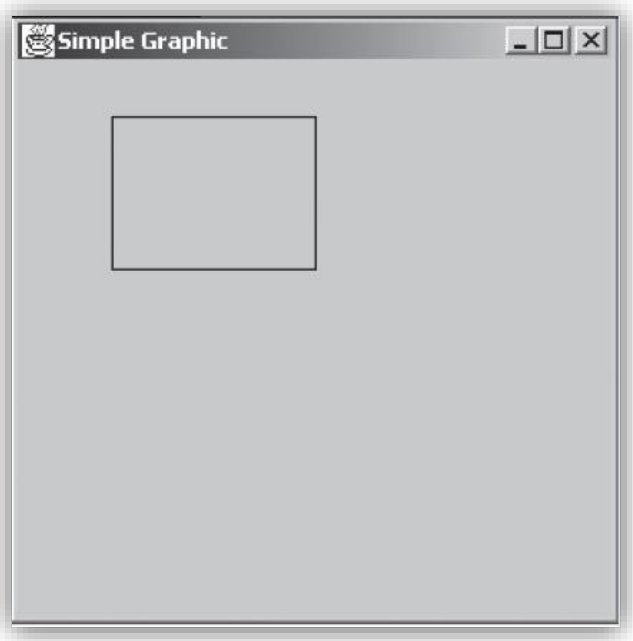
drawString(String s, int x, int y);

drawLine(int x1, int y1, int x2, int y2);

4. Rectangles

Drawing methods - Example

- A rectangle is drawn at an offset (50, 50) from the origin, with a width of 100 and height of 75.
  - The measurements are expressed in pixels, which stands for 'picture elements'→ a single dot in a displayed image.



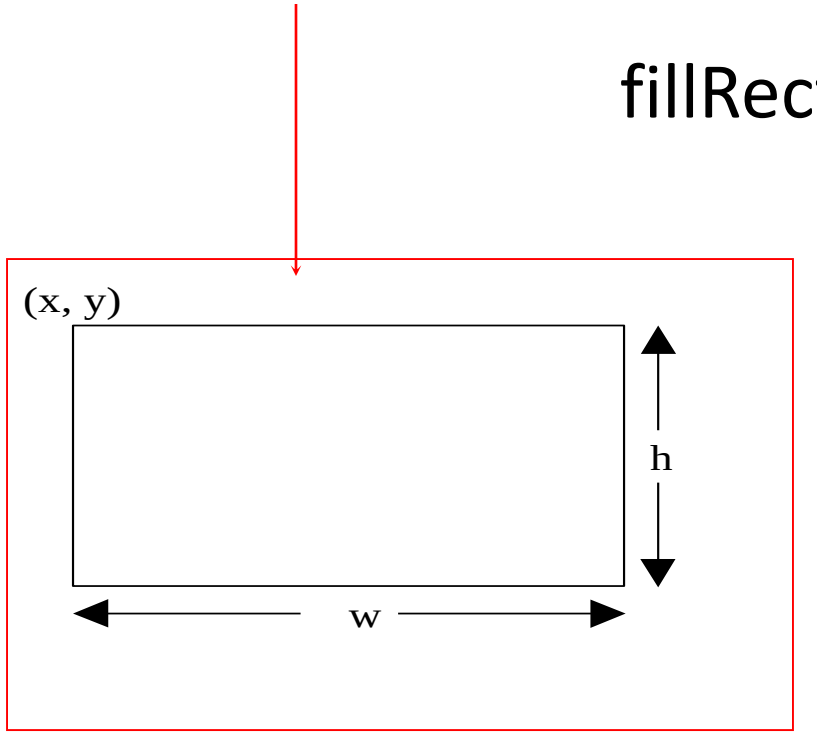

```
import javax.swing.*;
import java.awt.Graphics;
public class SimpleGraphic extends JFrame {
    //constructor
    public SimpleGraphic(String title) {
        super(title);
        setSize(300, 300);
    }
    /* Here we override the method paint to
    create what we want to appear on screen. */
    public void paint(Graphics g) {
        super.paint(g);
        g.drawRect(50, 50, 100, 75);
    }
}
```

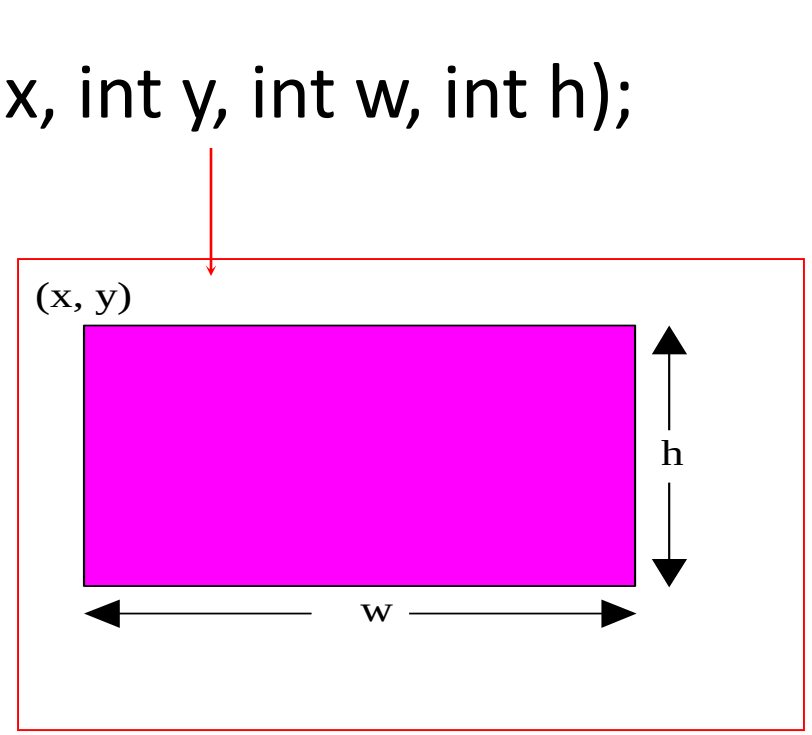
```
public class Main {
    public static void main(String[] args) {
        SimpleGraphic world = new SimpleGraphic("Simple Graphic");
        world.setVisible(true);
    }
}
```

Drawing Rectangles

drawRect(int x, int y, int w, int h);

fillRect(int x, int y, int w, int h);

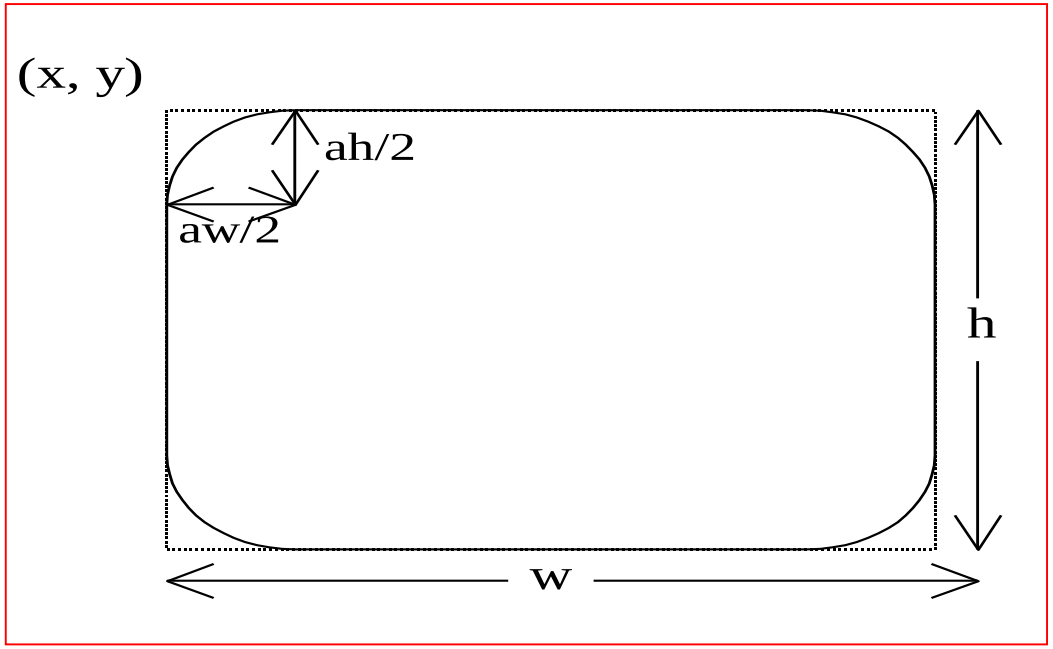




Drawing Rounded Rectangles

drawRoundRect(int x, int y, int w, int h, int aw, int ah);

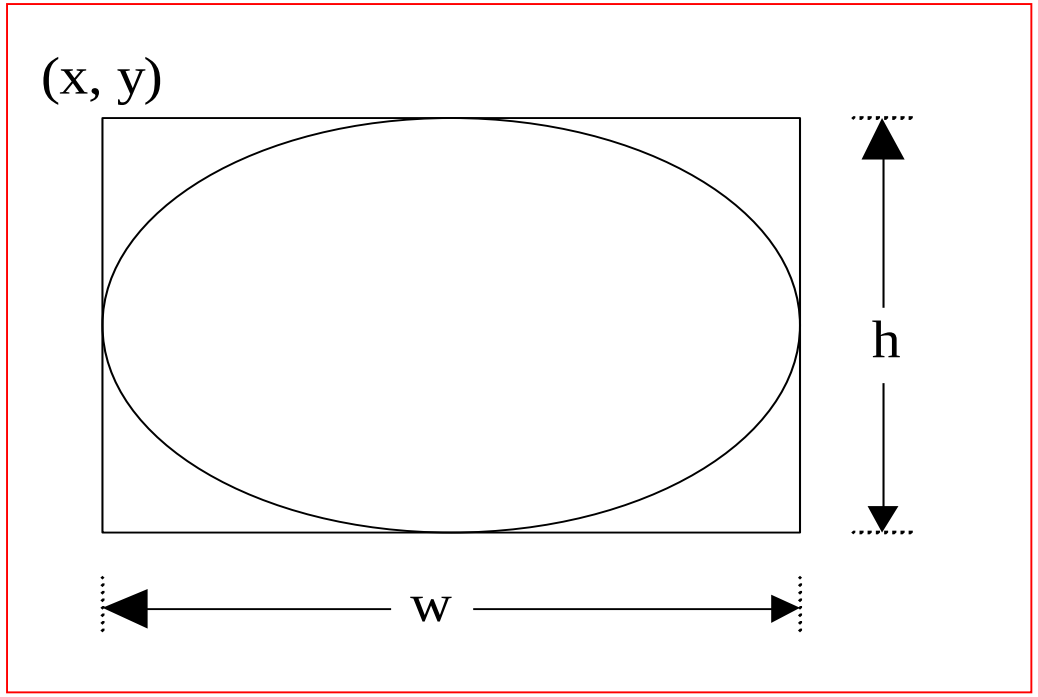
fillRoundRect(int x, int y, int w, int h, int aw, int ah);



Drawing Ovals

drawOval(int x, int y, int w, int h);

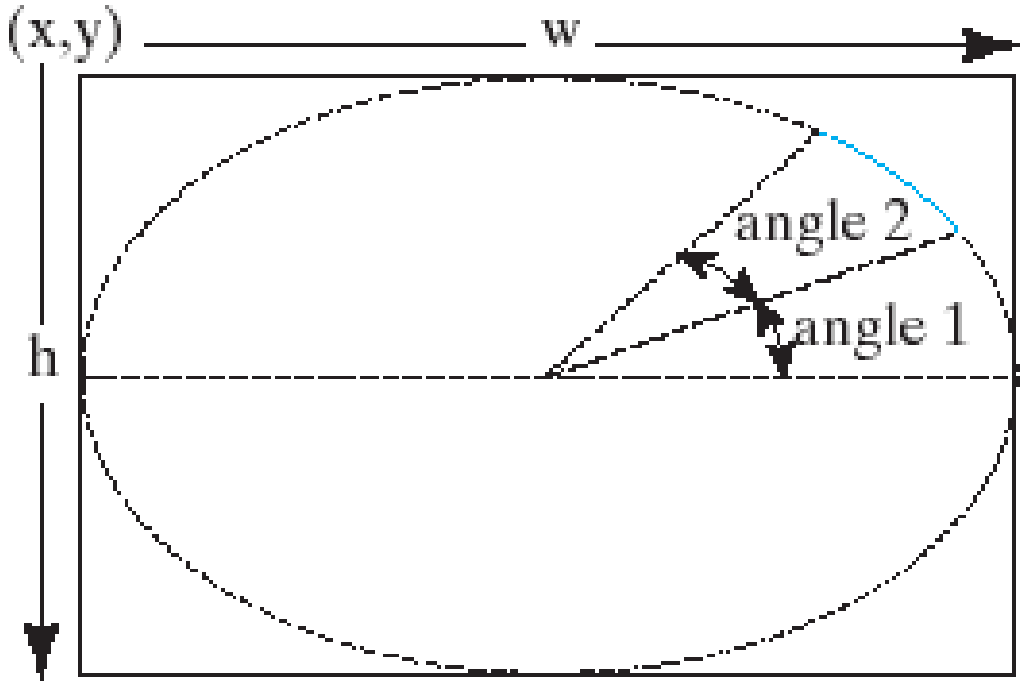
fillOval(int x, int y, int w, int h);



Drawing Arcs

drawArc(int x, int y, int w, int h, int angle1, int angle2);

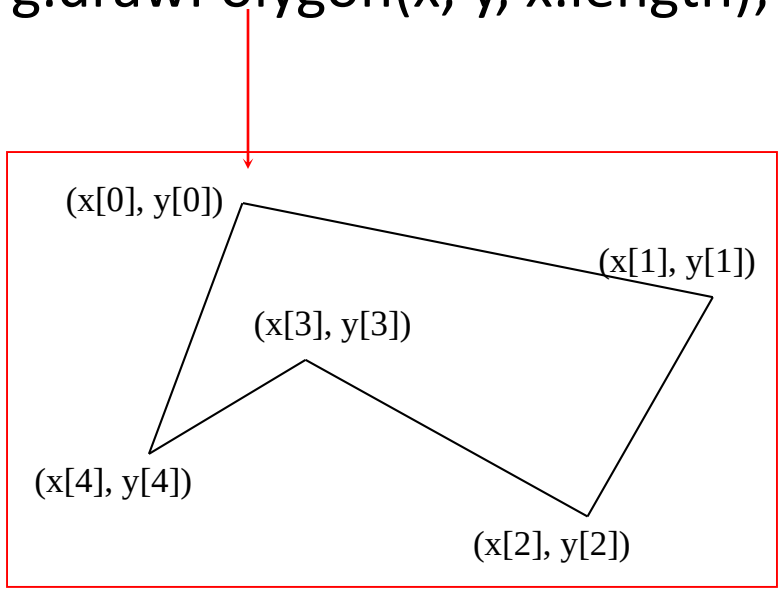
fillArc(int x, int y, int w, int h, int angle1, int angle2);



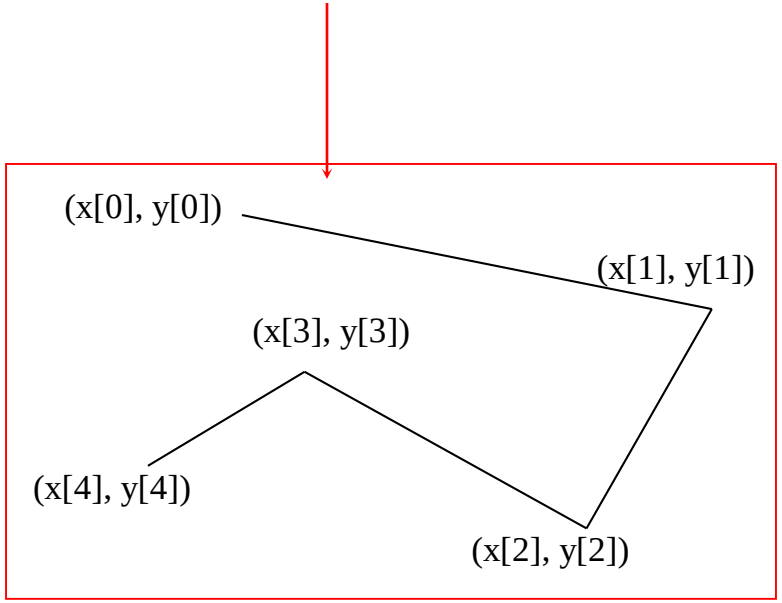
Angles are in degree

Drawing Polygons and Polylines

```
int[] x = {40, 70, 60, 45, 20};
int[] y = {20, 40, 80, 45, 60};
g.drawPolygon(x, y, x.length);
```




```
g.drawPolyline(x, y, x.length);
```



Drawing Polygons Using the Polygon Class

```
Polygon polygon = new Polygon();
polygon.addPoint(40, 59);
polygon.addPoint(40, 100);
polygon.addPoint(10, 100);
g.drawPolygon(polygon);
```

Using Panels as Sub-Containers

- Panels act as sub-containers for grouping user interface components.
- It is recommended that you place the user interface components in panels and place the panels in a frame. You can also place panels in a panel.
- To add a component to JFrame, you actually add it to the content pane of JFrame. To add a component to a panel, you add it directly to the panel using the add method.

paintComponent Example

In order to draw things on a component, you need to define a class that extends JPanel and overrides its paintComponent method to specify what to draw. The first program in this chapter can be rewritten using paintComponent.

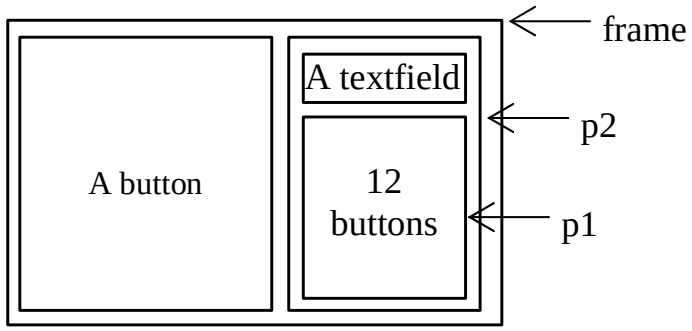
Creating a JPanel

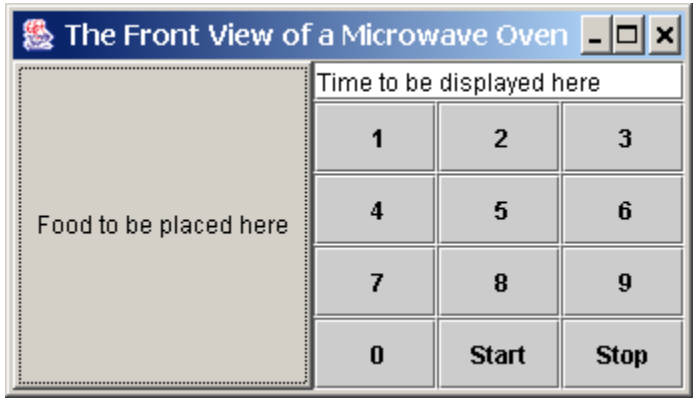
You can use new JPanel() to create a panel with a default FlowLayout manager or new JPanel(LayoutManager) to create a panel with the specified layout manager. Use the add(Component) method to add a component to the panel. For example,

```
JPanel p = new JPanel();
p.add(new JButton("OK"));
```

Testing Panels Example

This example uses panels to organize components. The program creates a user interface for a Microwave oven.





The Font Class

Font Names

Standard font names that are supported in all platforms are: SansSerif, Serif, Monospaced, Dialog, or DialogInput.

Font Style

Font.PLAIN (0), Font.BOLD (1), Font.ITALIC (2), and Font.BOLD + Font.ITALIC (3)

```
Font myFont = new Font(name, style, size);
```

Example:

```
Font myFont = new Font("SansSerif ", Font.BOLD, 16);
Font myFont = new Font("Serif", Font.BOLD+Font.ITALIC, 12);

JButton jbtOK = new JButton("OK");
jbtOK.setFont(myFont);
```

Getting all font names in JAVA

```
GraphicsEnvironment e=GraphicsEnvironment.getLocalGraphicsEnvironment();
String []FontNames=e.getAvailableFontFamilyNames();
for (int i=0;i<FontNames.length;i++)
{
    System.out.println(FontNames[i]);
}
```

Drawing Images

```
try
{
BufferedImage img = ImageIO.read(new File("d:\\1.gif"));
g.drawImage(img, 200, 300,null);
}
catch (IOException e)
{}
```

4. Graphics

Drawing methods – Methods to draw figures and lines

Method name	Description
`drawRect(int,int,int,int)`	This draws the outline of the specified rectangle using the current colour. The top left-hand corner of the rectangle is defined by a pair of *x*- and *y*-coordinates. The final two arguments give the *width* and *height*.
`drawLine(int,int,int,int)`	This draws a line, using the current colour, between the first pair of points and the second pair of points.
`drawRoundRect(int,int,int, int,int,int)`	This draws the outline of a rectangle with rounded corners using the current colour. The top left-hand corner of the rectangle is defined by *x*- and *y*-coordinates, which are the first two arguments; the rectangle's *width* is the third argument and its *height* is the fourth argument. The final two arguments give the *width* and *height* of the arcs used to create the rounded corners.
`drawOval(int,int,int,int)`	This draws the outline of an oval. The result is a circle or an ellipse that fits within the rectangle whose top left-hand corner is specified by the first two arguments, whose *width* is given by the third argument and whose *height* is specified by the final argument.
`drawString(String,int,int)`	This draws the specified string using the current font and colour. The starting position of the string is defined by the *x*- and *y*-coordinates.

Method name	Description
`drawPolygon(Polygon)`	This draws the outline of a polygon defined by the specified `Polygon` object.
`fill3DRect(int,int,int,int, boolean)`	This paints a filled 3D rectangle using the current colour. The first two arguments are the *x*- and *y*-coordinates of the top left-hand corner of the rectangle, the next two are the *width* and *height*, and the final argument indicates whether or not the rectangle is *raised* (it is, if the boolean argument evaluates to `true`).
`fillArc(int,int,int,int, int, int)`	This fills a sector using the current colour. The first two arguments are the *x* and *y* starting coordinates, the next two arguments are the *width* and *height* of the arc, the fifth argument is the starting angle of the arc and the last argument is the final angle. All angles are in degrees.
`fillOval(int,int,int,int)`	This fills an oval inside the specified rectangle using the current colour. The first two arguments give the coordinates of the left-hand corner of the bounding rectangle. The final two arguments give the *width* and *height* of the rectangle.
`fillPolygon(int[],int[], int)`	This fills a polygon with the current colour. The polygon's *x*-values are held in the first array and its *y*-values in the second array. The third argument represents the number of sides to the polygon.