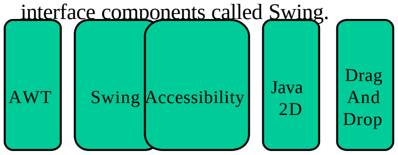
## The Abstract Windowing Toolkit

- Since Java was first released, its user interface facilities have been a significant weakness
  - The Abstract Windowing Toolkit (AWT) was part of the JDK form the beginning, but it really was not sufficient to support a complex user interface
- JDK 1.1 fixed a number of problems, and most notably, it introduced a new event model. It did not make any major additions to the basic components

## Java Foundation Classes

- In April 1997, JavaSoft announced the Java Foundation Classes (JFC).
  - a major part of the JFC is a new set of user interface components called Swing.



AWT

Swing

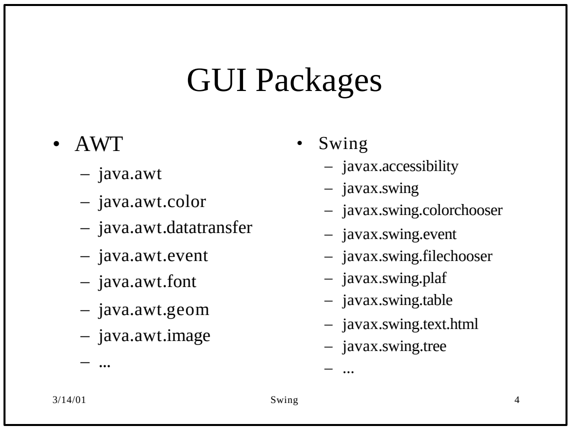
Accessibility

Java 2D

Drag And Drop

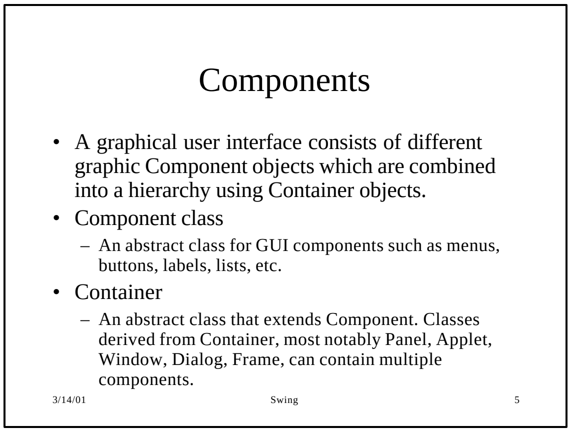
## Swing

- The Swing classes are used to build graphic user interfaces
  - Swing is built on top of the core 1.1 and 1.2 AWT libraries
- Swing makes 3 major improvements on the AWT
  - does not rely on the platform's native components
  - it supports " Pluggable Look-and-Feel" or PLAF
  - it is based on the Model-View-Controller (MVC) design pattern



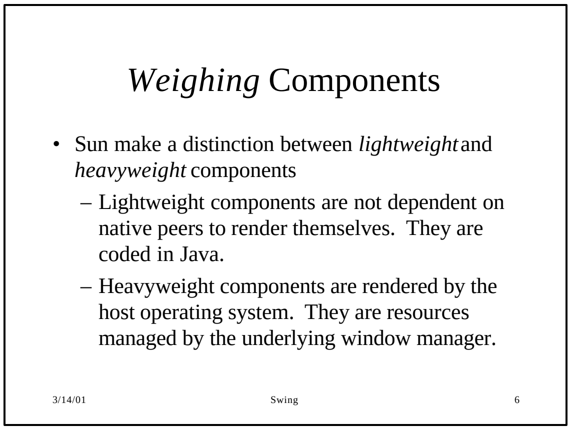
# GUI Packages

- AWT
  - java.awt
  - java.awt.color
  - java.awt.datatransfer
  - java.awt.event
  - java.awt.font
  - java.awt.geom
  - java.awt.image
  - ...
- Swing
  - javax.accessibility
  - javax.swing
  - javax.swing.colorchooser
  - javax.swing.event
  - javax.swing.filechooser
  - javax.swing.plaf
  - javax.swing.table
  - javax.swing.text.html
  - javax.swing.tree
  - ...



# Components

- A graphical user interface consists of different graphic Component objects which are combined into a hierarchy using Container objects.
- Component class
  - An abstract class for GUI components such as menus, buttons, labels, lists, etc.
- Container
  - An abstract class that extends Component. Classes derived from Container, most notably Panel, Applet, Window, Dialog, Frame, can contain multiple components.



# *Weighing* Components

- Sun make a distinction between *lightweight* and *heavyweight* components
  - Lightweight components are not dependent on native peers to render themselves. They are coded in Java.
  - Heavyweight components are rendered by the host operating system. They are resources managed by the underlying window manager.

## Heavyweight Components

- Heavyweight components were unwieldy for two reasons
  - Equivalent components on different platforms do not necessarily act alike.
  - The look and feel of each component was tied to the host operating system
- Almost all Swing components are lightweight except
  - JApplet, JFrame, JDialog, and JWindow

## Additional Swing Features

- Swing also provides
  - A wide variety of components (tables, trees, sliders, progress bars, internal frame, …)
  - Swing components can have *tooltips* placed over them.
  - Arbitrary keyboard events can be bound to components.
  - Additional debugging support.
  - Support for parsing and displaying HTML based information.

## Applets versus Applications

- Using Swing it is possible to create two different types of GUI programs
  - Standalone applications
    - Programs that are started from the command line
    - Code resides on the machine on which they are run
  - Applets
    - Programs run inside a web browser
    - Code is downloaded from a web server
    - JVM is contained inside the web browser
    - For security purposes Applets are normally prevented from doing certain things (for example opening files)
- For now we will write standalone applications

# JFrames

- A `JFrame` is a Window with all of the adornments added.
- A `JFrame` provides the basic building block for screen-oriented applications.

```
JFrame win = new JFrame( “title” );
```

# Creating a `JFrame`

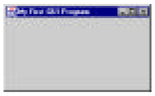
```
import javax.swing.*;

public class SwingFrame {
    public static void main( String args[] ) {
       JFrame win = new JFrame( "My First GUI Program" );

       win.show();
    }
} // SwingFrame
```

# `JFrame`

- Sizing a Frame
  - You can specify the size
    - Height and width given in pixels
    - The size of a pixel will vary based on the resolution of the device on which the frame is rendered
  - The method, `pack()`, will set the size of the frame automatically based on the size of the components contained in the content pane
    - Note that pack does not look at the title bar

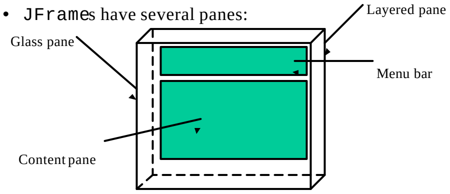
## Creating a `JFrame`

```
import javax.swing.*;

public class SwingFrame {
    public static void main( String args[] ) {
       JFrame win = new JFrame( "My First GUI Program" );

       win.setSize( 250, 150 );
       win.show();
    }
} // SwingFrame
```



## `JFrame`

- `JFrame`s have several panes:



- The content pane is where the components will be placed
- The entire collection of panes is called the `RootPane`

## Swing Components

- JComponent
  - JComboBox, JLabel, JList, JMenuBar, JPanel, JPopupMenu, JScrollBar, JScrollPane, JTable, JTree, JInternalFrame, JOptionPane, JProgressBar, JRootPane, JSeparator, JSlider, JSplitPane, JTabbedPane, JToolBar, JToolTip, Jviewport, JColorChooser, JTextComponent, …

# JLabels

- `JLabels` are components that you can put text into.
- When creating a label you can specify the initial value and the alignment you wish to use within the label.
- You can use `getText()` and `setText()` to get and change the value of the label.

```
lbl = new JLabel( ″text", JLabel.RIGHT ) ;
```

# Hello World

```
import javax.swing.*;

public class SwingFrame {
    public static void main( String args[] ) {
       JFrame win = new JFrame( "My First GUI Program" );

       JLabel label = new JLabel( "Hello World" );

       win.getContentPane().add( label );

       win.pack();
       win.show();
    }
} // SwingFrame
```

# JButtons

- `JButton` extends `Component` , displays a string and delivers an `ActionEvent` for each mouse click.
- Normally buttons are displayed with a border
- In addition to text, `JButtons` can also display icons

```
button = new JButton( ″text“ ) ;
```

## Buttons

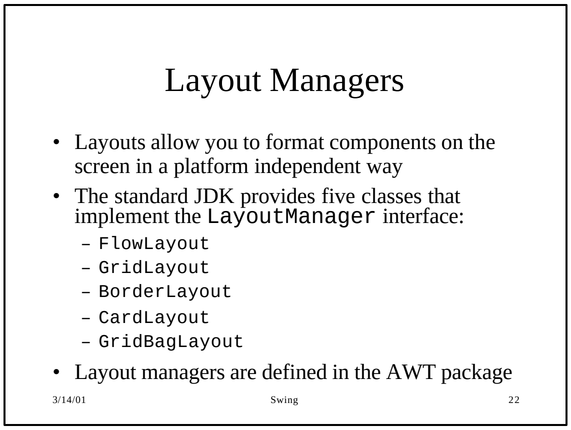
```
import javax.swing.*;

public class SwingFrame {
    public static void main( String args[] ) {
        JFrame win = new JFrame( "My First GUI Program" );

        JButton button = new JButton( "Click Me!!" );

        win.getContentPane().add( button );

        win.pack();
        win.show();
    }
} // SwingFrame
```

## Layout Manager

- Layout Manager
  - An interface that defines methods for positioning and sizing objects within a container. Java defines several default implementations of `LayoutManager`.
- Geometrical placement in a Container is controlled by a `LayoutManager` object

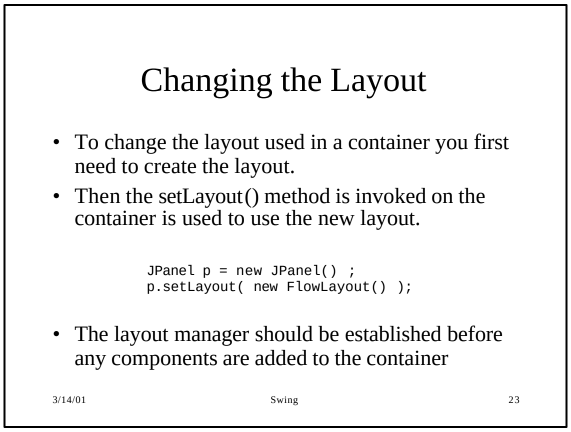
## Components, Containers, and Layout Managers

- Containers may contain components (which means containers can contain containers!!).
- All containers come equipped with a layout manager which positions and shapes (lays out) the container's components.
- Much of the action in the AWT occurs between components, containers, and their layout managers.



## Layout Managers

- Layouts allow you to format components on the screen in a platform independent way
- The standard JDK provides five classes that implement the `LayoutManager` interface:
  - `FlowLayout`
  - `GridLayout`
  - `BorderLayout`
  - `CardLayout`
  - `GridBagLayout`
- Layout managers are defined in the AWT package

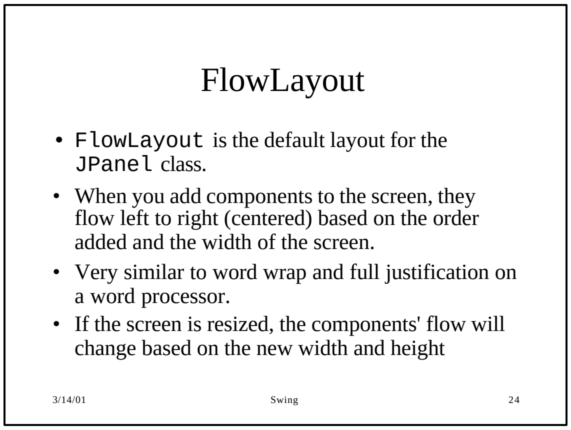


## Changing the Layout

- To change the layout used in a container you first need to create the layout.
- Then the setLayout() method is invoked on the container is used to use the new layout.

```
JPanel p = new JPanel() ;
p.setLayout( new FlowLayout() );
```

- The layout manager should be established before any components are added to the container



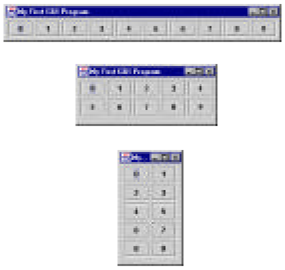
## FlowLayout

- `FlowLayout` is the default layout for the `JPanel` class.
- When you add components to the screen, they flow left to right (centered) based on the order added and the width of the screen.
- Very similar to word wrap and full justification on a word processor.
- If the screen is resized, the components' flow will change based on the new width and height

# Flow Layout

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

public class SwingFrame {
    public static void main( String args[] ) {
       JFrame win = new JFrame( "My First GUI Program" );

       win.getContentPane().setLayout( new FlowLayout() );

       for ( int i = 0; i < 10; i++ )
           win.getContentPane().add(
              new JButton( String.valueOf( i ) ) );

       win.pack();
       win.show();
    }
} // SwingFrame
```

# FlowLayout

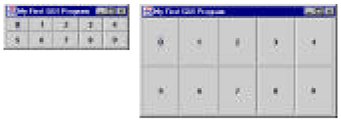


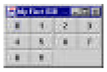
# GridLayout

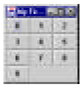
- The GridLayout arranges components in rows and columns.
  - If the number of rows is specified, the number of columns will be set to the number of components divided by the rows
  - If the number of columns is specified, the number of rows will be set to the number of components divided by the columns
  - Specifying the number of columns affects the layout only when the number of rows is set to zero.
- The order in which you add things is relevant.

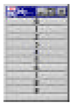
# GridLayout

gridLayout( 2, 4 )









gridLayout( 0, 4 ) gridLayout( 4, 4 ) gridLayout( 10, 10 )

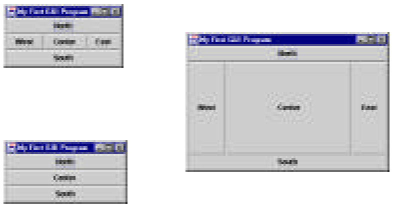
# BorderLayout

- `BorderLayout` provides 5 areas to hold components. These are named after the four different borders of the screen, North, South, East, West, and Center.
- When a Component is added to the layout, you must specify which area to place it in. The order in which components is not important.
- The center area will always be resized to be as large as possible

# BorderLayout

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

public class SwingFrame {
    public static void main( String args[] ) {
       JFrame win = new JFrame( "My First GUI Program" );
       Container content = win.getContentPane();

       content.setLayout( new BorderLayout() );
       content.add( "North", new JButton( "North" ) );
       content.add( "South", new JButton( "South" ) );
       content.add( "East", new JButton( "East" ) );
       content.add( "West", new JButton( "West" ) );
       content.add( "South", new JButton( "South" ) );
       content.add( "Center", new JButton( "Center" ) );

       win.pack();  win.show();
    }
} // SwingFrame
```

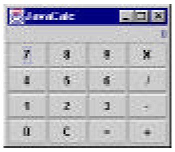
# BorderLayout



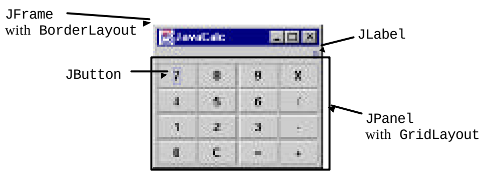
# Containers

- A `JFrame` is not the only type of container that you can use in Swing
- The subclasses of `Container` are:
  - `JPanel`
  - `JWindow`
  - `JApplet`
- Window is subclassed as follows:
  - `JDialog`
  - `JFrame`

# A Simple 4 Function Calculator



# Swing Components



JFrame
with BorderLayout

JLabel

JButton

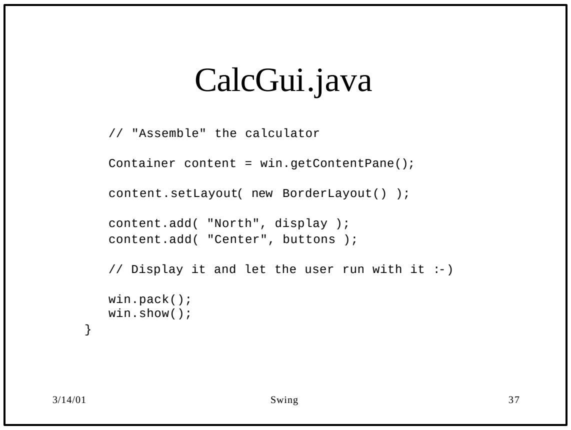
JPanel
with GridLayout

# CalcGui.java

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

public class CalcGui implements {
    // Labels for the buttons
    private static final String labels = "789X456/123-0C=+";

    private static final int NUMROWS = 4;
    private static final int NUMCOLS = 4;

    private JLabel display;  // The display

    public CalcGui( String name ) {
       // A Frame for the calculator

       JFrame win = new JFrame(name);
```

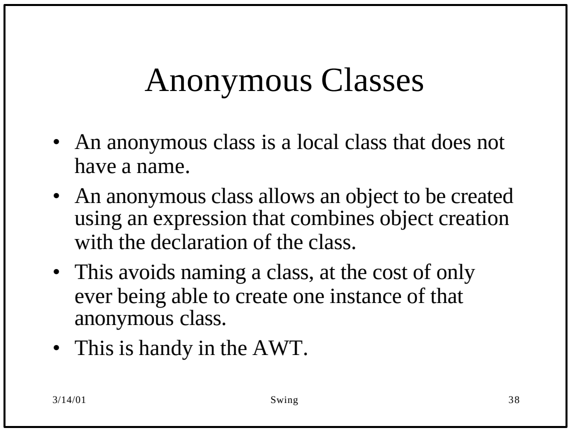
# CalcGui.java

```
// Create the button panel

JPanel buttons = new JPanel();
buttons.setLayout(new GridLayout(NUMROWS, NUMCOLS));

JButton b;

for ( int i = 0 ; i < labels.length() ; i++ ) {
    b = new JButton( labels.substring( i, i + 1 ) );
    buttons.add( b );
}

// Create the display

display = new JLabel( "0", JLabel.RIGHT )
```



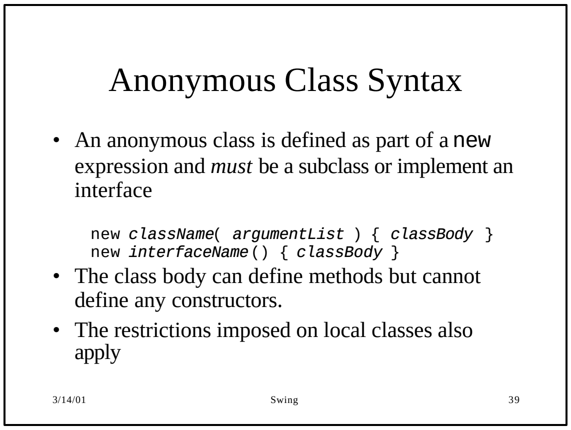
# CalcGui.java

```
    // "Assemble" the calculator

    Container content = win.getContentPane();

    content.setLayout( new BorderLayout() );

    content.add( "North", display );
    content.add( "Center", buttons );

    // Display it and let the user run with it :-)

    win.pack();
    win.show();
}
```



# Anonymous Classes

- An anonymous class is a local class that does not have a name.
- An anonymous class allows an object to be created using an expression that combines object creation with the declaration of the class.
- This avoids naming a class, at the cost of only ever being able to create one instance of that anonymous class.
- This is handy in the AWT.

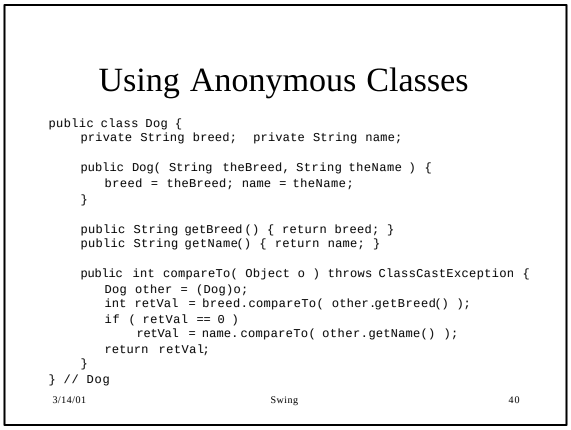


# Anonymous Class Syntax

- An anonymous class is defined as part of a `new` expression and *must* be a subclass or implement an interface

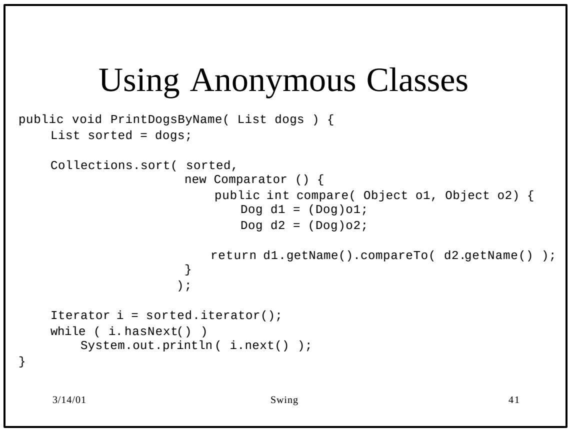
```
new className( argumentList ) { classBody }
new interfaceName() { classBody }
```

- The class body can define methods but cannot define any constructors.
- The restrictions imposed on local classes also apply



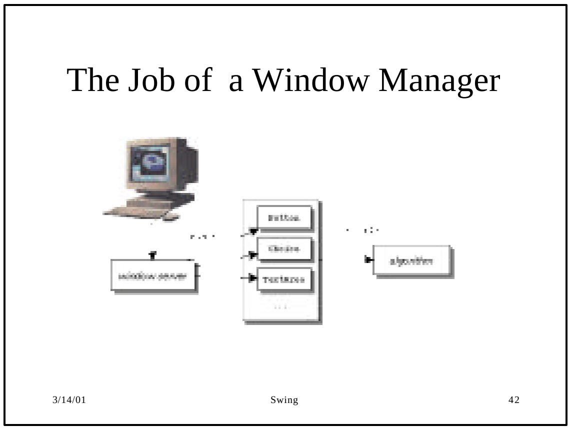
## Using Anonymous Classes

```
public class Dog {
    private String breed;  private String name;

    public Dog( String theBreed, String theName ) {
       breed = theBreed; name = theName;
    }

    public String getBreed() { return breed; }
    public String getName() { return name; }

    public int compareTo( Object o ) throws ClassCastException {
       Dog other = (Dog)o;
       int retVal = breed.compareTo( other.getBreed() );
       if ( retVal == 0 )
           retVal = name.compareTo( other.getName() );
       return retVal;
    }
} // Dog
```



## Using Anonymous Classes

```
public void PrintDogsByName( List dogs ) {
    List sorted = dogs;

    Collections.sort( sorted,
                     new Comparator () {
                         public int compare( Object o1, Object o2) {
                            Dog d1 = (Dog)o1;
                            Dog d2 = (Dog)o2;

                        return d1.getName().compareTo( d2.getName() );
                     }
                    );

    Iterator i = sorted.iterator();
    while ( i.hasNext() )
        System.out.println( i.next() );
}
```



## The Job of  a Window Manager

# Event Driven Programming

- Programs respond to events that are generated *outside* the control of the program
  - User types a key
  - The left mouse button is pressed
  - A CD is removed from the CD drive
- When an event occurs, it is handled by an event handler
- Event driven programming involves writing the handlers and arranging for the handler to be notified when certain events occur

# Event Handling

- Events are represented by objects that gives information about the event and identifies the event source
  - Event sources are typically components, but other kinds of objects can also be event sources
- A *listener* is an object that wants to be notified when a particular event occurs
  - An event source can have multiple listeners registered on it
  - A single listener can register with multiple event sources
- In order for an object to be notified when a particular event occurs, the object

# Swing Listeners

| | |
|---|---|
| Action | Listener Type |
| User clicks a button, presses return while typing in a text filed, or chooses a menu item | `ActionListener` |
| Users closes a frame (main window) | `WindowListener` |
| User presses a mouse button while the cursor is over a component | `MouseListener` |
| User moves the move over a component | `MouseMotionListener` |
| A component becomes visible | `ComponentListener` |
| A component gets the keyboard focus | `FocusListener` |
| A table of list selection changes | `ListSelectionListener` |

## Window Closing

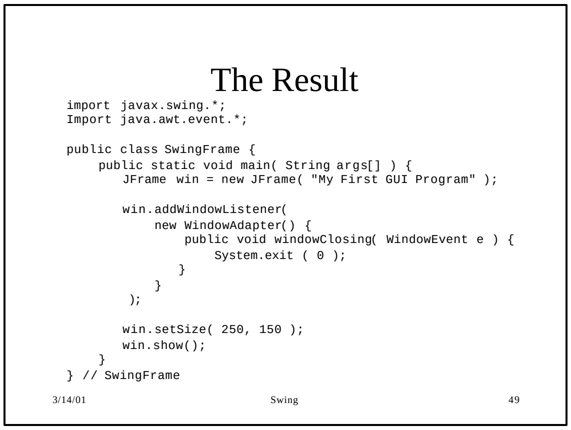
- A very common event directed towards a window is a *close* event
  - The default behavior is to simply hide the `JFrame` when the user closes the window
- Normally we would want the program to terminate when the user closes the main window
- Two steps required to accomplish this
  - Write an event handler for the close event that will terminate the program
  - Register the handler with the appropriate event source

## `WindowListener`

- The `WindowListener` interface
  - `void windowActivated(WindowEvent e) ;`
  - `void windowClosed(WindowEvent e) ;`
  - `void windowClosing(WindowEvent e) ;`
  - `void windowDeactivated(WindowEvent e) ;`
  - `void windowDeiconified(WindowEvent e) ;`
  - `void windowIconified(WindowEvent e) ;`
  - `void windowOpened(WindowEvent e) ;`
- A class that implements `WindowListener` must implement all of these methods!

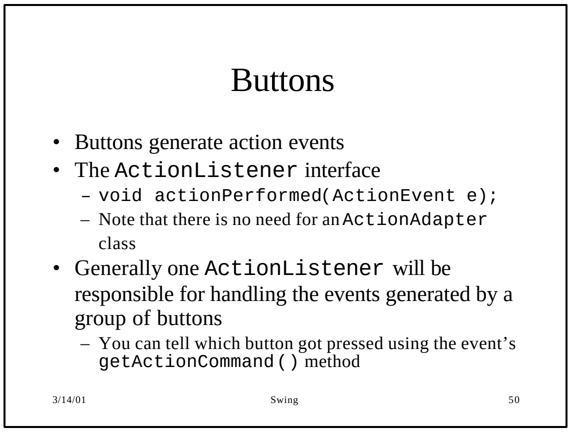
## `WindowAdapter`

- A class that implements the `WindowListener` interface
  - The methods in this class are empty. The class exists as convenience for creating listener objects.
- To use the `WindowAdapter` class:
  - Extend this class to create a `WindowEvent` listener
  - Override the methods for the events of interest
  - Create a listener object using the extended class and then register it with a Window using the window's `addWindowListener()` method.
- When the window's status changes the appropriate method in the listener object is invoked, and the `WindowEvent` is passed to it



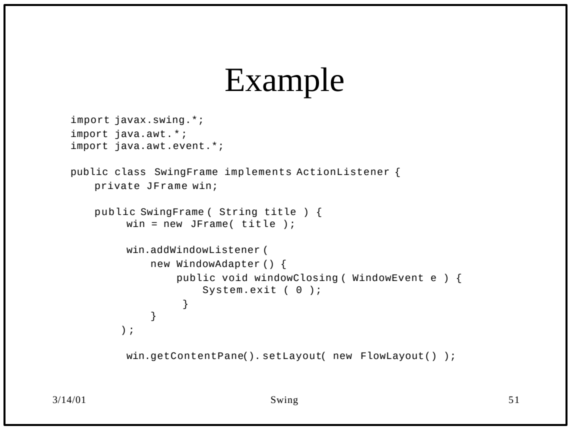
## The Result

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

public class SwingFrame {
    public static void main( String args[] ) {
       JFrame win = new JFrame( "My First GUI Program" );

       win.addWindowListener(
           new WindowAdapter() {
               public void windowClosing( WindowEvent e ) {
                   System.exit ( 0 );
               }
           }
        );

       win.setSize( 250, 150 );
       win.show();
    }
} // SwingFrame
```



## Buttons

- Buttons generate action events
- The `ActionListener` interface
  - `void actionPerformed(ActionEvent e);`
  - Note that there is no need for an `ActionAdapter` class
- Generally one `ActionListener` will be responsible for handling the events generated by a group of buttons
  - You can tell which button got pressed using the event's `getActionCommand()` method



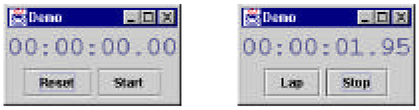
## Example

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

public class SwingFrame implements ActionListener {
    private JFrame win;

    public SwingFrame( String title ) {
        win = new JFrame( title );

        win.addWindowListener(
            new WindowAdapter() {
                public void windowClosing( WindowEvent e ) {
                    System.exit ( 0 );
                 }
            }
       );

        win.getContentPane().setLayout( new FlowLayout() );
```

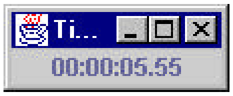
# Example

```
        for ( int i = 0; i < 10; i++ ) {
            JButton b = new JButton( String.valueOf( i ) );
            b.addActionListener( this );
            win.getContentPane().add( b );
        }

        win.pack();
        win.show();
    }

    public void actionPerformed( ActionEvent e ) {
        System.out.println( "Button " +
                             e.getActionCommand() +
                             " was pressed " );
    }

    public static void main( String args[] ) {
        SwingFrame f = new SwingFrame( "My First GUI" );
    }

} // SwingFrame
```

# StopWatch.java



# TimerLabel.java



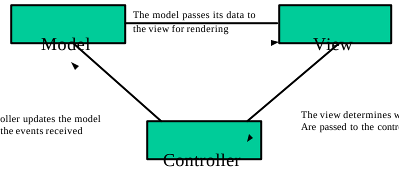
# GUI Program Design

- The GUI provides a view of the program, it is clearly not the program
- Making the GUI code as independent of the program code is a good strategy
  - Changes in the program do not necessarily change the GUI
  - Different GUIs can be developed for the same program
  - Debugging and maintaining both the GUI and the program code is easier

# Model-View-Controller

- The MVC pattern is commonly used to develop applications that have a GUI component
- Consists of three parts
  - Model
    - The program
  - View
    - The GUI
  - Controller
    - The event handling mechanism

# MVC

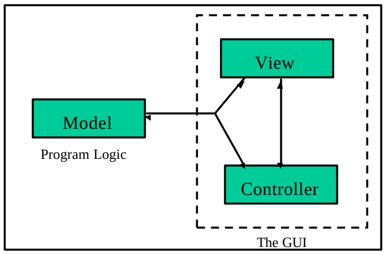


The model passes its data to the view for rendering

The controller updates the model Based on the events received

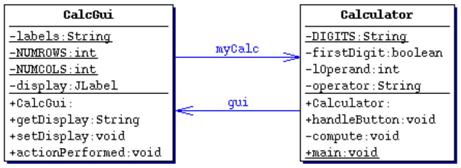
The view determines which events Are passed to the controller

## MVC in Swing



The Application

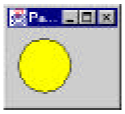
## A Simple 4 Function Calculator



## Painting

- When a GUI needs to change its visual appearance it performs a paint operation
- Swing components generally repaint themselves as needed
- Painting code executes on the event-dispatching thread
  - If painting takes a long time, no events will be handled during that time

## Example



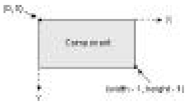
```
import javax.swing.*; import java.awt.*;

public class Painting extends JPanel {
    public Painting() {}

    public void paintComponent(Graphics g) {
       super.paintComponent(g);
       g.setColor( Color.yellow ); g.fillOval( 10,10,50,50 );
       g.setColor( Color.black ); g.drawOval( 10,10,50,50 );
    }

    public static void main( String args[] ) {
       JFrame win = new JFrame( "Painting" );
       win.setSize(100, 100);
       win.getContentPane().add( new Painting() );
       win.show();
    }}
```

## The `Graphics` Object

- The Graphics object both a context for painting and methods for performing the painting.
- The graphics context consists of state such as the current painting color, the current font, and the current painting area
  - The color and font are initialized to the foreground color and font of the component just before the invocation of paintComponent
- You can ignore the current painting area, if you like

## The Coordinate System

- Each component has its own integer coordinate system
  - Ranging from (0, 0) to (width - 1, height - 1)
  - Each unit represents the size of one pixel

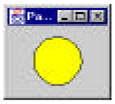


## Borders

- You must take into account the component's size and the size of the component's border
  - A border that paints a one-pixel line around a component changes the top leftmost corner from (0,0) to (1,1) and reduces the width and the height of the painting area by two pixels each
- You get the width and height of a component using its `getWidth` and `getHeight` methods.
- To determine the border size, use the `getInsets` method.

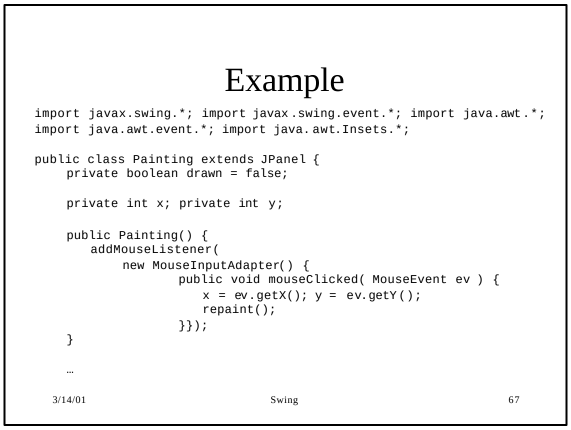
## Example

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

public class Painting extends JPanel {
    public Painting() {}

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

       Insets border = getInsets();
       int width = getWidth() - border.left - border.right;
       int height = getHeight() - border.top - border.bottom;

       int x = ( width / 2 ) - 25 + border.left;
       int y = ( height / 2 ) - 25 + border.top;

       g.setColor( Color.yellow ); g.fillOval( x, y, 50, 50 );
       g.setColor( Color.black ); g.drawOval( x, y, 50, 50 );
    }

} // Painting
```



## *Forcing* a Paint

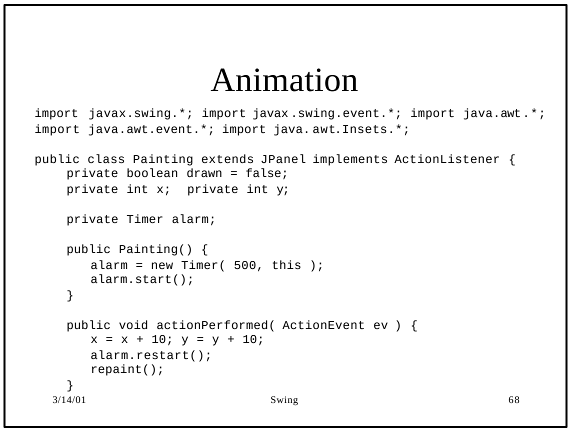
- The `repaint()` method schedules a paint operation for the specified component
  - A version of `repaint()` exists that allows you to specify the area that needs to be repainted
- Typically a component will invoke `repaint()` when it has done something to change its state



# Example

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

public class Painting extends JPanel {
    private boolean drawn = false;

    private int x; private int y;

    public Painting() {
       addMouseListener(
           new MouseInputAdapter() {
                  public void mouseClicked( MouseEvent ev ) {
                     x = ev.getX(); y = ev.getY();
                     repaint();
                  }});
    }

    …
```

3/14/01 Swing 67



# Animation

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

public class Painting extends JPanel implements ActionListener {
    private boolean drawn = false;
    private int x;  private int y;

    private Timer alarm;

    public Painting() {
       alarm = new Timer( 500, this );
       alarm.start();
    }

    public void actionPerformed( ActionEvent ev ) {
       x = x + 10; y = y + 10;
       alarm.restart();
       repaint();
    }
```

3/14/01 Swing 68