

Graphical User Interface (GUI)

Java's AWT and Swing APIs

Lecture 7

Dr. Tamer ABUHMED
Java Programming Course (SWE2023)
College of Computing

Outline

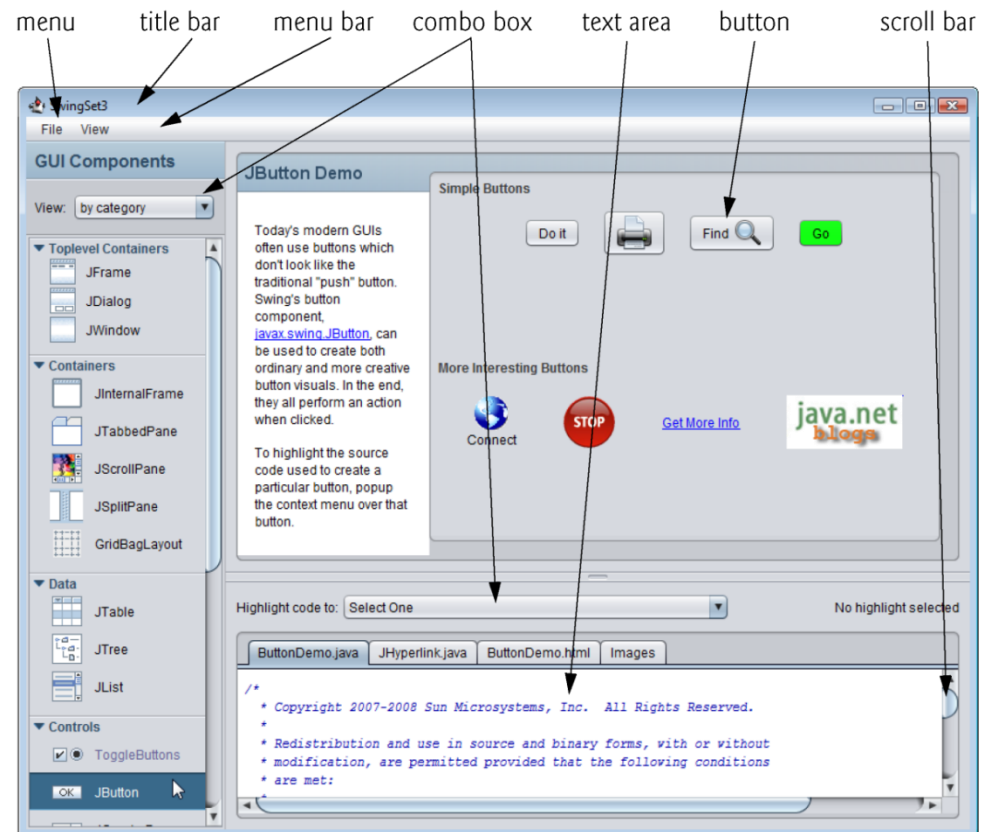


- Introduction to Java GUI
- AWT and Swing
- GUI Terminology
 - Swing Components
 - GUI Window: JFrame
 - GUI Component: JButton
- GUI Sizing and positioning
 - Containers and Layout
- Graphical Events
- Event Listeners
 - Action Events
 - Implementing a listener

Why Learn GUIs?



- Learn about *event-driven programming* techniques
- Practice learning and using a large, complex API
- A chance to see how it is designed and learn from it



Caution: There is here more than you can memorize.

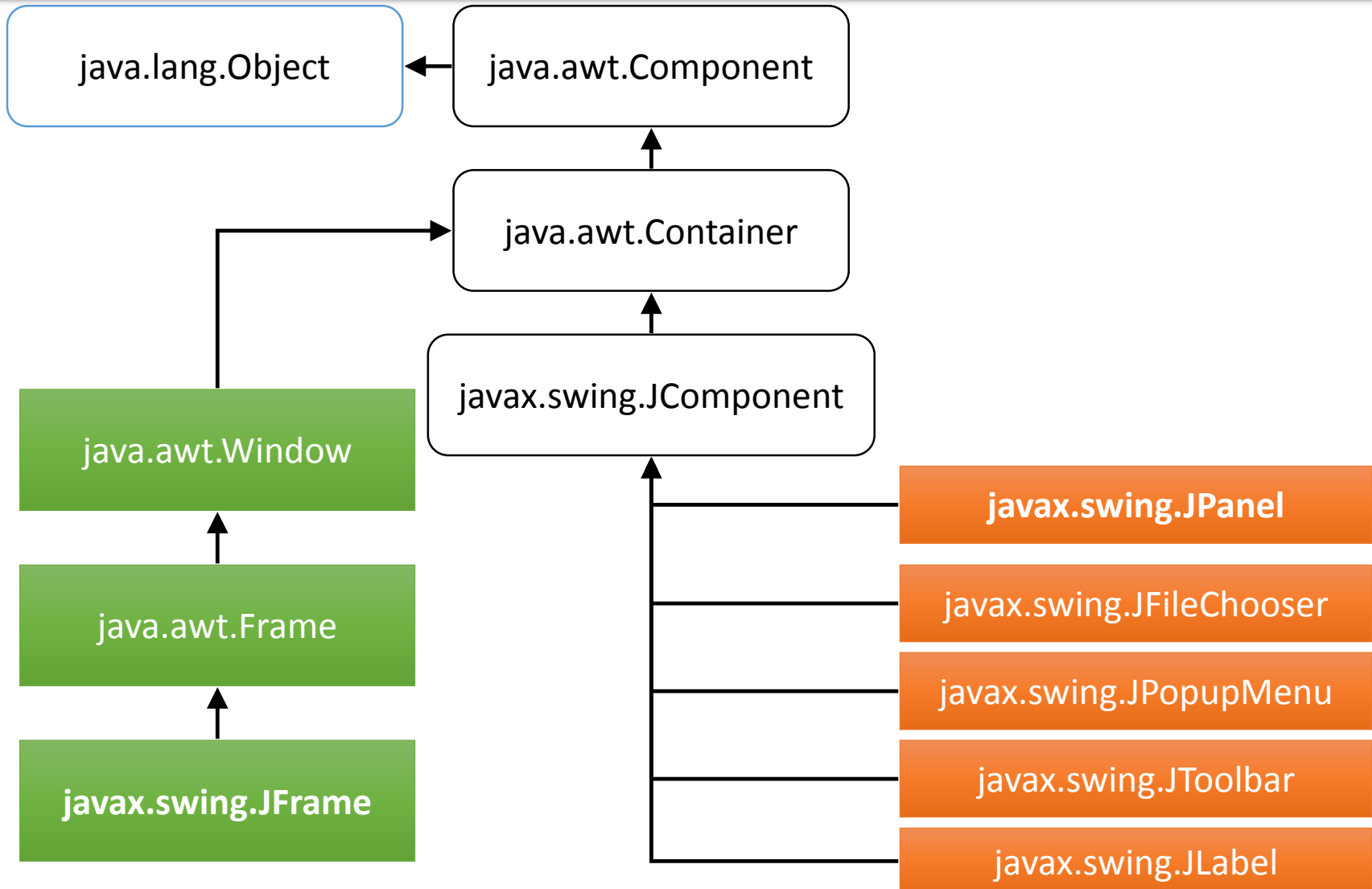
- Part of learning a large API is "letting go."
- You won't memorize it all; you will look things up as you need them.
- But you can learn the **fundamental** concepts and **general ideas**.

AWT and Swing



- Java provides two sets of components for GUI programming:
 - **AWT**: classes in the `java.awt` package (*JDK 1.0 - 1.1*)
 - **Swing**: classes in the `javax.swing` package (*JDK 1.2+*)
- AWT components depend on underlying system native GUI to handle their functionality. Thus, these components are often called **heavyweight** components.
- Swing is implemented entirely in Java and its components do not depend on system native GUI to handle their functionality. Thus, these components are often called **lightweight** components

Some AWT and Swing Classes



AWT: Pros and Cons



- Pros
 - Speed: native components speed performance.
 - Look and feel: AWT components more closely reflect the look and feel of the OS they run on.
- Cons
 - Portability: use of native peers creates platform specific limitations.
 - Features: AWT supports only the lowest common denominator—e.g. no tool tips or icons.

Swing: Pros and Cons

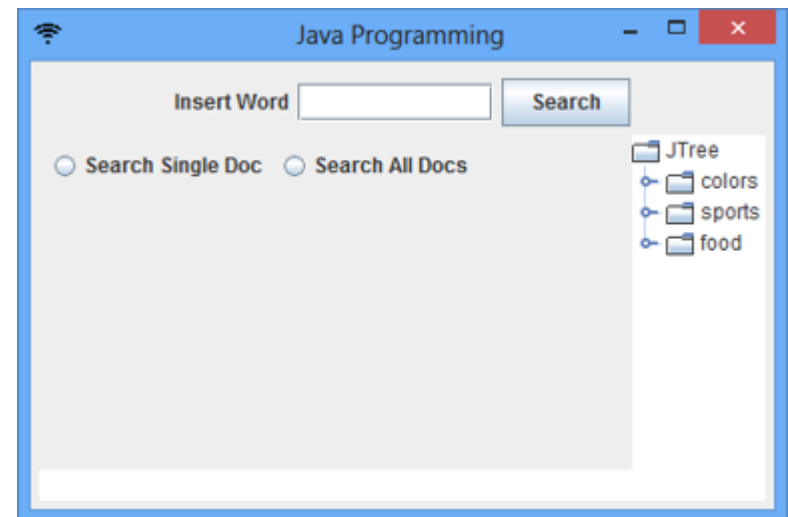
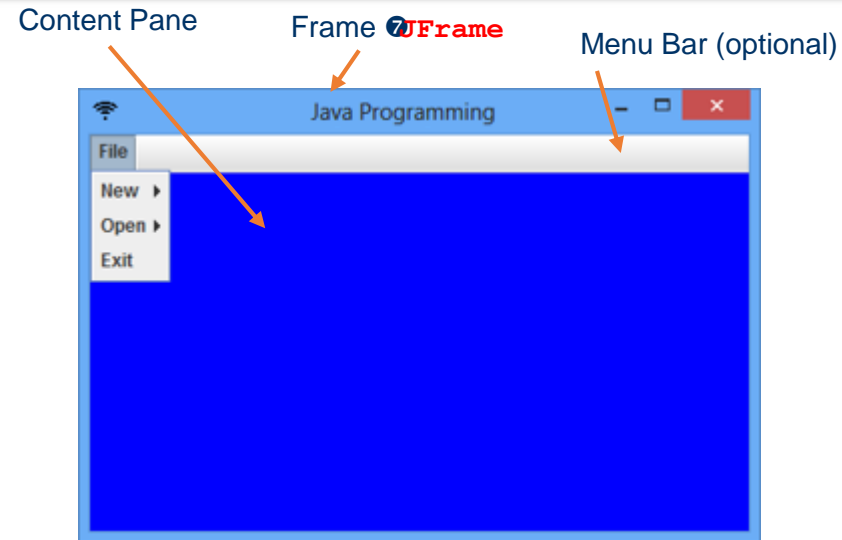


- Pros
 - Portability: Pure Java implementation.
 - Features: Not limited by native components.
 - Look and Feel: Pluggable look and feel. Components automatically have the look and feel of the OS their running on.
- Cons
 - Performance: Swing components handle their own painting (instead of using APIs like DirectX on Windows).
 - Look and Feel: May look slightly different than native components.

GUI Terminology



- **window**: A first-class citizen of the graphical desktop.
 - Also called a *top-level container*.
 - examples: **frame**, dialog box, applet
- **component**: A GUI widget that resides in a window.
 - Also called *controls* in many other languages.
 - examples: **button**, **text box**, **label**
- **container**: A logical grouping for storing components.
 - examples: **panel**, box



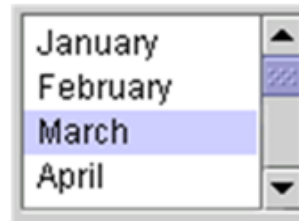
Swing Components



Buttons



Combo Box



List



TextField



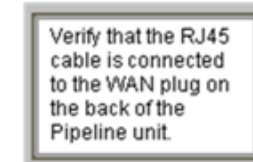
Slider



Menu



Label



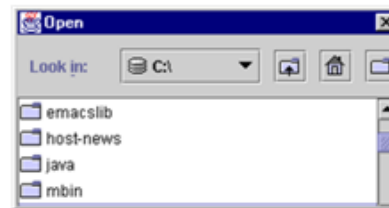
Text Area



Tool Tip



Progress Bar



File Chooser



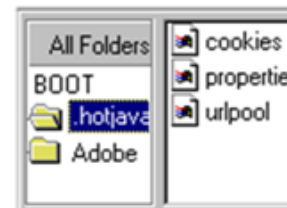
Color Chooser

First Na...	Last Name
Mark	Andrews
Tom	Ball
Alan	Chung
Jeff	Dinkins

Table



Tree



Split Pane



Tabbed Pane

Component Properties



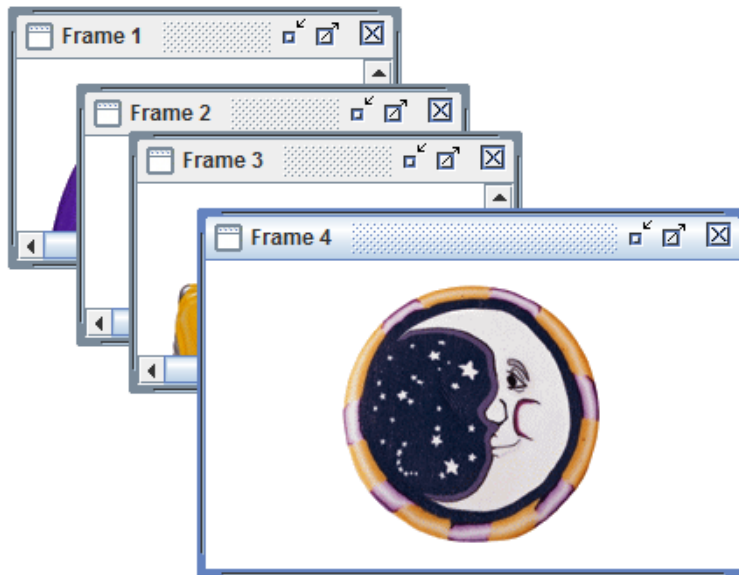
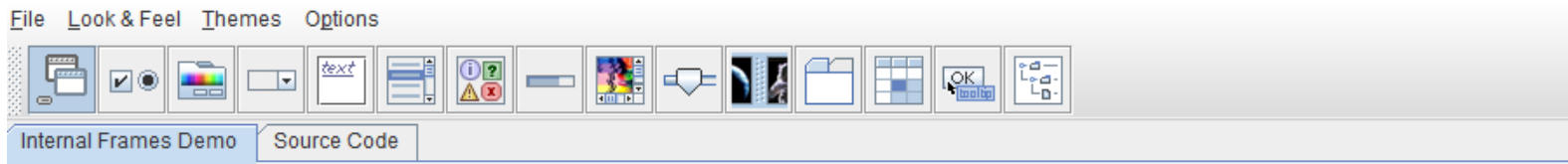
- Each component has a `get` (or `is`) accessor and a `set` modifier method.
 - examples: `getColor()`, `setFont()`, `setEnabled()`, `isVisible()`

name	type	description
background	Color	background color behind component
border	Border	border line around component
enabled	boolean	whether it can be interacted with
focusable	boolean	whether key text can be typed on it
font	Font	font used for text in component
foreground	Color	foreground color of component
height, width	int	component's current size in pixels
visible	boolean	whether component can be seen
tooltip text	String	text shown when hovering mouse
size, minimum / maximum / preferred size	Dimension	various sizes, size limits, or desired sizes that the component may take

Pluggable Look-and-Feel



The look-and-feel at swing is implemented in Java, but could **mimic** Windows, Motif, Classic, Aqua, etc.



Default Java Swing look-and-feel

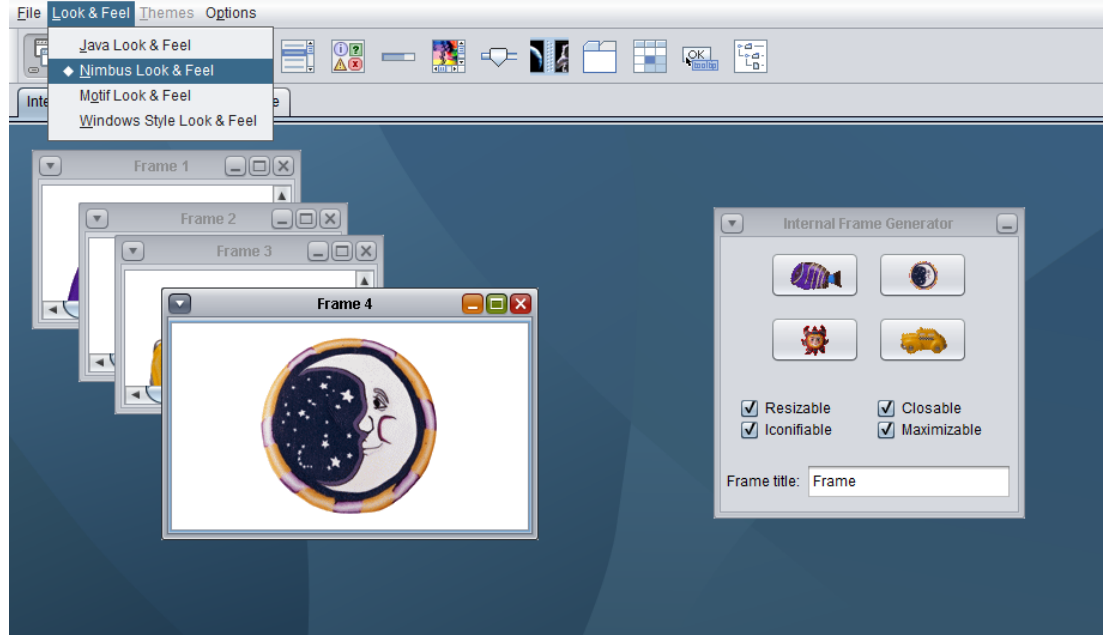
Pluggable Look-and-Feel



GUI Look-and-Feel can be assigned programmatically

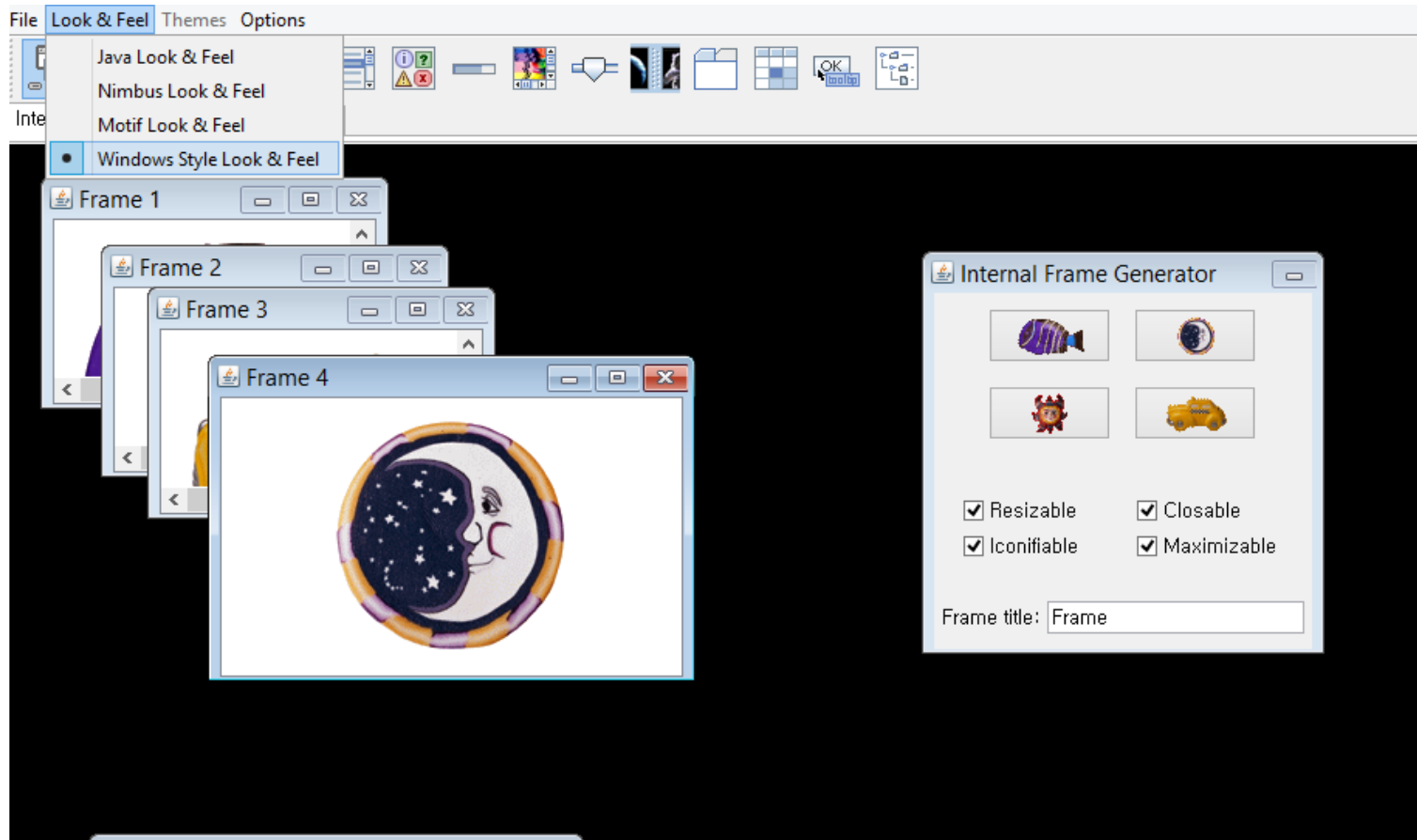
```
UIManager.setLookAndFeel (  
    "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```

```
UIManager.setLookAndFeel (  
    "javax.swing.plaf.metal.MetalLookAndFeel");
```



Nimbus Look-and-Feel

Pluggable Look-and-Feel



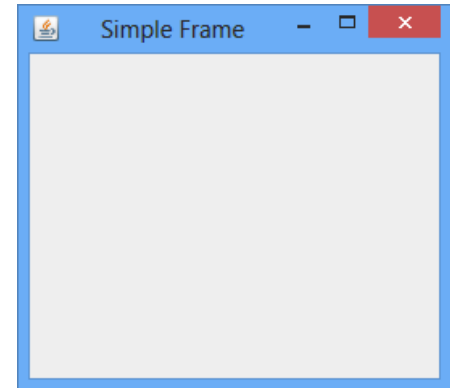
Windows Look-and-Feel

GUI Window: JFrame



a graphical window to hold other components

- `public JFrame()`
`public JFrame(String title)`
Creates a frame with an optional title.
 - Call `setVisible(true)` to make a frame appear on the screen after creating it.
- `public void add(Component comp)`
Places the given component or container inside the frame.



More JFrame



```
public void setDefaultCloseOperation(int op)
```

Makes the frame perform the given action when it closes.

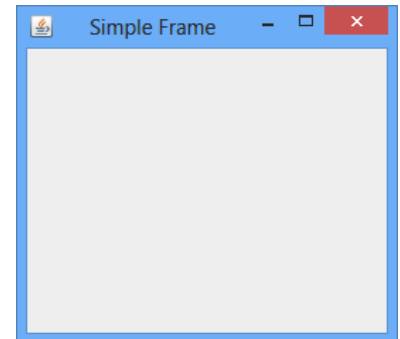
- Common value passed: `JFrame.EXIT_ON_CLOSE`
- If not set, the program will never exit even if the frame is closed.

```
• public void setSize(int width, int height)
```

Gives the frame a fixed size in pixels.

```
• public void pack()
```

Resizes the frame to fit the components inside it snugly.

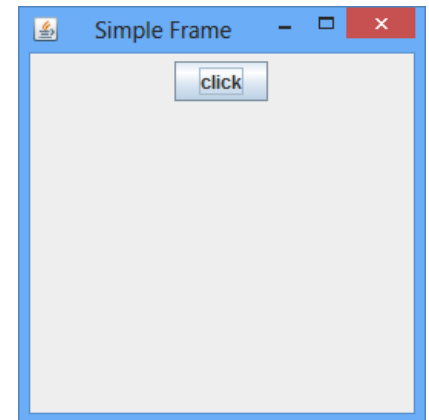


GUI Component: JButton



a clickable region for causing actions to occur

- `public JButton(String text)`
Creates a new button with the given string as its text.
- `public String getText()`
Returns the text showing on the button.
- `public void setText(String text)`
Sets button's text to be the given string.



GUI Example



```
import java.awt.*;           // Where is the other button?
import javax.swing.*;
```

Import Swing packages

```
public class GuiExample1 {
```

```
    public static void main(String[] args) {
```

```
        JFrame frame = new JFrame();
```

Create Frame

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        frame.setSize(new Dimension(300, 100));
```

```
        frame.setTitle("A frame");
```

Frame settings

```
        JButton button1 = new JButton();
```

```
        button1.setText("I'm a button.");
```

```
        button1.setBackground(Color.BLUE);
```

```
        frame.add(button1);
```

Create a JButton Component

```
        JButton button2 = new JButton();
```

```
        button2.setText("Click me!");
```

```
        button2.setBackground(Color.RED);
```

```
        frame.add(button2);
```

Add JButton to the Frame

```
        frame.setVisible(true);
```

```
    }
```

```
}
```



GUI Sizing and Positioning



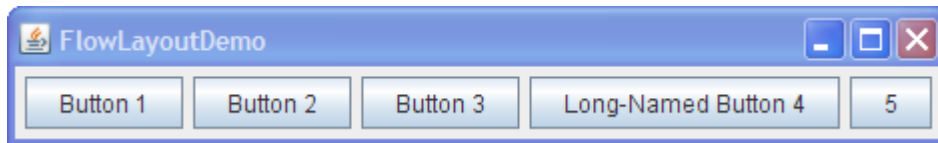
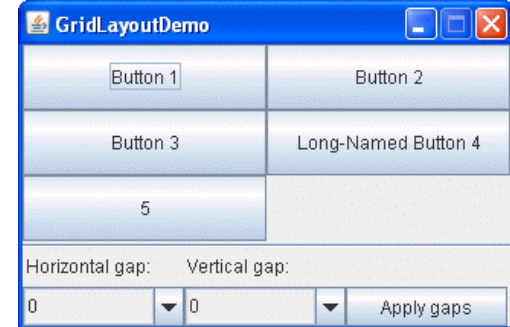
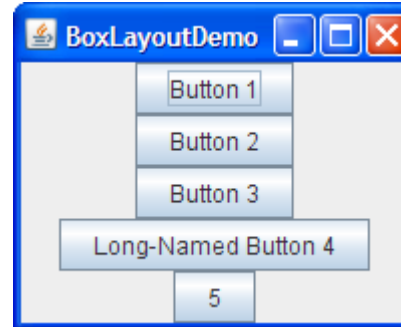
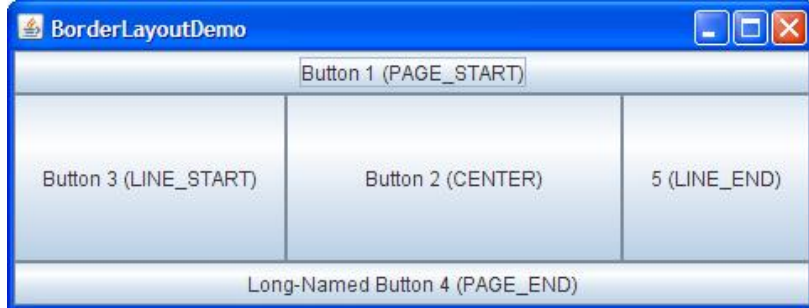
How does the programmer specify where each component appears, how big each component should be, and what the component should do if the window is resized / moved / maximized / etc.?

- **Absolute positioning** (C++, C#, others):
Programmer specifies exact pixel coordinates of every component.
 - "Put this button at (x=15, y=75) and make it 70x31 px in size."
- **Layout managers** (Java):
Objects that decide where to position each component based on some general rules or criteria.
 - "Put these four buttons into a 2x2 grid and put these text boxes in a horizontal flow in the south part of the frame."

Containers and Layout



- Place components in a *container*, add the container to a frame.
 - **container**: An object that stores components and governs their positions, sizes, and resizing behavior.



JFrame as Container



A `JFrame` is a container. Containers have these methods:

- `public void add(Component comp)`
`public void add(Component comp, Object info)`
Adds a component to the container, possibly giving extra information about where to place it.
- `public void remove(Component comp)`
- `public void setLayout(LayoutManager mgr)`
Uses the given layout manager to position components.
- `public void validate()`
Refreshes the layout (if it changes after the container is onscreen).

FlowLayout

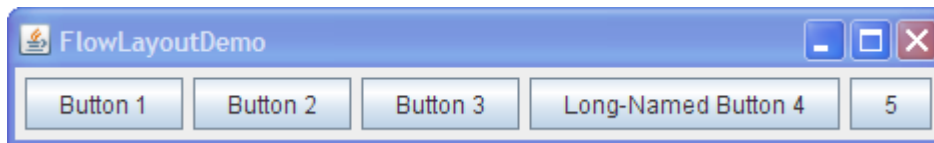


```
public FlowLayout()
```

- Treats container as a left-to-right, top-to-bottom "paragraph".
 - Components are given preferred size, horizontally and vertically.
 - Components are positioned in the order added.
 - If too long, components wrap around to the next line.

```
myFrame.setLayout(new FlowLayout());
```

```
myFrame.add(new JButton("Button 1"));
```



- The default layout for containers other than `JFrame` (seen later).

BorderLayout

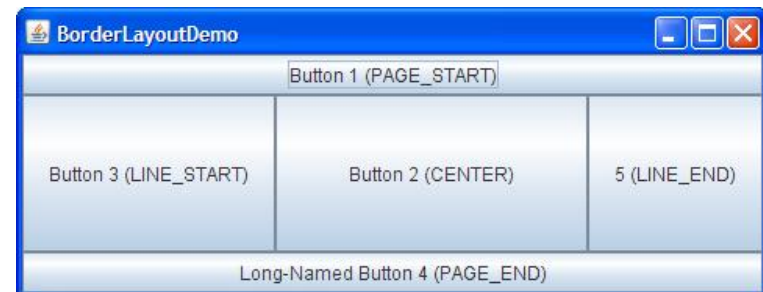


```
public BorderLayout()
```

- Divides container into five regions:
 - NORTH and SOUTH regions expand to fill region horizontally, and use the component's preferred size vertically.
 - WEST and EAST regions expand to fill region vertically, and use the component's preferred size horizontally.
 - CENTER uses all space not occupied by others.

```
myFrame.setLayout(new BorderLayout());  
myFrame.add(new JButton("Button 1"), BorderLayout.NORTH);
```

- This is the default layout for a JFrame.

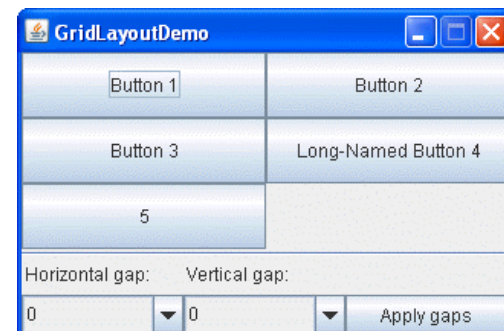


GridLayout



```
public GridLayout(int rows, int columns)
```

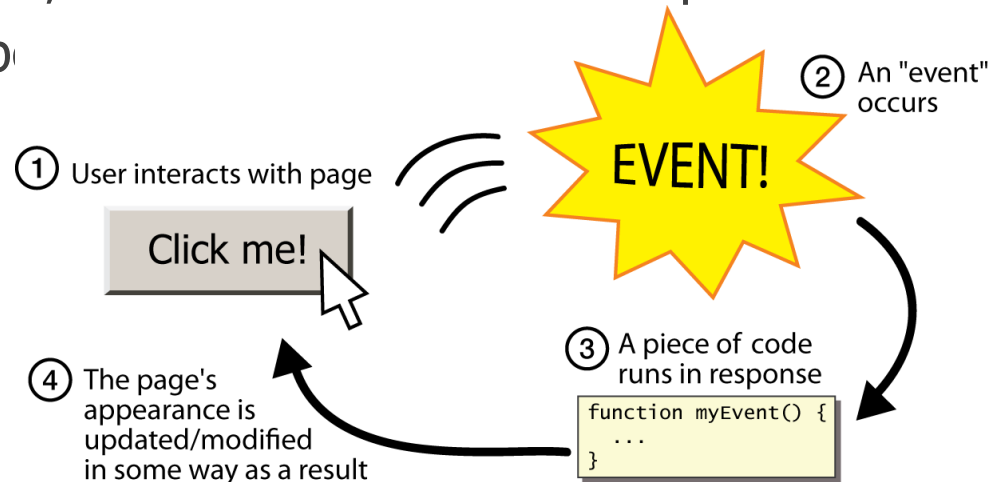
- Treats container as a grid of equally-sized rows and columns.
- Components are given equal horizontal / vertical size, disregarding preferred size.
- Can specify 0 rows or columns to indicate expansion in that direction as needed.



Graphical Events



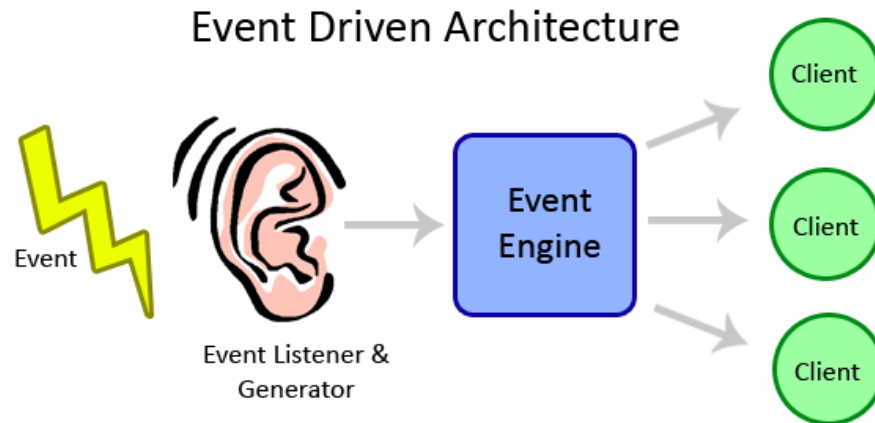
- **event:** An object that represents a user's interaction with a GUI component; can be "handled" to create interactive components.
- **listener:** An object that waits for events and responds to them.
 - To handle an event, attach a *listener* to a component.
 - The listener will be notified when an event occurs (e.g., button click).



Event-driven Programming



- **event-driven programming:** A style of coding where a program's overall flow of execution is dictated by events.
 - Rather than a central "main" method that drives execution, the program loads and waits for user input events.
 - As each event occurs, the program runs particular code to respond.
 - The overall flow of what code is executed is determined by the series of events in order.



Event Hierarchy

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

- EventObject
 - AWTEvent (AWT)
 - **ActionEvent**
 - TextEvent
 - ComponentEvent
 - FocusEvent
 - WindowEvent
 - InputEvent
 - KeyEvent
 - MouseEvent
- EventListener
 - AWTEventListener
 - **ActionListener**
 - TextListener
 - ComponentListener
 - FocusListener
 - WindowListener
 - KeyListener
 - MouseListener

Action Events



- **action event:** An action that has occurred on a GUI component.
 - The most common, general event type in Swing. Caused by:
 - button or menu clicks,
 - check box checking / unchecking,
 - pressing Enter in a text field, ...
 - Represented by a class named `ActionEvent`
 - Handled by objects that implement interface `ActionListener`



Implementing a listener



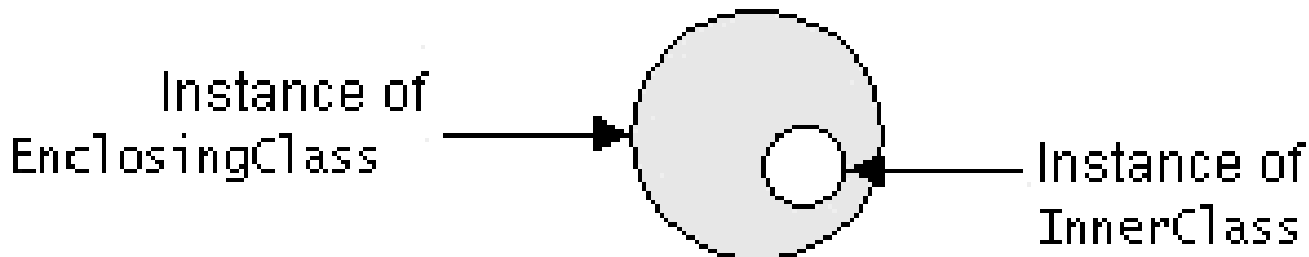
```
public class name implements ActionListener {  
    public void actionPerformed(ActionEvent event)  
    {  
        //code to handle the event;  
    }  
}
```

- JButton and other graphical components have this method:
 - `public void addActionListener(ActionListener al)`
Attaches the given listener to be notified of clicks and events that occur on this component.

Nested Classes



- **nested class:** A class defined inside of another class.
- Usefulness:
 - Nested classes are hidden from other classes (encapsulated).
 - Nested objects can access/modify the fields of their outer object.
- Event listeners are often defined as nested classes inside a GUI.



Nested Class Syntax



```
// enclosing outer class
public class Outer {
    ...

    // nested inner class
    private class Inner {
        ...
    }
}
```

- Only the outer class can see the nested class or make objects of it.

```
Outer.Inner innerObject = new Outer().new Inner();
```

- Each nested object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
- If necessary, can refer to outer object as **OuterClassName.this**

Static Inner Classes



```
// enclosing outer class
public class Outer {
    ...

    // non-nested static inner class
    public static class Inner {
        ...
    }
}
```

- Static inner classes are *not* associated with a particular outer object.
- They cannot see the fields of the enclosing class.
- *Usefulness:* Clients can refer to and instantiate static inner classes:
Outer.Inner name = new **Outer.Inner**(params) ;

GUI Event Example



```
public class MyGUI {
    private JFrame frame;
    private JButton stutter;
    private JTextField textfield;

    public MyGUI() {
        ...
        stutter.addActionListener(new StutterListener());
    }
    ...

    // When button is clicked, doubles the field's text.
    private class StutterListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            String text = textfield.getText();
            textfield.setText(text + text);
        }
    }
}
```


Buttons



- A *button* object is created from the class **JButton** and can be added to a **JFrame**
 - The argument to the **JButton** constructor is the string that appears on the button when it is displayed

```
JButton ExitButton = new  
    JButton("Click to end program.");  
frame.add(ExitButton);
```

Summary: Action Listeners and Action Events



- Clicking a button *fires an event*
- The event object is "sent" to another object called a listener
 - This means that a method in the listener object is invoked automatically
 - Furthermore, it is invoked with the event object as its argument
- In order to set up this relationship, a GUI program must do two things
 1. It must specify, for each button, what objects are its listeners, i.e., it must *register the listeners*
 2. It must *define the methods* that will be invoked automatically *when the event is sent to the listener*

Action Listeners and Action Events



```
EndingListener buttonEnd = new  
    EndingListener() ;  
ExitButton.addActionListener(buttonEnd) ;
```

- Above, a listener object named **buttonEnd** is created and registered as a listener for the button named **ExitButton**
 - Note that a **button** fires **events** known as *action events*, which are handled by listeners known as *action listeners*

Action Listeners and Action Events



- Different kinds of components require different kinds of listener classes to handle the events they fire
- An action listener is an object whose class implements the **ActionListener** interface
 - The **ActionListener** interface has one method heading that must be implemented
`public void actionPerformed(ActionEvent e)`

JButton Example

Buttons events handling by
ActionListener



```
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class ButtonFrame extends JFrame
{
    private JButton plainJButton; // button with just text
    private JButton fancyJButton; // button with icons

    // ButtonFrame adds JButtons to JFrame
    public ButtonFrame()
    {
        super( "Testing Buttons" );
        setLayout( new FlowLayout() ); // set frame layout

        plainJButton = new JButton( "Plain Button" ); // button with text
        add( plainJButton ); // add plainJButton to JFrame

        Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
        Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
        fancyJButton = new JButton( "Fancy Button", bug1 ); // set image
        fancyJButton.setRolloverIcon( bug2 ); // set rollover image
        add( fancyJButton ); // add fancyJButton to JFrame

        // create new ButtonHandler for button event handling
        ButtonHandler handler = new ButtonHandler();
        fancyJButton.addActionListener( handler );
        plainJButton.addActionListener( handler );
    } // end ButtonFrame constructor
```

```
// inner class for button event handling
private class ButtonHandler implements ActionListener
{
    // handle button event
    public void actionPerformed( ActionEvent event )
    {
        JOptionPane.showMessageDialog( ButtonFrame.this,
        String.format("You pressed: %s", event.getActionCommand() ) );
    } // end method actionPerformed
} // end private inner class ButtonHandler
} // end class ButtonFrame
```

```
import javax.swing.JFrame;

public class ButtonTest
{
    public static void main( String[] args )
    {
        ButtonFrame buttonFrame = new ButtonFrame(); // create Frame
        buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        buttonFrame.setSize( 300, 200 ); // set frame size
        buttonFrame.setVisible( true ); // display frame

    } // end main
} // end class ButtonTest
```

Summary



- Introduction to Java GUI
- AWT and Swing
- GUI Terminology
 - Swing Components
 - GUI Window: JFrame
 - GUI Component: JButton
- GUI Sizing and positioning
 - Containers and Layout
- Graphical Events
- Event Listeners
 - Action Events
 - Implementing a listener