Chapter 12 Part 1: GUI Programming (Simplified)

Yepang LIU (刘烨庞)
liuyp1@sustech.edu.cn

Objectives

- GUI and its brief history
- Build simple GUIs with containers and components
- Event handling
- Layout management

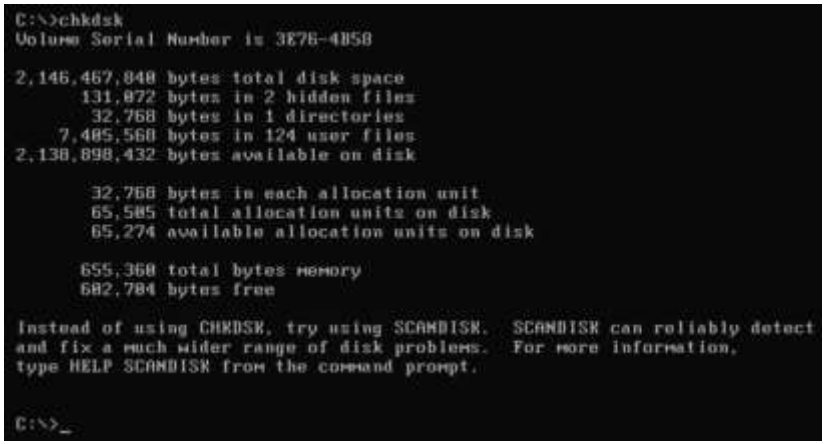
What is GUI?

- The Graphical User Interface (GUI, 图形用户界面), is a type of user interface that allows users to interact with electronic devices through graphical icons and visual indicators.

Windows 10

GUI vs. CLI

- Before GUI became popular, text-based Command-Line Interface (CLI, 命令行界面) was widely-used (mainly in 1970s and 1980s).
- Because CLIs consume little resources, they are still available in modern computers with GUIs and are widely-used by professionals.



MS-DOS

Java GUI History

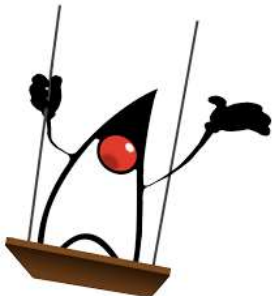
- Abstract Window Toolkit (AWT)
  - JDK 1.0 (1995)
  - Most of AWT's UI components have become obsolete
- Swing
  - JDK 1.2 (1997)
  - Enhancement of AWT
- JavaFX
  - JDK 8 (2008), replacement to Swing
  - Actively maintained and expected to grow in future

Java GUI Programming APIs

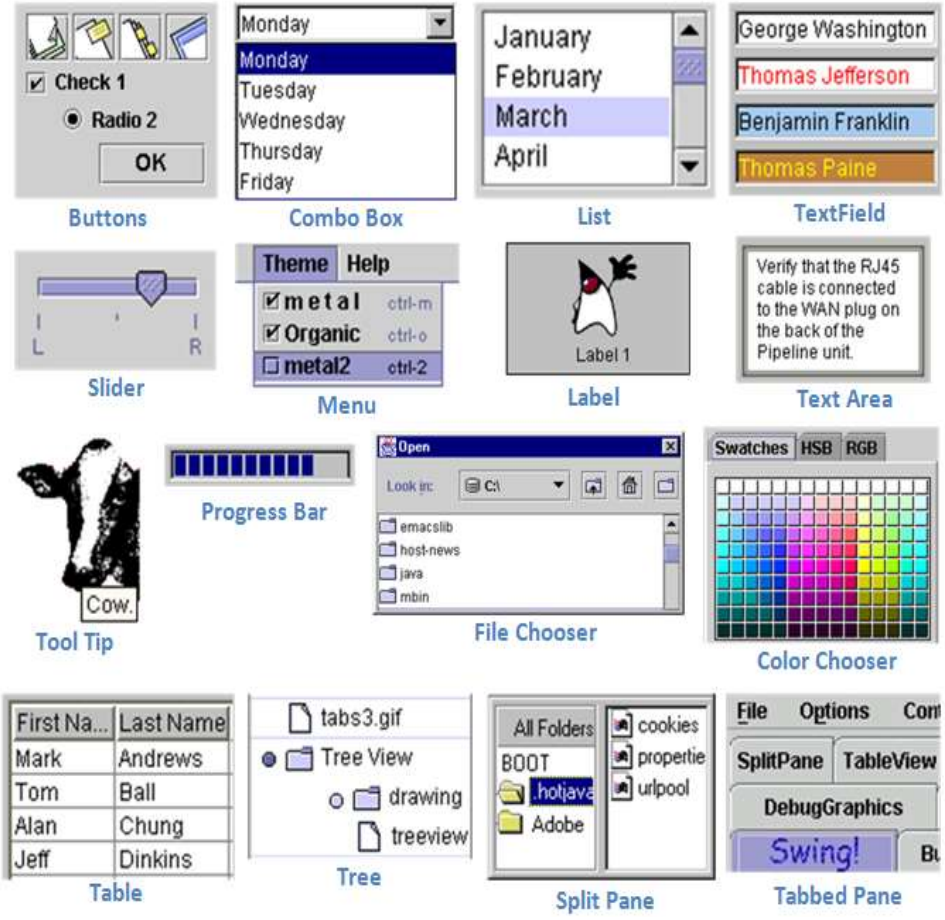
- **AWT** (Abstract Windowing Toolkit): introduced in JDK 1.0
- AWT components are **platform-dependent**. Their creation relies on the operating system's high-level user interface module.
- AWT contains 12 packages of 370 classes (Swing and FX are more complex, 650+ classes)
  - They are developed by expert programmers with advanced design patterns.
  - Writing your own graphics classes (re-inventing the wheels) is mission impossible!

https://www.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html

Java GUI Programming APIs

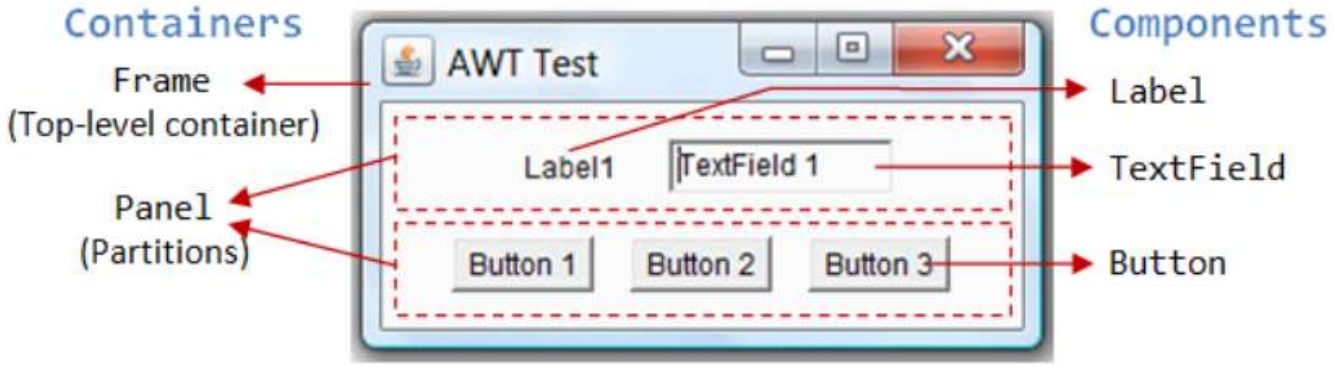


- **Swing**, introduced in 1997 after the release of JDK 1.1, provides a much more comprehensive set of UI widgets than AWT
- Unlike AWT's UI widgets, Swing's are not implemented by platform-specific code. They are written entirely in Java and **platform-independent**.
- **Pluggable look and feel:** Swing component can have the native platform's "look and feel" or a cross-platform look and feel (the "Java Look and Feel")



Java GUI Core Concepts

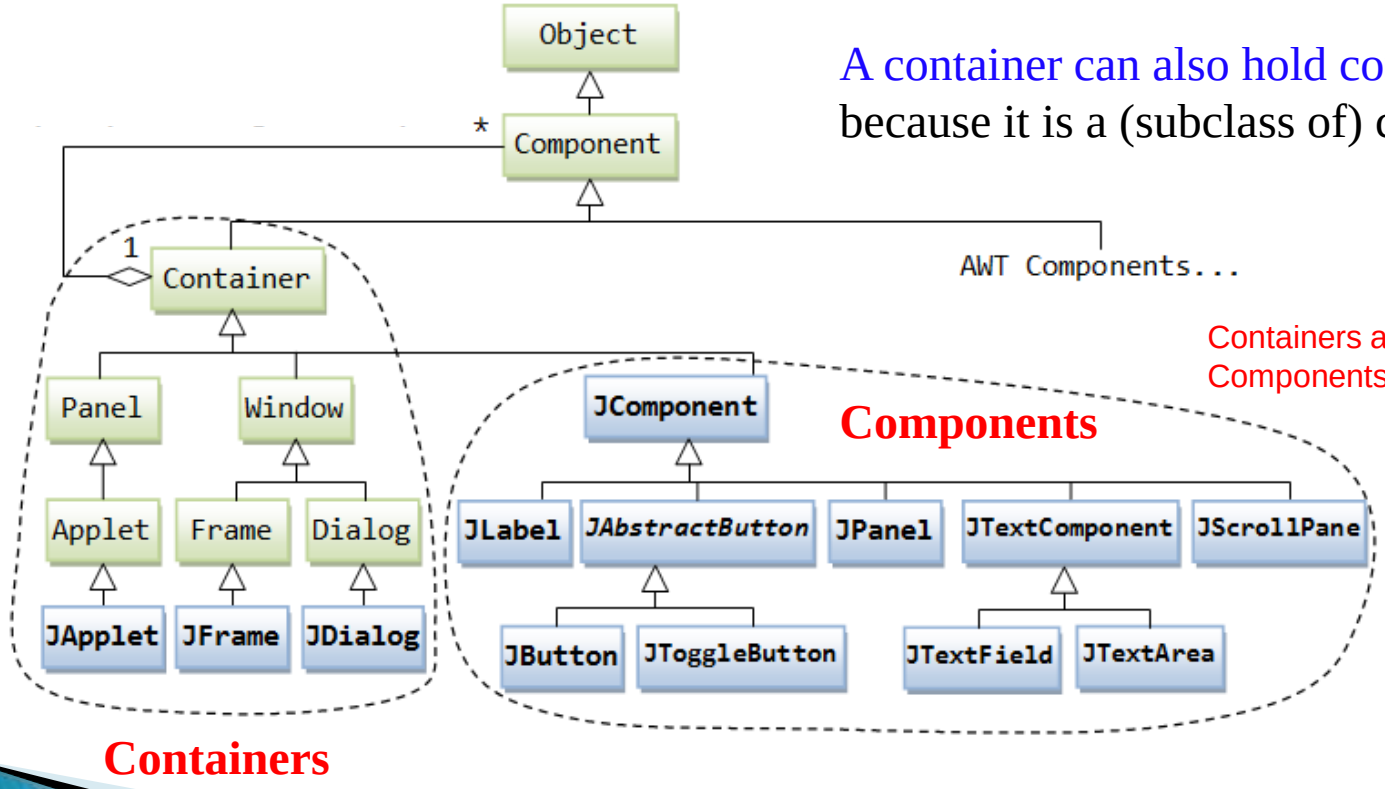
- **Component** (组件): Components are elementary GUI entities, such as Button, Label, and TextField.
- **Container** (容器): used to hold components in a specific layout
- **Event handling** (事件处理): decides what should happen if an event occurs (e.g., a button is clicked)



https://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html

Java GUI Class Hierarchy

- There are two groups of classes (in package `javax.swing`): **containers** and **components**. A container is used to hold components.

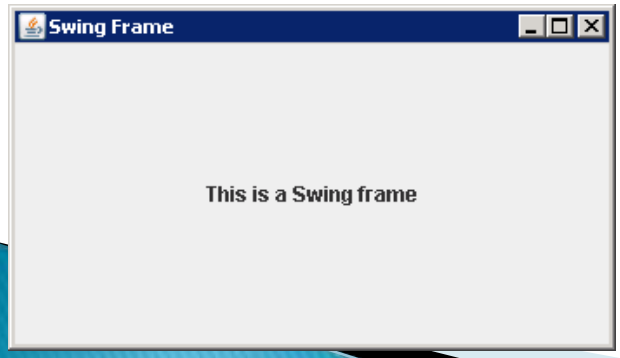


A container can also hold containers because it is a (subclass of) component

Containers are components
Components are also containers

Containers: top level container

- A Swing application requires a **top-level container** (a window that is not contained inside another window)
- There are three top-level containers in Swing:
  - **`JFrame (主窗体)`**: used for the application's main window (with an icon, a title, minimize/maximize/close buttons, an optional menu-bar, and a content-pane)
  - **`JDialog (对话框)`**: used for secondary pop-up window (with a title, a close button, and a content-pane).
  - **`JApplet`**: used for the applet's display-area (content-pane) inside a browser's window.



Containers: top level container

- A Swing application requires a **top-level container** (a window that is not contained inside another window)
- There are three top-level containers in Swing:
  - **`JFrame (主窗体)`**: used for the application's main window (with an icon, a title, minimize/maximize/close buttons, an optional menu-bar, and a content-pane)
  - **`JDialog (对话框)`**: used for secondary pop-up window (with a title, a close button, and a content-pane).
  - **`JApplet`**: used for the applet's display-area (content-pane) inside a browser's window.
- There are secondary containers (such as `JPanel面板`) which can be used to group and layout relevant components (布局).

Secondary containers are placed inside a top-level container or another secondary container

Building Our First Swing Program

```
import javax.swing.JFrame;

public class HelloWorld extends JFrame {
    public HelloWorld() {
        super("Our first Swing program");
    }

    public static void main(String[] args) {
        HelloWorld gui = new HelloWorld();
        gui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        gui.setSize(800, 600);
        gui.setVisible(true);
    }
}
```

Select a top-level container (mostly JFrame)

Creates a new, initially invisible Frame with the specified title.

Exit the application (process) when the close button is clicked.
Default value HIDE_ON_CLOSE hides the JFrame, but keeps the application running.

Building Our First Swing Program

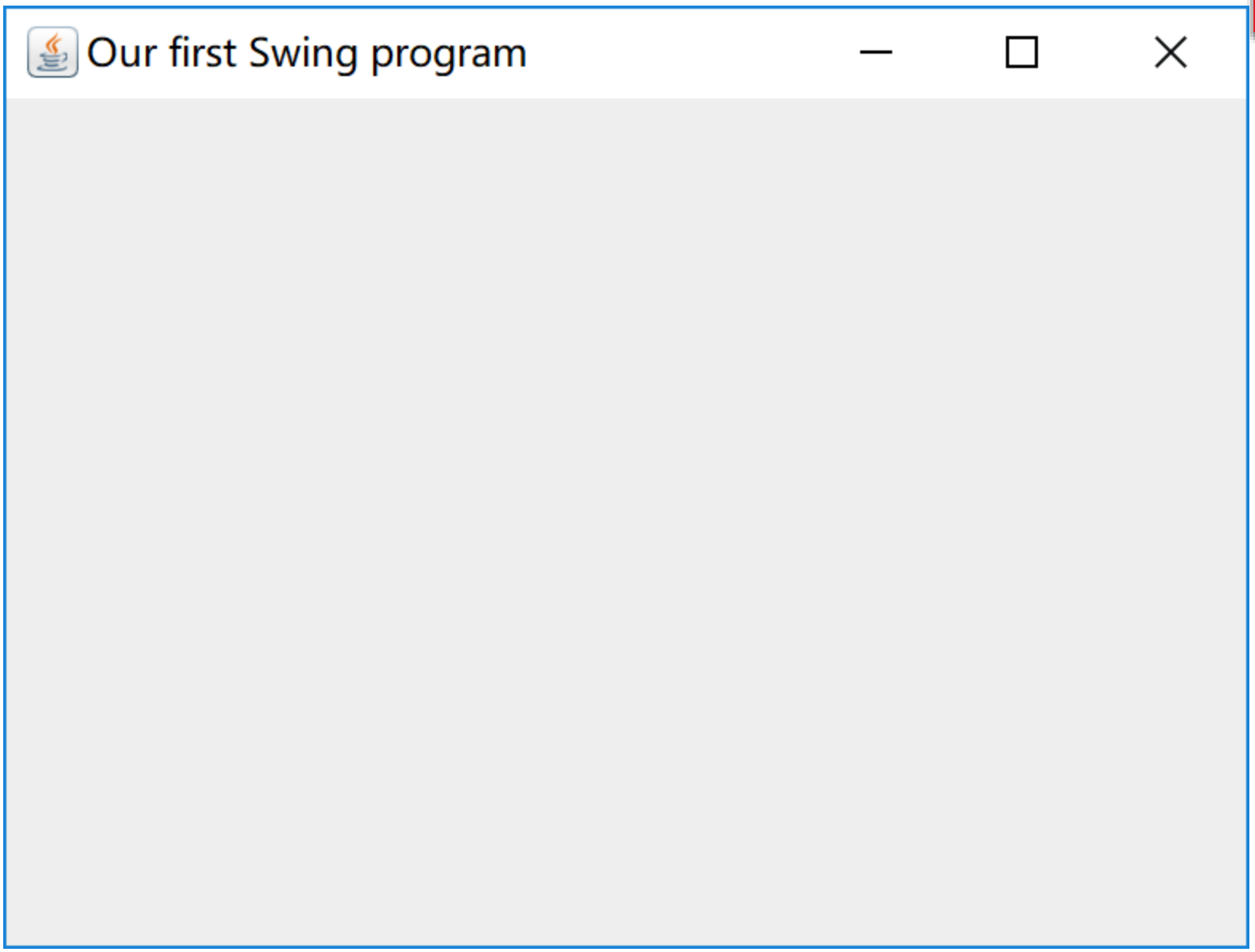
```
import javax.swing.JFrame;

public class HelloWorld extends JFrame {
    public HelloWorld() {
        super("Our first Swing program");
    }

    public static void main(String[] args) {
        HelloWorld gui = new HelloWorld();
        gui.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        gui.setSize(800, 600);
        gui.setVisible(true);
    }
}
```

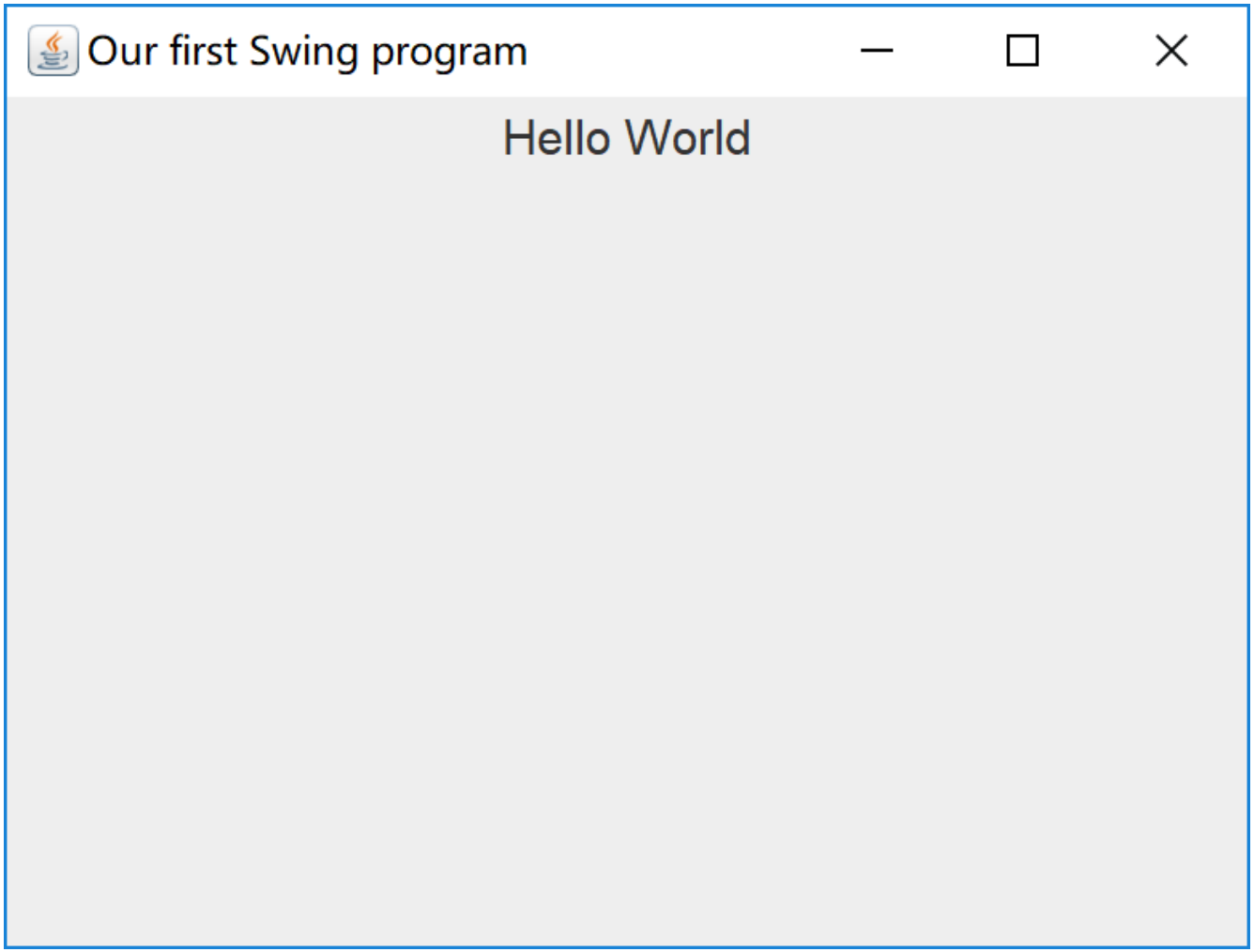
Select a top-level container (mostly JFrame)

Creates a new, initially invisible Frame with the specified title.

By default, a frame has a rather useless size of 0 × 0 pixels, which need to be resized properly

Display the JFrame





How to add the component?

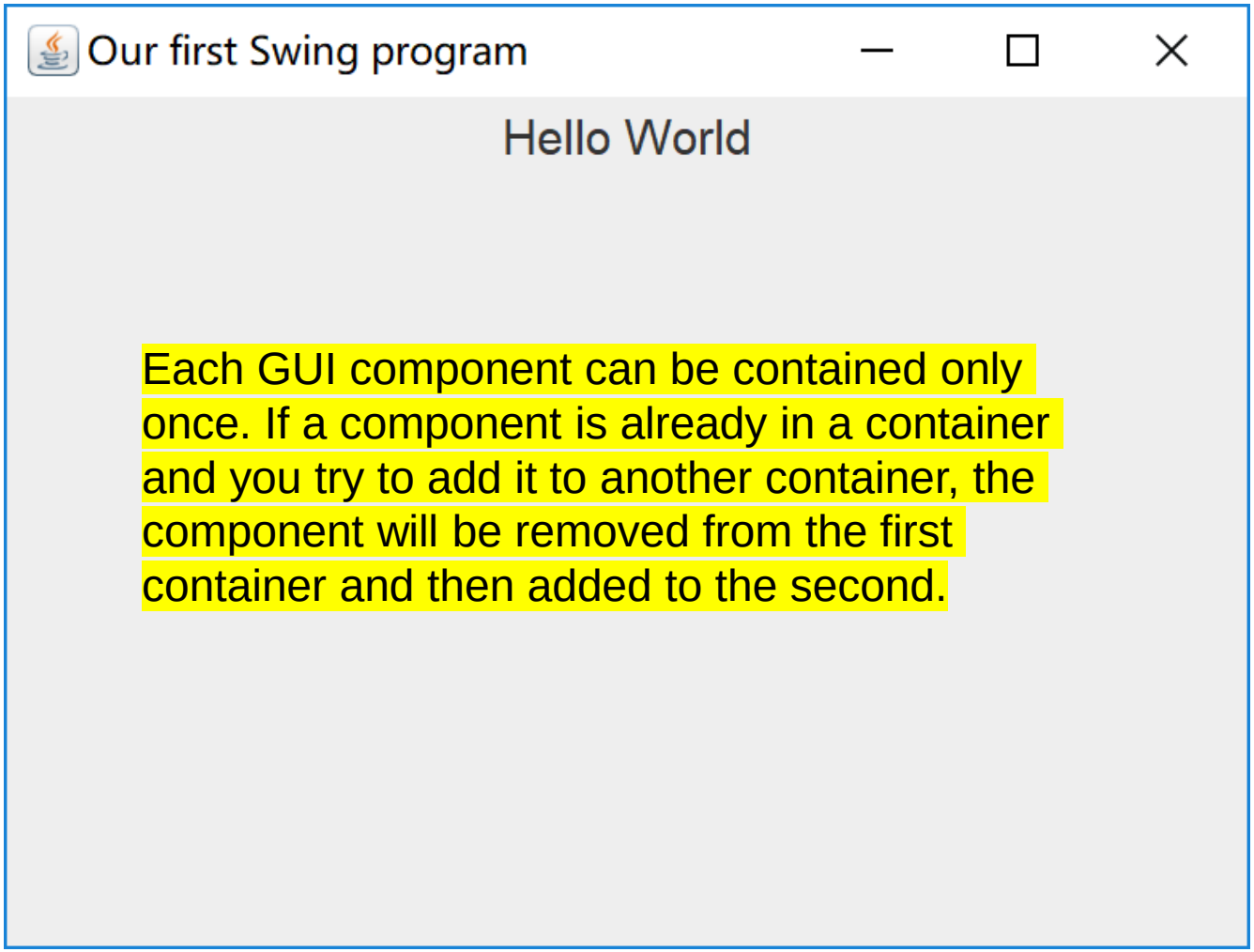
Building Our First Swing Program

```
public class HelloWorld extends JFrame {

    private JLabel label;

    public HelloWorld() {
        super("Our first Swing program");
        setLayout(new FlowLayout());
        label = new JLabel("Hello World");
        label.setFont(new Font("San Serif", Font.PLAIN, 30));
        add(label);
    }

    public static void main(String[] args) { // same as earlier }
}
```

Declaring GUI components as fields makes it easier to interact with the corresponding objects

Specifying layout (how to position GUI components)

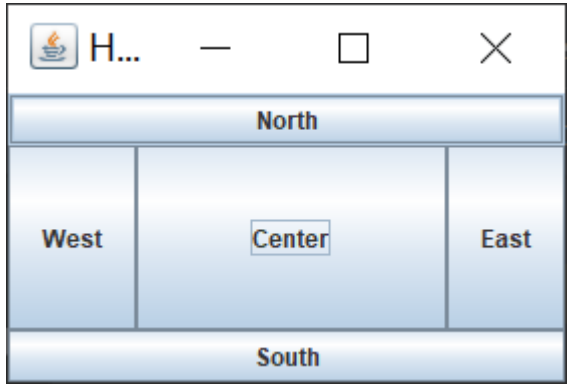
Creating GUI component (a label here) and add it to the JFrame (actually its content pane)



JPanel

JPanel is a container that can store a group of components and organize components in various layouts

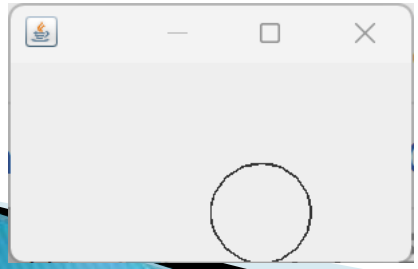
```
public class JPanelTest {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello World");

        //Create a panel and add components to it.
        JPanel panel = new JPanel(new BorderLayout());
        panel.add(new JButton("North"), BorderLayout.NORTH);
        panel.add(new JButton("South"), BorderLayout.SOUTH);
        panel.add(new JButton("West"), BorderLayout.WEST);
        panel.add(new JButton("East"), BorderLayout.EAST);
        panel.add(new JButton("Center"), BorderLayout.CENTER);

        frame.setContentPane(panel);
        frame.setSize(300,200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
```



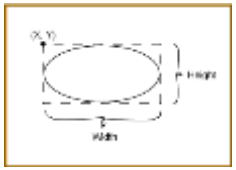
Draw a Component

- To draw on a component, you define a class that extends `JComponent` and override the `paintComponent` method in that class.
- The `paintComponent` method takes one parameter of type `Graphics`, which has methods that draw patterns, images, and text.

Measurement on a Graphics object for screen display is done in pixels. The (0, 0) coordinate denotes the top left corner of the component on whose surface you are drawing.



```
public class GraphicsDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        MyCircle circle = new MyCircle();
        frame.add(circle);
        frame.pack();
        frame.setVisible(true);
    }
}

class MyCircle extends JComponent{
    int X = 100;
    int Y = 50;

    @Override
    public void paintComponent(Graphics g){
        g.drawOval(X,Y,50,50);
    }

    @Override
    public Dimension getPreferredSize(){
        return new Dimension(200, 100);
    }
}
```



Draw a Component

- Never call the `paintComponent` method yourself. It is called automatically whenever a part of your application needs to be redrawn, and you should not interfere with this automatic process.

- What sorts of actions trigger this automatic response? For example,
  - Painting occurs when the user increases the size of the window
  - When users minimizes and then restores the window.

```
public class GraphicsDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        MyCircle circle = new MyCircle();
        frame.add(circle);
        frame.pack();
        frame.setVisible(true);
    }
}

class MyCircle extends JComponent{
    int X = 100;
    int Y = 50;

    @Override
    public void paintComponent(Graphics g){
        g.drawOval(X,Y,50,50);
    }

    @Override
    public Dimension getPreferredSize(){
        return new Dimension(200, 100);
    }
}
```

Draw a Component

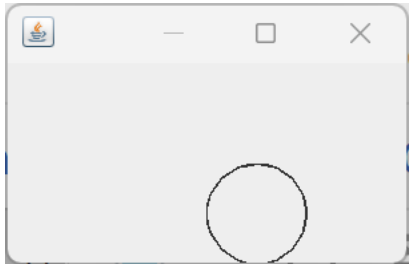
- Never call the `paintComponent` method yourself. It is called automatically whenever a part of your application needs to be redrawn, and you should not interfere with this automatic process.

- If you need to force repainting of the screen, call the `repaint()` method instead of paintComponent. The `repaint()` method will cause paintComponent to be called for all components, with a properly configured Graphics object.

```
public class GraphicsDemo {
   public static void main(String[] args) {
        JFrame frame = new JFrame();
        MyCircle circle = new MyCircle();
        frame.add(circle);
        frame.pack();
        frame.setVisible(true);
   }
}

class MyCircle extends JComponent{
   int X = 100;
   int Y = 50;

   @Override
   public void paintComponent(Graphics g){
      g.drawOval(X,Y,50,50);
   }

   @Override
   public Dimension getPreferredSize(){
      return new Dimension(200, 100);
   }
}
```

Draw a Component

- A component should tell its users how big it would like to be. Override the `getPreferredSize` method and return an object of the `Dimension` class with the preferred width and height
- When you fill a frame with one or more components, and you simply want to use their preferred size, call the `pack` method instead of the setSize method



```
public class GraphicsDemo {
   public static void main(String[] args) {
         JFrame frame = new JFrame();
         MyCircle circle = new MyCircle();
         frame.add(circle);
         frame.pack();
         frame.setVisible(true);
   }
}

class MyCircle extends JComponent{
   int X = 100;
   int Y = 50;

   @Override
   public void paintComponent(Graphics g){
      g.drawOval(X,Y,50,50);
   }

   @Override
   public Dimension getPreferredSize(){
      return new Dimension(200, 100);
   }
}
```

Dialogs (对话框)

- A Dialog window is an independent sub window meant to carry temporary notice apart from the main Swing Application Window
- Most Dialogs present an error message or warning to a user, but Dialogs can present images, directory trees, or just about anything compatible with the main Swing Application that manages them.

- To create simple, standard dialogs (标准对话框), you use the `JOptionPane` class
- To create a custom dialog (自定义对话框), use the `JDialog` class directly.

https://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html

JOptionPane

- `JOptionPane` is a widely-used Swing class for popping up a dialog box that prompts users for a value or informs them of something.

```
public static void main(String[] args) {
    String str1 = JOptionPane.showInputDialog("Enter 1st integer");
    String str2 = JOptionPane.showInputDialog("Enter 2nd integer");
    int num1 = Integer.parseInt(str1);
    int num2 = Integer.parseInt(str2);
    int sum = num1 + num2;
    JOptionPane.showMessageDialog(null, num1 + " + " + num2 + " = " + sum);
}
```

JOptionPane

- `JOptionPane` is a widely-used Swing class for popping up a dialog box that prompts users for a value or informs them of something.

Static method `showInputDialog()` prompts for user input

```
public static void main(String[] args) {
    String str1 = JOptionPane.showInputDialog("Enter 1st integer");
    String str2 = JOptionPane.showInputDialog("Enter 2nd integer");
    int num1 = Integer.parseInt(str1);
    int num2 = Integer.parseInt(str2);
    int sum = num1 + num2;
    JOptionPane.showMessageDialog(null, num1 + " + " + num2 + " = " + sum);
}
```

"123" will be read as a string

`null` will be read

JOptionPane

- `JOptionPane` is a widely-used Swing class for popping up a dialog box that prompts users for a value or informs them of something.

```
public static void main(String[] args) {
    String str1 = JOptionPane.showInputDialog("Enter 1st integer");
    String str2 = JOptionPane.showInputDialog("Enter 2nd integer");
    int num1 = Integer.parseInt(str1);
    int num2 = Integer.parseInt(str2);
    int sum = num1 + num2;
    JOptionPane.showMessageDialog(null, num1 + " + " + num2 + " = " + sum);
}
```

Static method `showMessageDialog()` tells user about something that has happened

Events (in GUI Programming)

- All GUI applications are event-driven.
- In GUI programming, events describe the change in the state of a GUI component when users interact with it

- For example, events will occur when
  - A button is clicked
  - The mouse is moved
  - A character is entered through keyboard
  - An item from a list is selected
  - …

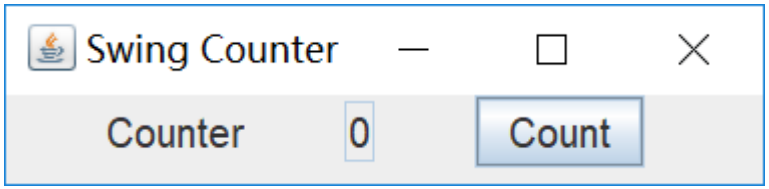
Event Handling

- Event handling is the mechanism that controls the event and decides what should happen if an event occurs. Three key concepts:
  - **Event source (事件源):** the GUI component with which the user interacts (e.g., a button)
  - **Event object (or simply event):** encapsulate the information about the event that occurred (e.g., a MouseEvent)
  - **Event listener (事件监听器):** an object that is notified by the event source when an event occurs.
    - A method of the event listener receives an event object when the event listener is notified of the event.
    - The listener then uses the event object to respond to the event.

Event Handling Example

- We use a counter program to illustrate the steps

```
public class SwingCounter extends JFrame {
    private JTextField tfCount;
    private JButton btnCount;
    private int count = 0;
    public SwingCounter() {
        setLayout(new FlowLayout(FlowLayout.LEFT, 50, 0));
        add(new JLabel("Counter"));
        tfCount = new JTextField("0");
        tfCount.setEditable(false); add(tfCount);
        btnCount = new JButton("Count"); add(btnCount);
    }
    public static void main(String[] args) { SwingCounter sc = new SwingCounter(); ... }
}
```



Nothing will happen when we click the button (we have not handled the event yet)

Event Handling Example

- **Step 1:** check what event will occur when JButton is clicked

- An `ActionEvent` (in `java.awt.event` package) will occur whenever the user performs a component-specific action on a GUI component
  - When user clicks a button
  - When user chooses a menu item
  - When user presses Enter after typing something in a text field…

Event Handling Example

- **Step 2:** define the event listener class by implementing the corresponding listener interface

```
public class ButtonClickListener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent arg0) {
        // code to react to the event
    }

}
```

`ActionListener` is from the package `java.awt.event`

Event Handling Example

- The event listener class is often declared as an inner class

```
public class SwingCounter extends JFrame {

    private JTextField tfCount;
    private JButton btnCount;
    private int count = 0;

    public class ButtonClickListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent arg0) {
            ++count;  tfCount.setText(count + "");
        }

    }
}
```

An inner class is a proper class. It can have constructors, fields, methods …

An inner class is a member of the outer class. Therefore, it can access the private members of the outer class **(this is very useful)**

Event Handling Example

- **Step 3:** register an instance of the event listener class as a listener on the corresponding GUI component (event source)

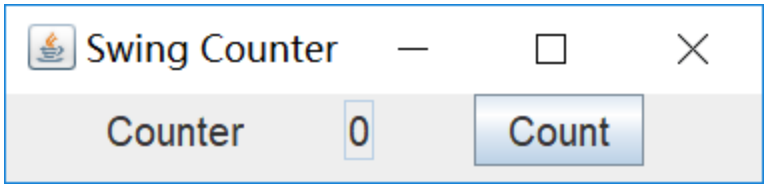
```
btnCount.addActionListener(new ButtonClickListener());
```

```
public class SwingCounter extends JFrame {
    private JTextField tfCount;
    private JButton btnCount;
    private int count = 0;
    public SwingCounter() {
        setLayout(new FlowLayout(FlowLayout.LEFT, 50, 0));
        add(new JLabel("Counter"));
        tfCount = new JTextField("0");
        tfCount.setEditable(false); add(tfCount);
        btnCount = new JButton("Count"); add(btnCount);
        btnCount.addActionListener(new ButtonClickListener());
    }
    public class ButtonClickListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent arg0) {
            count++; tfCount.setText(count + "");
        }
    }
    public static void main(String[] args) { … }
}
```

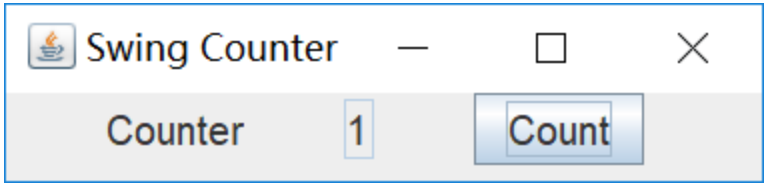
Event source

Event listener

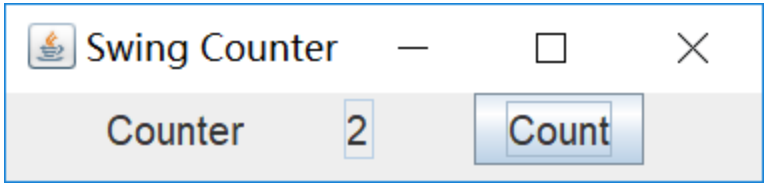
Event object will be passed here



Initial state



After one click



After two clicks

. . .

Implementing Event Listeners

- Inner class
  - A class defined within another class (outer class)
  - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
  - An inner class can access private members of the outer class
- Anonymous class
- Lambda expression

Implementing Event Listeners

- Anonymous class
  - Anonymous classes are inner classes with no name
  - We need to declare and instantiate anonymous classes in a single expression at the point of use.

```
new InterfaceName() {...}
```

name of the interface to implement — methods' implementations

```
btnCount.addActionListener(new ButtonClickListener());


public class ButtonClickListener implements ActionListener {
   @Override
   public void actionPerformed(ActionEvent arg0) {
      ++count;
      tfCount.setText(count + "");
   }
}
```

```
btnCount.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
      ++count;
      tfCount.setText(count + "");
   }
});
```

Implementing Event Listeners

- Lambda Expression
  - To implement interfaces that have just one method, we could use lambda expressions

```
public interface ActionListener extends EventListener {

    public void actionPerformed(ActionEvent e);

}
```

```
btnCount.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        ++count;
        tfCount.setText(count + "");
    }
});
```

```
btnCount.addActionListener(e -> {
    ++count;
    tfCount.setText(count + "");
});
```

Layout Management (布局管理)

- Layout managers control how to place the GUI components (containers can also be treated as components) in a container for presentation purposes.

- You can use the layout manager for basic layout capabilities instead of determine every GUI component's exact position and size (which is non-trivial and error-prone)

Layout Management (布局管理)

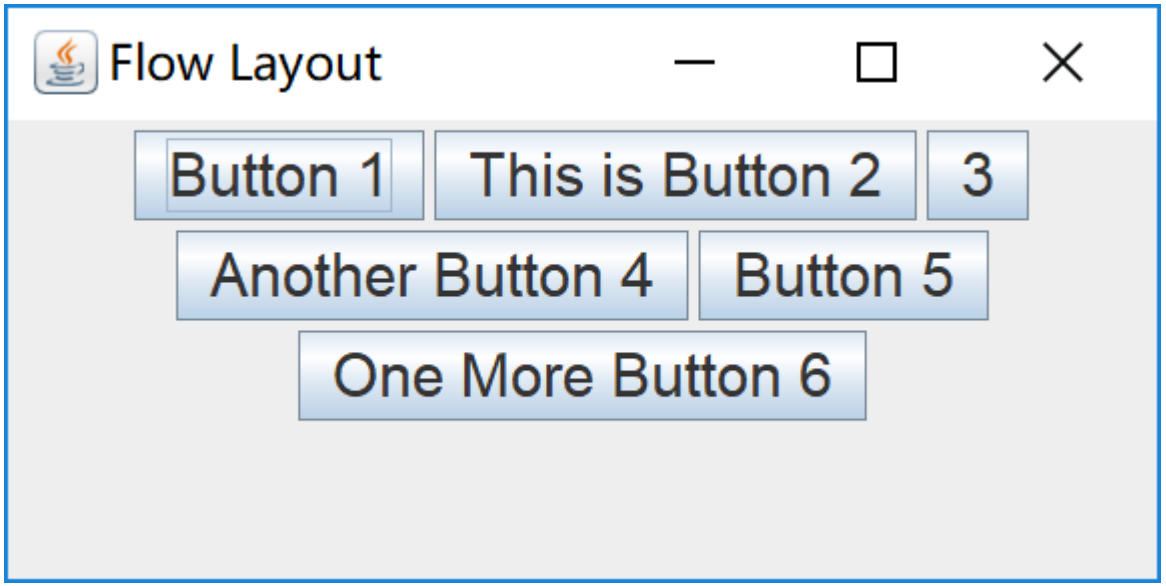
- All layout managers in Java implement the interface `LayoutManager` (in the package `java.awt`)

- Commonly-used layout managers: `FlowLayout`, `BorderLayout`, `GridLayout`

FlowLayout

```
public class FlowLayoutDemo extends JFrame {
    private JButton btn1, btn2, btn3, btn4, btn5, btn6;

    public FlowLayoutDemo() {
        super("Flow Layout");
        setLayout(new FlowLayout());
        btn1 = new JButton("Button 1"); add(btn1);
        btn2 = new JButton("This is Button 2"); add(btn2);
        btn3 = new JButton("3"); add(btn3);
        btn4 = new JButton("Another Button 4"); add(btn4);
        btn5 = new JButton("Button 5"); add(btn5);
        btn6 = new JButton("One More Button 6"); add(btn6);
    }

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

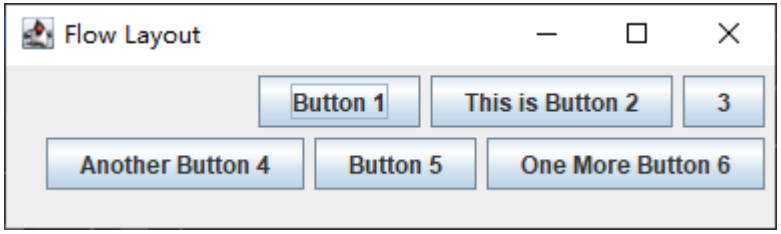
FlowLayout



- Default layout manager for the secondary container `javax.swing.JPanel`
- Places components in a straight horizontal line. If there is no enough space to fit all component into one line, simply move the next line

FlowLayout: Alignment


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



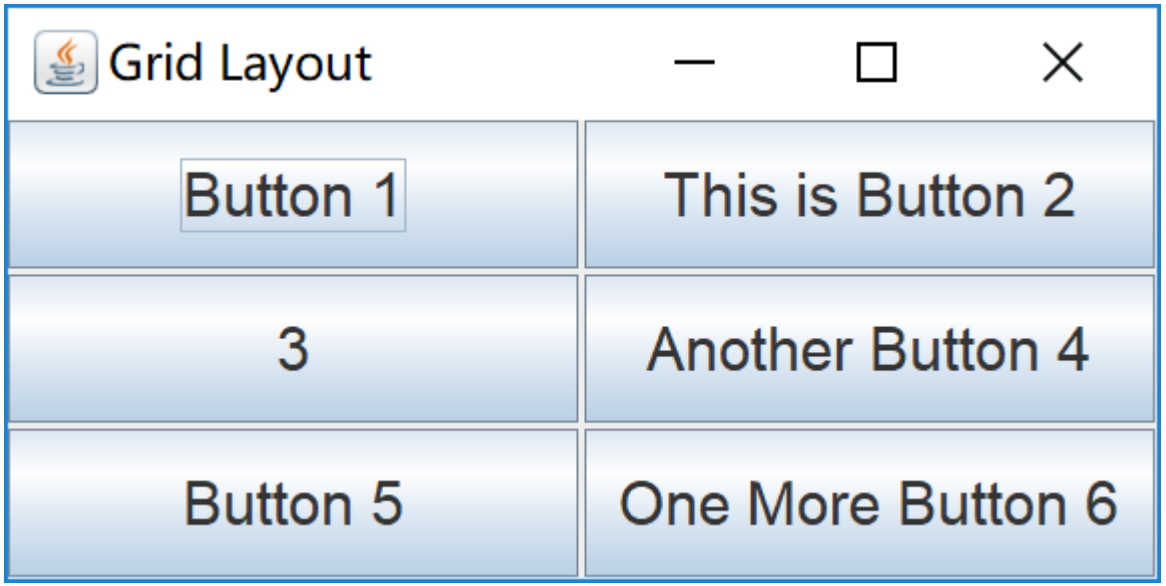

```
setLayout(new FlowLayout(FlowLayout.RIGHT));
```

GridLayout

```
public class GridLayoutDemo extends JFrame {
    private JButton btn1, btn2, btn3, btn4, btn5, btn6;

    public GridLayoutDemo() {
        super("Grid Layout");
        setLayout(new GridLayout(3, 2, 3, 3));
        btn1 = new JButton("Button 1"); add(btn1);
        btn2 = new JButton("This is Button 2"); add(btn2);
        btn3 = new JButton("3"); add(btn3);
        btn4 = new JButton("Another Button 4"); add(btn4);
        btn5 = new JButton("Button 5"); add(btn5);
        btn6 = new JButton("One More Button 6"); add(btn6);
    }

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

3 x 2 grid layout (3 rows, 2 columns)
Horizontal and vertical gaps between components: 3 pixels

GridLayout



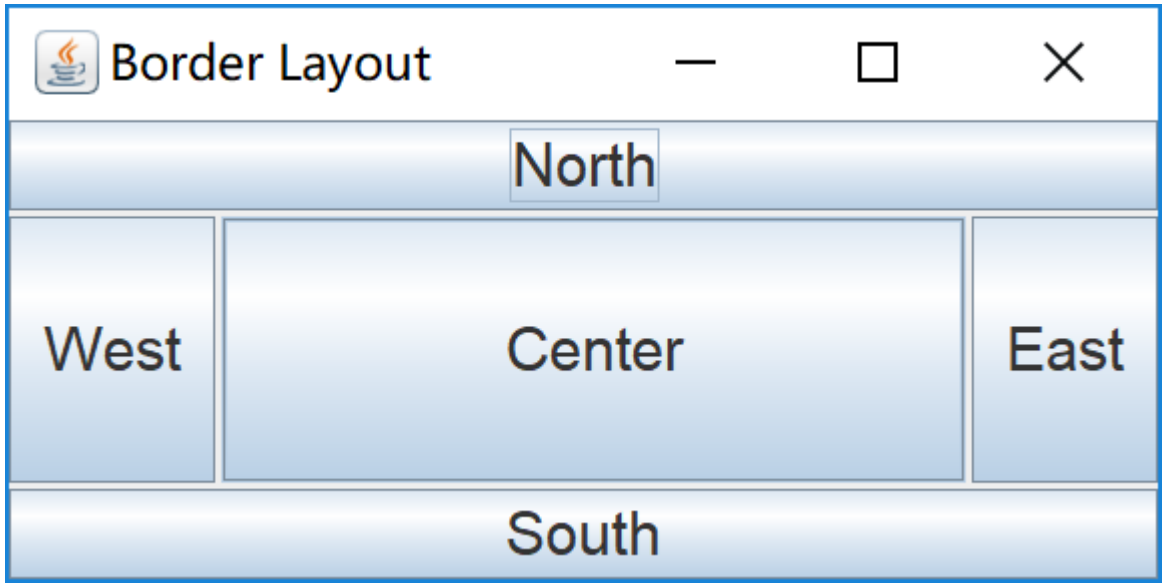
- Places components into rows and columns

BorderLayout

```
public class BorderLayoutDemo extends JFrame {
  private JButton btnNorth, btnSouth, btnCenter, btnEast, btnWest;

  public BorderLayoutDemo() {
    super("Border Layout");
    setLayout(new BorderLayout(3, 3));
    btnNorth = new JButton("North"); add(btnNorth, BorderLayout.NORTH);
    btnSouth = new JButton("South"); add(btnSouth, BorderLayout.SOUTH);
    btnCenter = new JButton("Center"); add(btnCenter, BorderLayout.CENTER);
    btnEast = new JButton("East"); add(btnEast, BorderLayout.EAST);
    btnWest = new JButton("West"); add(btnWest, BorderLayout.WEST);
  }

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

Horizontal and vertical gaps: 3 pixels

BorderLayout



- Default layout manager for the content pane of top level container `javax.swing.JFrame`
- Arranges the GUI components into five pre-defined areas: NORTH, SOUTH, EAST, WEST, CENTER

Using secondary containers for layout management

```
public class LayoutDemo extends JFrame {
    private JButton btn1, btn2, btn3, btn4, btn5, btn6;

    public LayoutDemo() {
        super("Layout demo");
        setLayout(new GridLayout(2, 1));
        JPanel panel1 = new JPanel(new FlowLayout());
        JPanel panel2 = new JPanel(new GridLayout(2, 2, 3, 3));
        add(panel1); add(panel2);
        btn1 = new JButton("Button 1"); panel1.add(btn1);
        btn2 = new JButton("This is Button 2"); panel1.add(btn2);
        btn3 = new JButton("Button 3"); panel2.add(btn3);
        btn4 = new JButton("Button 4"); panel2.add(btn4);
        btn5 = new JButton("Button 5"); panel2.add(btn5);
        btn6 = new JButton("Button 6"); panel2.add(btn6);
    }
    public static void main(String[] args) {...}
}
```

Create two `JPanel`s

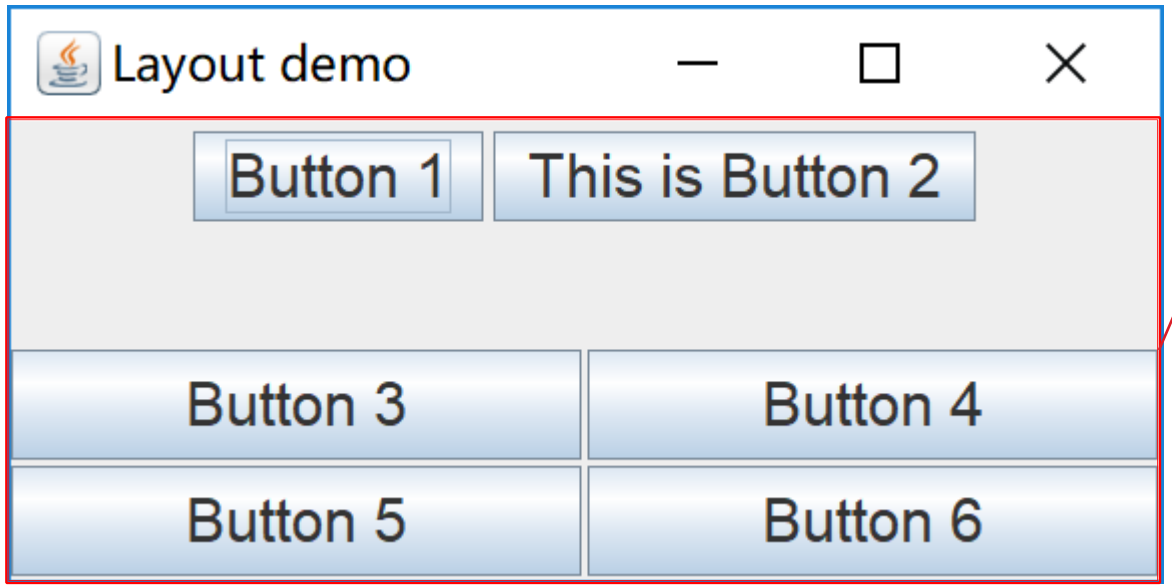
Group buttons

Using secondary containers for layout management

```
public class LayoutDemo extends JFrame {
    private JButton btn1, btn2, btn3, btn4, btn5, btn6;

    public LayoutDemo() {
        super("Layout demo");
        setLayout(new GridLayout(2, 1)); // Set the layout of JFrame's content pane
        JPanel panel1 = new JPanel(new FlowLayout());
        JPanel panel2 = new JPanel(new GridLayout(2, 2, 3, 3));
        add(panel1); add(panel2);  // add the two JPanels to the JFrame
        btn1 = new JButton("Button 1"); panel1.add(btn1);
        btn2 = new JButton("This is Button 2"); panel1.add(btn2);
        btn3 = new JButton("Button 3"); panel2.add(btn3);
        btn4 = new JButton("Button 4"); panel2.add(btn4);
        btn5 = new JButton("Button 5"); panel2.add(btn5);
        btn6 = new JButton("Button 6"); panel2.add(btn6);
    }
    public static void main(String[] args) {...}
}
```

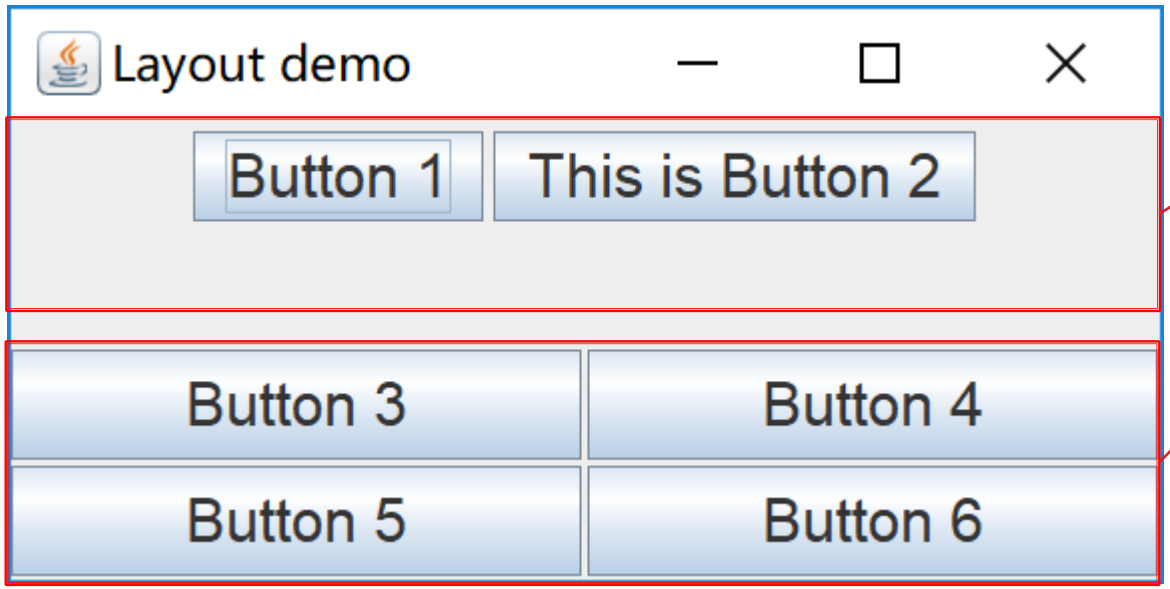
Set layout for the JPanels

Using secondary containers for layout management



JFrame's content pane
(grid layout, 2 rows, 1 col)

Using secondary containers for layout management



JPanel 1 contains two buttons (flow layout)

JPanel 2 contains 4 buttons (grid layout, 2 rows, 2 cols)