

Object-Oriented Programming

GUI Programming Part 1

Data Science
United International College

Outline

- AWT, Swing, JavaFX
- Frames
- Buttons
- Layout managers
- Colors and fonts
- Panels
- Simple input / output

Java and GUI Programming

- To this point, you have seen only how to write programs that take input from the keyboard, fuss with it, and then display the results on a console screen.
- Not what most users want. Modern programs don't work this way.
- This chapter starts you on the road to writing Java programs that use a graphical user interface (GUI)
 - Size and locate windows on the screen
 - Display text, images in a window
 - Receiving user input via mouse click or keyboard

Java and GUI Programming

Java has three different libraries for implementing Graphical User Interfaces:

1. AWT

- Original Java GUI library (1996).
- Only has basic GUI components.
- Quickly replaced by Swing.

2. Swing

- Main Java GUI library for a long time (1998).
- Still uses a few basic AWT components.
- Still used a lot today.
- Will still be part of Java until at least 2026.

Java and GUI Programming

3. JavaFX

- Newest GUI library (2008)
- Can use XML and graphical GUI builder to design GUI layout with the mouse, no coding.
- Can use CSS to specify visual style.
- Built-in animations.
- Better event handling than Swing.
- Not very successful compared to Swing...
- Not part of Java SE 11!
- Slowly dying?

So we will use Swing.

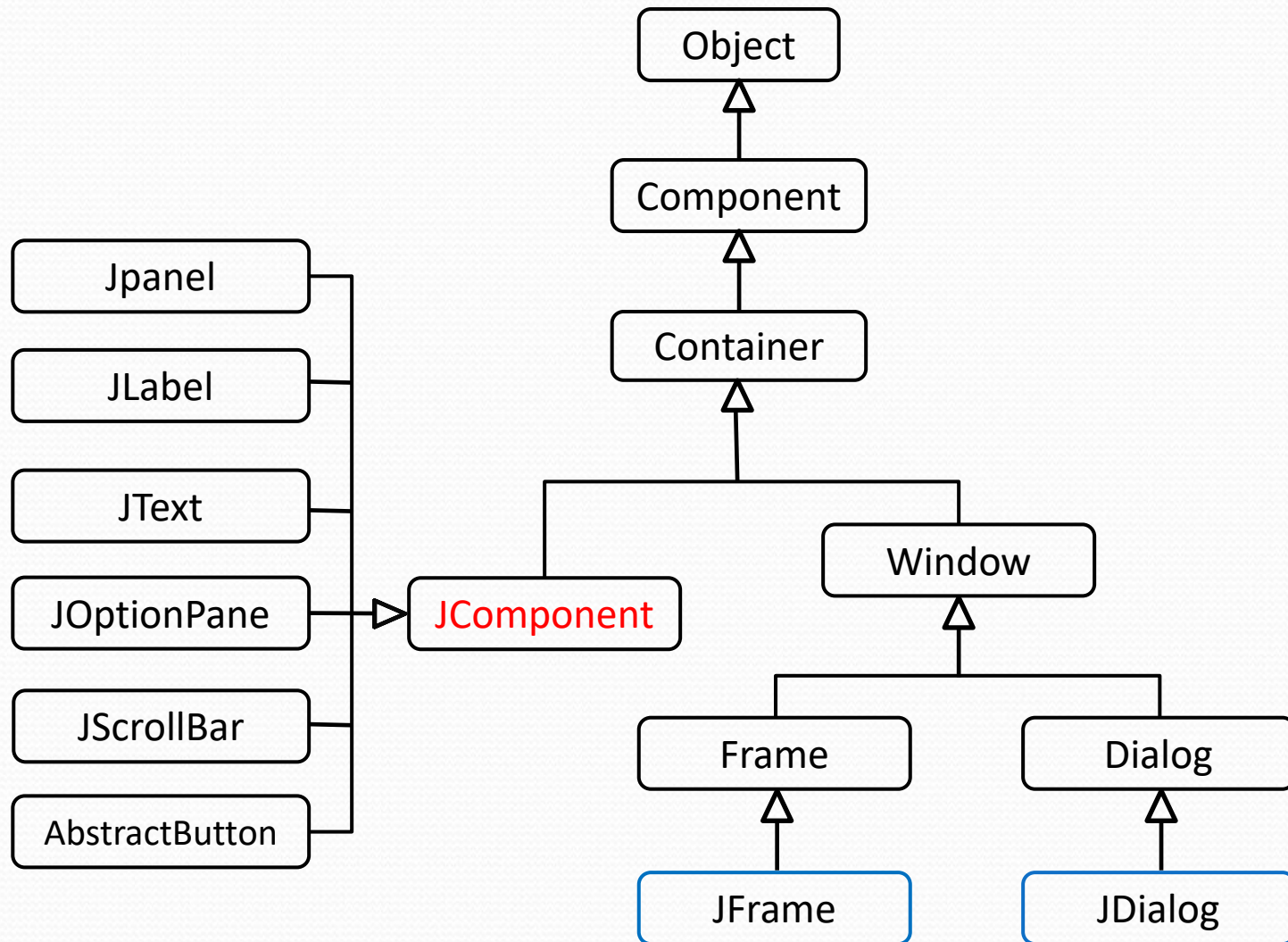
Java and GUI Programming

- Every user interface considers the following three main aspects –
 - **UI Elements** – These are the core visual elements the user eventually sees and interacts with.
 - **Layouts** – They define how UI elements should be organized on the screen and provide a final look and feel to the GUI .
 - **Behavior** – These are the events which occur when the user interacts with UI elements.
- No simple testing method: run, click, and look!
- Study only GUI design for now, connect with other classes later (Model-View-Controller).

Java and GUI Programming

- There are two types of UI elements:
 - **Components**: elementary UI entities, such as button, label and textField.
 - **Containers**: hold **components** in a specific layout. A container can also hold **sub-containers**. E.g., frame and panel.
- In a GUI program, a component must be kept in a container. You need to identify a container to hold the components.
 - Every container has a method called **add (Component c)**

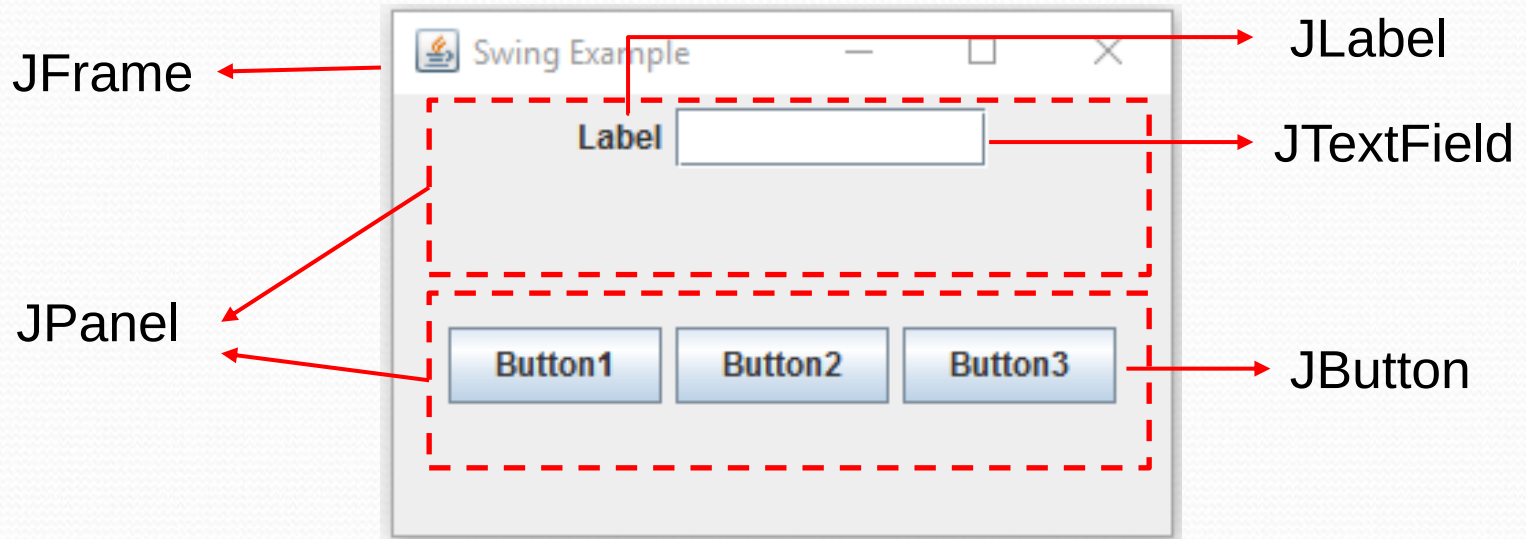
Inheritance Hierarchy of Swing Class



SWING UI Elements

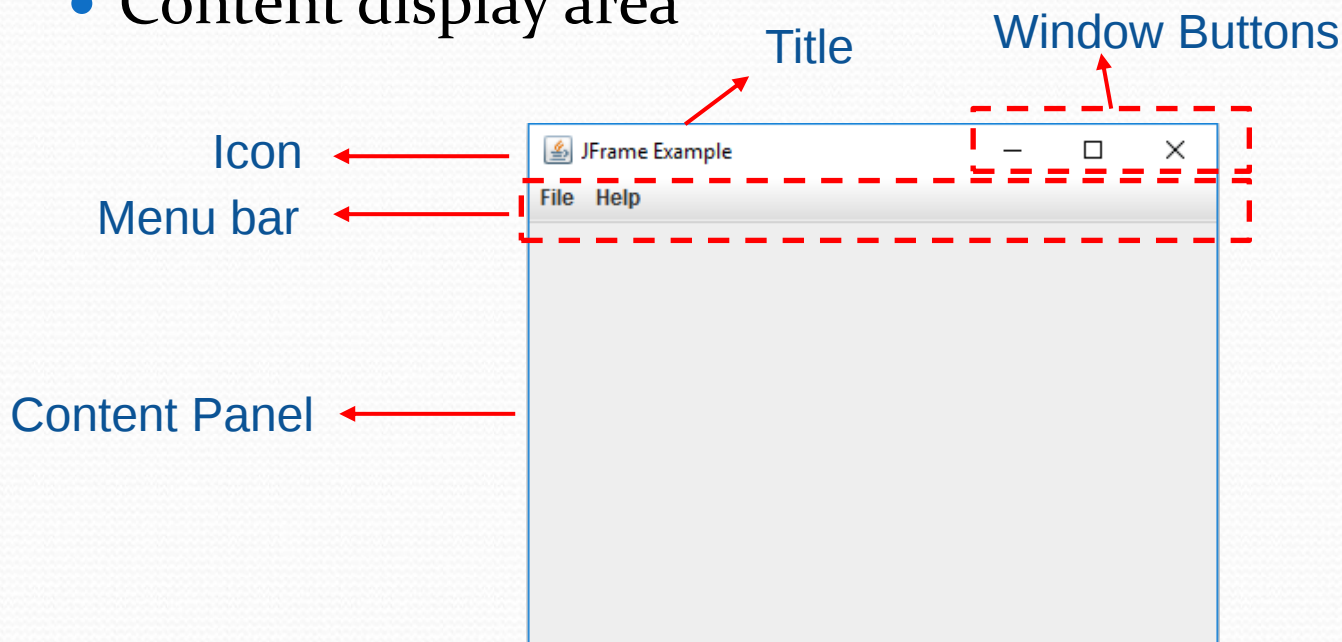
- Following is the list of commonly used controls while designing GUI using SWING
 - JLabel
 - JButton
 - JTextField
 - JTextArea
 - JOptionPane
 - JFileChooser
 - ...

Containers and Components



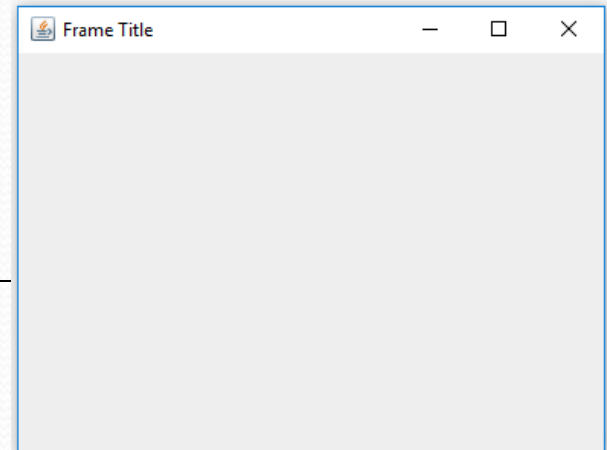
Frames

- A **JFrame** is a commonly used top-level container.
- Provides the “main window” for your GUI application
 - Title bar, contains an icon, a title, the minimize, maximize and close buttons
 - Optional menu bar
 - Content display area



Frames

```
import javax.swing.JFrame;
public class EmptyWindow {
    public static void main(String[] args) {
        // A JFrame is a top-level window with a title and a border.
        JFrame f = new JFrame("Frame Title");
        f.setSize(400, 300);
        // Closing the window terminates the program. Do not forget
        // this otherwise the program might keep running even after
        // the window is closed!
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Make the window visible to the user.
        f.setVisible(true);
        // The window is empty!
    }
}
```



Frames

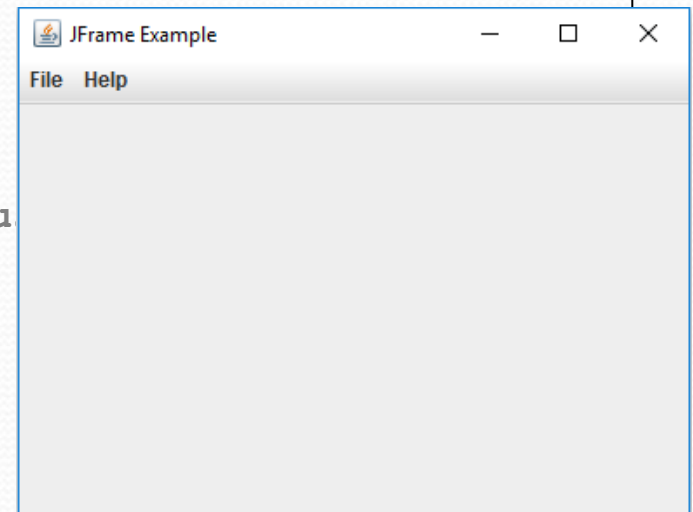
- To write a GUI program, we typically start with a subclass extending from `javax.swing.JFrame` to inherit from the main window:

```
public class MyFrame extends JFrame {  
    public MyFrame() {...}  
    // methods  
  
    public static void main (String[] args)  
    {  
        new MyFrame();  
    }  
}
```

```

import javax.swing.*;
public class MyWindow extends JFrame {
    public MyWindow()
    {
        this.setTitle("JFrame Example");
        this.setSize(400, 300);
        this.setLocation(600,300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JMenuBar menuBar = new JMenuBar(); // Window menu bar
        JMenu menuF= new JMenu("File");
        JMenuItem item1 = new JMenuItem("Open");
        JMenuItem item2 = new JMenuItem("Save As");
        menuF.add(item1);
        menuF.add(item2);
        JMenu menuH= new JMenu("Help");
        menuBar.add(menuF);
        menuBar.add(menuH);
        this.setJMenuBar(menuBar);
        // Make the window visible to the u
        this.setVisible(true);}
    public static void main(String[] args)
        new MyWindow();
    }
}

```



Buttons

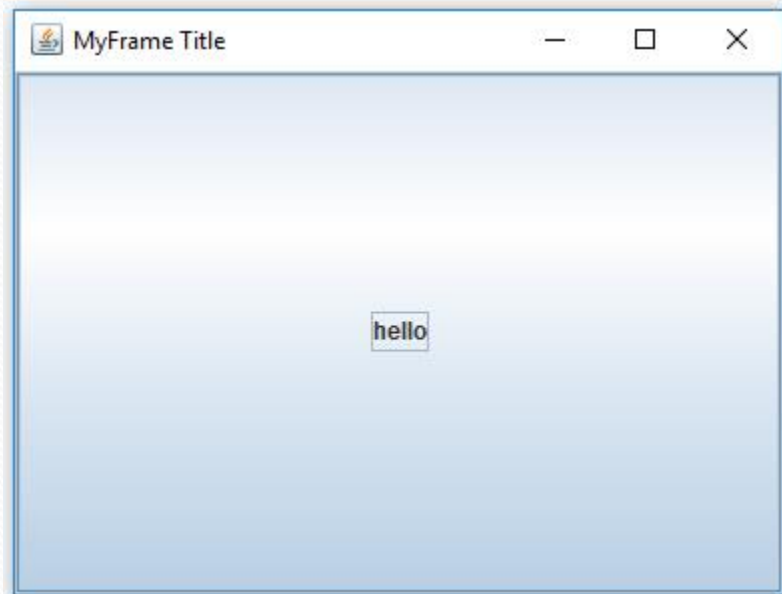
A **JButton** is a Swing object representing one button.

```
import javax.swing.JButton;
import javax.swing.JFrame;
public class MyFrameButton extends JFrame {
    public MyFrameButton() {
        this.setTitle("MyFrame Title");
        this.setSize(400, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Create a button with the text "hello".
        JButton b = new JButton("hello");
        // Add the button to the frame.
        this.add(b);
        // Always make the frame visible *after* adding all the components
        // (like buttons) otherwise the components will not be visible in
        // the frame.
        this.setVisible(true);
    }
}
```

Buttons

A **JButton** is a Swing object representing one button.

```
public class TestFrameButton {  
    public static void main(String[] args)  
    {  
        new MyFrameButton();  
    }  
}
```



Buttons

- A button is a graphical **component**:
[A Visual Guide to Swing Components \(MIT web site\)](#)
- Components needs to be:
 1. **Created** using **new**.
 2. **Added to a component container**.
- A frame is one possible kind of component container (more component containers later).
- Problems:
 - The button takes all the space in the window: we can solve this problem using a **layout manager**.
 - The button does nothing: we can solve this problem using an **event handler** (next week).

Layout Managers

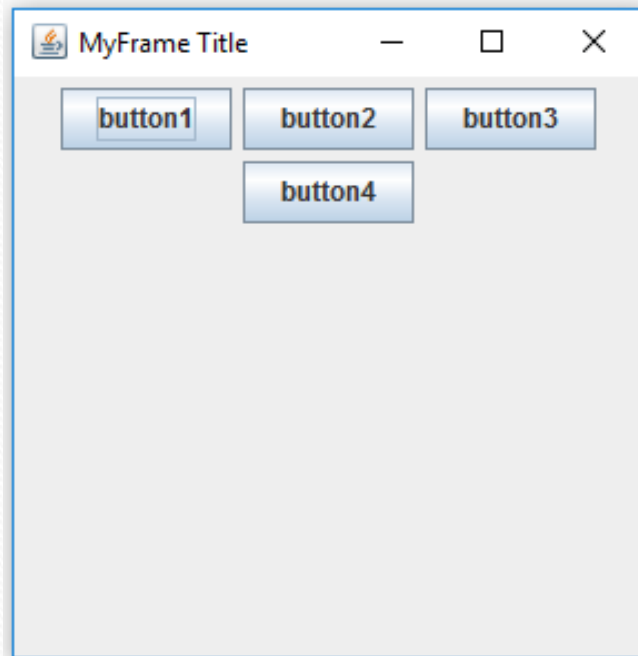
- Layout managers are special Swing objects that **automatically** determine for you the **size and position** of components (like buttons) inside a component container (like a frame).
- Many layout managers available in Swing:
[A Visual Guide to Layout Managers \(Oracle web site\)](#)
- Most important ones are:
 - **BorderLayout**: maximum of 5 components in 5 areas.
 - **FlowLayout**: components next to each other, going to the next line if necessary.
 - **GridLayout**: grid of components.

Layout Managers

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class MyFrameLayout extends JFrame {
    public MyFrameLayout() {
        this.setTitle("MyFrame Title");
        this.setSize(300, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // A flow layout manager lays out components in a single row,
        // going to the next row when the current row is full.
        // Possible arguments for the constructor are: FlowLayout.CENTER
        // (default), FlowLayout.LEFT, FlowLayout.RIGHT.
        FlowLayout fl = new FlowLayout();
        // Set the layout manager for the frame.
        this.setLayout(fl);
        for (int i = 1; i <= 4 ; i++) {
            JButton b = new JButton("button"+i);
            this.add(b);
        }
        this.setVisible(true);
    }
}
```

Layout Managers

```
public class TestFrame {  
    public static void main(String[] args)  
    {  
        new MyFrameLayout();  
    }  
}
```



Colors and fonts

All Swing components have:

- A background color.
- A foreground color.
- A font.

What these exactly mean depends on the type of component!

Colors and fonts

```
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Font;
import javax.swing.JButton;
import javax.swing.JFrame;
public class MyFrameCF extends JFrame {
    public MyFrameCF() {
        this.setTitle("MyFrame Title");
        this.setSize(400, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        FlowLayout fl = new FlowLayout();
        this.setLayout(fl);
        JButton b = new JButton("hello");
        b.setBackground(Color.BLUE);
        b.setForeground(new Color(255, 0, 0)); // red (RGB decimal values).
        // Font: name, style (Font.PLAIN, Font.ITALIC, Font.BOLD),
        // size (in pixels).
        b.setFont(new Font(Font.SERIF, Font.ITALIC, 40));
        this.add(b);
        this.setVisible(true);
    }
}
```



Panels

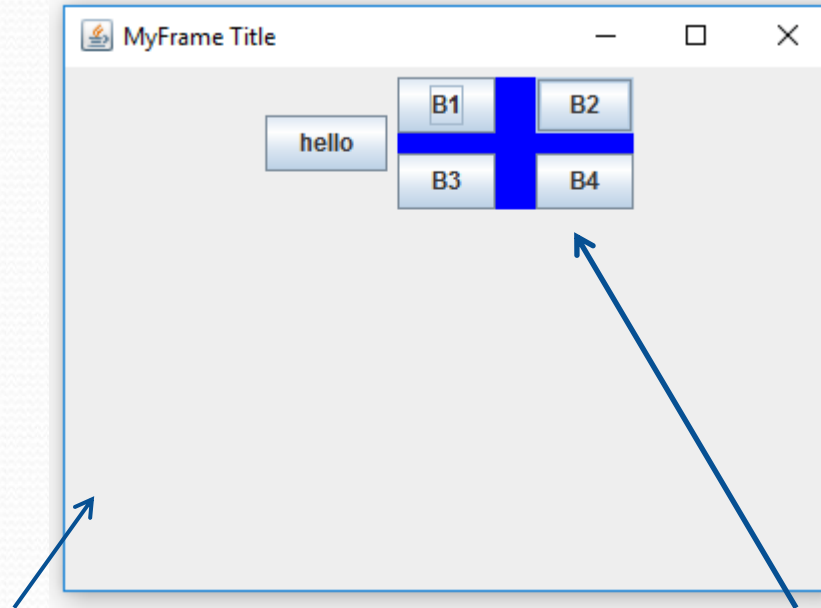
- A panel is a Swing object which is both:
 - a **component** (you can add it to a frame)
 - a **component container** (you can add components to it, including other panels)at the same time!
- A panel can have its **own layout manager**, independently of the frame or of other panels.
- Panels are very useful for **grouping components** and hierarchically organizing GUIs into multiple separate areas, even when you resize the frame!

Panels

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

public class MyFramePanel extends JFrame {
    public MyFramePanel() {
        this.setTitle("MyFrame Title");
        this.setSize(400, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new FlowLayout()); // Layout manager for the frame.
        this.add(new JButton("hello")); // Add button to the frame.
        JPanel p = new JPanel();
        p.setBackground(Color.BLUE);
        this.add(p); // Add panel to the frame.
        // Layout manager for the panel.
        // 2 rows, 2 columns, 20 pixels hgap, 10 pixels vgap.
        p.setLayout(new GridLayout(2, 2, 20, 10));
        // Create four buttons and add them to the panel (not the frame).
        // The four buttons always stay together even when you resize.
        for(int i = 0; i < 4; i++) {
            p.add(new JButton("B" + (i + 1)));
        }
        this.setVisible(true);
    }
}
```

Panels



Frame (grey)
with FlowLayout,
one button, and
one panel.

Panel (blue)
with GridLayout
and four buttons

Painting GUI Components

- In Java, components are rendered on screen in a process known as **painting**. Although this is usually handled automatically, there are occasions when you need to trigger repainting, or modify the default painting for a component.
- Painting is triggered when a GUI element is launched or altered in anyway
 - Frame to be visible, resize, etc.
 - An item is selected, a button is clicked, etc.
- When a paint request is triggered, the **paintComponent** is invoked.
- You can call the **repaint** method to invoke the **paintComponent** method indirectly.

Painting GUI Components

To draw on a component:

- Create your own `JComponent` subclass.
- Override the `paintComponent(Graphics g)` method:
 - Always call `super.paintComponent(g)` first .
 - If you forget this, the component might not be cleaned properly before you draw on it.
 - Use the methods of the graphics object `g` given by Swing as argument to your `paintComponent` method to draw on the component: `setColor`, `drawLine`, etc.

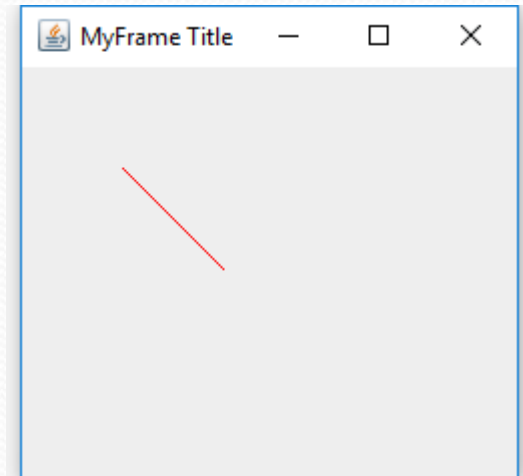
Painting GUI Components

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;
public class MyPanel extends JPanel {
    // The argument g is a Graphics object that lets you directly draw on
    // the panel. Swing automatically supplies g as argument to your
    // method every time Swing needs to call your paintComponent method.
    @Override
    protected void paintComponent(Graphics g) {
        // We must clean the panel before drawing on it.
        super.paintComponent(g);
        // We use g to draw a red line on the clean panel.
        g.setColor(Color.RED);
        g.drawLine(50, 50, 100, 100);
    }
}
```

Painting GUI Components

```
import javax.swing.JFrame;
public class MyFrame extends JFrame {
    public MyFrame() {
        this.setTitle("MyFrame Title");
        this.setSize(400, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // No layout manager for the frame, so the panel will
        // use the full size of the frame.
        this.add(new MyPanel());
        this.setVisible(true);
    }
}

public class Test { // As before.
    public static void main(String[] args) {
        new MyFrame();
    }
}
```



Painting GUI Components

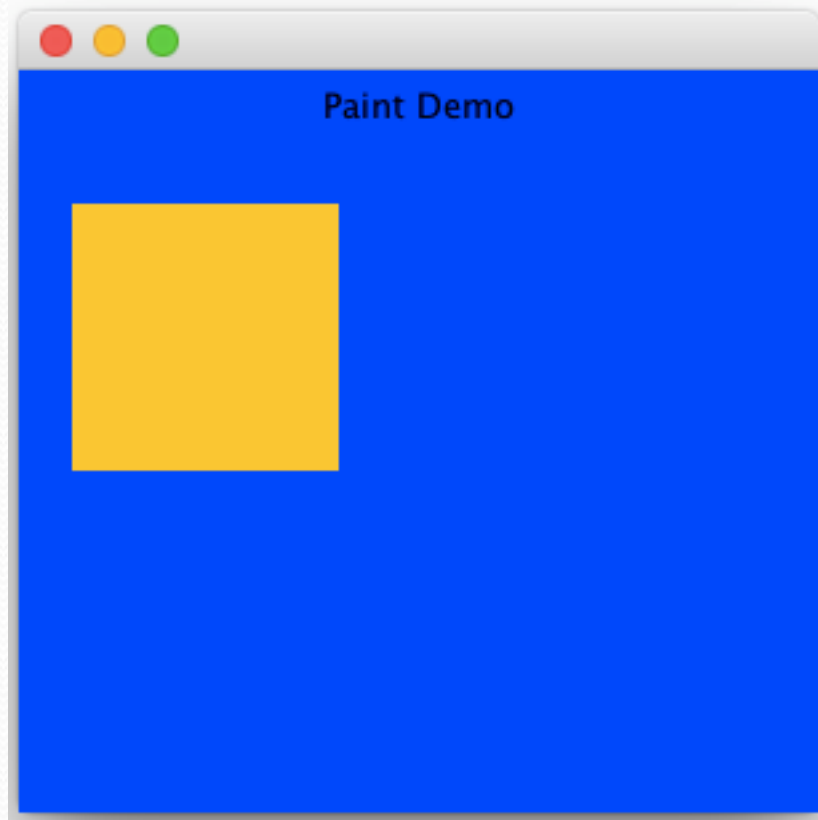
```
import javax.swing.*;
import java.awt.*;

public class MyDrawPanel extends JPanel {
    public MyDrawPanel() {
        JLabel label = new JLabel("Paint Demo");
        this.setBackground(Color.blue);
        this.add(label);
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g); //call paintComponent of JPanel
        g.setColor(Color.orange);
        g.fillRect(20, 50, 100, 100);
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        MyDrawPanel panel = new MyDrawPanel();
        frame.getContentPane().add(panel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        frame.setVisible(true);
    }
}
```

Painting GUI Components



Painting GUI Components

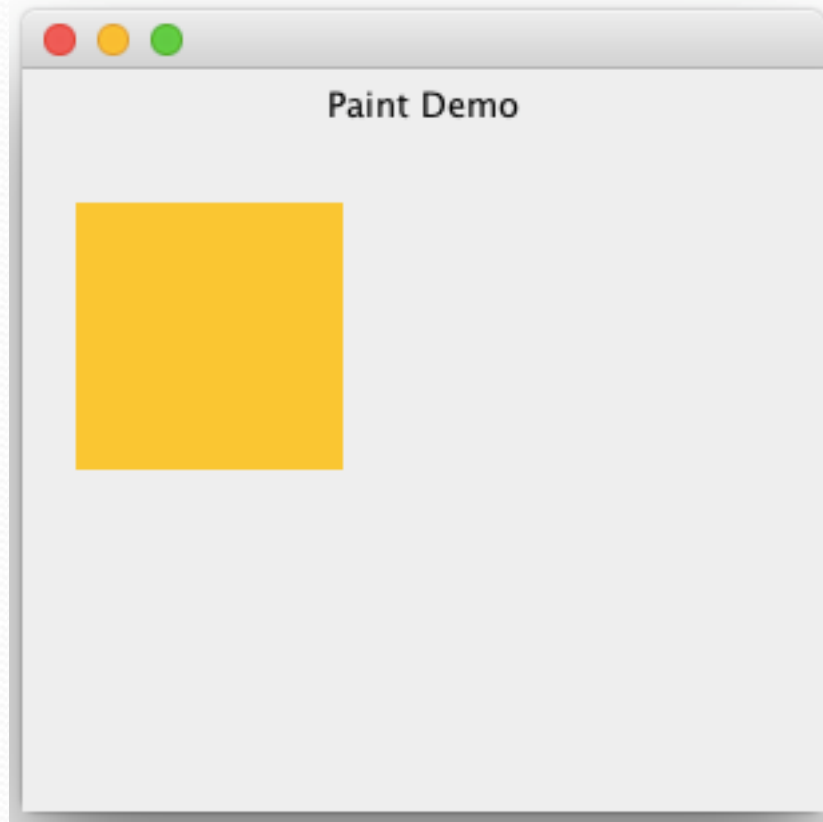
```
import ...

public class MyDrawPanel extends JPanel {
    public MyDrawPanel() {
        JLabel label = new JLabel("Paint Demo");
        this.setBackground(Color.blue);
        this.add(label);
    }

    public void paintComponent(Graphics g) {
        //super.paintComponent(g); //no paintComponent of JPanel
        g.setColor(Color.orange);
        g.fillRect(20, 50, 100, 100);
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        MyDrawPanel panel = new MyDrawPanel();
        frame.getContentPane().add(panel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        frame.setVisible(true);
    }
}
```

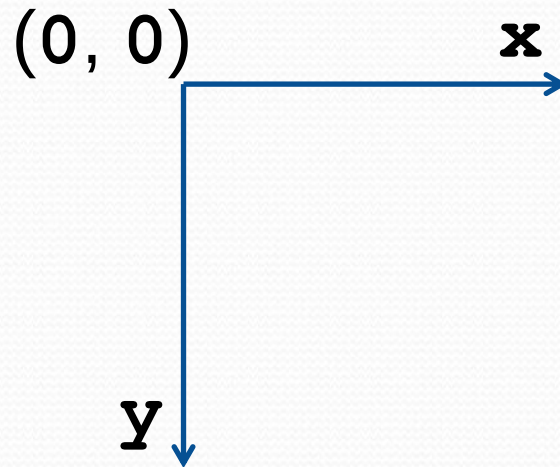
Painting GUI Components



Painting GUI Components

Warning:

- the location $(0, 0)$ is in the **upper-left corner** of the component!
- the horizontal **x** coordinate increases from left to right.
- the vertical **y** coordinate increases from **top to bottom**!



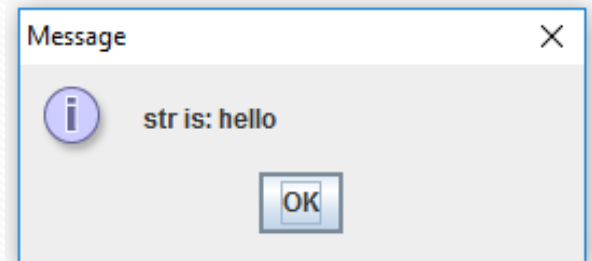
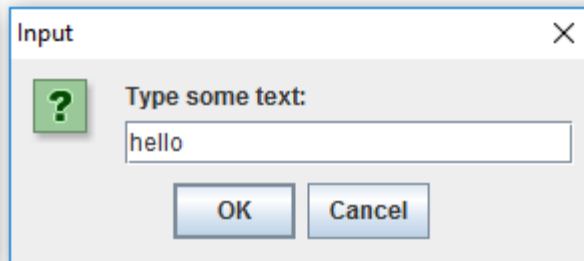
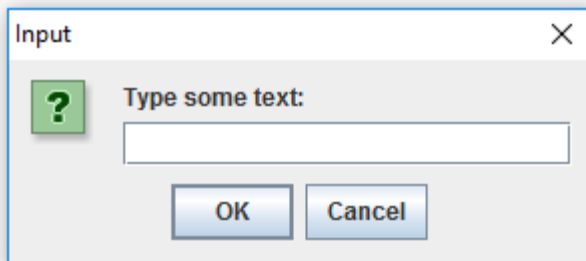
Simple input / output

- **JOptionPane** makes it easy to pop up a standard dialog box that prompts users for a value or informs them of something
- Using a **JOptionPane**:
 - **showInputDialog** for input.
 - **showMessageDialog** for output.
 - **showConfirmDialog** for Yes / No / Cancel questions.
 - Some options for title and style.
 - Default options are good enough.

Simple input / output

```
import javax.swing.JOptionPane;

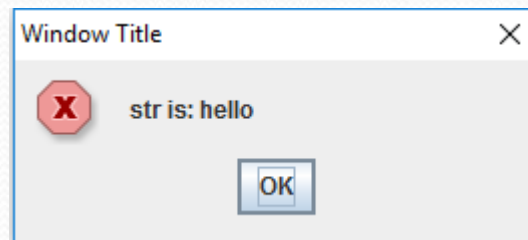
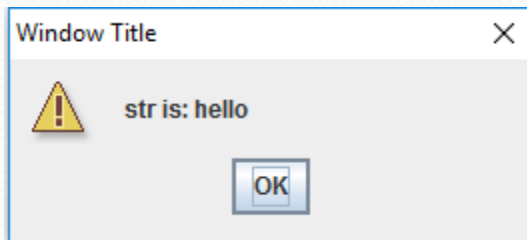
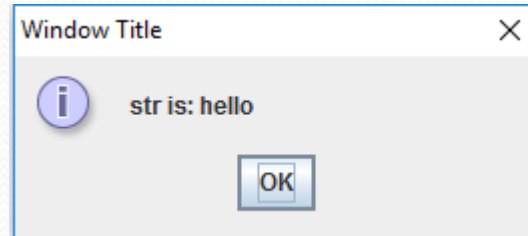
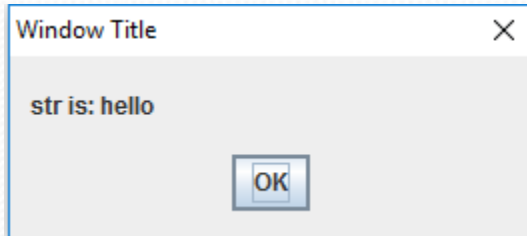
public class ShowInput {
    public static void main(String[] args) {
        // Input:
        // If the user clicks on "cancel" then str will be null.
        // If the user types something in the dialog and clicks
        // on "OK" then str will be what the user typed.
        String str = JOptionPane.showInputDialog("Type some text:");
        // Output:
        JOptionPane.showMessageDialog(null, "str is: " + str);
    }
}
```



Simple input / output

```
import javax.swing.JOptionPane;
public class TestIO {
    public static void main(String[] args) {
        // Input:
        String str = JOptionPane.showInputDialog(null, "Type some text:",
            "Window Title", JOptionPane.QUESTION_MESSAGE);
        // Output:
        JOptionPane.showMessageDialog(null, "str is: " + str, "Window Title",
            JOptionPane.PLAIN_MESSAGE);
        JOptionPane.showMessageDialog(null, "str is: " + str, "Window Title",
            JOptionPane.INFORMATION_MESSAGE);
        JOptionPane.showMessageDialog(null, "str is: " + str, "Window Title",
            JOptionPane.WARNING_MESSAGE);
        JOptionPane.showMessageDialog(null, "str is: " + str, "Window Title",
            JOptionPane.ERROR_MESSAGE);
    }
}
```

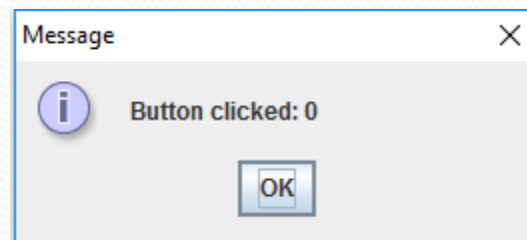
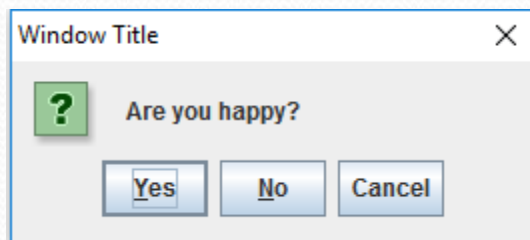
Simple input / output



Simple input / output

```
import javax.swing.JOptionPane;

public class ShowConfirm {
    public static void main(String[] args) {
        // Show a yes / no / cancel dialog.  Styles are: YES_NO_OPTION,
        // YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION, DEFAULT_OPTION (OK only).
        // The result is an integer indicating which button was clicked.
        int result = JOptionPane.showConfirmDialog(null, "Are you happy?",
            "Window Title", JOptionPane.YES_NO_CANCEL_OPTION);
        // Output:
        JOptionPane.showMessageDialog(null, "Button clicked: " + result);
    }
}
```



Event-Dispatching Thread

- In the previous examples, we invoke the constructor directly in the entry `main()` method to setup the GUI components.
- The constructor will be executed in the so-called "Main-Program" thread. This may cause multi-threading issues (such as unresponsive user-interface and deadlock).

```
public class MyFrame extends JFrame {  
    public MyFrame() {...}  
    // methods  
  
    public static void main(String[] args)  
    {  
        new MyFrame();  
    }  
}
```

Event-Dispatching Thread

- Swing uses threads:
 - One **main thread** running the **main** method.
 - One **event dispatch thread** automatically taking care of graphical events (mouse clicks, etc.)
 - More threads if you create them...
- Most Swing components can only be used by **one thread at a time**.
- Since the event dispatch thread is already using the Swing components, all of **your graphical code must be executed by the event dispatch thread** too! No choice.

Event-Dispatching Thread

To execute graphical code on the event dispatch thread, you must:

1. Create a class that implements the **Runnable** interface for thread code with a **public void run()** method.
2. Put all your graphical code inside the **run** method.
3. Create an object from that class.
4. Give the object to the event dispatch thread using the **javax.swing.SwingUtilities.invokeLater** method.

Event-Dispatching Thread

```
import javax.swing.JFrame;
class MyThreadCode implements Runnable {
    @Override
    public void run() {
        JFrame f = new JFrame("Frame Title");
        f.setSize(400, 300);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }
}

public class Test {
    public static void main(String[] args) {
        MyThreadCode m = new MyThreadCode();
        javax.swing.SwingUtilities.invokeLater(m);
    }
}
```

Event-Dispatching Thread

In practice it is easier to use an **anonymous class**:

```
import javax.swing.JFrame;
public class Test {
    public static void main(String[] args) {
        // The code inside the {} is the same code as before. It is
        // the code of an anonymous class (no name) that implements
        // the same Runnable interface as before.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                JFrame f = new JFrame("Frame Title");
                f.setSize(400, 300);
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                f.setVisible(true);
            }
        }); // End of anonymous class.
    }
}
```

Event-Dispatching Thread

It is even easier to **create your own frame subclass**:

```
import javax.swing.JFrame;
public class MyFrame extends JFrame {
    public MyFrame() {
        // We can use the setTitle method to set the title, or call
        // directly the constructor of the superclass JFrame with
        // the title: super("MyFrame Title");
        this.setTitle("MyFrame Title");
        this.setSize(400, 300);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }
}
```

Event-Dispatching Thread

And then **always use the same code to solve the event dispatch thread problem**, with a simple anonymous class:

```
public class Test {  
    public static void main(String[] args) {  
        javax.swing.SwingUtilities.invokeLater(new Runnable()  
        {  
            @Override  
            public void run() {  
                new MyFrame();  
            }  
        }));  
    }  
}
```

Summary

- AWT, Swing, JavaFX
- Frames
- Buttons
- Layout managers
- Colors and fonts
- Panels
- Simple input / output