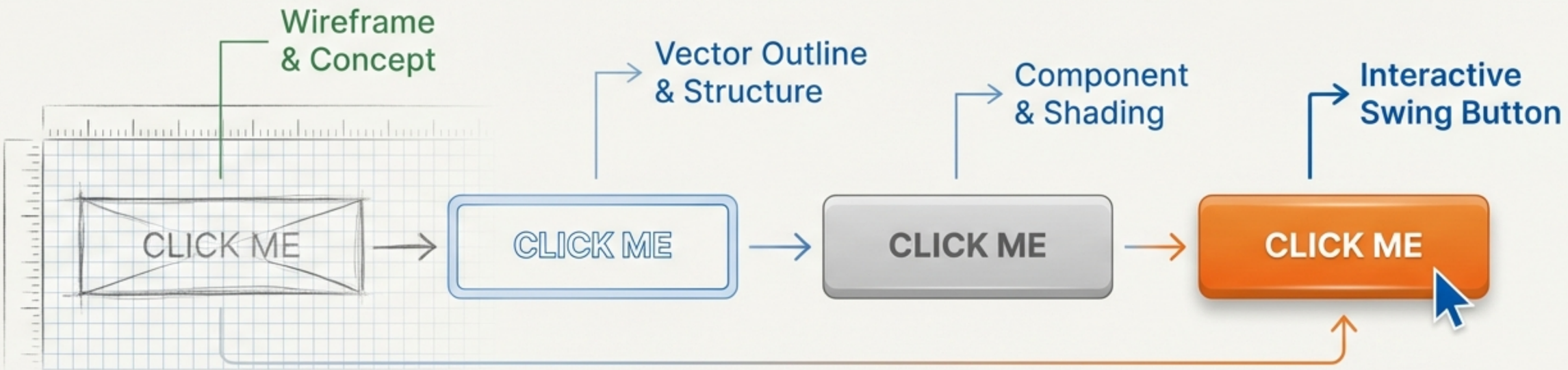
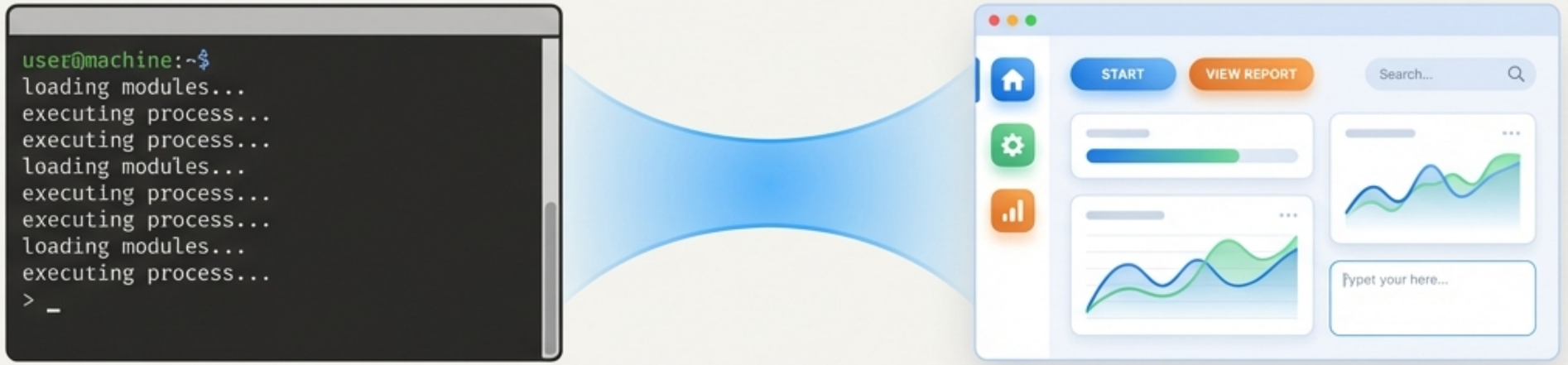
A Graphic Story: An Introduction to Java GUI with Swing

From a Blank Canvas to an Interactive World



Why Command-Line Isn't Enough



- Applications for ***other*** **people** need a graphical interface.
- Even tools you build for ***yourself*** are better with a GUI.
- Command-line apps are often **weak**, **inflexible**, and **unfriendly**.

- **GUIs are the bridge** between your code and the user, making applications intuitive, engaging, and powerful.

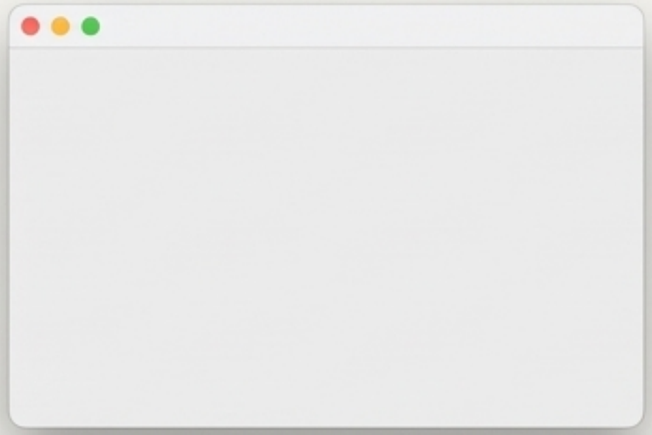
Our Mission Today

We will build our first **interactive application**, piece by piece.
You will go from a **blank screen** to a **program** that **responds to you**.

The Anatomy of a GUI

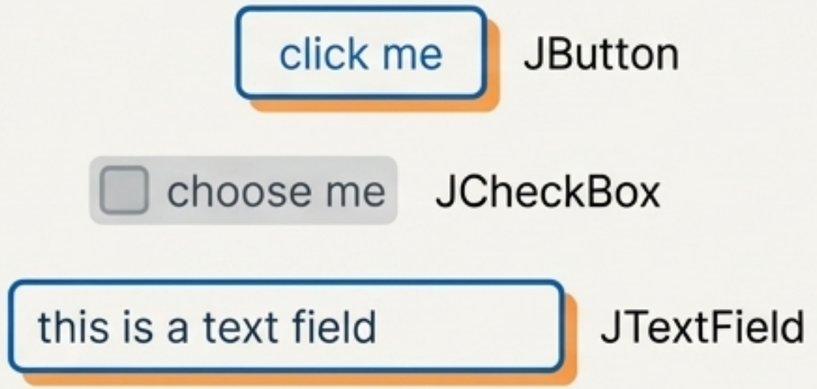
The Canvas (JFrame)

- A JFrame is the object that represents a window on the screen.
- It's where you put everything else: buttons, text fields, menus.
- It has the standard window controls (minimize, maximize, close) for your operating system.



The Actors (Widgets / JComponent)

- 'Widgets' (or more correctly, Components) are the interactive elements inside your window.
- They all live in the `javax.swing` package.
- **Examples**: `JButton`, `JCheckBox`, `JTextField`, `JLabel`, `JList`, `JTextArea`.



The 4 Magic Steps to Your First GUI

1. Make a frame (a JFrame)

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

2. Make a widget (e.g., a JButton)

```
JButton button = new JButton("click me");
```

3. Add the widget to the frame's content pane

```
frame.getContentPane().add(button);
```

* Note: You don't add things to the frame directly. Think of the frame as the trim around the window; you add things to the window pane.

4. Display it (give it a size and make it visible)

```
frame.setSize(300, 300);
frame.setVisible(true);
```

Code Breakdown: Our First Program (SimpleGui1)

```
import javax.swing.*;

public class SimpleGui1 {
    public static void main(String[] args) {

        JFrame frame = new JFrame();
        JButton button = new JButton("click me");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(button);

        frame.setSize(300, 300);

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

Don't forget to import the swing package.

Make a frame.

Make a button. You can pass the text to the constructor.

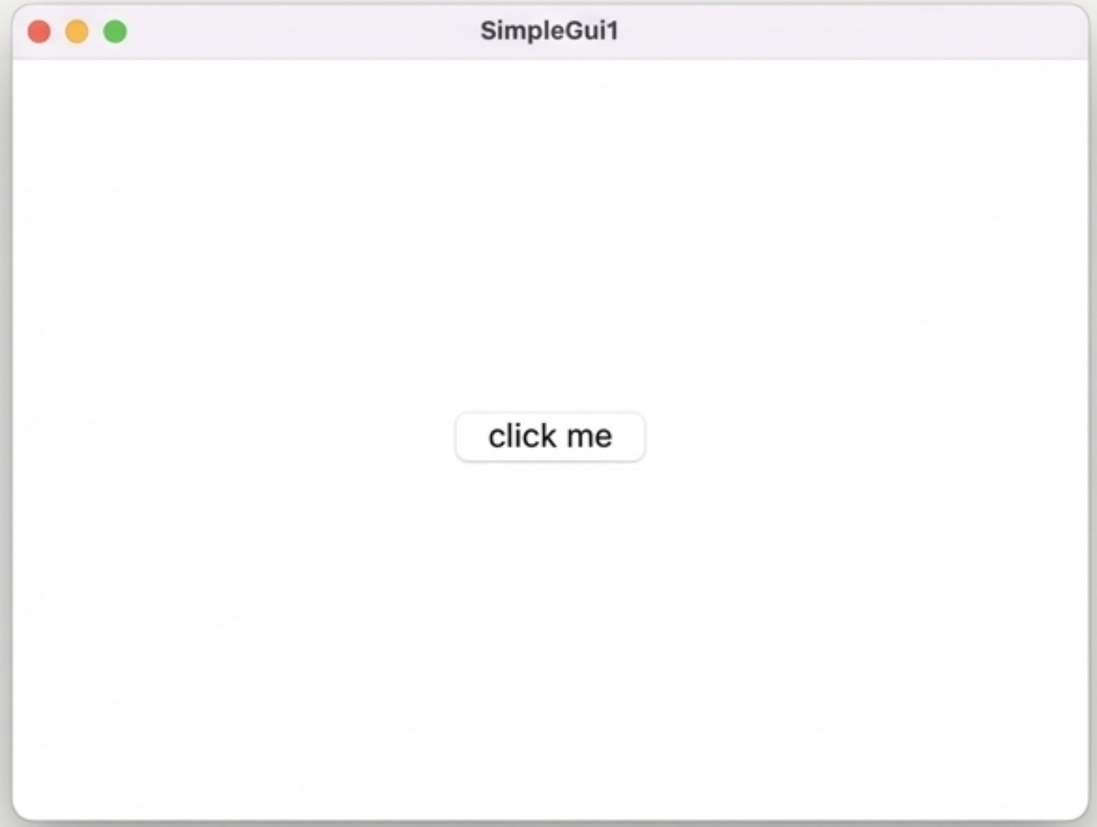
This makes the program quit when you close the window.

Add the button to the frame's content pane.

Give the frame a size in pixels.

Finally, make it visible! If you forget this, you won't see anything.

It’s Alive! (Almost...)



Success! We’ve created our first window.

The First Challenge

When you press the button, it shows that ‘pushed in’ look, but... *nothing happens*.

How do we get the button to *do something specific* when the user clicks it?

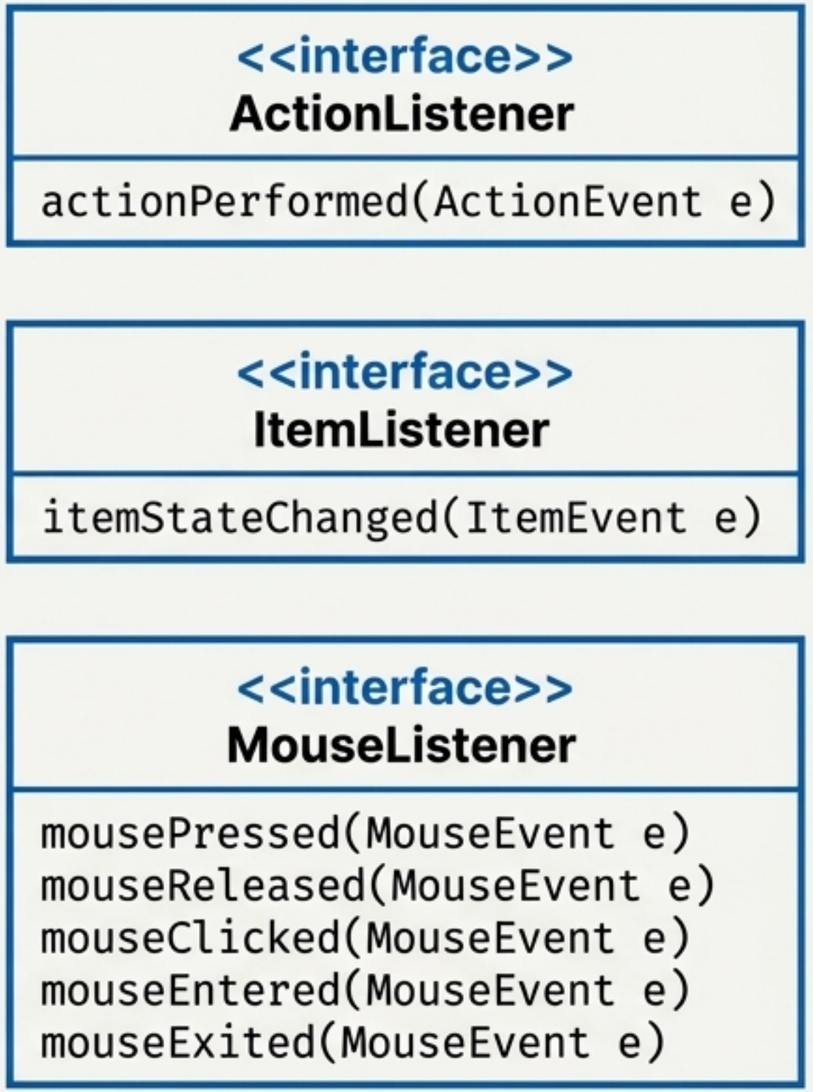
This leads us to... **Interactivity**.

The Event Handling Model: How Your Program Listens

***Flow*:** User clicks button (**Source**) → Button creates **'ActionEvent'** (**Event**) → Button calls a method on the **Listener** object (**Listener**) → Your **code runs.**

The Listener's Contract: Implementing the Interface

- To listen for an event, your class must implement the corresponding listener interface. This is the bridge between you (the listener) and the button (the event source).
- Every event type has a matching listener interface.
  - `ActionEvent` → `ActionListener`
  - `MouseEvent` → `MouseListener`
  - `ItemEvent` → `ItemListener`
- Implementing an interface means you *must* provide the code for all of its methods. For `ActionListener`, there's only one: `actionPerformed(ActionEvent e)`.
- This method is the "call-back": The button calls *your* method when the user clicks.



Code Breakdown: Making the Button Respond (SimpleGui2)

1

This says, “an instance of SimpleGui2 IS-A ActionListener”.

2

This tells the button: “Add *me* to your list of listeners.” The button will now call us back when an action occurs.

3

This is the actual event-handling method. The button calls this method to let you know an event happened.

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

public class SimpleGui2 implements ActionListener {
    private JButton button;

    public static void main(String[] args) {
        SimpleGui2 gui = new SimpleGui2();
        gui.go();
    }

    public void go() {
        JFrame frame = new JFrame();
        button = new JButton("click me");

        button.addActionListener(this);

        frame.getContentPane().add(button);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 300);
        frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
        button.setText("I've been clicked!");
    }
}
```

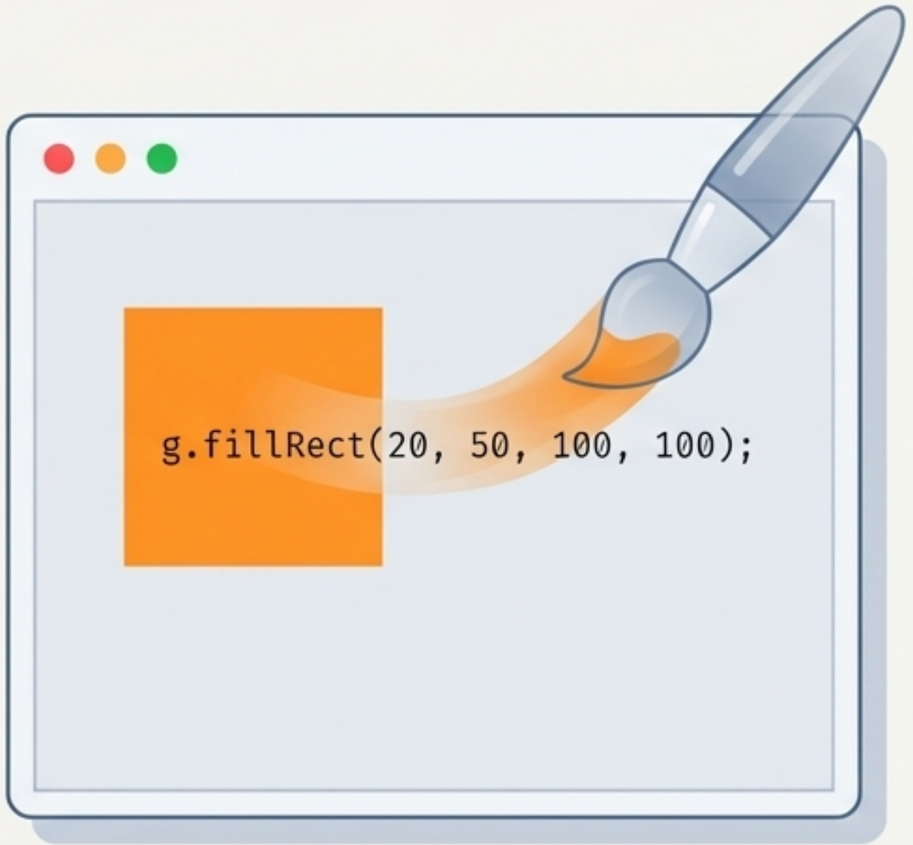
Beyond Widgets: Your Own Drawing Canvas

What if you want to draw something unique—a shape, a chart, a game character?

The Solution: Make your own paintable widget.

1. **Create a subclass of `JPanel`:**
   ```
   class MyDrawPanel extends JPanel { … }
   ```

2. **Override one method: `paintComponent()`:**
   ```
   public void paintComponent(Graphics g) { … }
   ```



The Golden Rule of `paintComponent()`

- You put all your drawing code inside this method.
- **You override** this method, **but you NEVER call it yourself.**
- The system calls it when your component needs to be drawn (e.g., when the window first appears, is resized, or is uncovered).

Your Magic Paintbrush: `Graphics` and `Graphics2D`

The `Graphics g` Parameter

Think of `g` as a painting machine. You tell it what color, shape, and coordinates to use.

```
g.setColor(Color.orange); // Basic Painting
g.fillRect(20, 50, 100, 100);
```

```
Image image = new ImageIcon("catzilla.jpg").getImage();
// Drawing an Image
g.drawImage(image, 3, 4, this);
```

Unlocking More Power

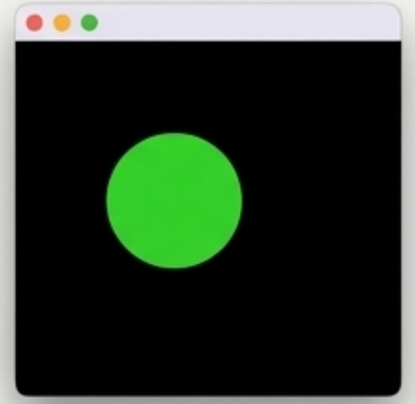
The `g` object is secretly an instance of the more powerful `Graphics2D` class.

- Why care? `Graphics2D` has advanced methods: `rotate()`, `scale()`, `setPaint()` for gradients, etc.
- How to use it: "You must cast it."

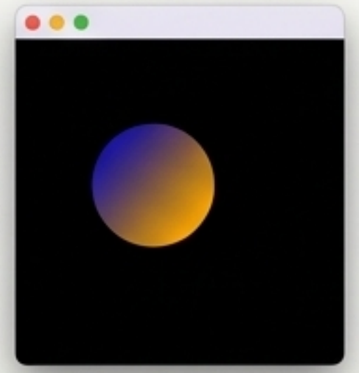
```
Graphics2D g2d = (Graphics2D) g;
GradientPaint gradient = new GradientPaint(70, 70,
       Color.blue, 150, 150, Color.orange);
g2d.setPaint(gradient);
g2d.fillOval(70, 70, 100, 100);
```

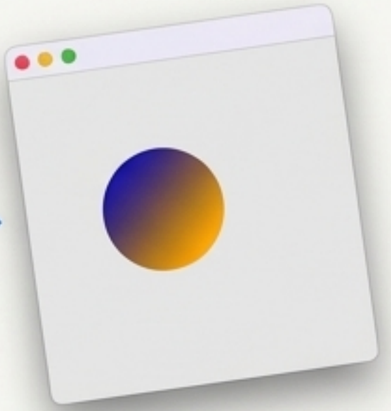
Visual Comparison

Left



Right

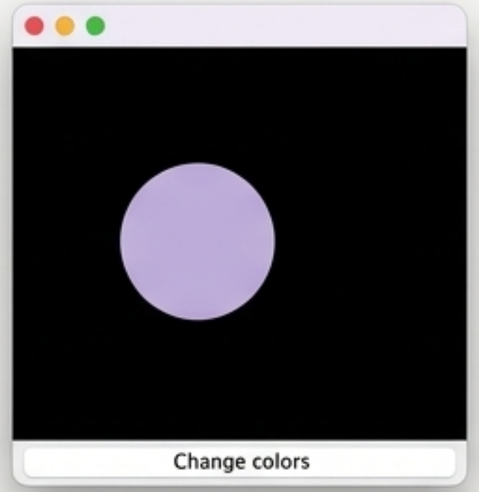




The ‘Aha!’ Moment: Connecting Events to Graphics

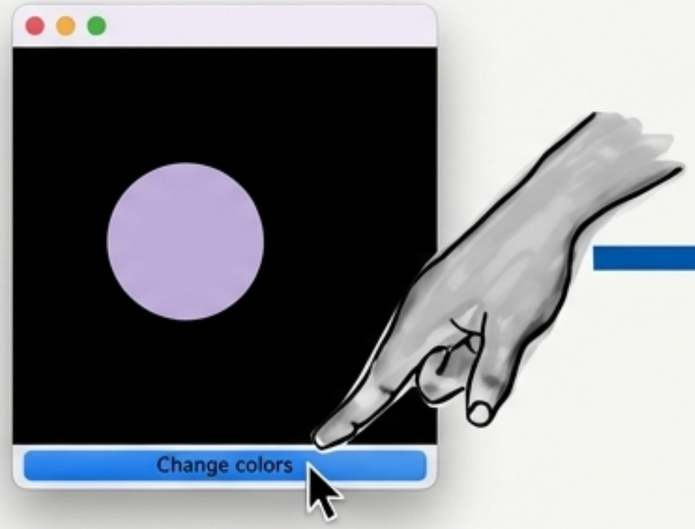
Goal: Make a circle change to a random color each time we click a button.

Step 1: Initial State



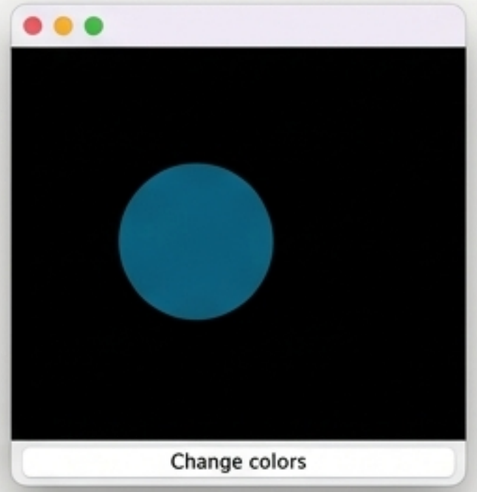
Application is waiting for user action.

Step 2: User Action



‘`actionPerformed`’ is called and executes one key line: `frame.repaint();`

Step 3: The Result



The system calls ‘`paintComponent()`’ again, which redraws the circle with a new random color.

The Next Challenge: Managing a Complex World

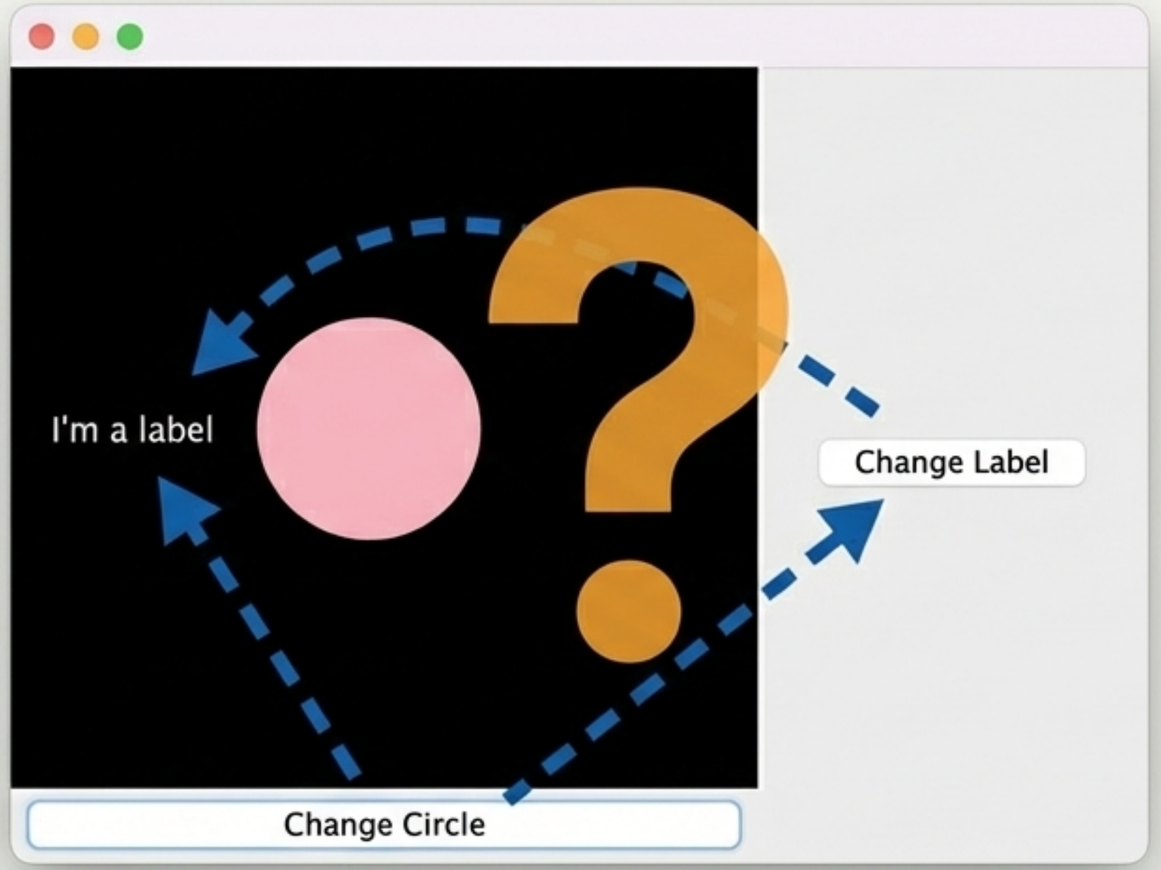
The Scenario:

Our app is growing. We need two buttons that do two different things.

- Button 1: Changes the color of a circle.
- Button 2: Changes the text on a label.

The Problem:

If we register one listener (`this`) with both buttons, how does our single `actionPerformed()` method know which button was clicked? We can't implement the same method twice in one class.



Solution 1: Ask the Event Object (`getSource()`)

The Logic:

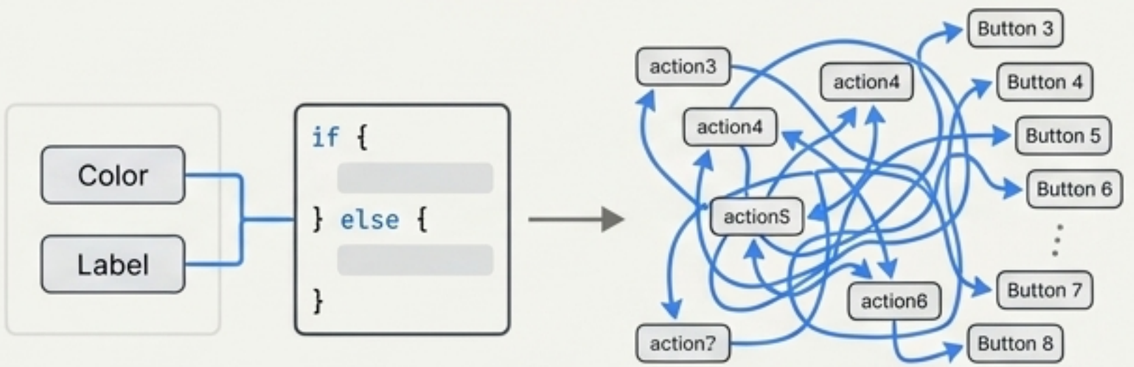
The `ActionEvent` object knows who sent it. We can query the event to find out which component was the source.

```
public void actionPerformed(ActionEvent event) {
    if (event.getSource() == colorButton) {
        frame.repaint();
    } else if (event.getSource() == labelButton) {
        label.setText("That hurt!");
    }
}
```

Identifies the button that fired the event

The Flaw:

- It works, but it's not very Object-Oriented. One method is responsible for many different actions.
- As you add more buttons, the `if/else` chain becomes long, messy, and hard to maintain.
- If you need to change how one source is handled, you have to mess with everybody's event handler.

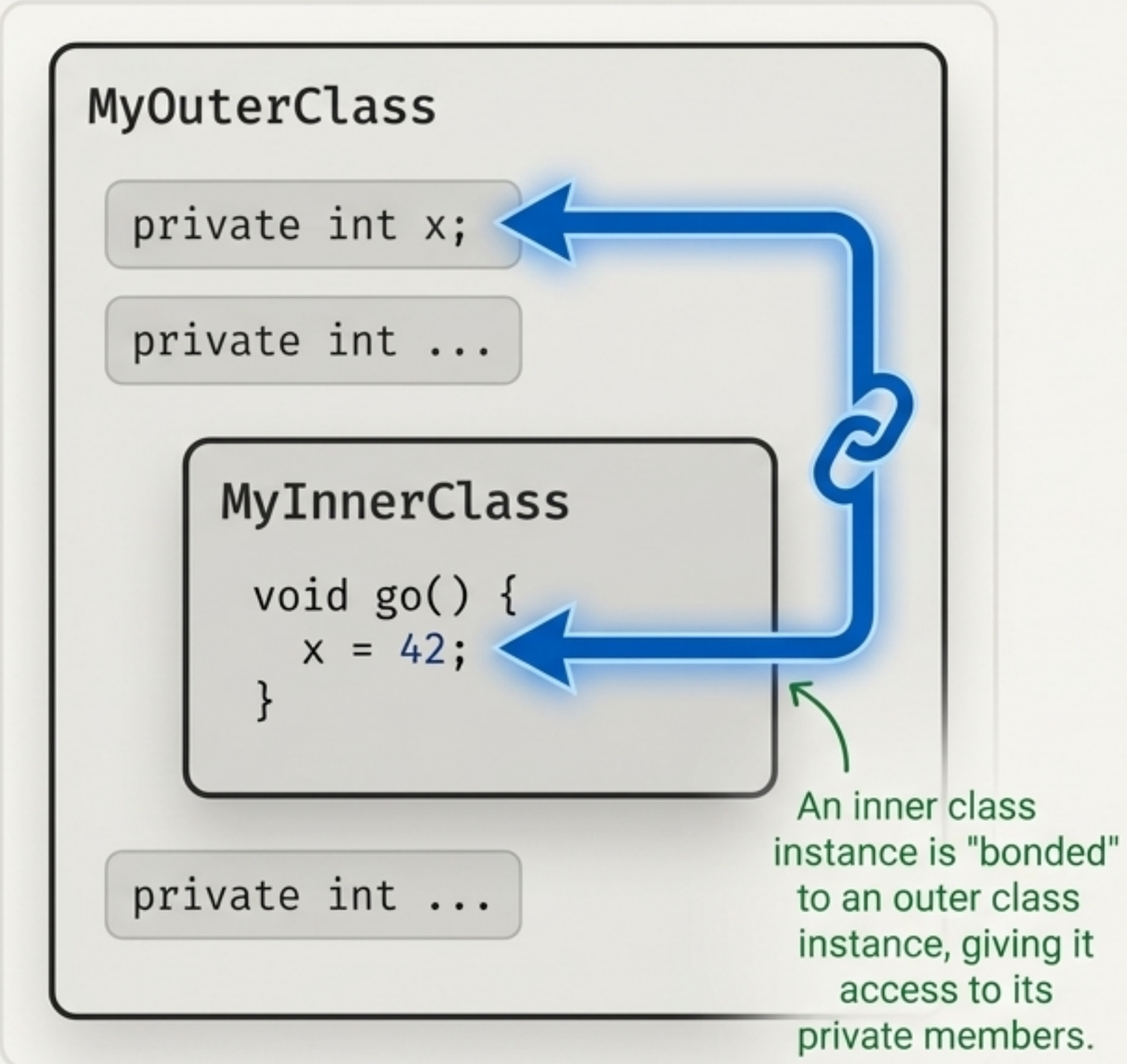


Tangled & Hard to Maintain

Solution 2: A Class Within a Class (Inner Classes)

What it is: A class defined inside the curly braces of another class.

The Superpower:
An inner class instance is "bonded" to an outer class instance. It can access all the methods and variables of the outer class, **even the private ones**, as if they were its own.



How it Works:
We create two separate inner classes, both implementing `ActionListener`. Each class has its own, focused `actionPerformed` method.

We register an instance of `LabelListener` with the label button, and `ColorListener` with the color button.

Solution 3: The Modern Way (Lambda Expressions)

The Insight:
`ActionListener` is a **Functional Interface**—it has only one abstract method (`actionPerformed`). This makes it a perfect candidate for a lambda.

Left Column: Inner Class (The Old Way - 5 lines)

```
class LabelListener implements ActionListener {
   public void actionPerformed(ActionEvent event) {
      label.setText("Ouch!");
   }
}

labelButton.addActionListener(new LabelListener());
```

Right Column: Lambda Expression (The New Way - 1 line)

```
labelButton.addActionListener(event -> label.setText("Ouch!"));
```

Why it's better:

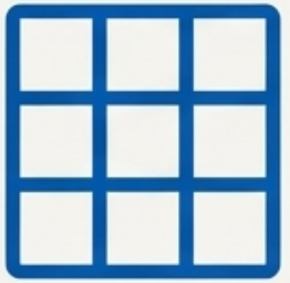
- **Concise:** Eliminates boilerplate code.
- **Clearer:** The action (`label.setText`) is written directly where the listener is registered. The intent is immediately obvious.

The Chaos of Layout

The Problem

What happens when you just add components to a frame? By default, `JFrame` uses `BorderLayout`. If you add multiple components without specifying a region, they stack on top of each other, with only the last one visible. It quickly becomes a jumble.

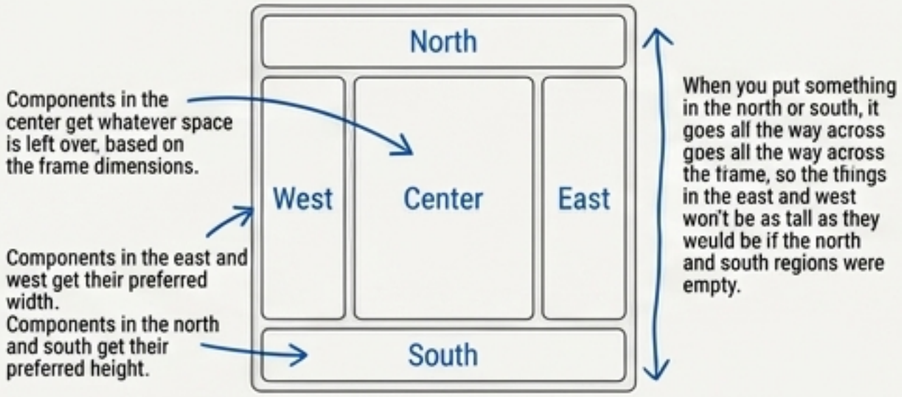
The Solution: Layout Managers



- A Java object that controls the size and placement of components within a container.
- Each container can have its own layout manager.
- This allows you to create complex, nested layouts that adapt gracefully when the window is resized.

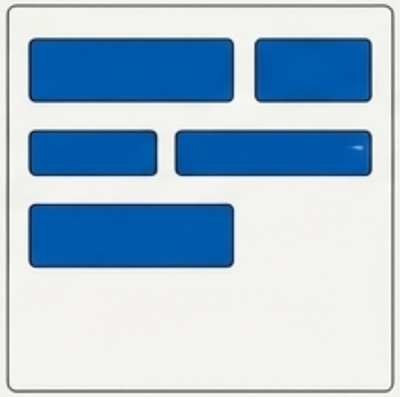
Meet the Layout Managers

BorderLayout (The Organizer)



- **Metaphor:** Divides the space into five strict regions: `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER`.
- **Behavior:** Components often don't get their preferred size. `CENTER` gets what's left.
- **Default for:** `JFrame`.

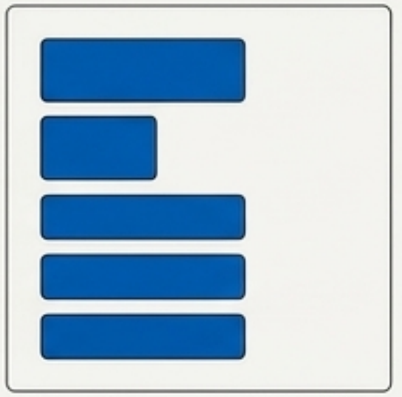
FlowLayout (The Word Processor)



Components added left to right, wrapping to a new line when needed

- **Metaphor:** Lays components out left-to-right, wrapping to the next line when full.
- **Behavior:** Respects each component's preferred size.
- **Default for:** `JPanel`.

BoxLayout (The Stacker)



Components added top to bottom, one per 'line' if it's set up vertically

- **Metaphor:** Stacks components in a single row or column.
- **Behavior:** Respects preferred size. Does NOT wrap automatically. Great for vertical button columns.
- **How to use:** `panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));`

Tying It All Together: A Simple Animation

How it Works

1. **The Canvas**: We use an **inner class** `MyDrawPanel` that extends `JPanel` to access the `x` and `y` instance variables from the main class.
2. **The Drawing**: The `paintComponent` method draws a circle at the current `x` and `y` coordinates. It first clears the panel to prevent "smearing".
3. **The Engine**: A **for loop** in the main `go()` method acts as our animation driver. In each iteration, it:
   - Increments the `x` and `y` coordinates.
   - Calls `drawPanel.repaint()` to schedule a redraw.
   - Pauses briefly with `TimeUnit.MILLISECONDS.sleep(50);`.

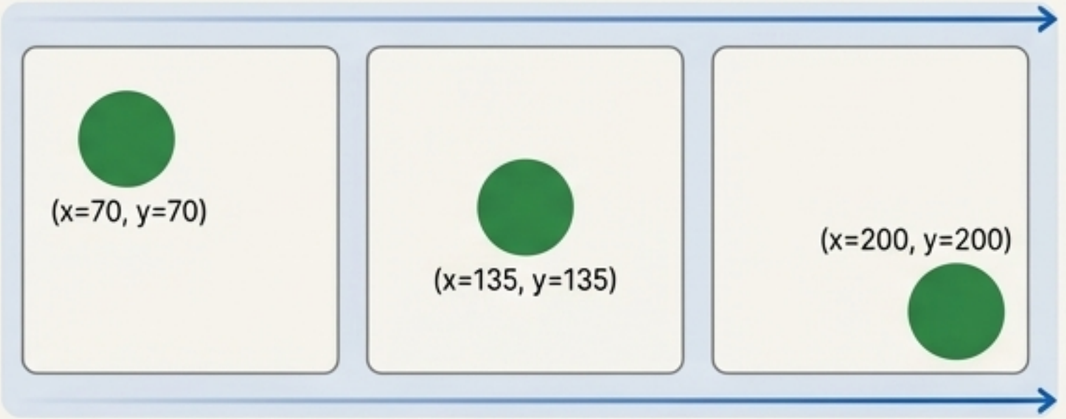
```
for (int i = 0; i < 130; i++) {
  xPos++; yPos++;
  drawPanel.repaint();
  try {
    TimeUnit.MILLISECONDS.sleep(50);
  } catch (Exception e) {}
}

class MyDrawPanel extends JPanel {
  public void paintComponent(Graphics g) {
    g.setColor(Color.green);
    g.fillOval(xPos, yPos, 40, 40);
  }
}
```

Increments xPos & yPos

Calls repaint()

Pauses

Animation Filmstrip

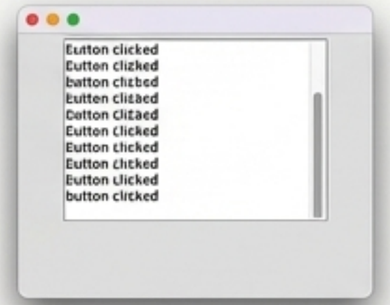


Frame 1 (Start) **Frame 2 (Midpoint)** **Frame 3 (End)**

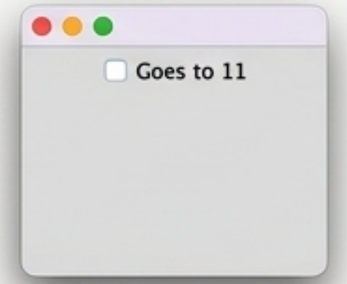
Your Swing Toolbox is Now Open

More Components to Explore

JTextField



JTextArea (in a JScrollPane)

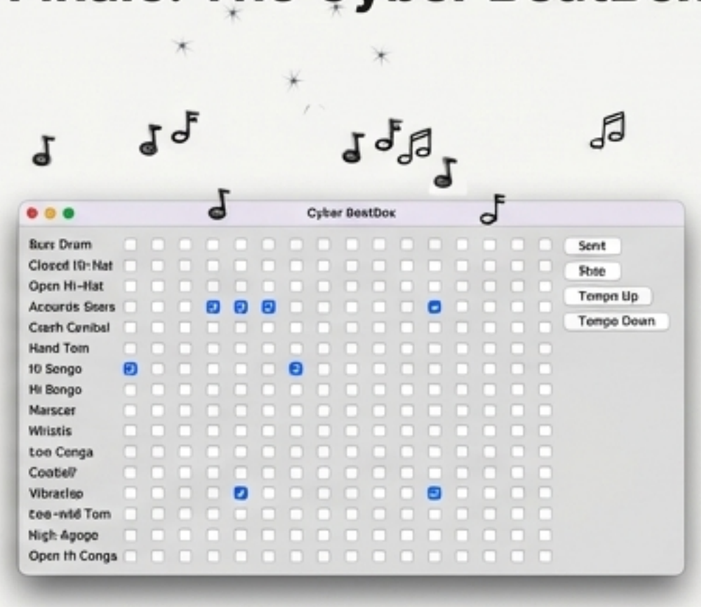


JCheckBox

JList

What You Can Build Now

Grand Finale: The Cyber BeatBox



This application uses everything we've learned:

- A `JFrame` with nested `JPanels` using `BorderLayout` and `GridLayout`.
- An array of `JCheckBox` components to create the drum pattern.
- `JButtons` with Lambda `ActionListeners` to start, stop, and change tempo.

It's a complex, fun, and powerful application—and you now have the foundational skills to build it.