GUI App Development in Java/Swing

Fernando Pedro Birra
Manuel Próspero dos Santos

Command line application

- No user interaction
- linear execution

program:

```
main()
{
	code;
	code;
	code;
	code;
	code;
	code;
	code;
	code;
	code;
	code;
	code;
	code;
}
```

Interactive console application

- User input
- non linear execution
- unpredictable order
- much idle time

program:

```
main()
{
    setup code;

    while(...)
    {
        get command;
        switch(command)
        {
            command1:
                code;
            command2:
                code;
            ...
        }
    }
}
```

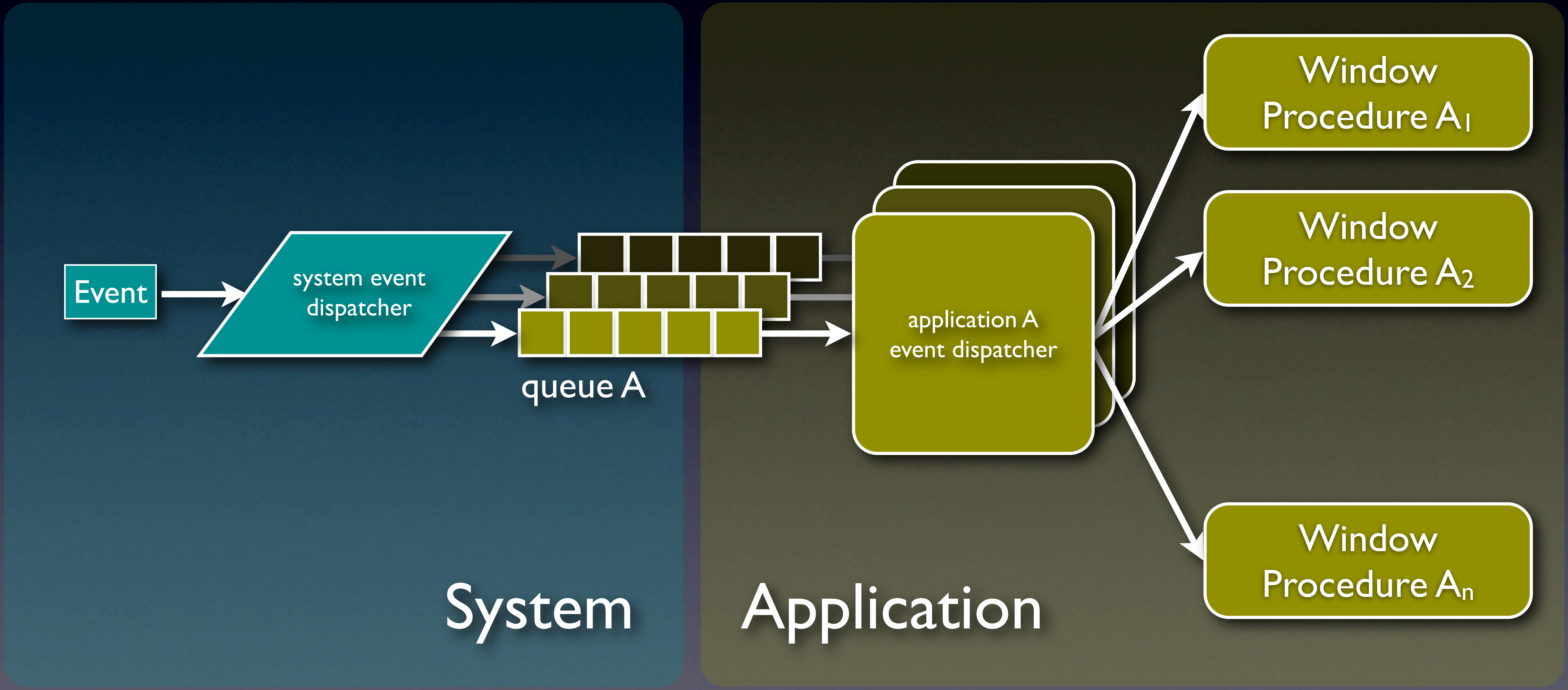
Interactive GUI application

- User input
- non linear execution
- unpredictable order
- much idle time
- callback procedures (to handle GUI events)

program:

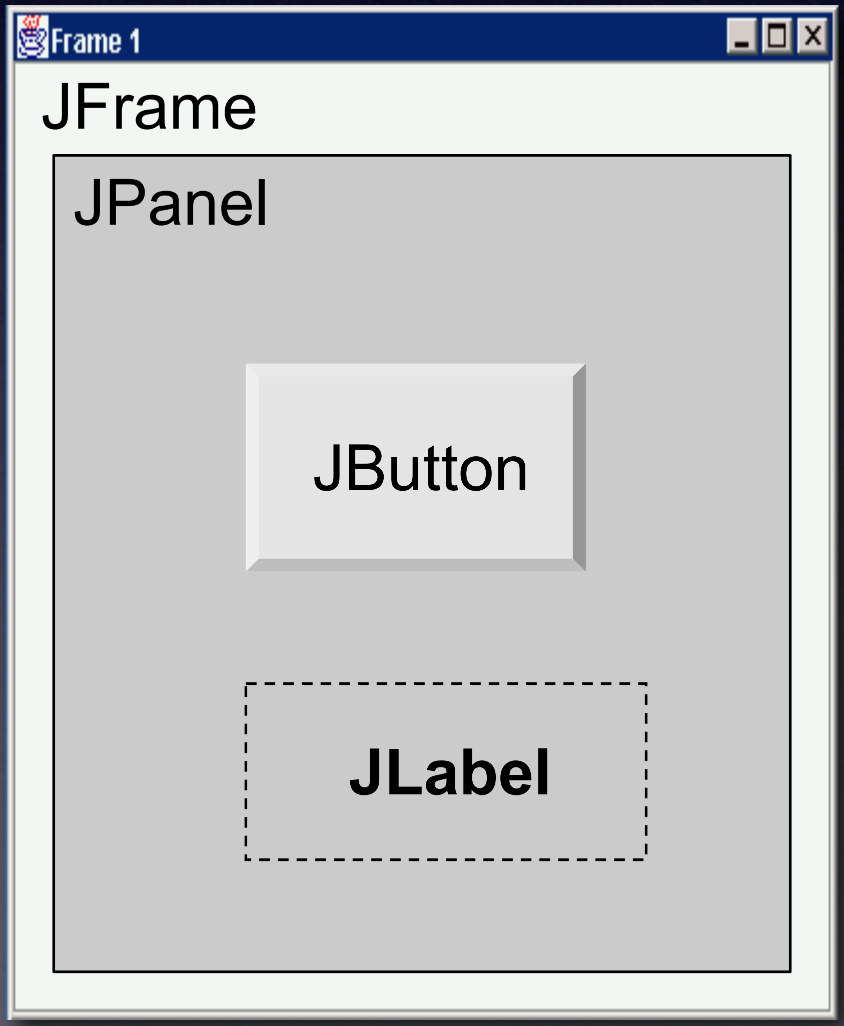
```
main()
{
        setup code;
        create gui;
        register callbacks;

        while(...)
        {
                get event;
                dispatch event;
        }
}

callback1(...){...}
callback2(...){...}
```

GUI Programming Model (C language)

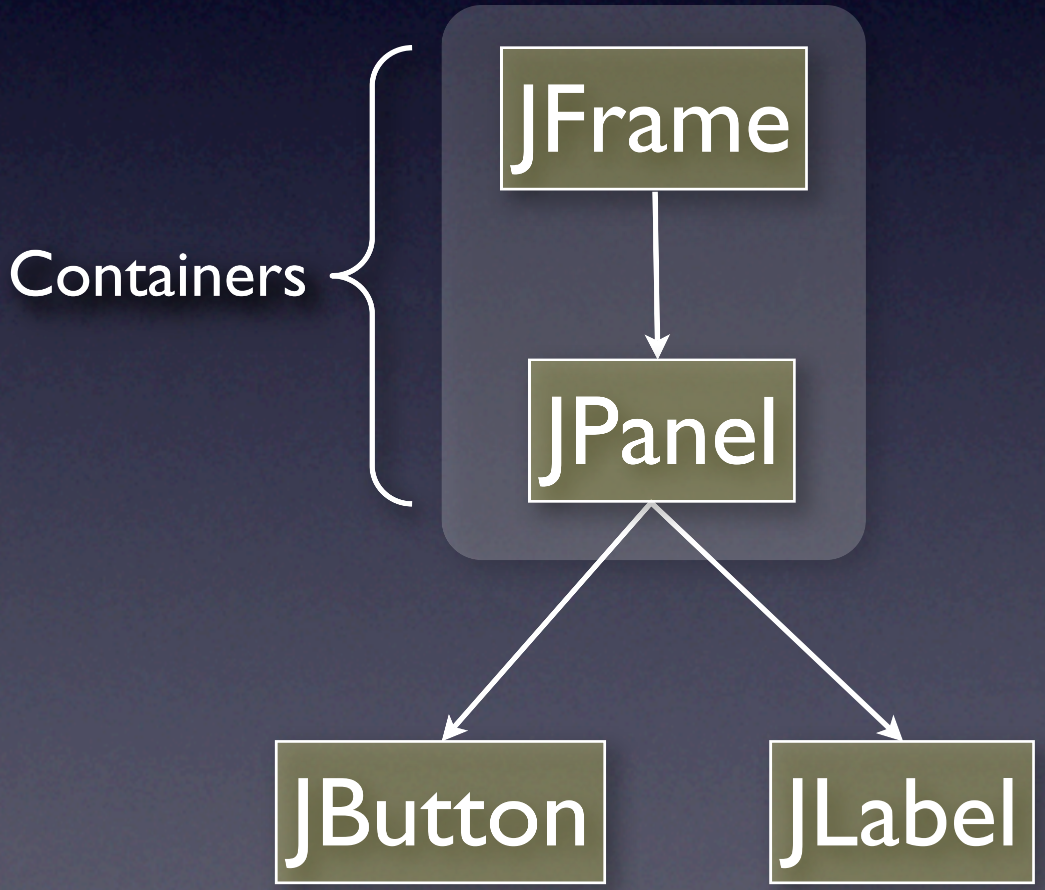


Anatomy of a Java GUI

Graphical User Interface



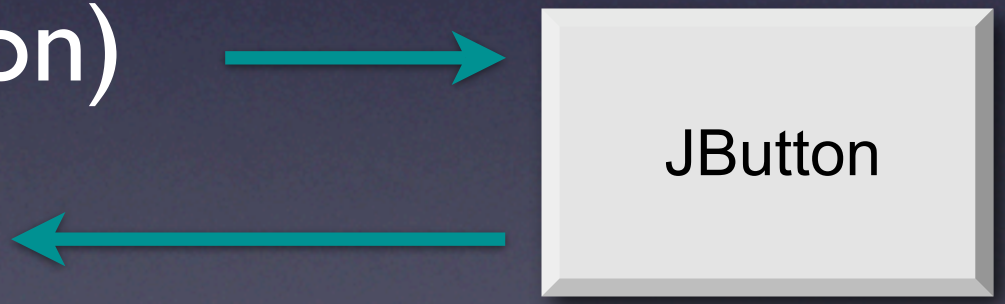
Internal Structure



Anatomy of a Java GUI Component

- GUI Components are modeled by classes (ex: JButton, JFrame, JPanel, etc)

- Methods (configuration)
- Events (behavior)

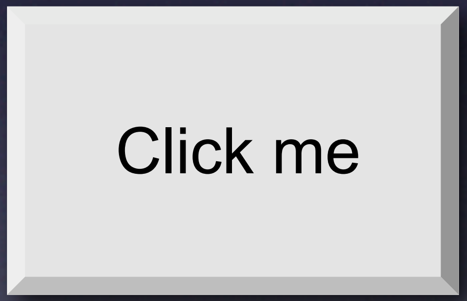


Using a GUI Component

1. Create it
   ```
   b = new JButton();
   ```

2. Configure it
   ```
   b.setText("Click me!");
   ```

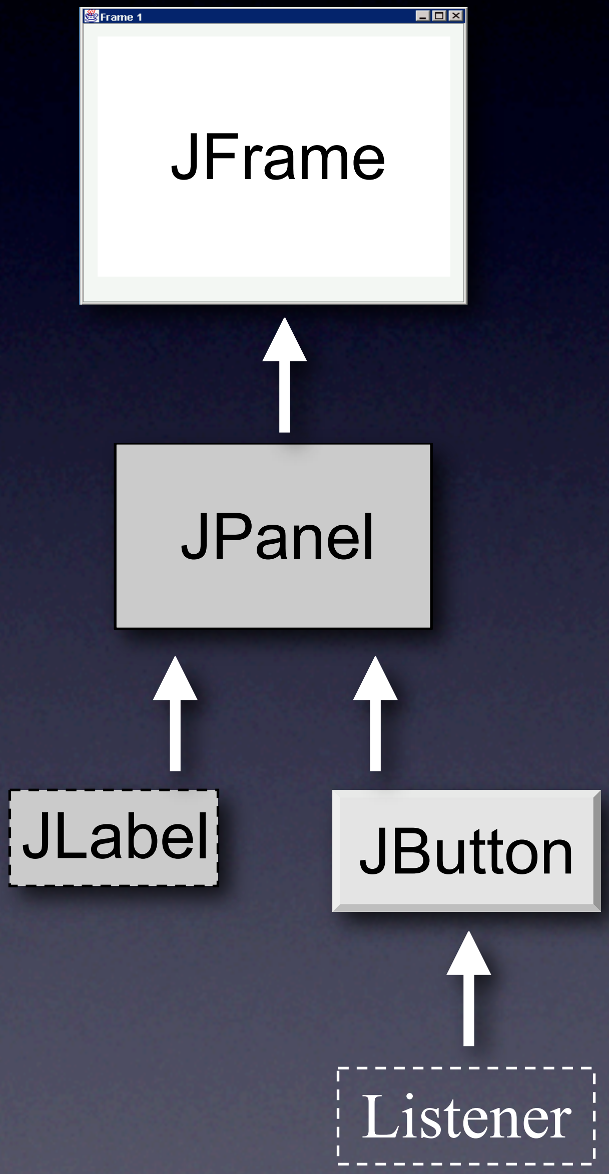
3. Add it to a parent container (if not JFrame)
   ```
   panel.add(b);
   ```



4. Listen to it
   Use ~~listeneres~~ to listen to events generated by the component.

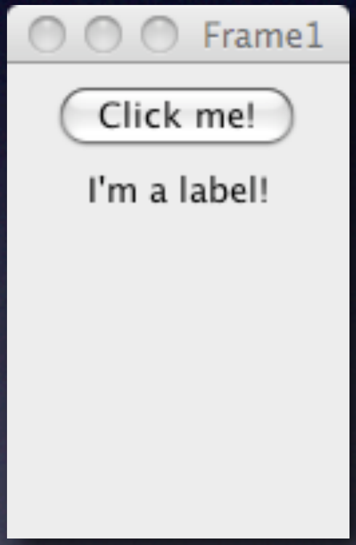
Building the Hierarchy

- Create:
  - frame
  - panel
  - components
  - listeners
- Add (bottom up):
  - listeners into components
  - components into panels
  - panel into frame



Code sample

```
JFrame frame = new JFrame(“Frame1”);
...
JPanel panel = new JPanel();
JButton button = new JButton("Click me!");
JLabel label = new JLabel("I'm a label!");
panel.add(button);
panel.add(label);
frame.setContentPane(panel);
...
```

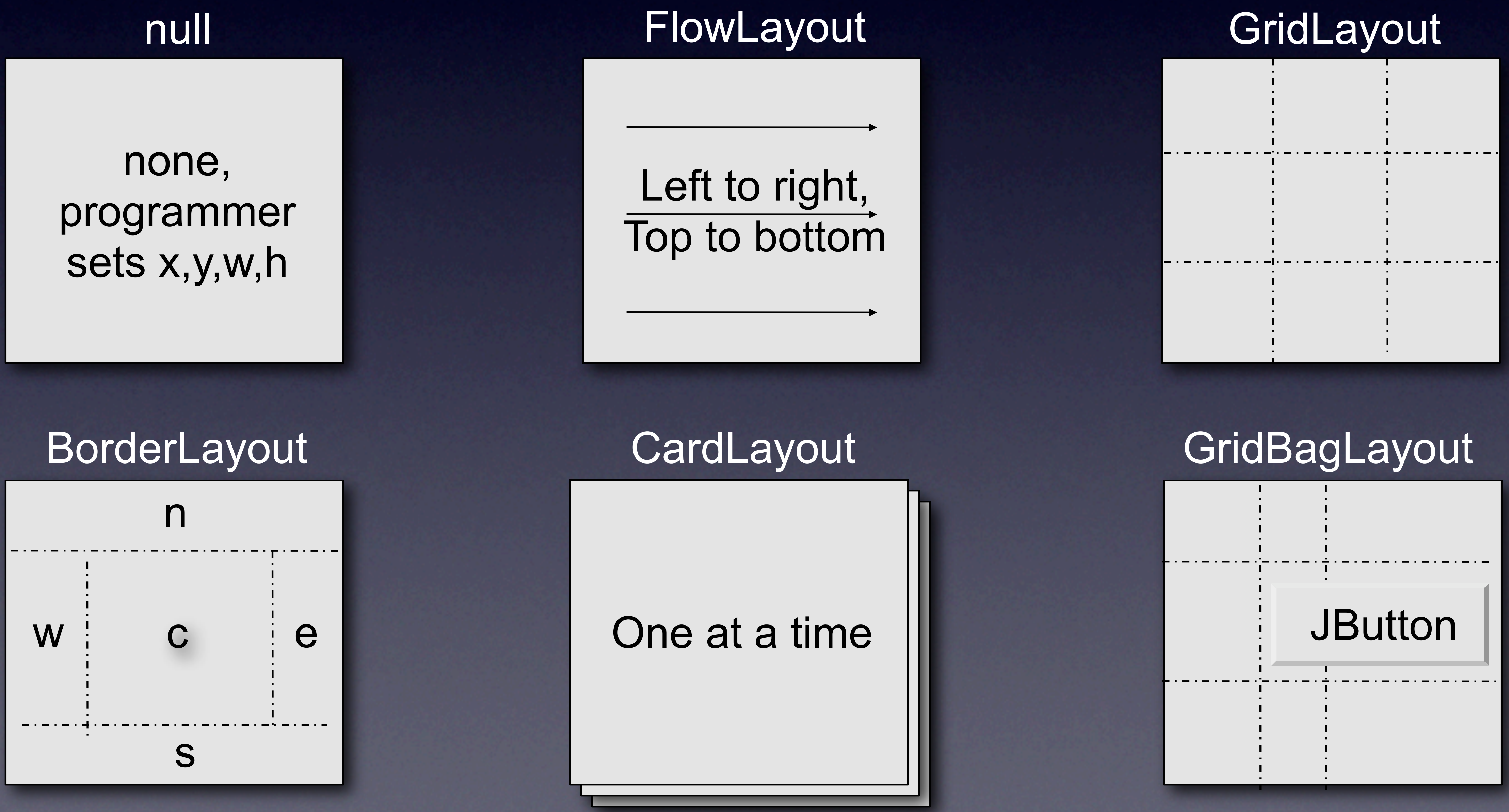


Full listing

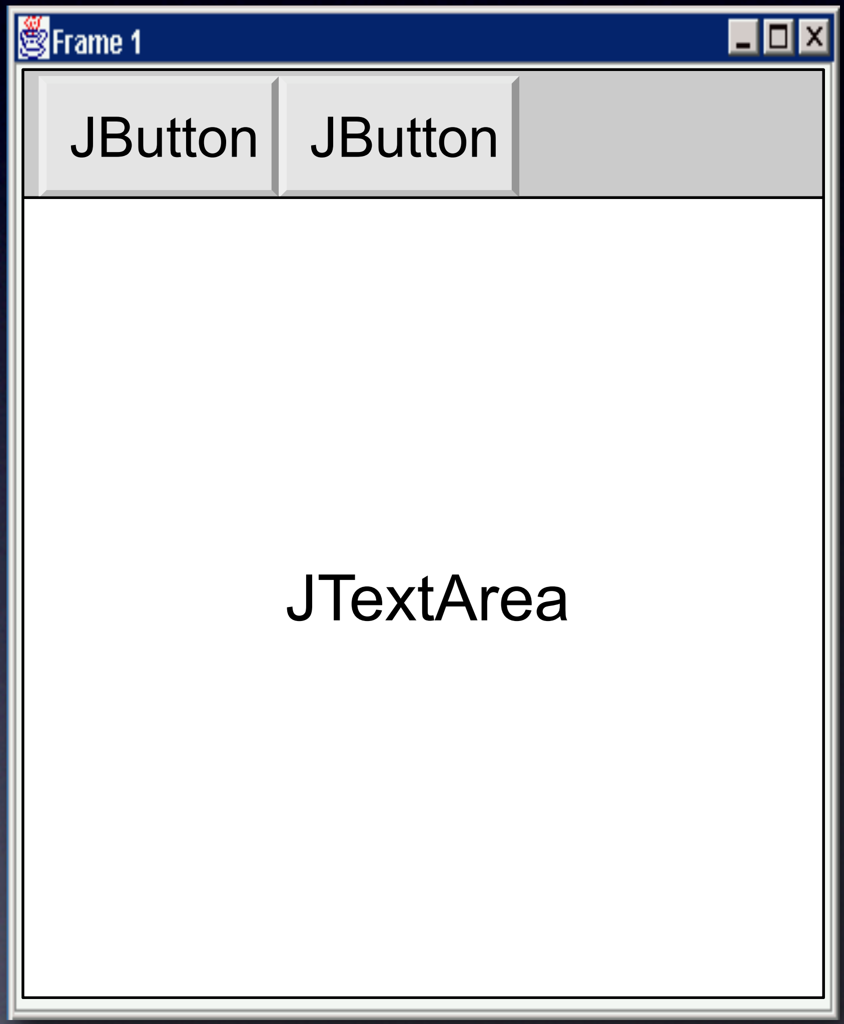
```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.JLabel;

public class Main {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Frame1");
        frame.setSize(100, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel();
        JButton button = new JButton("Click me!");
        JLabel label = new JLabel("I'm a label!");
        panel.add(button);
        panel.add(label);
        frame.setContentPane(panel);
        frame.setVisible(true);
    }
}
```

Layout Management

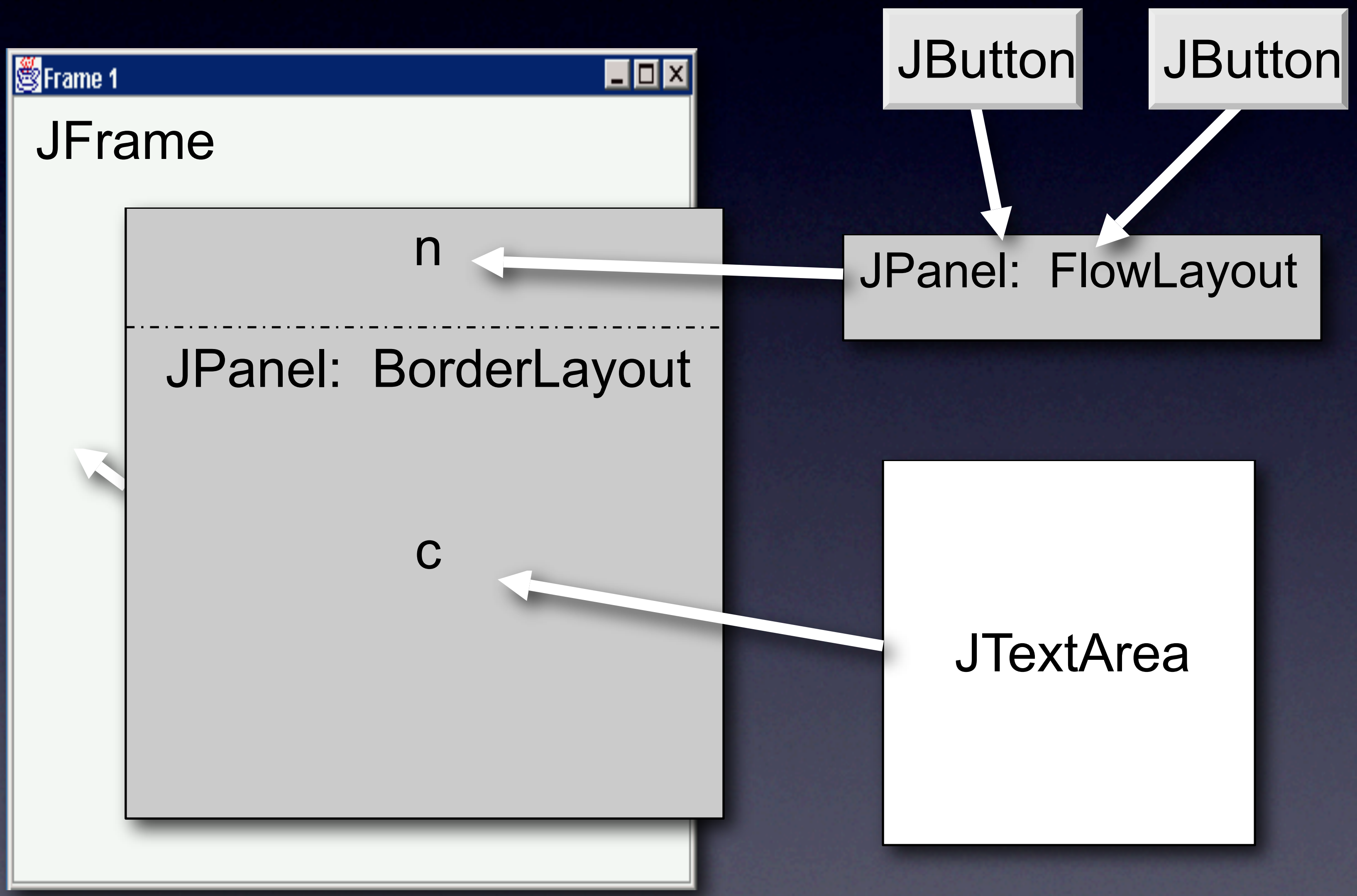
A layout manager automates the placement of components in a container:



Layout Combinations



Layout Combinations



Event handling with Swing

Event handling

- Events require you to use listeners (or adapters) and implement interfaces in order to receive notification of their occurence
- The listener object can be any, as long as the corresponding interface is implemented

Listener API

- Listeners must inherit from Java Listener base classes:

  ActionListener, KeyListener, MouseListener, WindowListener, ...

- MouseListener interface:

  mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()

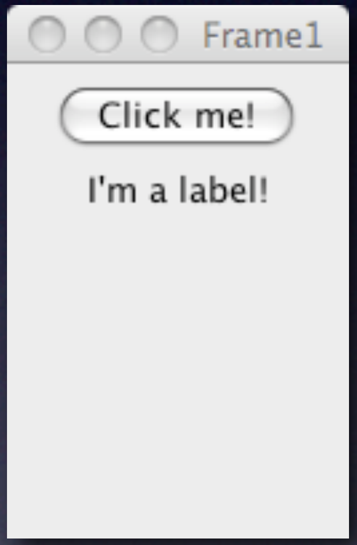
Listener: How To

1. Tell a component who's willing to receive its events
   - Provide a reference to a listener object
   - `btn1.addMouseListener(new MyMouseListener());`
2. Receive events generated by the component
   - component will call callback code on provided listener
   - `MyMouseListener.mouseClicked(event);`

Simple button click Example (I)

anonymous inner class

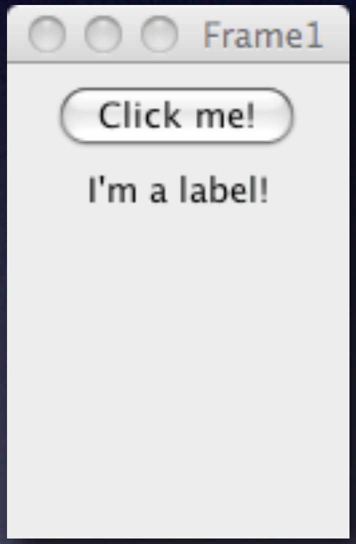
```
...
JButton button = new JButton("Click me!");

ActionListener listener = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.println(“Thank you!”);
  }
};

button.addActionListener(listener);
...
```



Simple button click Example (II)

implementing the interface

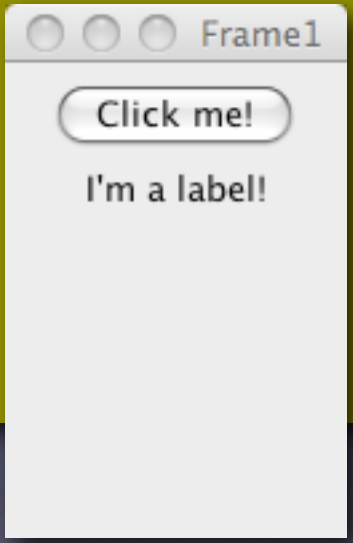
```
...
JButton button = new JButton("Click me!");
~~lister~~ = new MyListener();
button.addActionListener(listener);

class MyListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Thank You!");
  }
};
...
```



Simple button click Example (III)

implementing the interface at a top level class

```
public class MyFrame extends JFrame, implements ActionListener
{
  ...
  JButton button = new JButton("Click me!");
  button.addActionListener(this);

  public void actionPerformed(ActionEvent e){
    System.out.println(“Thank You!”);
  }
}
```



Simple button click Example (III)

implementing the interface at a top level class

```
public class MyFrame extends JFrame, implements ActionListener
{
  ...
  JButton button = new JButton("Click me!");
  button.addActionListener(this);

  public void actionPerformed(ActionEvent e){
    System.out.println("Thank You!");
  }
}
```

Why is this generally a bad idea?

Simple button click Example (III)

implementing the interface at a top level class

```
public class MyFrame extends JFrame, implements ActionListener
{
  ...
  JButton button = new JButton("Click me!");
  button.addActionListener(this);

  public void actionPerformed(ActionEvent e){
    System.out.println("Thank You!");
  }
}
```

Why is this generally a bad idea?
Just imagine more buttons!

Design considerations

- For simpler/smaller interfaces it is easy to implement their methods in our Listeners
- For larger interfaces, like MouseListener, one must implement every method! Even if we only needed one of them...

Design considerations

- Most Listener interfaces come hand-in-hand with stub classes called Adapters:

    MouseListener/MouseAdapter

    KeyListener/KeyAdapter

    MouseMotionListener/MouseMotionAdapter

- The adapter already provides stubs for each interface method. We only modify the ones we need

Mouse move example

```
...
MouseMotionListener listener = new MouseMotionAdapter() {
  public void mouseMoved(MouseEvent e) {
    System.out.println(“mouse moved: “ + e);
  }
};

panel.addMouseMotionListener(listener);
...
```

Using an Adapter

```
...
MouseMotionListener listener = new MouseMotionListener() {
  public void mouseMoved(MouseEvent e) {
    System.out.println(“mouse moved: “ + e);
  }
  public void mouseDragged(MouseEvent e) {
  }
};

panel.addMouseMotionListener(listener);
...
```

Using a Listener

Accessing event data

Each Listener type has an associated event type.

ex: MouseListener/MouseEvent

Rule of thumb

xxxxListener - Listener interface
xxxxAdapter - stubbed class
xxxxEvent - event type
Component.addxxxxListener()

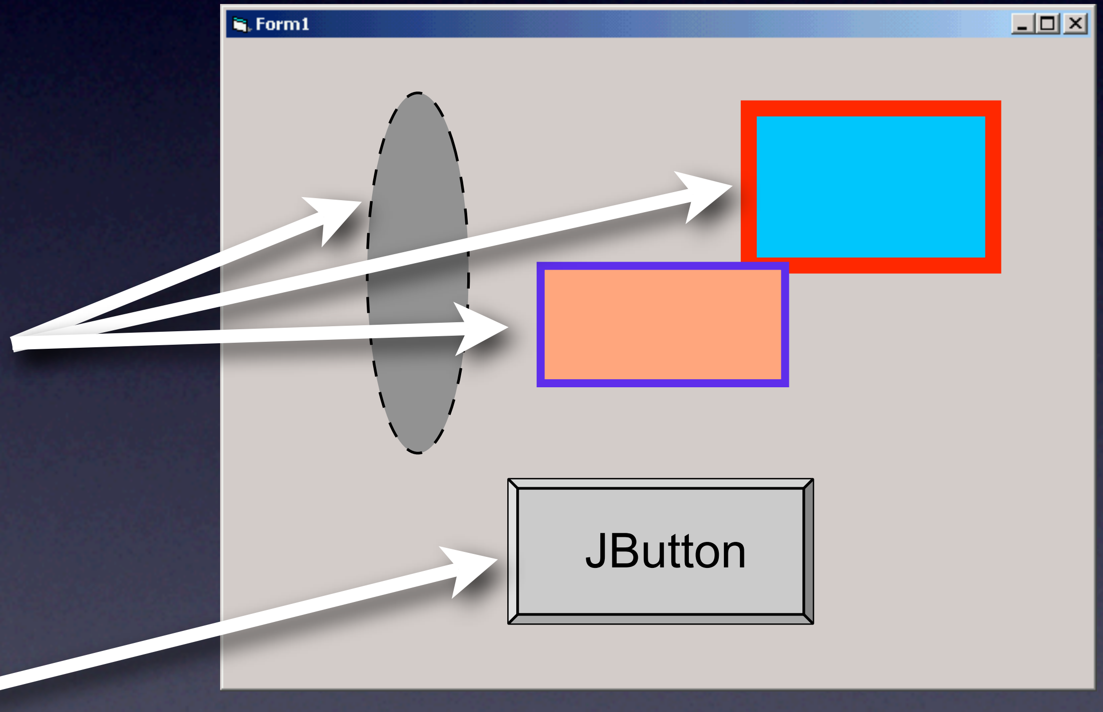
Find the mouse!

```
...
MouseMotionListener listener = new MouseMotionAdapter() {
  public void mouseMoved(MouseEvent e) {
    System.out.print(“mouse moved to “);
    System.out.println(“x=“ + e.getX());
    System.out.println(“y=“ + e.getY());
  }
};
```

mouse position

Drawing and Painting

- A window is like a painter's canvas
- Applications are responsible for painting its windows contents
- GUI componentes already know how to paint themselves

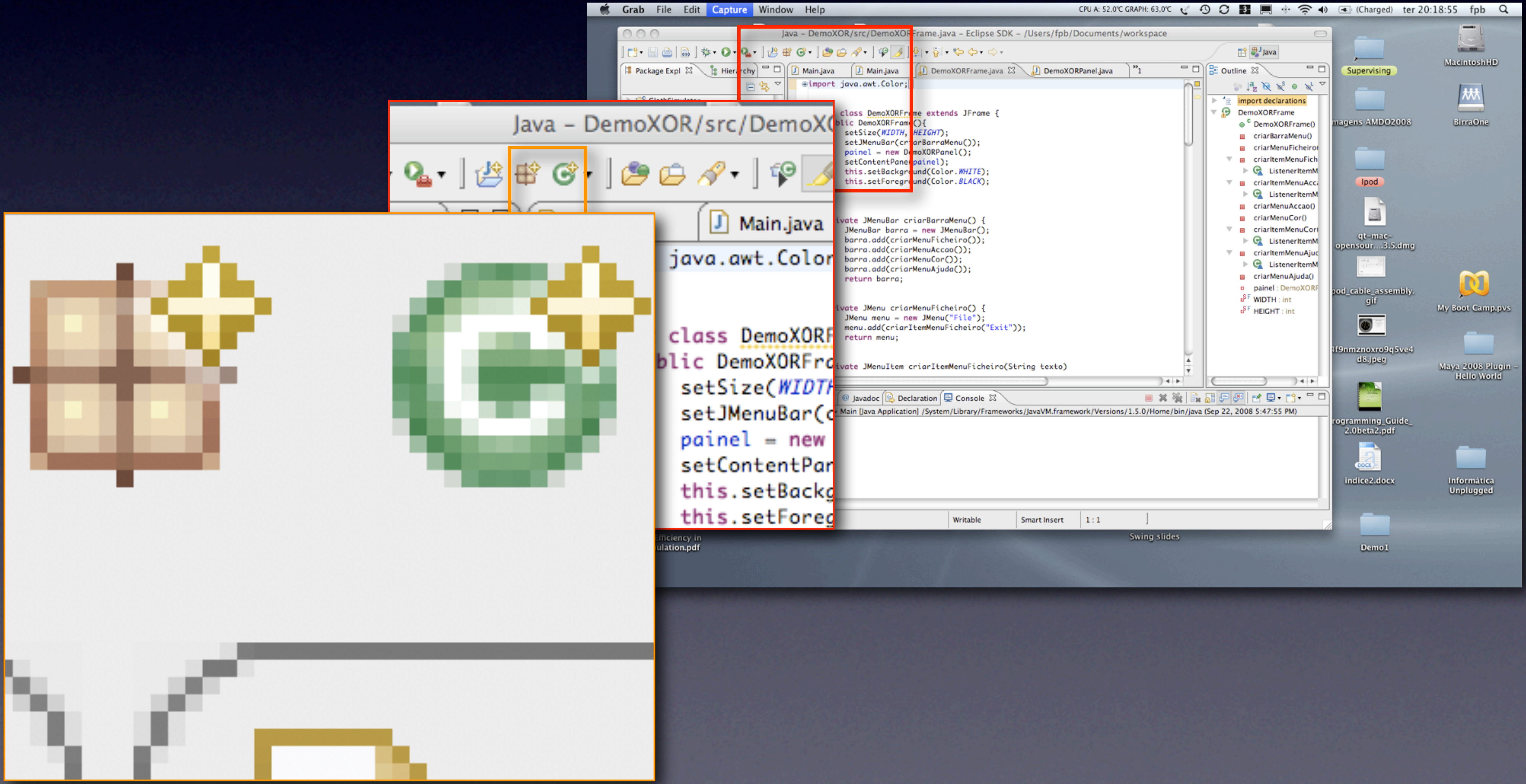


Drawing and Painting

How to Paint?

Painting: Basics

A window is a rectangular area of pixels

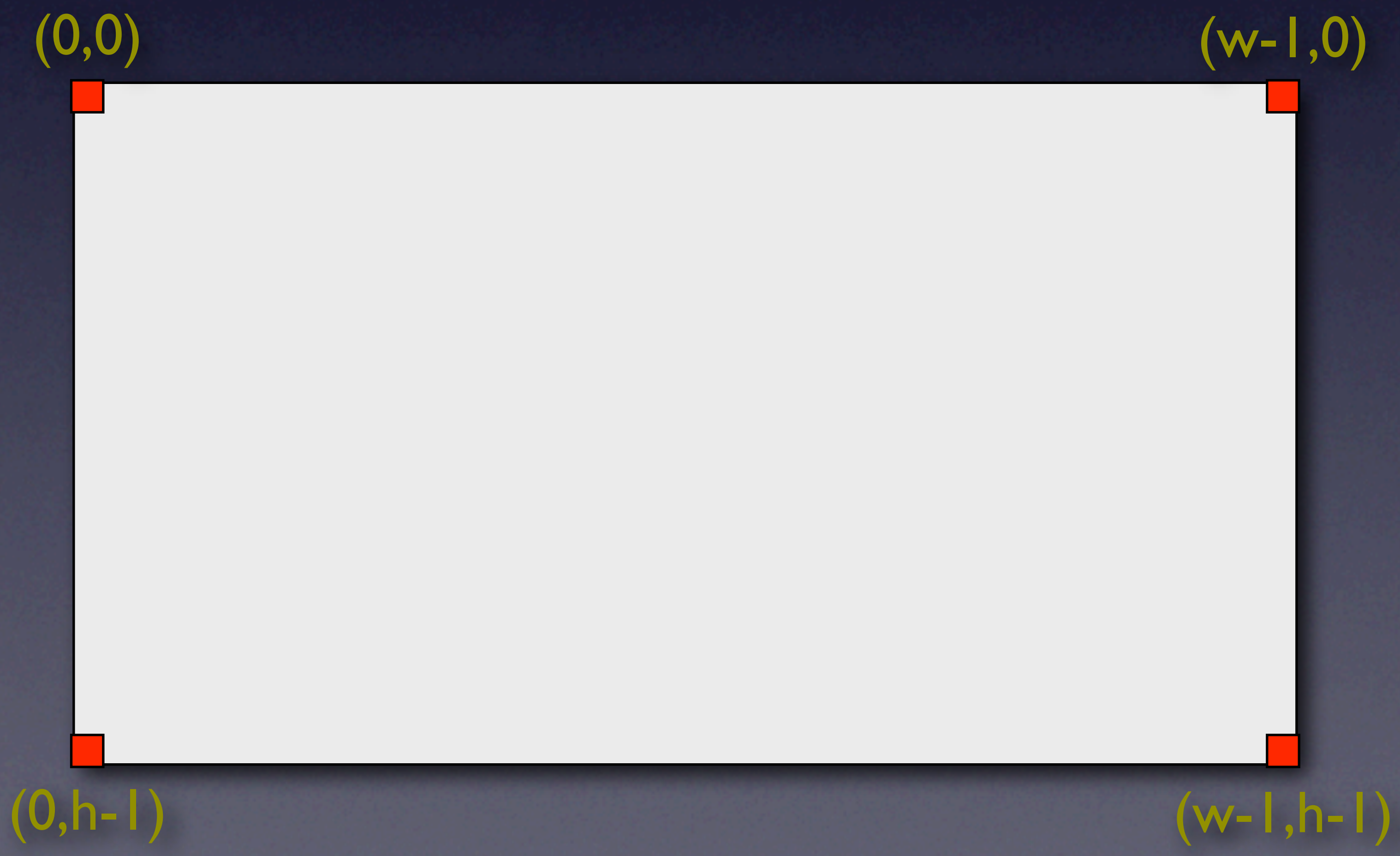


```
import java.awt.Color;

class DemoXORFrame extends JFrame {
blic DemoXORFrame(){
  setSize(WIDTH, HEIGHT);
  setJMenuBar(criarBarraMenu());
  painel = new DemoXORPanel();
  setContentPane(painel);
  this.setBackground(Color.WHITE);
  this.setForeground(Color.BLACK);

ivate JMenuBar criarBarraMenu() {
  JMenuBar barra = new JMenuBar();
  barra.add(criarMenuFicheiro());
  barra.add(criarMenuAccao());
  barra.add(criarMenuCor());
  barra.add(criarMenuAjuda());
  return barra;

ivate JMenu criarMenuFicheiro() {
  JMenu menu = new JMenu("File");
  menu.add(criarItemMenuFicheiro("Exit"));
  return menu;

ivate JMenuItem criarItemMenuFicheiro(String texto)
```

```
Main [Java Application] /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home/bin/java (Sep 22, 2008 5:47:55 PM)
```

Painting: Coordinates

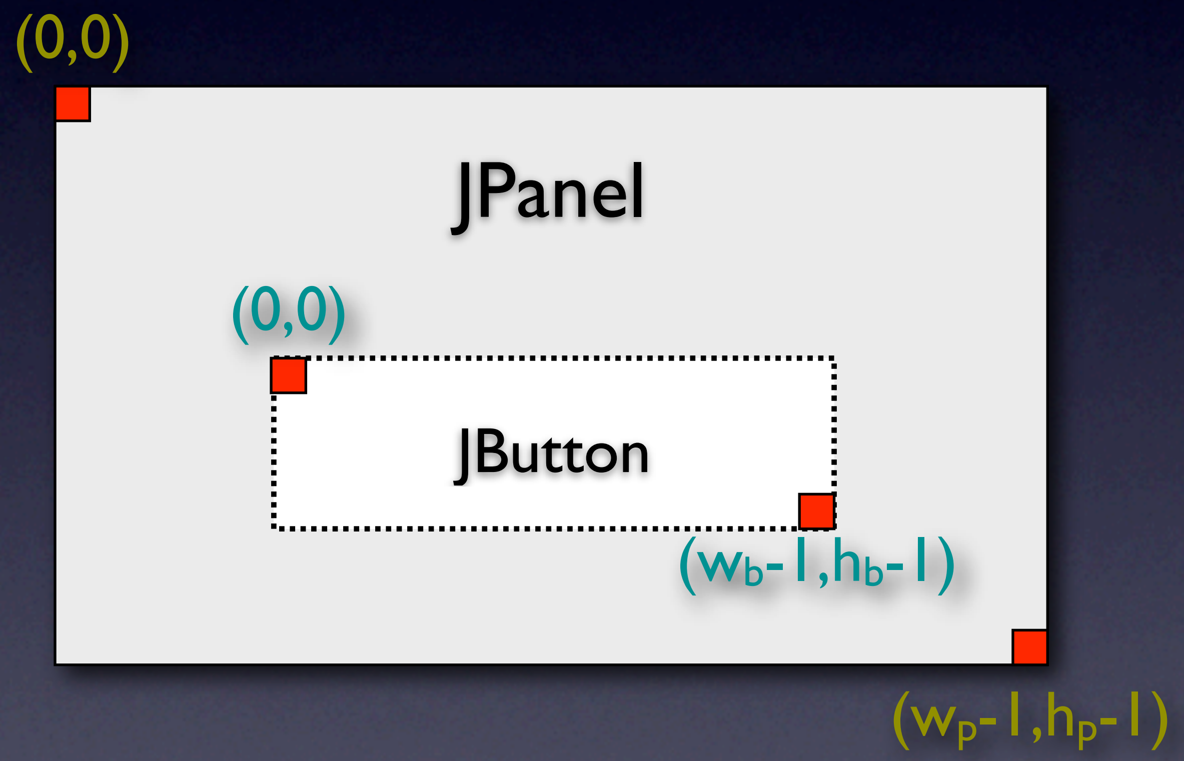
Pixels inside a component are referenced by their coordinates



Painting: Coordinates

Each component has:

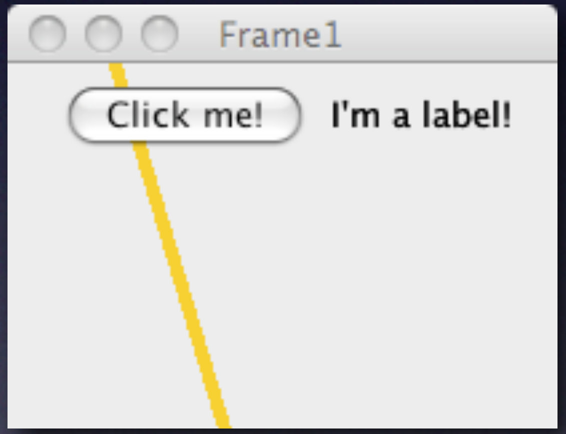
- its own sub-window (a rectangular area within parent component)
- its own coordinate system



Painting: Clipping

Due to clipping, each component:

- can’t paint outside its subwindow
- can’t paint over child components



Painting: Where in the code?

- Althouh we can paint inside all component types, the most suitable is probably a JPanel.
- Painting is handled by the method:

```
paintComponent(Graphics g)
{
    ...
}
```

Painting: Contexts

The parameter g is an object reference that is used for:

- interfacing with the device and invoking graphics operations
- maintaining the current state information (context), such as color, font, line style, etc.

```
paintComponent(Graphics g)
{
    ...
}
```

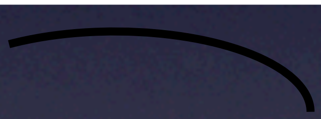
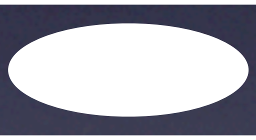
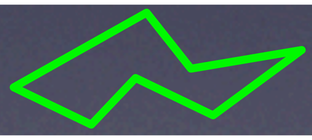
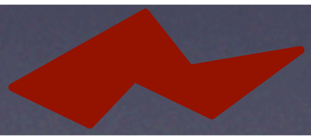
Painting: How To

```
import java.awt.Graphics;
import java.awt.Graphics2D;

paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    ...
    // use either g or g2 methods to draw
    g2.drawLine(x1, y1, x2, y2);
}
```

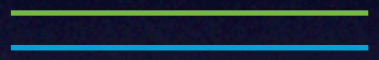
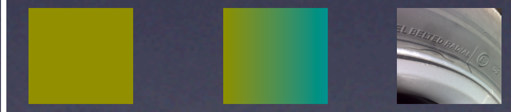
Offers more limitied drawing

Added in Java2 to provide advanced funcionality

Painting: Primitives

Type	Draw	Fill
Point	●	
Line		
PolyLine		
Arc		
Oval		
Rectangle/RoundRectangle		
Polygon		
Image		
Text	SAMPLE	

Painting: Attributes

Attributes	Sample
Color	
Font	aaaaa
Stroke (line width, dash, end caps, join, etc.)	
Paint (color, gradient, texture)	
Composite	
Transformations (translate, rotate, scale, etc.)	

Painting: Color

- Each color is a unique combination of three primary colors: red, green and blue
- Each color component lies in the range 0..255

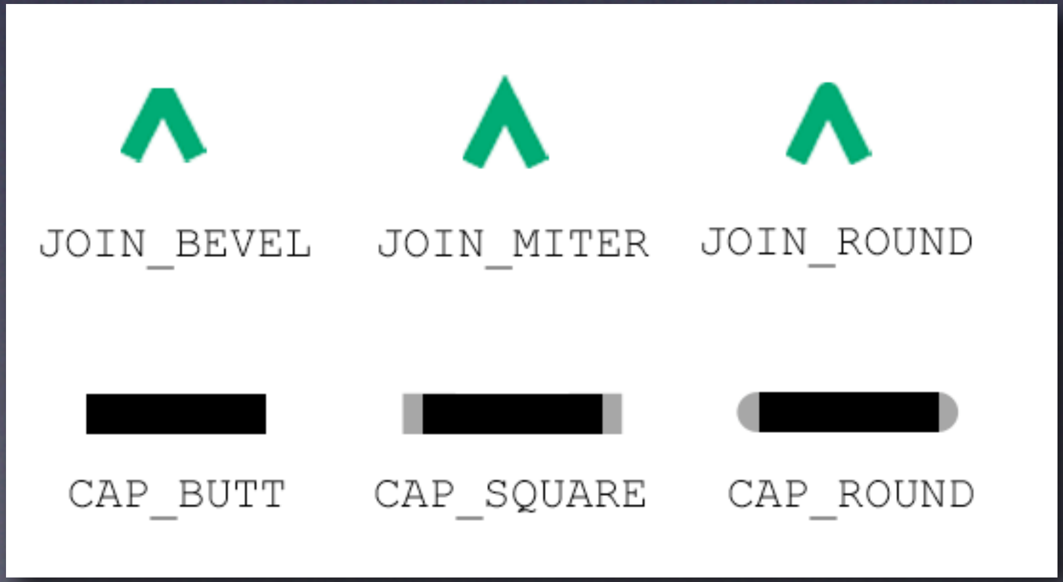
```
new Color(100, 20, 180);
```

Painting: Stroke

The current stroke determines how the outline of a specific shape or text is drawn

```
...
Stroke stroke = new BasicStroke(5.0f ,                 // Width of stroke
                           BasicStroke.CAP_ROUND,  // End cap style
                           BasicStroke.JOIN_MITER, // Join style
                           15.0f,                  // Miter limit
                           new float[] {10.0,10.0} // Dash pattern
                           5.0);
g2.setStroke(stroke);
...
```

Join and End cap styles:



Drawing and Painting

When to Paint?

Painting: Repainting

- All windows draw on the same surface (screen or painter’s canvas)
- Windows don’t remember what’s under them
- Drawing is triggered upon request, when needed: Repainting

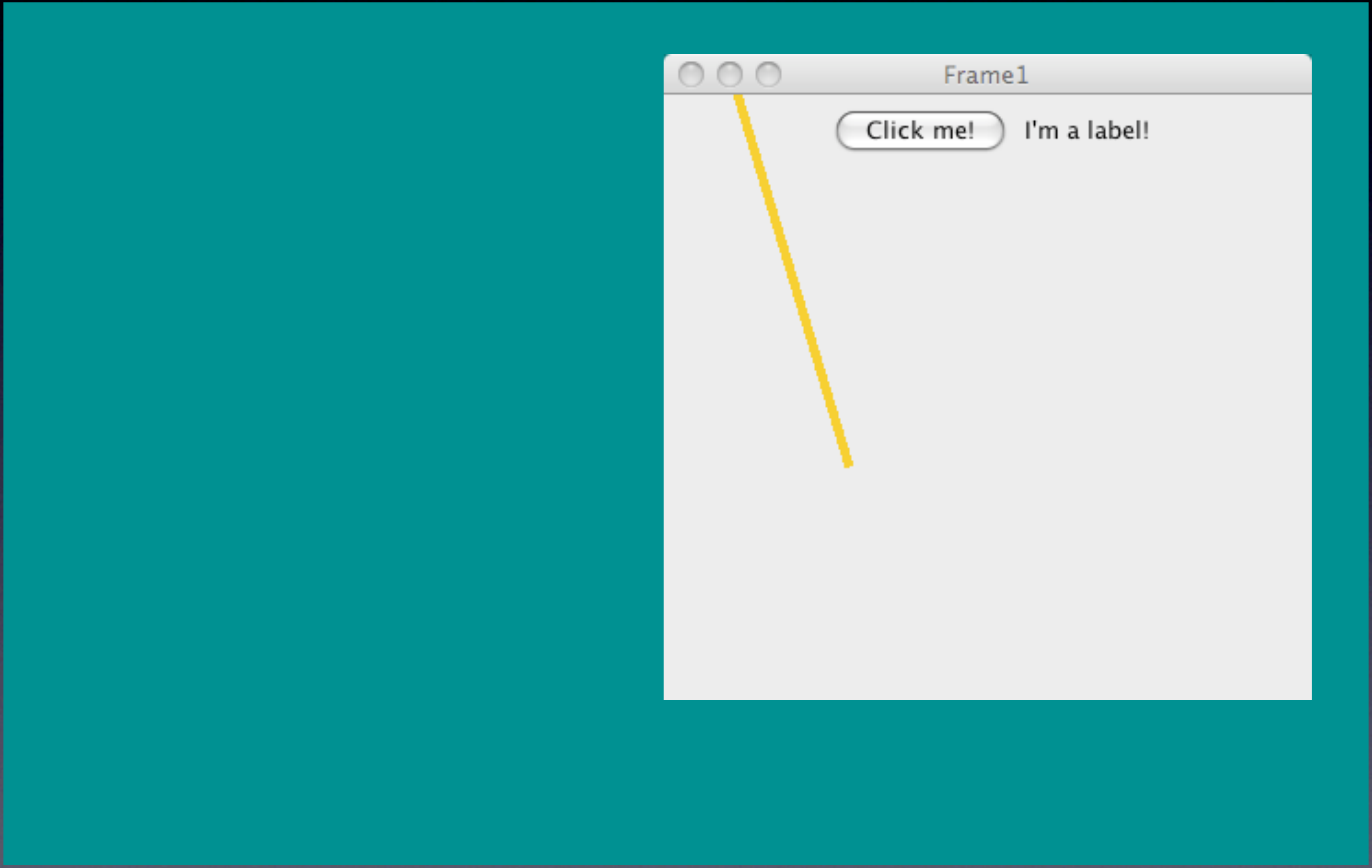
Painting: Repainting

Examples of when (re)painting is needed:

- A window becomes visible for the first time or is “brought to front”
- A window is restored after being minimized
- A window is partially exposed due to other windows on top of it closing, being dragged, etc.

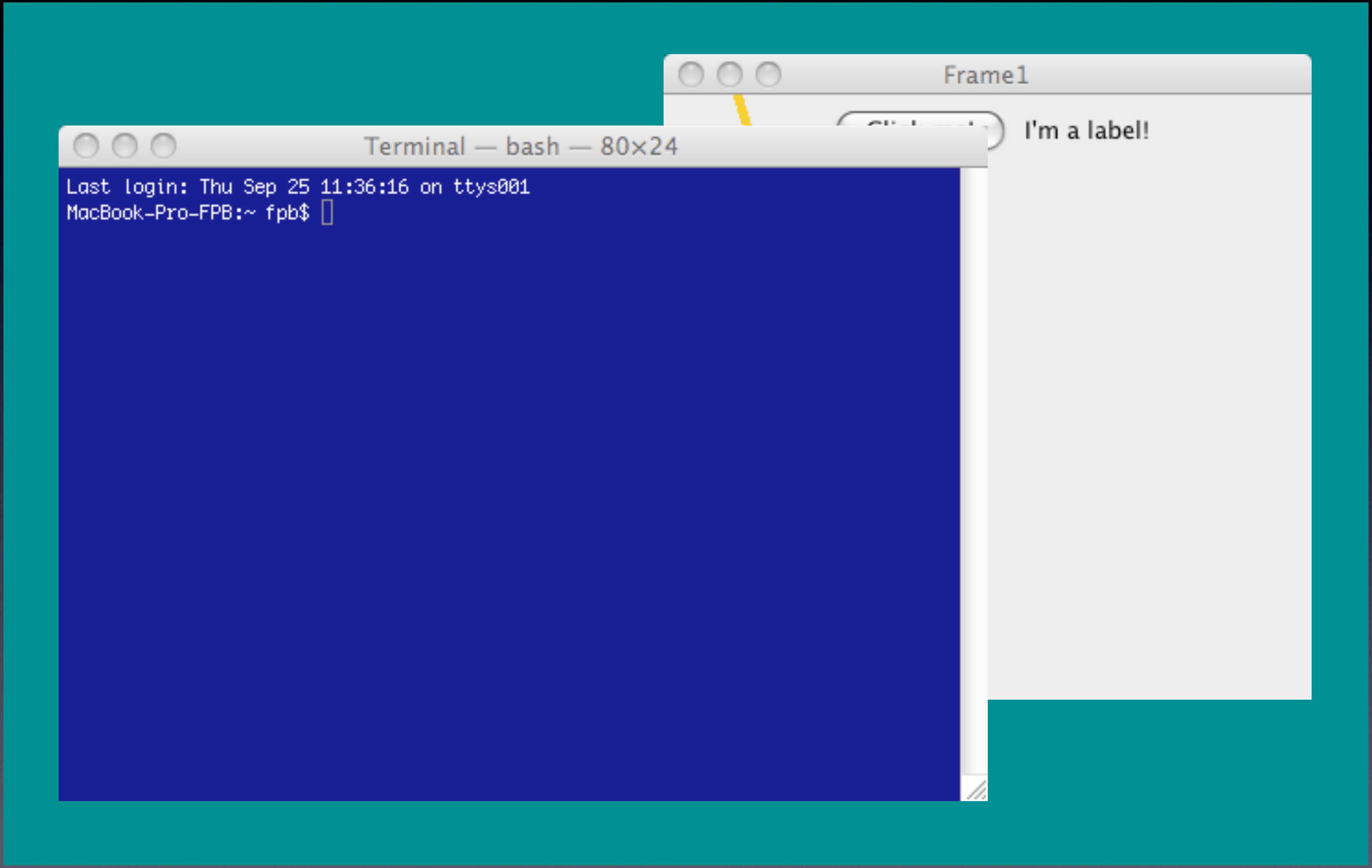
Painting

Frame1 is visible



Painting

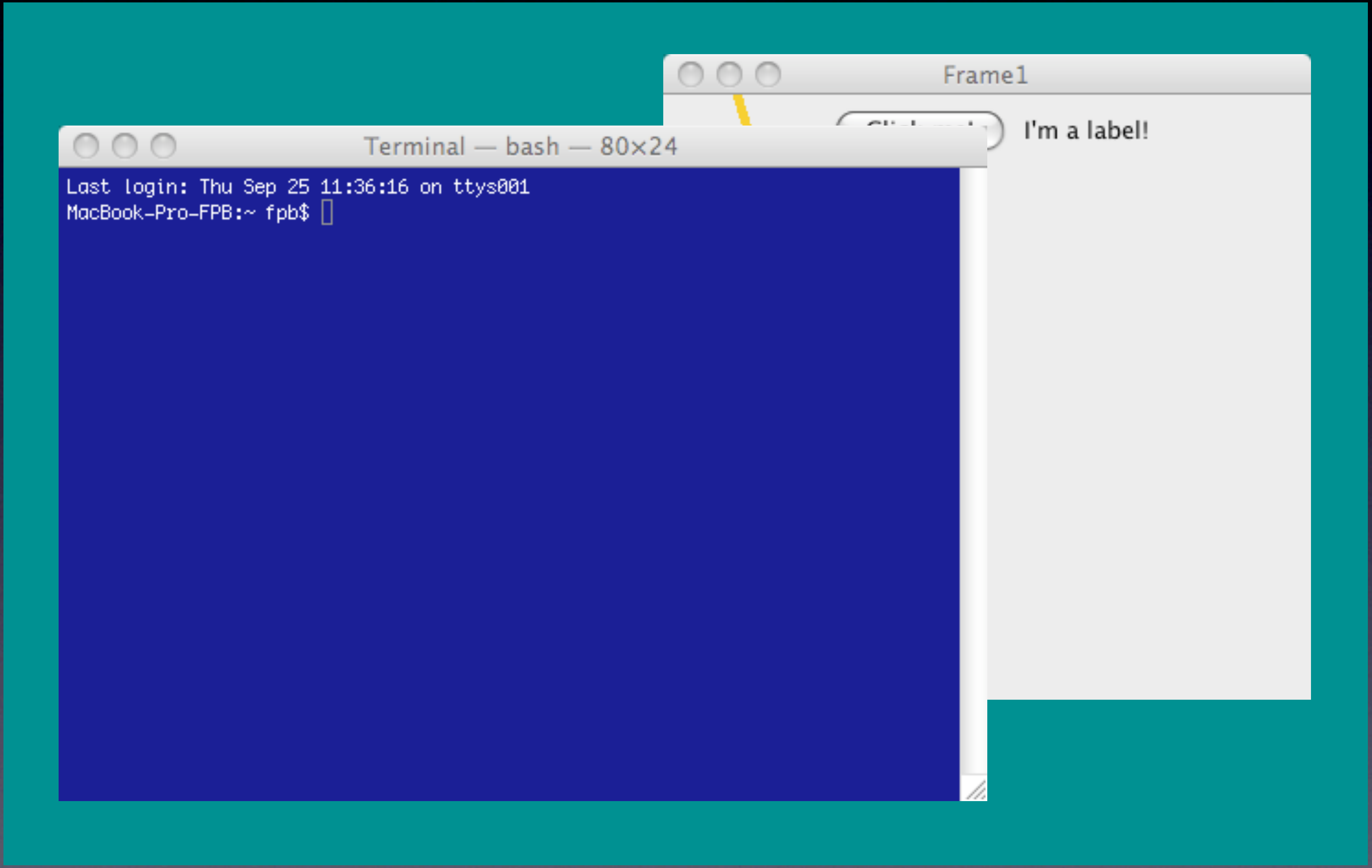
Open Terminal



```
Last login: Thu Sep 25 11:36:16 on ttys001
MacBook-Pro-FPB:~ fpb$ 
```

Painting

Close Terminal

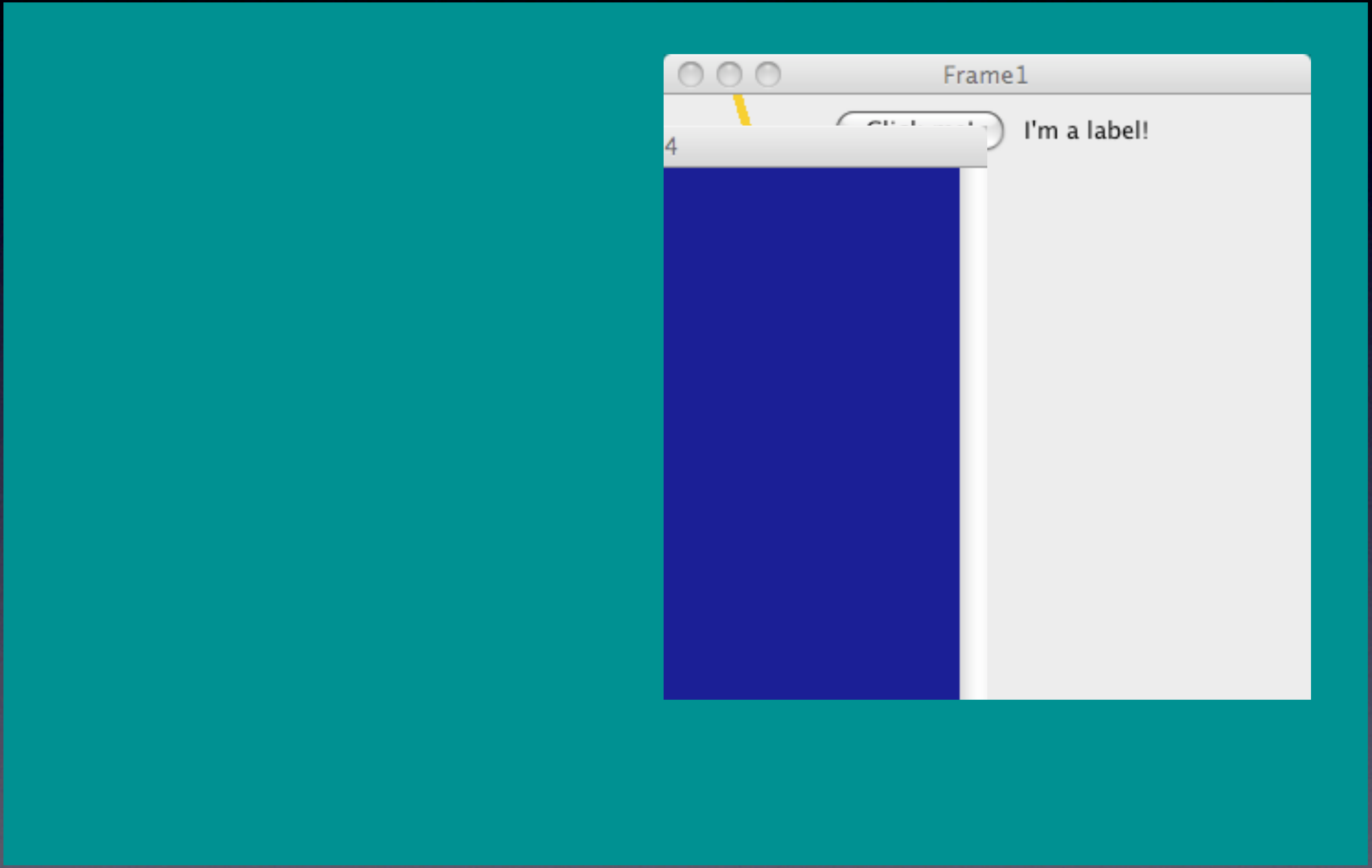


```
Last login: Thu Sep 25 11:36:16 on ttys001
MacBook-Pro-FPB:~ fpb$ 
```

Repaint events are sent to Desktop and Frame1

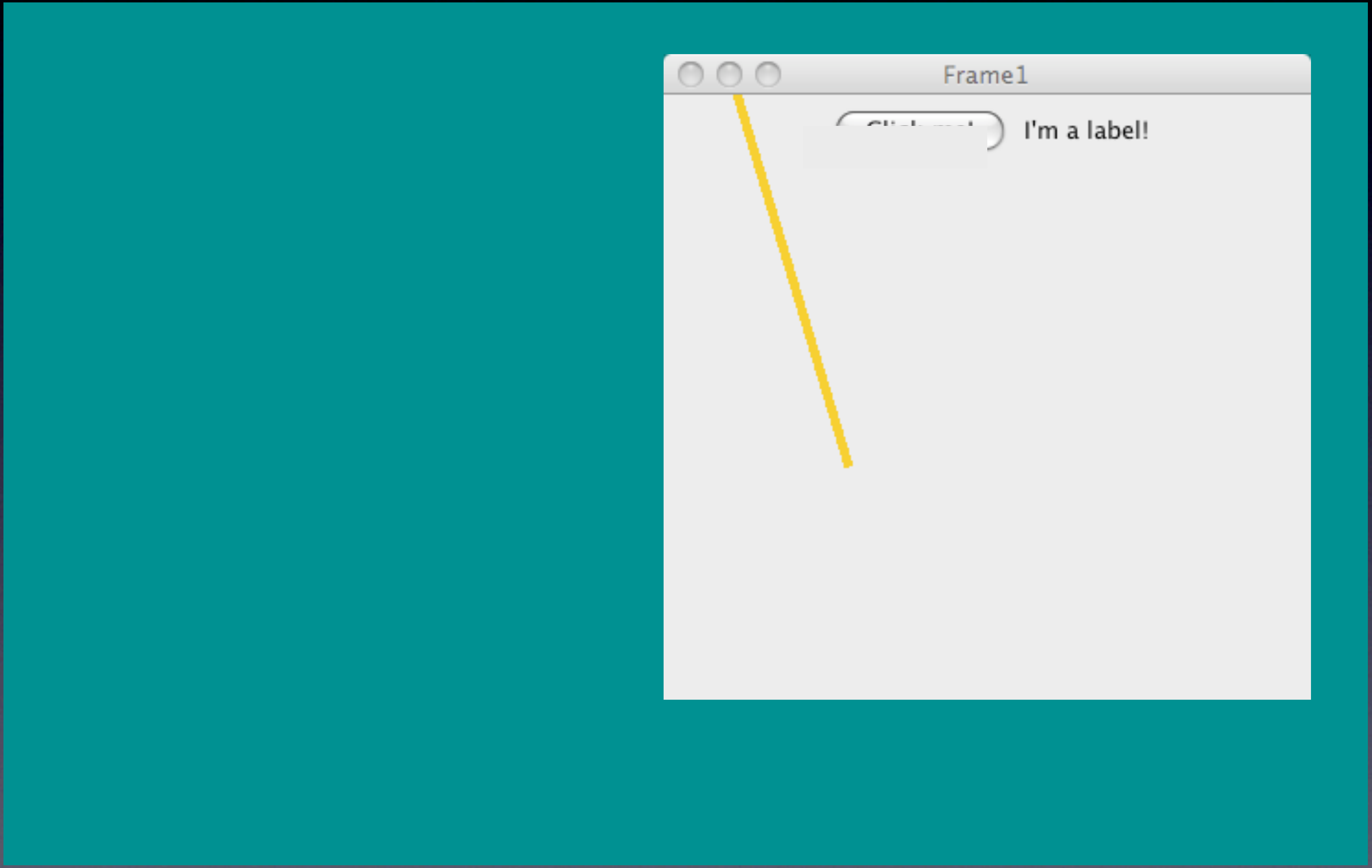
Painting

Desktop gets repainted



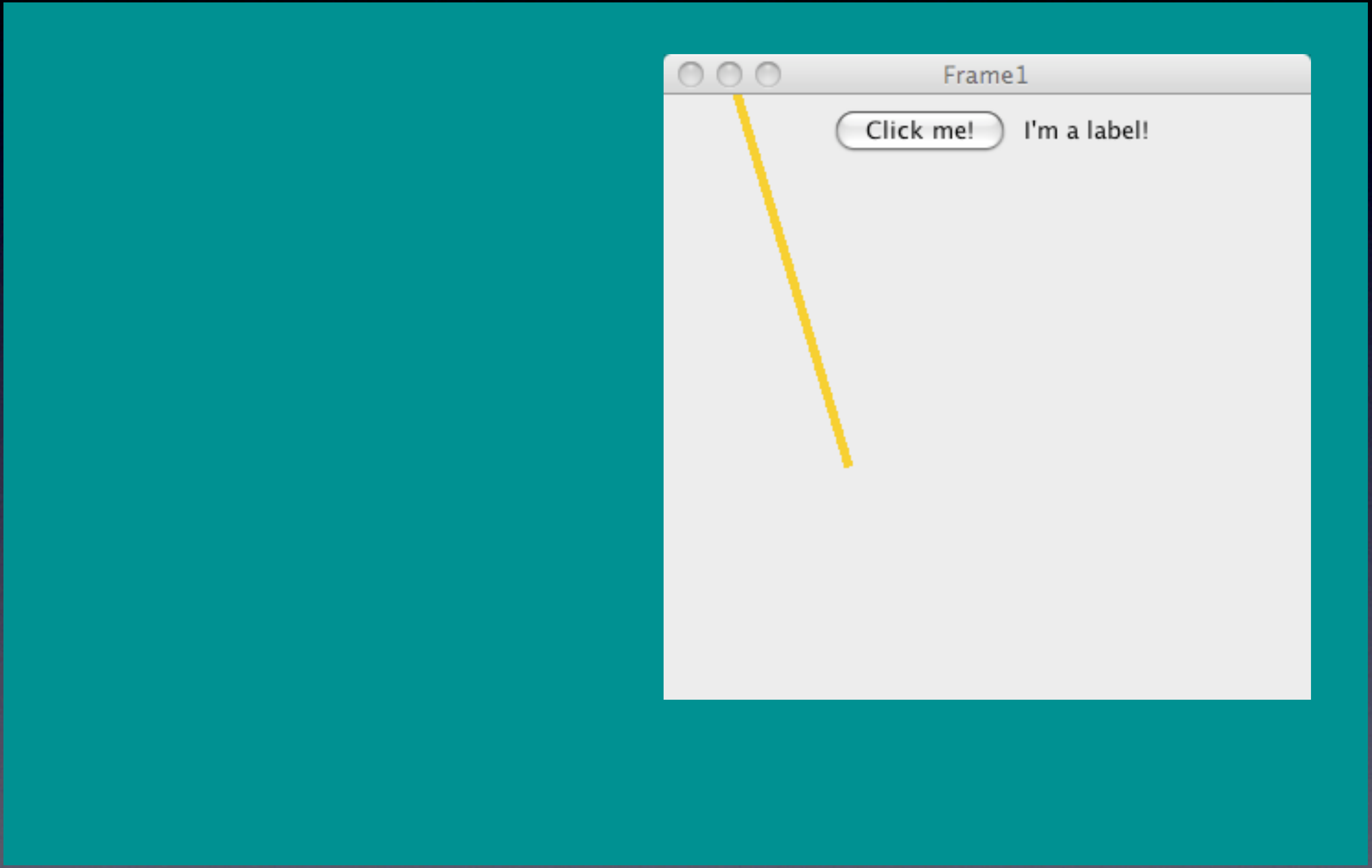
Painting

Panel gets repainted



Painting

Panel forwards repaint to button



Painting: Repainting

- Java Swing components catch repaint event and call their paintComponent() method
- Default paintComponent() implementation paints the component:

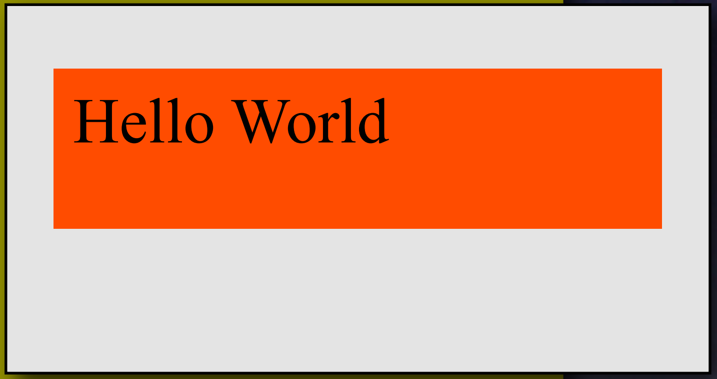
  e.g. panel erases background, button draws its shape and label, etc.

Painting: Repainting Recipe for our classes

- Subclass component (typically JPanel)
- Override paintComponent()
- when needed, invoke repaint() to get repaint events instead of calling paintComponent() directly.

Painting: Repainting Code sample

```
public class MyPanel extends JPanel {

    public void paintComponent(Graphics g){
     super.paintComponent(g);     // erases background
     Graphics2D g2 = (Graphics2D)g;   //cast for java2

     // my graphics:
     g2.setColor(new Color(255,0,0));
     g2.fillRect(10,10,200,50);
     g2.setColor(new Color(0,0,0));
     g2.drawString("Hello World", 10, 10);
  }
}
```



Painting: Repainting Typical framework

- Store data structure of window contents
    - E.g. user drawn picture in paint program
- Repaint event:
    - Erase window (draw background color)
    - Draw window contents using data structure
- Other event that alters window contents:
    - modify the data structure
    - send repaint event