Programming in Java – Basics of Swing

Paolo Vercesi

ESTECO SpA

Agenda

Hello, World!

The rules of the game

Working with Swing components

The humble dialog
Decoupling the view from the application logic

Hello, World!

Graphical user interfaces (GUI) and OOP

Object-oriented programming is very well suited for GUI programming

GUI components or controls are natural objects: windows, buttons, labels, text fields, etc., GUI programming is naturally asynchronous and event oriented

In a GUI application, the main method is responsible to initialize and assemble the GUI and the application logic, and then to make the GUI visible

GUI libraries for Java

- *Swing* ←
  - *Abstract Widget Toolkit (AWT)*
  - *Part of Java SE*
- *JavaFX*
- *Standard Widget Toolkit (SWT)*
- *All available for Windows, Linux, and MacOS*

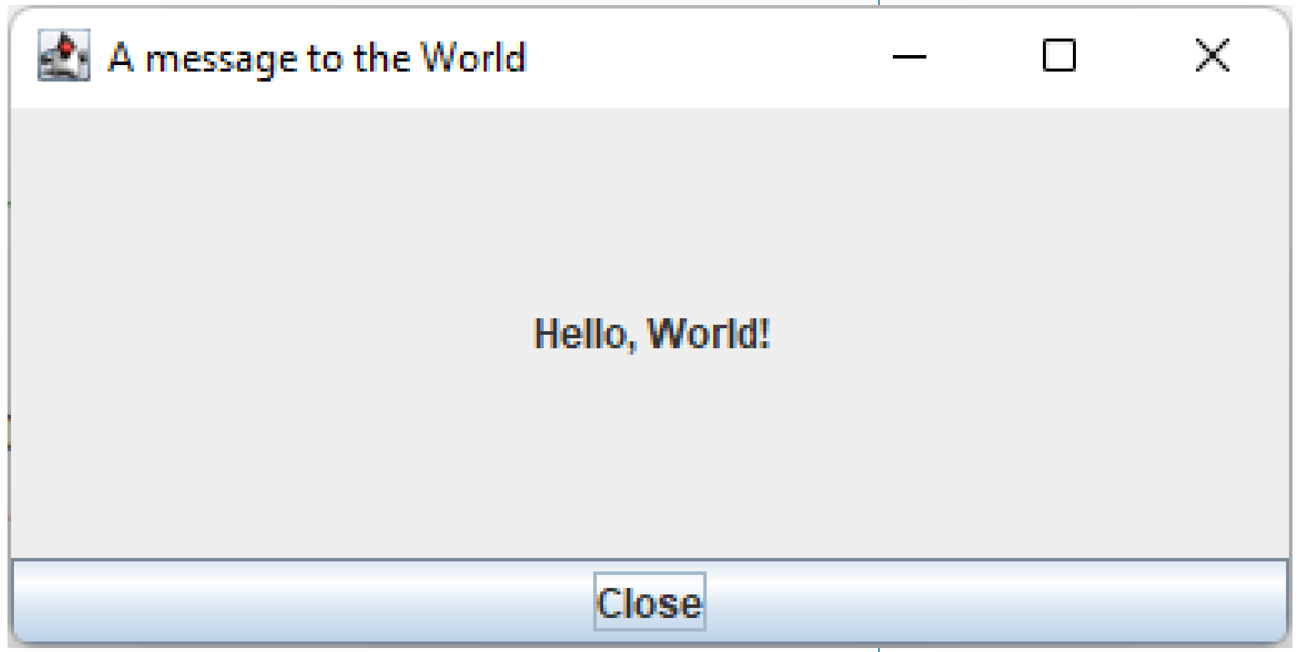
How to learn Java Swing

- *Official tutorial https://docs.oracle.com/javase/tutorial/uiswing/index.html*
  - *be aware that it is based on Java 8*
  - *released in 2014*
  - *the API hasn’t changed in the meanwhile*
  - *good to understand how the components wors*
- *Study the Java documentation*
- *Look at the source code*
- *Attend this introduction to Java Swing*
- *Do a lot of experiments*

Hello, World!

HelloWorld.java

```java
public class HelloWorld {

    public static void main(String[] args) {
        SwingUtilities.invokeLater(HelloWorld::helloWorld);
    }

    private static void helloWorld() {
        JFrame frame = new JFrame("A message to the World");
        frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

        JLabel label = new JLabel("Hello, World!");
        label.setHorizontalAlignment(SwingConstants.CENTER);
        frame.getContentPane().add(label, BorderLayout.CENTER);

        JButton closeButton = new JButton("Close");
        closeButton.addActionListener(x -> frame.dispose());
        frame.getContentPane().add(closeButton, BorderLayout.SOUTH);

        frame.setSize(400, 200);
        frame.setVisible(true);
    }
}
```



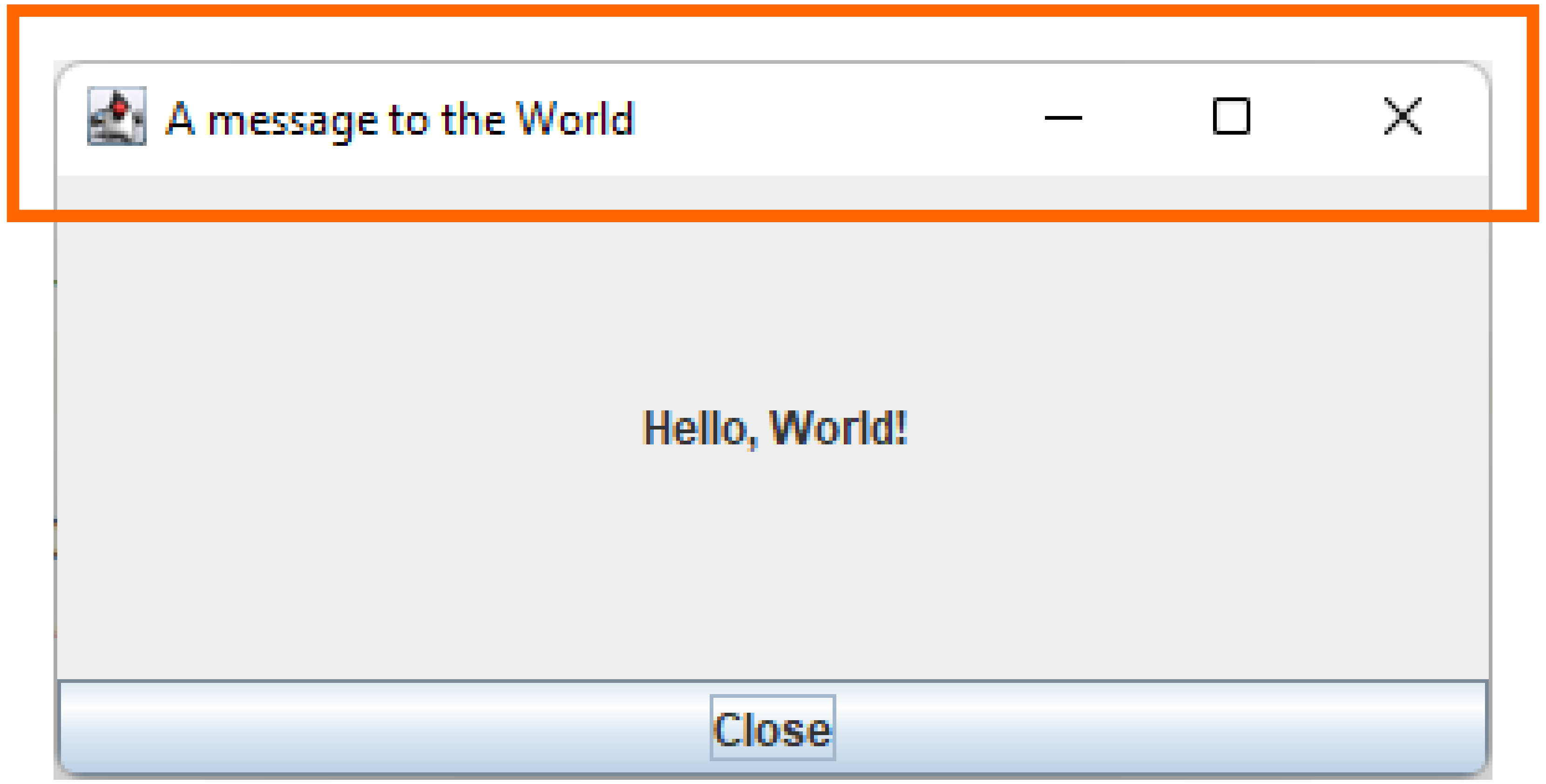
A message to the World

Hello, World!

Analysis of HelloWorld.java 1/5

HelloWorld.java

```
JFrame frame = new JFrame("A message to the World");
frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
```



A JFrame represents a window with all the decorations: icon, title, and buttons to minimize, maximize, and close

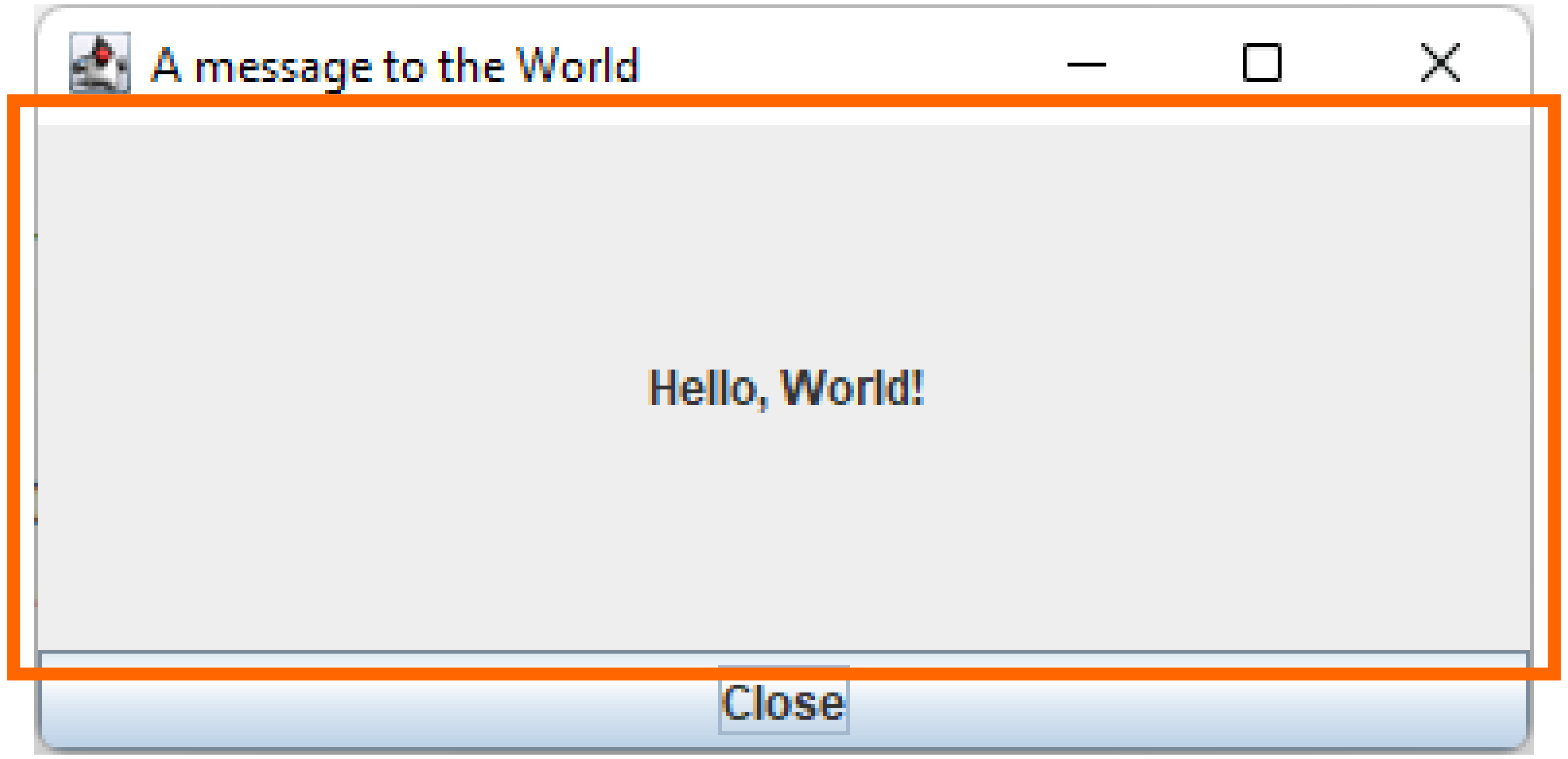
The behavior of the close button can be customized, for example to dispose the JFrame

By disposing a JFrame, we close the JFrame, if open, and we release all the resources associated to this JFrame

Analysis of HelloWorld.java 2/5

HelloWorld.java

```
JLabel label = new JLabel("Hello, World!");
label.setHorizontalAlignment(SwingConstants.CENTER);
frame.getContentPane().add(label, BorderLayout.CENTER);
```



The content pane of a JFrame uses the BoderLayout manager by default

A JLabel is a Swing component used to represents a piece of text with an icon

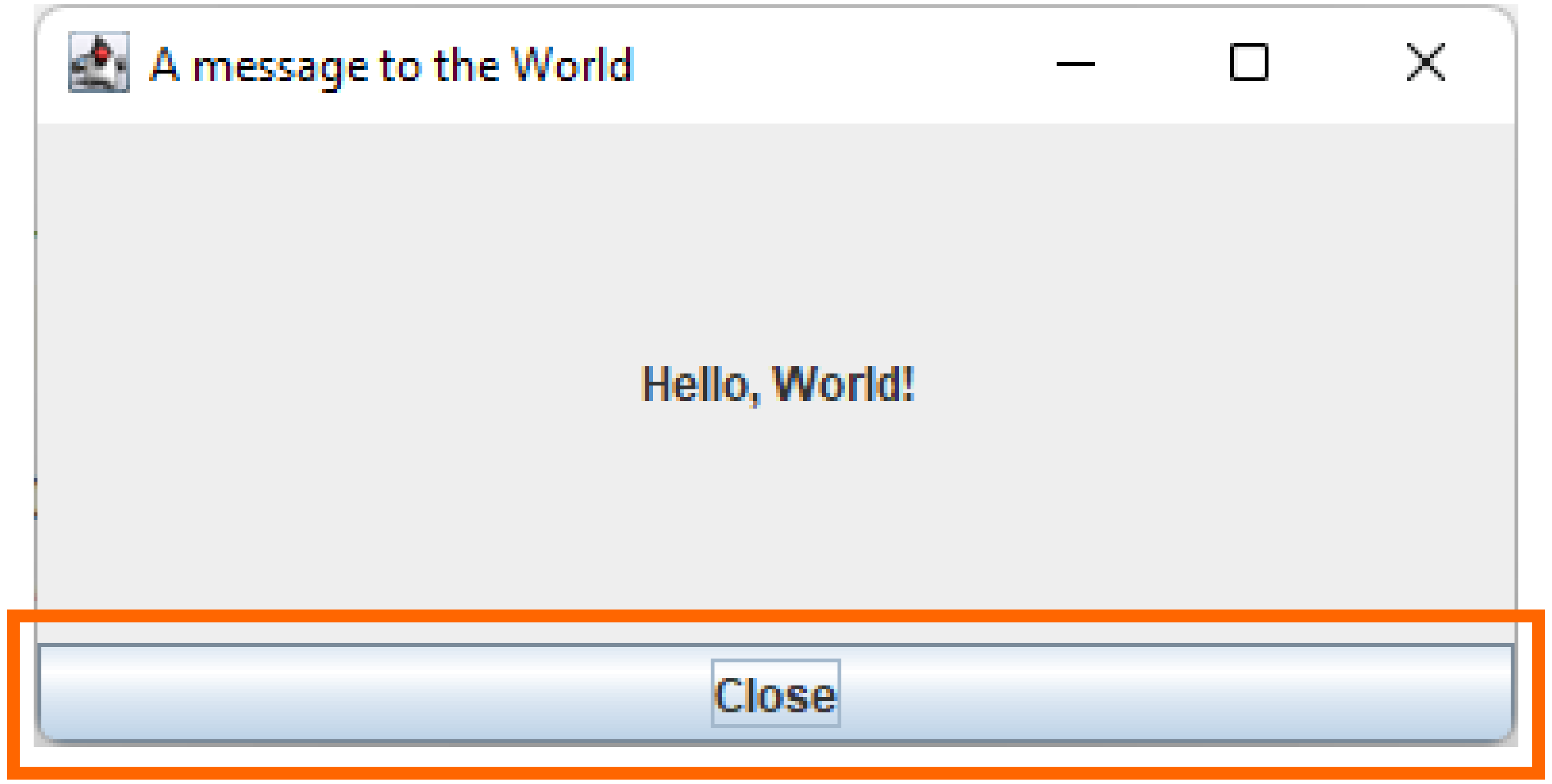
To make a Swing component visible, we must add it to a container, if there are no intermediate containers, we can add it to the content pane of the JFrame directly

A container uses a layout manager to layout the components it contains. When adding a component to a container we can specify a constraint

Analysis of HelloWorld.java 3/5

HelloWorld.java

```
JButton closeButton = new JButton("Close");
closeButton.addActionListener(x -> frame.dispose());
frame.getContentPane().add(closeButton, BorderLayout.SOUTH);
```

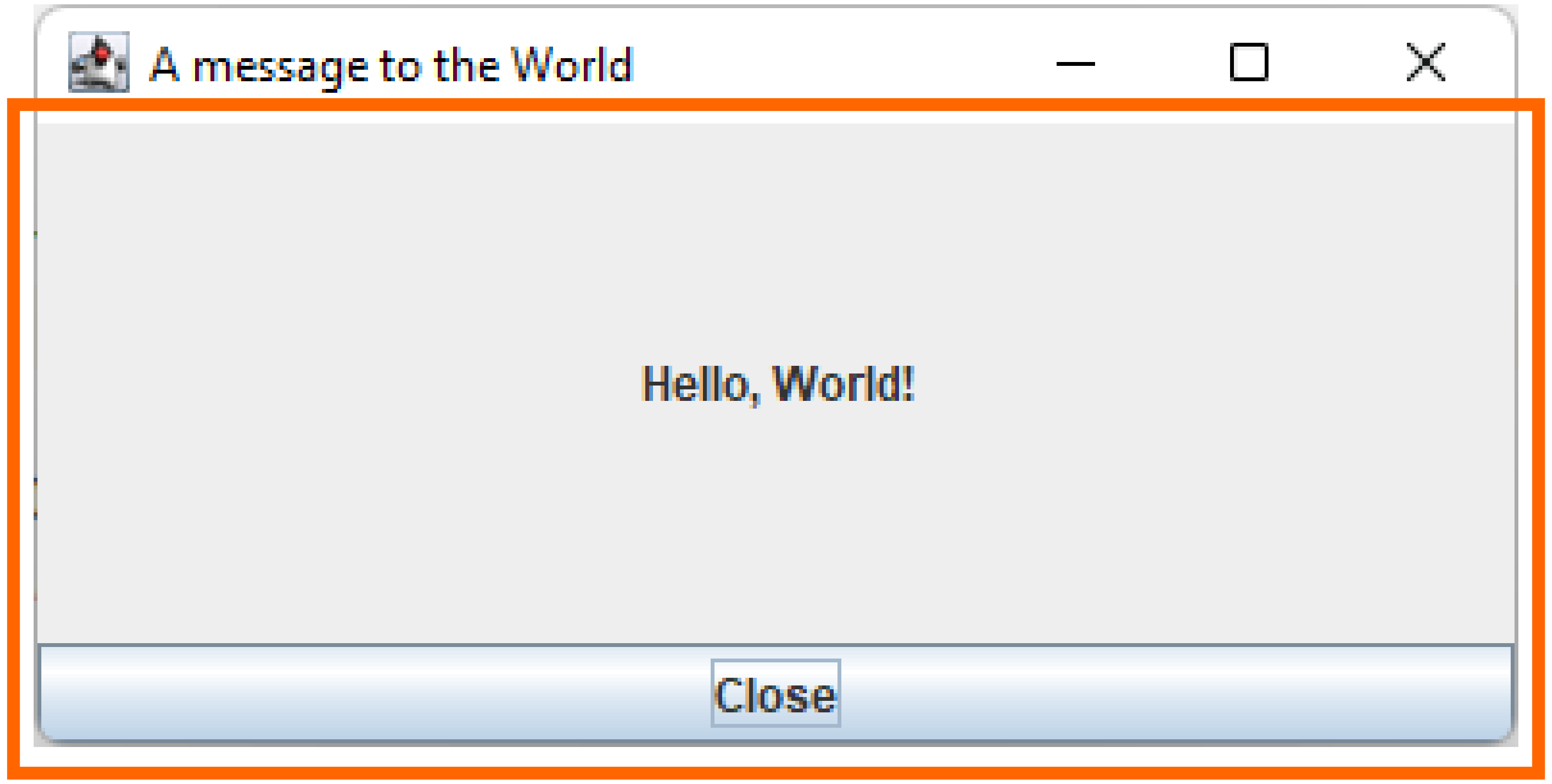


A JButton is a Swing component able to respond to user actions. For example, when the user clicks on the button, it triggers an action listener

Analysis of HelloWorld.java 4/5

HelloWorld.java

```
frame.setSize(400, 200);
frame.setVisible(true);
```



A JFrame and its content pane are shown in the screen when we make the frame visible

Analysis of HelloWorld.java 5/5

HelloWorld.java

```java
public static void main(String[] args) {
    SwingUtilities.invokeLater(HelloWorld::helloWorld);
}
```

Almost all GUI code **MUST** *run on the Event Dispatch Thread by using either invokeLater or invokeAndWait*

`static void invokeAndWait(Runnable doRun)`	Causes *doRun.run()* to be executed synchronously on the AWT event dispatching thread.
`static void invokeLater(Runnable doRun)`	Causes *doRun.run()* to be executed asynchronously on the AWT event dispatching thread.

More on this topic in the next section!

Take aways

- ❑ *Swing is a library used to develop a graphical user interface (GUI) for Java programs*
- ❑ *Swing is part of the “The Java Platform, Standard Edition (Java SE) APIs”*

The rules of the game

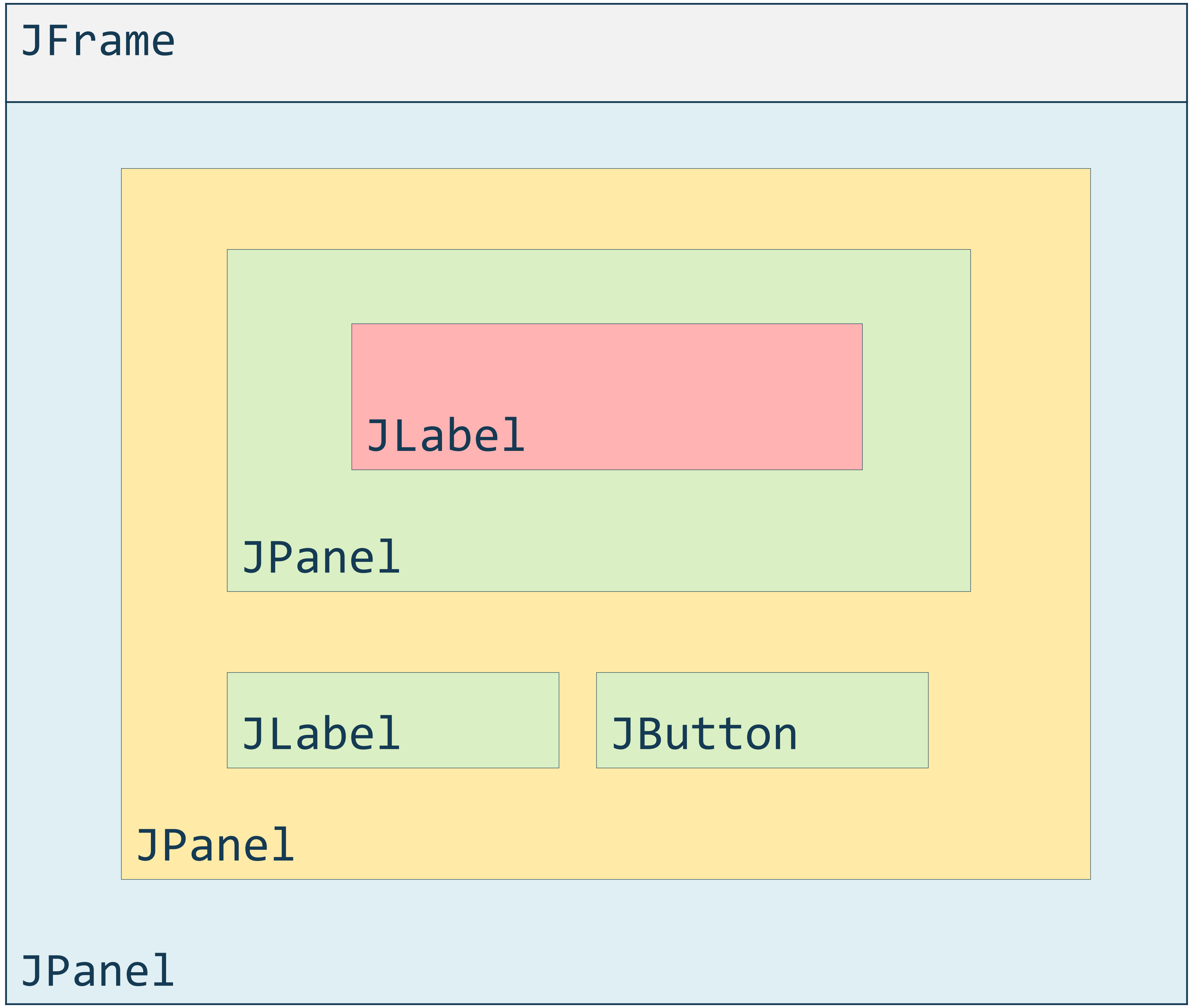
Containment hierarchy

To make a component visible, its containment hierarchy must be included into a JFrame o another window object

JPanels are containers to which usually we add components

Each component can belong to just one container

Other containers to which we add components are JToolBar, JMenu, and JPopupMenu



Swing windows

	JFrame	JDialog	JWindow
Title bar	*Yes*	*Yes*	*No*
Window buttons	*Minimize, maximize, and close*	*Close*	*None*
Border	*Yes*	*Yes*	*No*
Modal	*No*	*Yes*	*No*
Independent	*Yes*	*No*	*No*

A GUI application usually visualizes just one JFrame instance

- *When a frame is minimized, all the child dialogs and windows are minimized*
- *When a frame is disposed, all the child dialogs and windows are disposed*

Disposing windows

Windows (JFrame, JDialog, and JWindow) must be disposed after usage

```
public void dispose()
```

Releases all of the native screen resources used by this `Window`, its subcomponents, and all of its owned children. That is, the resources for these `Components` will be destroyed, any memory they consume will be returned to the OS, and they will be marked as undisplayable.

The `Window` and its subcomponents can be made displayable again by rebuilding the native resources with a subsequent call to `pack` or `show`. The states of the recreated `Window` and its subcomponents will be identical to the states of these objects at the point where the `Window` was disposed (not accounting for additional modifications between those actions).

Note: When the last displayable window within the Java virtual machine (VM) is disposed of, the VM may terminate. See AWT Threading Issues for more information.

Dispose vs hide

DisposedFrame.java

```java
public class DisposedFrame {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(DisposedFrame::disposeFrame);
    }

    private static void disposeFrame() {
        JFrame frame = new JFrame("A frame that will be disposed");
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.setSize(400, 200);
        frame.setVisible(true);
    }
}
```

This program terminates

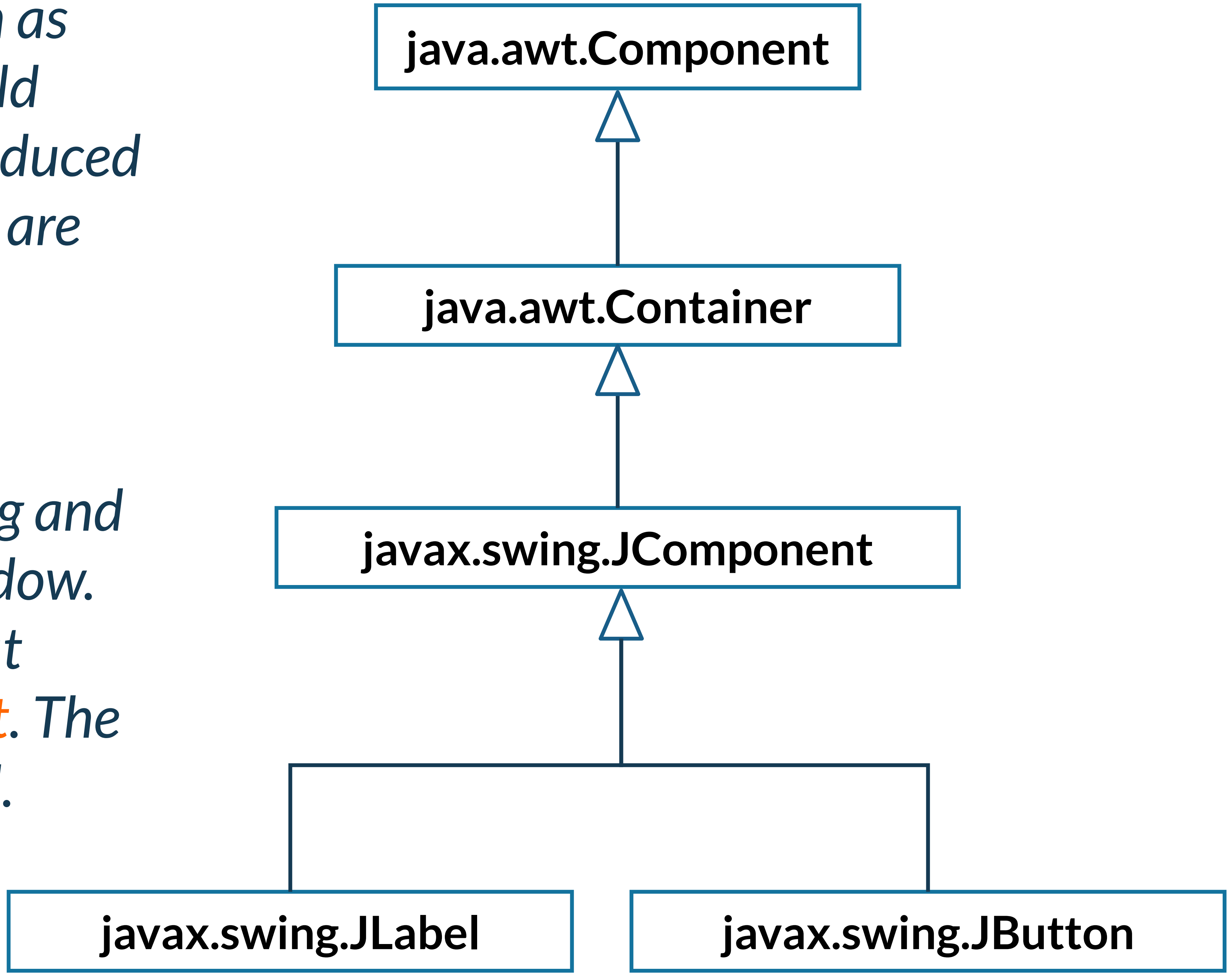
This program doesn't terminate

HiddenFrame.java

```java
public class HiddenFrame {

    public static void main(String[] args) {
        SwingUtilities.invokeLater(HiddenFrame::hideFrame);
    }

    private static void hideFrame() {
        JFrame frame = new JFrame("A frame that will be hidden");
        frame.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
        frame.setSize(400, 200);
        frame.setVisible(true);
    }
}
```

Swing components and AWT

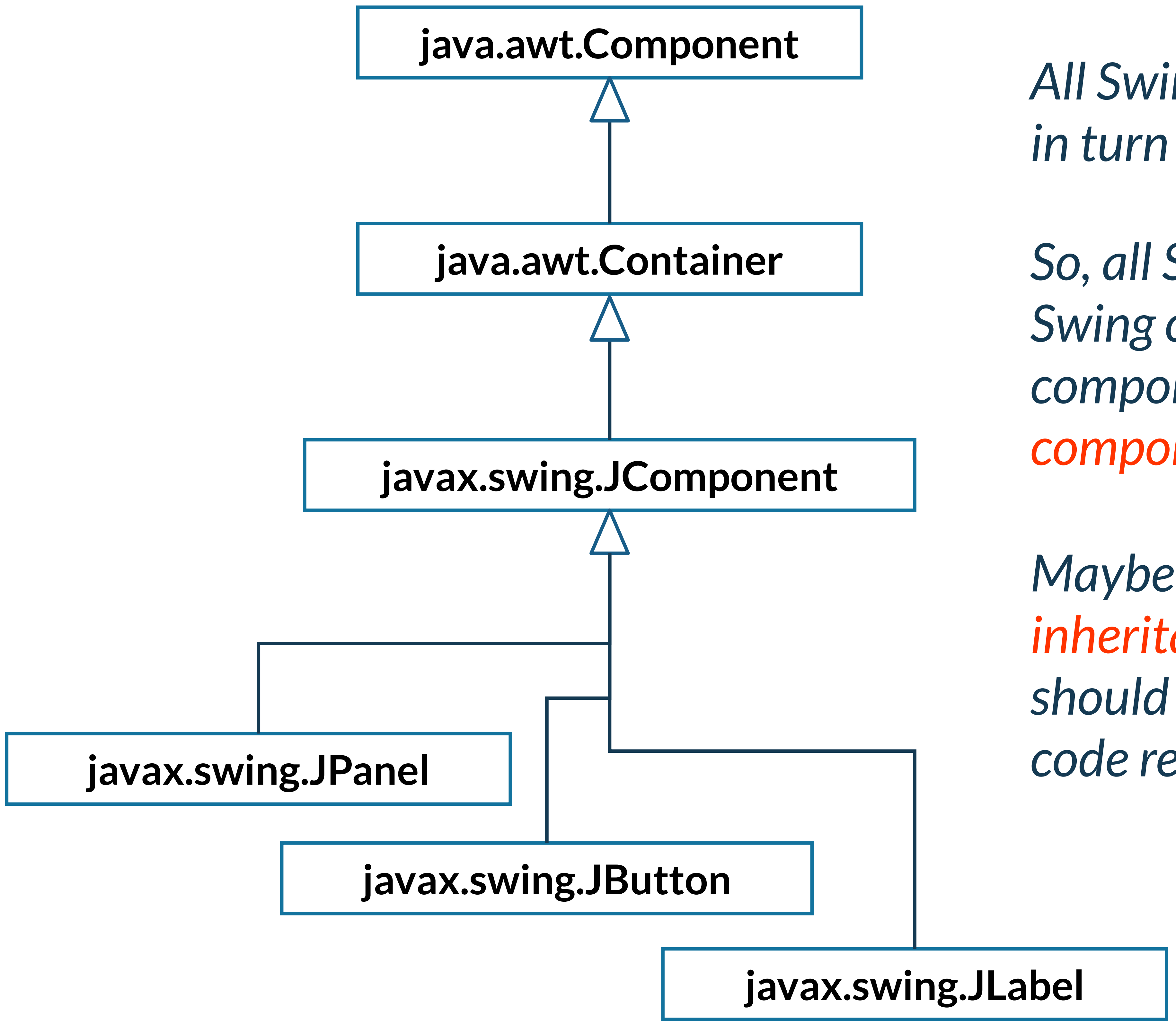
In Java Swing there are other windows classes, such as Frame, Dialog, and Window. These are part of the old AWT library available since Java 1. Swing was introduced since Java 2. Graphic classes without the 'J' in front are usually part of AWT and you should not use them.



Some Swing classes, like for example JFrame, JDialog and JWindow still inherits from Frame, Dialog, and Window. All Swing components inherit from JComponent that inherit from Container that inherit from Component. The API of Container and Component is still widely used.

Assignment: explore the API of Container, Component and JComponent.

Inheritance hierarchy



All Swing components, inherits from JComponent that in turn inherits from Container

So, all Swing components are containers but not all Swing components are meant to contain other components. E.g., is not appropriate to add a component to a JButton

Maybe this is not a very appropriate use of inheritance, but sometimes software engineers should accept trade-offs, in this case they traded code reuse with a "misuse" of inheritance

When to use inheritance

Both classes are in the same logical domain

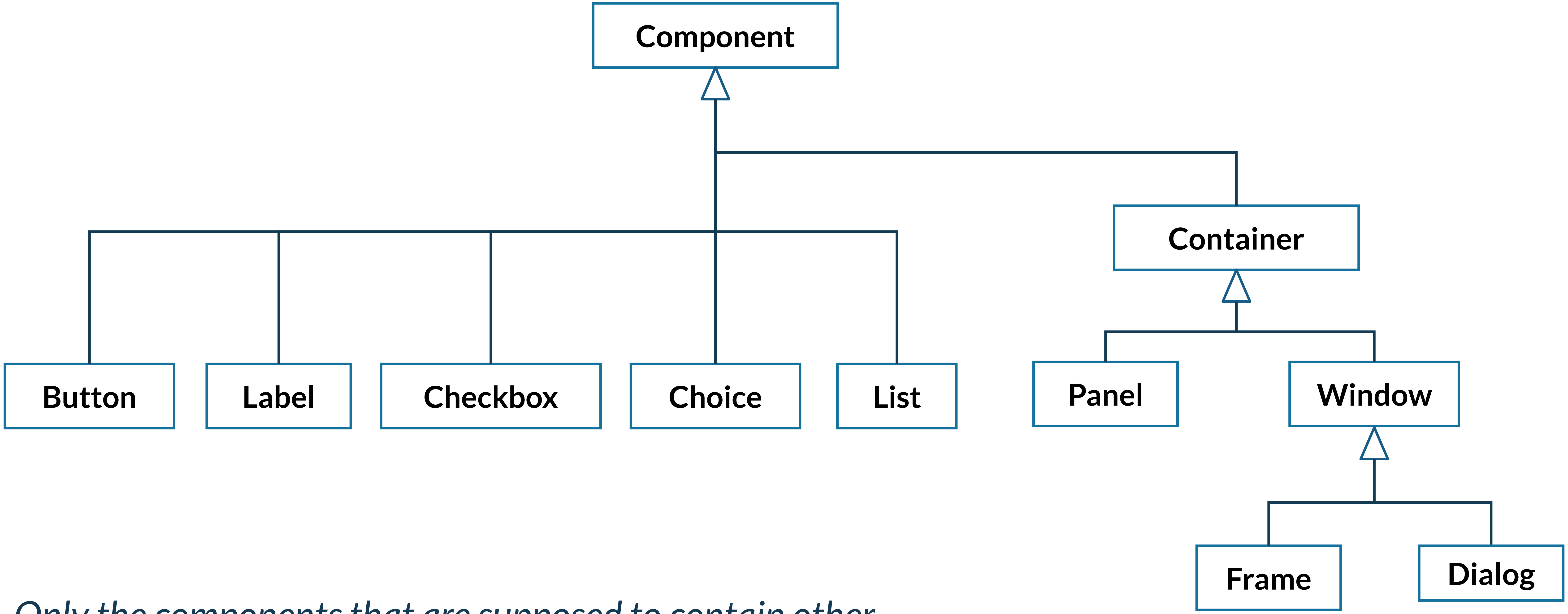
The subclass is a proper subtype of the superclass

The implementation of the superclass is necessary or appropriate for the subclass

The enhancements made by the subclass are primarily additive

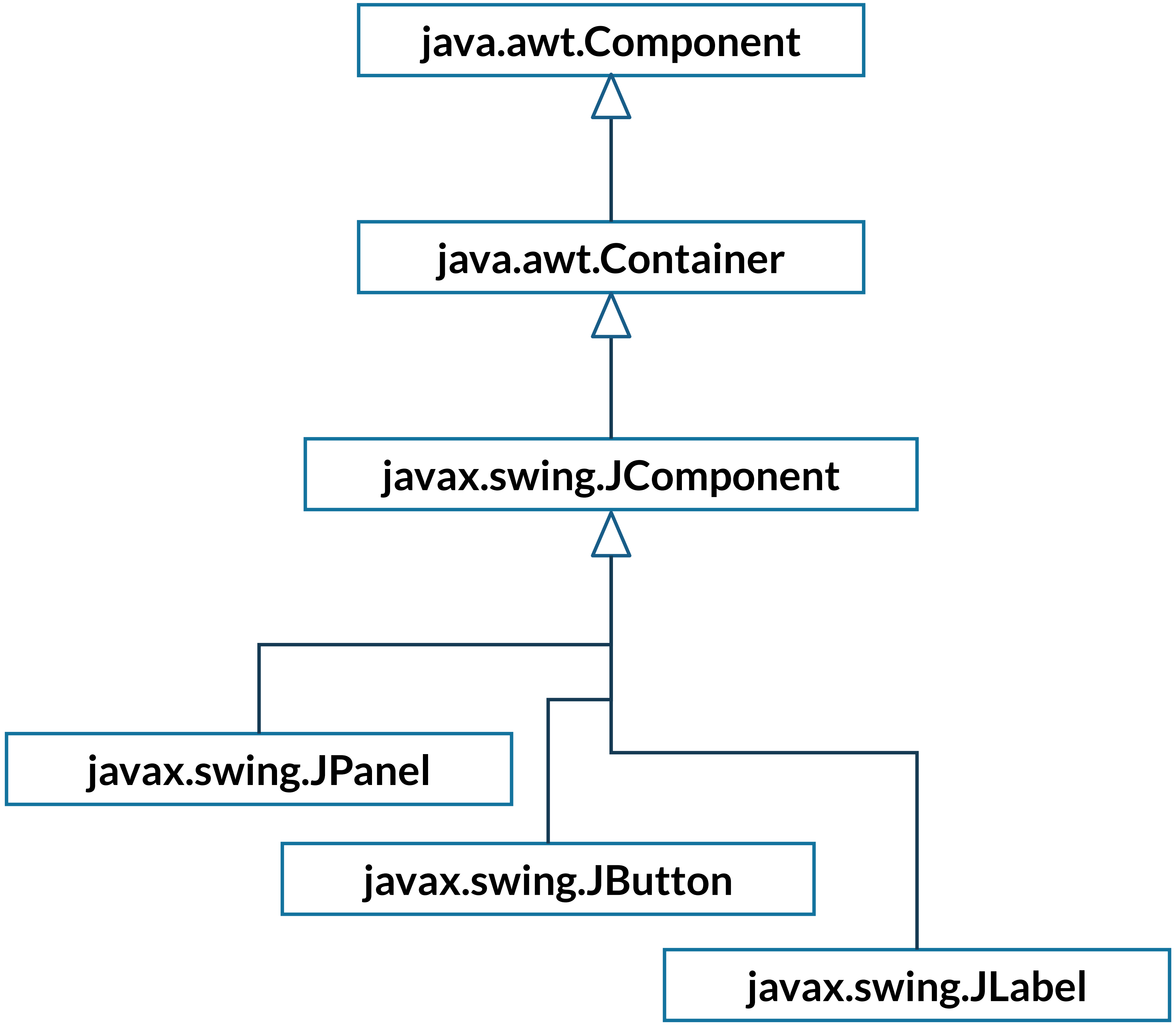
https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

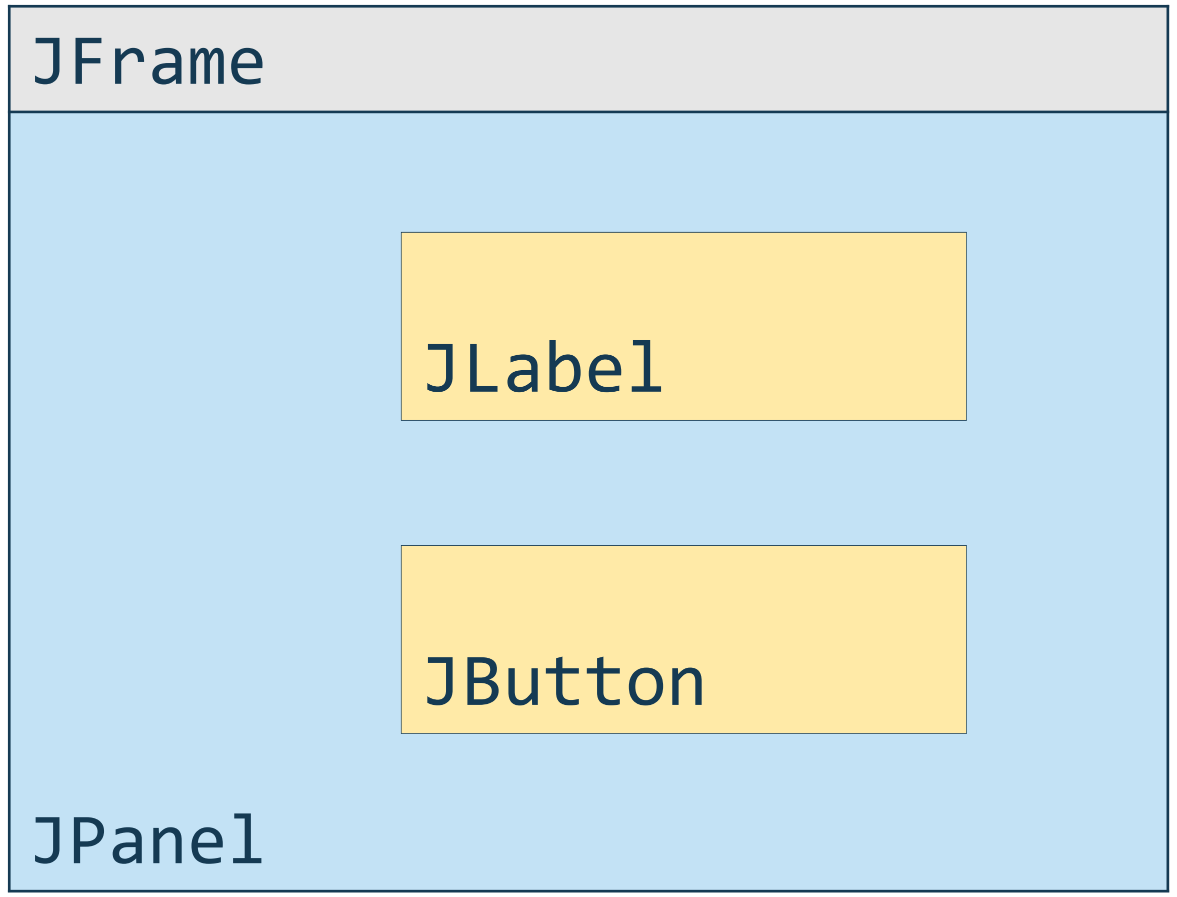
Digression - AWT inheritance hierarchy



Only the components that are supposed to contain other components are subclasses of Container

Inheritance vs containment hierarchy





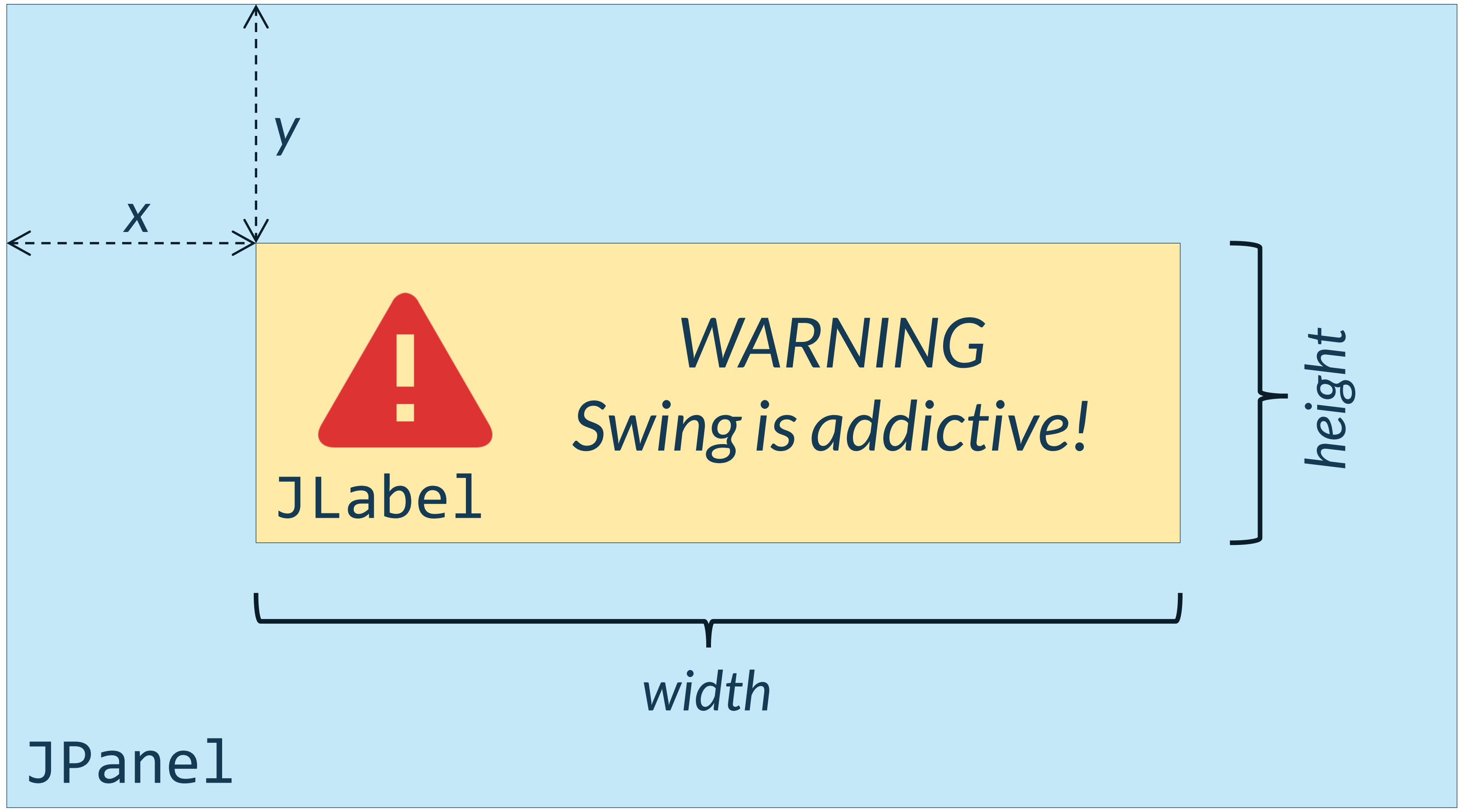
Label and button are child components of a panel

Don’t get confused!

Exercises

1. *Modify the HelloWorld example to use a JDialog and a JWindow instead of a JFrame*
   1. *Explore how window closing works*
   2. *Explore how program termination works*
2. *Modify the Hello World example to open an "Hello, World!" popup (use both JDialog and JWindow) when pressing the button*
   1. *Explore how modality of JDialog works*
   2. *Explore window closing and program termination*

Almost “NO” fixed layout



The position, size and location, of a component is decided by the layout manager of its container

Each component is responsible to indicate its preferred, minimum and maximum sizes

Each Swing component knows how to calculate its preferred, minimum and maximum sizes

Each container has its own layout manager

A layout manager has two main responsibilities

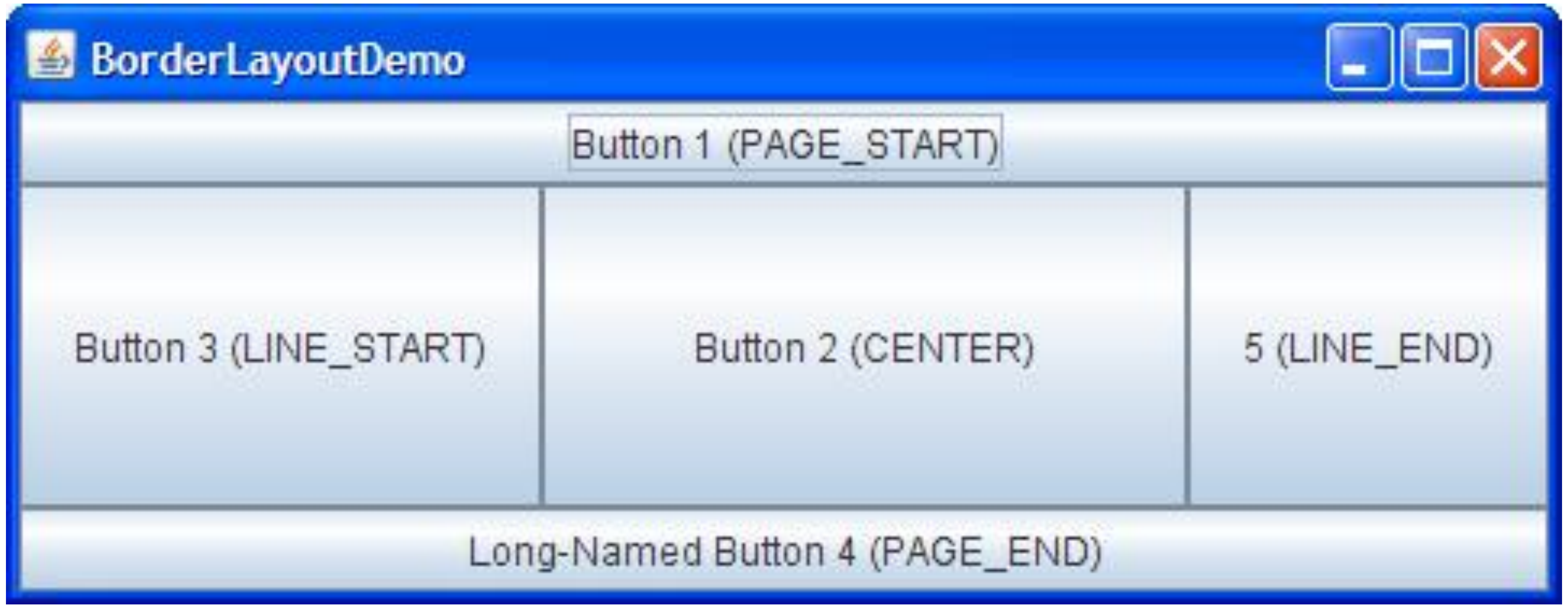
1. *layout the child components given their preferences and eventually a set of constraints*
2. *calculate the container preferred, minimum, and maximum sizes*

Since each container has its own layout manager, the process is “recursive”

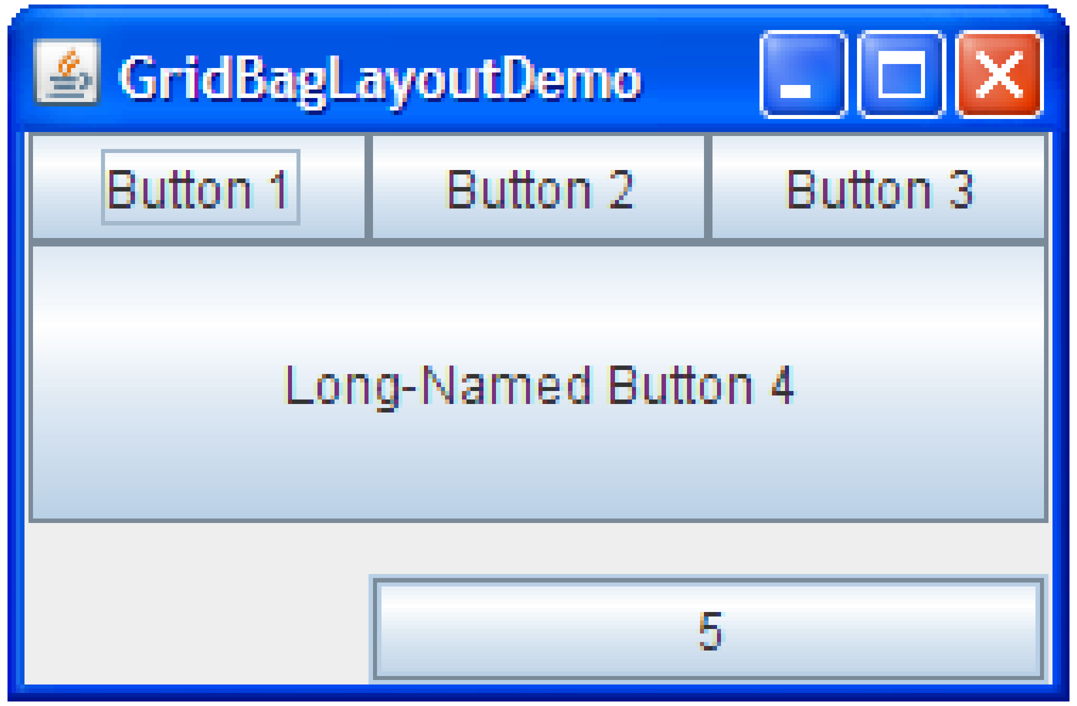
Layout managers

Common (my favorites) layout managers

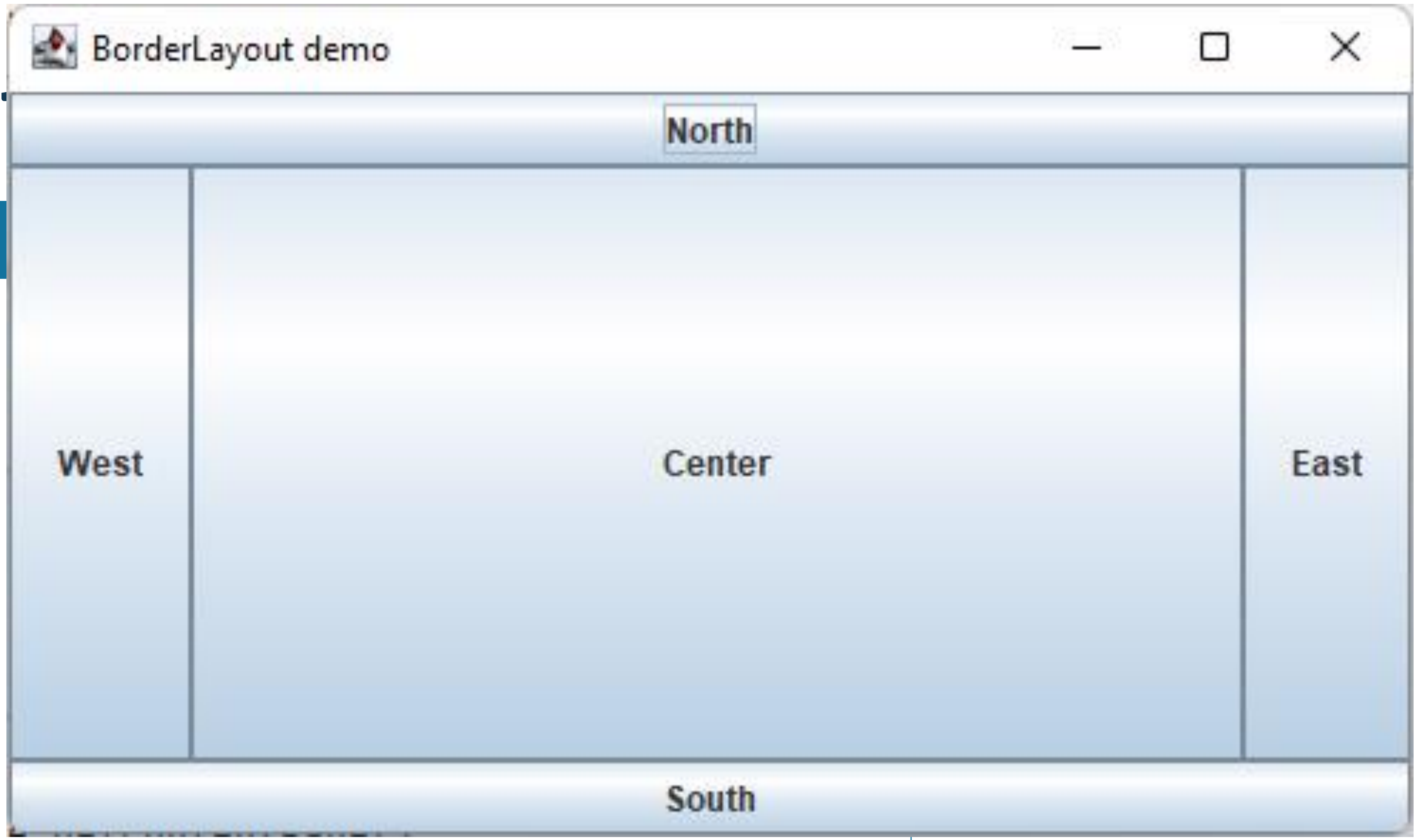
BorderLayout



GridBagLayout



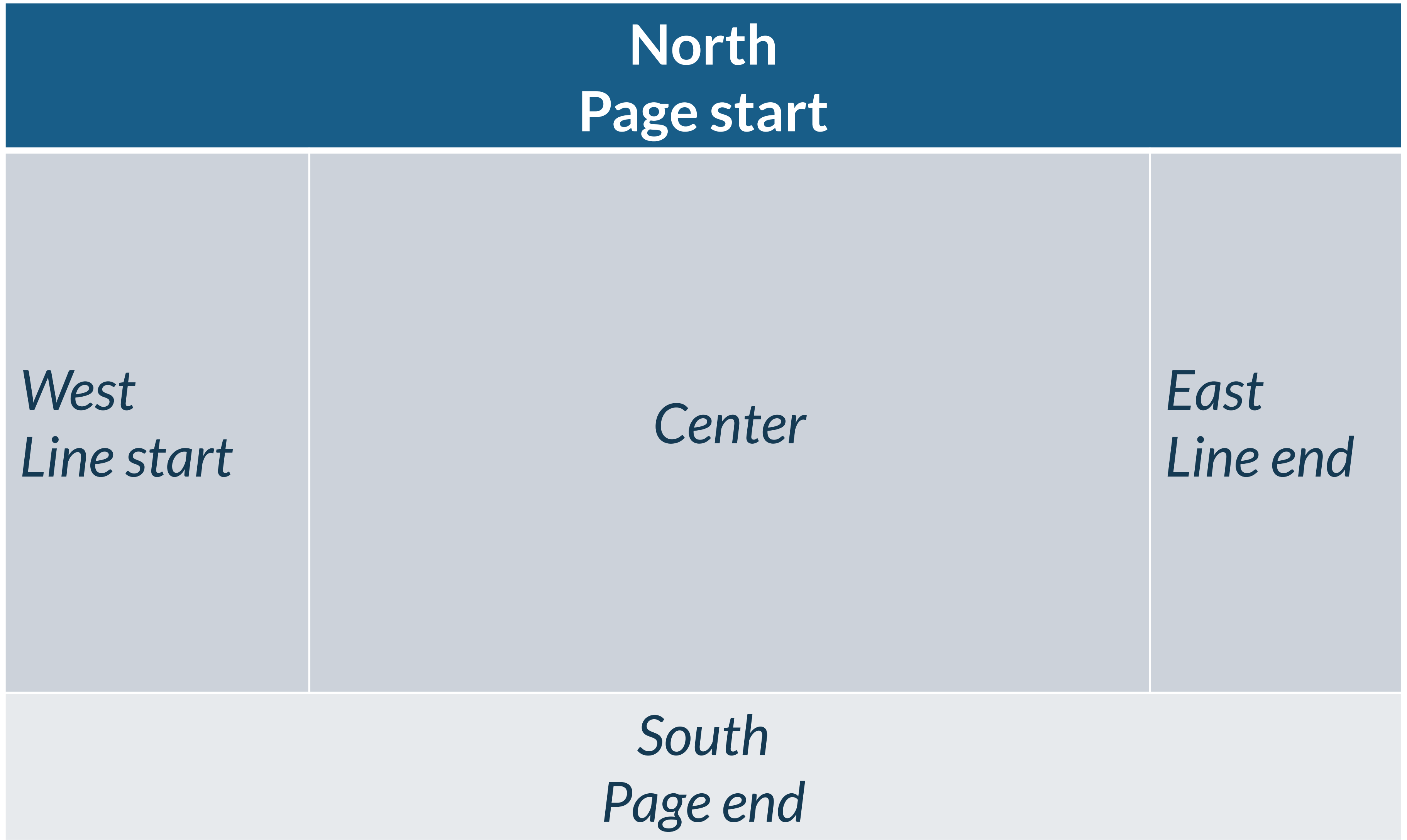
BorderLayou



BorderLayoutDemo.java

```
public class BorderLayoutDemo {

    public static void main(String[] args) {
        SwingUtilities.invokeLater(BorderLayoutDemo::run)
    }

    private static void run() {
        JFrame frame = new JFrame("BorderLayout demo");
        frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        Container cp = frame.getContentPane();
        cp.setLayout(new BorderLayout());
        cp.add(new JButton("North"), BorderLayout.NORTH);
        cp.add(new JButton("South"), BorderLayout.SOUTH);
        cp.add(new JButton("East"), BorderLayout.EAST);
        cp.add(new JButton("West"), BorderLayout.WEST);
        cp.add(new JButton("Center"), BorderLayout.CENTER);
        frame.setSize(500, 400);
        frame.setVisible(true);
    }
}
```

BorderLayout



North
Page start

West
Line start

Center

East
Line end

South
Page end

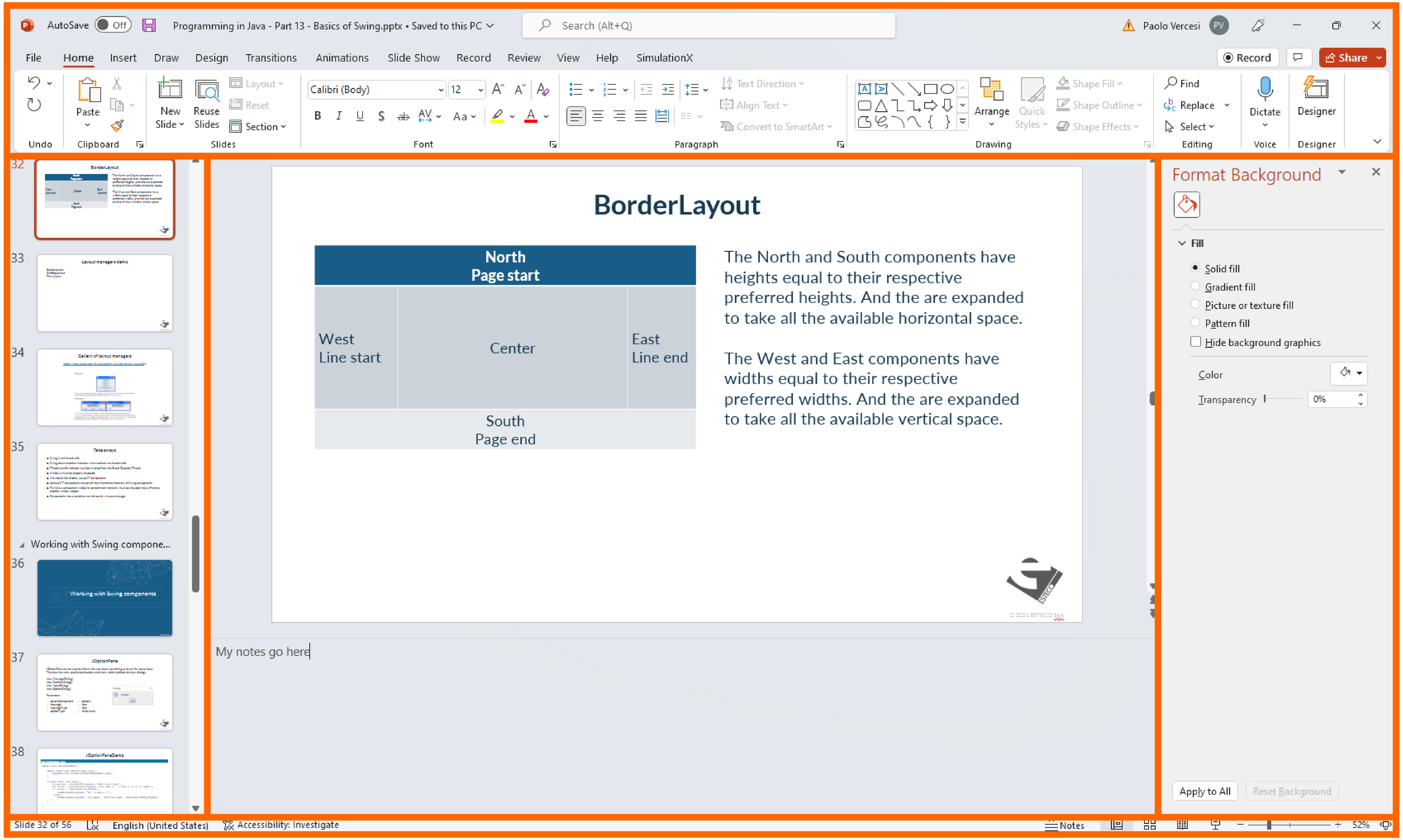
The maximum number of components is 5

The position of the component in the layout defines the constraints to which a component is subject

When using the BorderLayout

- *The North and South components have heights equal to their respective preferred heights. And they are expanded to take all the available horizontal space.*
- *The West and East components have widths equal to their respective preferred widths. And they are expanded to take all the available vertical space.*
- *The Center component takes all the available horizontal and vertical space.*

Familiar enough!



Programming in Java - Part 13 - Basics of Swing.pptx • Saved to this PC

BorderLayout

North Page start		
West Line start	Center	East Line end
South Page end		

The North and South components have heights equal to their respective preferred heights. And the are expanded to take all the available horizontal space.

The West and East components have widths equal to their respective preferred widths. And the are expanded to take all the available vertical space.

© 2021 ESTECO SpA

My notes go here

Format Background

Fill

- ● Solid fill
- ○ Gradient fill
- ○ Picture or texture fill
- ○ Pattern fill
- □ Hide background graphics

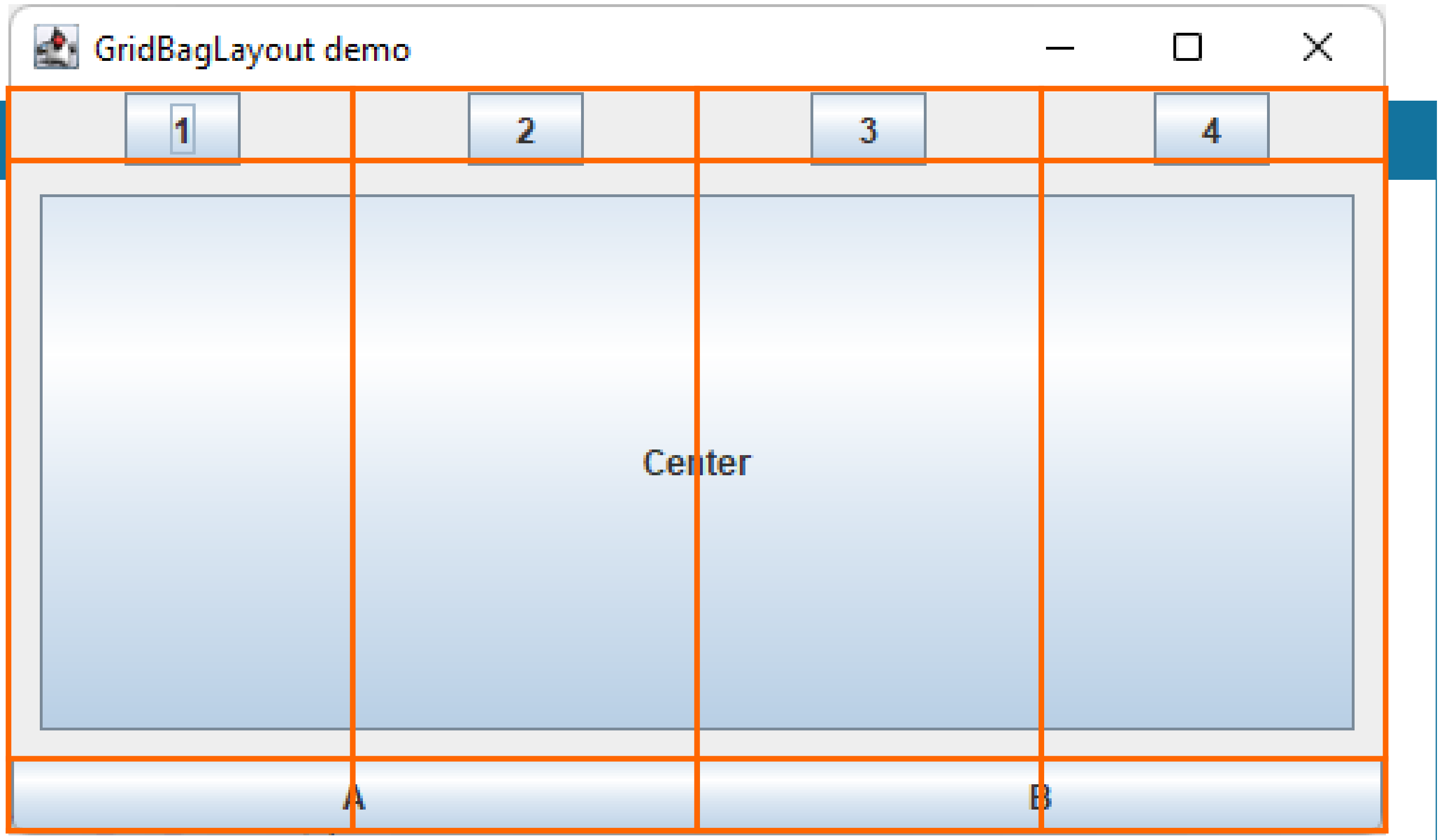
Color

Transparency 0%

Apply to All | Reset Background

Slide 32 of 56 | English (United States) | Accessibility: Investigate | Notes | 52%

GridBagLayout demo

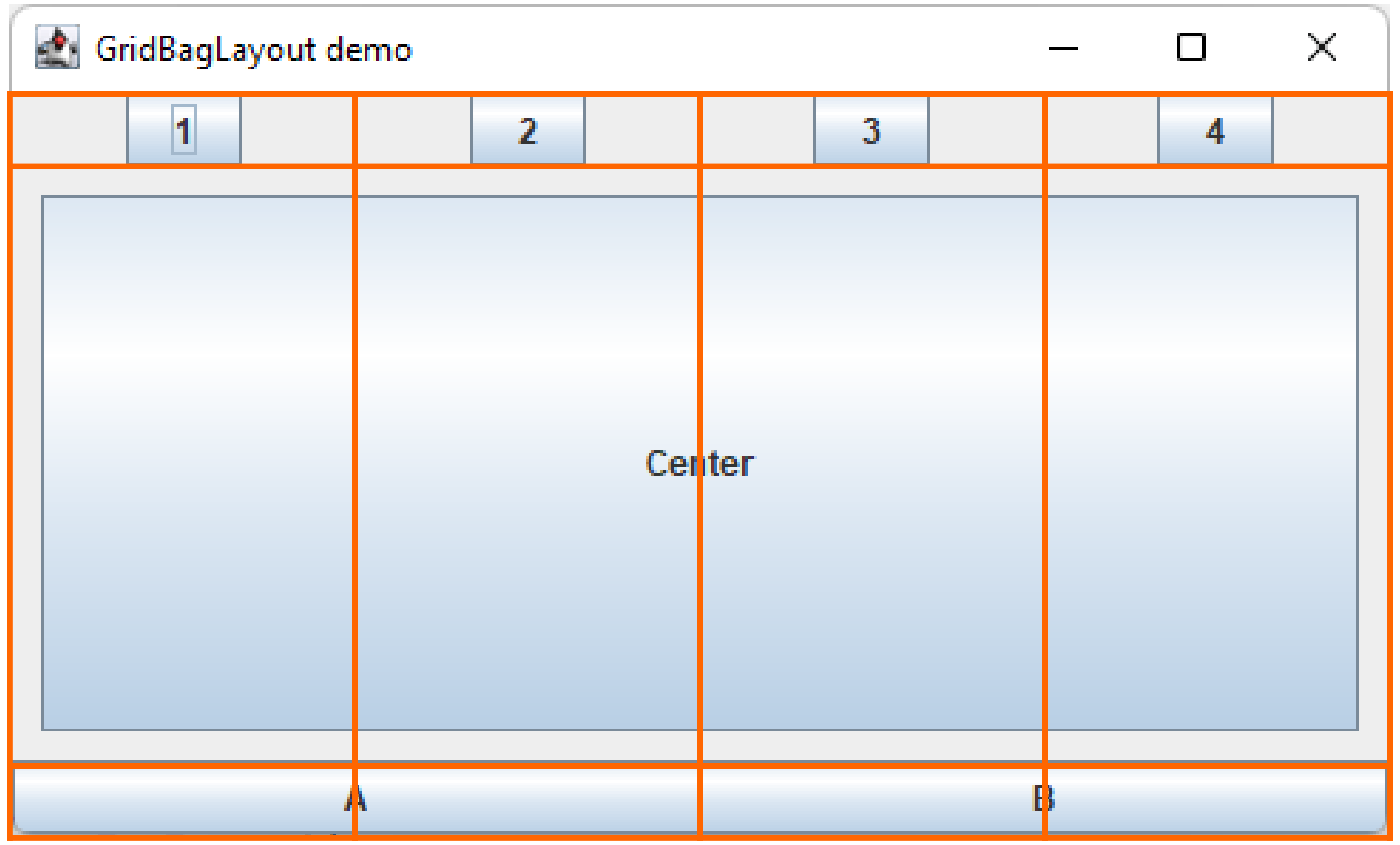


GridBagLayoutDemo.java

```
public class GridBagLayoutDemo {

  public static void main(String[] args) {
    SwingUtilities.invokeLater(GridBagLayoutDemo::run);
  }

  private static void run() {
    JFrame frame = new JFrame("GridBagLayout demo");
    frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    Container cp = frame.getContentPane();
    cp.setLayout(new GridBagLayout());
    cp.add(new JButton("1"), new GridBagConstraints(0, 0, 1, 1, 1.0, 0.0, CENTER, NONE, new Insets(0, 0, 0, 0), 0, 0 ));
    cp.add(new JButton("2"), new GridBagConstraints(1, 0, 1, 1, 1.0, 0.0, CENTER, NONE, new Insets(0, 0, 0, 0), 0, 0 ));
    cp.add(new JButton("3"), new GridBagConstraints(2, 0, 1, 1, 1.0, 0.0, CENTER, NONE, new Insets(0, 0, 0, 0), 0, 0 ));
    cp.add(new JButton("4"), new GridBagConstraints(3, 0, 1, 1, 1.0, 0.0, CENTER, NONE, new Insets(0, 0, 0, 0), 0, 0 ));
    cp.add(new JButton("Center"), new GridBagConstraints(0, 1, 4, 1, 1, 1, CENTER, BOTH, new Insets(10, 10, 10, 10), 0, 0 ));
    cp.add(new JButton("A"), new GridBagConstraints(0, 2, 2, 1, 1.0, 0.0, CENTER, HORIZONTAL, new Insets(0, 0, 0, 0), 0, 0 ));
    cp.add(new JButton("B"), new GridBagConstraints(2, 2, 2, 1, 1.0, 0.0, CENTER, HORIZONTAL, new Insets(0, 0, 0, 0), 0, 0 ));
    frame.setSize(500, 300);
    frame.setVisible(true);
  }
}
```

x, y, width, height, weightx, weighty, anchor, fill, insets, padx, pady

GridBagLayout



The GridBagLayout creates a "virtual" grid that can be extended indefinitely.

Each components is subject to many constraints

- *x, y position in the grid*
- *width, height horizontal and vertical span*
- *weightx, weighty define the weight of the corresponding columns (rows), Horizontal (vertical) extra space is assigned based to the column (row) weight. Define also how much horizontal (vertical) extra space is given to the component*
- *anchor how to position the component in the cell*
- *fill how to resize the component in the cell, depending on its weight*
- *insets how much space we should put around the component*
- *padx, pady internal padding of the component*

Assignment 1

Define the GridBagConstraints that, when used with a GridBagLayout, produce the same effects of the five constraints of the BorderLayout, NORTH, WEST, CENTER, EAST, SOUTH.

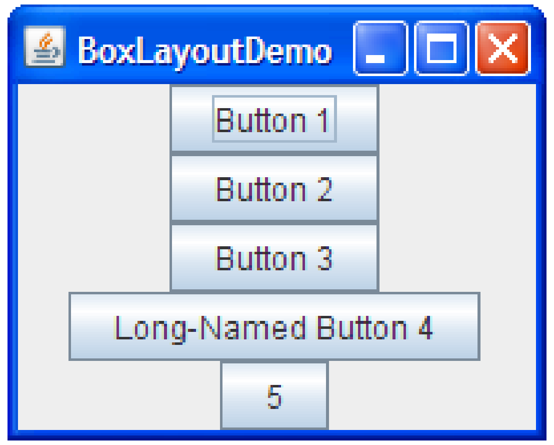
Assignment 2

Implement a "fixed" layout manager

Gallery of layout managers

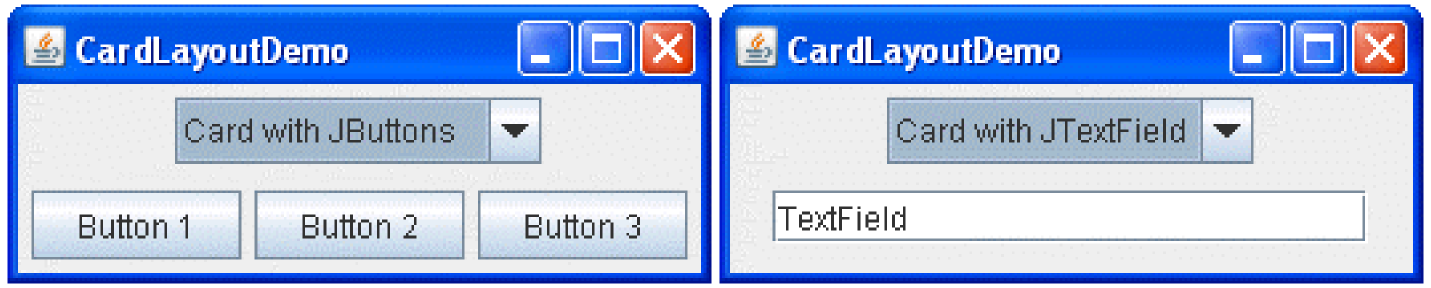
https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

BoxLayout



The `BoxLayout` class puts components in a single row or column. It respects the components' requested maximum sizes and also lets you align components. For further details, see How to Use BoxLayout.

CardLayout



The `CardLayout` class lets you implement an area that contains different components at different times. A `CardLayout` is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the `CardLayout` displays. An alternative to using `CardLayout` is using a tabbed pane, which provides similar functionality but with a pre-defined GUI. For further details, see How to Use CardLayout.

Swing is not thread-safe

Most Swing object methods are not thread-safe, invoking them from multiple threads risks thread interference or memory consistency errors

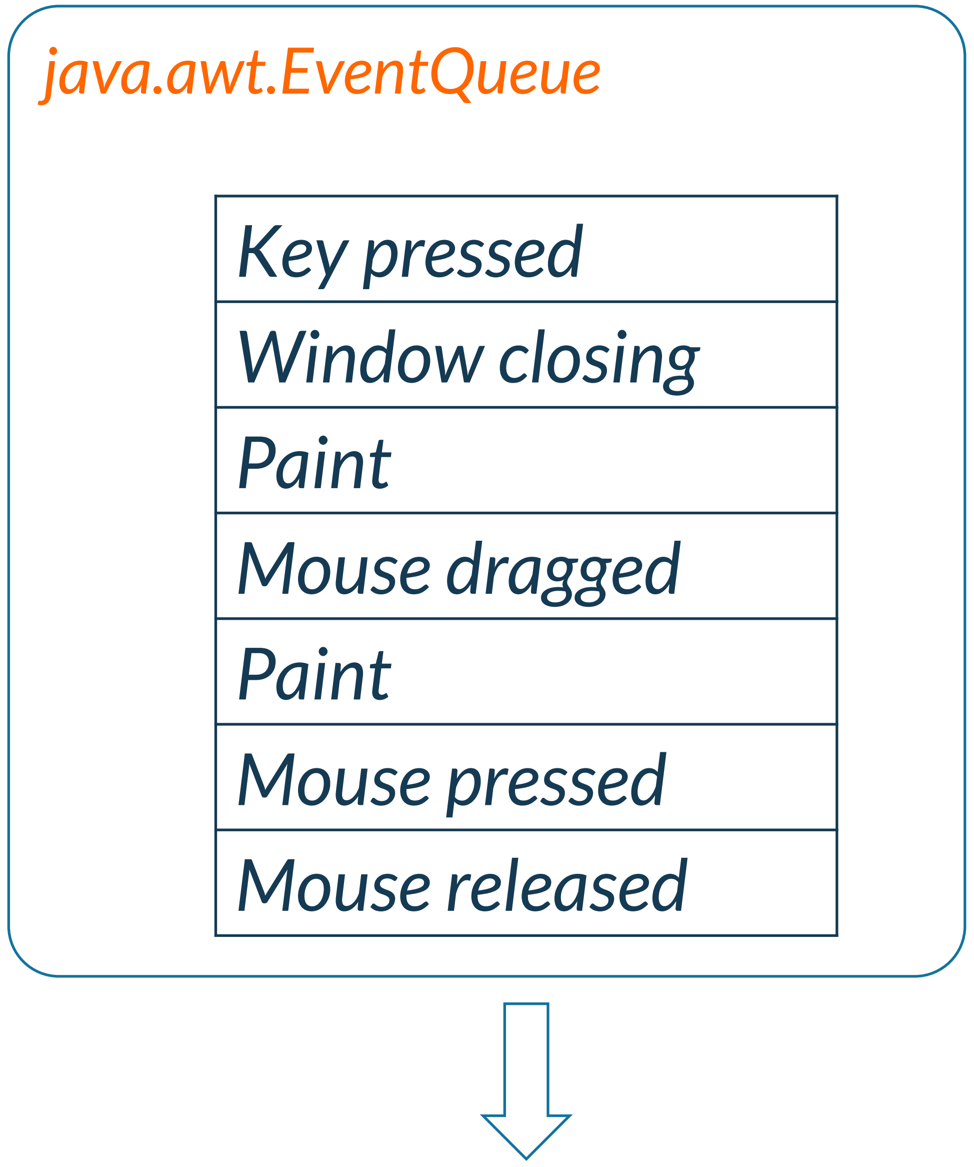
Some Swing component methods are labelled thread-safe in the API specification; these can be safely invoked from any thread. All other Swing component methods must be invoked from the event dispatch thread

Swing event handling code runs on a special thread known as the event dispatch thread (EDT) and most of the code that invokes Swing methods also runs on this thread

Programs that ignore this rule may seems to run correctly most of the times but are subject to unpredictable errors that are difficult to reproduce

https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/swing/package-summary.html

The event queue & event dispatch thread



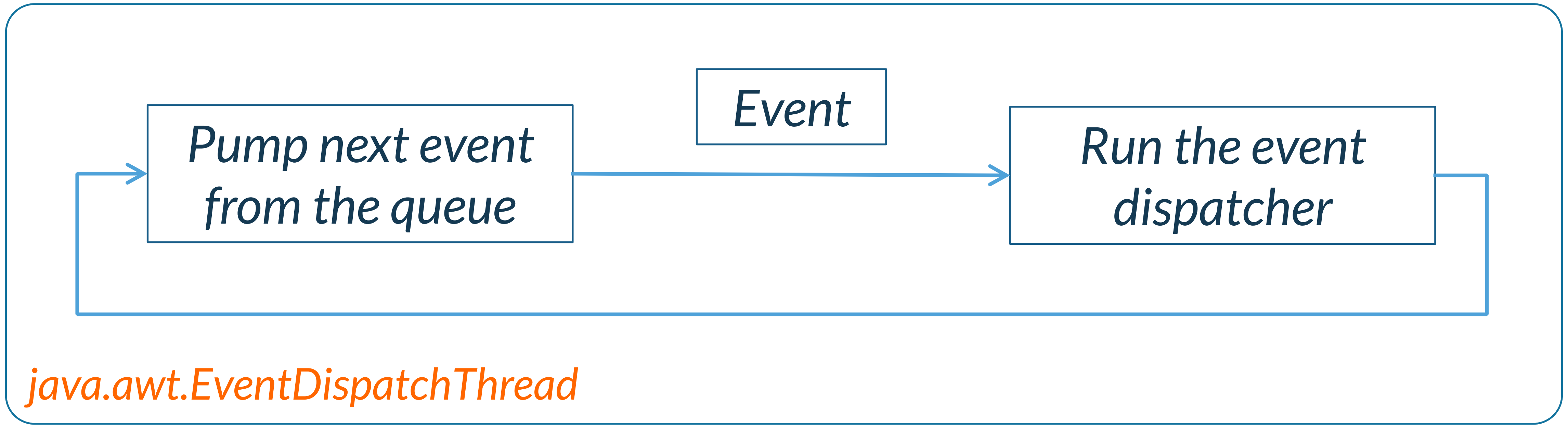
java.awt.EventQueue

- Key pressed
- Window closing
- Paint
- Mouse dragged
- Paint
- Mouse pressed
- Mouse released

The event dispatch thread is a thread used to process the events enqueued in an event queue

Swing/AWT has several types of events

- *Action*
- *Component*
- *Container*
- *Mouse*
- *Mouse wheel*
- *Key*
- *Window*
- *Focus*
- *Text*
- *etc.*



Pump next event from the queue

Event

Run the event dispatcher

java.awt.EventDispatchThread

Using the event dispatch thread

The code that handles Swing events is invoked from the event dispatch thread

If you need to determine whether your code is running on the event dispatch thread, invoke [javax.swing.SwingUtilities.isEventDispatchThread]()

Tasks on the event dispatch thread must finish quickly; if they don't, unhandled events back up and the user interface becomes unresponsive

Longer tasks should run in background, i.e., without blocking the GUI by using a SwingWorker

Take aways

- ❑ *To make a component visible, its containment hierarchy must be included into a visible JFrame o another visible window object*
- ❑ *Swing provides three types of windows*
- ❑ *In general, an application has just one JFrame and it can have more instances of JDialog or JWindow*
- ❑ *We should not directly use AWT components, even if we still use AWT classes*
- ❑ *Windows must be properly disposed*
- ❑ *Most Swing components are subclasses of AWT components*
- ❑ *Components into a container are laid out by a layout manager*
- ❑ *Swing is not thread safe*
- ❑ *Swing documentation indicates what methods are thread-safe*
- ❑ *Thread-unsafe methods must be invoked from the event dispatch thread*

Working with Swing components

Interactions with the GUI

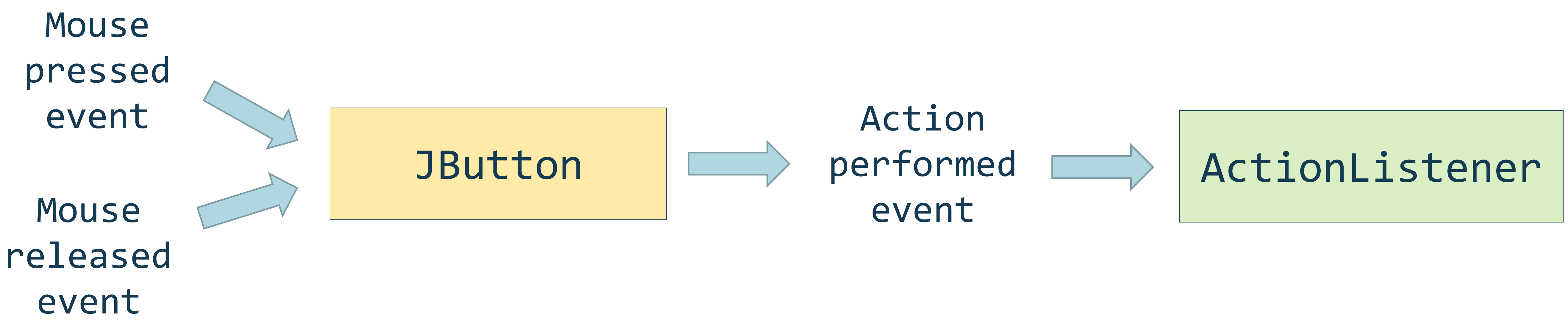
Swing components receive mouse and keyboard events from the window system, and they translate these events into events at the component level

In other words, Swing components fire events in response to user actions

Event processing happens in the event dispatch thread, as the name suggest

While processing events, it's always (thread) safe to invoke Swing methods from the same thread

From GUI events to component events



GUI events are dispatched to components. E.g., the Mouse pressed, and Mouse released events are dispatched to the JButton

Components translate GUI events into component events. E.g., the Mouse pressed and Mouse released events trigger an Action performed event

Registered listeners receive the component event. E.g., an ActionListener registered to the JButton receives the Action performed event

All these events are dispatched through the event dispatch thread

Swing components

- *buttons*
  - *push button*
  - *check box*
  - *toggle button*
  - *radio button*
- *choosers*
  - *color chooser*
  - *file chooser*
- *combo box*
- *list*
- *menus*
  - *menu bar*
  - *popup menu*
  - *menu*
  - *menu item*
- *option pane*
- *panes*
  - *editor pane*
  - *text pane*
- *panel*
- *progress bar*
- *scroll pane*
- *separator*
- *slider*
- *spinner*
- *split pane*
- *tabbed pane*
- *table*
- *text components*
  - *text field*
  - *password field*
  - *text area*
  - *text pane*
- *tool bar*
- *tool tip*
- *tree*

Google “Swing components library”

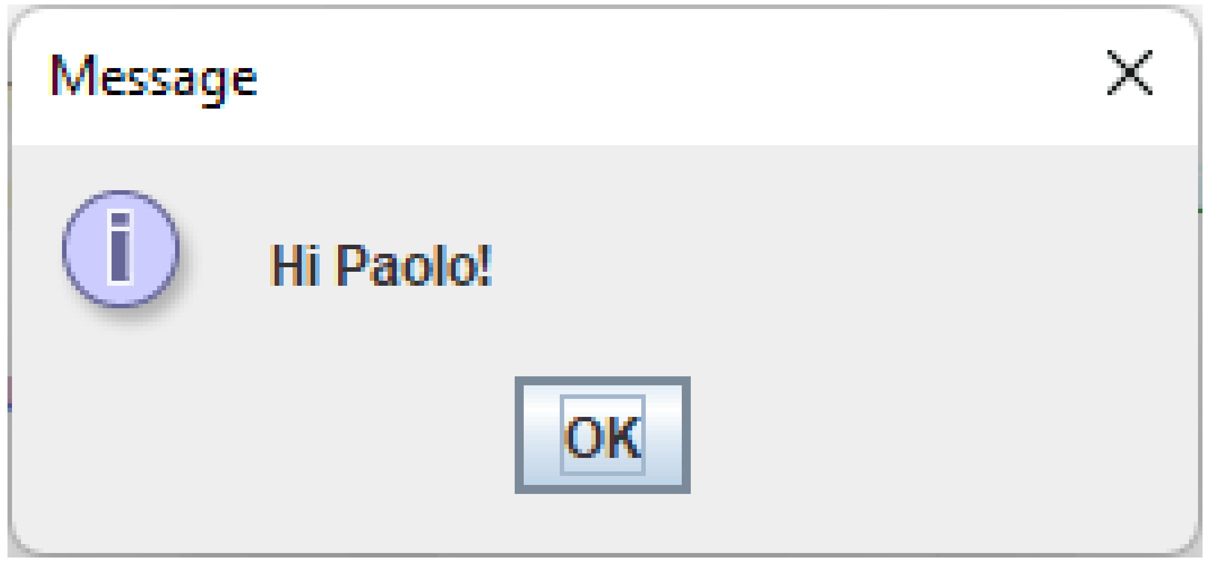
JOptionPane

JOptionPane can be used to inform the user about something or to ask for some input.
The class has many public constructors and many static methods to show dialogs.

```
showMessageDialog()
showConfirmDialog()
showInputDialog()
showOptionDialog()
```

Parameters

- *parentComponent*
- *message*
- *messageType*
- *optionType*
- *options*
- *icon*
- *title*
- *initialvalue*



Message

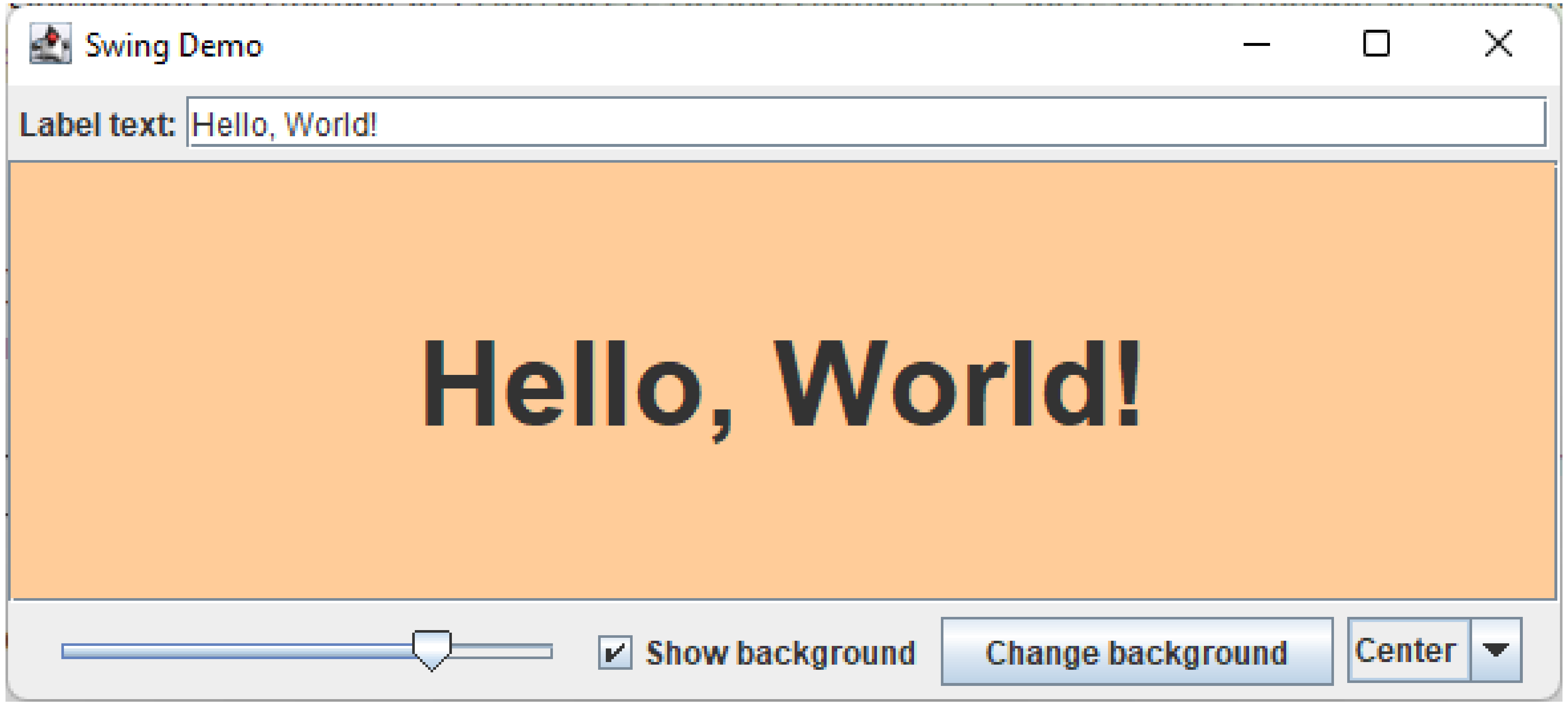
Hi Paolo!

JOptionPaneDemo

OptionPaneDemo.java

```java
import static javax.swing.JOptionPane.showConfirmDialog;
import static javax.swing.JOptionPane.showInputDialog;
import static javax.swing.JOptionPane.showMessageDialog;

public class JOptionPaneDemo {

    public static void main(String[] args) {
        SwingUtilities.invokeLater(JOptionPaneDemo::demo);
    }

    private static void demo() {
        String name = showInputDialog(null, "What's your name");
        int result = showConfirmDialog(null, "Your name is: " + name + "\n Is it right?");
        if (result == JOptionPane.OK_OPTION) {
            showMessageDialog(null, "Hi " + name + "!");
        } else {
            showMessageDialog(null, "Try again", "Incorrect name", JOptionPane.ERROR_MESSAGE);
        }
    }
}
```

SwingDemo



Label text: Hello, World!

Hello, World!

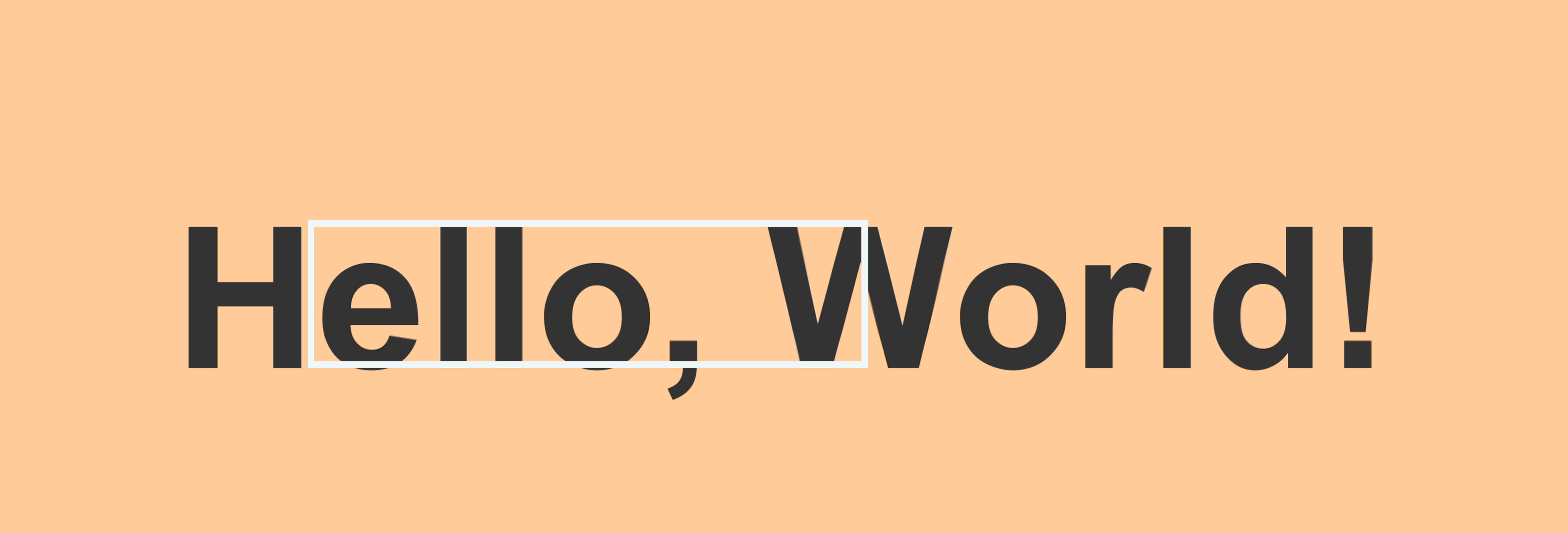
☑ Show background

Center

Swing demo – Setting up and showing the JFrame

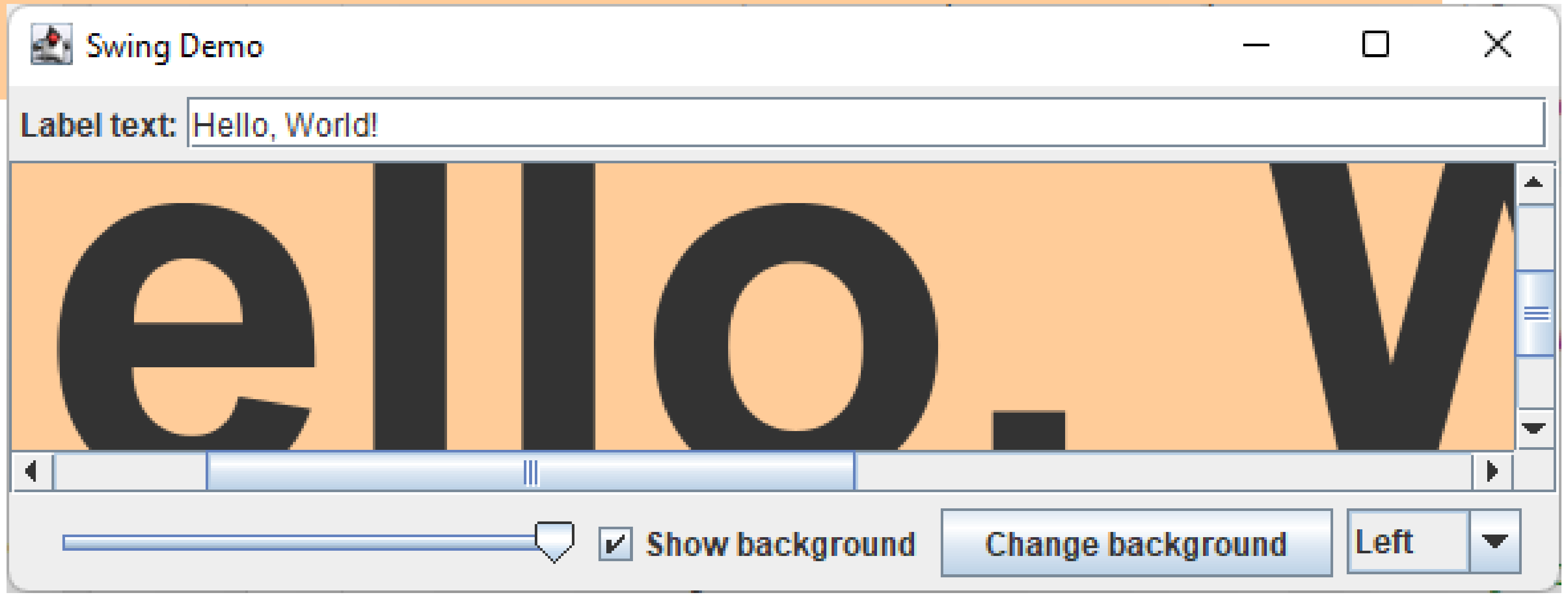
```
JFrame frame = new JFrame("Swing Demo");
frame.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
Container cp = frame.getContentPane();
cp.setLayout(new BorderLayout());
JLabel label = new JLabel("Hello, World!");
label.setOpaque(true);
    …
cp.add(new JScrollPane(label), BorderLayout.CENTER);
cp.add(northPanel, BorderLayout.NORTH);
cp.add(southPanel, BorderLayout.SOUTH);
frame.setSize(600, 200);
frame.setVisible(true);
```

The JScrollPane



The JScrollPane shows the component through a viewport

When the viewport is not wide enough, scrollbars are added to the view



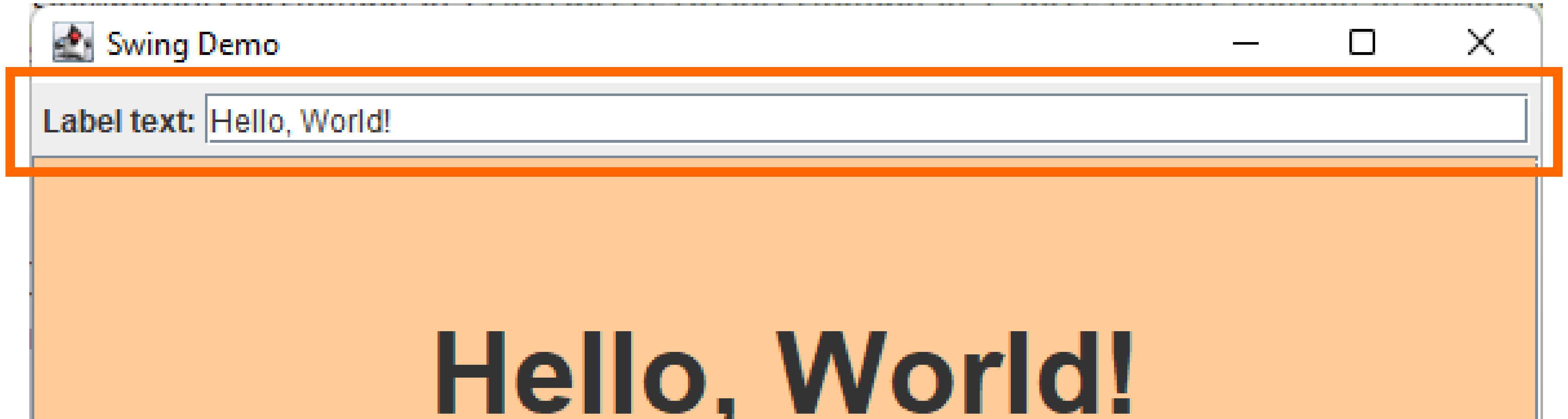
Label text: Hello, World!

☑ Show background

Left

The North panel

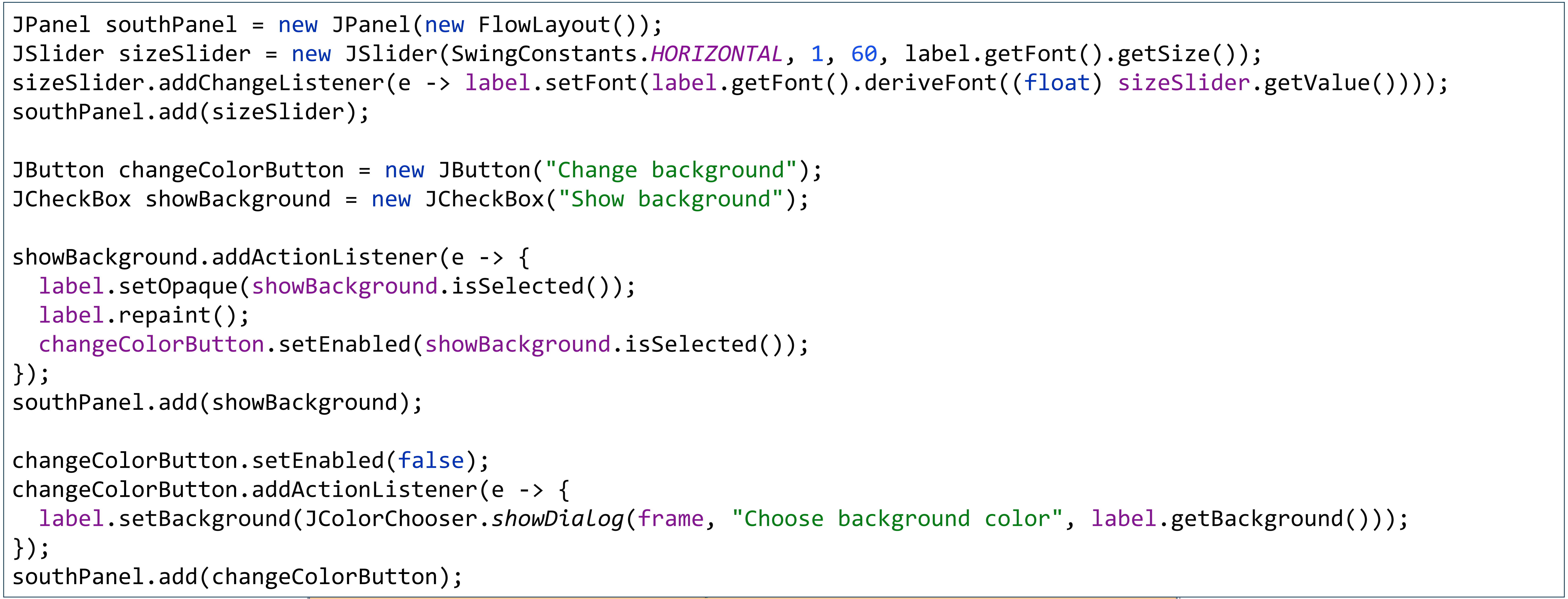
```
JPanel northPanel = new JPanel(new GridBagLayout());
JLabel textLabel = new JLabel("Label text:");
northPanel.add(textLabel, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(0, 4,0, 0), 0, 0));
JTextField textField = new JTextField(30);
textField.addActionListener(e -> label.setText(textField.getText()));
northPanel.add(textField, new GridBagConstraints(1, 0, 1, 1, 1.0, 0.0,
GridBagConstraints.WEST, GridBagConstraints.HORIZONTAL, new Insets(4, 4, 4, 4), 0, 0));
```



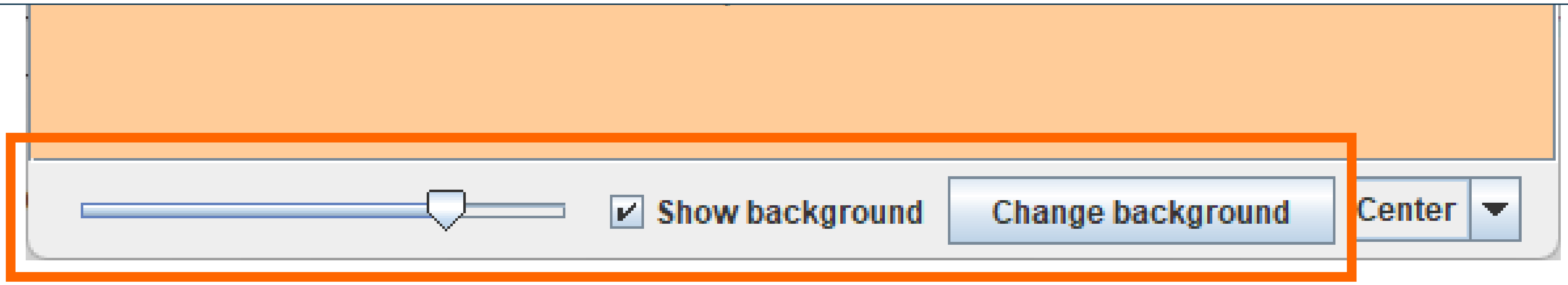
Label text: Hello, World!

Hello, World!

The South panel 1/2

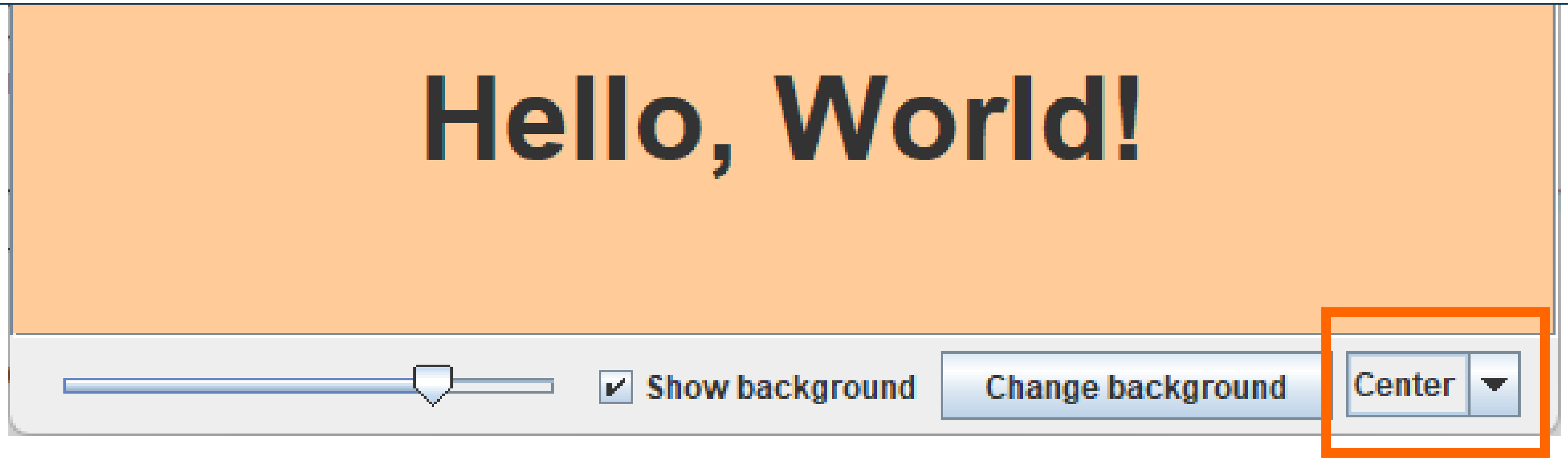


```
JPanel southPanel = new JPanel(new FlowLayout());
JSlider sizeSlider = new JSlider(SwingConstants.HORIZONTAL, 1, 60, label.getFont().getSize());
sizeSlider.addChangeListener(e -> label.setFont(label.getFont().deriveFont((float) sizeSlider.getValue())));
southPanel.add(sizeSlider);

JButton changeColorButton = new JButton("Change background");
JCheckBox showBackground = new JCheckBox("Show background");

showBackground.addActionListener(e -> {
  label.setOpaque(showBackground.isSelected());
  label.repaint();
  changeColorButton.setEnabled(showBackground.isSelected());
});
southPanel.add(showBackground);

changeColorButton.setEnabled(false);
changeColorButton.addActionListener(e -> {
  label.setBackground(JColorChooser.showDialog(frame, "Choose background color", label.getBackground()));
});
southPanel.add(changeColorButton);
```



The South panel 2/2

```
JComboBox<Integer> alignmentComboBox = new JComboBox<>(
       new Integer[]{SwingConstants.LEFT, SwingConstants.CENTER, SwingConstants.RIGHT});

alignmentComboBox.setRenderer(new DefaultListCellRenderer() {
  @Override
  public Component getListCellRendererComponent(JList<?> list, Object value, int index, boolean isSelected, boolean cellHasFocus) {
    switch ((Integer) value) {
      case SwingConstants.LEFT -> value = "Left";
      case SwingConstants.CENTER -> value = "Center";
      case SwingConstants.RIGHT -> value = "Right";
    }
    return super.getListCellRendererComponent(list, value, index, isSelected, cellHasFocus);
  }
});
alignmentComboBox.setSelectedItem(label.getHorizontalAlignment());
alignmentComboBox.addActionListener(e -> {
  label.setHorizontalAlignment((Integer) alignmentComboBox.getSelectedItem());
});

southPanel.add(alignmentComboBox);
```



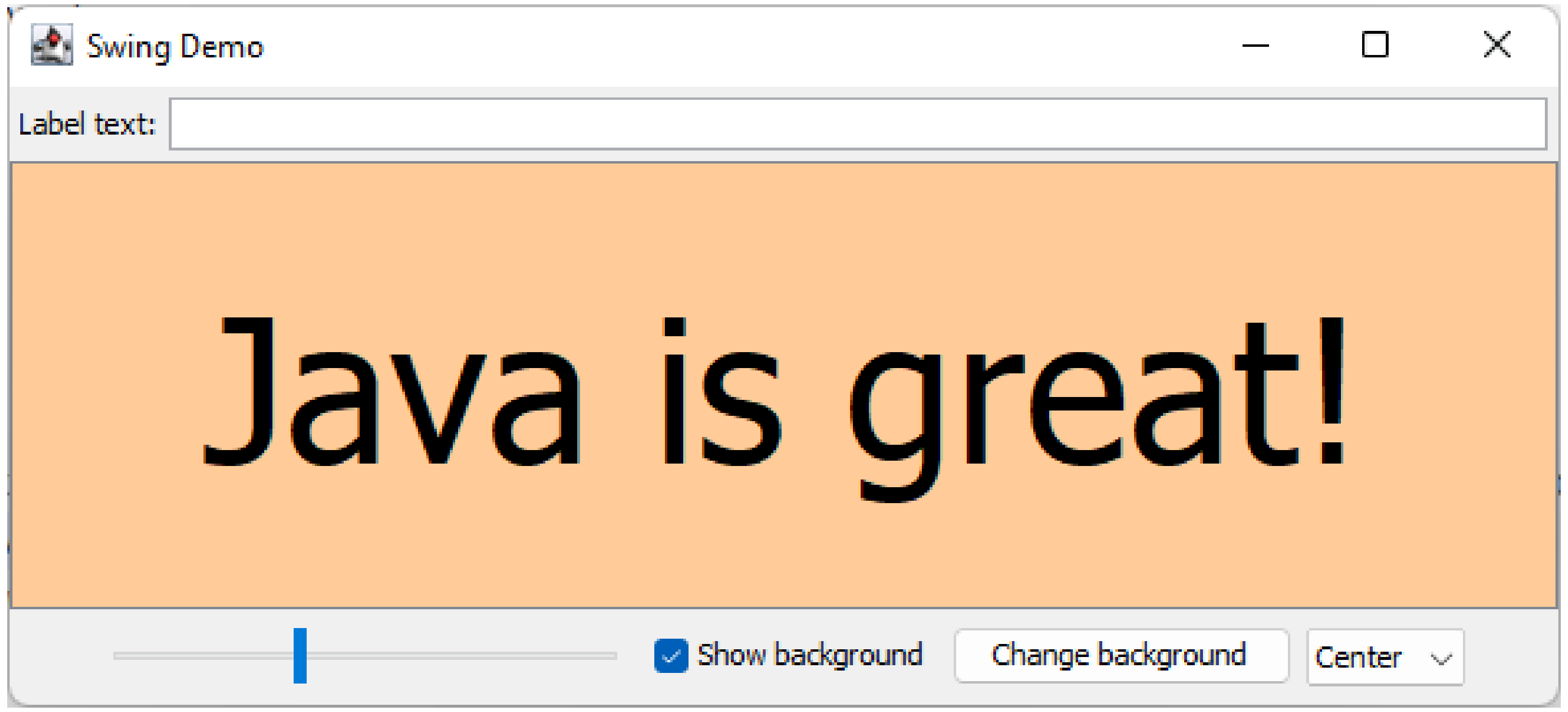
Look-and-feel

Swing allows to change the look-and-feel (L&F) of GUI applications, to adapt the appearance and the behavior of GUI components

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

or

```
UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());
```



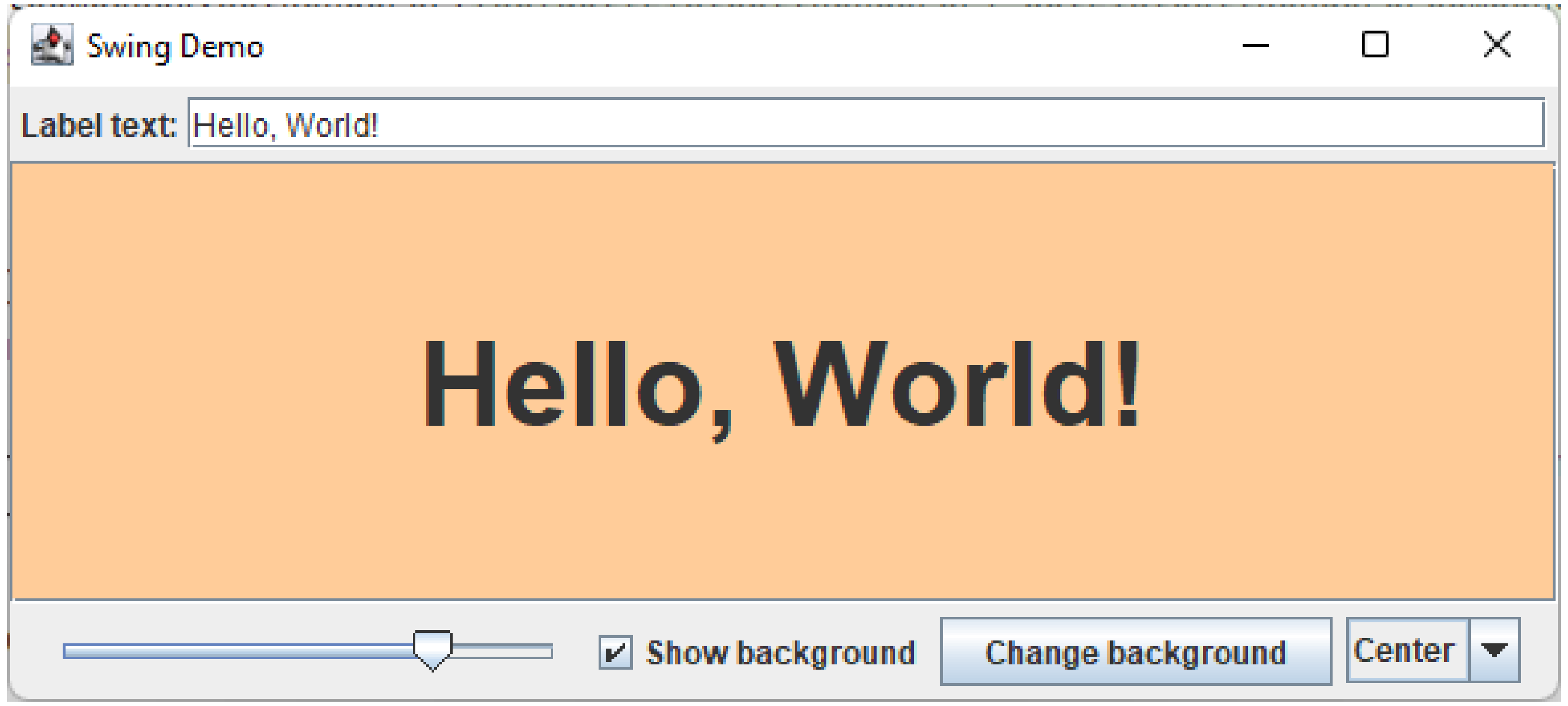
Swing Demo

Label text:

Java is great!

☑ Show background

Center



Swing Demo

Label text: Hello, World!

Hello, World!

☑ Show background

Center

https://www.oracle.com/java/technologies/a-swing-architecture.html

Take aways

❑ *Components fire events in response to user actions*

❑ *Swing has a rich and comprehensive set of components*

❑ *Swing supports multiple look-and-feels*

Writing custom Swing components

Writing custom Swing components

When writing a custom swing component, you must consider the following responsibilities

- *Painting*
- *Response to GUI events*
- *Size preferences (preferred/minimum/maximum)*

PaintDemo.java

```java
public class PaintDemo extends JComponent {

    public PaintDemo() {
    }

    @Override
    protected void paintComponent(Graphics g) {
        Graphics2D scratch = (Graphics2D) g.create();
        try {
            Dimension size = getSize();
            scratch.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
            scratch.drawLine(0, 0, size.width, size.height);
            scratch.drawLine(0, size.height, size.width, 0);
        } finally {
            scratch.dispose();
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                JFrame frame = new JFrame();
                frame.getContentPane().add(new PaintDemo());
                frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
                frame.setSize(200, 200);
                frame.setVisible(true);
            }
        });
    }
}
```

Painting

1. *Extend JComponent*
2. *Override paintComponent()*
    1. *create a new Graphics and remember to dispose it*

[*https://www.oracle.com/java/technologies/painting.html*](https://www.oracle.com/java/technologies/painting.html)

Graphics2D

- *Control the coordinate system, through affine transformation*
- *Rendering*
  - *Shapes*
  - *Text*
  - *Images*
- *Control rendering attributes*
  - *Paint*
  - *Font*
  - *Stroke*
  - *Composite*
  - *Clip*

Repainting

When does paint happen?

- *It happens in the Event Dispatch Thread*
- *You cannot decide when*
- *You can inform Swing that a component (or a part of that component) should be repainted*
  - *public void repaint()*
  - *public void repaint(int x, int y, int width, int height)*

Event management

Swing events

- *Component events*
- *Focus events*
- *Hierarchy events*
- *Input method events*
- *Key events*
- *Mouse events*
- *Mouse motion events*
- *Mouse wheel events*
- *Window events*
- *… look at subclasses of AWTEvent*

To manage events

- *Register listeners*
- *Or enable event and override processXXXEvent*

EventDemo.java

```java
public class EventDemo extends JComponent {

    private boolean pressed;

    public EventDemo() {
        enableEvents(MOUSE_EVENT_MASK | MOUSE_MOTION_EVENT_MASK);
    }

    @Override
    protected void processMouseEvent(MouseEvent e) {
        switch (e.getID()) {
            case MouseEvent.MOUSE_PRESSED -> {
                pressed = true;
                repaint();
            }
            case MouseEvent.MOUSE_RELEASED -> {
                pressed = false;
                repaint();
            }
        }
        super.processMouseEvent(e);
    }

    @Override
    protected void paintComponent(Graphics g) {
        Graphics2D scratch = (Graphics2D) g.create();
        try {
            scratch.setPaint(pressed ? Color.YELLOW : Color.BLUE);
            scratch.fillRect(0, 0, getWidth(), getHeight());
        } finally {
            g.dispose();
        }
    }
}
```

AnimationDemo.java 1/2

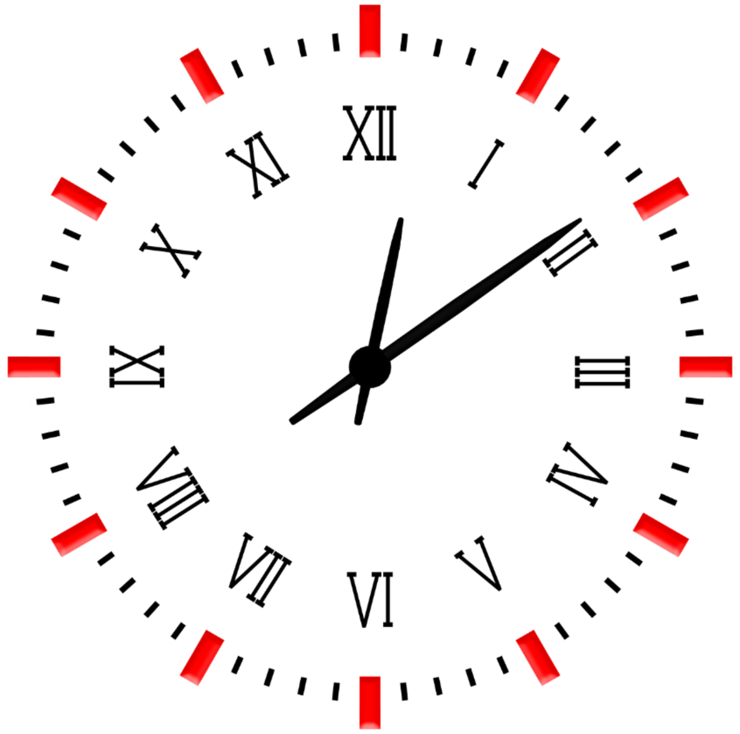
```java
public class AnimationDemo extends JComponent {

    private Dimension speed;
    private Point2D position;
    private Timer timer;

    public AnimationDemo() {
        position = new Point2D.Double(0, 0);
        speed = new Dimension(1, 1);
        timer = new Timer(50, this::update);
    }

    public void start() {
        timer.start();
    }

    public void stop() {
        timer.stop();
    }

    @Override
    protected void paintComponent(Graphics g) {
        Graphics2D scratch = (Graphics2D) g.create();
        try {
            scratch.fillOval((int) position.getX(), (int) position.getY(), 3, 3);
        } finally {
            scratch.dispose();
        }
    }

    …

}
```

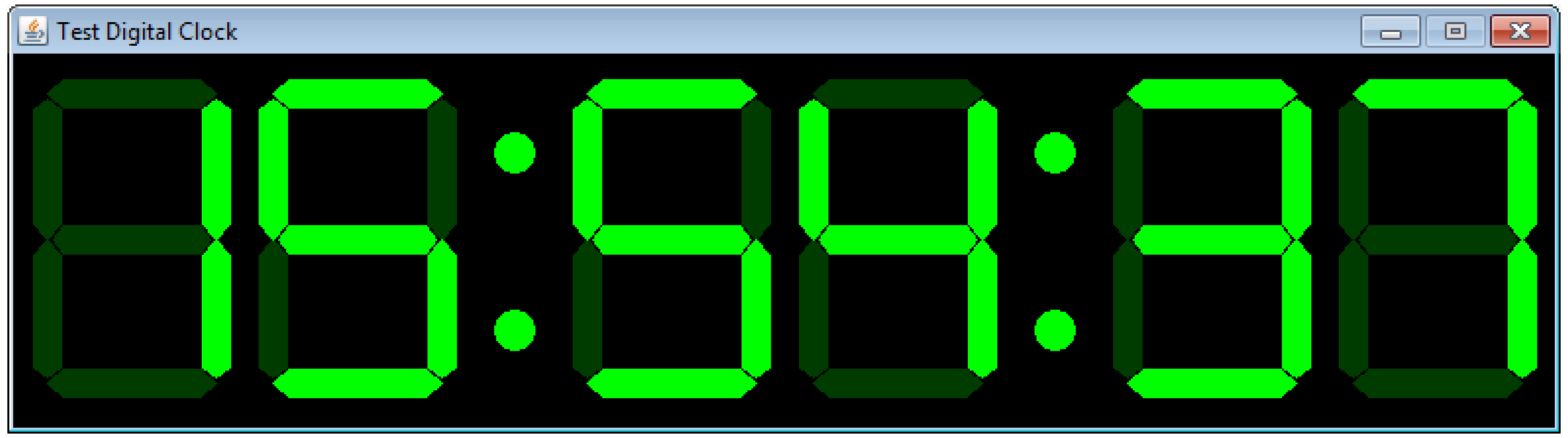
AnimationDemo.java 2/2

```java
public class AnimationDemo extends JComponent {

    public void update(ActionEvent e) {
        position.setLocation(position.getX() + speed.width, position.getY() + speed.height);
        if (position.getX() < 0) {
            speed.width = -speed.width;
            position.setLocation(-position.getX(), position.getY());
        } else if (position.getX() > getWidth()) {
            speed.width = -speed.width;
            position.setLocation(getWidth() - position.getX(), position.getY());
        }
        if (position.getY() < 0) {
            speed.height = -speed.height;
            position.setLocation(position.getX(), -position.getY());
        } else if (position.getY() > getHeight()) {
            speed.height = -speed.height;
            position.setLocation(position.getX(), getHeight() - position.getY());
        }
        repaint();
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            JFrame frame = new JFrame("Animation demo");
            AnimationDemo animationDemo = new AnimationDemo();
            frame.getContentPane().add(animationDemo);
            frame.setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
            frame.addWindowListener(new WindowAdapter() {
                @Override
                public void windowClosing(WindowEvent e) {
                    animationDemo.stop();
                    frame.dispose();
                }});
            frame.setSize(200, 200);
            frame.setVisible(true);
        });
    }
}
```

Assignment

Implement an analog and/or a digital clock





Multithreaded programming

Concurrency

“more than one task running simultaneously on a system”

Writing correct programs is hard;
writing correct concurrent programs is harder.

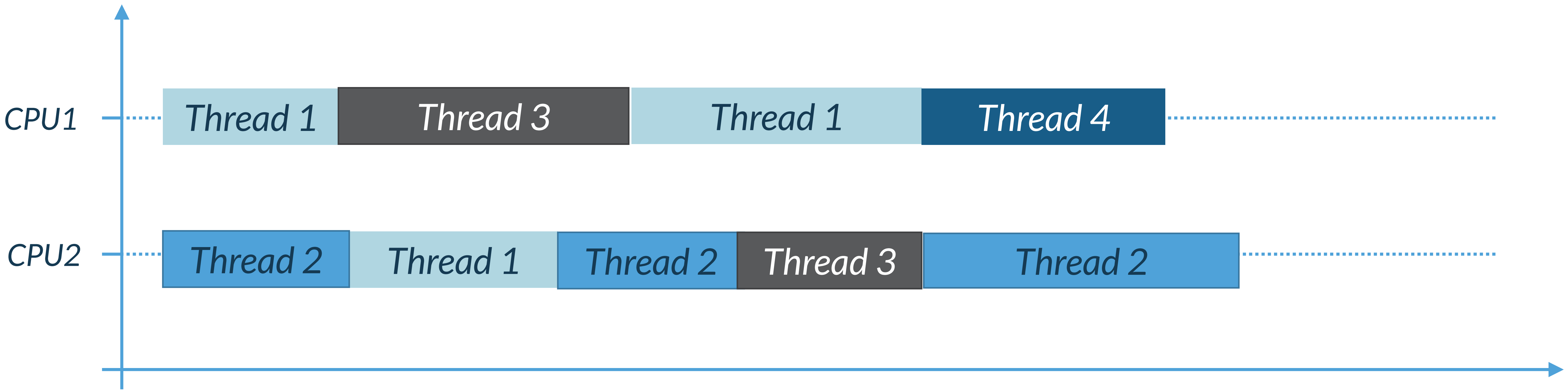
Processes and threads

A process is an executing program

A Java process contains several concurrent threads executing in a shared memory environment

A thread of execution, or simply a thread, is the smallest unit of execution to which a scheduler allocates a CPU

Java threads are scheduled (started, interrupted, and resumed) by the scheduler of the underlying operating system



The execution of a thread is assigned to a CPU until the scheduler decides to interrupt the thread execution to schedule another thread

Threads

A thread consists of a stack of calls, a program counter, and an id

In Java, threads are instances of the `java.lang.Thread` *class*

```
"RMI TCP Accept-0" #24 daemon prio=6 os_prio=0 cpu=0.00ms elapsed=325.06s tid=0x0000021167497000 nid=0x253c runnable
[0x0000004e2e0fe000]
   java.lang.Thread.State: RUNNABLE
        at java.net.PlainSocketImpl.accept0(java.base@11.0.15/Native Method)
        at java.net.PlainSocketImpl.socketAccept(java.base@11.0.15/PlainSocketImpl.java:159)
        at java.net.AbstractPlainSocketImpl.accept(java.base@11.0.15/AbstractPlainSocketImpl.java:474)
        at java.net.ServerSocket.implAccept(java.base@11.0.15/ServerSocket.java:565)
        at java.net.ServerSocket.accept(java.base@11.0.15/ServerSocket.java:533)
        at it.esteco.rmi.ssl.SslRMIServerSocketFactory$1.accept(SslRMIServerSocketFactory.java:24)
        at sun.rmi.transport.tcp.TCPTransport$AcceptLoop.executeAcceptLoop(java.rmi@11.0.15/TCPTransport.java:394)
        at sun.rmi.transport.tcp.TCPTransport$AcceptLoop.run(java.rmi@11.0.15/TCPTransport.java:366)
        at java.lang.Thread.run(java.base@11.0.15/Thread.java:829)
```

A thread is executing the code of a single method, namely the current method for that thread and the program counter contains the address of the instruction currently being executed

Starting a Thread

```java
public static void main(String[] args) throws Exception {
  var thread = new Thread(new Runnable() {
    @Override
    public void run() {
      while (true) {
        System.out.println("Running");
        try {
           Thread.sleep(2000);
        } catch (Exception ex) {
           ex.printStackTrace();
        }
      }
    }
  });
  thread.start();
  System.out.println("End of main");
}
```

```
End of main
Running
Running
Running
…
```

The Java Virtual Machine allows an application to have multiple threads of execution running concurrently, regardless the number of processors

The Thread class is used to create and start new threads of execution

The Thread class

```
public class Thread implements Runnable {

    public Thread()

    public Thread(Runnable target)

    public Thread(String name)

    public Thread(Runnable target, String name)

    …
}
```

```
public interface Runnable {

    public abstract void run();
}
```

The Runnable object that a thread runs can be either the Thread itself or the target thread passed to the constructor

The Thread class implements an empty run() method

The two constructors without the Runnable target should be used by subclasses only

To use a thread, you must either pass a Runnable in the constructor or to override the run() method

Thread instance methods

```
public void start()

@Override
public void run()

public void interrupt()

public boolean isInterrupted()

public final boolean isAlive()

public final void setName(String name)

public final String getName()

public final void join(final long millis)

public final void join(long millis, int nanos) throws InterruptedException

public final void join() throws InterruptedException

public final void setDaemon(boolean on)

public final boolean isDaemon()
```

A thread does not return a value nor throw any exception

Some of Thread static methods

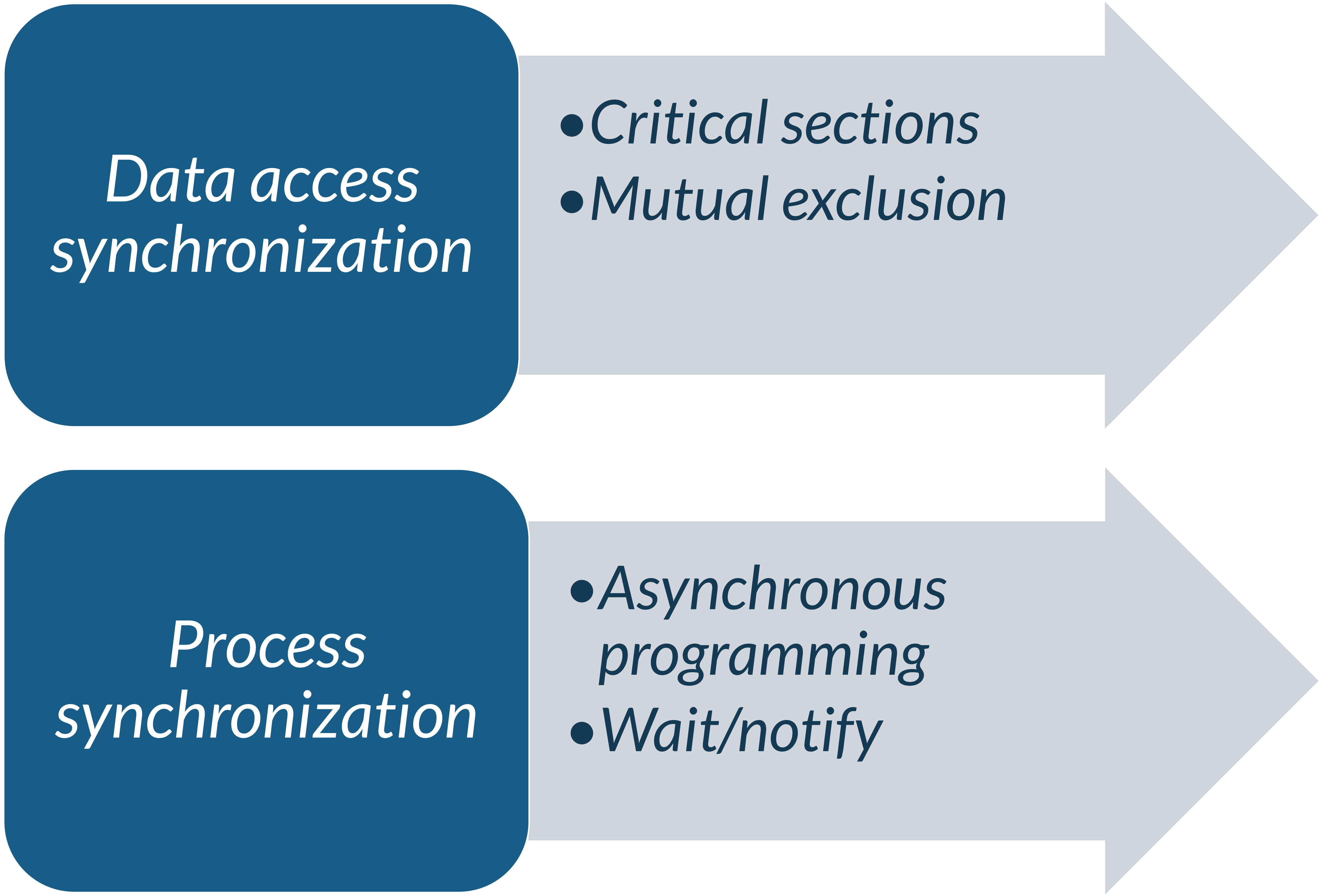
```
public static Thread currentThread()
public static void yield()
public static void sleep(long millis) throws InterruptedException
public static void sleep(long millis, int nanos) throws InterruptedException
public static boolean interrupted()
public static void dumpStack()
```

Waiting for a thread to finish

```java
public static void main(String[] args) throws Exception {
    var thread = new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                System.out.println("Running");
                try {
                    Thread.sleep(1000);
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        }
    });
    thread.start();
    System.out.println("Start waiting for the thread to finish");
    thread.join();
    System.out.println("End of main");
}
```

```
Start waiting for the thread
to finish
Running
Running
Running
Running
Running
End of main
```

Issues in concurrent programming



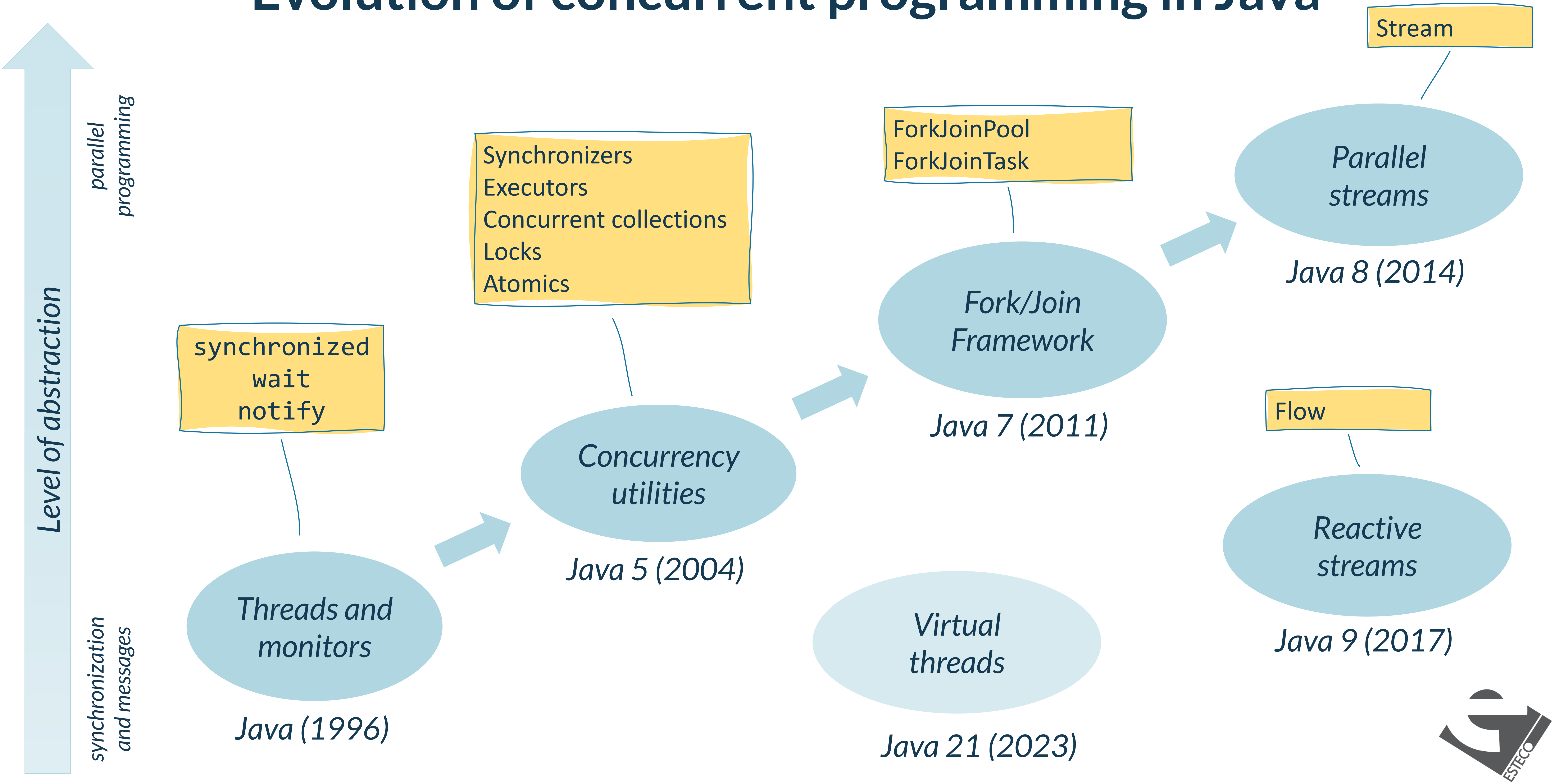
Data access synchronization

- *Critical sections*
- *Mutual exclusion*

Process synchronization

- *Asynchronous programming*
- *Wait/notify*

Evolution of concurrent programming in Java



Level of abstraction

parallel programming

synchronization and messages

synchronized
wait
notify

Threads and monitors

Java (1996)

Synchronizers
Executors
Concurrent collections
Locks
Atomics

Concurrency utilities

Java 5 (2004)

ForkJoinPool
ForkJoinTask

Fork/Join Framework

Java 7 (2011)

Virtual threads

Java 21 (2023)

Stream

Parallel streams

Java 8 (2014)

Flow

Reactive streams

Java 9 (2017)

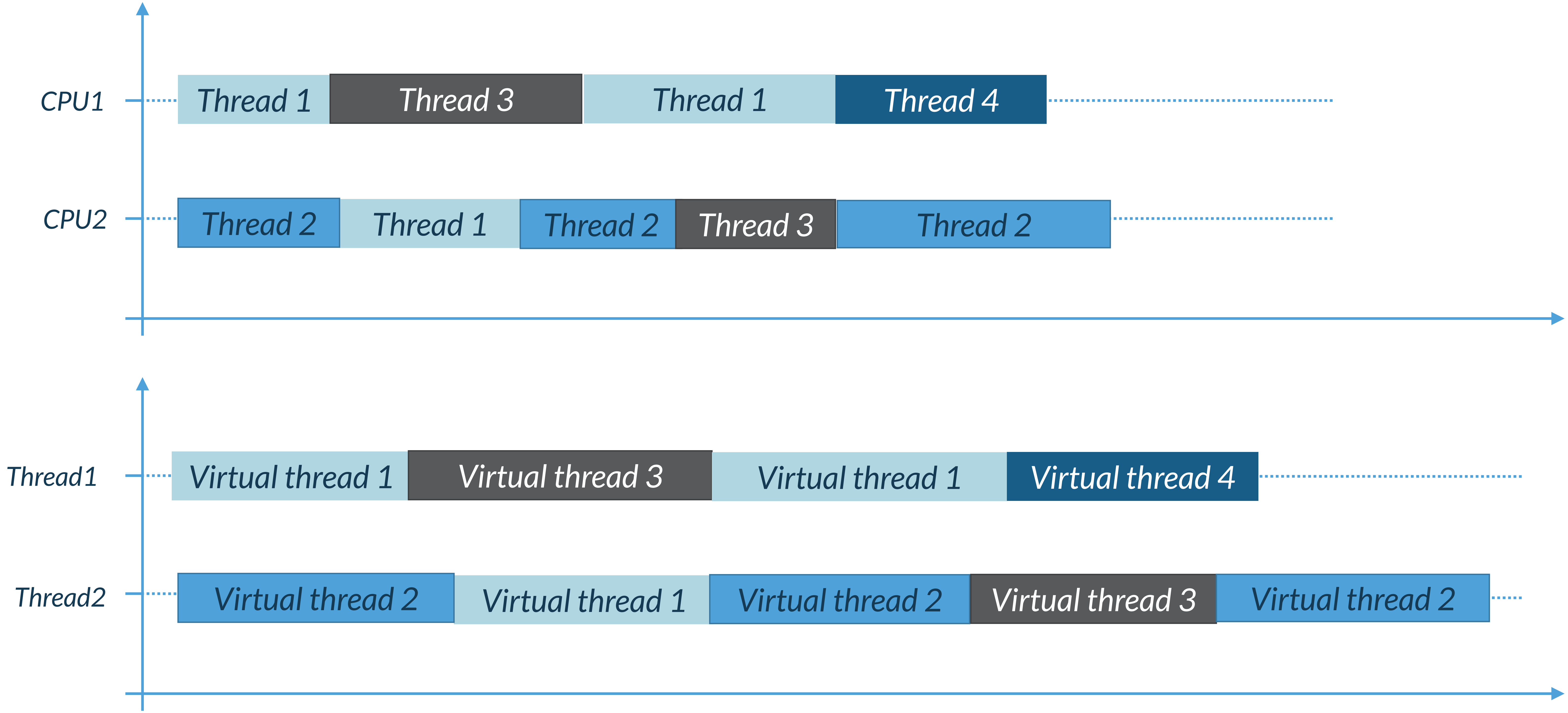
Virtual threads

- *So far, we have used the so-called platform threads*
- *A platform thread is a thin wrapper around an operating system thread*
- *A virtual thread is also an instance of* `java.lang.Thread`
  - *but it isn't tied to a specific operating system thread*
- *Virtual threads are stopped and resumed by the JVM not by the OS*
- *Virtual threads are not faster than platform threads*
- *So why were they introduced?*

High-throughput applications

- *Consider for example an application/web server*
- *It can concurrently run many database queries and/or many http connections*
- *Each of these queries or connections run on its own thread*
- *And each of these threads might be blocked waiting for the query results or the connection response*
- *Many operating system threads spend most of their lifetime waiting for some blocking I/O operation*
- *Operating system thread are considered a scarce resource*
- *So, it's a waste of resources to keep them blocked in a waiting state*
- *When a virtual threads is waiting, the associated operating system thread can be assigned to another virtual thread*

Platform vs virtual threads



https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html#GUID-DC4306FC-D6C1-4BCC-AECE-48C32C1A8DAA

Creating virtual threads

```
java.lang.Thread
public static Thread startVirtualThread(Runnable runnable)
public static Thread.Builder.OfPlatform ofPlatform()
public static Thread.Builder.OfVirtual ofVirtual()
```

```
java.lang.Thread.Builder
public Thread start(Runnable runnable)
public Thread unstarted(Runnable runnable)
```

Multithreading in Swing

MVC in Swing

Observer pattern

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Problem

An object, the observer, wants to know when a state change occurs in another object, the subject.

Example

A home automation control system wants to know when a Television object is turned on, to dim the lights of the room.

Synchronous solution

```java
public class HomeAutomation {

    private final Light light;
    private final Television television;

    public HomeAutomation(Light light, Television television) {
        this.light = light;
        this.television = television;
    }

    public void run() {
        while (true) {
            if (television.isOn()) {
                light.low();
            } else {
                light.high();
            }
        }
    }
}
```

This solution has some problems

The control system is busy by continuously checking the television

You can add a delay to save a few CPU cycle, but you'll that delay to the response too

The control system is doing just one task

You can loop through a list of televisions of appliances to check, but what happens if one of them does not return?

Asynchronous solution 1

```java
public class Television {

    private HomeAutomation ha;
    private boolean on;

    public boolean isOn() {
        return on;
    }

    public void setHomeAutomation(HomeAutomation ha) {
        this.ha = ha;
    }

    public void turnOn() {
        on = true;
        notify();
    }

    public void turnOff() {
        on = false;
        notify();
    }

    private void notify() {
        ha.update();
    }
}
```

```java
public class HomeAutomation {

    private final Light light;

    public HomeAutomation(Light light, Television tv) {
        this.light = light;
        tv.setHomeAutomation(this);
    }

    public void update() {
        if (television.isOn()) {
            light.low();
        } else {
            light.high();
        }
    }
}
```

This solution has one problem

Television and HomeAutomation are tightly coupled

Asynchronous solution 2

```java
public class Television {

    private TelevisionObserver observer;
    private boolean on;

    public boolean isOn() {
        return on;
    }

    public void attach(TelevisionObserver o) {
        this.observer = o;
    }

    public void turnOn() {
        on = true;
        notify();
    }

    public void turnOff() {
        on = false;
        notify();
    }

    private void notify() {
        observer.update();
    }
}
```

```java
public interface TelevisionObserver {

    void update();
}
```

```java
public class HomeAutomation implements TelevisionObserver {

    private final Light light;

    public HomeAutomation(Light light, Television tv) {
        this.light = light;
        tv.attach(this);
    }

    @Override
    public void update() {
         if (television.isOn()) {
             light.low();
         } else {
             light.high();
         }
     }
}
```

You can register only one observer, and you cannot detach it

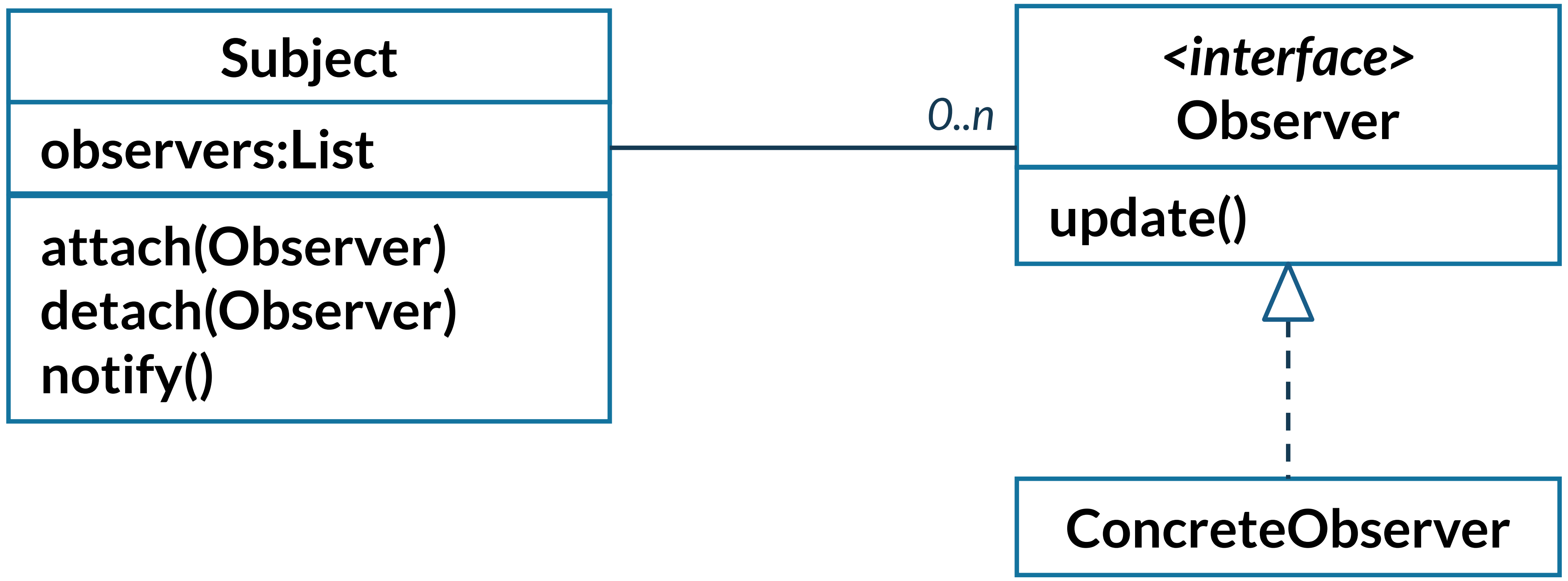
Asynchronous solution 3

```java
public class Television {

    private List<TelevisionObserver> observers;
    private boolean on;

    public boolean isOn() {
        return on;
    }

    public void attach(TelevisionObserver o) {
        observers.add(o);
    }

    public void detach(TelevisionObserver o) {
        observers.remove(o);
    }

    public void turnOn() {
        on = true;
        notify();
    }

    public void turnOff() {
        on = false;
        notify();
    }

    private void notify() {
        observers.forEach(o -> o.update());
    }
}
```

```java
public interface TelevisionObserver {

    void update();
}
```

```java
public class HomeAutomation implements TelevisionObserver {

    private final Light light;

    public HomeAutomation(Light light, Television tv) {
        this.light = light;
        tv.attach(this);
    }

    @Override
    public void update() {
        if (television.isOn()) {
            light.low();
        } else {
            light.high();
        }
    }
}
```

Observer pattern class diagram



Subject

observers:List

attach(Observer)
detach(Observer)
notify()

0..n

<interface>
Observer

update()

ConcreteObserver

A few variations

```
public interface TelevisionObserver {

    void update(Television subject);
}
```

When an observer observes more than one subject could be useful to know the source of the notification

```
@Override
public void update() {
    if (television.isOn()) {
         light.low();
     } else {
         light.high();
     }
}
```

The update method does not bring any information about the new status, you have to pull the subject to know it’s state

```
@Override
public void update(boolean on) {
    if (on) {
         light.low();
     } else {
         light.high();
     }
}
```

In the push model, the changed status is pushed to the observer

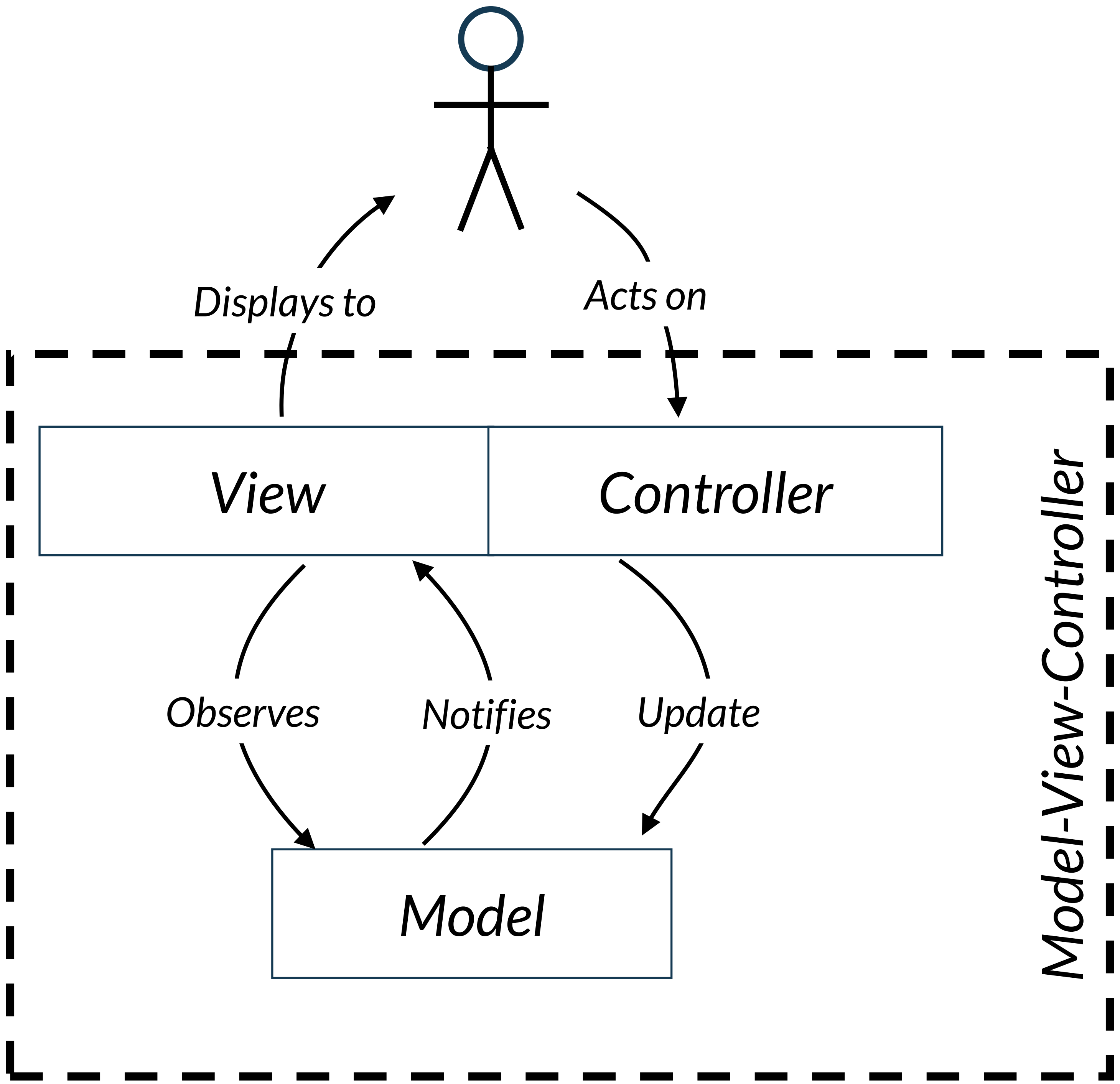
```
@Override
public void update(Context context) {
     …
}
```

When the subject can notify different state changes, you can push more information through a Context object

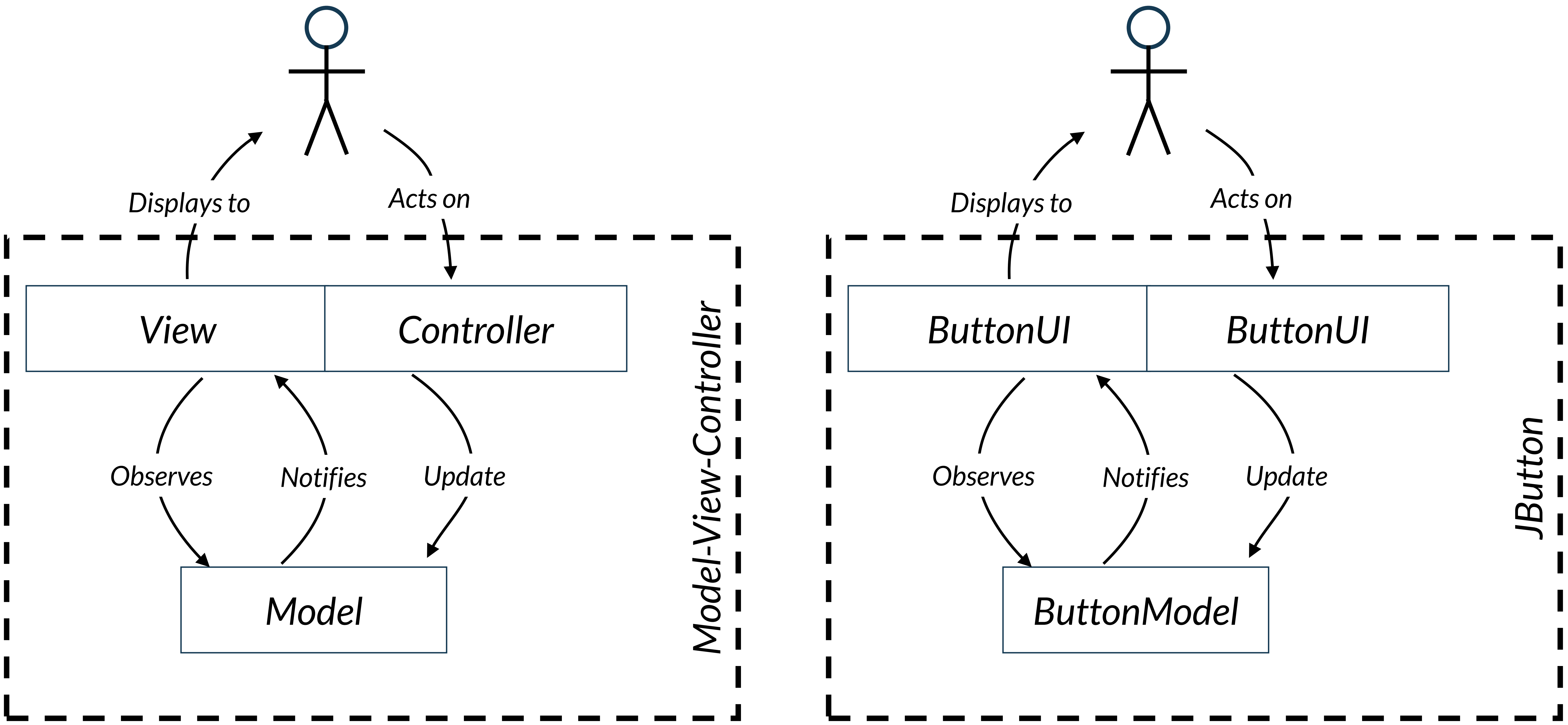
Observer pattern in Swing

- *The observer pattern is ubiquitous in Swing*
- *Observers are called listeners*
- *Status changes are notified as events*
- *Notifications follow the push model and they bring all the relevant information in an Event object*
- *Observer/listener can attach/register themselves to a variety of event*

MVC


Displays to
Acts on
View
Controller
Model-View-Controller
Observes
Notifies
Update
Model

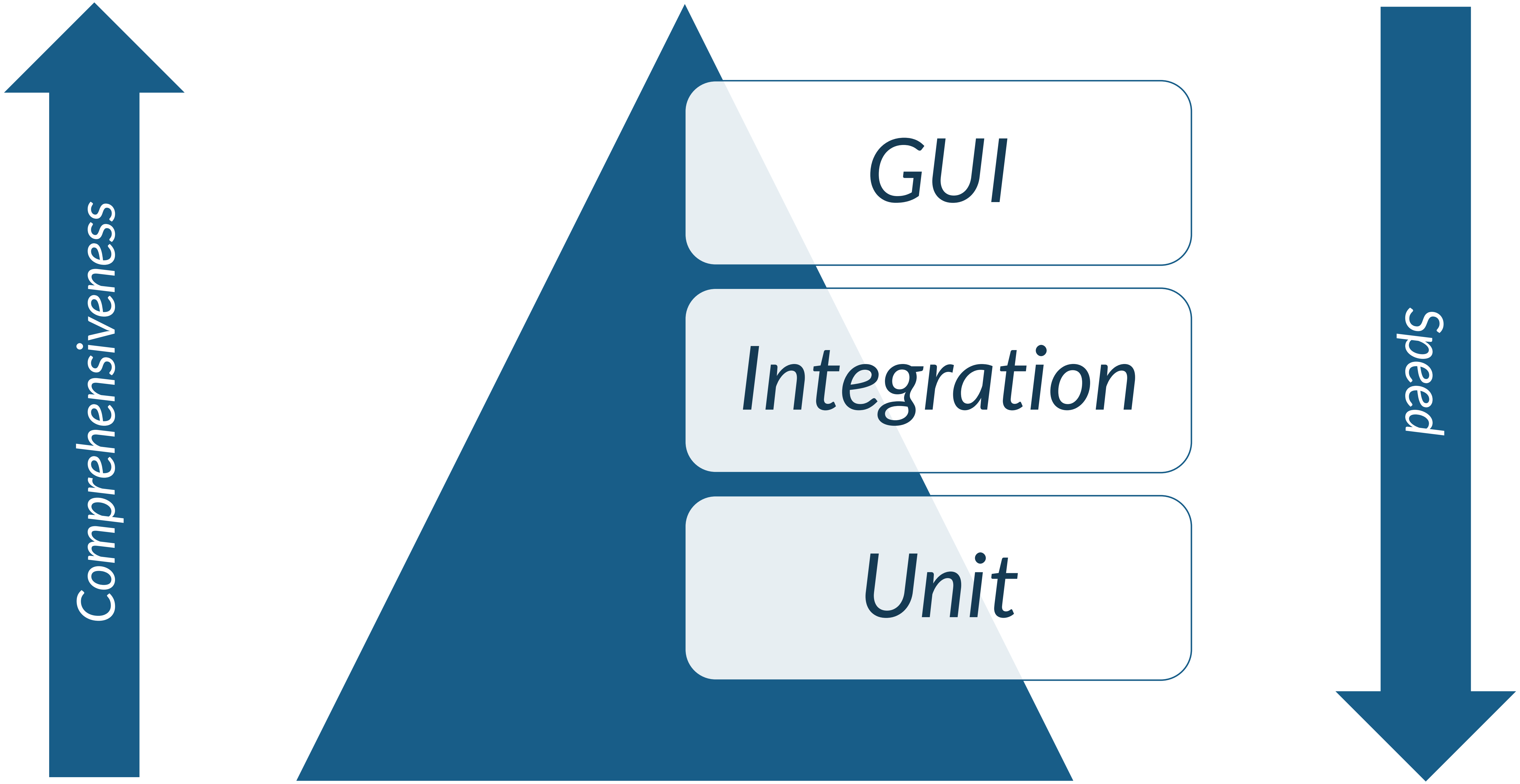
MVC – The JButton case



The humble dialog

Decoupling the view from the application logic

The test pyramid



Automation of GUI code

- *Special frameworks*
- *Complex setup*
- *Slow running*

How to test GUI code

- *GUI code is hard to test automatically and hard to develop by using Test Driven Development*
- *One strategy to make a GUI application more testable is to ensure that the GUI code have the absolute minimum of behavior (code)*
- *For example, through the implementation of the Humble Object pattern http://xunitpatterns.com/Humble%20Object.html*

Humble Object pattern

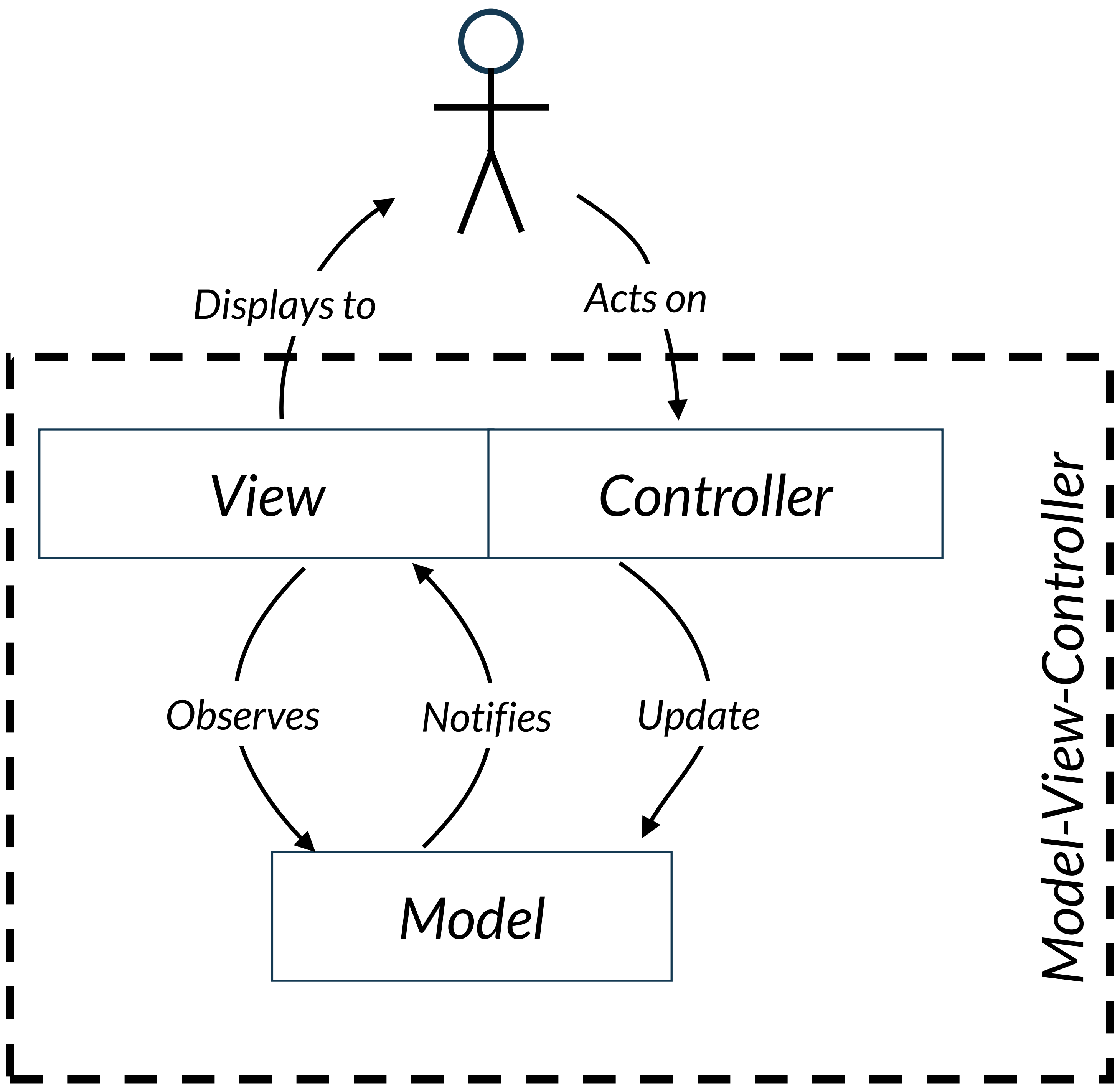
This pattern is applied at the boundaries of the system, where things are often difficult to test, in order to make them more testable. We accomplish the pattern by reducing the logic close to the boundary, making the code close to the boundary so humble that it doesn't need to be tested. The extracted logic is moved into another class, decoupled from the boundary which makes it testable.

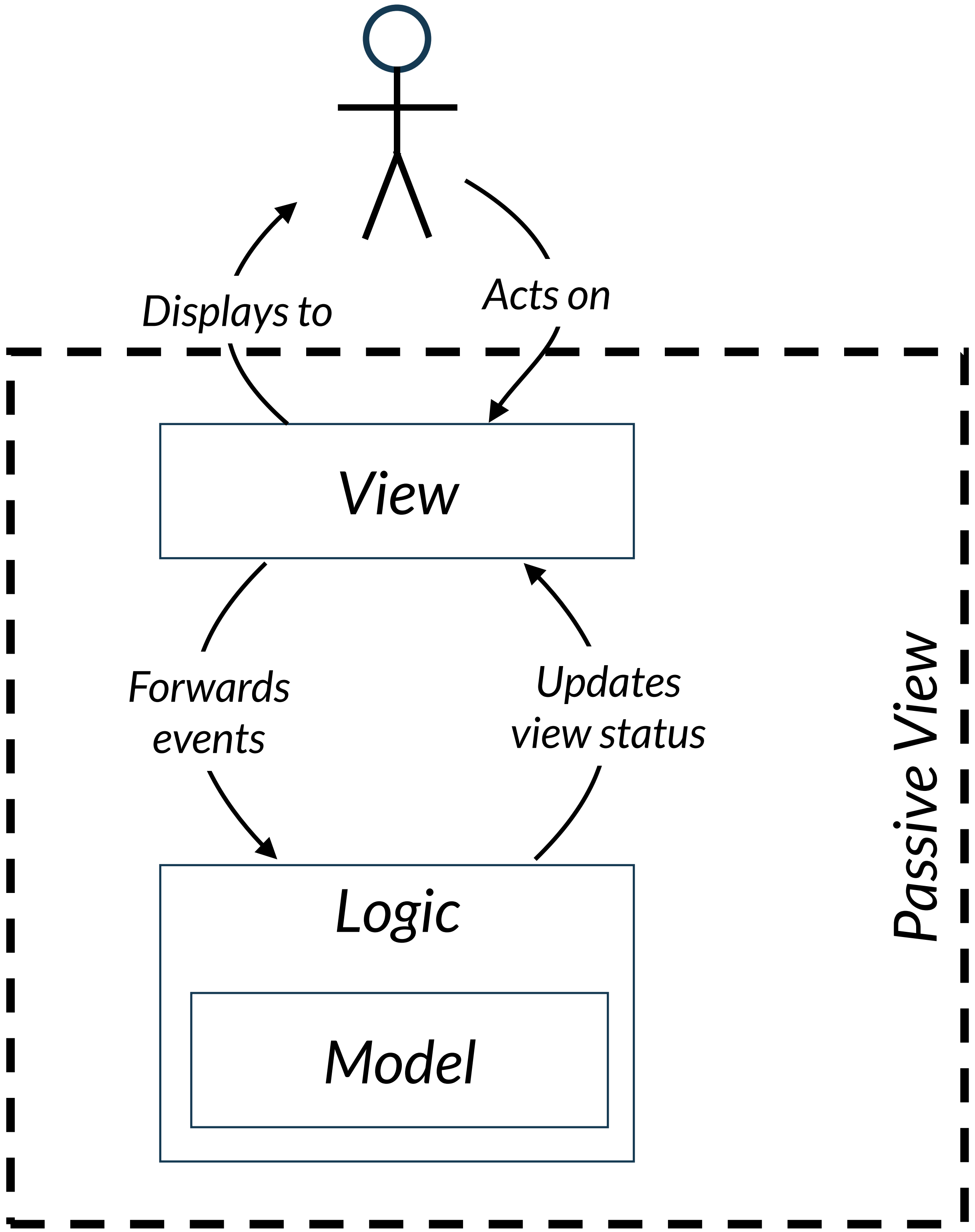
- Robert C. Martin

The Humble Object pattern in GUI programming

1. *Passive View (variation of the MVC pattern)* *https://stefanoborini.com/book-modelviewcontroller/02-mvc-variations/02-variations-on-the-view/02-passive-view.html*
2. *Humble dialog pattern* *https://martinfowler.com/articles/humble-dialog-box.html*

Model View Controller vs Passive View





The Humble Dialog

1. *Create a class for the smart object, and an interface class for the view. Pass the view to the smart object*
2. *Develop commands against the smart object, test first. Write your tests against a mock view.*
3. *Create your dialog class and implement the view interface on it. Gestures on the dialog should delegate to commands on the smart object. Calls from the smart object to the dialog should resolve to simple setter methods.*

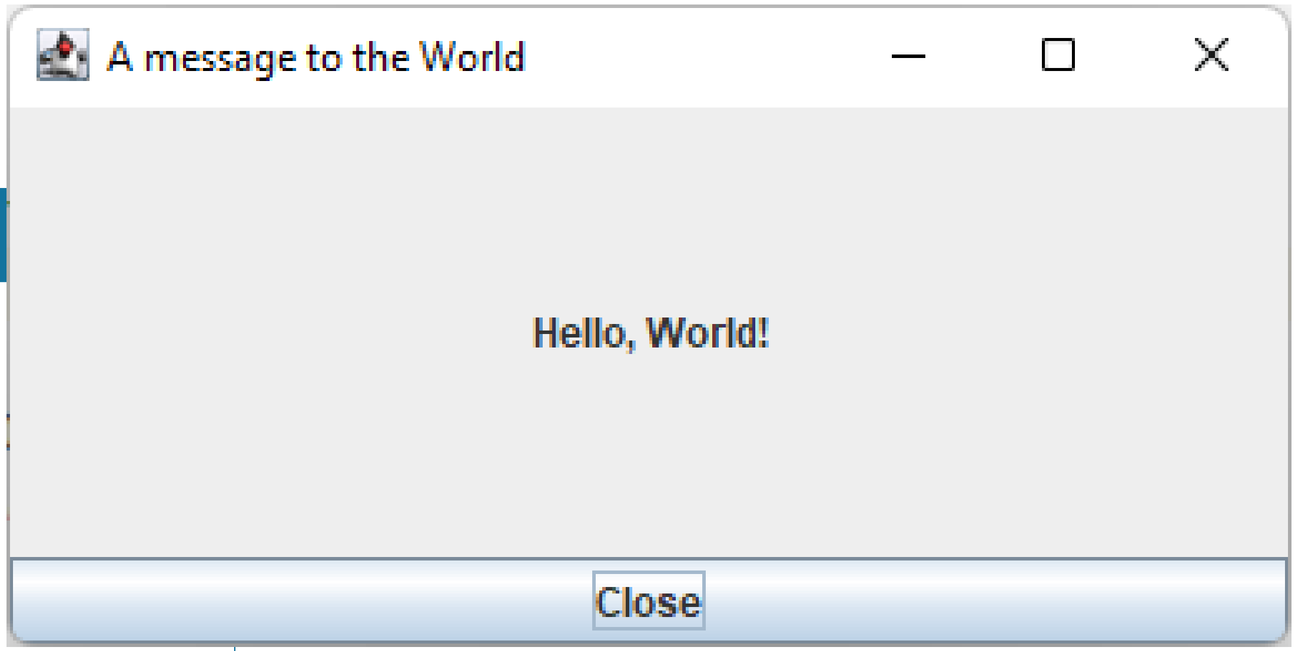
When you follow these steps, you end up with tested code and a great interface for driving acceptance tests programmatically.

- Michael Feathers, The Humble Dialog Box

Humble Dialog example

HelloWorld.java

```java
public class HelloWorld {

    public static void main(String[] args) {
        SwingUtilities.invokeLater(HelloWorld::helloWorld);
    }

    private static void helloWorld() {
        JFrame frame = new JFrame("A message to the World");
        frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

        JLabel label = new JLabel("Hello, World!");
        label.setHorizontalAlignment(SwingConstants.CENTER);
        frame.getContentPane().add(label, BorderLayout.CENTER);

        JButton closeButton = new JButton("Close");
        closeButton.addActionListener(x -> frame.dispose());
        frame.getContentPane().add(closeButton, BorderLayout.SOUTH);

        frame.setSize(400, 200);
        frame.setVisible(true);
    }
}
```



A message to the World

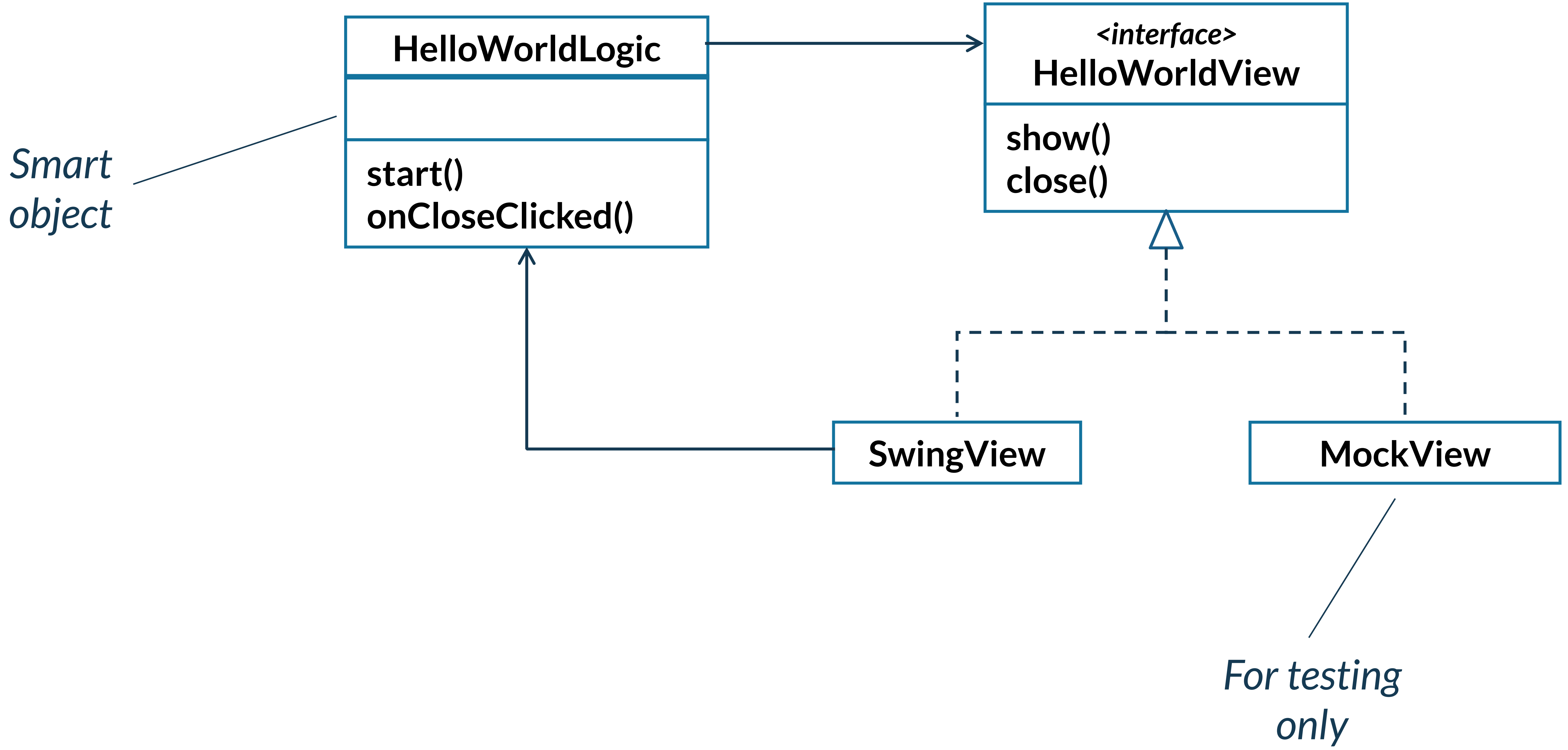
Hello, World!

What is the logic in this class?

What shall we test?

We want to test that when we click on the Close button the window is disposed

In practice


HelloWorldLogic
start()
onCloseClicked()
<interface>
HelloWorldView
show()
close()
SwingView
MockView
Smart object
For testing only

HelloWorld Logic & View

HelloWorldLogic.java

```java
public class HelloWorldLogic {

    private final HelloWorldView view;

    public HelloWorldLogic(HelloWorldView view) {
        this.view = view;
    }

    public void start() {
        view.show();
    }

    public void onCloseClick() {
        view.close();
    }

    public static void main(String[] args) {
        SwingHelloWorld view = new SwingHelloWorld();
        HelloWorldLogic logic = new HelloWorldLogic(view);
        view.installLogic(logic);
        logic.start();
    }
}
```

HelloWorldView.java

```java
public interface HelloWorldView {

    void close();

    void show();
}
```

Swing implementation

SwingHelloWorld.java

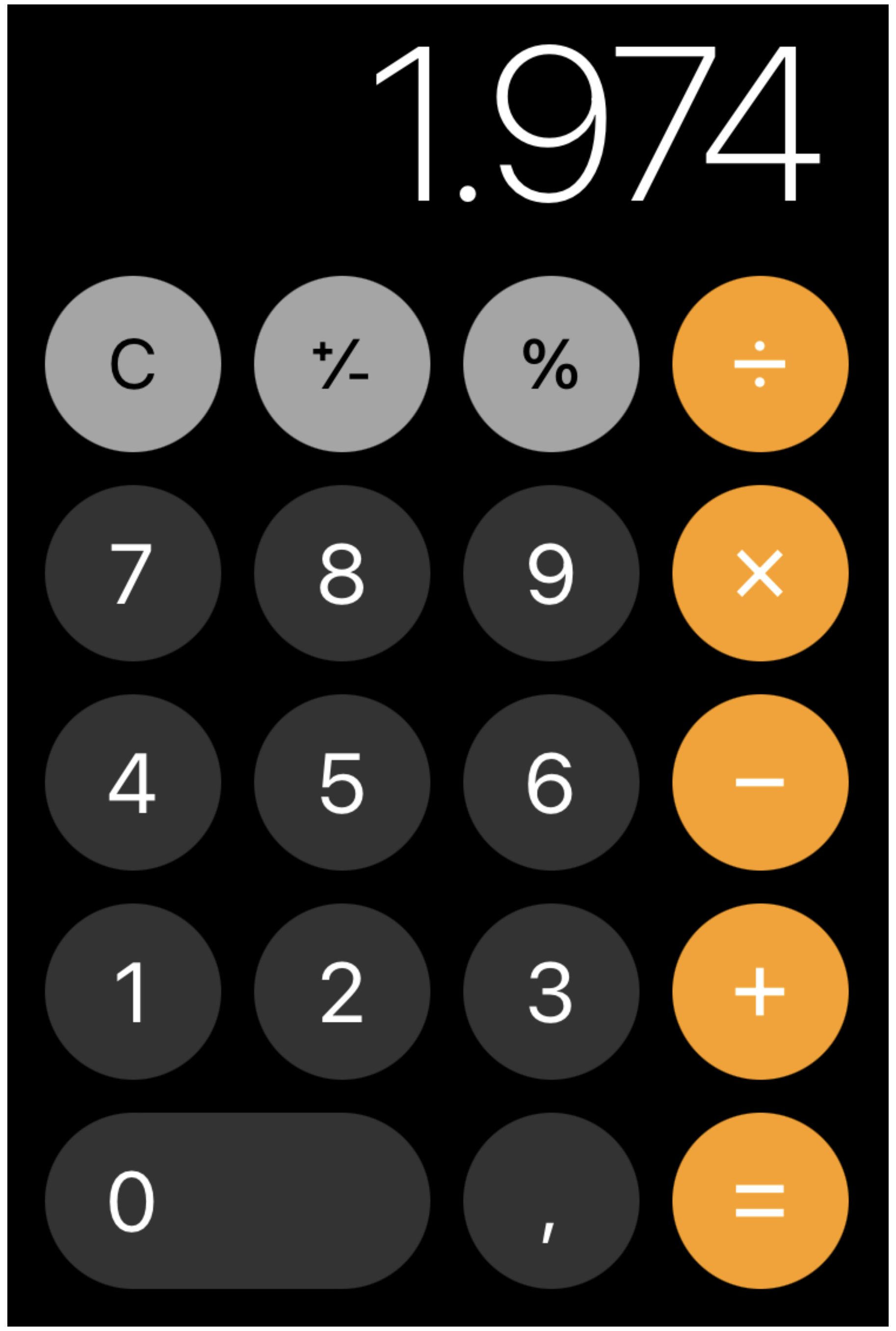
```java
public class SwingHelloWorld implements HelloWorldView {

    private JFrame frame;
    private HelloWorldLogic logic;

    public void installLogic(HelloWorldLogic logic) {
        this.logic = logic;
    }

    private void buildAndShow() {
        frame = new JFrame("A message to the World");
        frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        JLabel label = new JLabel("Hello, World!");
        label.setHorizontalAlignment(SwingConstants.CENTER);
        frame.getContentPane().add(label, BorderLayout.CENTER);
        JButton closeButton = new JButton("Close");
        closeButton.addActionListener(x -> logic.onCloseClicked());
        frame.getContentPane().add(closeButton, BorderLayout.SOUTH);
        frame.setSize(400, 200);
        frame.setVisible(true);
    }

    @Override
    public void show() {
       SwingUtilities.invokeLater(this::buildAndShow);
    }

    @Override
    public void closeWindow() {
        SwingUtilities.invokeLater(frame::dispose);
    }
}
```

“Trivial” unit testing of HelloWorldLogic

HelloWorldLogicTest.java

```java
@Test
void logicShowsTheView() {
    HelloWorldViewSpy view = new HelloWorldViewSpy();
    new HelloWorldLogic(view).start();
    assertTrue(view.shown);
}

@Test
void logicClosesTheView() {
    HelloWorldViewSpy view = new HelloWorldViewSpy();
    new HelloWorldLogic(view).onCloseClicked();
    assertTrue(view.closed);
}

private static class HelloWorldViewSpy implements HelloWorldView {

    private boolean shown;
    private boolean closed;

    @Override
    public void closeWindow() {
        closed = true;
    }

    @Override
    public void show() {
        shown = true;
    }
}
```

Assignment 3



Define a calculator class that

1. *receives “events” from a calculator keyboard*
2. *sends the output to a Display object*

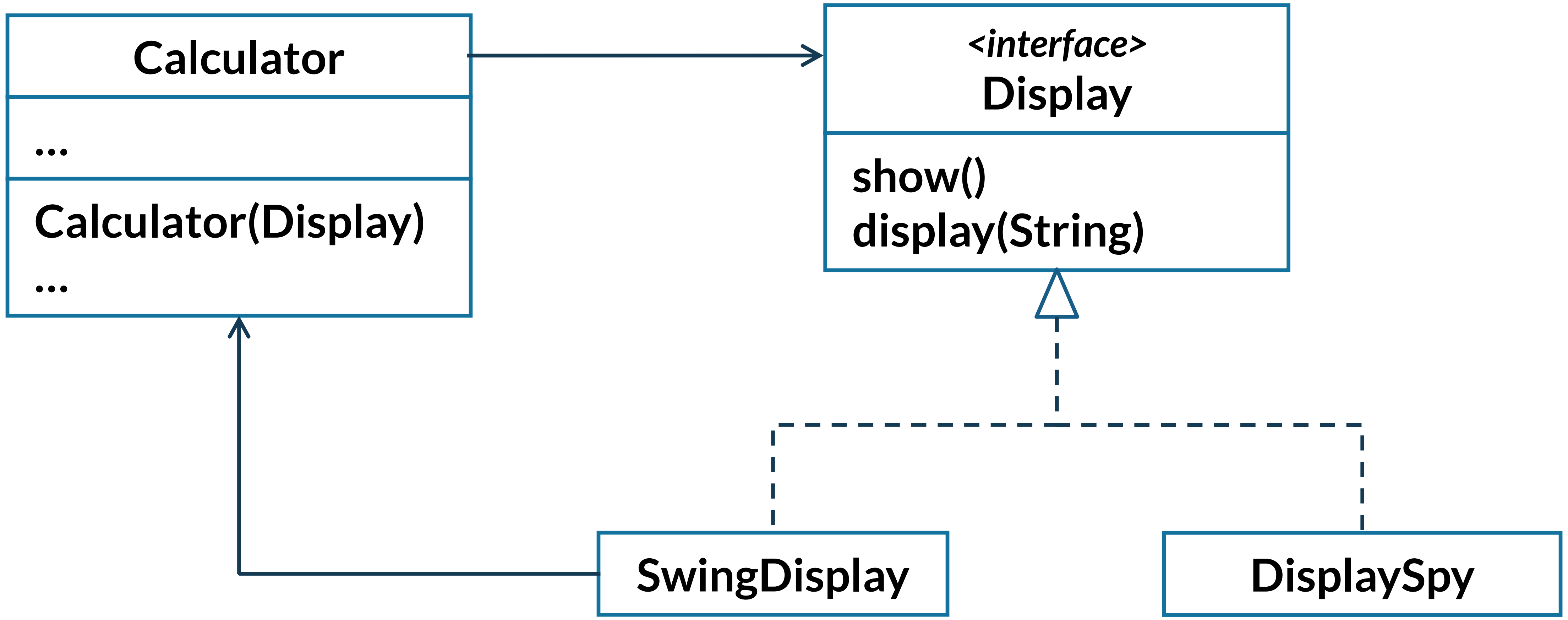
```
class Display {
    void display(String text) {
        System.out.println(text);
    }
}
```

```
class Calculator {

    final Display display;
    //...

    Calculator(Display display) {
        this.display = display;
    }

    void plusPressed() {
        //...
    }

    void zeroPressed() {
        //...
    }

    //...
}
```

Create a class for the smart object and an interface for the view

```
class Calculator {

    private final Display display;

    Calculator(Display display) {
        this.display = display;
    }
    …
}
```

```
interface Display {

    void show();

    void display(String text);
}
```



Sample unit testing of Calculator

CalculatorTest.java

```java
@Test
void start() {
    DisplaySpy display = new DisplaySpy();
    Calculator calculator = new Calculator(display);
    calculator.start();

    assertTrue(display.shown);
    assertEquals("0", display.displayed);
}
…
@Test
void division() {
    DisplaySpy display = new DisplaySpy();
    Calculator calculator = new Calculator(display);
    calculator.start();
    calculator.twoPressed();
    calculator.dividePressed();
    calculator.threePressed();
    calculator.equalPressed();

    assertEquals("0.6666666666666666", display.displayed);
}

static class DisplaySpy implements Display {

    String displayed;
    boolean shown;

    @Override
    public void show() {
        shown = true;
    }

    @Override
    public void display(String text) {
        displayed = text;
    }
}
```

Practical tips

- *The view interface should contain only methods to set the state of the view*
- *Swing components implement the MVC pattern on their own and they update their state by their own, we don’t need to test those implementation of MVC*
- *Try to avoid state duplication between the Swing components and the logic (not always easy)*

References

Stefano Borini, Understanding Model-View-Controller
https://stefanoborini.com/book-modelviewcontroller/

Michael Feathers, The Humble Dialog Box
https://martinfowler.com/articles/images/humble-dialog-box/TheHumbleDialogBox.pdf

Take aways

- ❑ *GUI applications are usually hard-to-test and they require special tools and setup*
- ❑ *We should move as much logic as possible out of the hard-to-test element into other more test-friendly parts of the code base, by applying the Humble Object pattern*
- ❑ *In GUI applications the Humble Object pattern takes the form of the Humble Dialog that implements the Passive View, a Model-View-Controller architectural pattern in which the View is completely passive and does not update its state from the Model*

Thank you!

esteco.com