Programming in Java – Basics of Swing

Paolo Vercesi

Technical Program Manager

Agenda

Hello, World!

The rules of the game

Working with Swing components

The humble dialog
Decoupling the view from the application logic

Hello, World!

Why are we still teaching GUI and desktop programming in 2022?

Because there are still many desktop applications around (even if they are less and less used)

Because the same concepts and the same knowledge can be applied in the development of apps for smartphones and tablet

Furthermore, will see that if we decouple the GUI, the presentation layer, from the application logic, the GUI becomes just a front-end of our application logic and it will be easy to switch from one front-end to another

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 an application with a GUI, 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

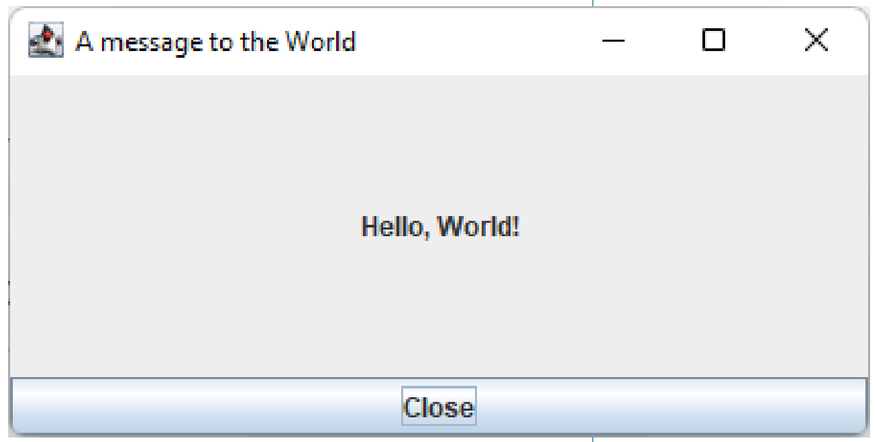
How to learn Java Swing

- Official tutorial https://docs.oracle.com/javase/tutorial/uiswing/index.html
  - be aware that it is based on Java 8 and some technologies such as Applets and Web Start have been deprecated or removed in the next releases of Java
  - and do a lot of experiments
- Study the Java documentation
  - and do a lot of experiments
- Look at the source code
  - and do a lot of experiments
- Ask a colleague
  - and do a lot of experiments
- Attend this introduction to Java Swing
  - and…

Hello, World!

HelloWorld.java

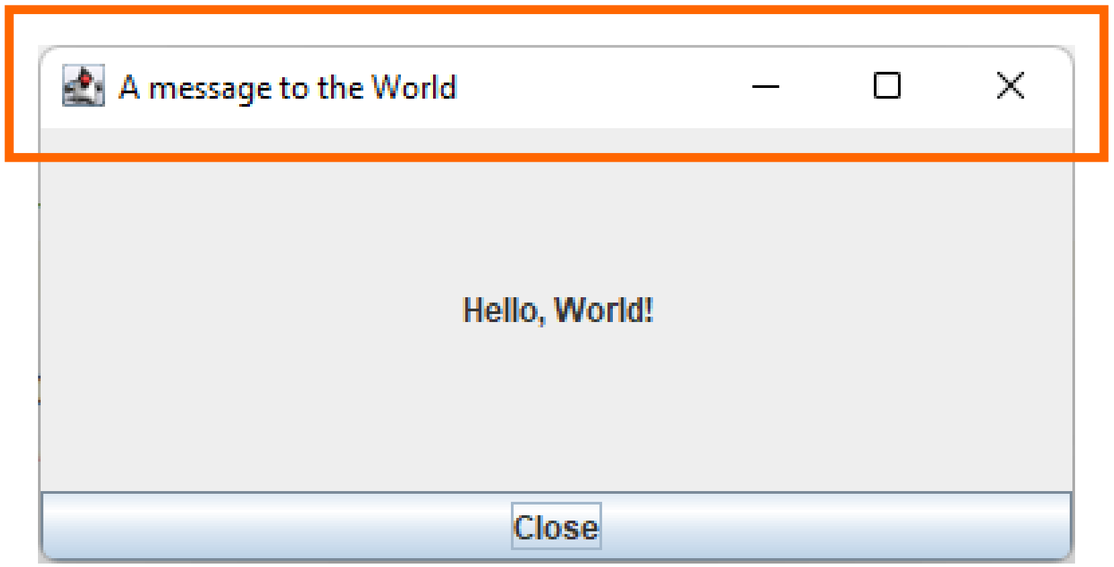
```
public class HelloWorld {

    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);
    }
}
```



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

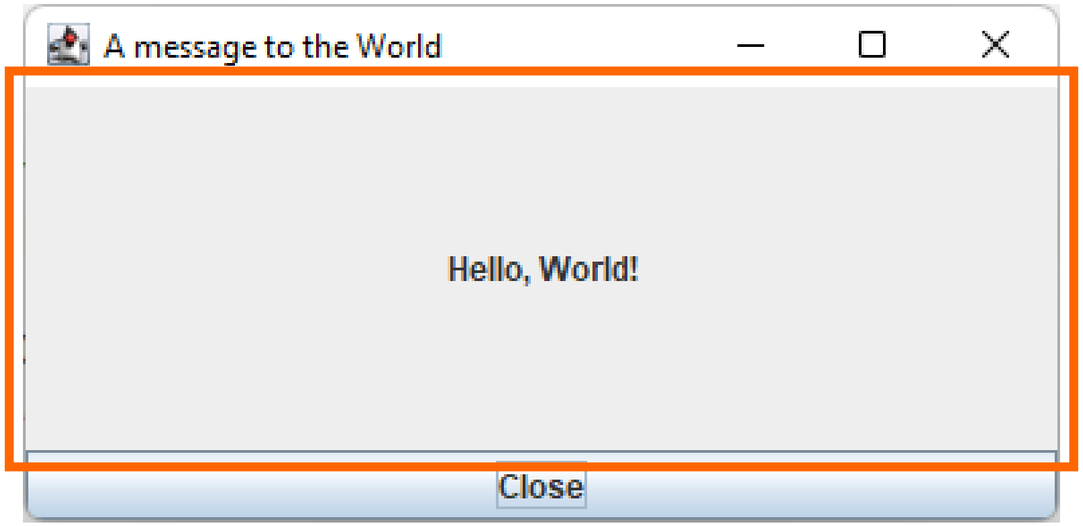
The behavior of the close button can be customized, for example to dispose the JFrame

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

```
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

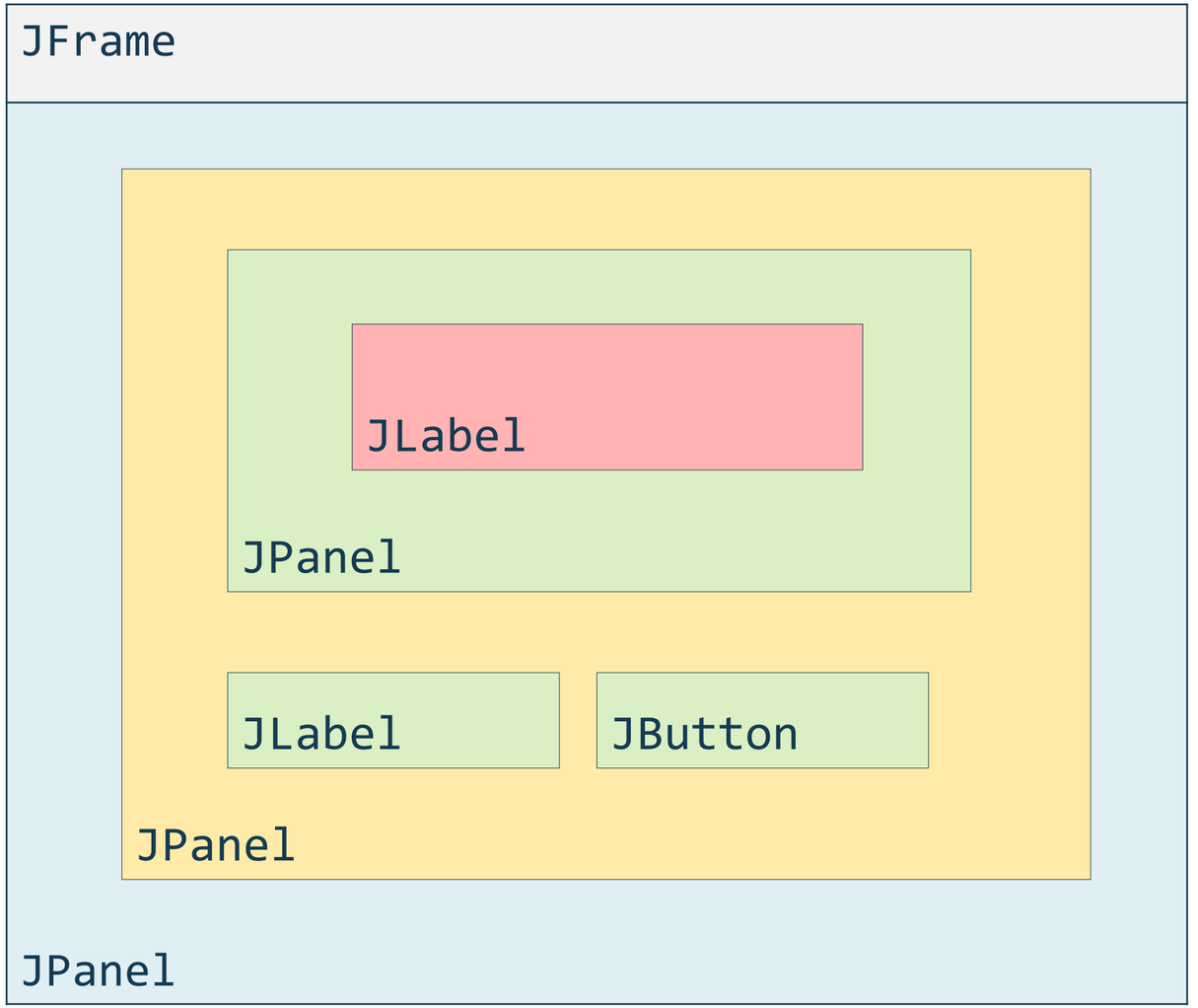
Containment hierarchy

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

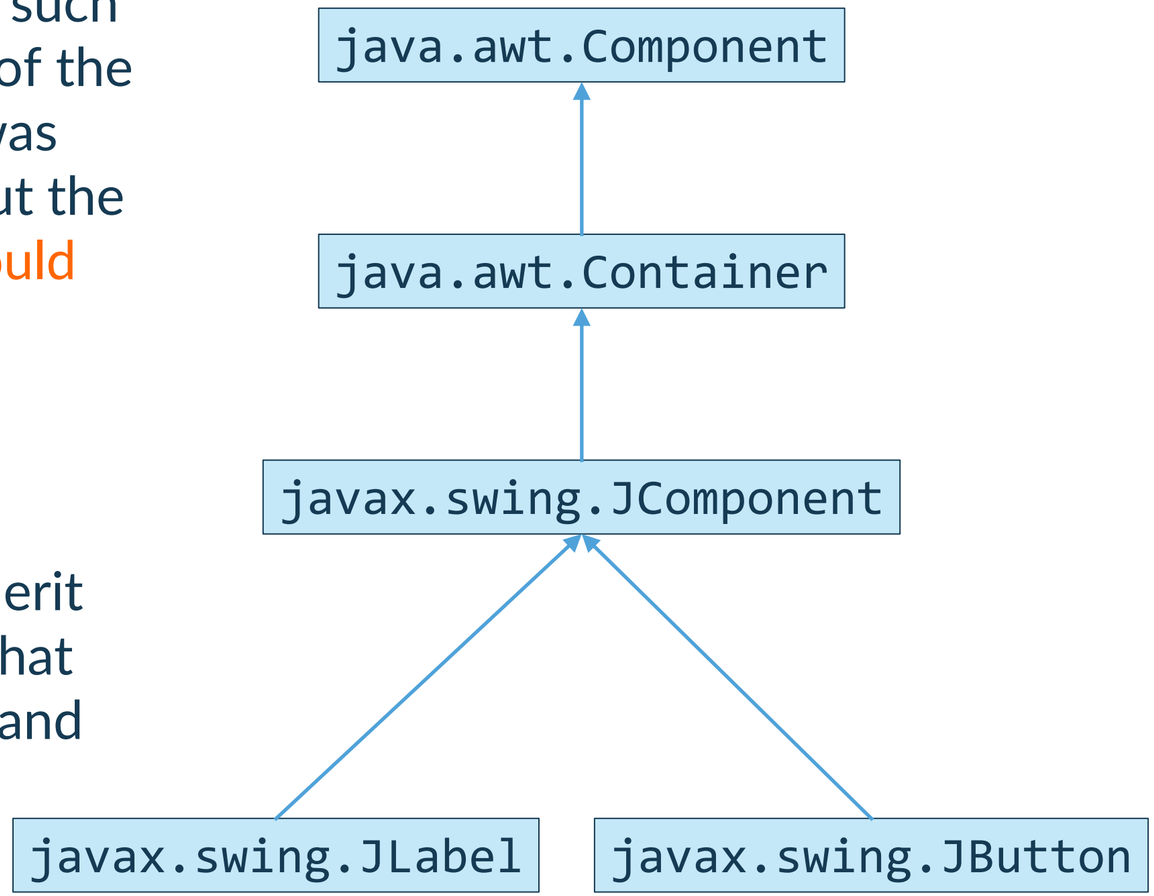
A GUI application usually visualizes just one JFrame instance

- 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

More on 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.

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

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

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

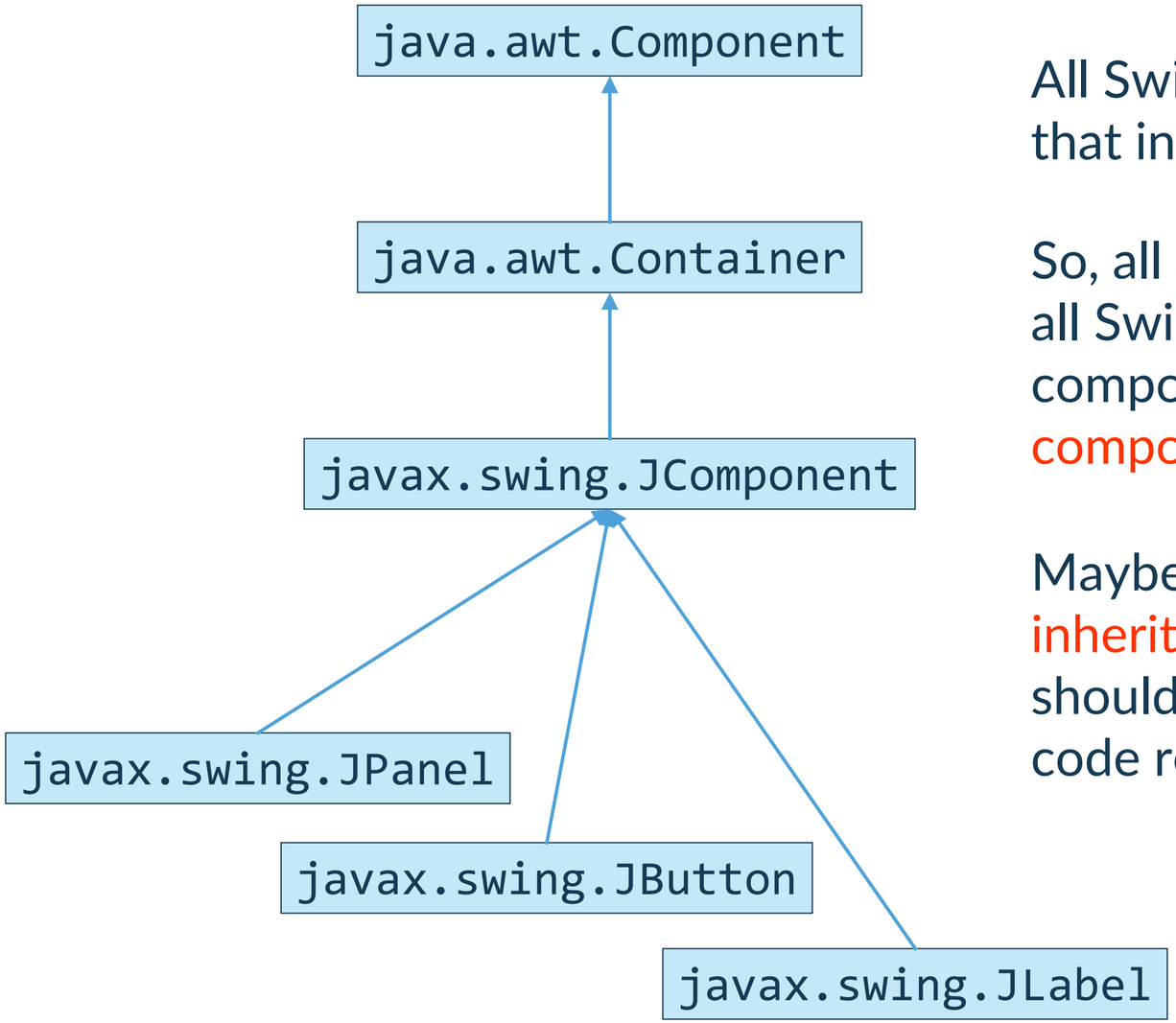
This program terminates

This program doesn't terminate

HiddenFrame.java

```
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);
    }
}
```

Inheritance hierarchy

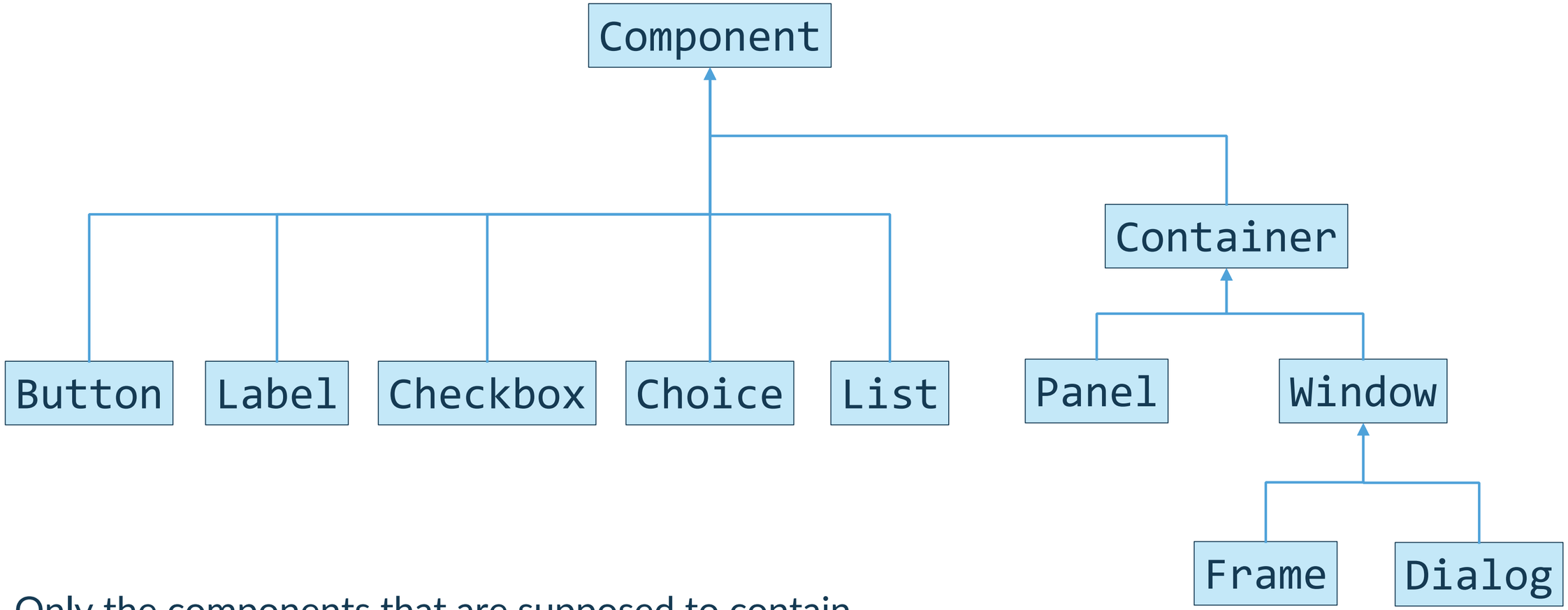


All Swing components, inherits from JComponent that in turn inherits from Container

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

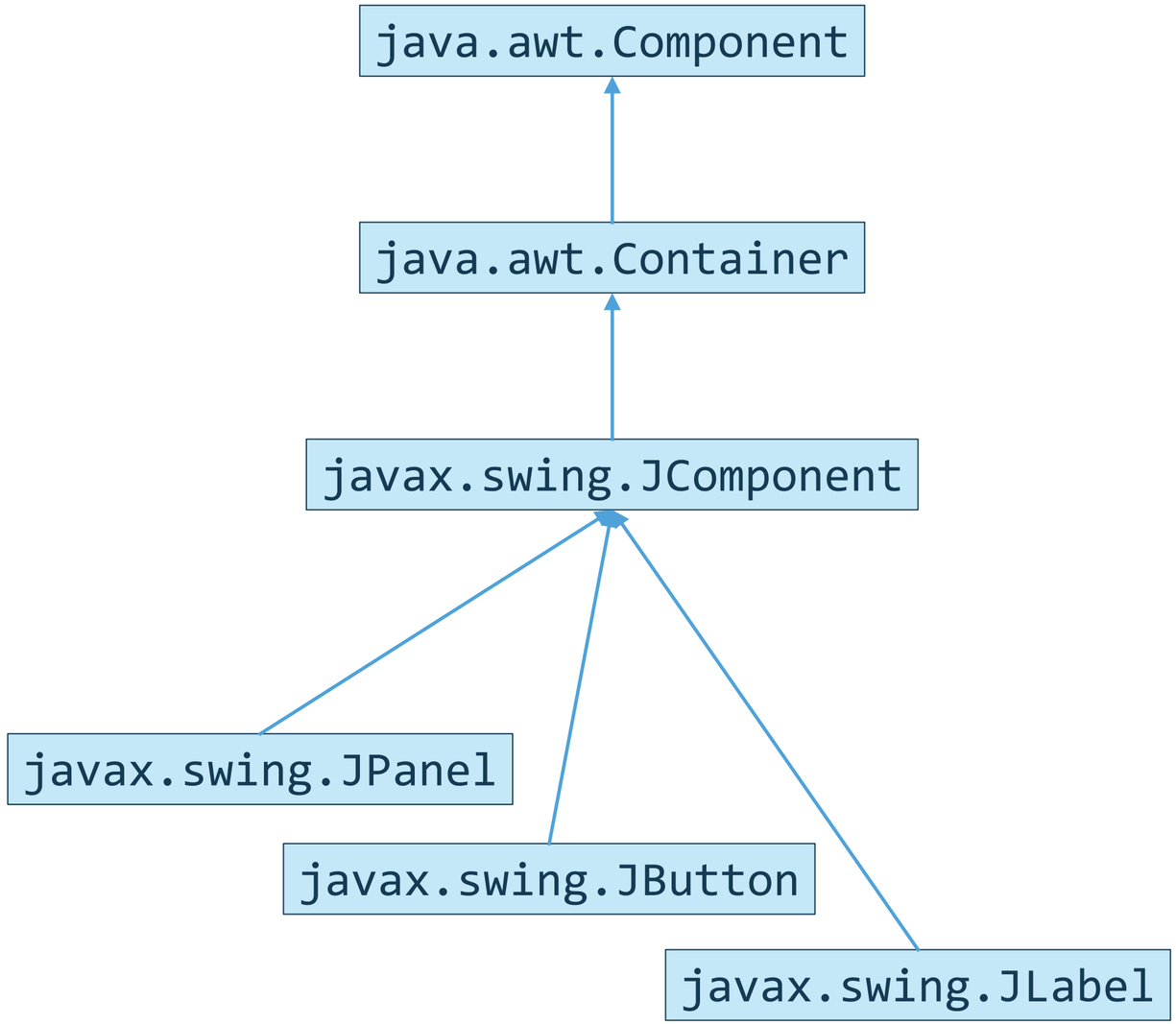
Maybe this is not a very appropriate use of inheritance, but sometimes software engineers should accept trade-offs, in this case they traded code reuse with a “misuse” of inheritance

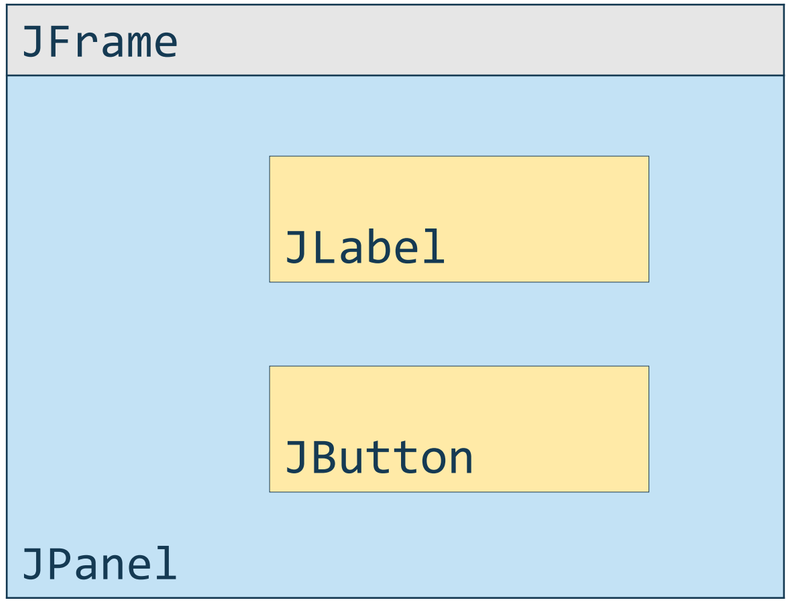
Digression - AWT inheritance hierarchy



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

Inheritance vs containment hierarchy





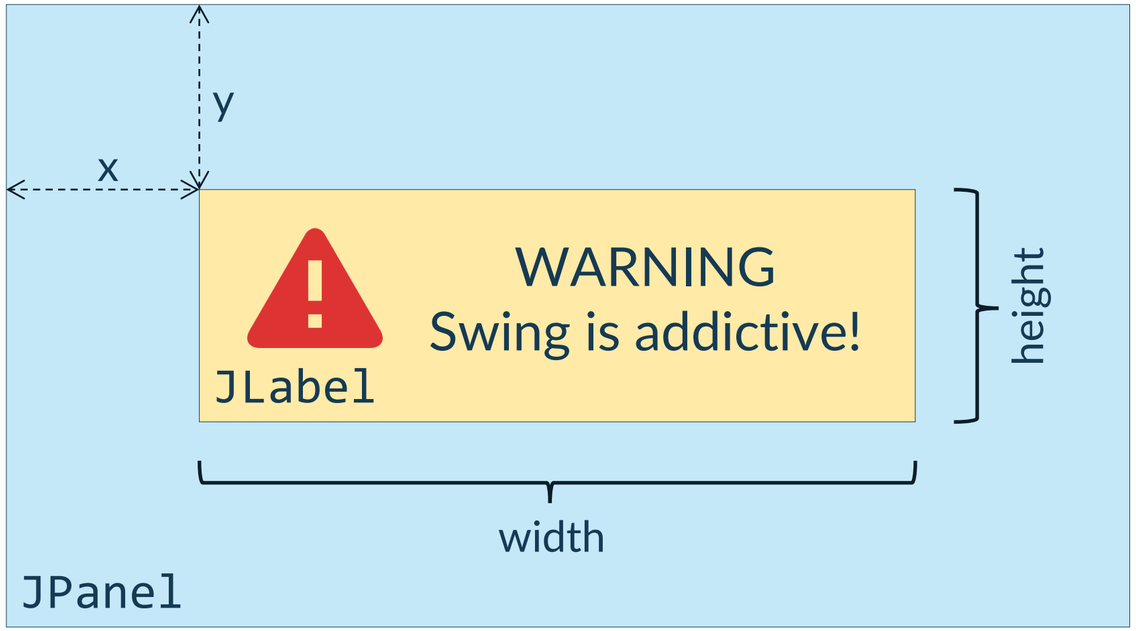
Label and button are child components of a panel

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

“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

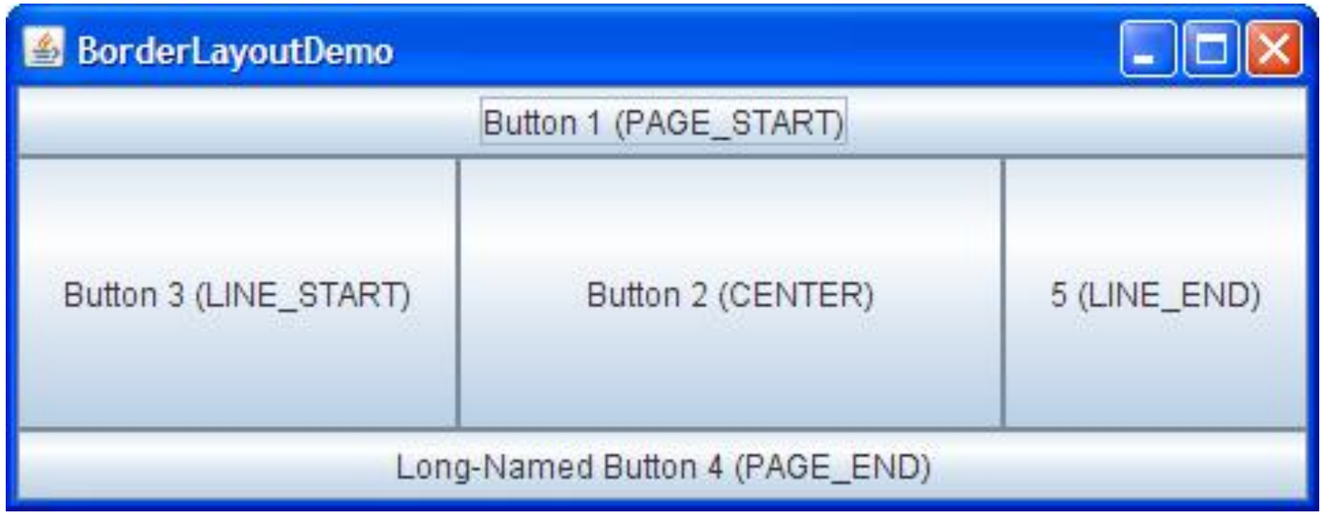
1. layout the child components given their preferences and eventually a set of constraints
2. calculate the container preferred, minimum, and maximum sizes

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

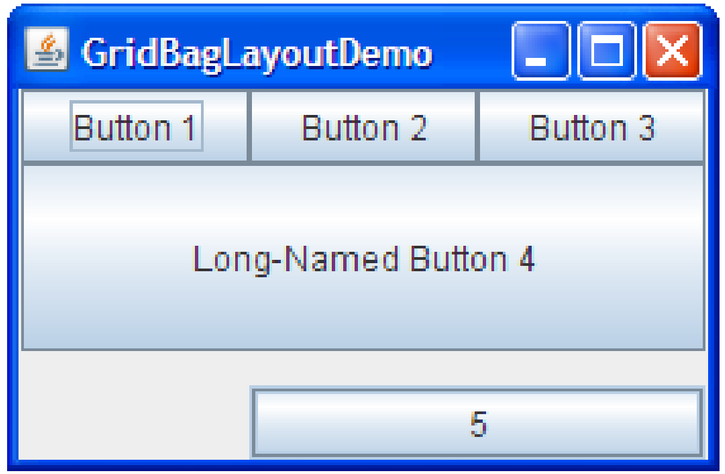
Layout managers

Common (my favorites) layout managers

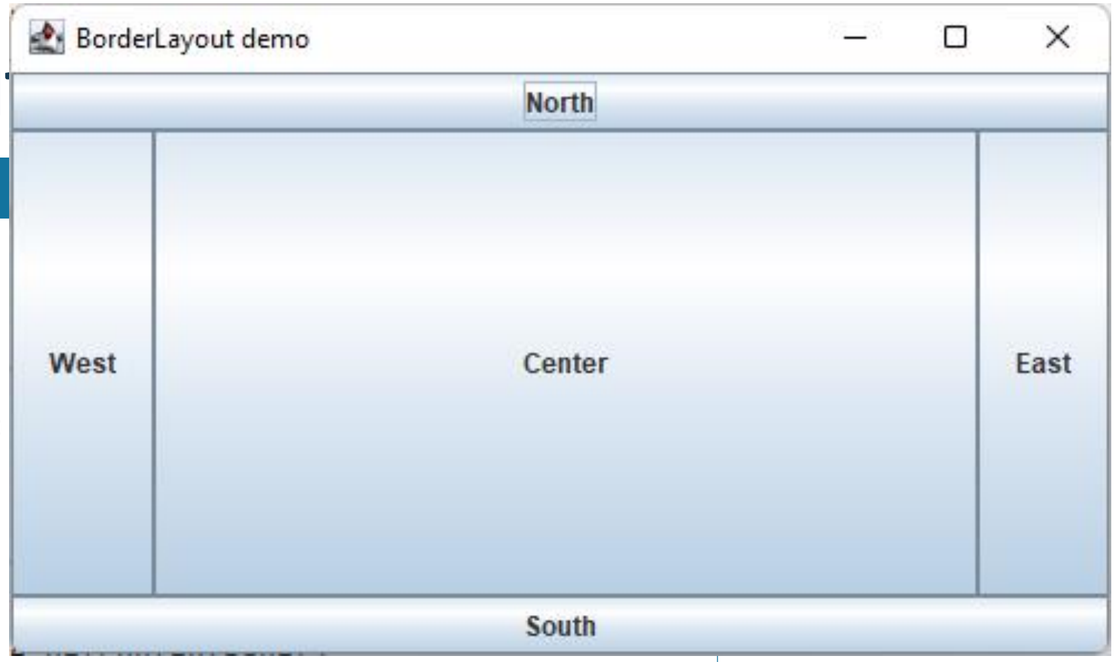
BorderLayout



GridBagLayout



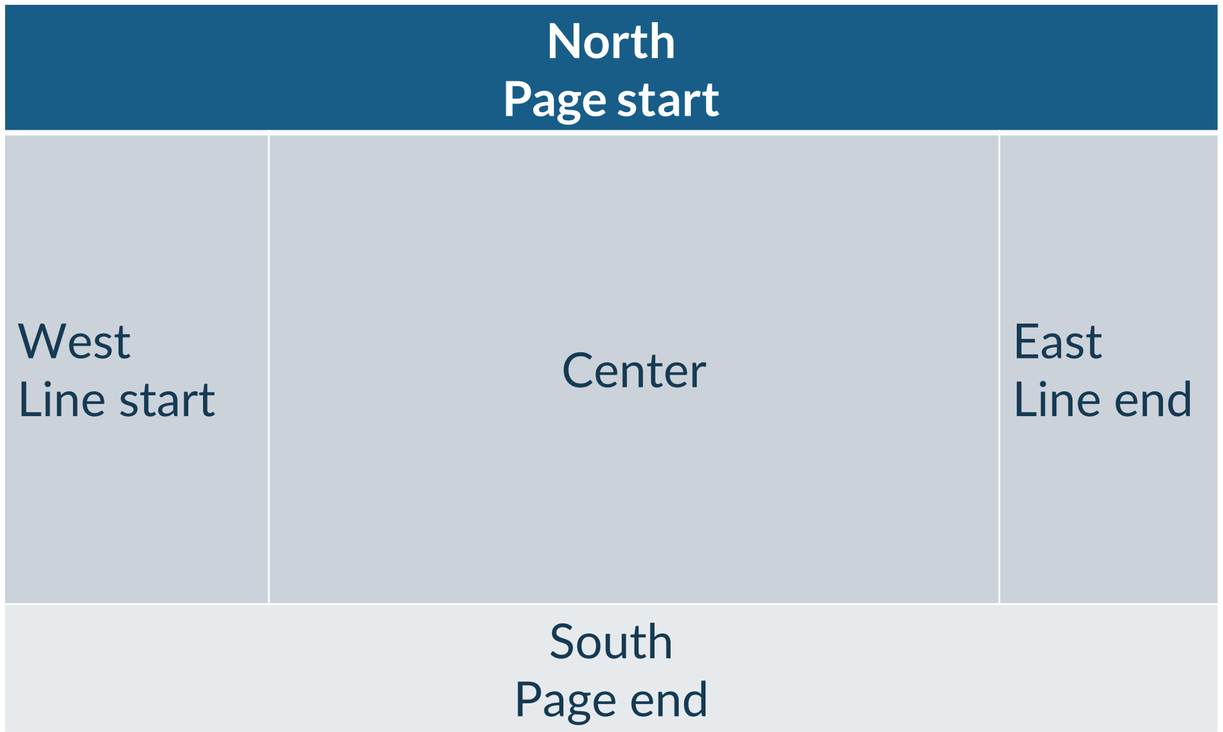
BorderLayou



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



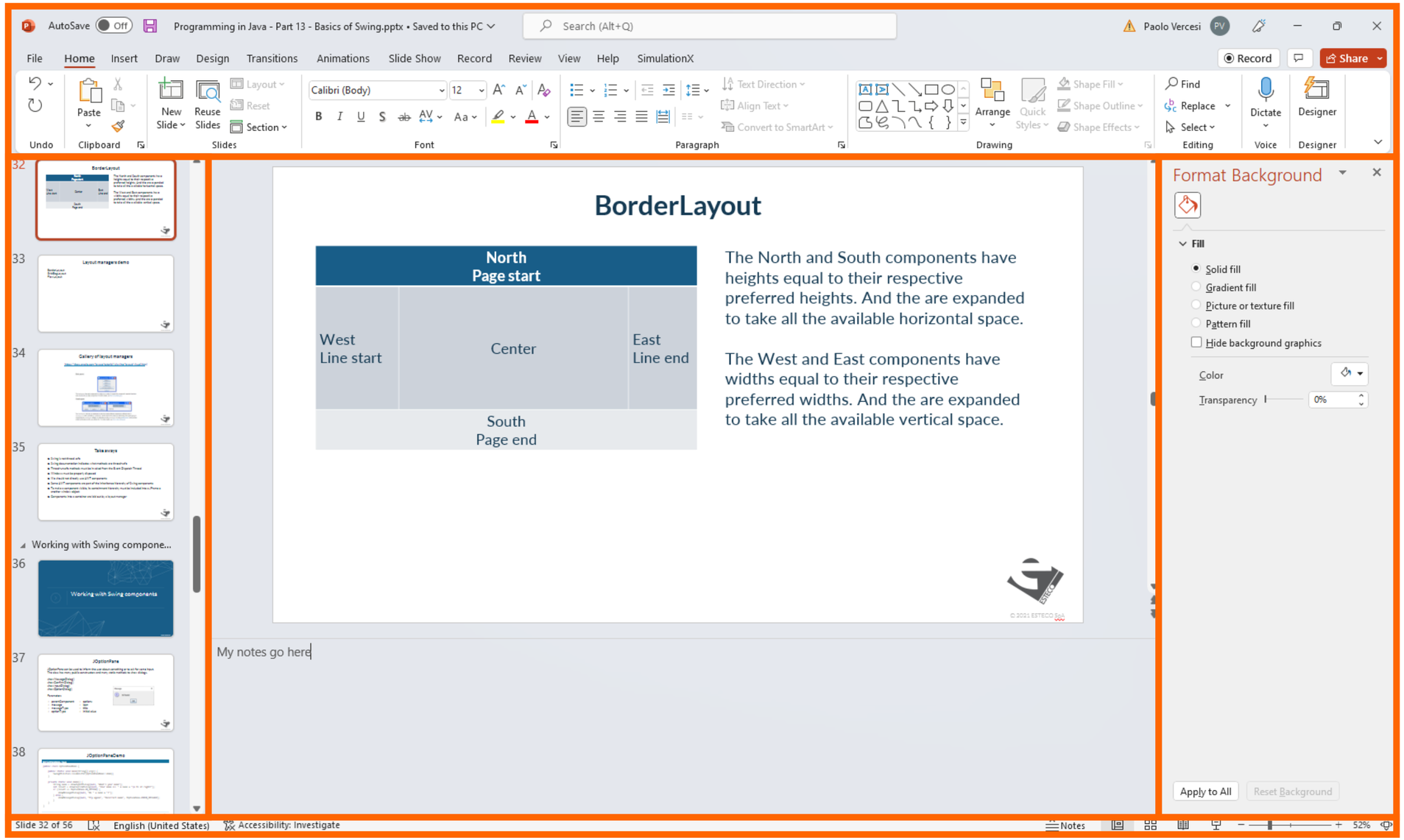
The maximum number of components is 5

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

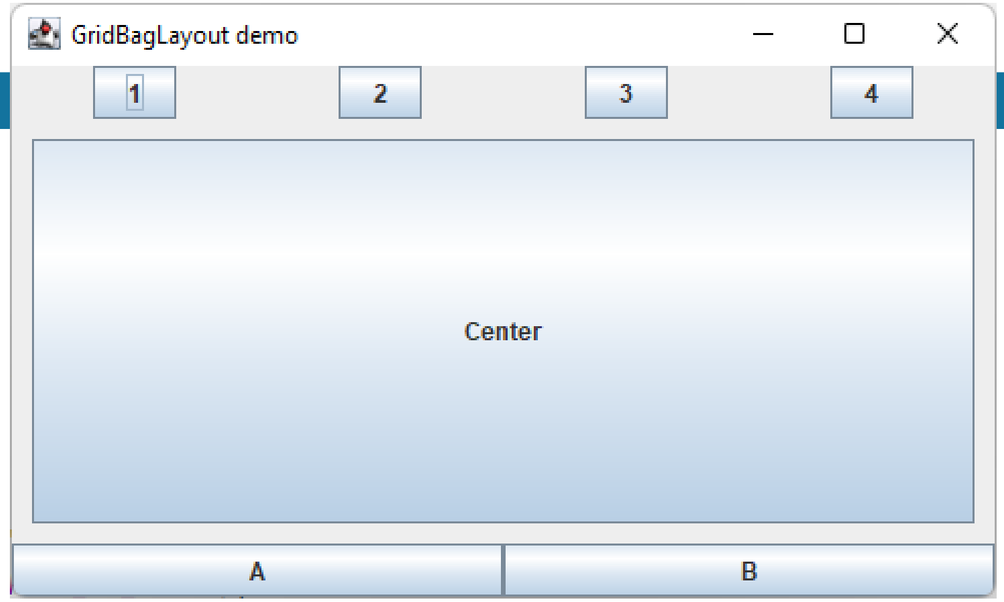
When using the BorderLayout

- 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!



GridBagLayout demo

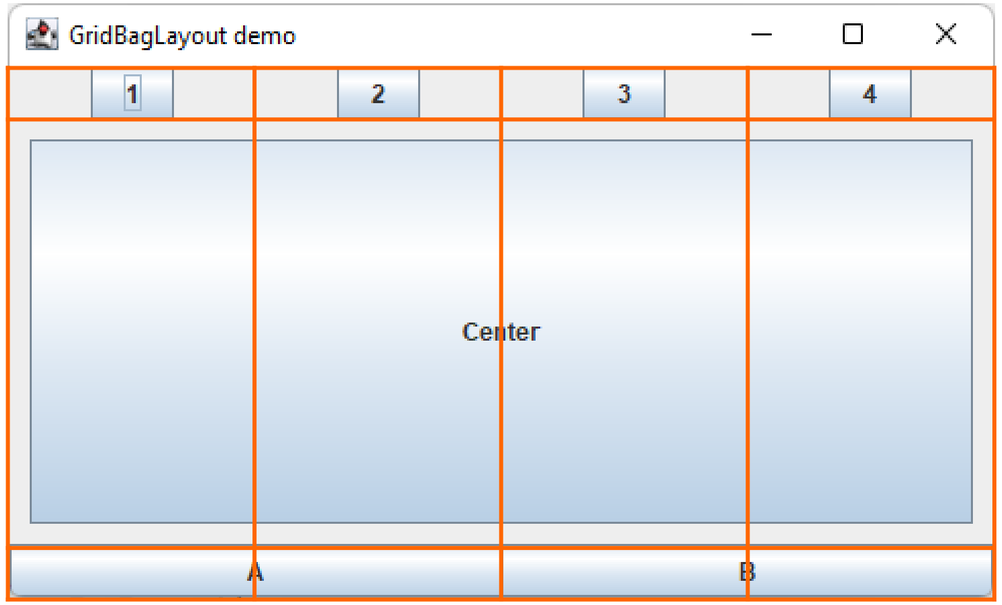


GridBagLayoutDemo.java

```
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

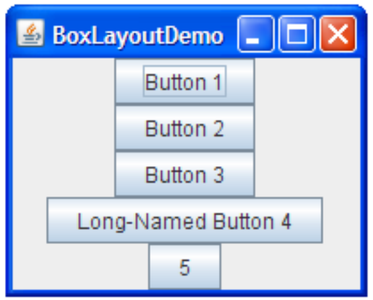
Assignment

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.

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

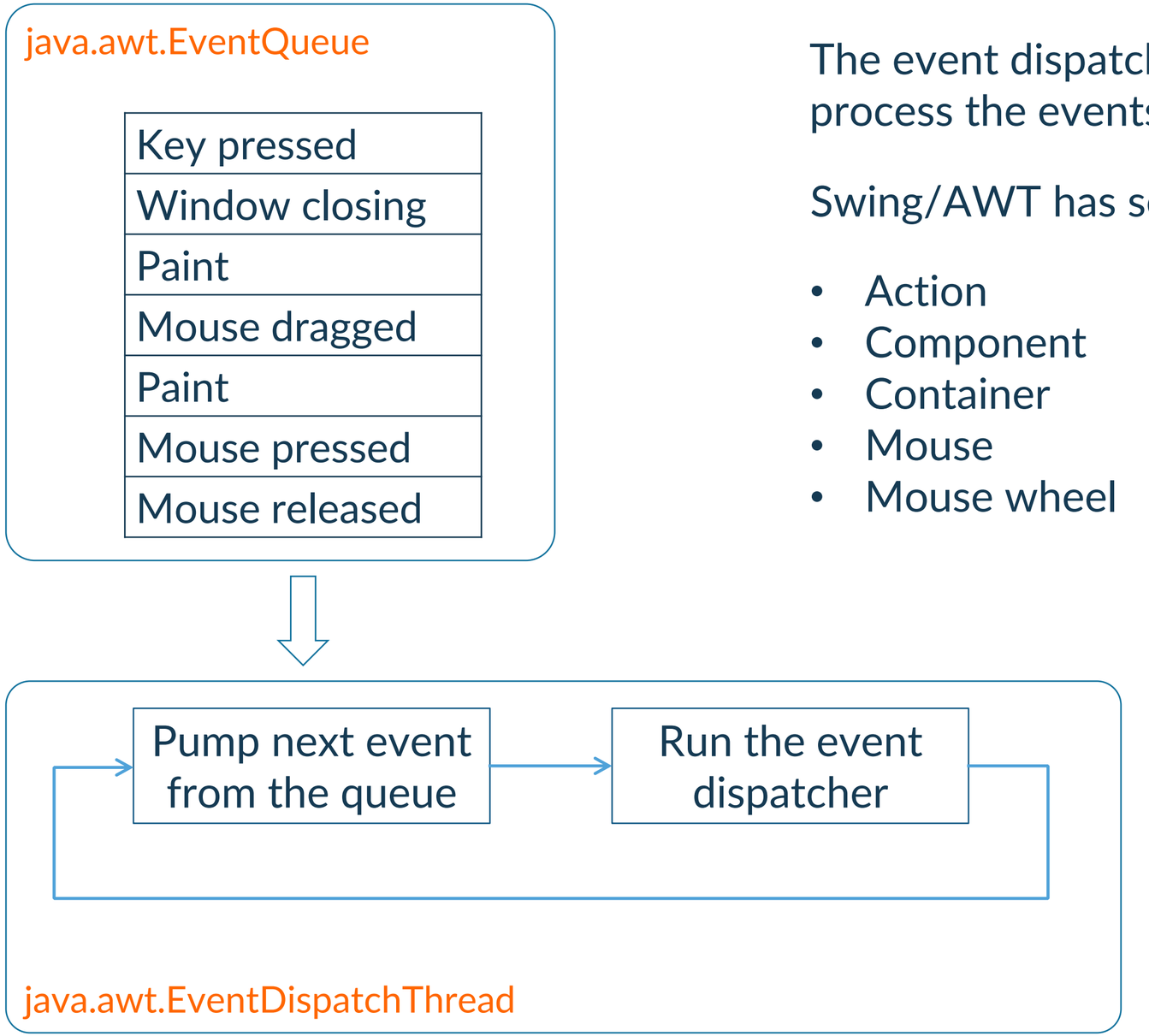
Some Swing component methods are labelled thread-safe in the API specification; these can be safely invoked from any thread. All other Swing component methods must be invoked from the event dispatch thread

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/17/docs/api/java.desktop/javax/swing/package-summary.html

The event queue & event dispatch thread



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.

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

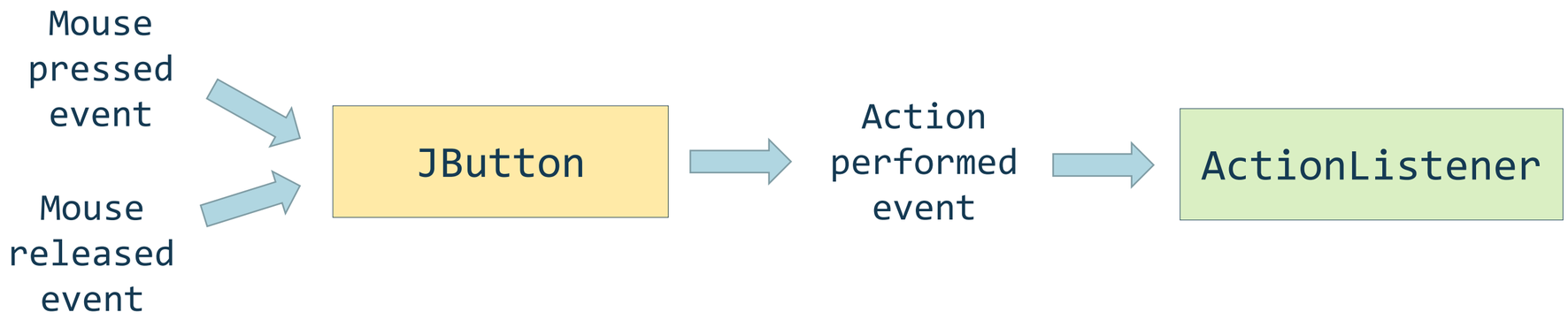
Swing components receive mouse and keyboard events from the window system and they translate these events into component events

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 event. 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

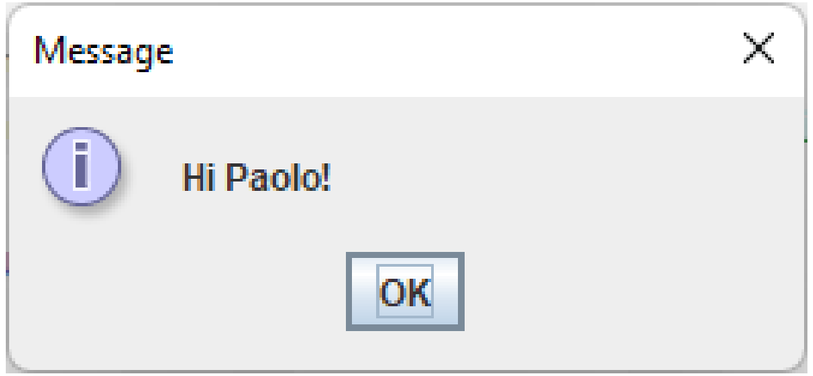
Swing components

- 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

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

JOptionPaneDemo

OptionPaneDemo.java

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

public class OptionPaneDemo {

    public static void main(String[] args) {
        SwingUtilities.invokeLater(OptionPaneDemo::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

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);
```

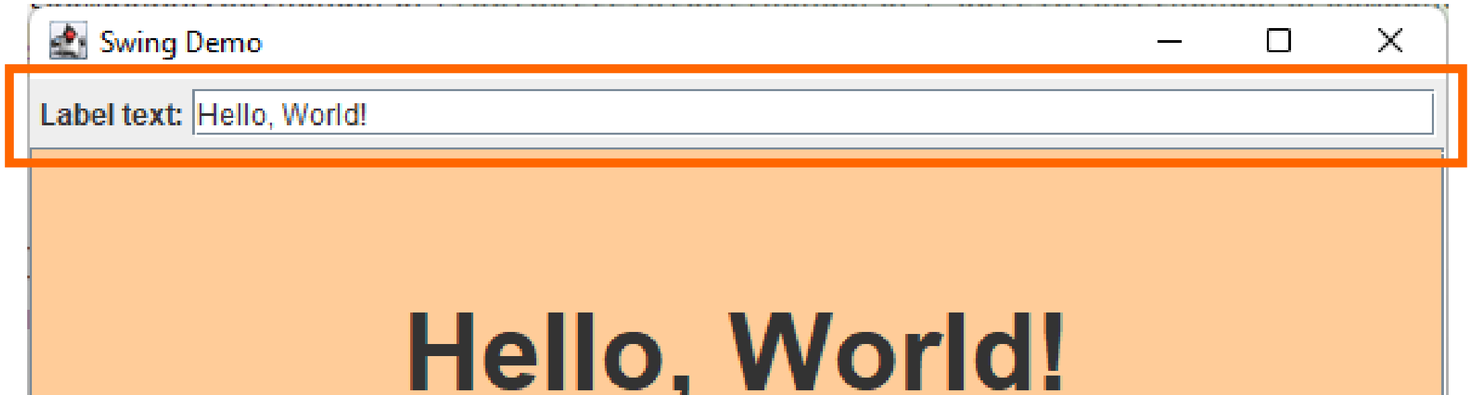
The JScrollPane

The JScrollPane shows the component through a viewport

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

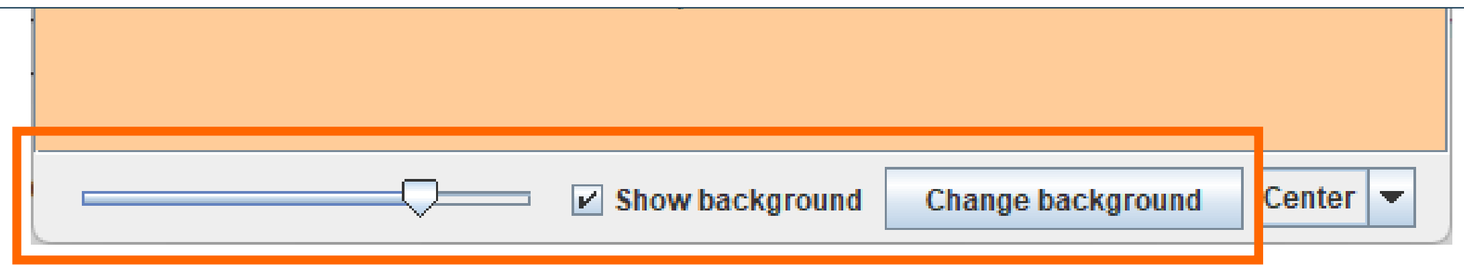
The North panel

```
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));
```



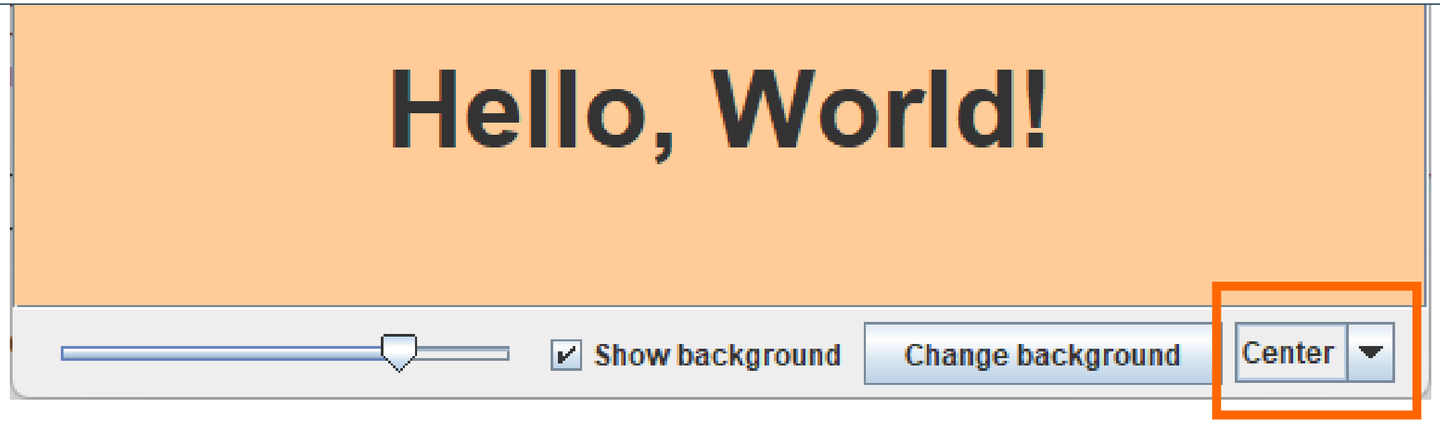
The South panel 1/2

```
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());
```

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

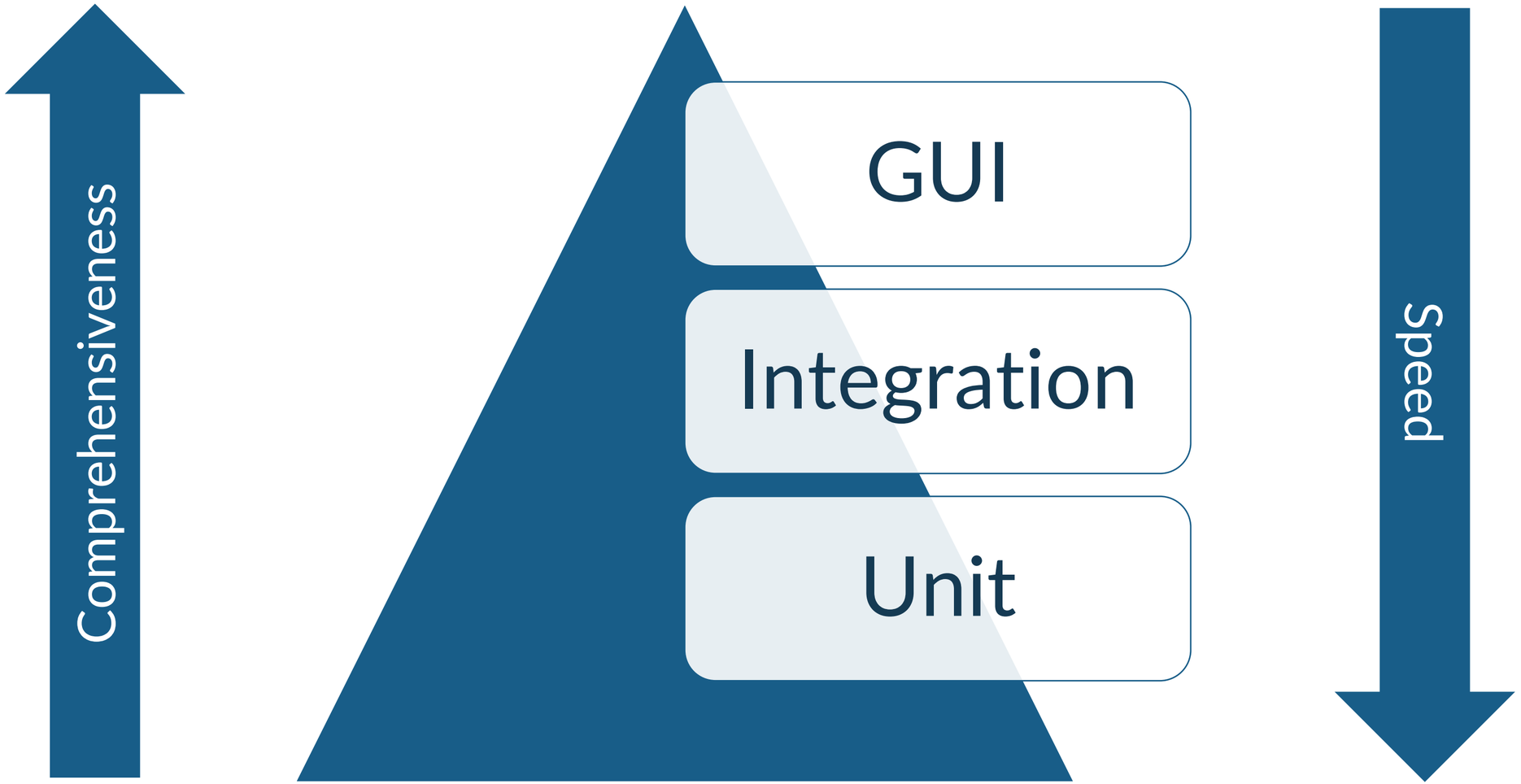
Take aways

- ❑ Components fire events in response to user actions
- ❑ Swing has a rich and comprehensive set of components
- ❑ Swing supports multiple look-and-feels

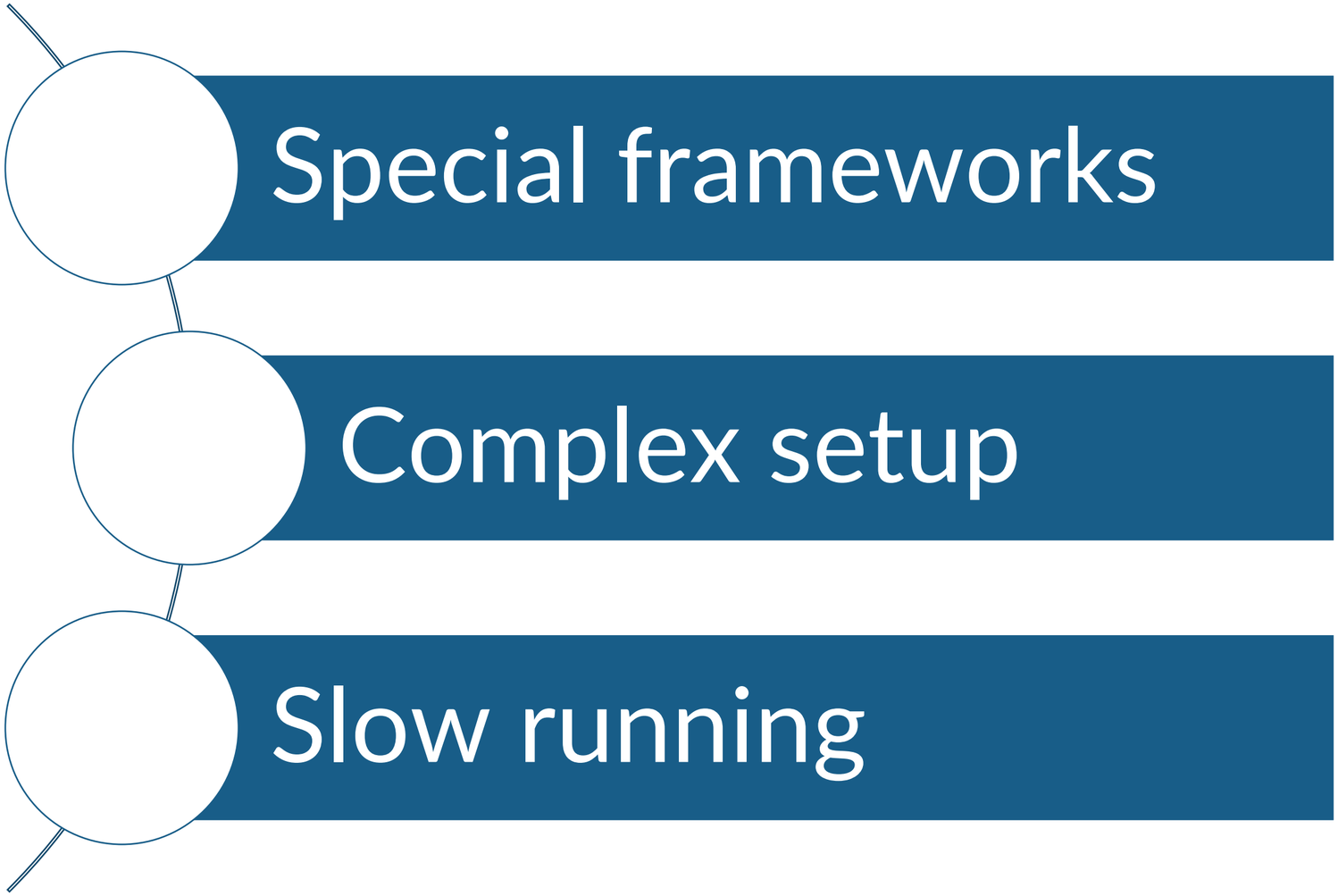
The humble dialog

Decoupling the view from the application logic

The test pyramid



Automation of GUI code



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

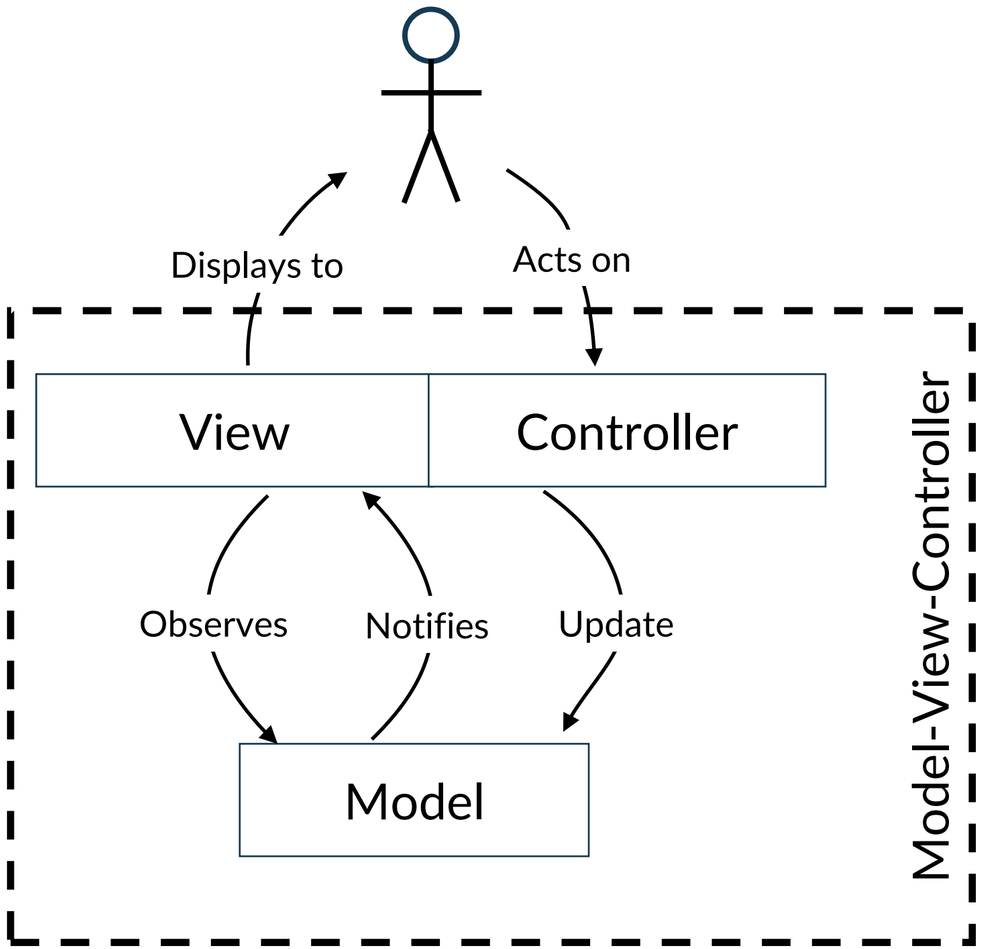
This pattern is applied at the boundaries of the system, where things are often difficult to test, in order to make them more testable. We accomplish the pattern by reducing the logic close to the boundary, making the code close to the boundary so humble that it doesn't need to be tested. The extracted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

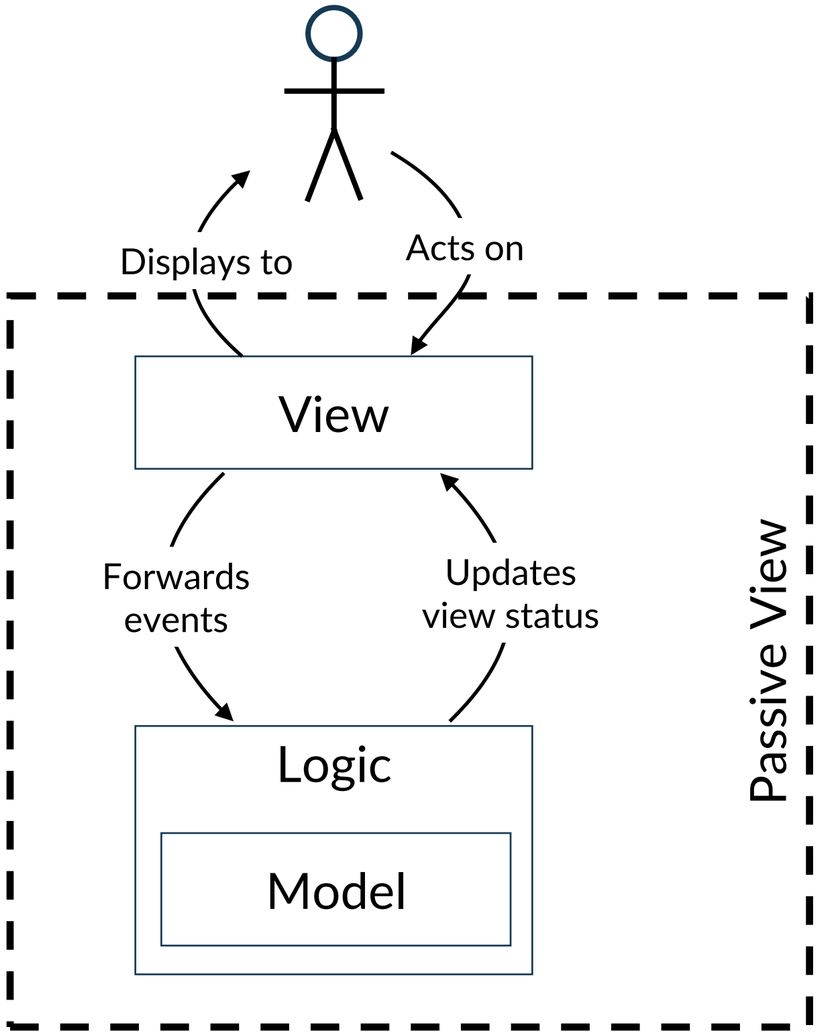
- Robert C. Martin

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





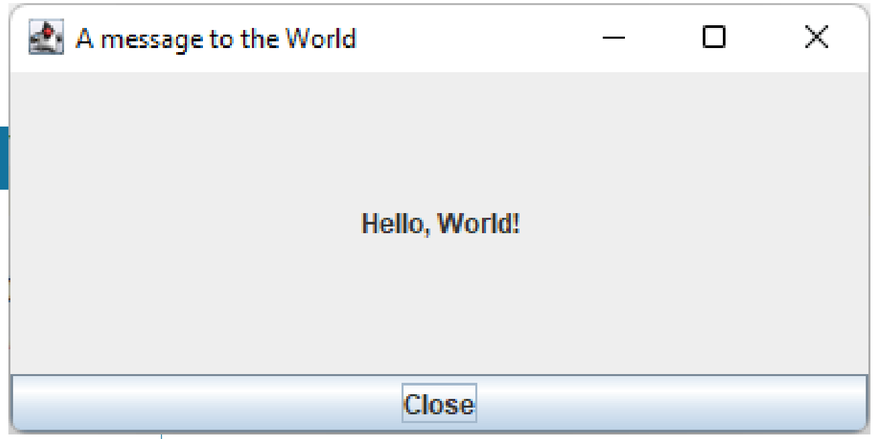
The Humble Dialog

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

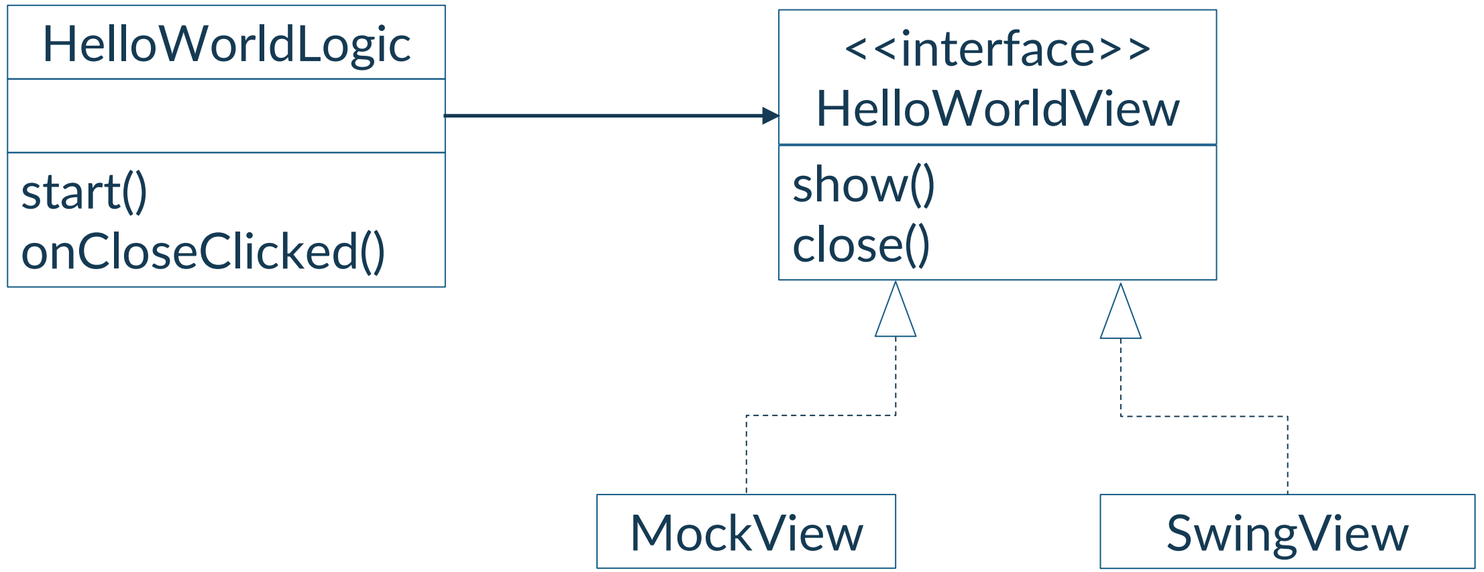
```
public class HelloWorld {

    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);
    }
}
```

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



HelloWorld Logic & View

HelloWorldLogic.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

```
public interface HelloWorldView {

    void close();

    void show();
}
```

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);
    }
}
```

Swing implementation

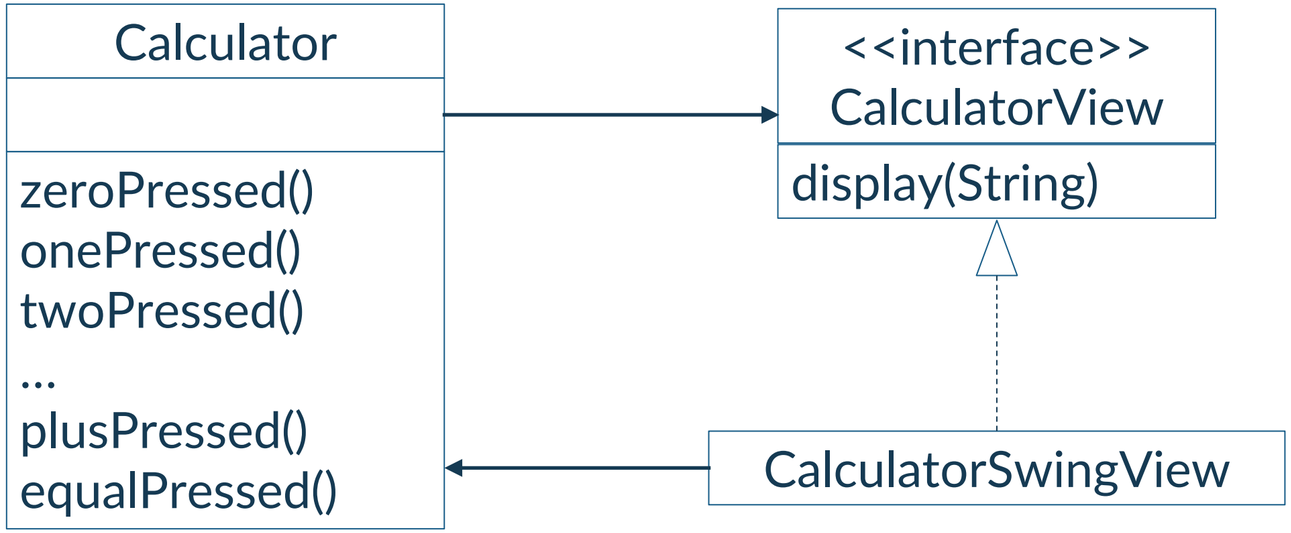
Practical tips

- 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)

Assignment

Implement the whole user interface for the Calculator (pad + display) and couple it with the Calculator class you have already implemented.

- Implement the Calculator class (or at least a few use cases) by using TDD



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
- 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