Graphical interfaces & event-driven programming

Lecture 12 of TDA 540 (Objektorienterad Programmering)

Carlo A. Furia Alex Gerdes

Chalmers University of Technology – Gothenburg University

Fall 2017

Pop quiz!

1. Go to `kahoot.it`
2. Enter PIN shown on projector screen
3. Pick a nickname and go!

Polymorphism: reminder of formal definitions

More rigorously, polymorphism is a type compatibility rule:

If `S` is a subtype of `T`, an expression of type `S` can be used wherever an expression of type `T` is expected.

Since objects of type `S` are a specialization of objects of type `T` (a `Convertible` is a `Car`!), polymorphism still supports compiler checks and avoids type incompatibility errors.

```
interface List<E> {
  E get(int index);
  void add(int index, E e);
  int size();
}
```

```
List<String> l;
// the following operations are
// consistent with the List interface
l.add(0, "hej");
l.add(1, " då");
if (l.size() > 0)
 System.out.println(l.get(0) + l.get(1));
```

GUIs: Graphical User Interfaces

This class is about programming graphical interfaces (GUIs) in Java.

The Java language framework provides an extensive collection of libraries for GUI programming. Thus, programming GUIs means learning how to use those libraries.

GUI programming is a domain where object-oriented programming shines:

- classes model different graphical components (windows, buttons, scroll bars, . . .)
- the relations between components (e.g. different buttons, or a button as a specialized component) are captured by inheritance
- polymorphism supports flexible reuse of the different components, without worrying about implementation details

More documentation about Java GUIs

Some pictures in this class are taken from the detailed Java GUI programming tutorial by Chua Hock Chuan at `www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html`

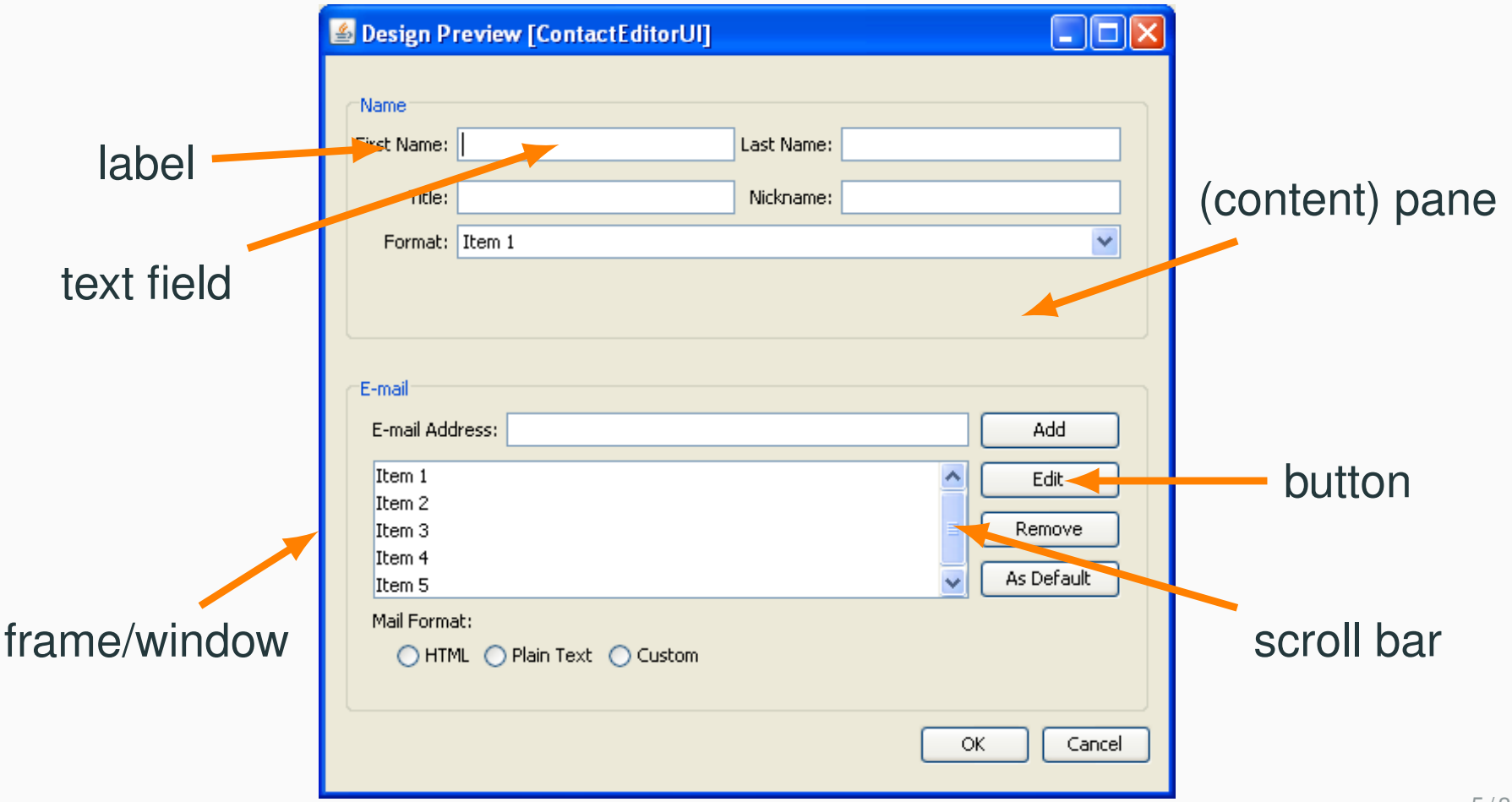
Another recommended tutorial is the official Java Swing tutorial at `http://docs.oracle.com/javase/tutorial/uiswing/index.html`

As usual, the AWT and Swing API documentations are also useful:

- `https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html`
- `https://docs.oracle.com/javase/8/docs/api/java/awt/package-summary.html`

Graphical components

GUI programming relies extensively on the notion of (graphical) components. A component is a class modeling an actual graphical element of the GUI.



AWT vs. Swing

AWT (Abstract Windowing Toolkit) and Swing are the two main libraries for GUI programming in Java.

We will use mostly Swing, which is newer and has some advantages over AWT. However, every GUI typically needs at least some basic AWT components, and hence we have to learn at least the basics of AWT as well.

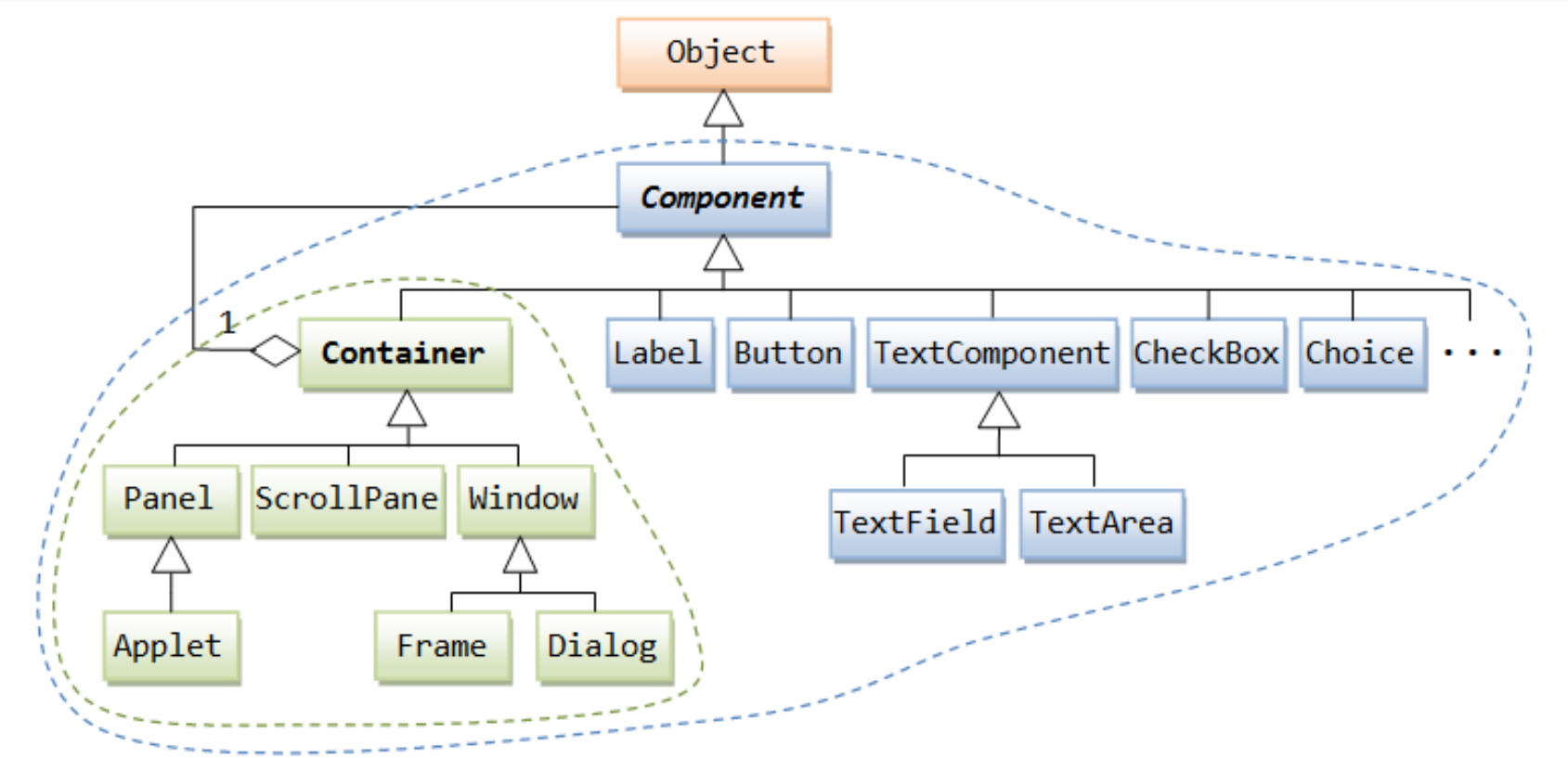
AWT	Swing
heavyweight: Java interface to native GUI components implementations	lightweight: Java GUI components implementations
may look different on different systems	consistent "look and feel" across different systems
generally faster (native)	slower (implemented in Java)
	relies on AWT for top-level containers

AWT

- Historically, first Java library for GUI programming
- AWT components are Java classes wrapping native libraries on different operating systems (OS X, Windows, GTK+, . . .)
- Platform independent (like all Java), but does not look the same on all systems
- Swing uses AWT for top-level components, to bootstrap the visualization of the GUI

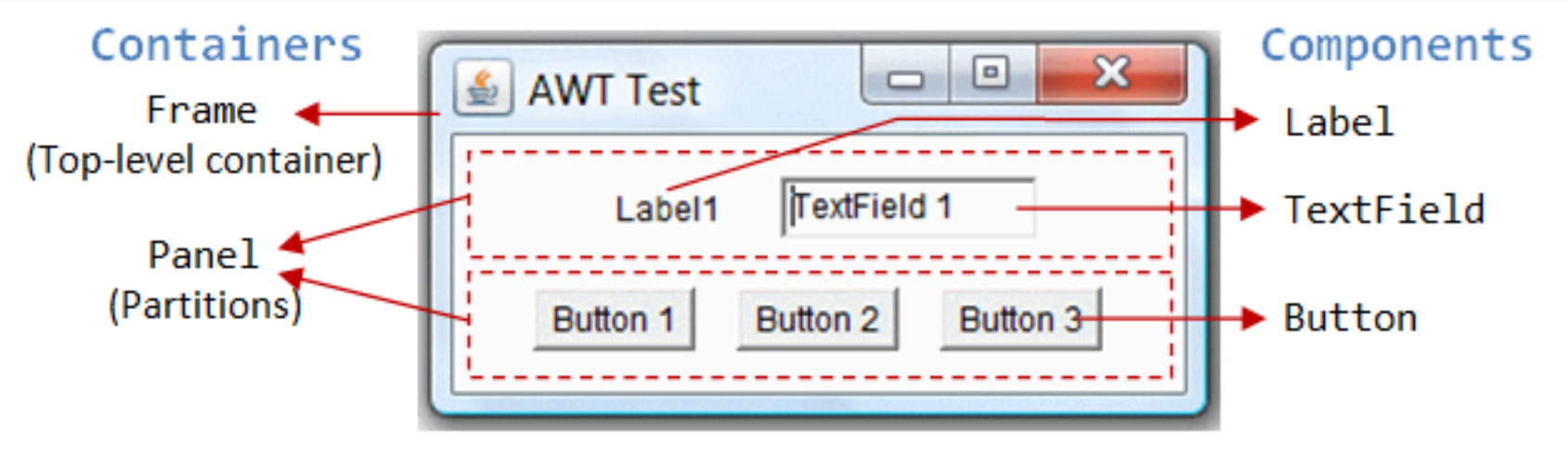
AWT components

AWT classes distinguish between components and containers: each container can include one or several components.



AWT components

AWT classes distinguish between components and containers: each container can include one or several components.



AWT in a nutshell

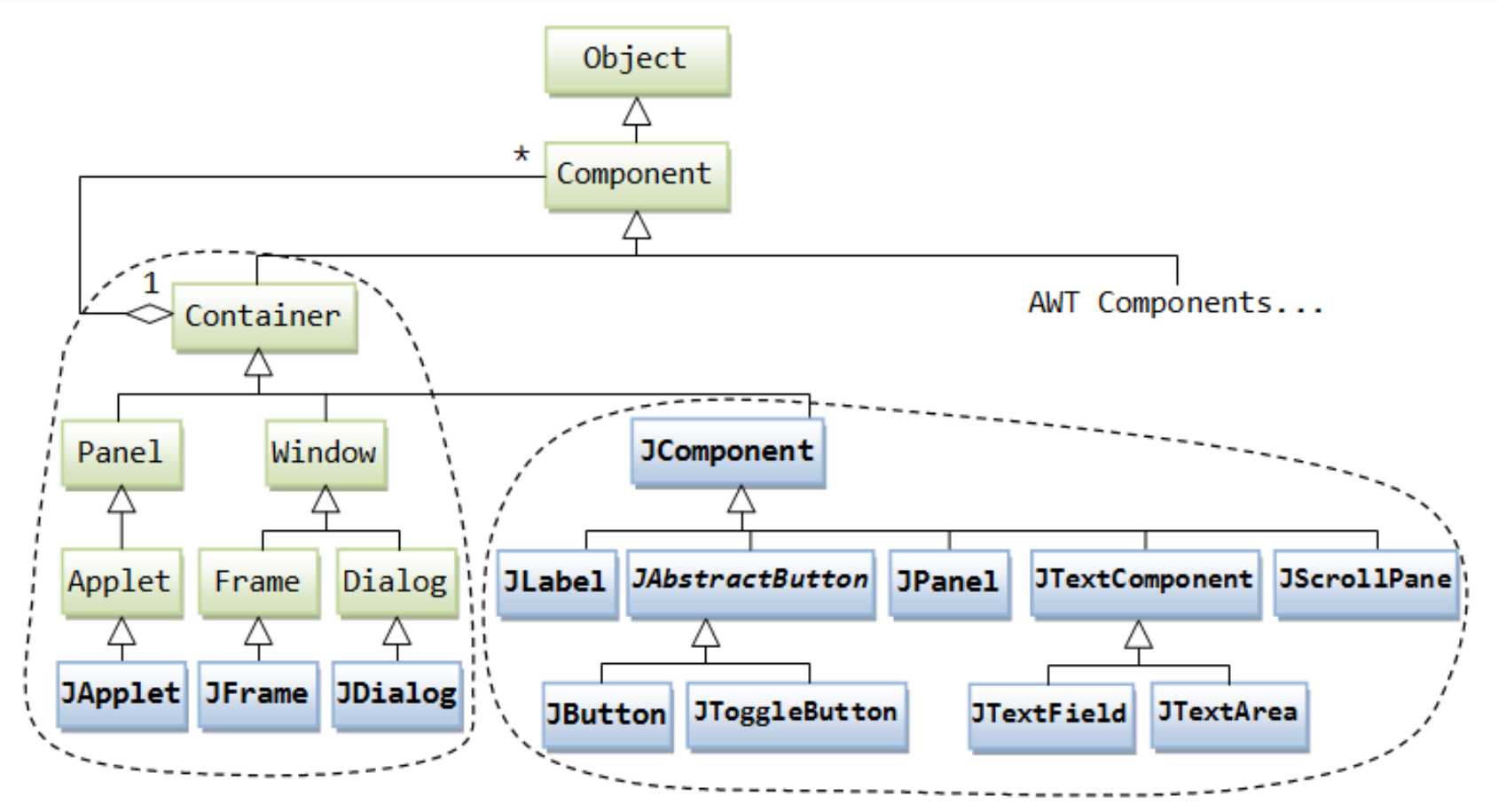
AWT is huge: 12 packages, hundreds of classes.

Main packages, which are also (partially) used in Swing applications:

- `java.awt`: components and containers, and layout managers
- `java.awt.event`: event-handling library

Swing components

Swing classes have names that start with `J`, so we can immediately distinguish them from AWT components.



Swing in a nutshell

Swing is huge: 18 packages, hundreds of classes.

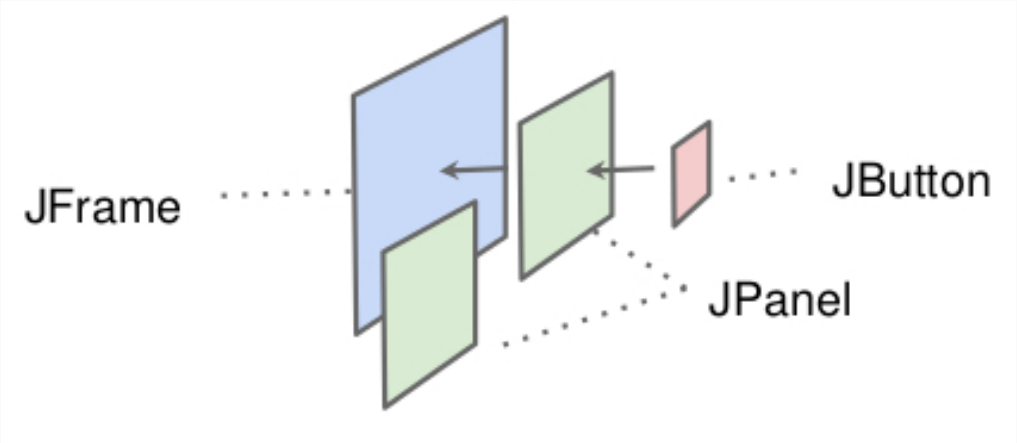
Main package:

- `javax.swing`: Swing components and containers

Since Swing components are implemented on top of AWT components, you typically need to import some of the libraries from AWT even when developing a pure Swing application.

The structure of Swing GUIs

- Top-level component: usually `JFrame` or `JDialog`
- Secondary components (containers), used to group and layout simpler components: often `JPanel`
- Atomic components, which correspond to the various GUI elements: `JButton`, `JTextField`, `JTable`, `JScrollBar`, `...`



The layout of the atomic components within the containers is normally done using layout managers, which are introduced independently of the component structure and content.

How to build a Swing frame

1. Create a class that inherits from a top-level component (`JFrame`)
2. Set up a content pane within the top-level component class
   - 2.1 either get the default content pane from the top-level component
   - 2.2 or create your own `JPanel` and set it to the top-level component
3. Normally, set a layout manager of the content pane
4. Add components to the content pane
5. Set the top-level frame to visible, and possibly specify other options

In all examples use:

```
   // basic AWT components, and layout managers
import java.awt.*;
   // event-handling library
import java.awt.event.*;
   // Swing components
import javax.swing.*;
```

A simple Swing frame: version 1

This example uses the default content pane and no layout manager.

```
public class SimpleFrame extends JFrame {
  protected Container contentPane;

  SimpleFrame() {
    // get the default content pane, which is an AWT container
    contentPane = this.getContentPane();
    // add a text label
    contentPane.add(new JLabel("I'm just a simple frame!"));

    // close the window when clicking on close
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // resize frame to fit components tightly
    pack();
    // do display the frame!
    setVisible(true);  } }
```

A simple Swing frame: creating and displaying

This is how you can create and display a frame defined by class `SimpleFrame` from `main`.

```
public static void main(String[] args)
{   SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
            new SimpleFrame();
        }
    });  }
```

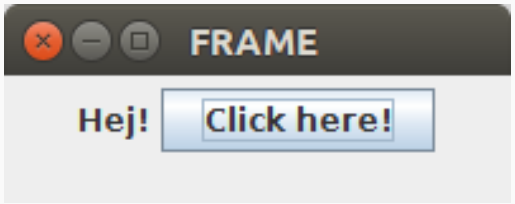
While we could just create `SimpleFrame` directly, using `invokeLater` is a better practice that avoids problems when writing more complex GUI applications.

A simple Swing frame: version 2

This example uses a `JPanel` content pane and a flow layout manager.

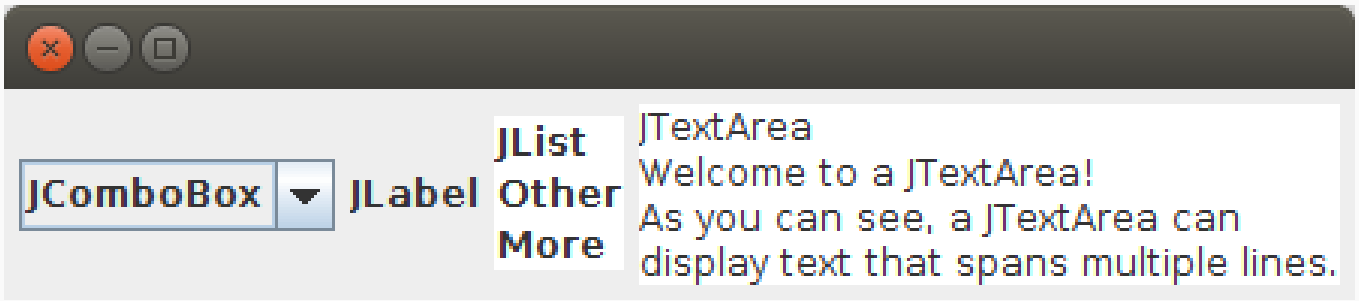
```
public class SimpleFrame extends JFrame {
  protected JPanel contentPane;

  SimpleFrame() {
    // set the content pane to a Swing JPanel
    contentPane = new JPanel();
    this.setContentPane(contentPane);
    // set a "flow" layout manager
    contentPane.setLayout(new FlowLayout());
    // add a text label, and a button
    contentPane.add(new JLabel("Hej!"));
    contentPane.add(new JButton("Click here!"));
    setTitle("FRAME");  // frame title
    setSize(200, 50);   // fixed size
    setVisible(true);   // do display the frame!
  } }
```



A few useful Swing components

- `JLabel`: simple text label or picture
- `JButton`: clickable button
- `JComboBox`: pull-down menu with mutually-exclusive options
- `JList`: list of selectable options
- `JTextField`: single-line text
- `JTextArea`: multi-line text
- `JScrollPane`: scroll bar
- `JToolBar`: list of clickable buttons (or other components)
- `JOptionPane`: pop-up dialog boxes (typically spawned in response to events)



Layout managers

Layout managers are AWT facilities (also usable with Swing components) that simplify arranging components in a frame. Layout managers provide flexibility:

- no absolute positioning, and hence no dependence on the settings of the computer that will display the GUI
- when the user resizes the frame, layout managers rearrange the components to fit the new frame

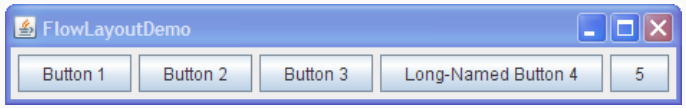
Each layout manager follows a different criterion to position components as they are added to a frame by calling `add`.

For more complex layouts, nest `JPanel`s at certain positions, and use different layout managers in each nested panel.

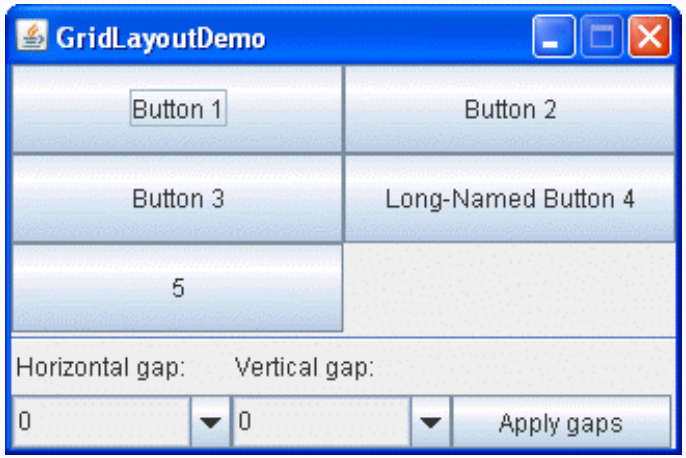
Setting the layout manager to `null` `contentPane.setLayout(null)` uses absolute positioning, where we have to manually specify the coordinates and size of each component. This is usually very tedious, inflexible, and not necessary.

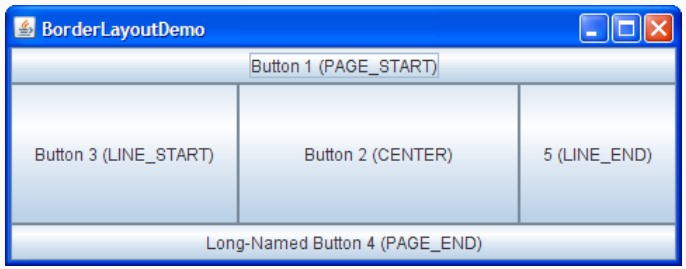
Some examples of layout managers

Each layout manager follows a different criterion to position components as they are added to a frame by calling `add`.



- `FlowLayout` (the default of `JPanel`): add components on a row, left to right; when the row is filled, start a new row below
- `GridLayout`: add components in a matrix of fixed dimensions; add left to right, and top to bottom
- `BorderLayout`: add each component to one of five fixed zones (north, south, east, west, and center)

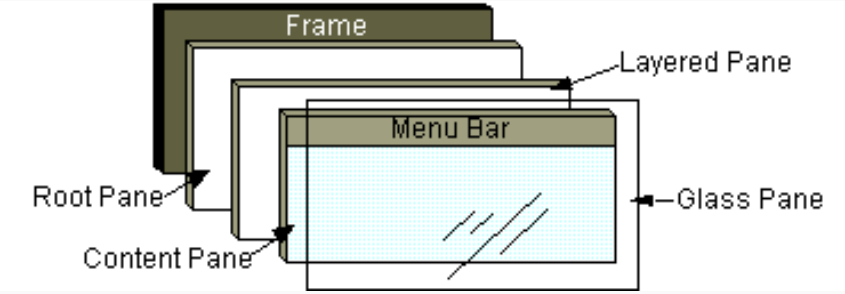




More layouts shown at `http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html`

Root panes

Top-level components such as `JFrame` are themselves layered into several panes:



We have already seen the (default) content pane.

- The layered pane is to control positioning of the menu bar with respect to the content pane
- The menu bar is optional, and disabled by default
- The glass pane is invisible by default; it can be used for adding color or clickable areas across components

Changing the way components look

Swing provides a consistent look across platforms, but it also offers the possibility of switching to a customizable "look and feel".

- Metal (or Cross Platform) is the Swing default
- System is the native look and feel of the operating system running the application (Windows, macOS, GTK, . . .)
  Note: use System if you need DPI scaling – if text looks too small in high-resolution environments
- Synth is a customizable look and feel
- Multiplexing supports using different look and feels for different parts

```
// before everything GUI-related
try { UIManager.setLookAndFeel(  // set System look & feel
          UIManager.getSystemLookAndFeelClassName());
} catch (ClassNotFoundException | InstantiationException |
      IllegalAccessException | UnsupportedLookAndFeelException e)
  { System.out.println("Using default look and feel!"); }
```

Event-driven programming

The programs we have written so far are sequential: statements are executed one after the other, from the entry point in `main`, according to which conditions occur during execution.

However, a GUI must be able to execute statements according to some events that may occur at any time while the program is running: mouse clicks, window resizing, keys pressed, ... Thus, reactive GUIs follow a style of programming called event-driven programming.

WHAT HAPPENS	EXAMPLE
An event involving a component occurs	A close-window button is clicked
The component notifies handlers of the event	The button notifies the window manager
The handler executes and reacts to the event	The window manager closes the windows

Publish/subscribe communication model

Java GUIs support event-driven programming according to the publish/subscribe model.

- Components are publishers of events.
- Other objects implement event-handling code specific to a certain event.
- A handler for an event E subscribes (that is, registers) to a component that publishes events E.
- Whenever a component triggers an event E, it notifies all handlers for that event that have been registered.
- A notified handler executes its code in response to the event.
- Eventually, control returns to the component.

Publish/subscribe event-handling in Java

Java GUIs support event-driven programming according to the publish/subscribe model.

- Swing components include methods of the form `addEListener(EListener handler)` to register handlers of a specific event `E`
- Type `EListener` corresponds to an **`interface`**, whose methods are called whenever the corresponding events are triggered
- A handler for event `E` is an object of a subtype of `EListener`; that is, the handler's class **`implements`** interface `E`
- Whenever a component triggers an event E, it notifies all handlers for that event that have been registered by calling the proper method of `EListener`

In general, handlers and publishers are completely independent. In practice, they often need to communicate and share state, because "handling" an event often requires to change some parts of the GUI.

A button that changes text

Swing component `JButton` includes a method `addActionListener(ActionListener handler)` to register handlers of the “action” event (clicking the button).

```
public class ClickableButton extends JFrame {


  ClickableButton() {
    // ...
    // initial text of button
    JButton button = new JButton("You never clicked!");
    // ...
    // CountClicks handler registers with the button
    button.addActionListener(new CountClicks(button))
    // ...
  }
}
```

A button that changes text

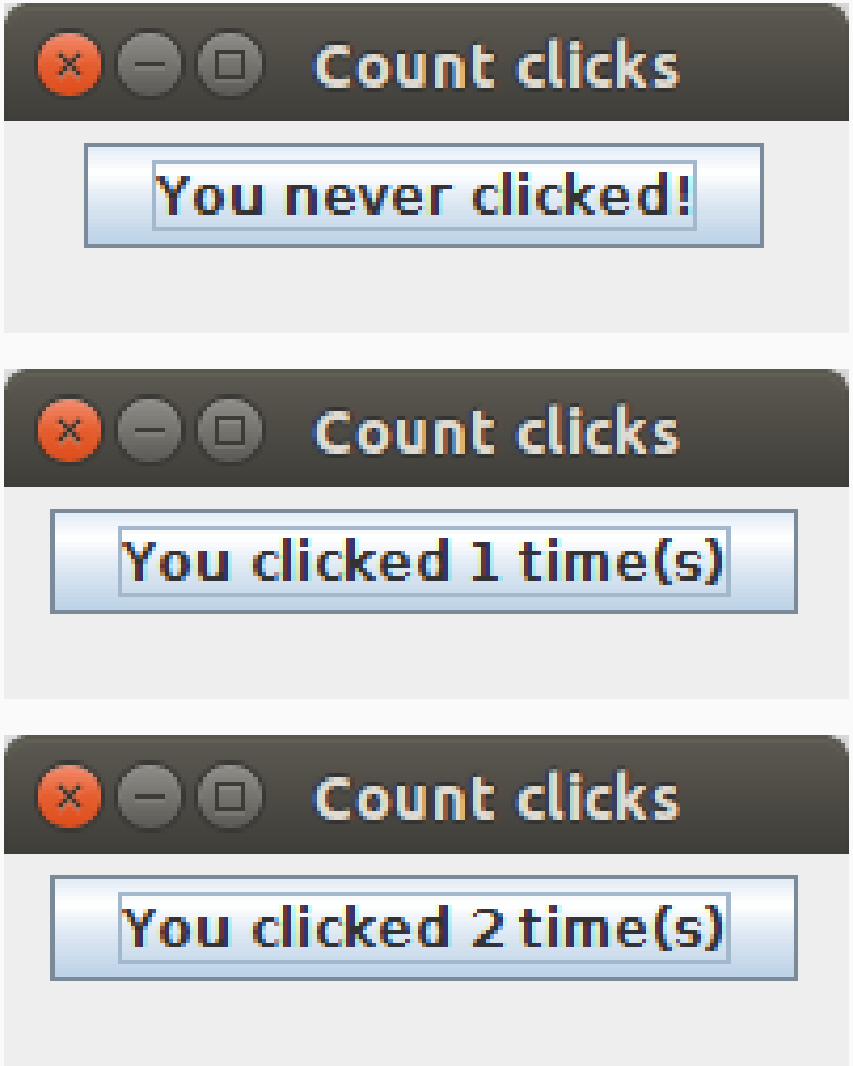
Interface `ActionListener` includes a single method `actionPerformed`, called when the corresponding event is triggered (clicking the button).

```
public class CountClicks implements ActionListener {
  private JButton button;  private int count;

  CountClicks(JButton button)
  { this.button = button; count = 0; }

  // whenever a click occurs
  public void actionPerformed(ActionEvent evt) {
    // increment counter
    count++;
    // change the button's text
    button.setText("You clicked " + count + " time(s)");
  }
}
```

A button that changes text



Some events and listener interfaces

LISTENER	METHODS	COMPONENTS
ActionListener	actionPerformed	JButton, JComboBox, JTextField,...
FocusListener	focusGained	JComponent
	focusLost	
MouseListener	mouseClicked	JComponent
	mouseClicked	
	mouseEntered	
	mouseExited	
	...	
KeyListener	keyPressed	JComponent
	keyReleased	
	keyTyped	
InputMethodListener	caretPositionChanged	JTextComponent
	inputMethodTextChanged	

Inner classes

The handler class is often coupled with the component's class, and it is very simple (it only implements the listener interface's methods). Java offers anonymous inner classes to supply methods implementations directly where we instantiate the listener.

```
public class ClickableButton extends JFrame {
  ClickableButton() {
    // ...
    JButton button = new JButton("You never clicked!");
    // ...
    // instantiate an object of the anonymous inner class
    // which is being defined directly
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         button.setText("You clicked!");
      }});
    // ...
  } }
```

Nested classes

Anonymous inner classes are just one type of nested classes. A nested class is a class defined inside another class, which may access some of its private data. The opposite of nested class is top-level class: these are the classes we have seen so far.

- static nested class: no reference to the outer (non-static) instance
- inner classes: can reference the outer class instance
- anonymous inner classes: inner class without a name, defined as part of an expression
- local inner classes: inner class with name, defined as part of an expression

Static nested classes

A static nested class:

- can reference only static members of the enclosing class (besides its own arguments and locals)
- can itself include both static and non-static members
- it behaves as a top-level class: nesting affects naming, not behavior

```
public class Top {
  static class SN {
    static int five()
    { return 5; }
    int three()
    { return 3; }
  }
}
```

```
int y = Top.SN.five(); // y == 5
Top.SN o = new Top.SN();
int x = o.three(); // x == 3
```

Inner classes

An inner class:

- cannot include static members (other than constants)
- can reference the outer class instance: all instances of the inner class refer to the instance of the outer class used to create them
- can be created only through an instance of the outer class

```
public class Top {
  int a;
  class I {
    int three()
    { return 3; }
    int getA()
    { return a; }
  }
}
```

```
Top t = new Top();
Top.I i = new t.new I();
int x = i.three(); // x == 3
t.a = 4;
Top.I j = new t.new I();
int y = j.getA();  // y == 4 == i.m()
```

Anonymous and local inner classes

An anonymous or local inner class:

- cannot include static members (other than constants)
- can reference the outer class instance: all instances of the inner class refer to the instance of the outer class used to create them
- cannot access local variables of its enclosing class (except constants)

```
public class ClickableButton extends JFrame {
  int counter = 0;
  ClickableButton() {
    // ...
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         button.setText("Counter: " + counter);
    }});
    // ...
  } }
```

Heavy computations in GUIs

In event-driven programs such as GUIs, a handler that takes too long to react may compromise the responsiveness of the whole GUI.

```
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
     // encode 2-hour video in high resolution
   }
 });
```

To avoid this problem, we should allocate such heavy computations to a special executor (thread) in the Java system, which can run the computation in the background without blocking the GUI.

Swing workers

We allocate heavy computations to a special executor (thread) in the Java system, which can run the computation in the background without blocking the GUI.

```
SwingWorker worker = new SwingWorker<Void, Void>() {
  @Override
  public Void doInBackground() {
     // encode 2-hour video in high resolution
     return null;
   }
};
```

- Start the worker's computation: `worker.execute()`
- Check whether the worker has finished computing: `worker.isDone()` (returns `boolean`)
- Get the results of the worker's computation after completion: `worker.get()`