Lesson – 7
Building GUIs in Java with Swing

Wholeness of the Lesson

Swing is a windowing toolkit that allows developers to create GUIs that are rich in content and functionality. The ultimate provider of tools for the creation of beautiful and functional content is pure intelligence itself; all creativity arises from this field's self-interacting dynamics.

Background

- Swing – A set of GUI classes
  - Part of the Java's standard library
  - Much better than the previous library: AWT
    - Abstract Window Toolkit
    - AWT Still Used. Swing components still make use of aspects of the AWT – Swing is built "on top of" the old AWT. In particular, handling of events relies on the old event-handling model.
- Highlights
  - Swing has a rich and convenient set of user interface elements.
  - It supports MVC pattern.
  - Platform independent.
  - Components are often called ***lightweight*** components.
- ***JavaFX***. In 2014, Oracle declared that Swing libraries would be developed no further, and that the windowing toolkit of choice had become JavaFX. JavaFX has more modern-looking components and has a more flexible API. Since Swing is still (as of 2015) far more widely used than JavaFX, Swing is presented here. With the release of Java 8, JavaFX became an integral part of the JRE (and JDK)

- **Return of Swing.** In 2018, Oracle announced that, starting with JDK 11, JavaFX will no longer be bundled with the JDK, but will be available through a separate download. The JDK 8 version of JavaFX will continue to be supported through the "open source" project through 2022. On the other hand, Oracle has announced that it will resume support of Swing (along with AWT) in JDK 8 and 11 and for the foreseeable future.
- ***Industry Standard***. For standalone GUI development in Java, Swing is the toolkit most often used.
- ***Visual Designers***
  - Most widely used (as of 2016) is Netbeans, which provides excellent visual support for Swing.
  - Visual designers are better for prototypes than for creating production-quality UIs that need to be maintained
  - Usually, to use a visual designer effectively, you need to have a good understanding of how to write code to produce the effects you want.
  - SceneBuilder is a visual tool that comes with JavaFX – to use this tool, it is essential that a developer already knows how to program in JavaFX.

Outline of Topics

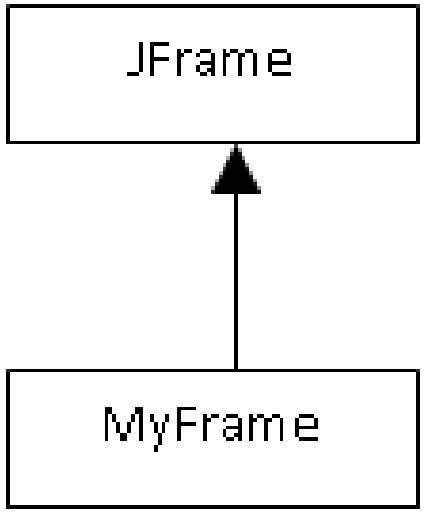
- Swing Components and Containers
- Inheritance in Swing
- Laying Out Components with Layout Managers
- Handling Events
- Additional Technique: Displaying Pop-up Windows
- MVC Design Pattern

Main Idea in Swing

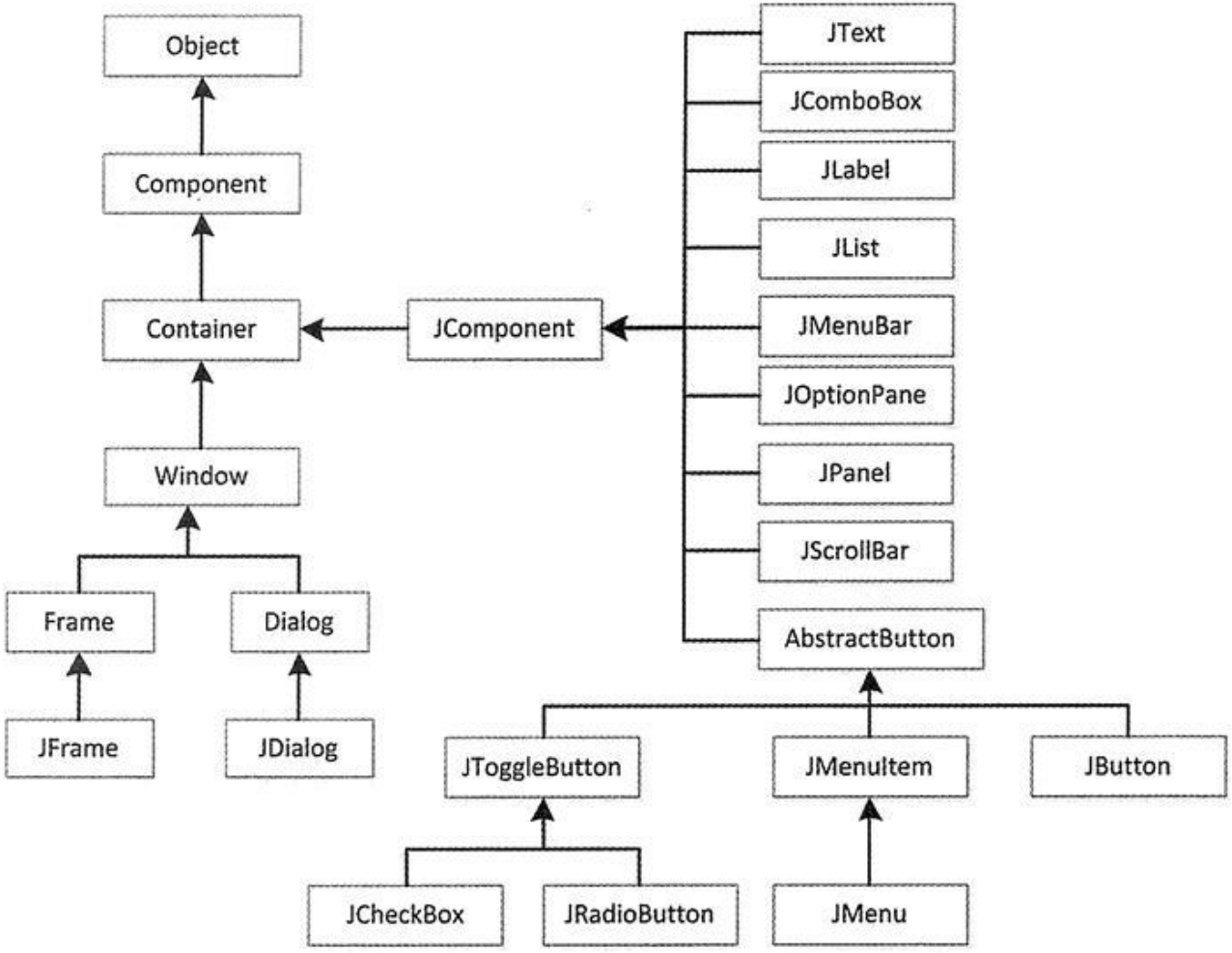
- ***Components and containers***. Swing provides components (like text boxes, buttons, checkboxes) and containers (frames, panels, applets, dialog) in which such components can be placed.
- ***Containers placed in other containers***. In Swing, a container is also considered to be another kind of component, so containers can be placed in other containers.
- ***Layout Managers for containers***. Every container supports the use of a layout strategy. To achieve the visual objectives in building Swing screens requires skillful use of layouts on multiple containers. We will do this in a simple way.
- ***Listeners = Event Handlers***. A Swing GUI becomes responsive to user actions (like button presses, item selections, etc.,) by means of an event handling model. In this model, there are "listeners" for user actions (like button presses and mouse clicks). When a relevant user action occurs, the listener is informed and the code that you have written to handle the event will then be executed.

Inheritance in Swing

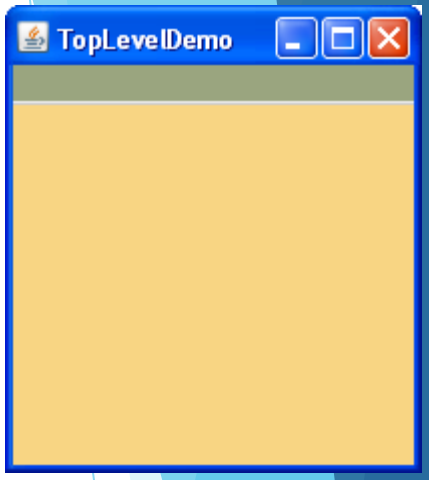
The code makes it clear that, when you design a Swing application, you start by creating a *subclass* of Jframe. The class diagram in UML is the following:



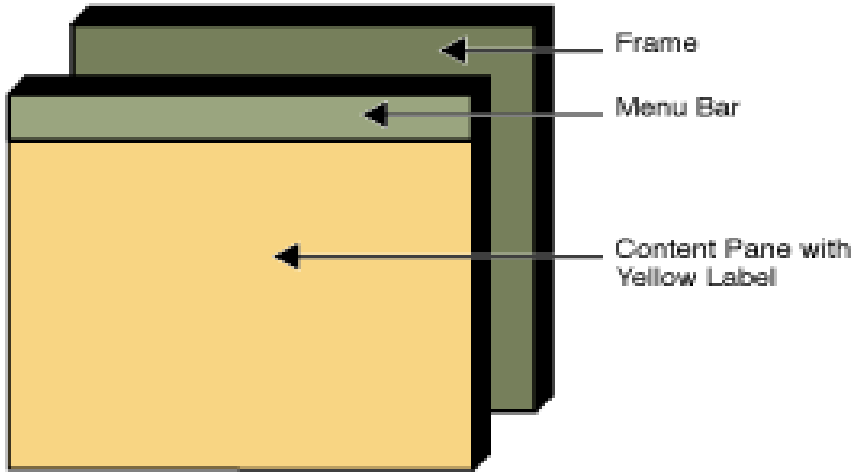
Inheritance Hierarchy for Swing



Top Level Containers: JFrame

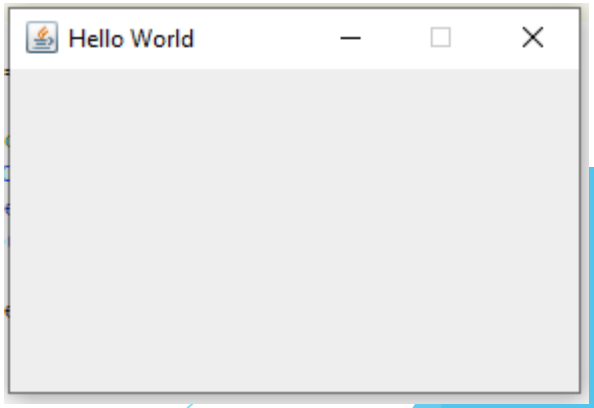


- **`Import javax.swing.JFrame:`**
  - A main and Top-level window with a title and a border.
  - Can add components to the Contentpane(Container).
  - ability to minimize, maximize, and close the window
  - The contentPane is the area of the JFrame one can modify. Access it using the getContentPane()method of a JFrame object.



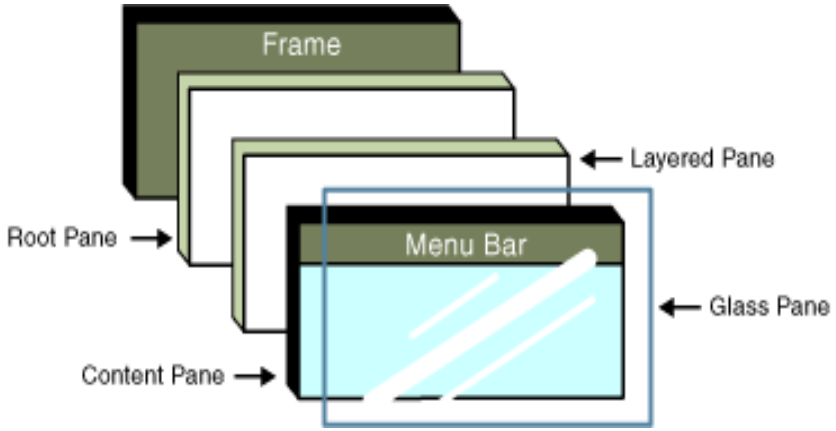
My First Swing Program

```
public class HelloWorldFrame extends JFrame {

    public static void main(String[] args) {

        EventQueue.invokeLater(()->{

            HelloWorldFrame frame = new HelloWorldFrame();

            frame.setTitle("Hello World");
            frame.setSize(300, 200);
            frame.setResizable(false);
            frame.setLocation(600, 400);

            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        });
    }
}
```



Content pane

- Every top-level container indirectly contains an intermediate container known as a *content pane*.
- The content pane contains components in the window's GUI.
- To add a component to a container, you use one of the various forms of the `add` method.



Contentpane

```
public class ContentPaneDemo extends JFrame {

    public ContentPaneDemo() throws HeadlessException {
        setTitle("Background Changing");
        setSize(300, 300);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
        Container contentPane = getContentPane();
        contentPane.setBackground(Color.blue);
    }

    public static void main(String[] args) {
        new ContentPaneDemo();
    }
}
```

Laying out components

- Layout managers – basically tells form how to align components when they're added.
- Each Container has a layout manager, but you can always set your own.
- A JPanel is a Container – used to gather elements together.

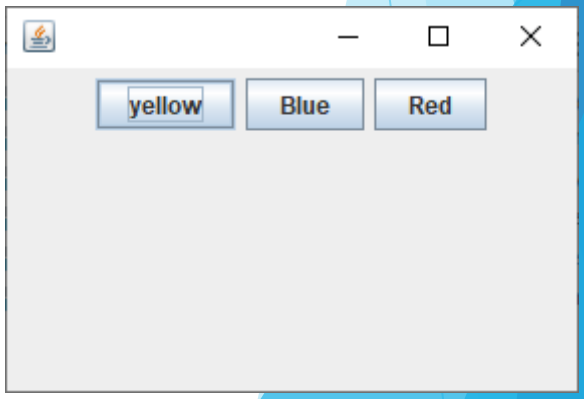
  to have different layout managers associated with different parts of a form, tile with JPanels and set the desired layout manager for each JPanel, then add components directly to panels.
- Most common and easiest to use are
  - FlowLayout
  - BorderLayout
  - GridLayout

FlowLayout manager

- Components are placed in a row from left to right in the order in which they are added.
- The default is to center the components but you can align them to the left or right by specifying LEFT or RIGHT in the constructor.
- A new row is started when no more components can fit in the current row.
- You can choose how you want to arrange components in this row.
- **`FlowLayout` is the default layout for `JPanels`.**

FlowLayout – Example

```
public class ButtonFrame extends JFrame {
    private JPanel buttonPanel;
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    public ButtonFrame() {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        setLocationByPlatform(true);

        JButton yellowButton = new JButton("yellow");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");

        buttonPanel = new JPanel();

        buttonPanel.add(yellowButton);
        buttonPanel.add(blueButton);
        buttonPanel.add(redButton);

        add(buttonPanel);

    }

    public static void main(String[] args) {
        ButtonFrame frame = new ButtonFrame();
        frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```



Border Layout

- **This is the default layout manager for the JFrame.**
- It has 5 sections - North, South, East, West, and Center.
- If you don’t specify NORTH, SOUTH, EAST, WEST, or CENTER then CENTER is assumed by default.
- The programmer specifies the area in which a component should appear.
- The relative dimensions of the areas are governed by the size of the components added to them.

Border-Layout Example

```
public class ButtonFrame extends JFrame {
    private JPanel buttonPanel;
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    public ButtonFrame() {
        setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        setLocationByPlatform(true);

        JButton yellowButton = new JButton("yellow");
        JButton blueButton = new JButton("Blue");
        JButton redButton = new JButton("Red");

        buttonPanel = new JPanel();

        buttonPanel.add(yellowButton);
        buttonPanel.add(blueButton);
        buttonPanel.add(redButton);

        add(buttonPanel, BorderLayout.SOUTH);

    }

    public static void main(String[] args) {
        ButtonFrame frame = new ButtonFrame();
        frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Setting layout managers

- Very easy to associate a layout manager with a component. Simply call the **setLayout** method on the Container:

```
JPanel p1 = new JPanel();
p1.setLayout(new FlowLayout(FlowLayout.LEFT));

JPanel p2 = new JPanel();
p2.setLayout(new BorderLayout());

JPanel p3 = new JPanel();
p3.setLayout(null);
```

As Components are added to the container, the layout manager determines their size and positioning.

Null Layout : ButtonSetDifferent.java

Swing Components

Text Input

- Text fields
  - Use JTextField to accept one line of text

```
JPanel panel = new JPanel();
JTextField textField = new JTextField("Default Input", 20);
panel.add(textField);
```

  - You can use the setText method to change the text later.

```
textField.setText("Hello");
```

  - You can use the getText method to get the text from the field. Use the trim method to get rid of leading / trailing spaces.

```
String text = textField.getText()
```

Swing Text Components

- JLabel - not editable text

```
JLabel firstNameLabel = new JLabel("First Name");
```

- JPasswordField - hides typed characters

```
JPasswordField passField = new JPasswordField(8);
char[] password = passField.getPassword();
passField.setEchoChar('*');
```

- JTextArea - multi-line text entry and/or display

```
JTextArea commentArea = new JTextArea(2,30);
commentArea.setText(comment);
String x= commentArea.getText();
```

Swing Button Components

```
Jbutton okButton = new JButton("OK");
okButton.setBounds(70, 125, BUTTON_WIDTH,  BUTTON_HEIGHT);

JScrollPane scrollText= new JScrollPane(textArea);
scrollText.setBounds(50, 5, 200, 135);
contentPane.add(scrollText);

JRadioButton unixButton = new JRadioButton("Unix",true);
JRadioButton winButton = new JRadioButton("Window",false);

JCheckBox Box1 = JCheckBox("Java",true));
JCheckBox Box2 = new JCheckBox("Perl",false));
```

Main Point

Swing classes are of two kinds: *components* and *containers*. A screen is created by creating components (like buttons, textfields, labels) and arranging them in one or more containers. ***Components and containers are analgous to the manifest and unmanifest fields of life; manifest existence, in the form of individual expressions, lives and moves within the unbounded container of pure existence.***

Key to *Interactive* User Interfaces: Events

- An event is an object that represents an action:
  - user clicks the mouse
  - user presses a key on the keyboard
  - user closes a window
- In Swing, objects add or implement *listeners* for events.
  - Listeners are *interfaces*.
  - Interfaces are not classes: They define functionality that other classes implement.
  - It's a contract that certain functionality will be provided.

How to Implement an Event Handler

Every event handler requires three pieces of code:

1. declaration of the event handler class that implements a listener interface

```
public class MyClass implements ActionListener
```

2. registration of an instance of the event handler class as a listener

```
someComponent.addActionListener(instanceOfMyClass);
```

3. providing code that implements the methods in the listener interface in the event handler class

```
public void actionPerformed(ActionEvent e) {
   //code that reacts to the action...
}
```

// Implementing Event Listener actionPerformed method in the current class

```
public class ActionListenerDemo implements ActionListener {
    JButton button;

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

    public void go() {
        JFrame frame = new JFrame();
        button = new JButton("Click");
        frame.getContentPane().add(button);
        button.addActionListener(this);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 200);
        frame.setTitle("Click Demo");
        frame.setVisible(true);
    }

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

Implementing Events in Inner class

```
public class EventSourceDemo extends JFrame {

    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;

    private JLabel label;
    private JButton button;
    private JLabel label1;
    private JButton button1;
```

```
    public EventSourceDemo() {
        setLayout(new FlowLayout());
        button = new JButton( text: "Click to Get Text");
        add(button);
        label = new JLabel( text: "");
        add(label);
        button1 = new JButton( text: "Click to Clear Text");
        add(button1);
        // User defined class to handle events
        MyListener1 listener1 = new MyListener1();
        button.addActionListener(listener1);
        MyListener2 listener2 = new MyListener2();
        button1.addActionListener(listener2);
    }
```

```
    //Inner Class
    class MyListener1 implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e) {
            label.setText("Now you can see the text of Label");
        }
    }
```

```
    class MyListener2 implements ActionListener {

        @Override
        public void actionPerformed(ActionEvent e1) {
            label.setText(" ");
            setBackground(Color.PINK);
        }
    }
```

```
    public static void main(String[] args) {
        EventSourceDemo frame = new EventSourceDemo();
        frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        frame.setLocationByPlatform(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("First Event Frame");
        frame.setVisible(true);
    }
}
```

Implementing in Anonymous Inner class

```
public EventAnonymousClassDemo() {
    setLayout(new FlowLayout());
    button = new JButton( text: "Click to Get Text");
    add(button);
    label = new JLabel( text: "");
    add(label);
    button1 = new JButton( text: "Click to Clear Text");
    add(button1);

    button.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            label.setText("Now you can see the text of Label");
        }
    });

    button1.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            label.setText(" ");
            setBackground(Color.PINK);
        }
    });
}
```

Displaying Pop-up Messages

The Swing class `JOptionPane` makes it easy to pop up a standard dialog box that prompts users for a value or informs them of something (such as error messages). See the Java API docs for all the different options in using this class. We focus on one common usage here:

Example: In our example, we will add one more piece of functionality. When the user types in the word "error" in the text box, the GUI will respond by displaying a popup with an error message:

Model-View-Controller Design Pattern

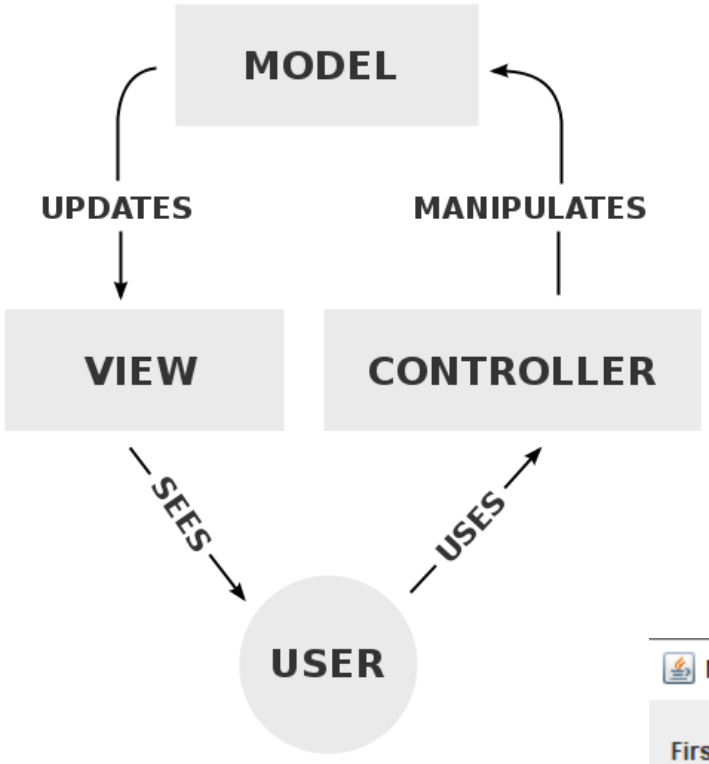
- To better understand Swing UI components, it is a good idea to review how Swing works at an architectural level.
- All Swing components (e.g. buttons, check boxes, text fields, etc) have three basic characteristics:
  - Its contents (Model)

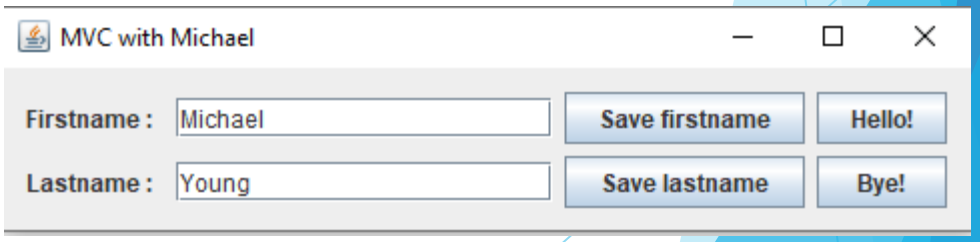
    such as the state of a button (pushed in or not), the text in a text field
  - Its visual appearance(View-Display Contents)

    color, size, etc.
  - Its behavior(Controller – Handles User Inputs)

    reaction to events
- To implement these characteristics, Swing uses a well-known OO-design pattern - the Model-View-Controller pattern.

MVC Example

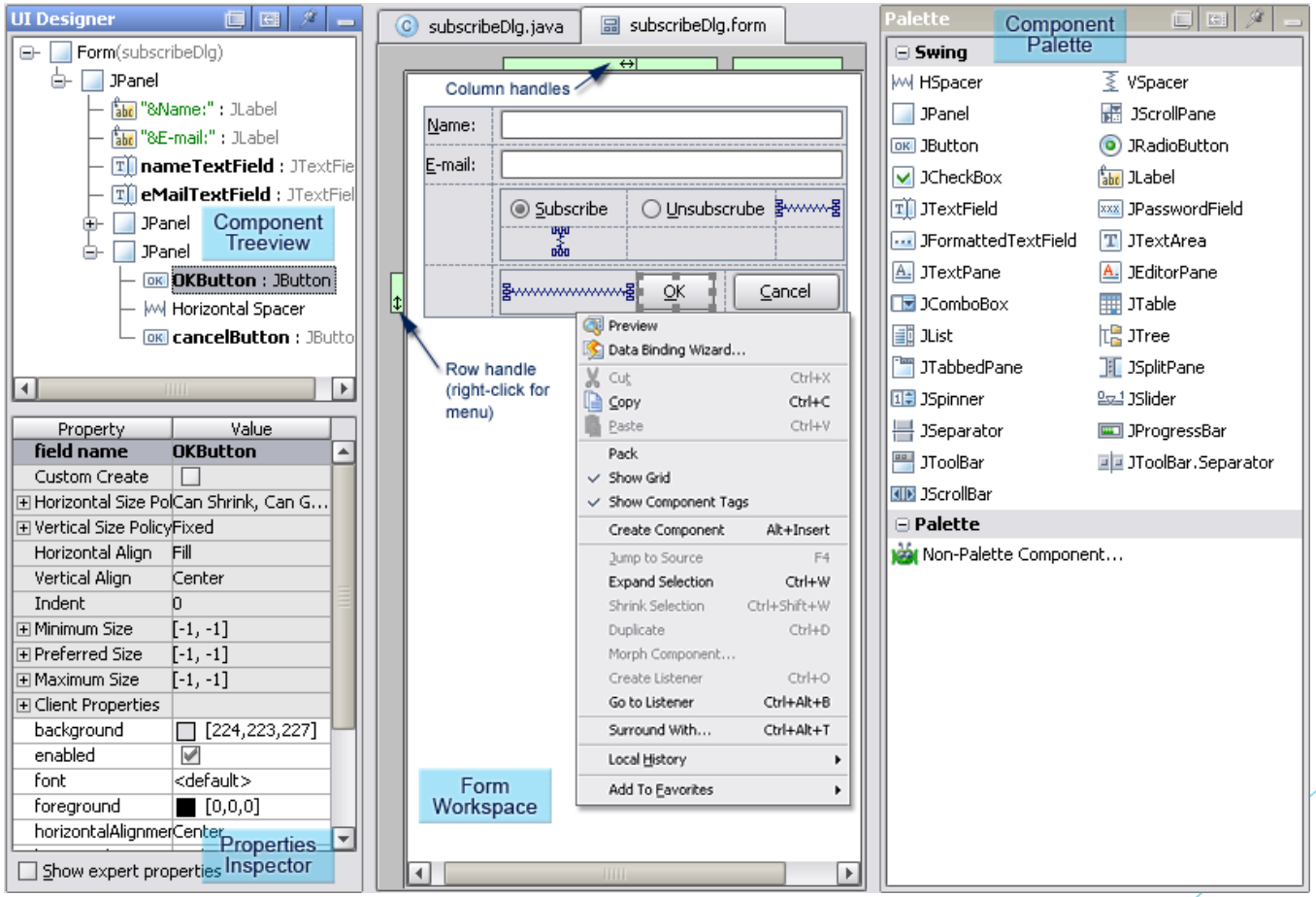

MODEL
UPDATES
MANIPULATES
VIEW
CONTROLLER
SEES
USES
USER


MVC with Michael
Firstname :
Michael
Save firstname
Hello!
Lastname :
Young
Save lastname
Bye!

Events and Listeners

Event	Listener	Example
ActionEvent	ActionListener	Button Pressed
AdjustmentEvent	AdjustmentListener	Move a scrollbar
FocusEvent	FocusListener	Tab into a textarea
ItemEvent	ItemListener	Checkbox checked
KeyEvent	KeyListener	Keystroke occurred in a component
MouseEvent	MouseListener	Mouse button click
MouseEvent	MouseMotionListener	Mouse moves or drags
TextEvent	TextListener	A text's component text changed
WindowEvent	WindowListener	Window was closed

Visual Designer



Main Point

A GUI becomes responsive to user interaction (for example, button clicks and mouse clicks) through Swing's event-handling model in which event sources are associated with listener classes, whose actionPerformed method is called (and is passed an event object) whenever a relevant action occurs. To make use of this event-handling model, the developer defines a listener class, implements actionPerformed, and, when defining an event source (like a button), registers the listener class with this event source component. ***The "observer" pattern that is used in Swing mirrors the fact that in creation, the influence of every action is felt everywhere; existence is a field of infinite correlation; every behavior is "listented to" throughout creation.***

Summary

Development in Swing requires knowledge of three areas:

1. ***Containers and Components***. The elements that a user makes use of to interact with a UI – like buttons, textfields, etc – are *components*, which are arranged in Swing *containers*.

2. ***Layout Managers***. Design of a UI first requires the developer to visualize, and sketch out, the desired appearance of windows. This design is translated into Swing components and containers by skillful use of LayoutManagers, which provide rules that determine dimensions and positions of components on the window

3. ***Event-Handling***. The functionality of a UI – by which a user can initiate an action to obtain a response – is achieved in Swing with *listeners*. Typically on a UI, *ActionListeners*, which are implemented with event-handling code, are attached to components. The event-handling mechanism of Java translates user actions into events that causes the ActionListener code to execute.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

The self-referral dynamics

arising from the reflexive association of container classes

1. In Swing, components are placed and arranged in container classes for attractive display.
2. In Swing, containers are also considered to be components; this makes it possible to place and arrange container classes inside other container classes. These self-referral dynamics support a ~~much broader range of possibilities in the design of GUIs.~~

3. **Transcendental Consciousness:** TC is the self-referral field of all possibilities.
4. **Wholeness moving within itself**: In Unity Consciousness, all activity is appreciated as the self-referral dynamics of one's own