# JAC444 - Introduction to Java for C++ Programmers

Lesson 6: GUI Programming

## Agenda

- Java GUI API
- Swing Classes
- Layout Managers
- Event-driving Programming

## About the JFC and Swing

- The Java Foundation Classes (JFC) are a graphical framework for building portable Java-based graphical user interfaces (GUIs).

- JFC consists of the
  - Abstract Window Toolkit (AWT)
  - Swing
  - Java 2D.

# AWT vs Swing

- AWT GUI components are replaced by more versatile and stable Swing GUI.
  - AWT components are referred to as heavyweight components.

- There is a equivalent Swing component for most AWT components.
  - Swing components are named JXxx.
    - JFrame, JPanel, JApplet, JDialog, JButton, etc.
  - Mixing AWT and Swing is doomed

# The Java GUI API

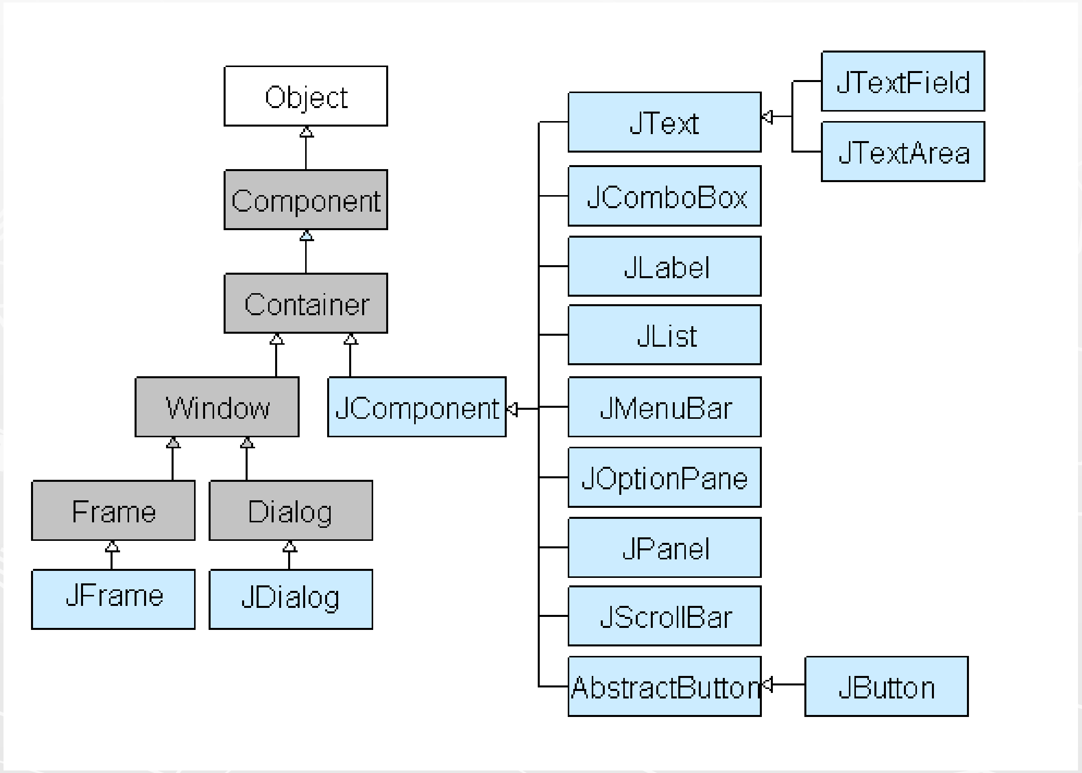
- GUI API is classified into 3 groups:
  - Component Classes, Container Classes and Helper Classes
- Component Classes
  - An instance of Component can be displayed on the screen.
  - Component is the root class of all the UI classes.
  - JComponent is the root class of all the lightweight Swing components.
    - JComponent and its subclasses are in **javax.swing** package.
  - JComponent classes includes:
    - Jbutton, Jlabel, JTextField, JTextArea, JRadioButton, JComboBox, …

# The Java GUI API

- Container Classes
  - An instance of Container can hold instances of Component.
  - Container classes that work with Swing components:
    - Container, JFrame, JPanel, JApplet, JDialog.
- GUI Helper Classes
  - Helper classes are used to describe the properties of GUI components.
  - Helper classes are in the java.awt package. e.g.
    - Graphics, Color, Font, FontMetrics, Dimension and
    - LayoutManager

# Swing Hierarchy

- The Swing library is built on top of the AWT.
- All components in Swing are Jcomponents
  - Jcomponents can be added to windows like JFrames or JDialogs.



# Frame

- A frame (JFrame) is one of the top-level containers (Window, JFrame, Frame, JDialog, Dialog and Applet).
- Unlike other Swing components, we have to specifically set a JFrame to visible. e.g.
  - `jp.setVisible(true); // jp is a container obj`
- Closing the window (using the "X" button of the title bar for example) will hide the frame, but it will not terminate the program. e.g.
  - `jp.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //` will make the program terminate when the window is closed. `jp is a container obj.`

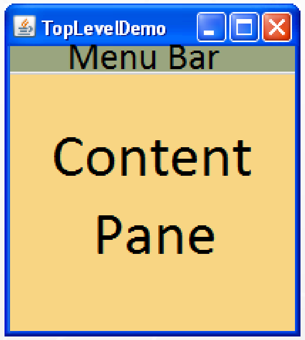
## Creating a JFrame

- creates an instance of JFrame ()
  - e.g. – JButtonDemo.java, JFrameDemo3.java

- e.g. – extends JFrame
  - e.g. –
    - JFrameDemo7.java, JFrameDemo7B.java, JTextDemo_V1.java Reservation_V1.java, Reservation_V2.java, Reservation_V17.java

# Content Pane and Panel

- Content Pane: each top-level container may consist of 2 areas:
  - a Menu Bar
  - a content pane



- A panel (JPanel) in a general-purpose container.
  - Used as subcontainers to group GUI components.

## Layout Manager

- Each container contains a layout manager.
  - e.g.

    JPanel jp = new JPanel();

    jp.setLayout( *aLayoutManagerObject);*
- Some basic layout managers:
  - BorderLayout - default layout manager for JFrame
    - BorderLayout.*NORTH,* BorderLayout.*EAST,* BorderLayout.*CENTER,* BorderLayout.*WEST,* BorderLayout.*SOUTH*
    - *E.g.*

```
jp.setLayout( new BorderLayout() );
jp.add( new JButton( "subscribe" ), BorderLayout.NORTH );
```

- FlowLayout
  - Add components to container from left to right
  - Alignment control:
    - FlowLayout.LEFT, FlowLayout.CENTER (default), FlowLayout.RIGHT

e.g.
```
jp.setLayout( new FlowLayout( FlowLayout.RIGHT, 5, 20 ) );
    jp.add( new JButton( "subscribe" ) );
```

- GridLayout
  - Arrange components in a grid (matrix) formation.
    - E.g.

```
jp.setLayout(new GridLayout(3, 3));
    jp.add( new JButton( "subscribe" ) );
```

- BoxLayout
  - javax.swing.Box.*createHorizontalBox()*
  - javax.swing.Box.*createVerticalBox()*
  - javax.swing.Box.*createHorizontalStrut(10))*

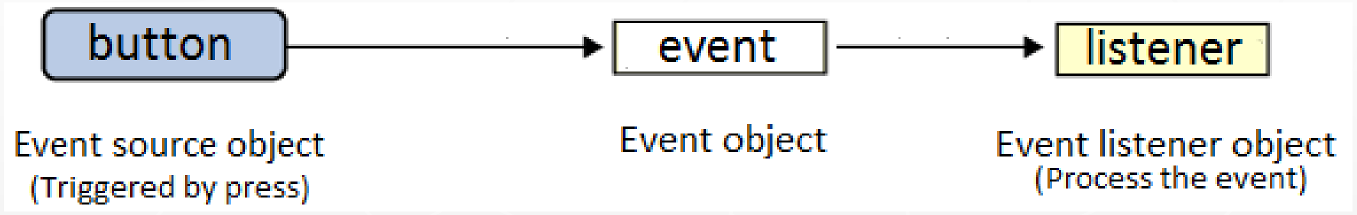
## Examples

- BorderLayoutManager.java
- FlowLayoutManager.java
- GridLayoutManager.java

- BoxLayoutDemo.java
- BoxLayoutDemo2.java
- BoxLayoutDemo3.java

# Event-driven Programming

- The simple event model



- To be a listener of an action event:
  - The object must be an instance of the ActionListener interface.
  - The object, listener, must be registered with the event source object:
    - e.g. source.addActionListener(listener)

# Event Types, Listener Interface, Methods, User Action

| Event Types | Listener Interface | Listener Interface Methods | Source Object & Events |
|---|---|---|---|
| ActionEvent | ActionListener | actionPerformed(ActionEvent e) | JButton, clicked; JTextField, Enter pressed; JComboBox, item Selected; JRadioButton, (un)checked; JCheckBox, (un)checked; |
| ItemEvent | ItemListener | itemStateChanged(ItemEvent e) | JComboBox, item Selected; JRadioButton, (un)checked; JCheckBox, (un)checked; |
| MouseEvent | MouseListener | `mouseClicked`(MouseEvent e)<br>`mousePressed`(MouseEvent e)<br>`mouseReleased`(MouseEvent e)<br>`mouseEntered`(MouseEvent e)<br>`mouseExited`(MouseEvent e) | Mouse: pressed, released, clicked, entered, exited |
| | MouseMotionListener | `mouseDragged`(MouseEvent e)<br>`mouseMoved`(MouseEvent e) | Mouse: moved, draged |
| KeyEvent | KeyListener | `keyPressed`(MouseEvent e)<br>`keyReleased`(MouseEvent e)<br>`keyTyped`(MouseEvent e) | Key: pressed, released, typed |

## Coding Listener Classes

- Based on the event type (e.g. ActionEvent), create the event listener class by implementing appropriate listener interface (e.g. ActionListener)

```
Class OKListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // TODO …
     }
}
```

- A listener can listen more than one events of the same event type:

```
class RadioHandler implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        if ( (e.getSource() == jrb1)
                && (e.getStateChange() == ItemEvent.SELECTED) )
                        info[0] = "VIP";
        else if ( (e.getSource() == jrb2)
                && (e.getStateChange() == ItemEvent.SELECTED) )
                        info[0] = ((JRadioButton) e.getSource()).getText(); // if this
way, no need to check which button is selected!
        }
}
```

## Example Code

- Example of a JButton object fires ActionEvent:

```
// Create source object
JButton jbtOK = new Jbutton(“OK”);

// create listener object
OKListenerClass listener1 = new OKListenerClass();

// register listener
jbtOK.addActionListener(listener1);
```

## Creating Listener Classes

- Top-level classes
- Inner Class Listeners
- Anonymous (inner) Class Listeners

# Examples

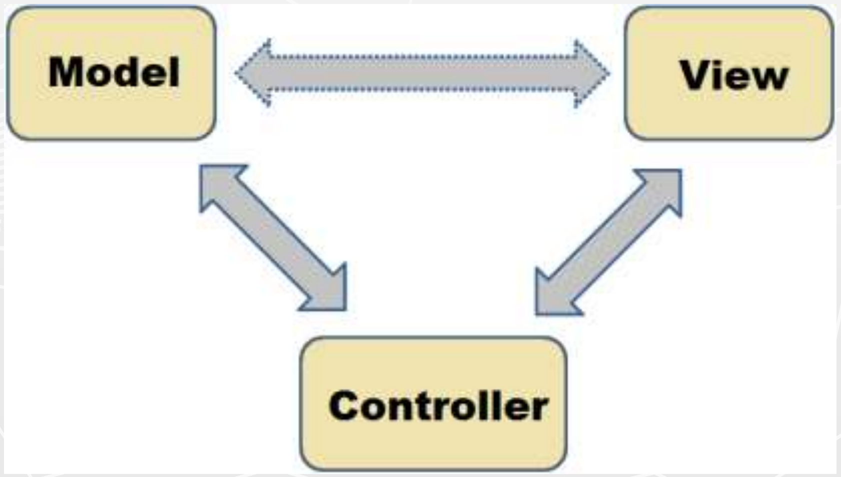
- creates an instance of JFrame ()
  - Event handlers are top-level classes (passing or not passing message.)
    - e.g. JButtonDemo.java, JFrameDemo3.java
- extends JFrame
  - Event handlers are named inner classes
    - JFrameDemo7.java, JFrameDemo7B.java, Reservation_V17.java
  - Event handlers are Anonymous inner classes
    - JTextDemo_V1.java
    - Reservation_V1.java, Reservation_V2.java, Reservation_V17.java

# Some JComponent classes

- Jbutton
- Jlabel
- JTextField
  - getText()
- JTextArea
  - setText(str)
- JRadioButton
  - setSelected(true), isSelected()
- JCheckBox
- JComboBox
  - setSelectedIndex(index), getSelectedIndex() // start from 0
  - getSelectedItem()

# Java SE Application Design With MVC

- What Is Model-View-Controller (MVC)?
  - In OOP development, model-view-controller (MVC) is the name of a methodology or design pattern for decoupling data access and business logic from the manner in which it is displayed to the user.



## Model-View-Controller (MVC)

MVC can be broken down into three elements:

- **Model** - The model represents data and the rules that govern access to and updates of this data.
- **View** - The view renders the contents of a model. It specifies exactly how the model data should be presented. If the model data changes, the view must update its presentation as needed.
  - This can be achieved by using a *push model*, in which the view registers itself with the model for change notifications, or a *pull model*, in which the view is responsible for calling the model when it needs to retrieve the most current data.
- **Controller** - The controller translates the user's interactions with the view into actions that the model will perform. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in an enterprise web application, they appear as GET and POSTHTTP requests.

## Resourceful Links

- Model-View-Controller (MVC) Structure
- Java SE Application Design With MVC
- JavaFX

Thank You!