

Introduction to Computation and Problem Solving

Class 15: Constructing Interfaces with Swing

Prof. Steven R. Lerman
and
Dr. V. Judson Harward

1

Topics

- In this lecture, we are going to explore how to build more complex interfaces with Swing.
- We will not demonstrate many components; for that see <http://java.sun.com/docs/books/tutorial/uiswing/components/components.html>
- We will explore
 - how to build up an interface component by component and container by container;
 - how to use layout management to configure your GUI
- Tuesday's active learning will focus on how to make your program respond to events.

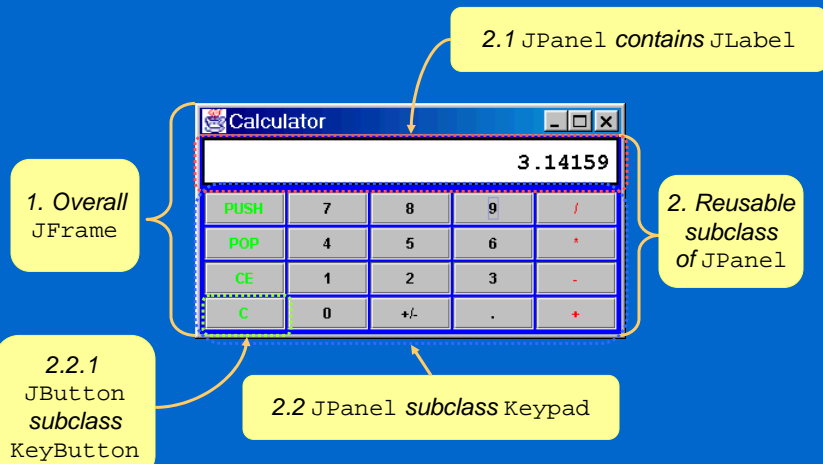
2

Constructing GUIs

- Later in this course, we are going to build a simple calculator.
- We will build the GUI by nesting simpler components into containers, and then combining the containers into larger containers.
- `JPanel` is the workhorse of most complicated interfaces. It is
 - a good all purpose container
 - the standard drawing surface (Lecture 17)
 - a base class for many composite components

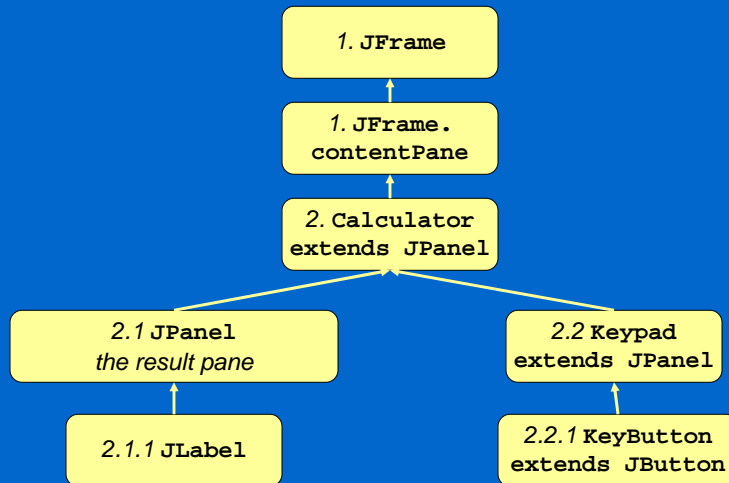
3

Calculator GUI



4

Calculator GUI Diagram



5

Layout Management, 1

- **Layout management is the process of determining the size and location of a container's components.**
- **Java containers do not handle their own layout. They delegate that task to their layout manager, an instance of another class.**
- **Each type (class) of layout manager enforces a different *layout policy*.**
- **If you do not like a container's default layout manager, you can change it.**

```
Container content = getContentPane();
content.setLayout( new FlowLayout() );
```

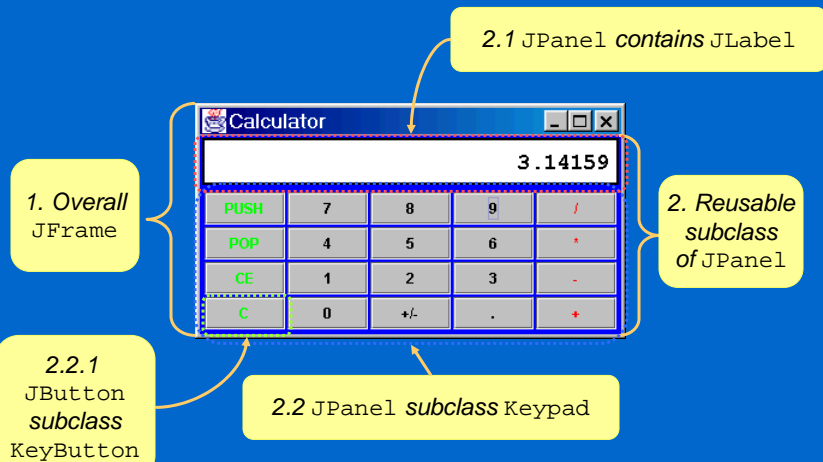
6

Layout Management, 2

- Layout management proceeds top down in the containment hierarchy.
- If a container encloses another container, the enclosing container can not position the inner container nor size itself until it knows how big the inner container needs to be.
- And the inner container can not size itself until it queries its contents.

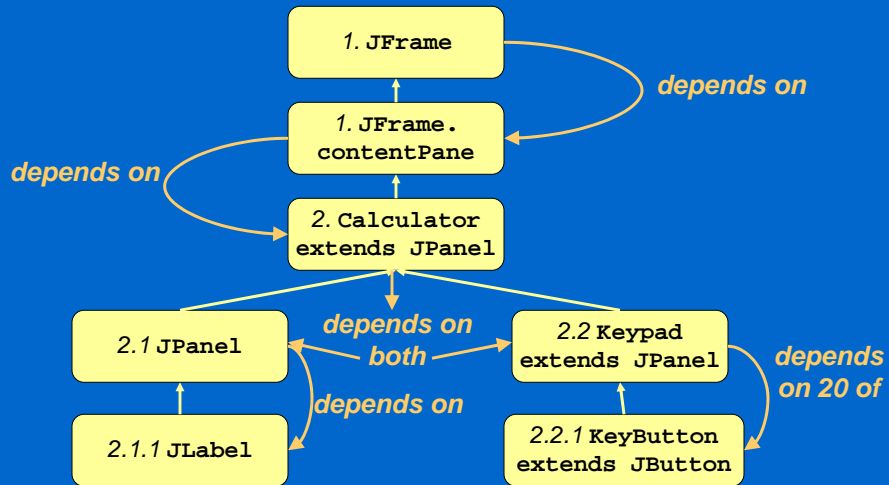
7

Calculator GUI



8

Calculator GUI Diagram



9

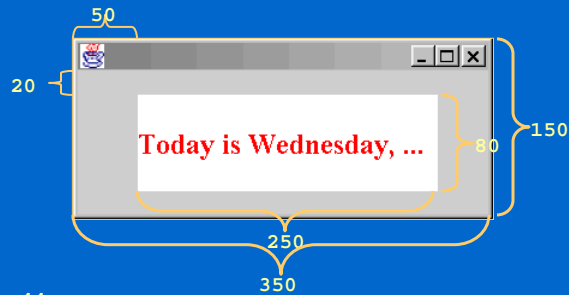
Turning off Layout Management

- Although layout managers can introduce additional complexity, they are there for a very good reason.
- The effect of layout management can be turned off by eliminating a container's layout manager via a call to `setLayout(null)`.
- The user must then explicitly lay out each component in absolute pixel coordinates through calls to `setSize()` and `setLocation()` or a single call to `setBounds()`.
- The problem with this strategy is that it lacks flexibility.

10

Today Without Layout Management

```
public Today1a( String dStr ) {  
    . . .  
    getContentPane().setLayout( null );  
    getContentPane().add( dateLabel );  
    dateLabel.setBounds( 50, 20, 250, 80 );  
    setSize( 350, 150 );  
}
```

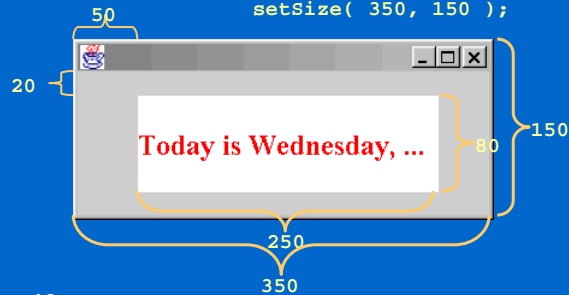


11

Coordinates

- Measured in pixels (e.g. 640 by 480, 1024 by 768, etc.)
- Upper left hand corner is origin (0,0)
- X axis goes from left to right, y from top to bottom
- Each component is anchored in its parent's coordinate system.

```
dateLabel.setBounds( 50, 20, 250, 80 );  
setSize( 350, 150 );
```



12

3 Good Reasons to Use Layout Management

1. Often you do not know how large your application will be. Even if you call `setSize()`, the user can still physically resize the window of an application.
2. Java knows better than you how large components should be. It is hard to gauge the size of a `JLabel`, for instance, except by trial and error. And if you get the size correct on one system and then run it on another with a different set of fonts, the `JLabel` will not be correctly sized.
3. Once you lay out a GUI, you may want to make changes that will compromise a layout done by hand. If you use layout management, the new layout happens automatically, but if you are laying out the buttons by hand, you have an annoying task ahead of you.

13

Layout Managers

- There used to be seven layout manager classes supplied by Swing. They ranged from the simple `FlowLayout` to the flexible but at times frustrating `GridBagLayout`.
- Now each release contains new layout managers.
- Each manager class implements a particular layout *policy*.
- Swing containers have default layout managers. A `JFrame` content pane uses `BorderLayout` and a `JPanel` uses `FlowLayout`.

14

FlowLayout

- The `FlowLayout`, the simplest of the managers, simply adds components left to right until it can fit no more within its container's width.
- It then starts a second line of components, fills that, starts a third, etc.
- Each line is centered within the enclosing container.
- `FlowLayout` respects each component's preferred size and will use it to override a size set by `setSize()`.

15

FlowLayout Example, 1

```
public class Flow extends JFrame {
    private Font labelFont;
    private Border labelBorder;

    public Flow( ) {
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        labelFont = new Font( "SansSerif",Font.BOLD,24 );
        labelBorder =
            BorderFactory.createLineBorder(Color.red,1);
        getContentPane().setLayout( new FlowLayout() );
        setSize(200, 200 );
    }
}
```

16

FlowLayout Example, 2

```
public void addLabel( String labelStr ) {
    JLabel label = new JLabel( labelStr );
    label.setFont( labelFont );
    label.setBorder( labelBorder );
    getContentPane().add( label );
}

public static void main (String args[])
{
    Flow flow = new Flow();
    flow.addLabel( "one" );
    . . .
    flow.setVisible( true );
}
```

17

BoxLayout

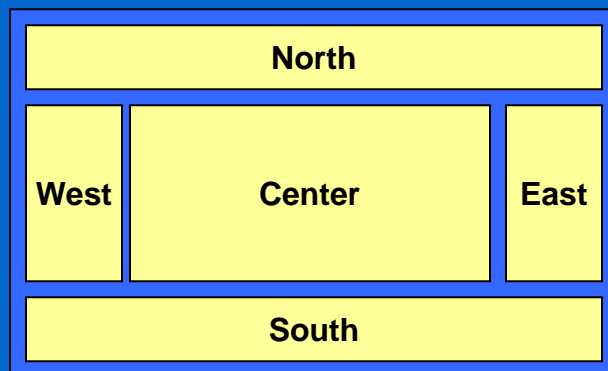
- The `BoxLayout` is the new fancier version of the `FlowLayout`.
- It allows the use of "glue" and "spacers" to separate components as well as more varied alignment.

```
public void addLabel( String labelStr, float align ) {
    JLabel label = new JLabel( labelStr ); ...
    label.setAlignmentX( align );
    getContentPane().add( label );
}

public static void main (String args[]) {
    BoxL box = new BoxL();
    box.addLabel( "one", 0F );
    box.addLabel( "two", 1F );
    box.addLabel( "three", 0.5F );
    box.getContentPane().add( Box.createVerticalGlue() );
    . . .
}
```

18

BorderLayout Zones



19

BorderLayout Policy

- You specify the zone via a second `String` argument in the `add()` method. For example, the following line of code adds the button labeled "DoIt" to the middle of a container.

```
add( new JButton( "DoIt" ), "Center" );  
// "Center" == BorderLayout.CENTER
```
- A `BorderLayout` may horizontally stretch its North and South components (if they exist), vertically stretch its East and West components, and stretch the Center component both ways to accommodate its container's size and the constraints of its other four sectors.
- This can be useful. If you put a `JPanel` in the Center zone of a container managed by a `BorderLayout`, the manager will always resize the `JPanel` to take up all extra space, which is usually what you want if you are using it as a drawing surface.

20

Grid Layout

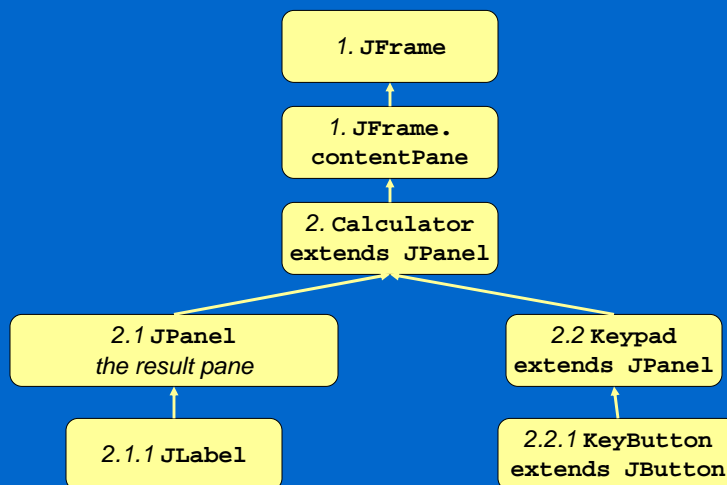
- The `GridLayout` class is a layout manager that lays out a container's components in a rectangular grid.
- The container is divided into equal-sized rectangles, and one component is placed in each rectangle.
- In the normal constructor you specify the number of rows or columns but not both. The one that is not zero has a fixed number of elements; the other grows as you add components.

```
getContentPane().setLayout( new GridLayout(0,2));
```

would set a `JFrame`'s layout to a two column grid. The number of rows would depend on the number of added components.

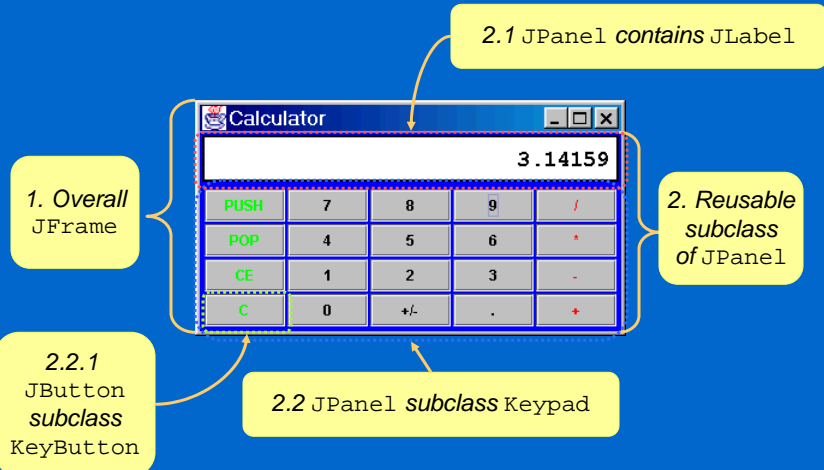
21

Calculator GUI Diagram



22

Calculator GUI



23

Building the Calculator, 1

```
public class CalculatorApp {  
    public static void main( String[] args ) {  
        JFrame top = new JFrame( "Calculator" );  
        top.setDefaultCloseOperation(  
            JFrame.EXIT_ON_CLOSE );  
        Calculator calc = new Calculator();  
        top.getContentPane().add( calc,  
            BorderLayout.CENTER );  
        top.pack();  
        top.setVisible(true);  
    }  
}
```

24

Building the Calculator, 2

```
public class Calculator
  extends javax.swing.JPanel {
    private JLabel display = null;
    static final String EMPTYSTR = " ";
    public Calculator() {
      setLayout( new BorderLayout(4, 4) );
      JPanel displayPanel = new JPanel();
      displayPanel.setLayout( new FlowLayout(
        FlowLayout.RIGHT ));
      display = new JLabel( EMPTYSTR );
      displayPanel.add( display );
      add( displayPanel, BorderLayout.NORTH );
      add( new Keypad(), BorderLayout.CENTER );
    }
  }
```

2.1.1

2.1

2.2

25

Why Make a Component an Instance Member?

- `JLabel display` is an instance member but the panel that contains it (`JPanel displayPanel`) isn't.
- We make components members when we will need to refer to them, usually to change them, after they are create in the constructor.
- `JLabel display` is going to hold the results of calculations. We are going to need a reference so that we can update the display.

26

Building the Calculator, 3

2.2

```
class Keypad extends JPanel {
    Keypad() {
        setLayout( new GridLayout( 0, 5, 2, 2 ));
        setBackground( Color.blue );
        . . .
        add( new KeyButton( . . . ));
        . . .
    }
}
class KeyButton extends JButton { . . . }
```

2.2.1

27

Setting a Component's Size

- Most components know how to size themselves. For instance, a button sets its default size to accommodate its image and/or its label and font size.
- If you want to override the default size, you might think that using the explicit `setSize()` method is the way to do it.
- The problem with `setSize()` is that the chosen size will only last until the Java environment lays the container out again (or for the first time).
- A much better strategy is to override the component's method(s) that it uses to communicate its preferred size to its layout manager.

28

Setting a Component's Size, 2

- Components communicate their layout needs to their enclosing container's layout manager via the methods:
 - `public Dimension getMinimumSize()`
 - `public Dimension getPreferredSize()`
 - `public Dimension getMaximumSize()`
- There are three corresponding set methods that allow you to change a component's *size hints*.
 - `public Dimension setMinimumSize(Dimension d)`
 - `Dimension d = new Dimension(int width, int height)`
 - `public Dimension setPreferredSize(Dimension d)`
 - `public Dimension setMaximumSize(Dimension d)`

29

The TwoSize Program

```
public class TwoSize extends JFrame {
    public TwoSize() {
        setDefaultCloseOperation( EXIT_ON_CLOSE );

        JPanel center = new JPanel();
        Font labelFont =
            new Font( "SansSerif", Font.BOLD, 18 );

        JLabel small = new JLabel( "small",
                                   SwingConstants.CENTER );
        small.setFont( labelFont );
        small.setBackground( Color.yellow );
        small.setOpaque( true );
        small.setSize( 300, 100 );
    }
}
```

30

The TwoSize Program, 2

```
JLabel big = new JLabel( "big",  
                        SwingConstants.CENTER );  
big.setFont( labelFont );  
big.setBackground( Color.yellow );  
big.setOpaque( true );  
big.setPreferredSize( new Dimension( 300, 100 ) );  
  
center.add( small ); center.add( big );  
getContentPane().add( center, BorderLayout.CENTER );  
pack();  
}  
  
public static void main( String [] args ) {  
    TwoSize ts = new TwoSize();  
    ts.setVisible( true );  
}  
}
```

31

When does Layout Occur?

Swing will automatically (re)layout a GUI

1. when it first becomes visible,
2. when a component changes its size because the user physically changes its size (by resizing the window) or because the contents have changed (for instance, changing a label) .

32

revalidate() and pack()

- But if you change a component's size once it is visible by changing one of its size hints or if you add a new component, a new layout of the parent will not be triggered until you call `revalidate()` on the *component* (not the container).
- The only other time when you must explicitly call for a layout is when you use a `JFrame`. A frame won't fit itself to its content unless you call `pack()` on it. You can also set its size, using `setSize()`, but frames do not have size hints. If you don't call `pack()` or `setSize()`, the frame will shrivel up, and you will have a hard time finding it on your screen.

33

The AddButton Program, 1

```
public class AddButton extends JFrame
    implements ActionListener {
    private JButton button;
    private int count = 0;
    private Font bigFont;

    public AddButton() {
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        bigFont= new Font( "SansSerif", Font.BOLD, 24 );

        button = new JButton( "Push Me" );
        button.setFont( bigFont );
        button.addActionListener( this );
        getContentPane().setLayout( new FlowLayout() );
        getContentPane().add( button );
        setSize( 600, 100 );
    }
}
```

34

The AddButton Program, 2

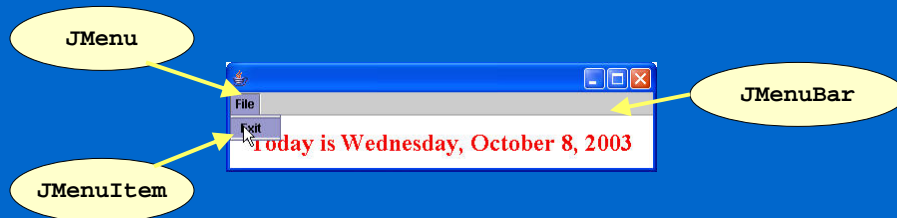
```
public void actionPerformed( ActionEvent e ) {
    JButton newButton = new JButton(
        String.valueOf( ++count )
    );
    newButton.setFont( bigFont );
    getContentPane().add( newButton );
    //newButton.revalidate();
}

public static void main (String args[])
{
    AddButton app = new AddButton();
    app.setVisible( true );
}
}
```

35

Menus

Not everything you will want to use lives in the content pane:



36

TodayWithMenu

```
public TodayWithMenu( String dStr ) {
    . . .
    JMenuBar menubar = new JMenuBar();
    setJMenuBar( menubar );
    JMenu file = new JMenu( "File" );
    menubar.add( file );
    JMenuItem exit = new JMenuItem( "Exit" );
    exit.addActionListener( this );
    file.add( exit );
    . . .
}
```

37

Components or Dialogs?

Sometimes you would prefer to use a popup version of a component rather than have it take up screen real estate all the time.

Remember `FavoriteColor` from last lecture:

```
public void actionPerformed((ActionEvent e) {
    Color c = JColorChooser.showDialog( this,
        "Try a New Color", currentColor );
    if ( c != null ) {
        currentColor = c;
        colorLabel.changeColor( currentColor );
    }
}
```

38

Event Driven Programming

- In event-driven programming the user controls program execution. The operating system (Windows, JVM):
 - Monitors keystroke, mouse, other I/O events from input devices
 - Dispatches event messages to programs that need to know
 - Each program decides what to do when an event occurs
- This is the reverse of console-oriented programming, where the program asks the operating system (OS) to get input when it wants it
- **Event sources:** menus, buttons, scrollbars, etc.
 - Have methods allowing event listeners to register with them
 - When an event occurs, source sends message (an event object) to all registered listener objects
- **Event listeners:** objects in your program that respond to events
 - Event delegation allows programmer to pick the object