CS416

Introduction to Computer Science II

Spring 2018

5 Graphical User Interfaces

- Previously, in 416
  - Java event model
  - Basic event-based programming in Java
  - Simple time-based animation

Preview

- Principles of user-centered design
- Swing/AWT classes
    - windows, pushbuttons, radio buttons
- Component Layout
- Mouse input handling

User-Centered Design

- A user interface should work for the user
    - usable, functional, safe, efficient for the user, and easy to learn
- How do we create a user-centered design?
    - Same as any software design: analysis, design, implementation, testing, maintenance

UI Analysis Phase

- Analysis for UI includes user analysis
    - user information: who are they, what do they know, what is their computer experience
    - how will UI be used:
        - occasionally by each user,
        - all the time?
    - how is the task being done now?
        - more likelihood of success if transition is easy
        - can the user keep the same mental *model* of the process

UI Design Phase

- Consider the *application* domain
    - Need to *model* the aspects of the application domain relevant to this program
    - This is no different from any software design
- Consider the *user*
    - <u>What</u> functionality/information from the domain will be available to the user?
    - <u>How</u> should the interface provide that functionality?

Modeling the Domain

- As with any software design:
    - what objects of interest are in the domain?
    - what properties do the objects have”
    - what actions can they perform?
    - what are the relationships between objects?
    - how can we model those objects in a program?

Modeling the User Interface

- A user interface provides <u>information exchange</u>
  - how does user provide *input* to the program?
  - how does program respond to input
    - which objects provide the <u>feedback</u> (confirmation of input)?
    - which objects provide the system <u>response</u> (action requested)?
    - how is needed information provided to these objects?
  - how is all the information, feedback, response provided?
    - both <u>spatial</u> and <u>temporal</u> *layout* are important

User-Centered Design Guidelines

- Let the user be in control!
- The interface should be
  - comfortable and easy to use
  - efficient for the user
  - easy to learn and easy to remember
  - fun to use
  - unobtrusive
- Make it harder to err, rather than easier
- Follow the *Law of Least Astonishment*

Guidelines

- Applying the least astonishment law
  - means knowing your users, which might be hard
- Some guidelines are contradictory
  - efficiency for experienced users conflicts with ease of learning for novice users

Implementation/Testing Phases

- Success of a UI product is largely determined by user response to the software
- Need active and frequent involvement of users
  - involve both *novice* and *expert* users, if possible
  - generate *prototype* interfaces for users to test
  - add UI features *incrementally* to gauge user response

GUI Tools

- Basic GUI components (also called “widgets”)
  - Window, icons, menus, text, push buttons, toggle buttons, radio buttons, combo boxes, and sliders
  - Often called a WIMP GUI (Window Icon Menu Pointer)
- Layout managers
  - Make it easier for programmer to distribute multiple components in the window
- Event handling mechanism
  - How to get the widget interaction information to the application code

GUI Widgets

- *Icon* - graphical shape that represents an object
- *Menu* - list of predefined choices
  - *Menu bar* - list always visible
  - *Pull down / pop up* menus - user action shows menu
- *Text* - for output and input
- Buttons
  - *PushButton* - boolean event happens
  - *ToggleButton* - state changes between *true* and *false*
  - *RadioButton* - state changes between *n* choices
- *Slider* - allows user to choose a value within a range

Layouts

- Java provides *layout manager* classes to handle the low level details of widget layout
- Most commonly used
  - *BorderLayout*
  - *FlowLayout*
  - *GridLayout*
- These are especially powerful when the user is allowed to *resize* the window

Event Handling

- Each component (widget) is an event *source* (just like *Timer*)
  - Each maintains a list of *Listeners*
  - Each passes every event to each *Listener*
- Each *Listener* has a reference to an application *response* object
  - When it receives an event from the *source*, the *Listener* calls the appropriate method of the *response* object
- The *response* object performs the application-specific behavior

Swing/AWT GUI Tools

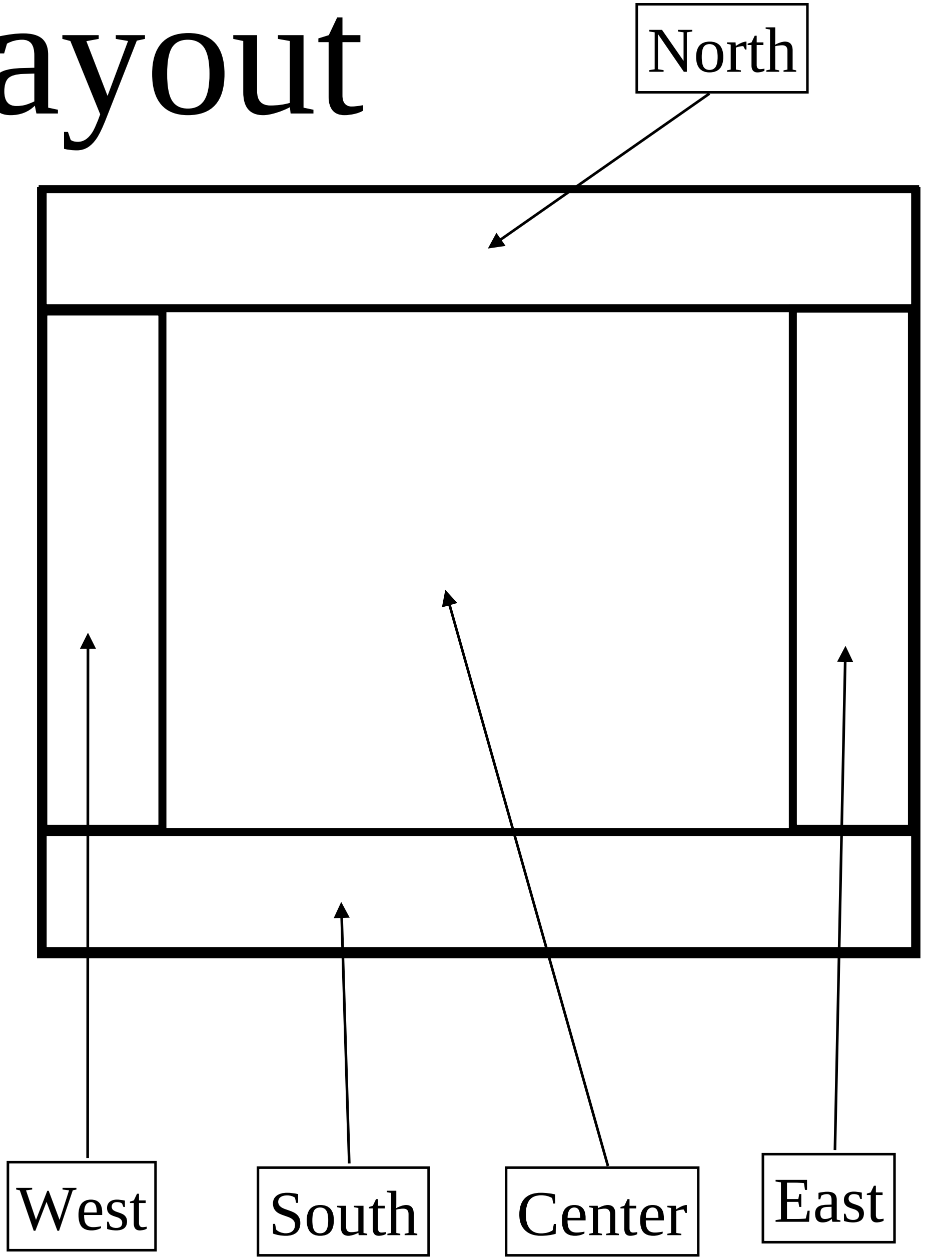
- Swing/AWT has lots of classes to support GUIs
  - Top-level containers - separately controlled windows
    - *JApplet*, *JOptionPane*, *JFrame*
  - Mid-level containers - contained in a top-level container or another mid-level container
    - *JPanel*, *JScrollPane*, *JSplitPane*, *JTabbedPane*, *JToolBar*
  - Components that can accept user input
    - *JButton*, *JComboBox*, *JList*, *JMenu*, *JRadioButtons*, *JSlider*, *JSpinner*, *JTextField*, *JPasswordField*, *JFormattedTextField*, *JTextArea*, *JColorChooser*, *JFileChooser*
  - Components that are output-only
    - *JLabel*, *JProgressBar*, *JToolTip*

Border Layout

- Default layout for *JFrame*
- 5 regions: North, East, South, West, Center
- Each region can have just 1 component
- Region is 2nd argument to *add*

```
add(comp, BorderLayout.NORTH)
add(comp, BorderLayout.SOUTH), ...
```

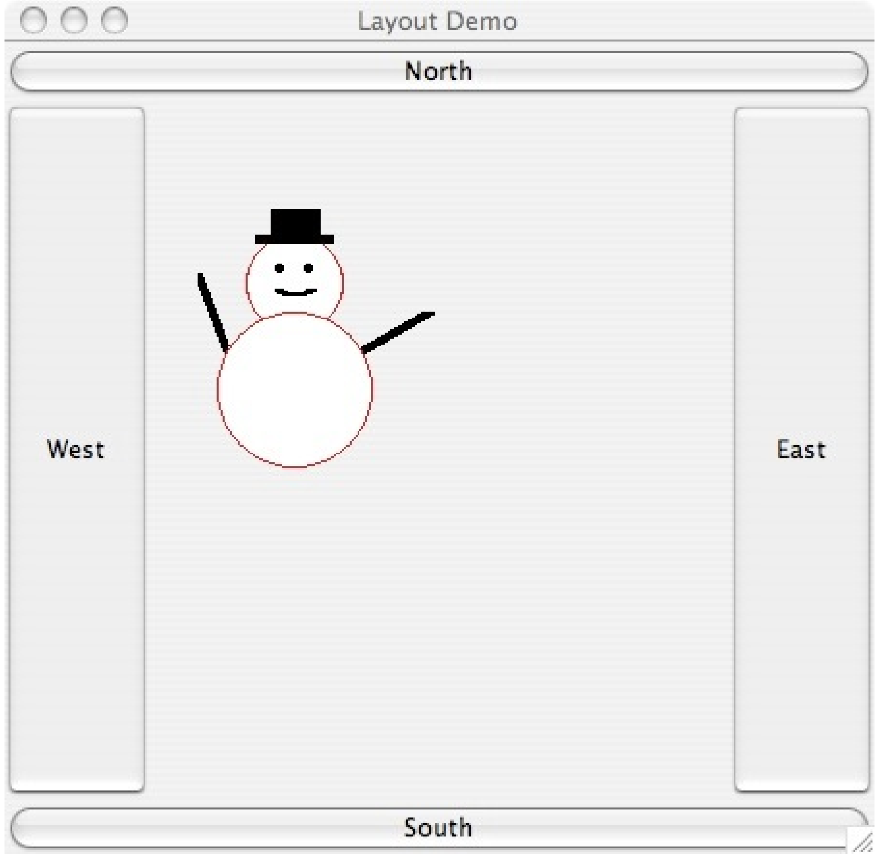
- Regions change size to “fit” what is added to them.
  - Center is lowest priority



BorderLayout Example

```
public class Demo extends JFrame
{
  public Demo()
  {
    // default layout for JFrame is
    //   BorderLayout
    ...
    add( new DrawPanel( this ));

    this.add( new JButton( "East"),
              BorderLayout.EAST );
    this.add( new JButton( "North"),
              BorderLayout.NORTH );
    this.add( new JButton( "West"),
              BorderLayout.WEST );
    this.add( new JButton( "South"),
              BorderLayout.SOUTH );
    ...
    }
  }
```

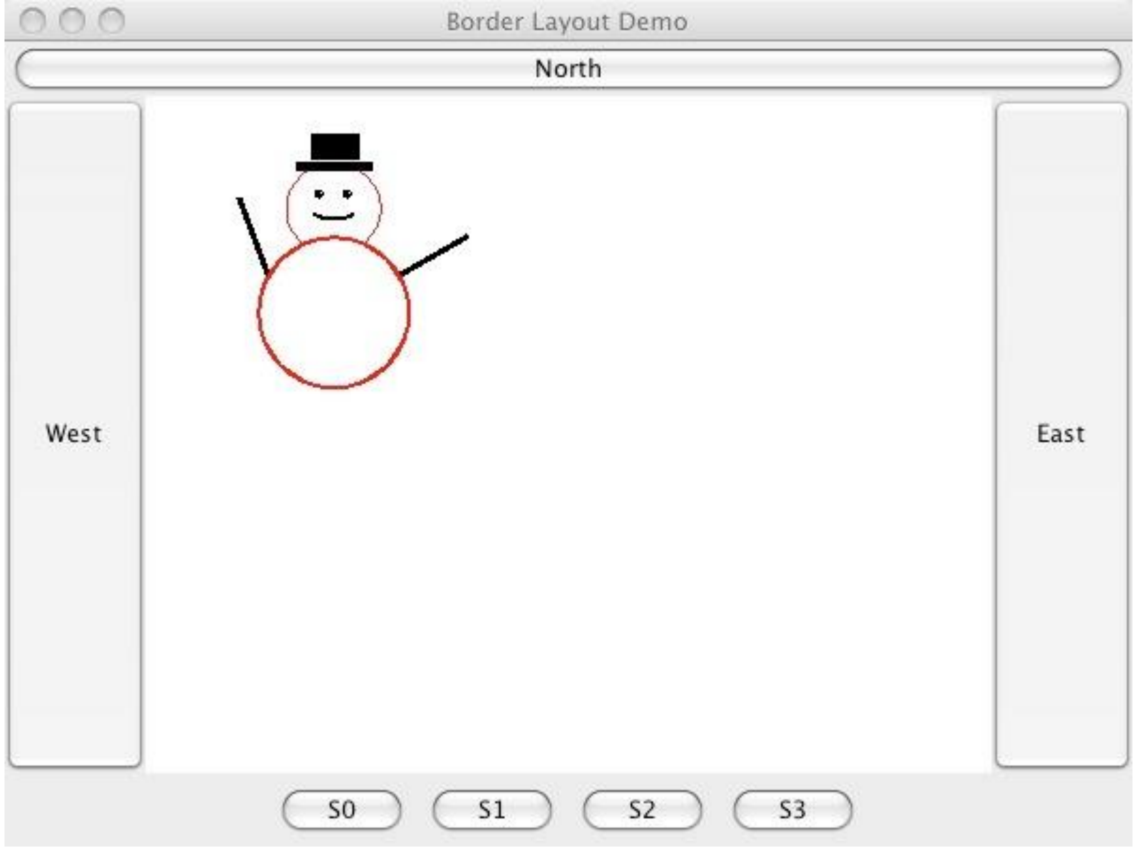


Nested Panels

- Each BorderLayout region can have exactly <u>one</u> component in it -- or at least one that is visible!
- What if we want *multiple* components in a region?
- Can *nest* panels. For example
  - Create a *JPanel*
  - Put multiple components in it
  - Put the panel into a *BorderLayout* region

Nested Panel Example

- *FlowLayout* is default layout for *JPanel*



Buttons aligned horizontally in panel

Add buttons to panel

```
...
public BorderDemo()
{
  ...
  jp = new JPanel();
  jp.add( new JButton( "S0" );
  jp.add( new JButton( "S1" );
  jp.add( new JButton( "S2" );
  jp.add( new JButton( "S3" );
  this.add( jp,
         BorderLayout.SOUTH );
  ...
}
```

Create JPanel

Add to South position of frame

JButton Event Handling

- Event handling for buttons is the same as for *Timer*
- Need an *ActionListener* added to the event *source*.

Let button know who needs to get event.

Method called when event occurs

```
...
button = new JButton( String.valueOf( i ));
button.addActionListener( new ButtonListener( i ));
...
// public inner class for event handler:
public class ButtonListener implements ActionListener
{
  int _btnId;
  public ButtonListener( int btnId )
  {
    _btnId = btnId;   // save button id for later
  }
  public void actionPerformed( ActionEvent ev )
  {
    System.out.println("Button "+_btnId+" event.");
  }
}
```

GUI Application Framework

- Adding GUI components complicates an application: let's define a revised canonical application framework
  - *DrawPanel* becomes an Application GUI class (*AppGUI)*
    - still extends *JPanel* to be used for the graphics
    - creates GUI widgets and adds them to its containing *JFrame* (passed to it by *SwingApp*) and/or to a new *JFrame*
  - *SwingApp* contains the *static main* method
    - parses command line arguments; sets associated *static* variables in either *SwingApp* (or *AppGUI*) directly or via *static* methods
    - creates the Application GUI class
  - The name *SwingApp* <u>should</u> be changed for each application; *AppGUI* might be changed

Revised *SwingApp*

```
public class SwingApp extends JFrame
{
   //----------------- class variables --------------------
   int  speed;   // travel speed, "package" access
   int  seed;    // Random variable seed, "package" access
   public SwingApp( String title, String[] args )
   {
      super( title );
      this.setSize( 600, 450 );
      this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      // here add processing of command line arguments.
    //   Should be done in a method if more than a few lines
      speed = getIntArg( args, 0, 10 );
      seed  = getIntArg( args, 1, 1 );

      AppGUI appGUI = new AppGUI( this );

      this.add( appGUI );
      this.setVisible( true );
   }
   ...
}
```

AppGUI Framework

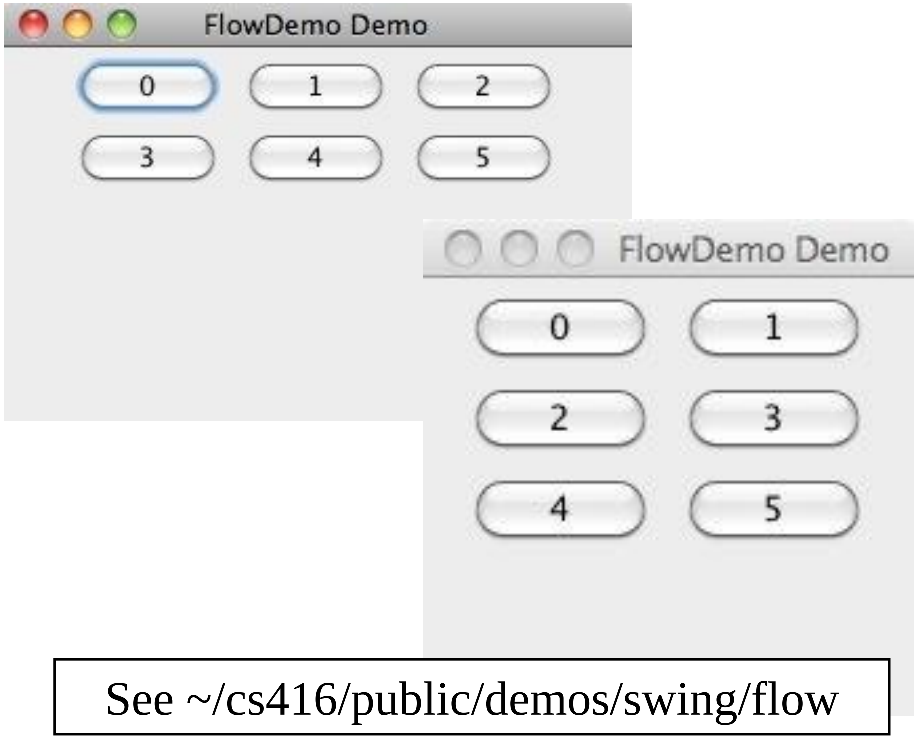
```
public class AppGUI extends JPanel
{

  //------------- instance variables ---------------------
  ...
  //--------------- constructor --------------------------
  public MoverGUI( JFrame frame )
  {
    super();
    setLayout( . . . ); // Probably BorderLayout
    buildGUI( this );  // add GUI components to this panel
    DrawPanel panel = DrawPanel( ... );
    buildDisplay( panel ); // build initial display
    ...                 //  more constructor code
  }
  ...     // other AppGUI methods
}
```

FlowLayout

- Simplest layout for multiple similar components
- Components added L to R, top to bottom in panel
- If window resized, layout re-computed
- A *pack* method may cause window to be re-sized to "fit" the components in it

```
public class FlowDemo extends JFrame
{
  public FlowDemo()
  {
    // flow is default for JPanel
    JPanel jp = new JPanel();
    this.add( jp );
    int n = 6;
    for ( int i = 0; i < n; i++ )
      gp.add( new JButton( ... ));
  }
```



See ~/cs416/public/demos/swing/flow

Grid Layout

- Grid layout is convenient for creating a 2D array of equal size components

```
public class GridDemo
                           extends JFrame
{
   public GridDemo()
   {
       ....
      JPanel gp = new JPanel(
                  new GridLayout(2,3));
      this.add( gp );
      int n = 6;
      for ( int i = 0; i < n; i++ )
         gp.add( new JButton( ... );
      ....
   }
```

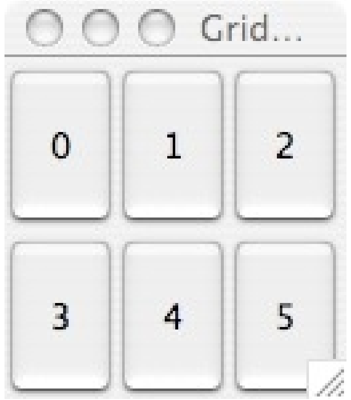
Create panel with grid layout of 2 rows and 3 columns (nominally)

Add 6 components to panel

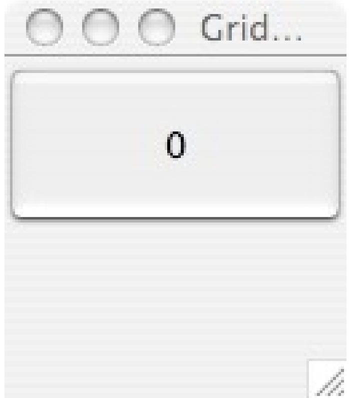
See ~cs416/public/demos/swing/grid

GridLayout Example

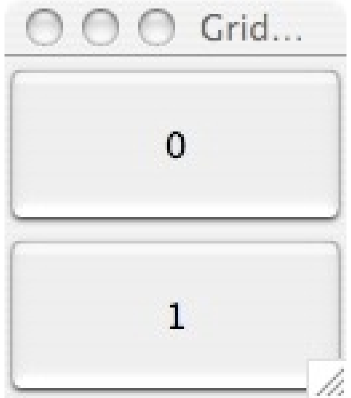
- 2 x 3 grid with 6 components
- What if user adds less than 6? or more?



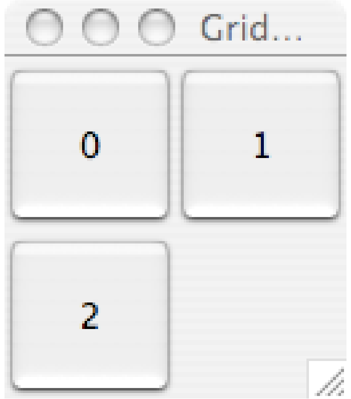
n = 1



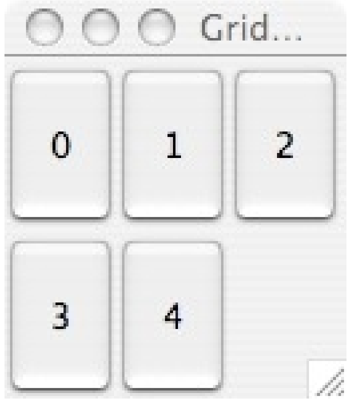
n = 2



n = 3

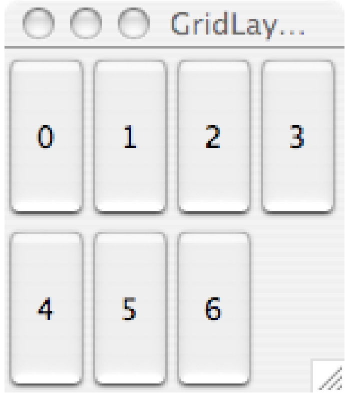


n = 5

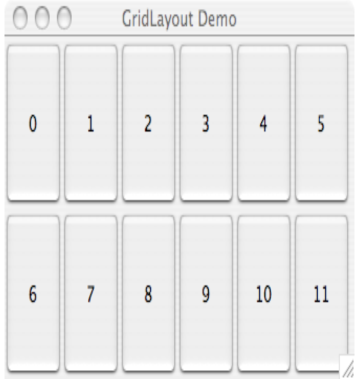


Note that panel size does not automatically change

For *n* > 6, number of rows never exceeds 2! Columns are added as long as needed.



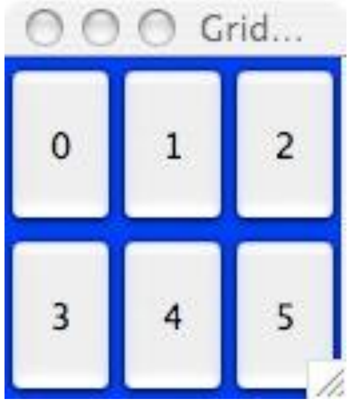
n = 7

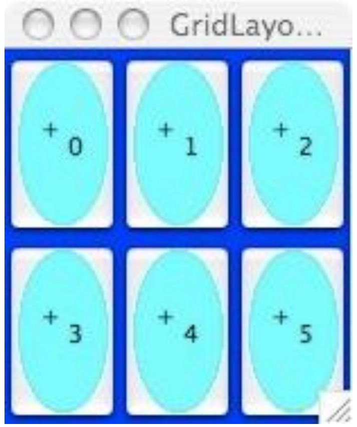


n = 12

The *JButton* Component

- All examples so far have used *JButton* with text labels
- Can change the background color
- Can also put an Icon onto the button
- Implement icon image with *Graphics* drawing methods in *paintIcon* method that is called automatically (like *paintComponent*)
- Can put an *Image* on an *Icon*





Mouse Event Handling

- AWT/mouse handling is defined by the interfaces:
  - *java.awt.event.MouseListener*

```
public void mousePressed( MouseEvent me )
public void mouseClicked( MouseEvent me ) {}
public void mouseEntered( MouseEvent me ) {}
public void mouseExited( MouseEvent me ) {}
public void mouseReleased( MouseEvent me ){}
```

  - *java.awt.event.MouseMotionListener*

```
public void mouseDragged( MouseEvent me )
public void mouseMoved( MouseEvent me ) {}
```

- Mouse listening is started by *Component* methods:

```
public void addMouseListener( MouseListener );
public void addMouseMotionListener( MouseMotionListener );
```

A Mousing SnowMan

```
public class SnowMan extends JComponent
   implements Mover, MouseListener, MouseMotionListener
public SnowMan( Container parent, int x, int y )
{
  ....
  this.addMouseListener( this );
  this.addMouseMotionListener( this );
}
public void mousePressed( MouseEvent me ){ ... }
public void mouseDragged( MouseEvent me ){ ... }
public void mouseClicked( MouseEvent me ){ ... }
public void mouseEntered( MouseEvent me ){ ... }
public void mouseExited( MouseEvent me ){ ... }
public void mouseMoved( MouseEvent me ){ ... }
public void mouseReleased( MouseEvent me ){ ... }
```

Implement 2 mouse listener interfaces

Add *this* object to its own Mouse listeners.

Only include code for the ones you need.

JComponent is a *Component*, so it supports the mouse listener lists

Mouse Dragging Example

```
public void mousePressed( MouseEvent me )
{
  _lastMouse = getParent().getMousePosition();
}
public void mouseDragged( MouseEvent me )
{
  Point curMouse;
  curMouse = getParent().getMousePosition();

  if ( _lastMouse != null && curMouse != null
       && !curMouse.equals( _lastMouse ))
  {
    int dx = curMouse.x - _lastMouse.x;
    int dy = curMouse.y - _lastMouse.y;
    setLocation( getX() + dx, getY() + dy );
    _lastMouse = curMouse;
    getParent().repaint();
  }
}
```

mousePressed is similar to our *wheels* code

except getting the mouse location is different; *Components* "know" where the mouse is **and** we want position in parent

if mouse is not in parent, *null* is returned, so must check for it

schedule parent for updating

Event Handler Design

- Notice that this event handling framework is a bit different from our previous ones.
- Instead of creating an inner class that plays the role of *listener*, the graphical object class does it directly.
    - It works pretty cleanly for our J classes and standard behavior, like dragging.
    - In fact, we can push the implementation of standard code up to *JShape* and share it for all J objects.
    - If you need something non-standard, create your own class that extends *JRectangle*, and/or *JEllipse* and override default behavior

MouseInputAdapter

- Mouse handling via interfaces is a bit tedious
  - *MouseListener* interface requires 5 methods *MouseMotionListener* 2 more
  - We often don’t need all 7 methods
- There is another way!
  - *MouseInputAdapter* is a <u>class</u> with all 7 mouse handling methods implemented, but with no code.
  - When we are creating new *Listener* classes, they can easily inherit from a class, if an appropriate one exist.

Alternative Mouse Handling

```
public class SnowMan extends JComponent
    ~~implements MouseListener, MouseMotionListener~~
    public Shape( Container parent, int x, int y )
    {
        ...
        ~~this.addMouseListener( this );~~
        ~~this.addMouseMotionListener( this );~~
        MyMouser myListener = new MyMouser(...);
this.addMouseListener( myListener );
        this.addMouseMotionListener( myListener );
    }
    ...
    public MyMouser extends MouseInputAdapter
    {
         public void mousePressed( ... ) { ... };
         public void mouseDragged( ... ) { ... };

    }
}
```

Delete the mouse listener interfaces (and 7 methods).

Extend *MouseInputAdapter*.

If you only care about dragging, you don't need to implement empty versions of the other methods -- they are already in the adapter

JRadioButton

- Set of buttons representing mutually exclusive states
- One button is always the currently selected one (except when first create, when no button need be selected)
- Picking a button generates 2 kinds of events
  - *ActionEvent* sent to selected button (just like *JButton*)
  - *ItemEvents* sent to the de-selected and selected buttons
    - Only needed if something must be done when leaving a state and it is not convenient to identify exiting state in the *ActionEvent* code

JRadioButton Example

```
//--- code in main JPanel ---------------
bGroup = new ButtonGroup();

bPanel = new JPanel();
for(int i=0; i<labels.length; i++){
   JRadioButton button = new JRadioButton( labels[i]);
   button.addActionListener(
                  new ButtonListener( i ));
   button.addItemListener(
                  new ButtonItemListener( i ));


   bGroup.add( button );
   bPanel.add( button );
}
this.add( _bPanel, BorderLayout.WEST );
```

ButtonGroup controls the exclusive behavior

This listener is just like that for *JButton*

ItemListener gets called for both selected and de-selected buttons

Add button to group and panel

JRadioButton Action Events

```
public class ButtonListener implements ActionListener
{
  int _buttonId;
  public ButtonListener( int buttonId )
  {
    buttonId = buttonId;
  }
  public void actionPerformed( ActionEvent ev )
  {
    JRadioButton button = new JRadioButton(...)
    button.addItemListener(
                new ButtonItemListener( _numButtons ));
    ....

    bPanel.validate();
  }
}
```

In this example, we add more buttons to the panel and give each an ItemListener

validate updates display when a panel is changed after it has been made visible.

JRadioButton ItemListener

```
public class ButtonItemListener implements ItemListener
{
  int _buttonId;
  public ButtonItemListener( int buttonId )
  {
    _buttonId = buttonId;
  }
  public void itemStateChanged( ItemEvent ev )
  {
    System.out.print( "Button " + _buttonId );
    int state = ev.getStateChange();
    if ( state == ItemEvent.SELECTED )
      System.out.println( " selected." );
    else if ( state == ItemEvent.DESELECTED )
      System.out.println( " deselected." );
  }
}
```

ItemListener interface

itemStateChanged is the key method

getStateChange() tells whether current state is SELECTED or not

JSlider

- Allow users to enter numbers “directly” without having to type text.
- Of course, the numerical resolution is limited to the number of pixels occupied by the slider
- Generate *ChangeEvents*
- Lots of parameters:
  - horizontal or vertical
  - minimum, maximum and initial values
  - automatic labels and tick marks

JSlider Example

```
//--- code in main JPanel ----------------
xSlider = new JSlider();
xSlider.setMinimum( 0 );
xSlider.setMaximum( 400 );
xSlider.setValue( 200 );
addLabels( xSlider );
this.add( xSlider, BorderLayout.SOUTH );

ySlider = new JSlider( JSlider.VERTICAL, 0, 500, 250 );
ySlider.setInverted( true );

this.add( ySlider, BorderLayout.WEST );

xSlider.addChangeListener(
           new SliderListener( _shape, xSlider, "x" ));
ySlider.addChangeListener(
           new SliderListener( _shape, ySlider, "y" ));
```

Lots of user controlled slider parameters including many in the *addLabels* method.

Many can be passed to constructor

Normal vertical sliders have min y value at bottom

add ChangeListeners

JSlider Event Handling

```
public class SliderListener implements ChangeListener
{
  private JShape  target;
  private String  field;
  private JSlider slider;
  public SliderListener( JShape t, JSlider s, String f )
  {
    target = t;
    slider = s;
    field  = f;
  }
  public void stateChanged( ChangeEvent ev )
  {
   if ( field.equals( "x" ))
     target.setLocation( slider.getValue(), target.getY());
   else if ( field.equals( "y" ))
     target.setLocation( target.getX(), slider.getValue());
   else if ( field.equals( "s" ))
     target.setSize( slider.getValue(), _slider.getValue());
   drawPanel.repaint();

}
```

ChangeListener is event handler

It’s a bit hacky to use the string to figure out which slider it is, but it works

We'll provide a *LabeledSlider* class that combines a label and slider.

It may be cleaner and easier to have separate Listener class for each JSlider.

Review

- Principles of user-centered design
- Swing GUI widgets
- Layout Managers
- Input handling; mouse, timer, change events, etc.

Next, in 416

- We’re done with graphics/Swing/awt
  - You’ve got exposed to the basic ideas
  - The rest is just using it and plowing through API and other web resources
- Now we’re ready for *real* "stuff":
  - Recursion and data structures