GUI Programming: Swing and Event Handling

Sara Sprenkle

June 29, 2006

Announcements

- No class next Tuesday
  - My Fourth of July present to you: No quiz!
- Assignment 3 due today
- Review
  - Collections: List, Set, Map
  - Inner classes
  - AWT, Swing programming

Jar Files

- Archives of Java files
  - Essentially a Zip of the Java class files
  - Includes a special metadata file with the path META-INF/MANIFEST.MF
    - `jar` creates a default metadata file, if not specified
- Why jars?
  - Package code into a neat bundle
    - Easier, faster to download
    - Easier to use
- Works similarly to tar
  - `jar cf myapplication.jar *.class`

Jar file: Metadata

- Example metadata file that will allow you to execute the JAR with *java*

```
Manifest-Version: 1.0
Main-Class: MyApplication
```

  - Note the newline at the end

Specifying the metadatafile

- To create the jar file:

```
jar cmf myManifest myapplication.jar *.class
```

- Run it, using java
  - `java -jar myapplication.jar`

Useful later when we create applets

Using jars

- Add jar files to CLASSPATH to use classes in jar file
- Example: adding a jar file to the current classpath variable ($CLASSPATH)

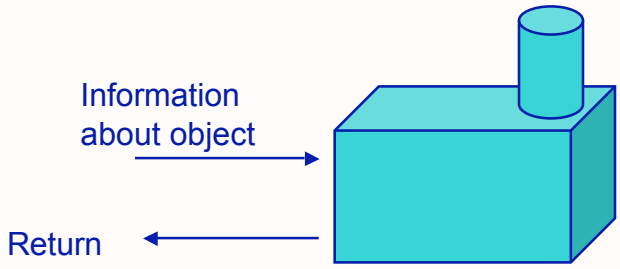
```
setenv CLASSPATH $CLASSPATH:myapplication.jar
```

In Eclipse, you need to "Configure Build Path"

Factories

- A Factory class exists just to create instances of other classes
  - Often produces compatible instances of several related classes
- Given information about an object, will create the object
- Allows something else to determine the appropriate customization of object

Factories



Factory
May have several different implementations of object to choose from; makes selection

Factories

- Tend to have private constructors
  - Cannot make an object using "new"
- Use a static method to get the factory
  - Ensures that there is only one factory for all executing classes
  - Factory factory = Factory.getFactory();

Reviewing Swing

- Frame/JFrame
  - Swing Implementation: JFrame
    - Swing naming convention: begin with J
  - Basic window with title
  - Positioning, sizing
  - Set image if iconified
  - How handles closing (setDefaultCloseOperation)
- Toolkit
  - Get information about screen size

Reviewing Swing

- Components
  - Everything you *see*
  - JFrame is a component
  - Use setVisible to display the frame
- Windows
  - activation
- Panels
  - Contain content; useful for layout (more today)
- Drawing on panels
  - Using different fonts

More GUI components

- Label
  - Basically, just a string
- Buttons
  - Like a label but generates events
- Checkbox
  - Buttons with state about if checked
- CheckboxGroup
  - Radio buttons - only one can be selected at a time

More GUI Components

- Choice
  - Drop-down list
- FileDialog
  - Opening and saving files
- List
  - Scrollable
  - Allows multiple selections
- ScrollPane
  - scrollbars

More GUI Components

- TextField
  - Single line of text
- TextArea
  - Multiple lines of text

Menus

- MenuBar
  - Thing across top of frame
  - Frame.setMenuBar(MenuBar mb);
- Menu
  - The dropdown part
  - A sequence of MenuItems
  - Menu is a subclass of MenuItems, so can have submenus

Combining Components

- Create a panel with three buttons on it
  - Use panel's add method

Placement of Components

- How does the panel know where to place a button?
- How does the panel know where to place the next button?
- How does the panel know where to place *any* component that is added to it?

Layout Managers

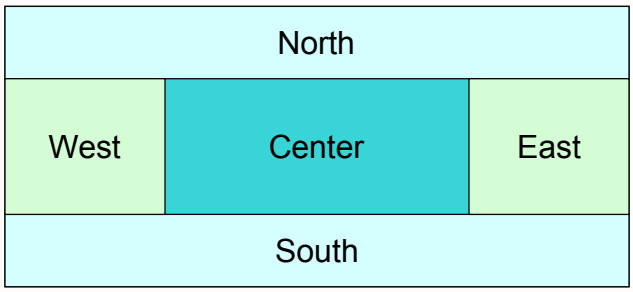
- Java uses a concept of layout managers to place components inside a container
- The LayoutManager is an object that handles the placement of components
  - When a component is added to a container, through add(), the layout manager decides where to place the component

The Border Layout Manager

- Default layout manager of the content pane for JFrame
- Lets you choose where you want to place each component
  - Center
  - North
  - South
  - East
  - West

with respect to the container

Border Layout Regions



- The edge components are laid out first
- Center occupies remaining space

Border Layout Rules

- The border layout grows all components to fill the available space
  - The flow layout gives each component its *preferred size*
- If the container is resized, the edge components are redrawn and the center region size recomputed.
- To add a component to a container using a border layout, say the content pane of a JFrame:

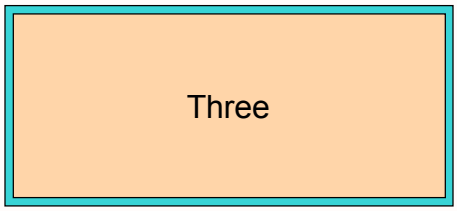
```
Container contentPane = getContentPane();
contentPane.add(yellowButton, BorderLayout.SOUTH);
```

Adding Components Using a Border Layout

```
Container contentPane = getContentPane();
contentPane.add(yellowButton, BorderLayout.SOUTH);
```

- If no region of the layout is specified
  - Assumes center region
- Since border layout grows the component to fit the specified region
  - What happens if we add multiple components, e.g., three buttons, without specifying a region?

A Border Layout Limitation



- The three button grows to completely fill the center region
- The one and two buttons were discarded/overwritten by each subsequently added component

The Flow Layout Manager

- Default layout manager for a panel (not JFrame)
  - What I changed our JFrame to use
- Lines the components up horizontally until there is no more room in the container
  - Then starts a new row of components
- If the user resizes the component, the layout manager automatically reflows the components

The Flow Layout Manager

- You can choose how to arrange the components in each row
  - Default: center each row
  - Other options: left or right align
- Change alignment using the setLayout() method
  - causes the panel to use a flow layout manager, with the row components aligned to the left

```
setLayout(new FlowLayout(FlowLayout.LEFT));
```

- Also has `hgap` and `vgap` for gaps to put around components

Using Panels w/ Border Layout

- Panels act as (smaller) containers for UI elements
- Can be arranged inside a larger panel by a layout manager
- Use additional panels to address Border Layout problem
  - create a panel
  - add some buttons to it
  - add that panel to the south region of content pane

Using Additional Panels

- Get fairly accurate and precise placement of components
- Use Nested panels with
  - Border layouts - for content panes and enclosing panels
  - Flow layouts - for panels containing the buttons and other UI components

FlexibleLayout.java

Changing Layout Managers

- Any container can use any layout manager
- Use setLayout() to change layout manager before you add components

```
// sets the layout to a new flow layout manager which
// aligns row components to the left and uses a 20 pixel
// horizontal separation and 20 pixel vertical separation
setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));

// sets the layout to a new border layout manager which
// uses a 45 pixel horizontal separation between components
// (regions) and a 20 pixel vertical separation
setLayout(new BorderLayout(45, 20));
```


The Grid Layout Manager

- Divides the container into columns and rows of equal size, which collectively occupy the entire container region
- Rows and columns are aligned like a spreadsheet
  - When the container is resized, the “cells” grow and/or shrink
  - Cells always maintain identical sizes

Grid Layout Manager Construction

- The number and rows to be used in the layout is specified in the construction of the grid layout manager:

```
panel.setLayout(new GridLayout(5, 4)); // 5 rows, 4 cols
```

- As with the border and flow layout managers, you can specify a horizontal and vertical separation (between rows and columns):

```
panel.setLayout(new GridLayout(5, 4, 20, 20));
// 5 rows, 4 cols, 20 pixels between rows & between cols
```


Adding Components to a Grid Layout

- Components are added to a grid layout sequentially
- The first add() adds the component to the 1st row and 1st column
- The second add() adds the component to the 1st row, 2nd column.
- And so forth until the 1st row is filled in
- Then the 2nd row begins with the 1st column
- Continues until the entire container is filled

Grid Layout Rules

- When a component is added to a cell, it is resized to take up the entire cell.
- This layout manager is quite restrictive but can be useful for some applications
- Example:
  - to create a row of buttons with identical size
  - make a panel that has a grid layout with one row
  - add a button to each cell
    - set some horiz/vert separation, so the buttons are not touching

Handling User Interactions

Event Basics

- An event is generated from an *event source* and is transmitted to an *event listener*
- Event sources allow event listeners to *register* with them
  - Registered listener is requesting that the event source send its event to the listener when it occurs
    - Think of as mailing list for favorite band but notifies of *past* releases!
- All events are objects of *event classes*
- All event classes derive from **EventObject**

Java / AWT Event Handling

- Listener object: implements a listener interface
- Event source: can register listener objects and send them event objects
  - Event source sends out event objects to all registered listeners when that event occurs.
- Listener objects use the event object to determine their reaction to the event

Java / AWT Event Handling

- A listener is registered with an event source by using a line of code similar to…

```
eventSourceObject.addEventListener(
                  eventListenerObject);
```

- For example…

```
ActionListener listener1 = . . .;
JButton button1 = new JButton("Click Me!");
button1.addActionListener(listener1);
```

- listener1 is notified whenever an "action event" occurs in the button
  - For buttons, an action event is a button click

Listener Objects

- A listener object must belong to a class that implements the appropriate interface
  - For buttons, that's **ActionListener**
- The listener class must implement the method actionPerformed(ActionEvent event)

Listener Objects and Event Handling

- When the user clicks the button (button1 in our example), the JButton object generates an ActionEvent object
  - Which makes JButton a what?
- JButton calls the listener object's actionPerformed() method and passes that method the generated event object
- A single event source can have multiple listeners listening for its events
  - The source calls actionPerformed() on each of its listeners

An Example of Event Handling

- Suppose we want to make a panel that has three buttons on it
  - Each button has a color associated with it
  - When the user clicks a button, the background color of the panel changes to the corresponding color
- We need two things:
  - A panel with three buttons on it
  - Three listener objects, each registered to listen for events on one of the buttons

Event Handling Example

- Make some buttons and add them to panel

```
public class ColoredBackground extends JFrame {
   public ColoredBackground() {
      …
      JButton red = new JButton("Red");
      red.setForeground(Color.red);
      JButton yellow = new JButton("Yellow");
      yellow.setBackground(Color.yellow);
      JButton blue = new JButton("Blue");
      blue.setForeground(Color.blue);
      cp.add(red);
      cp.add(yellow);
      cp.add(blue);
…
```

JButton constructor takes a String (the button's label)

Listener Objects

- Now that we have buttons (event sources), we need listeners
  - An action listener can be any class, as long as it implements the ActionListener interface.
- Make a new class that implements the interface
  - actionPerformed method should set the background color of the panel

Our Listener Class

```
class ColorAction implements ActionListener
{
  public ColorAction(Color c)
  { backgroundColor = c; }

  public void actionPerformed(ActionEvent evt1)
  {
      // set panel background color here
      . . .
  }

  private Color backgroundColor;
}
```

How can we do this?

Registering Our Listener Class

- Now that we have a class that listens for the buttons' ActionEvents, we need to create objects and register them with the buttons…

```
ColorAction yellowAction = new ColorAction(Color.yellow);
ColorAction blueAction   = new ColorAction(Color.blue);
ColorAction redAction    = new ColorAction(Color.red);

yellow.addActionListener(yellowAction);
blue.addActionListener(blueAction);
red.addActionListener(redAction);
```

These are JButtons

Registering Our Listener Class

- When a user clicks the button with the label "Yellow", the yellow JButton object generates an ActionEvent
  - passes this event object to the yellowAction's actionPerformed() method
  - method can then set the background color of the frame

Any problems?

The Listener Class & the Frame

- The ColorAction objects have no access to the frame
  - How can they change the background color?
- There are two possible solutions:
  - Add a frame instance field to the ColorAction class and set it in the constructor
    - ColorAction object knows which frame it is associated with and can call the appropriate method of that frame to change its background color
  - Make ColorAction an inner class of ButtonPanel1

Listener as an Inner Class

- Let's make it an inner class…

```
class ColoredBackground extends JFrame
{
   . . .
   private class ColorAction implements ActionListener
   {
      . . .
     public void actionPerformed(ActionEvent evt)
     {
      setBackground(backgroundColor);
      repaint();
     }
     private Color backgroundColor;
   }
}
```


The actionPerformed() Method

```
public void actionPerformed(ActionEvent evt) {
      setBackground(backgroundColor);
      repaint();
}
```

- ColorAction class does not have setBackground() or repaint() methods
- Since it is an inner class of ColoredBackground, it can directly access that class' instance fields and methods
- Inner class calls the outer class's method using its own private inner data (backgroundColor)
- Then it calls the outer class's repaint() method
  - Redraw the frame

Event Listeners as Inner Classes

- A common and beneficial practice
- Event listener objects typically need to do something to other objects when their corresponding event occurs
- It is often possible to place the listener class inside the class whose state the listener should modify
- It makes good sense from an OOP design standpoint
  - Not violating encapsulation rules …
  - also makes things easier

A Different Listener Approach

- Any object of a class that implements ActionListener can listen for action events from a source
  - we could make our ColoredBackground class listen for its own buttons' events
  - implement the interface and do the correct registering with the buttons

A Different Listener Approach

```
class ColoredBackground2 extends JFrame
      implements ActionListener
{
   public ColoredBackground2()
   {
      . . .
      yellow.addActionListener(this);
      blue.addActionListener(this);
      red.addActionListener(this);
   }
   . . .
   public void actionPerformed(ActionEvent evt)
   {
      // set background color
      . . .
   }
}
```


A Different Listener Approach

- The ColoredBackground method actionPerformed() will run whenever any of the buttons is clicked
  - How do we find out which button was pressed?

```
public void actionPerformed(ActionEvent evt)
{
      Object source = evt.getSource();
      // gets the source that generates this event

      if (source == yellow) . . .
      else if (source == blue) . . .
      else if (source == red) . . .
}
```

Why ==, not equals()?

Which way is better?

- The inner class approach makes sense from an OOP design point
  - Each event source gets its own listener, which can directly modify the panel as it needs to do
- Having the panel itself listen is much more straightforward
  - Since the panel needs to change, have it listen!
  - But, the handling method must determine the source of the event and switch its behavior
- Consider: How easy to add additional event sources for each case?

Which Way is Better?

- Neither way is “better.”
- If a container has multiple UI components that generate events, the container listening for and handling them all gets really confusing and challenging
- Inner classes make sense
  - Somewhat confusing at first
  - Great benefits
  - We will tend to use the inner class listeners

Simplification of our Event Handlers

- For each button, we do four things:
  - Construct the button with a label string
  - Add the button to the panel
  - Construct an action listener with the appropriate color
  - Register that listener with the button
- Make a method that does this for us to simplify the code

Simplification of our Event Handlers

```
void makeButton(String label, Color backgroundColor)
{
  JButton button = new JButton(label);
  add(button);
  ColorAction action = new ColorAction(backgroundColor);
  button.addActionListener(action);
}
```

- Makes the ColoredBackground constructor much simpler…

```
public ColoredBackground()
{
      makeButton("Yellow",Color.yellow);
      makeButton("Blue",Color.blue);
      makeButton("Red",Color.red);
}
```

One More Step

- We only use the ColorAction class in the makeButton() method
- How can we further simplify the code?

One More Step

- Make the ColorAction class an anonymous inner class
  - only use this class at one point
  - define it on the fly

An Anonymous Class Listener

```
void makeButton(String label, final Color bgColor)
{
  JButton button = new JButton(label);
  add(button);
  button.addActionListener( new ActionListener()
      {
         public void actionPerformed(ActionEvent evt)
         {
            setBackground(bgColor);
            repaint();
         }
      } );
}
```


Window Events

- Not every event is as simple to handle as a button click
- When a user closes a window, the window simply stops being displayed
  - Program will not end
- Suppose we want our program to end when a certain frame is closed
- Closing a frame is a window event
  - in contrast to an action event

Window Events

- To catch window events, you need to create an object of a class that implements the **WindowListener** interface
  - WindowListener object is registered with the frame using the frame's addWindowListener() method
- Note the parallels with action events
  - change the type of listener and register it using a different (but similar) method call

The WindowListener Interface

- The WindowListener interface contains seven methods
  - One for each type of window event
  - A class that implements WindowListener must have all seven methods

The WindowListener Interface

```
public interface WindowListener
{
      void windowOpened(WindowEvent e);
      void windowClosing(WindowEvent e);
      void windowClosed(WindowEvent e);
      void windowIconified(WindowEvent e);
      void windowDeiconified(WindowEvent e);
      void windowActivated(WindowEvent e);
      void windowDeactivated(WindowEvent e);
}
```


Implementing a WindowListener

- To create an object that can listen for window events on a frame and end the program when the frame is closed…

```
class Terminator implements WindowListener
{
      public void windowClosing(WindowEvent evt)
      {
            System.exit(0);
      }
      public void windowOpened(WindowEvent e) {}
      public void windowClosed(WindowEvent e) {}
      public void windowIconified(WindowEvent e) {}
      public void windowDeiconified(WindowEvent e) {}
      public void windowActivated(WindowEvent e) {}
      public void windowDeactivated(WindowEvent e) {}
}
```

For JFrame's use setDefaultClosedOperation

Adapter Classes

- Typing the code for the six methods that don't do anything is somewhat tedious
- Most AWT listener interfaces have a corresponding adapter class
  - Class that implements all of the methods in the interface but does nothing inside all of them.
  - There are no adapter classes for AWT interfaces with only one method (such as ActionListener)

Adapter Classes

- If you want an WindowListener class that does nothing with six of the seven window events, but ends the program when the window is closed
  - create a new class that extends the **WindowAdapter** class and override the method(s) you are interested in
  - When could extending a class be a problem?
  - How big of a concern is that for this specific case/type of class?

Extending an Adapter Class

- We can now redefine our Terminator class in much less code…

```
class Terminator extends WindowAdapter
{
      public void windowClosing(WindowEvent evt)
      {
            System.exit(0);
      }
      // all other methods are the same as in
      // WindowAdapter, all do nothing.
}
```


Registering a WindowListener

- Now, we can register our Terminator class to listen for window events.
- Assuming that our "main" window frame is named frame1 (e.g., if frame1 is closed the program should exit)…

```
WindowListener listener1 = new Terminator();
frame1.addWindowListener(listener1);
```

One Step Farther…

- Using the WindowAdapter class instead of writing out all the methods in the WindowListener interface is nice, but we can go one step farther…

```
frame1.addWindowListener( new
      WindowAdapter()
      {
            public void windowClosing(WindowEvent evt)
            {
                  System.exit(0);
            }
      } );
```


One Step Farther…

- This code does five things:
  - Defines a new class without a name that extends the WindowAdapter class
  - Adds a windowClosing() method to that anonymous class (the exits the program)
  - Inherits the remaining six empty methods from the WindowAdapter class
  - Creates an object of this new class
    - This object also does not have a name
  - Passes this new no-name object to the addWindowListener method of frame1

The AWT Event Hierarchy

- There are 10 different types of events in the AWT…

AWT Event Types – Semantic Events

- Four types of semantic events
  - an event that expresses what a user did, such as clicking a button.
- **ActionEvent** – button click, menu selection, selecting a list item, pressing ENTER in a text field
- **AdjustmentEvent** – user adjusted a scroll bar
- **ItemEvent** – user made a selection from a set of checkboxes or list items
- **TextEvent** – the contents of a text field or text area were changed

AWT Event Types – Low-Level Events

- 6 types low-level *events*
  - An event that makes a semantic event possible
- **ComponentEvent** – component was changed (resized, moved, shown, etc…)
- **KeyEvent** – a key was pressed or released
- **MouseEvent** – the mouse was moved, dragged, or a button was pressed
- **FocusEvent** – component got or lost focus
- **WindowEvent** – window was activated, closed, etc.
- **ContainerEvent** – component was added or deleted.

AWT Event Types

- Example: adjusting a scrollbar is a semantic event
  - made possible by some low-level events such as dragging the mouse
- As a general rule, low-level events cause semantic events to happen

Event Listeners

- 11 interfaces that correspond to AWT event listeners:
  - **ActionListener, AdjustmentListener, ItemListener, TextListener, ComponentListener, ContainerListener, FocusListener, KeyListener, MouseListener, MouseMotionListener,** and **WindowListener**.

Event Listeners

- See Javadocs for interfaces and all the methods that are in them
- Each listener interface with more than one method has a corresponding adapter class
  - implements the interface with all empty methods

Components and ComponentEvents

- A component is a user interface element
  - such as a button, textfield, or scrollbar
- All low-level events inherit from **ComponentEvent**
  - getComponent() returns the component that originated the event
    - same thing as getSource(), but it returns the object as a Component and not an Object
- For example, if a key event was generated because of an input into a text field, then getComponent returns a reference to that text field

Containers and ContainerEvents

- A container is a screen area or component
  - can contain components, such as a window or a panel
- A **ContainerEvent** is generated whenever a component is added or removed from the container
  - supports the programming of dynamically-changing user interfaces

FocusEvents

- A **FocusEvent** is generated when a component gains or loses focus
- **FocusListener** must implement two methods:
  - focusGained(): called whenever the event source the listener is registered with gains the focus
  - focusLost(): called whenever the event source the listener is registered with loses the focus

KeyEvents

- A **KeyEvent** is generated when a key is pressed or released.
- A **KeyListener** must implement three methods:
  - keyPressed() will run whenever a key is pressed
  - keyReleased() will run whenever that key is released
  - keyTyped() combines the two – it runs when the key is pressed and then released, and signifies a keystroke

KeyEvents

- With what type of object does a KeyListener register with?
- What is an event source for a KeyEvent?
- Any Component can be an event source for a KeyEvent
  - A component generates a KeyEvent whenever a key is typed in that component
- For example, if the user types into a textfield, that textfield will generate the appropriate KeyEvents

MouseEvents

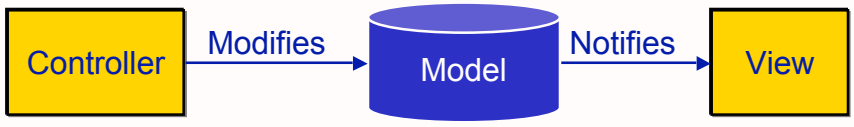
- MouseEvents are generated like KeyEvents
  - mousePressed()
  - mouseReleased()
  - mouseClicked()
  - You can ignore the first two if you only care about clicking
- Call getClickCount() on a MouseEvent object to distinguish between a single and a double click
- You can distinguish between the various mouse buttons by calling getModifiers() on a MouseEvent object
  - E.g., middle button

MouseEvents

- MouseEvents are also generated when the mouse pointer enters and leaves components (mouseEntered() and mouseExited()).
- All of those methods are part of the **MouseListener** interface
- The actual movement of the mouse is handled with the **MouseMotionListener** interface.
  - most applications only care about where you click and **not** how and where you move the mouse pointer around

Model - Viewer - Controller (MVC)

- A common design pattern for designing GUIs
  - Design Pattern: proven way to design something
- Separate
  - Model: application data
  - View: graphical representation
  - Controller: input processing



Model-Viewer-Controller

- Can have multiple viewers and controllers
- Can modify one component without affecting others

Model

- Code that carries out some task
- Nothing about how view presented to user
- Purely functional
- Must be able to register views and notify views of changes

Multiple Views

- Provides GUI interface components for model
- User manipulates view
  - Informs controller of change
- Example of multiple views: spreadsheet data
  - Rows/columns in spreadsheet
  - Pie chart

Multiple Controllers

- Update model as user interacts with view
  - Calls model's mutator methods
- Views are associated with controllers

Designing a Card Game

- What is the
  - Model
  - View(s)
    - Solitaire?
    - Poker?
  - Controller(s)

Project 1: Freecell

- Part 1: Understand code base
  - Given a Solitaire code base, Javadocs (API), example code that uses code base
  - Compile, use, and understand
- Part 2: Use code base to implement Freecell
  - GUI, Event handling
  - Implementing/Enforcing the Freecell rules

Project 1: Freecell

- Submission: printed, electronic, and demo
  - Submission due Thursday, July 13
  - New: Test Plan
    - Individual classes, whole application
  - Documentation: describe design decisions
  - Demo with Ke
    - Schedule demos Monday-Wednesday of following week
    - (Project 2 demos with Sara)
  - Script file doesn't make as much sense