Objectives

- Event handling
- Design Patterns
  - Composition
  - Strategy

Other types of events

EVENT HANDLING

Window Events

- When a user closes a window, the window simply stops being displayed
  - Program does 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*

Catching Window Events

- To catch window events, create an object of a class that implements `WindowListener` interface
  - `WindowListener` is registered with frame using its `addWindowListener` method
- Note the parallels with action events
  - Different listener type and register it using a different (but similar) method call

The `WindowListener` Interface

- Contains *7* methods
  - One for each type of window event
  - A class that implements `WindowListener` must implement **all 7** methods

```
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);
}
```


Example: Implementing a `WindowListener`

What does this class do?

```
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) {}
}
```

Adapter Classes

- Writing code for 6 methods that don't do anything is somewhat tedious
  - Eclipse helps
- Most AWT listener interfaces have a corresponding ***adapter class***
  - Implements each of interface's methods but does nothing inside each
  - No adapter classes for AWT interfaces with only one method (such as `ActionListener`)

Adapter Classes

- If you want a `WindowListener` class that does nothing with most window events
  - Create a new class that **`extends`** `WindowAdapter` and override relevant method(s)
- When is extending a class a problem?
  - How big of a concern is that for this specific case/ type of class?

Extending an Adapter Class

- Redefine `Terminator` 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`

- Register `Terminator` to listen for window events
  - Assuming that our "main" window frame is named `frame`
- Result if `frame` is closed, the program should exit

```
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

Alternative: Registering a WindowListener

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

What is going on in this code?

TYPES OF EVENTS

AWT Event Hierarchy

- 10 different types of events in AWT
  - Semantic events
  - Low-level events

> Rule of thumb: low-level events cause semantic events to happen

- Example:
  - Adjusting a scrollbar is a *semantic* event
  - Made possible by low-level events, such as dragging the mouse

AWT Event Types: Semantic Events

- **Semantic event**: event that expresses what a user did

Type	Cause
`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

- **Low-level event**: makes a semantic event possible

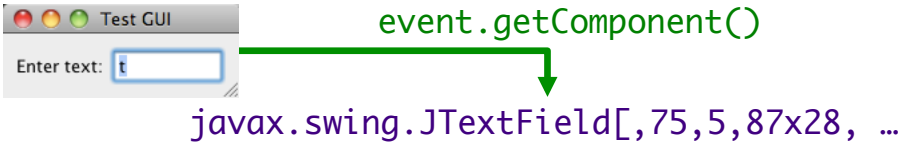
Type	Cause
`ComponentEvent`	component changed (resized, moved, shown, etc…)
`KeyEvent`	a key pressed or released
`MouseEvent`	mouse moved or dragged, or mouse button pressed
`FocusEvent`	component got or lost focus
`WindowEvent`	window activated, closed, etc.
`ContainerEvent`	component added or deleted

AWT Event Listeners

- 11 Event Listener Interfaces
  - `ActionListener, AdjustmentListener, ItemListener, TextListener, ComponentListener, ContainerListener, FocusListener, KeyListener, MouseListener, MouseMotionListener,` and `WindowListener`
- See API for interfaces and their methods
- Each listener interface with > 1 method has a corresponding ***adapter class***
  - Implements interface with all empty methods

Components and ComponentEvents

- A ***component*** is a user interface element
  - Examples: button, textfield, scrollbar
- All low-level events inherit from `ComponentEvent`
  - `getComponent()` returns component that originated event
    - Similar to `getSource()` but returns object as a `Component` and not an `Object`
- Ex: A user inputs text into a text field, generating a key event. Calling `getComponent()` on the event returns a reference to that text field



Containers and ContainerEvents

- A ***container*** is a screen area or component
  - Can contain components, such as a panel
- A `ContainerEvent` is generated whenever a component is added or removed from the container
  - Supports dynamically-changing user interfaces

FocusEvents

- A `FocusEvent` is generated when a component gains or loses focus
- `FocusListener` must implement two methods:
  - `focusGained()`: called whenever listener's event source gains focus
  - `focusLost()`: called whenever listener's event source loses focus

KeyEvents

- A `KeyEvent` is generated when a key is pressed or released
- A `KeyListener` must implement 3 methods:
  - `keyPressed()` will run whenever a key is pressed
  - `keyReleased()` will run whenever that key is released
  - `keyTyped()` combines the two above
    - Runs when key is pressed and then released
    - Signifies a keystroke

KeyEvents

- Any `Component` can be an *event source* for a `KeyEvent`
  - A component generates a `KeyEvent` whenever a key is typed in that component
- Example:
  1. User types into a text field
  2. That text field generates appropriate `KeyEvent`s

MouseEvents

- `MouseEvents` are generated like KeyEvents
  - `mousePressed()`
  - `mouseReleased()`
  - `mouseClicked()`
  - You can ignore first 2 if you only care about clicking
- Call `getClickCount()` on a `MouseEvent` object to distinguish between a single and a double click
- Distinguish between mouse buttons by calling `getModifiers()` on a `MouseEvent` object
  - E.g., middle button

MouseEvents

- MouseEvents are also generated when mouse pointer enters and leaves components (mouseEntered() and mouseExited())
  - Part of MouseListener interface
- Actual movement of mouse is handled with MouseMotionListener interface
  - Most applications only care about where you click and not how and where you move mouse pointer around

DESIGN PATTERNS

Design Pattern

General reusable solution to a commonly occurring problem in software design

- Not a finished design that can be transformed directly into code
- Description or *template* for how to solve a problem that can be used in many different situations
  - "Experience reuse" rather than code reuse

Defined Design Patterns

- Software best practices
- Catalogued and discussed in *Design Patterns: Elements of Reusable Object-Oriented Software*
  - Written by the "**Gang of Four**": Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
    - Erich Gamma also co-wrote JUnit framework
  - Didn't design the patterns; identified them

Understanding Code

1. Recognize design pattern in code base you're using
2. Understand code design better

Applying Design Patterns

1. Recognize problem as one that can be solved by a design pattern
2. Apply pattern to your problem

Danger: over-applying design patterns
- Fall back: Identify and resolve code smells