# Event-driven Programming

Reacting to the user

# Outline

- Sequential programming
- GUI program organization
- Event-driven programming
- Modes

# Sequential Programming

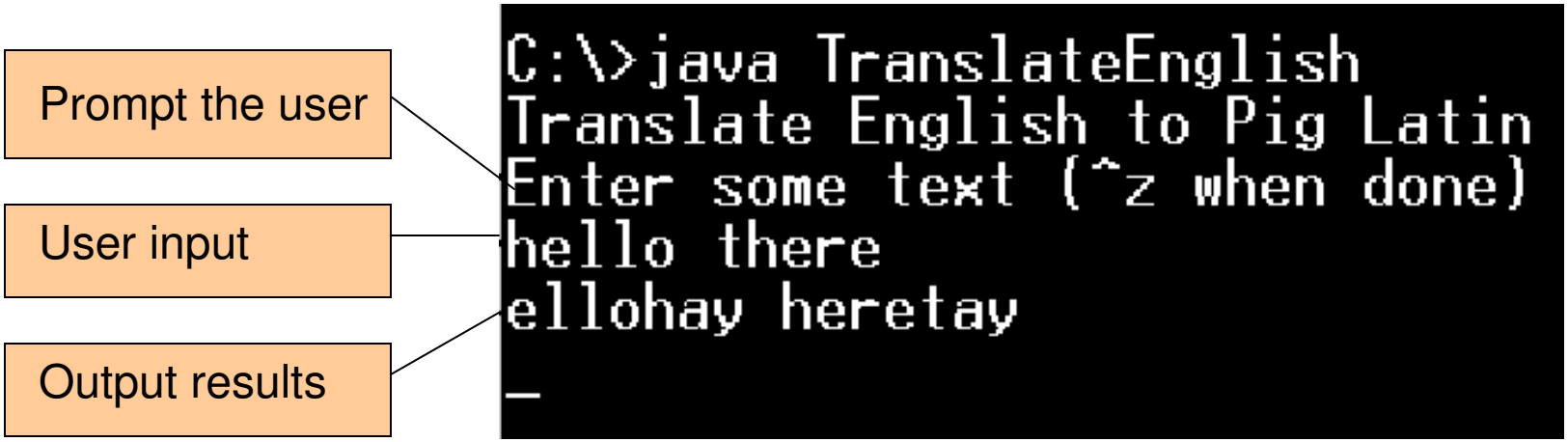
- In sequential programs, the program is under control
- The user must synchronize with the program:
  - Program tells user it is ready for input
  - User enters input and it is processed
- Examples:
  - Command-line prompts (DOS, UNIX)
  - LISP interpreters
- *Shouldn't the program be required to synchronize with the user?*

# Sequential Programming (2)

- Flow of a typical sequential program
  - Prompt the user
  - Read input from the keyboard
  - Parse the input (determine user action)
  - Evaluate the result
  - Generate output
  - Repeat

# Example

**DemoTranslateEnglishConsole.java**



# Sequential Programming (3)

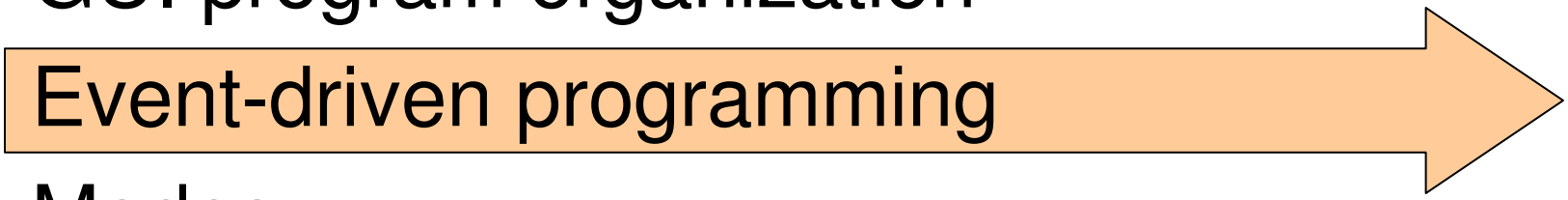
- Advantages
  - Architecture is iterative (one step at a time)
  - Easy to model (flowcharts, state machines)
  - Easy to build
- Limitations
  - Can’t implement complex interactions
  - Only a small number of features possible
  - Interaction must proceed according to a pre-defined sequence
- To the rescue… Event-driven programming
- But first…

# Outline

- Sequential programming
- GUI program organization
- Event-driven programming
- Modes

# Outline

- Sequential programming
- GUI program organization
- Event-driven programming
- Modes



# Event-driven Programming

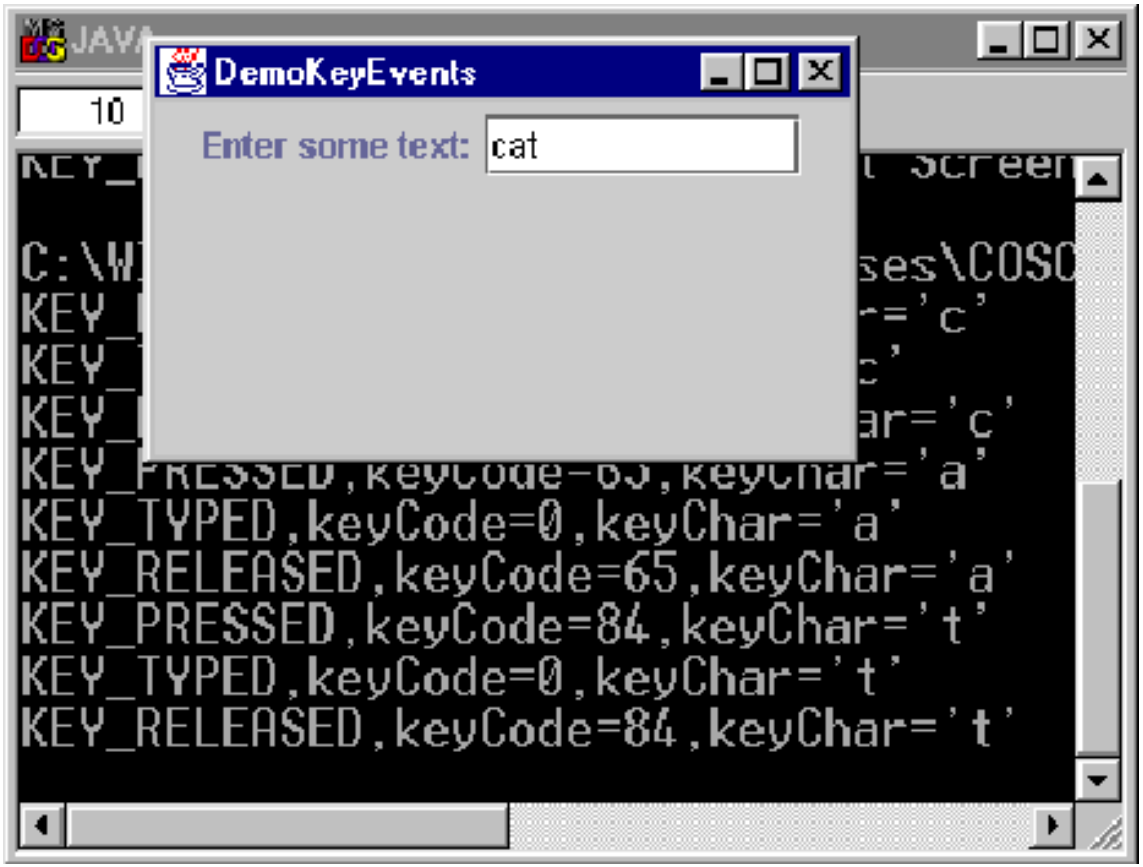
- Instead of the user synchronizing with the program, the program synchronizes with, or reacts to, the user
- Communication from user to computer occurs via events and the code that handles the events
- An event is an action that happens in the system; e.g.,
  - A mouse button pressed or released
  - A keyboard key is hit
  - A window is moved, resized, closed, etc.

# Classes of Events

- Typically, two classes of events:
  - User-initiated events
    - Events that result directly from a user action
    - E.g., mouse click, move mouse, key press
  - System-initiated events
    - Events created by the system, as it responds to a user action
    - E.g., scrolling text, re-drawing a window
- Both classes need to be processed in a UI
- User-initiated events may generate system-generated events

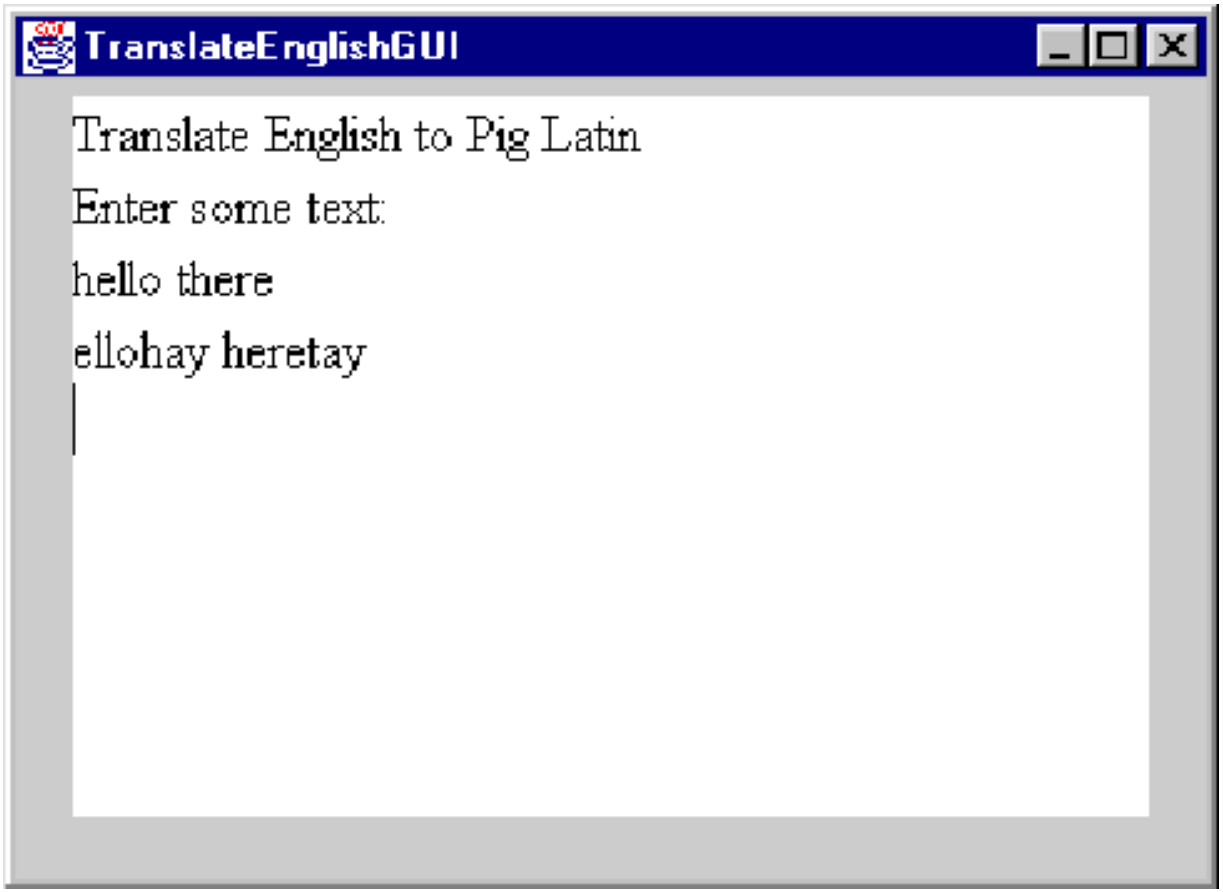
# Example Program

## DemoKeyEvents.java



# Example Program

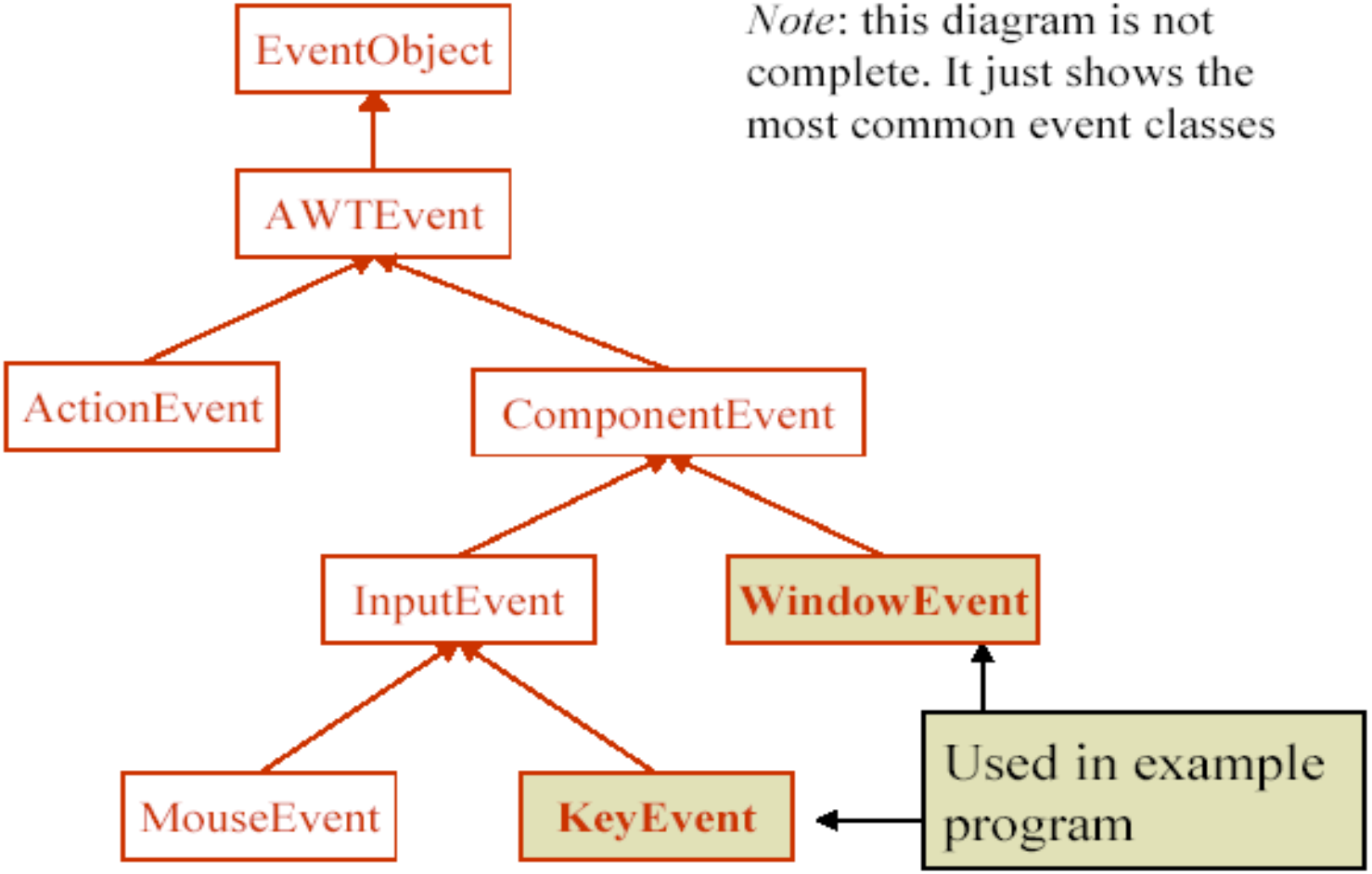
**DemoTranslateEnglishGUI.java**



# What just Happened?

- Event-driven programming takes us far beyond sequential programming
- Let's examine the example program
- First, a few words on Java events

# Java's Event Class Hierarchy




*Note*: this diagram is not complete. It just shows the most common event classes

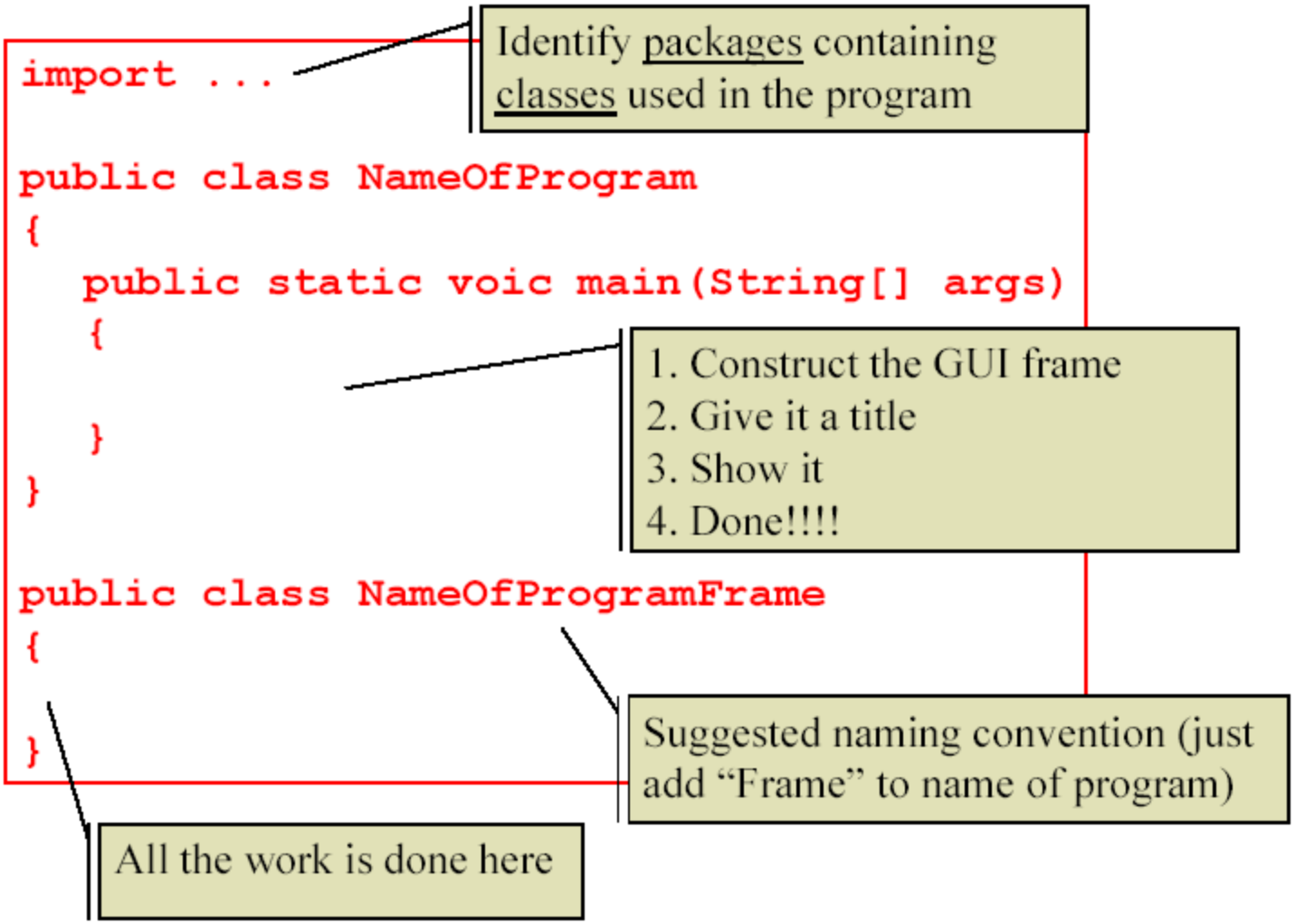
# Java Events

- When a user types characters or uses the mouse, Java's window manager sends a notification to the program that an event has occurred
- E.g., when the user presses a key on the keyboard, a key pressed event occurs
- There are many, many kinds of events (e.g., key pressed, key released, key typed)
- Many are of no interest
- Some are of great interest

# Java Events (2)

- To receive notification of events of interest, a program must install event listener objects
- If is not enough to know that an event has occurred; we also need to know the event source
- E.g., a key was pressed, but in which of several text fields in the GUI was the key pressed?
- So, an event listener must be installed for particular components that wish to listen for the event
- Let's look at the code. First, the big picture…

# Code Organization (review)



```
import ...

public class NameOfProgram
{
   public static voic main(String[] args)
   {

   }
}

public class NameOfProgramFrame extends JFrame
{

}
```

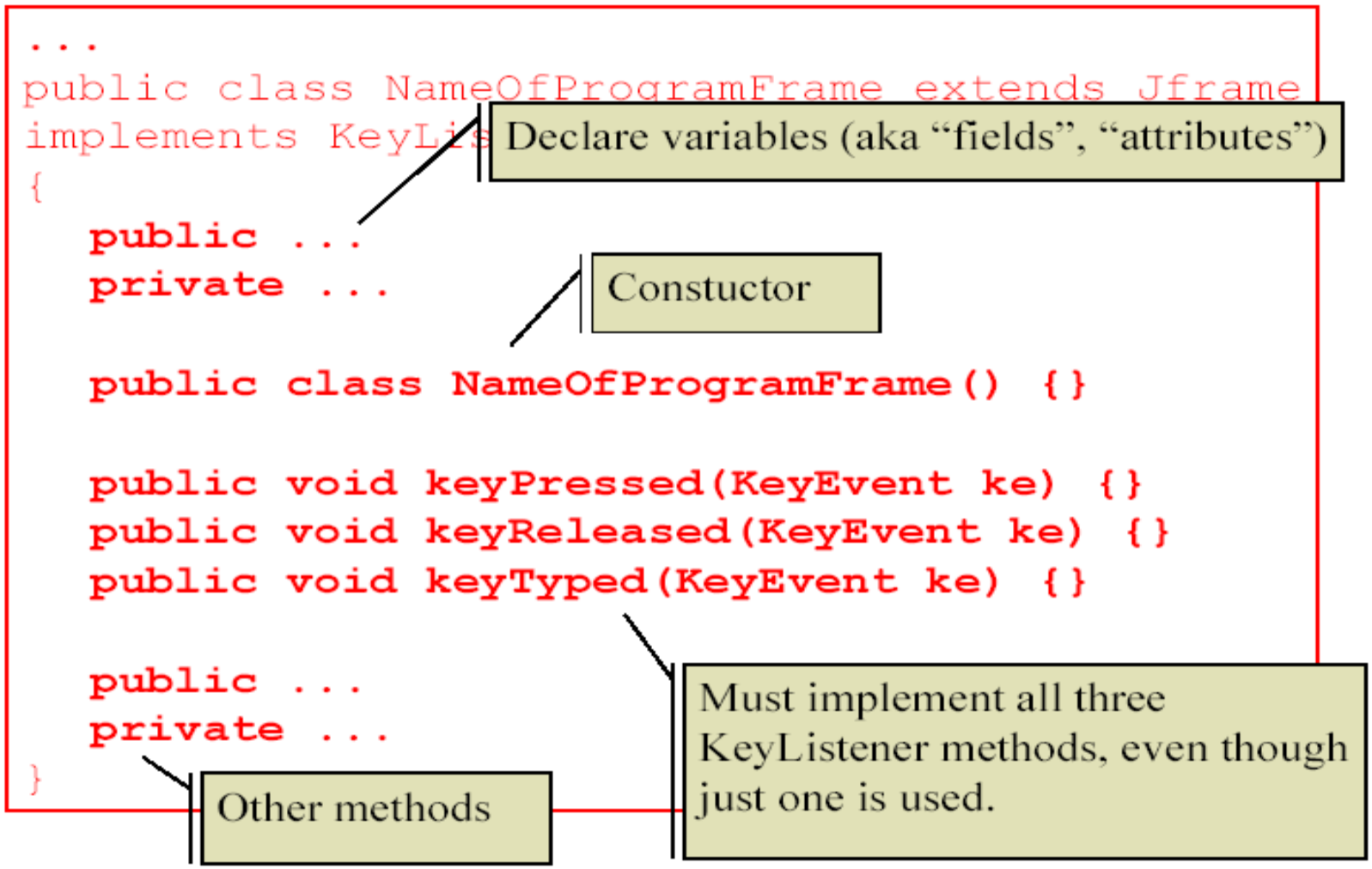
Our GUI frame class is actually a JFrame – extended and modified to suit our needs

# JFrame

- Class in the `javax.swing` package
- Sun's Swing toolkit is Java's most advanced toolkit
- Life before Swing…
  - AWT (abstract windowing toolkit)
    - AWT used "native" UI components (unique to local system)
    - This creates inconsistencies across platforms
  - UI components of AWT are now obsolete
  - AWT still used for drawing, images,etc.
- Swing paints the components itself
  - Advantage: code is consistent across platforms
  - Disadvantage: results in "big" programs (lots of memory!)
- Swing still uses many features of AWT

```
import ...

public class NameOfProgram
{
   public static voic main(String[] args)
   {

   }
}

public class NameOfProgramFrame extends Jframe
implements KeyListener
{

}
```

Our GUI class implements the methods of the KeyListener listener


...
public class NameOfProgramFrame extends Jframe
implements KeyLi
{
public ...
private ...
public class NameOfProgramFrame() {}
public void keyPressed(KeyEvent ke) {}
public void keyReleased(KeyEvent ke) {}
public void keyTyped(KeyEvent ke) {}
public ...
private ...
}
Declare variables (aka "fields", "attributes")
Constuctor
Must implement all three KeyListener methods, even though just one is used.
Other methods

# Frame Constructor

- Must…
  - Create and configure the GUI components
  - Install (“add”) listeners
    - Listeners are not just installed, they must be associated with particular GUI components
  - Arrange components in panel(s)
  - Either
    - Get the JFrame’s content pane
    - Add panel(s) to content pane
  - Or
    - Set the outer-most panel to be the JFrame’s content pane

# Listeners

- Java's listener classes are actually interfaces (not classes)
- What is an interface?

# Interfaces vs. Classes

- The definition of a class includes both the design of the class and its implementation
- Sometimes it is desirable only to design a class, leaving the implementation for later
- This is accomplished using an interface
- An interface contains only the design of a class
  - Includes signatures for its members (methods and fields)
  - No implementation provided

# Characteristics of Interfaces

- Do not have instance variables
- Cannot instantiate an object of an interface
- Include only abstract methods
- Methods have a signature (i.e., a name, parameters, and return type)
- Methods do not have an implementation (i.e., no code)
- Include only public methods and fields (does not make sense to define private members if the public members that could potentially use them are themselves not implemented)

# Listener Example

- The signature of our extended JFrame class includes the clause implements KeyListener
- This means our class must include definitions for the methods of the KeyListener listener
- Thus…

```
public void keyPressed(KeyEvent ke) {}
public void keyReleased(KeyEvent ke) {}
public void keyTyped(KeyEvent ke) {}
```

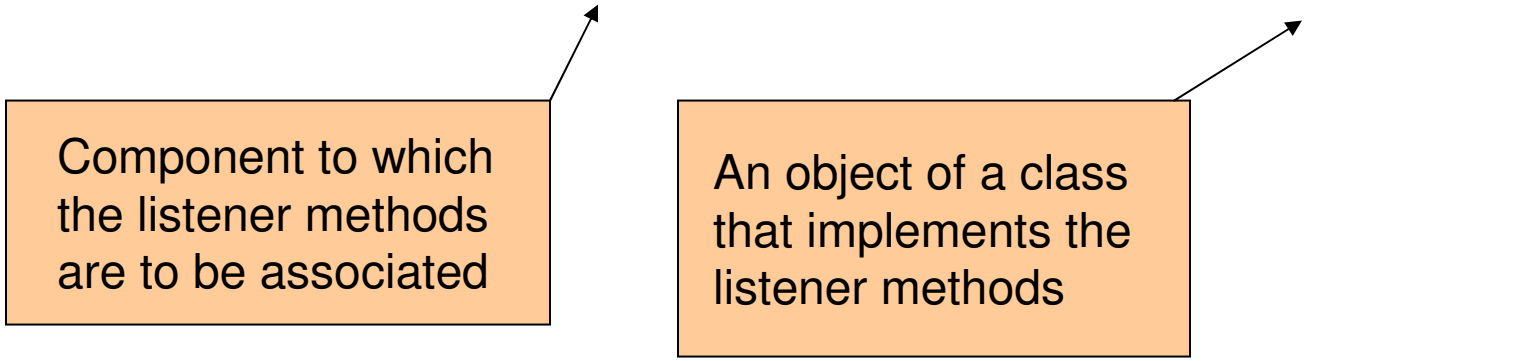
- Our implementation includes the code we want executed when a key is pressed, released, and/or typed
- Q: What is the difference between “pressed” and “typed”?

  A: Look in the API Spec!

# Installing Listeners

- It is not enough simply to implement the methods of a listener
- The listener must also be “installed” (aka “registered”,“added”)
- Furthermore, it must be installed for the component to which the listener methods are to be associated
- Thus (from our example program)

```
enterArea.addKeyListener(this);
```



# Installing Listeners (2)

- Consider the method addKeyListener
  - Fact #1: addKeyListener is a method of the Component class (check the API Spec)
  - Fact #2: enterArea (from our example) is an object (instance variable) of the JTextArea class
  - Fact #3: Through inheritance, a JTextArea object is a Component object
  - Conclusion: the addKeyListener method can be invoked on enterArea

# Installing Listeners (3)

- Signature for the addKeyListener method:
  **public void addKeyListener(KeyListener)**
- Description:
  **Adds the specified key listener to receive key events from this component.**
- In our example, we used this as the “specified key listener”
- Indeed, the current instance of our extended JFrame class (“this”) is a key listener because it implements the key listener methods
- Result: when a key pressed event occurs on the enterArea component, the keyPressed method in our extended JFrame class will execute!

# Let's Say That Again…

When a key pressed event occurs on the `enterArea` component, the `keyPressed` method in our extended `JFrame` class will execute!

# Processing Events

- Signature for the keyPressed method:

  **public void keyPressed(KeyEvent ke)**
- When our keyPressed method executes, it receives a KeyEvent object as an argument
- We use the KeyEvent object to
  - Determine which key was pressed, using
    - getKeyChar, getKeyCode, etc.
  - Determine the source of the event, using
    - getSource
- "Determine the source of the event" is important if there is more than one component registered to receive key events (not the case in our example program)

# Event Sources

- Java's event classes are all subclasses of EventObject (see earlier slide)
- EventObject includes the getSource method:

**public Object getSource()**

- Didn't need this in our example program, because only one object (enterArea) was registered to generate key events
- So, when the keyPressed method executes we know it is because a key was pressed in enterArea
- But, let's say we have two JTextArea components: enterArea1 and enterArea2 (next slide)

# Event Sources (2)

```
public void keyPressed(KeyEvent ke)
{
        if (ke.getSource() == enterArea1)
        {
                // code for enterArea1 key pressed events
        }
        else if (ke.getSource() == enterArea2)
        {
                // code for enterArea2 key pressed events
        }
}
```

# Adapter Classes

- What is an adapter class?
- A class provided as a convenience in the Java API
- An adapter class includes an empty implementation of the methods in a listener
- Programmers extend the adapter class and implement the methods of interest, while ignoring methods of no interest

# WindowAdapter (see Java API)

```
public abstract class WindowAdapter
implements WindowListener
{
        void windowActivated(WindowEvent we) {}
        void windowClosed(WindowEvent we) {}
        void windowClosing(WindowEvent we) {}
        void windowDeactivated(WindowEvent we) {}
        void windowDeiconified(WindowEvent we) {}
        void windowIconified(WindowEvent we) {}
        void windowOpened(WindowEvent we) {}
}
```

# Using the WindowAdapter Class

- Define an inner class that extends the WindowAdapter class
  - Implement the listener methods of interest
  - Ignore other listener methods
- In the frame constructor, use the appropriate “add” method to add an object of the extended WindowAdapter class
- In our example program…

**this.addWindowLisener(new WindowCloser());**

# Extending WindowAdapter

```
...
public class NameOfProgramFrame extends Jframe
implements KeyListener
{
...
        private class WindowCloser extends WindowAdapter
        {
                public void windowClosing(WindowEvent we)
                {
                        System.exit(0);
                }
        }
}
```

# Examples of Listeners and Adapters

| Listeners (# methods) | Adapters |
|---|---|
| KeyListener (3) | KeyAdapter |
| WindowListener (7) | WindowAdapter |
| MouseListener (5) | MouseAdapter |
| MouseMotionListener (2) | MouseMotionAdapter |
| MouseInput Listener (7) | MouseInputAdapter |
| ActionListener (1) | - |
| ItemListener (1) | - |
| FocusListener (2) | FocusAdapter |

(Note: MouseInputListener combines MouseListener and MouseMotionListener)

# Extending Adapters vs. Implementing Listeners

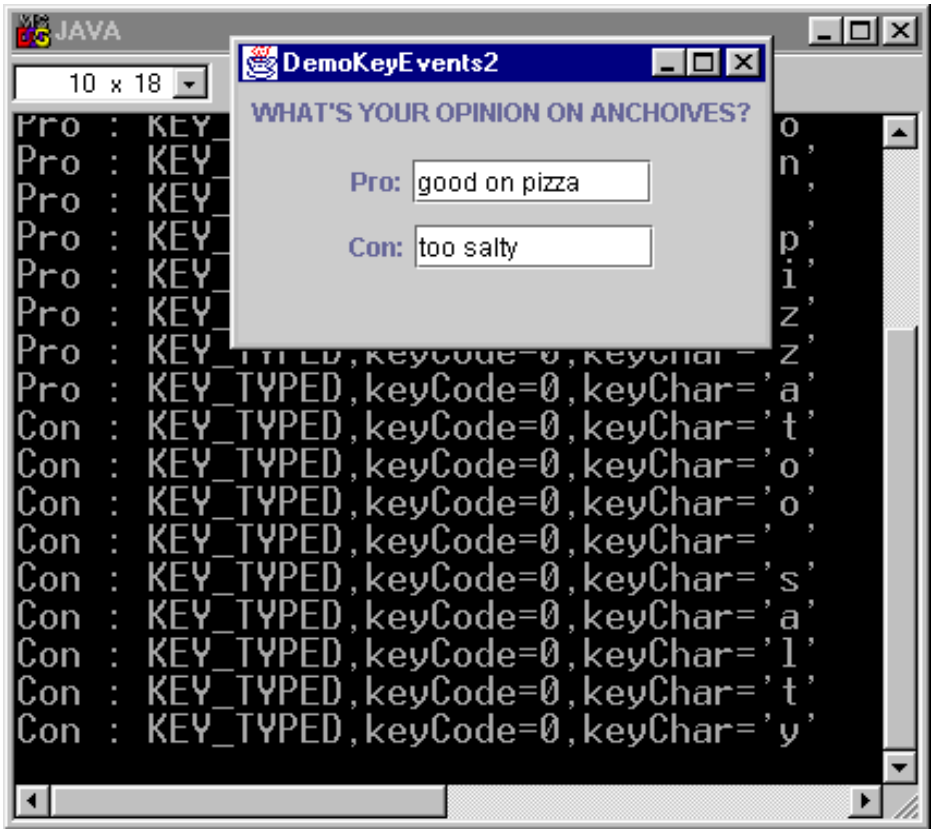
- Largely a matter personal choice
- Our example program does both
  - The KeyListener methods were implemented
  - The WindowAdapter class was extended
- Could have done the opposite, i.e.,
  - Extend the KeyAdapter class
  - Implement the WindowListener methods
- Note: a Java class can implement many listeners, but it can extend only one class
  - Java does not include multiple inheritance (unlike C++)

# Pros and Cons

- Using adapter classes
  - Advantage
    - Only the listener methods needed are defined
  - Disadvantage
    - A bit complicated to setup
    - Need to defined an inner class, then instantiate an object of the inner class to pass to the appropriate “add” method
- Implementing listener methods
  - Advantage
    - A class can implement many different listener interfaces
  - Disadvantage
    - Must implement all the methods defined in the listener (even those not used)

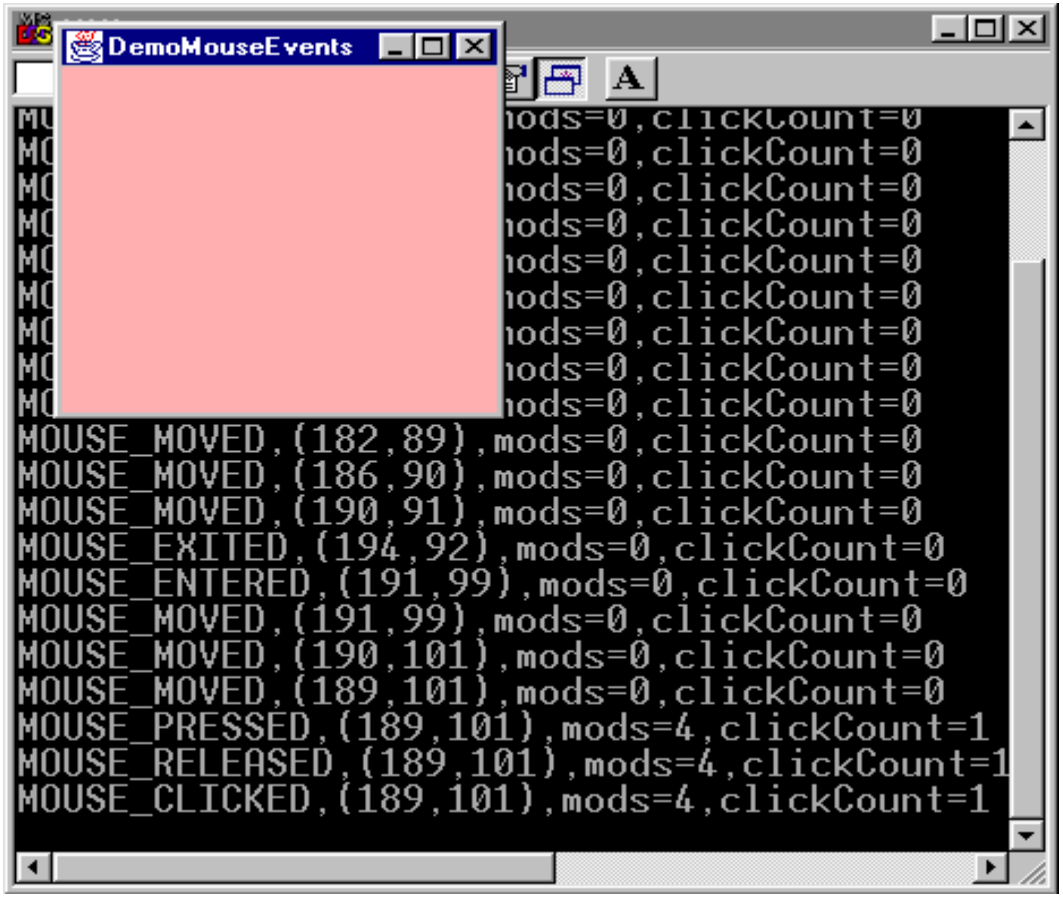
# Example Program

## DemoKeyEvents2.java



# Example Program

## DemoMouseEvents.java



# Example Program

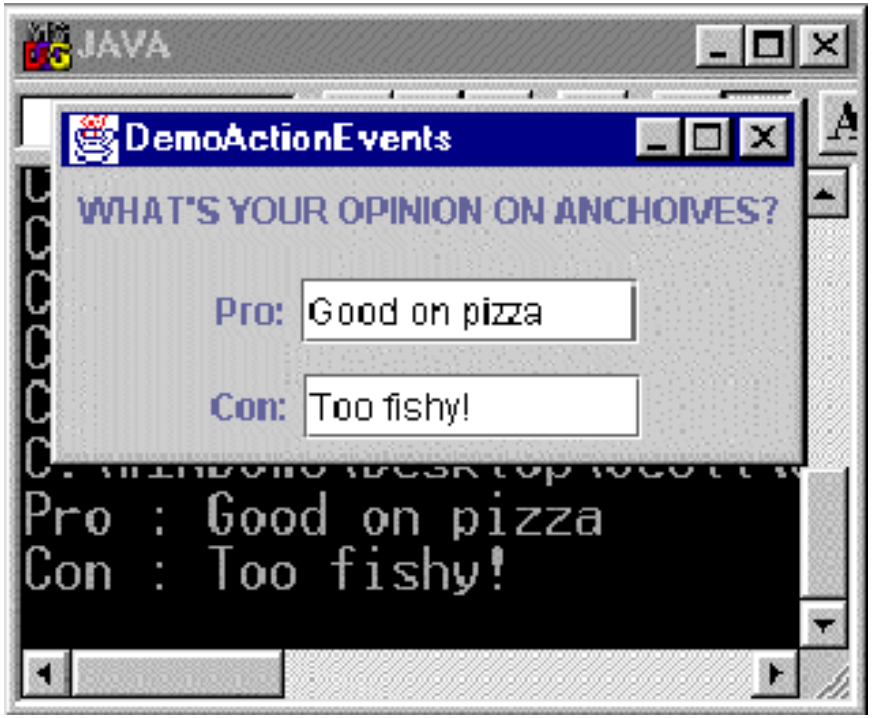
**DemoLowLevelEvents.java**

**DemoHighLevelEvents.java**

# Example Program

**DemoActionEvents.java**

**DemoFocusEvents.java**



# Outline

- Sequential programming
- GUI program organization
- Event-driven programming
- Modes

# Coping With Complexity

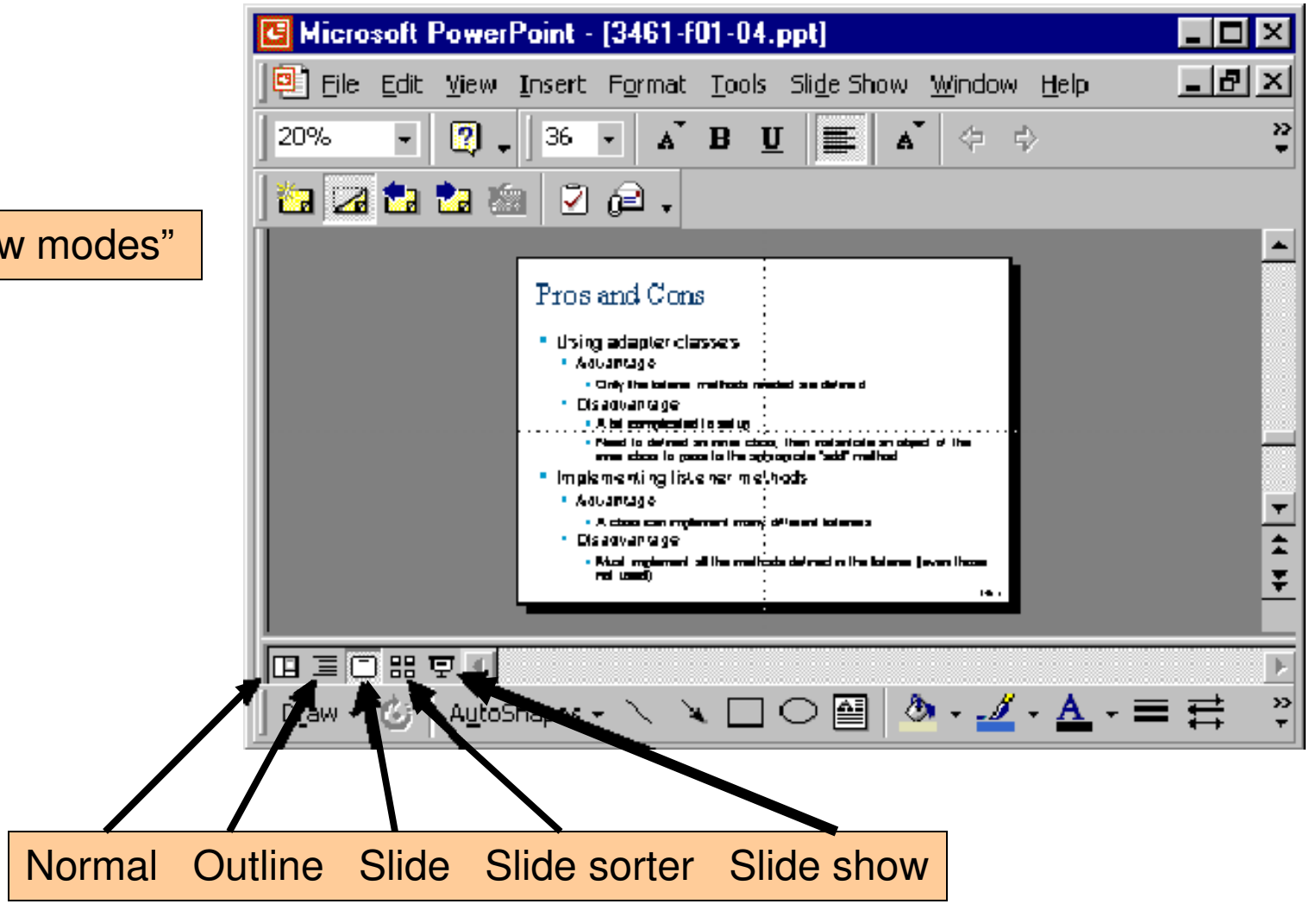
- How do we cope with complexity?
- Typically, the interface includes modes
- Each mode represents a different state of the system
- User input must be appropriate for the current state
- Moded systems require lots of variables to represent the state of the system

# Examples of Modes

- Draw programs
  - Use a mode to determine what is being drawn
  - E.g., line mode, rectangle mode, circle mode
- Universal remote controls
  - E.g., TV mode, VCR mode
- vi editor on unix
  - Insert mode, command mode
- Sport watches (Yikes! Too many modes!)

# Example - MS PowerPoint

Five “View modes”



# Modes in GUIs

- One simple way is to use radio buttons

# Example

## DemoTranslateEnglishGUI2.java

# Problems With Modes

- Confusing if too many modes (the user has to remember the salient ones)
- More modes make it easier for users to make errors (right command, wrong mode)
- Need feedback as to current mode (vi doesn't!).
- Need a mechanism to switch modes
- Modes do not scale well
- Need a more advanced model to simplify windows programming