CSE3461

Control Flow Paradigms:
Reacting to the User

Control Flow: Overview

- Definition of *control flow*:
  - The sequence of execution of instructions in a program.
  - Control flow is determined at run time by the input data and by the control structures (e.g., "if" statements) used in the program.
- In the case of *sequential control*:
  - Control starts are the first instruction in the main method
  - Control *flows* from the present instruction to the next one until the last one is reached, at which point the program terminates.

Control Flow: Revisited

- In more abstract terms, the *flow of control* is:
  - the thread(s) of execution in a software system, and their reaction to *exogenous* events, if any
  - An *exogenous event* is an event from outside the system (e.g., user input, disk space becomes full, network connection is lost)
- Two primary mechanisms for control flow:
  - Sequential
  - Event Driven

An Example of Sequential Control

Example 2.1:

DemoTranslateEnglishConsole.java

Notes about Code Examples

- Examples in this class SHALL NOT make use of the packages `type.io.*`, `type.util.*`
  - Use of these packages will not be allowed for the assignments
- Use the command line interface for compiling and invoking applications; use editor of your choice
  - use your own Integrated Development Environment (IDE) if you wish (e.g., Eclipse)

Notes about Ex 3.1

- Mechanisms that allows user to exit program:
  - provide "sentinel"
  - CTRL-Z is `null` in Windows (CTRL-D in Unix)

Sequential Programs (1)

- Typical flow of control:
  - Prompt the user
  - Read input from the keyboard
  - Parse the input (in order to interpret the user's action)
  - Evaluate the result
  - Generate output
  - Continue until application determines it is time to stop (or until user terminates application)

Sequential Programs (2)

- In sequential programs, control is held by the application:
  - the application decides when the user may perform input actions
  - application tells user whether it's ready for more input
  - user enters more input and it is processed
- Examples:
  - Command-line prompts (DOS, UNIX)
  - LISP interpreters
- The user is required to respond to the program
  - *Shouldn't it be the other way around? Shouldn't the program respond to the user?*

Sequential Programs (3)

Advantages
- Architecture is iterative (one step at a time)
- Easy to model using flowcharts or finite state automata
- Relatively easy to build

Limitations
- Difficult to implement complex interactions
- Only a small number of features may be practical to implement
- The sequence in which the interaction may proceed must be pre-defined

Event-driven Programs

- All communication from the user to the application occurs via *events*
- An *event* is an action that happens:
  - A mouse button pressed or released
  - A keyboard key is pressed or released
  - A window is moved, resized, closed, etc.
- Code is set up and waiting to handle these events

An Example of an Event-Driven Application

Example 2.2:

DemoTranslateEnglishGUI.java

Notes about Ex 3.2

The main method has only four lines:

```
public static void main(String[] args)
{
  DemoTranslateEnglishGUIFrame frame
     = new DemoTranslateEnglishGUIFrame();
  frame.setTitle("DemoTranslateEnglishGUI");
  frame.pack();
  frame.show();
}
```

```
import ...

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

   }
}

public class NameOfProgramFrame
{

}
```

Identify the packages containing classes used in the program

Only one public class per file

1. Construct a `JFrame`
2. Give it a title
3. Cause the frame to be sized
4. Cause the frame to be visible

Naming convention (just add "Frame" to name of program)

The core of the application is found here

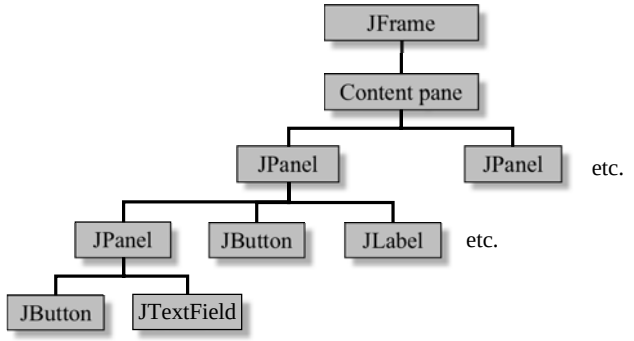

```
import ...

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

   }
}

class NameOfProgramFrame extends JFrame
{

}
```

Our instance of `JFrame` is actually an sub-class instance — it is a `JFrame` extended and modified to suit our needs

Basic Concepts

- The graphical user interface consists of components
  - all components are instances of `JComponent` (or child classes)
    - *Containers* (aka *non-atomic components*) can contain other components (e.g., `JTabbedPane`)
    - *Atomic components* cannot contain other components (e.g., `JComboBox`, `JButton`, `JLabel`)
- All components have a position within the containment hierarchy
  - atomic components can exist only as leaves, but not all leaves need to be atomic
  - a top-level container must be at the root of the hierarchy — the three top level components are `JFrame`, `JApplet`, and `JDialog`

Containment Hierarchy for JFC/Swing



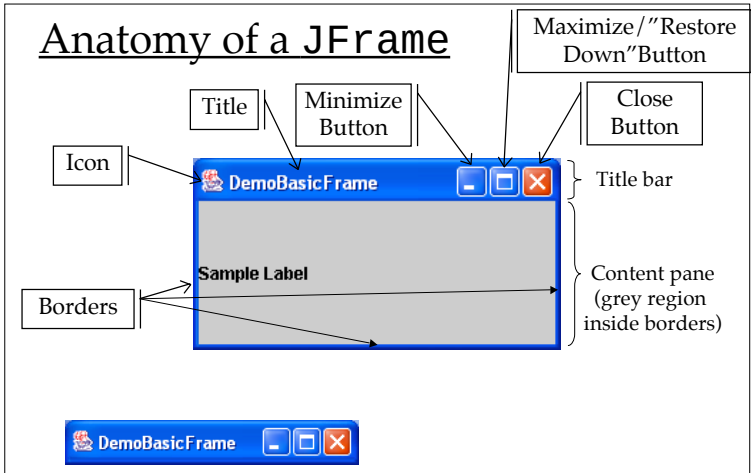
What is a `JFrame`?

1. It is a window
   - It has *window decorations*, such as borders, a titlebar and title, and buttons for closing and iconifying the window
   - The style of these decorations is derived from the "Look-and-Feel"
2. It is a *top-level container*
   - It has a *content pane* **and** a *menu bar*
     - The menu bar is optional
   - It is the **root** of a containment hierarchy

Instantiating the `JFrame`

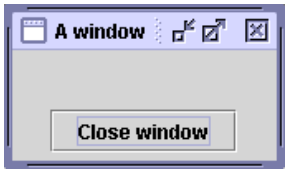
1. Must have a look and feel
   - if not explicitly specified, resort to default
2. Must specify reaction to *close* operation
   - otherwise, resort to default
3. Must add components to content pane
   - otherwise, content pane will not appear, frame will consist only of titlebar
   - must define components, add them to contentPane, then register listeners on them

Anatomy of a `JFrame`

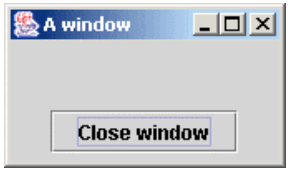


An instance of a `JFrame` without a content pane

Different Types of "Look and Feel" for a `JFrame`



Using Java's Look and Feel



Using MS Window 2000's Look and Feel

Using MS Window XP's Look and Feel

Instantiating the `JFrame`

1. Must specify look and feel
   - otherwise, resort to default

   [we'll use the dafault for the time being]
2. Must specify reaction to *close* operation
   - otherwise, resort to default
3. Must add components to content pane
   - otherwise, content pane will not appear, frame will consist only of titlebar
   - must define components, add them to contentPane, then register listeners on them

Examples

`DemoVeryBasicV1, V2, V3, V4 , V5`

- A trivial application to demonstrate some basic things about frames
  - three different versions
  - `V1`: behaviour for close action left as the default
  - `V2`: behaviour for close action specified using method from `JFrame`
  - `V3, V4, V5`: behaviour for close action specified using a method that is inherited from `Window`

Responding to the "Close" Operation

- An instance of a `JFrame` knows that something needs to be done when user performs a "close" on it
  - "close" performed on the frame when the close-application button or keystroke is used
    - rightmost button in top right-hand corner; Windows keyboard shortcut, ALT-F4
- The default behaviour for close is to *hide* the window (see `DemoVeryBasicV1`)
  - We'll next look at **two** ways to change this default

Responding to the "Close" Operation

Option #1 (`DemoVeryBasicV2`)

- Invoke the following method in the constructor of the `JFrame` subclass:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

(alternatively, can invoke the method on the instance of the `JFrame` in the main method)

- The default is:

```
setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
```

See the API for the class `JFrame` for this method

Responding to the "Close" Operation

Option #2 (DemoVeryBasicV3)

- Handle the operation with a registered `WindowListener` object.

```
addWindowListener( <instance of WindowListener> );
```

- This method is inherited from the parent class of `JFrame`, which is `Window`
  - therefore, need to import `java.awt.event.*`

DemoVeryBasicV4

```
class DemoVeryBasicFrame extends JFrame {
  public DemoVeryBasicFrame () {
    setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    addWindowListener(
      new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.out.println( "in windowClosing method" );
          System.out.println( e );
          System.out.println( e.getSource() );
          System.exit(0);
        }
      }
    );
  }
}
```

* This is advanced; this code defines a `WindowListener`. That's all you need to know for now…

How does this code work?

- The method `addWindowListener` **registers** the passed parameter on the instance of `DemoVeryBasicFrame`
  - the passed parameter is an instance of a `WindowListener`
- What is a *listener*?
  - A specialized piece of code that specifies what should happen for a *specific type of event* occurrence
  - By specific type, we don't just mean the *type of action*, but rather the *type of action with respect to a component*
  - We need to understand what events are and how they are created

What are Events?

- Each component in an application is a potential source of events
- When something happens, *an instance of an event object gets created* by built-in Swing code
  - Events are represented by objects
  - the instance itself contains information that identifies the source of the event
- An event *always has a source*
  - for now, we will assume that the source is a component — it "fires" the event

Java Events

- **Basic idea**: set up listeners to detect when events of interest have occurred
  - But the listening is *always connected* to a component
  - An application **registers** *listener objects* on the various components
- An event listener **must be** installed for each components from which events may be generated
  - otherwise, any events that are generated will pass undetected

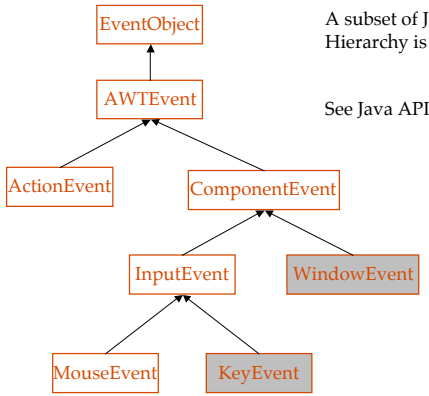
Types of Events

- There are two types of events: *low-level* and *semantic*
- A low-level event is:
  - a window-system occurrence, or
  - a low-level input (e.g., mouse button press, mouse button released, mouse button click (pressed and released), mouse cursor enter, mouse cursor exit, down, mouse up, key pressed, key released, key typed).
- A semantic event is any occurrence that is not a low-level event.

Types of Events

User Action	**Event that Occurs**
click a button	`ActionEvent`
press Enter while in a text field	`ActionEvent`
choose a menu item	`ActionEvent`
close a frame (main window)	`WindowEvent`
press a mouse button (while the cursor is over a component)	`MouseEvent`
move the mouse over a component	`MouseMotionEvent`
component becomes visible	`ComponentEvent`
component gets the keyboard focus	`FocusEvent`
Table or list selection changes	`ListSelectionEvent`

Java's Event Class Hierarchy



A subset of Java's Event Class Hierarchy is shown here

See Java API for full hierarchy

Example: `DemoVeryBasicV3`

1. The user presses the close button in the titlebar
2. This caused the creation of a `WindowEvent`
3. The source of the event is the instance of `DemoBasicFrame`
4. A `WindowListener` is registered on that instance; its `windowClosing` method gets invoked
5. The body of the `windowClosing` method invokes `System.exit(0)`

Comments

- Any window state change generates an event:
  - e.g., being opened, closed, activated or deactivated, iconified or deiconified
  - Registered listeners are notified
    - The relevant methods in the listener is invoked (depending on the type of `WindowEvent`).
    - The instance of the `WindowEvent` is always passed to the method.
- Other types of events might be generated
  - e.g., it the user mouses-down or presses keys in `DemoBasicFrame`
  - `MouseListener`, `KeyListener` not registered, so no methods get invoked

Comments

- What happens if *two* listeners are registered?
  - both listeners will "hear" the relevant events
  - the methods will be invoked in the order in which the listeners were registered
- Each event source can have multiple listeners registered on it.
- Conversely, a single listener can register with multiple event sources.

Instantiating the `JFrame`

1. Must specify look and feel
   - otherwise, resort to default
2. Must specify reaction to *close* operation
   - otherwise, resort to default
3. ➡ Must add components to content pane
   - otherwise, content pane will not appear, frame will consist only of titlebar
   - must define components, add them to contentPane, then register listeners on them

Example 2.4, four versions

`DemoBasicV1`, `V2`, `V3`, `V4`

- A trivial application to demonstrate some basic things about adding components to frames
  - four different versions
  - `V1`: adding directly to `contentPane`
  - `V2`: adding to `contentPane` via intermediate container, `JPanel`
  - `V3`: adding `ActionListener` to a component, using an *inner class*
  - `V4`: adding `ActionListener` to a component, without using an *inner class*

Adding Components

- To start, let's use `JLabel`
  - can display either text, an image, or both
  - does not react to input events
  - cannot get the keyboard focus

```
JLabel sampleLabel = new JLabel("Sample Label");
sampleLabel
  .setFont(new Font("sanserif", Font.PLAIN, 16));
sampleLabel
  .setPreferredSize(new Dimension(250, 100));
```

- The methods `setFont` and `setPreferredSize` are inherited from `JComponent`

How do we add `sampleLabel` to `frame`?

Approach #1

- access `contentPane` using method `getContentPane()`
- add `sampleLabel` to content pane directly

- e.g., `DemoBasic_v1`
  - default `LayoutManager` for `JFrame`'s `contentPane` is `BorderLayout`

How do we add `sampleLabel` to `frame`?

Approach #2

- Construct a new `JPanel`, say `panel`
- add `sampleLabel` to `panel`
- use method `setContentPane(panel)`

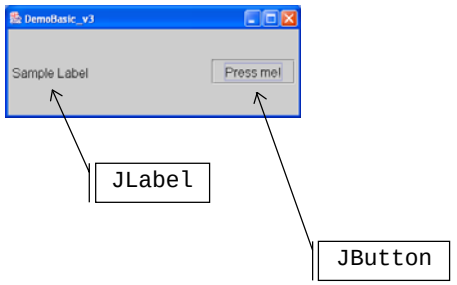
- e.g., `DemoBasic_v2`
  - default `LayoutManager` for `JPanel` is `FlowLayout`

Adding Other Components

- `JButton`
  - can display either text, an image, or both
  - an implementation of a "push" button
  - generates events
    - `ActionEvent` for mouse click
    - also `MouseEvent`, `KeyEvent`

```
toggleButton = new JButton("Press me!");
toggleButton
  .setFont(new Font("sanserif", Font.PLAIN, 16));
```


`DemoBasicV3`, `DemoBasicV4`



Registering an `ActionListener`

Register by invoking the following from within constructor `DemoBasicFrame` constructor:

```
toggleButton
  .addActionListener( <*****> );
```

NOTICE!!!!!

- The parameter `<*****>` must be an instance of an `ActionListener`
- How do we define an `ActionListener`?

Creating an `ActionListener`

- How do we define an `ActionListener`?
  - `ActionListener` is an interface
  - In fact, *all* of the various listeners (for all of the various types of events) are interfaces
  - Recall that an interface cannot be instantiated
- What can we do?
  - Interfaces can be implemented
  - The compiler enforces the rule that if a class `implements` an interface, it must provide bodies for *all* of the methods defined in the interface

Example of a Class Implementing an Interface

```
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) {}
}
```


More on this example…

- So the class `WindowAdapter` implements the `WindowListener` interface…
  - Compiler will enforce rule that the class must provides a body for each of the methods defined in the interface
  - But the compiler will allow a class to define all of the bodies to be empty!!!
    - Q1: What do we call such a class?
    - Q2: Why would we want such a class anyway?

More on Interfaces

- Q1: What do we call such a class?
  - An *adapter* class
  - The Java Foundation Classes (JFC) include several of these
- Q2: Why would we want such a class anyway?
  - we can *extend* an adapter class and *override* selected methods
  - it can be easier to do this with a adapter class from JFC than to implement the interface ourselves

Listeners and Corresponding Adapters

Listener interface (# methods)	Adapter class
`WindowListener` (7)	`WindowAdapter`
`ActionListener` (1)	**not defined**
Later we'll discuss…	
`KeyListener` (3)	`KeyAdapter`
`MouseListener` (5)	`MouseAdapter`
`MouseInputListener` (7)*	`MouseInputAdapter`
`ItemListener` (1)	**not defined**
`FocusListener` (2)	`FocusAdapter`

* `MouseInputListener` combines `MouseListener` and `MouseMotionListener`

Creating an `ActionListener`

`ActionListener` is an interface; it cannot be instantiated

Option **A**:
- create a *named inner* class that implements the interface
- e.g., `DemoBasicV3`

Option **B**:
- make the sub-class of `JFrame` implement the interface
- e.g., `DemoBasicV4`

Option **C**:
- create an *anonymous inner* class that implements the interface
- e.g., `DemoVeryBasicV4`

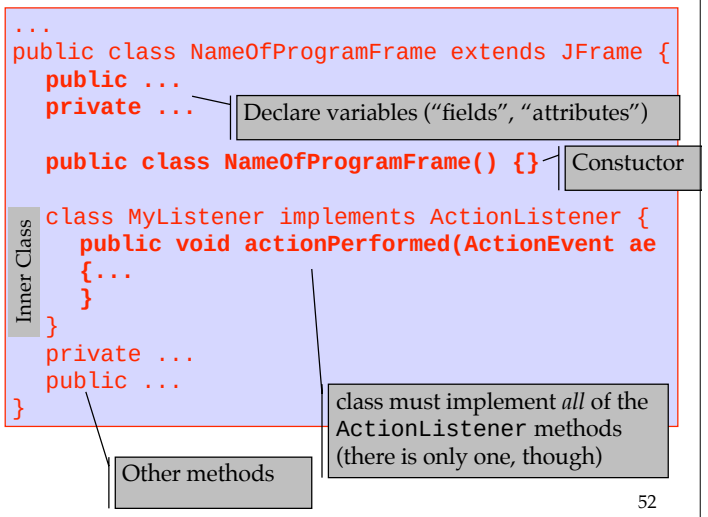
A: Using a Named Inner Classes (1)

What is an inner class?
- A nested class is a class that is a member of another class.
- A non-static nested class is called an *inner class*.

[The Java™ Tutorial, "Implementing Nested Classes"]

A: Using a Named Inner Classes (2)

To make use of a *named inner* class:
- define a class *within* the class definition for the `JFrame` sub-class
  - e.g., in `DemoBasicV3`, the class `MyListener` is defined within the class `DemoBasicFrame`
- design the class to *implement* `ActionListener`



A: Registering an `ActionListener`

- Suppose an `ActionListener` can be instantiated from a *named inner class*
- To *register* it, invoke the following:

```
toggleButton
  .addActionListener( new MyListener() );
```

**This was done in `DemoBasicV3`

B: Using the `JFrame` sub-class

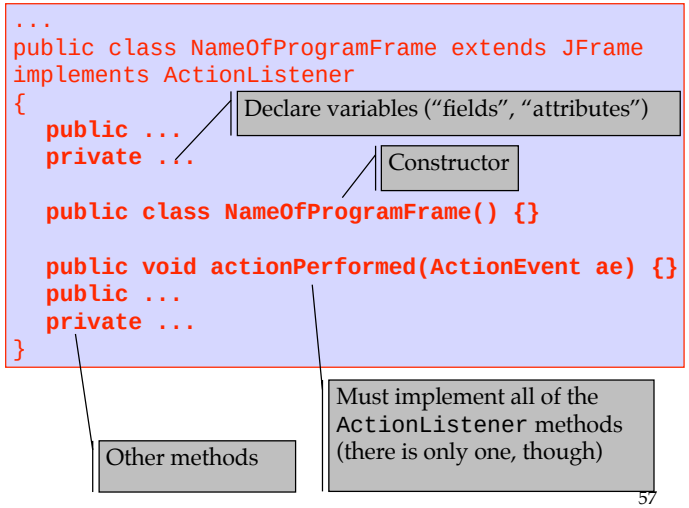
- In all of these examples, `DemoBasicFrame` was defined as a sub-class of `JFrame`
- What prevents us from also making `DemoBasicFrame` an `ActionListener`?

B: Using the `JFrame` sub-class

- How to make `DemoBasicFrame` an `ActionListener`:
  - need to assert that it implements the `ActionListener` interface
    - use keyword `implements` in class definition
  - need to ensure that method bodies are provided for all of the methods defined in the interface
    - the compiler will ensure this
  - e.g., in `DemoBasicV4`, the class `DemoBasicFrame` implements `ActionListener`


```
import ...

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

   }
}

public class NameOfProgramFrame extends JFrame
implements ActionListener
{

}
```

Our GUI class implements the methods of the `ActionListener` listener



B: Registering an `ActionListener`

- Suppose the constructor of the `JFrame` sub-class also instantiates an `ActionListener`
- To register it, invoke the following (from within constructor):

```
toggleButton.addActionListener(this);
```

**This was done in `DemoBasic_v4`

C: Using an Anonymous Inner Class

What is an *anonymous inner class*?

- an inner class that is declared without a name
- Syntax:
  `new <ClassOrInterfaceName>()`
  `{ <body> }`
- This can be used in places where an instance is needed

C: Using an Anonymous Inner Class

Finally, we revisit `DemoVeryBasicV3`:

- Recall we added a window listener to `DemoVeryBasicFrame`,
- the argument to the `addWindowListener` method was:

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

- this defines a `WindowListener` using an anonymous inner class

C: Using an Anonymous Inner Class

We can do the same for `ActionListener`:

```
new ActionListener() {
  public void actionPerformed(ActionEvent ae)
  { ... }
}
```

- this defines an `ActionListener`

**This was done in `DemoBasic_v5`

Discussion

- To make use of a particular listener:
  - should I extend its adapter class, or
  - should I implement the interface?
- If I implement the interface, which class should do it?
  - the `JFrame` child class,
  - a named inner class, or
  - an anonymous inner class?
- If I extend the adapter, should I do it with
  - an anonymous inner class, or
  - a named inner class?

Discussion

- When extending adapter classes:
  - need to provide code only for methods that are needed
  - need to define an additional inner class (either named or anonymous)
  - if you use an anonymous inner class, your code can be difficult to read
    - some say use sparingly, only for classes with one or two short methods; some say don't use at all
  - if you use a named inner class, you need to instantiate an additional object
  - you can only extend one adapter class (compare with next option)
    - Java does not include multiple inheritance (unlike C++ or Eiffel)

Discussion

- When implementing interfaces:
  - need to provide code for all of the methods
    - whether you need them or not
    - can define method bodies to be empty, though
  - don't need to define an additional inner class; you can use the `JFrame` subclass
  - a class can implement many different listeners

Whether to Extend Adapters or Implement Listeners…

- Largely a matter of personal choice
- Sample applications in this course will do both
  - in `DemoBasicV3`, `V4`, we implemented the `ActionListener` interface
    - note that no `ActionAdapter` class is defined in Java
  - `DemoVeryBasicV4` extended `WindowAdapter`
    - this was done using an anonymous inner class
    - you could easily define a named inner class instead (in fact, the code probably would be more readable this way)