CS112 Lecture: Graphical User Interfaces; Event-Driven Programming

Objectives:

1. To review the structure of the JComponent class hierarchy in package javax.swing
2. To introduce the Java LayoutManager classes
3. To review using the Container classes JFrame, JDialog, and JPanel
4. To discuss event-driven programming with multiple event sources
5. To introduce inner classes
6. To introduce creating and using menus

Materials:

1. Handout on swing package, and version to project of it
2. Demo program - ComponentAndLayoutManagerDemo.java, .class
3. Demo program - WindowDemo.java, .class
4. Demo program - PanelDemo.java, .class
5. Demo program - JTabbedPaneDemo.java, .class + separate handout
6. Demo programs - MultipleEvents1/2/3/4.java, .class + separate handout
7. Demo program - MouseEvents.java
8. Demo program - MenuOptionsDemo.java, .class

I. Introduction

   A. Some time ago, we took a first look at using the package javax.swing to create graphical user interfaces. Recall that we saw, at that time, that classes in the awt and swing packages fall into four basic categories.

      1. Windows and components that can be displayed in windows.

      2. Classes used to manage the layout of windows.

      3. Classes supporting adding menus to a window.

      4. Classes used to support graphics (e.g. the Graphics and Color classes we have used for drawing.

   B. Today, we are going to take a further look at some of these classes. In particular, we will:

      1. Consider the JComponent class hierarchy in more detail.

      2. Learn about some additional Container classes
3. Learn about LayoutManagers

4. Learn about the use of menus

II. The JComponent class Hierarchy, Containers, and Layout Managers

A. The class JComponent is the root of a hierarchy of classes that represent things that are visible on the screen.

1. This hierarchy has the following structure:

   HANDOUT, PROJECT - page 1

2. Each type of JComponent has a distinct visual appearance and behavior

   HANDOUT, PROJECT - page 3

   DEMO - ComponentAndLayoutDemo.class - show how each component looks/behaves using GridBagLayout window; demonstrate interaction with each.

B. Containers are used to hold other Components

1. The arrangement of Components within a Container is governed by a LayoutManager

   HANDOUT, PROJECT - pages 3-6

   DEMO - show effect of resizing the window with each kind of container

   HANDOUT, PROJECT - go over code on pages 6-8

   HANDOUT, PROJECT - discussion of various managers on page 9

2. Containers are of two basic types - Windows and Panels. Windows represent “top level” entities on the screen

   HANDOUT, PROJECT - page 10

   DEMO - WindowDemo.class - Show what happens while modal dialog is visible and after it is closed

   HANDOUT, PROJECT - code pn page 11
3. A Panel is used to group Components into subgroups within a larger Container, and for other purposes

HANDOUT, PROJECT - page 12

DEMO - PanelDemo.class - show what happens when the two windows are resized

CODE - page 13

C. There are several special classes in the Swing package that are useful in specific situations.

1. The JOptionPane class is used for simple dialog boxes. You have already made use of this class in previous labs/projects. In particular, using this class it is possible to create simple dialogs for

   a) Displaying a message

   b) Getting a String from the user

   c) Getting a yes/no or yes/no/cancel choice from the user

      You will use the message box option in Project 5 for displaying messages to the user. You will use yes/no choice dialog for the dialog that asks the user to confirm deletion of an individual from the address book. You will use the yes/no/cancel choice dialog for offering to save changes to the address book.

2. You have already made use of the JFileChooser class in Lab 11.

3. The JTabbedPane class can be used to create a screen with multiple panes selectable by tab.

PROJECT - DEMO - HANDOUT: JTabbedPaneDemo
III. Event-Driven Programming and Inner classes

A. At the heart of the way a user interacts with a GUI is the notion of *event-driven programming* - actions taken by the user (*gestures*) result in *events* that are handled by *listeners*.

1. A user gesture on a particular component results in the creation of an event object. The component that creates the object is called its *source*.

   *EXAMPLE:* When a user clicks on a button, an *ActionEvent* object is created. The *Button* object is the source of the event.

2. An object that is interested in responding to events must be registered with the source object by calling the source object’s `add___Listener()` method.

   *EXAMPLE:* To respond to action events originating from a button, an object must be registered with the button by a call to its `addActionListener()` method.

3. When an event occurs, the source calls the appropriate method of each object that is registered as a listener for that kind of event.

   *EXAMPLE:* When a button is clicked, the `actionPerformed()` method of each *ActionListener* registered with the button is called.

B. The java.awt package defines 12 different categories of events, of which 4 are primarily used internally by the awt. (We will discuss the other 8). Each event has its own kind of listener, with one or more appropriate methods.

*HANDOUT, PROJECT* - pages 15-16

C. One interesting question that arises is how events are handled when a given program has more than one event source.

1. One option is to have a single listener object that handles all events. In this case, it must check to see which source the event the comes from before deciding what to do with it.

   a) One way to do this is to use the `getSource()` method of the event object, and then compare it to known sources.

   *DEMO, SHOW* - MultipleEvents1.java
Note that the JButtons have to be instance variables, because they are needed both by the constructor and by the action listener (which has to compare the event source to each of them).

b) Another way -- commands

DEMO, SHOW - MultipleEvents2.java

Note that the JButtons can now be local variables in the constructor. (Preferred, to avoid cluttering the class)

Note that this approach relies on the fact that the event objects carry with them the label of the JButton that is their source. This also works for menu items, but it does not work for events in general.

2. An alternate approach - and a better one when there are many event sources - is to use a different listener object for each event source.

a) By using multiple, parameterized instances of a single listener class specifically created for this purpose

DEMO, SHOW - MultipleEvents3.java

(1) Note that the ColorChangeListener class is declared inside the main class. This is what is called an inner class, and is a capability added to Java 1.1 largely to support the “new” event model. Why does ColorChangeListener have to be an inner class here, rather than being a regular (“top-level”) class?

ASK

The actionPerformed() method has to be able to invoke the setBackground method() on the content pane of the frame. An instance of an inner class has access to both its own variables and method and those of the object which created it - which must be an object of the class in which it is contained.

(2) Note, further, that this inner class is declared as private. That means that objects of this class can only be created by objects of the class in which it is contained - which is appropriate.

(3) Finally, note that when the compiler compiles this source file, it creates two class files from it.

SHOW names of files in directory
b) By using multiple specialized listener classes, each created at the place where it is needed

**DEMO, SHOW** - MultipleEvents4.java

(1) Note that we are creating three different listener classes - one for each button - with one instance of each. Each actionPerformed method sets the frame to the appropriate color. This results in the compilation producing a total of four class files

*SHOW* names of files in directory

(2) Note that the classes we are creating are *local* - they are declared inside a method (just like local variables are). (Contrast this to the previous example, where the inner class was declared at class level, outside of any method.)

(3) Note that these classes we are creating are *anonymous*. Since each class is used to create exactly one object, and class declaration and object creation are done in the same statement, the class does not need a name.

(4) There are a number of specialized rules that apply to anonymous local classes, which we won’t go into here, except for noting one obvious point: since they are anonymous, they cannot have a constructor!

(5) Note also the formatting convention used for declarations of anonymous classes:

```java
new <base class or interface> () {
    final line of declaration has closing } followed immediately by whatever punctuation is needed to close the statement in which the new occurred (here “);”).
```

D. Mouse events are a particularly interesting kind of event, so it is worth spending a bit more time on them.

**SHOW, DEMO** MouseEvents.java

1. Note use of an anonymous class to extend JComponent to paint the Cheese.
2. Note two types of listeners needed - one for MouseEvents, one for MouseMotionEvents

3. Note how clicks are handled

4. Note how multiple clicks are handled

IV. Menus

A. Contemporary Graphical User interfaces are sometimes called “WIMP” interfaces - which is not a commentary on the people who use them! WIMP stands for “Windows, Icons, Menus, and Pointing Devices”. We have already discussed windows and the things displayed in them in detail, and pointing devices implicitly through our discussion of the events that various uses of the mouse can trigger (not just MouseEvents, but also events such as ActionEvents that are triggered by mouse actions such as clicking.) Icons are largely an issue for the operating system to deal with, not individual applications.

B. The final aspect of GUI’s that we need to discuss is Menus. Note that Swing allows a program to have two different kinds of menus (though rarely would a single program have both, except for demo purposes)

HANDOUT, PROJECT - page 17

DEMO, SHOW - MenuOptionsDemo.java, .class

Note that MenuItems can have ActionListeners just like buttons. Note that, in this case, they have been implemented as anonymous local classes, with each actionPerformed method calling an appropriate method of the main object.