CS112 Lecture: Graphical User Interfaces; Event-Driven Programming

Last revised 3/17/05

Objectives:

1. To introduce event-driven programming (including with multiple event sources)
2. To introduce the structure of the JComponent class hierarchy in package javax.swing, and the various container classes
3. To introduce the Java LayoutManager classes
4. To introduce inner classes
5. To introduce creating and using menus

Materials:

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

I. Event-Driven Programming

A. In the history of computer programming, there have arisen three major paradigms of program structure.

1. Batch

2. Interactive

3. Event-driven

These should not be viewed as one paradigm replacing the one before it - rather, there will continue to be problems for which - say - the batch paradigm is the best one.

B. Batch

1. The basic idea of the batch paradigm is this

a) All the input needed for a program is prepared ahead of time

b) When the program runs, it reads its input and produces its output, without any direct interaction with a human user.

c) Then the output of the program is used in some appropriate way.

2. A good example of an application where the batch paradigm is still the most suitable is running a weekly or monthly payroll.

   The input to the payroll system is timecard information, that is all accumulated before the program is run. The output of a payroll system is a set of paychecks that are then distributed to the employees.

   Other examples? ASK

3. Your Lab 9 basically followed this paradigm as well - a file of exam scores was prepared, then the program was run and statistics were printed to another file.

C. Interactive

1. The basic idea of the interactive paradigm is this: while the program is running, it interacts with a human user via text - e.g. the program displays information or questions and the user types commands or answers.

   The interaction is fairly structured - e.g. the program may ask the user a question and then wait for the user's response

2. An example of this paradigm is the process one goes through when making a reservation for an airline flight or a hotel or what have you. The person making the reservation must answer a series of specific questions (name, address, credit card number, etc)

3. Another example of this paradigm is computerized versions of various two-player games, in which the human player and the computer alternate making moves.

D. Event-Driven

1. The basic idea of the event-driven paradigm is that a unit of computation is initiated by the occurrence of an event on an *event source*. Typically, numerous computations occur during the course of the run of a program.

2. Many programs with graphical user interfaces follow this paradigm.

   For example, for Project 3 each button is a potential event source, and clicking a button initiates a computation that has some effect on the image being displayed. Moreover, the various menu options are also event sources, and can initiate computations such as saving the current state of the image of loading a new image.

3. But this paradigm is not confined to GUI programs. In terms of sheer number, most of the computers in the world are actually used in embedded systems, where the events represent things occurring in the physical world, such as:

   • A button on the control panel of a microwave oven (or a VCR or ...) being pressed.
   • The door sensor detecting that the door of a microwave oven has been opened, or the tape sensor of a VCR detecting that a tape has been put into it, or ...
   • A particular pedal being depressed in a car
   • A message arriving over a network connection to a router
   ...

E. We will discuss the event-driven programming paradigm in the context of graphical user interfaces in Java, for which Java provides a great deal of support. In fact, though, the same Java mechanisms can be used in other contexts - e.g. a "micro-edition" of Java can be used to program devices like cell phones. (In fact, Java was first invented for use in the development of TV set-top boxes).

II. 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. The main / initialization code of such a program typically sets up the graphical user interface and then terminates. All further computation takes place as the result of user gestures on various components.

   Example: we will use your project 2 as an example. As you recall, the applet code (which I gave you) had an `init()` method which created the clock(s) and the fields for input of the time. You may have needed to modify this code to create multiple clocks and / or adjust positioning.

   PROJECT: `init()` method of Project2.java

2. 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 pressed "return" on one of the time entry fields, an ActionEvent object is created. The text field object is the source of the event.

3. Each kind of event that is significant for a particular program is handled by some object called its *event handler*. An event-handler object must be registered with the source object by calling the source object's add____Listener() method.

   EXAMPLE: Note code for registering action listeners with the two input fields. (The same object is used for both, because the same action is taken regardless of which field the cursor is in when the user presses return)

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

   EXAMPLE: When the user signifies that new input has been entered by pressing return in a text field, the actionPerformed() method of the applet is called.

B. 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:

```
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 ");").

C. 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

III. Graphical User Interfaces

A. Throughout the course, we have worked with graphical user interfaces in various contexts, without actually saying much about the details. This was appropriate, because the GUI facilities are quite complex.. Now, though, we want to spend some time on using the GUI facilities of Java.

B. One of the distinctive features of Java is its built-in support for implementing graphical user interfaces. This is done through a portion of the standard Java library called the *abstract windowing toolkit* (often referred to as awt) and another portion - which builds on the awt - called Swing. It is possible to build significant GUI's using just the awt, but Swing provides much richer facilities. We will focus on using Swing in this course, though we will have to make some reference to portions of the awt that Swing uses.

1. The classes comprising the awt reside in the package `java.awt`, and those comprising Swing reside in the package `javax.swing`. To use Swing in a program, one normally includes the statement

   ```
   import javax.swing.*;
   ```

   and might also need

   ```
   import java.awt.*;
   ```

   and often.

   ```
   import java.awt.event.*;
   ```

   (Both awt and Swing make use of these classes.)

2. The awt and Swing are quite large - consisting of over 90 classes in the awt package and 120 in the Swing package in JDK 1.4, plus several subpackages with additional classes. We will only give a first introduction to them now.

C. The 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.)

IV. 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 situtations.

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 9.

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

   PROJECT - DEMO - HANDOUT: JTabbedPaneDemo

V. Support for Event Handling

A. As we have noted previously, the Java GUI facilities make use of event-driven programming to handle user gestures on components.

B. The `java.awt.event` package defines various different categories of events, of which some are primarily used internally by the awt. Most of the rest are used both by the awt and by Swing. The swing package `javax.swing.event` adds quite a number of events that are unique to swing components. Each event has its own kind of listener, with one or more appropriate methods.

HANDOUT, PROJECT - pages 15-16

Show contents page of `java.awt.event` and `javax.swing.event` packages .

VI. 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 <u>two</u> 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.