CPS122 Lecture: Graphical User Interfaces and Event-Driven Programming

Last revised February 25, 2022

Objectives:

1. To introduce the notion of a "component" and some basic Swing components (JLabel, JTextField, JTextArea, JButton, JComboBox)
2. To introduce the concept of Containers and layouts, including standard layout managers
3. To introduce NetBeans support for Free Design layout
4. To introduce use of event-driven programming with GUI widgets (including with multiple event sources)
5. To introduce inner classes
6. To introduce menus

Materials:

1. Dr. Java to demonstrate individual operations
2. Demo programs: ComboBoxDemo, Component and LayoutDemo, JPanelDemo, GUIEventsDemo, MultipleEvents1/2, MouseEvents, MenuOptionsDemo
3. Demo of ATM System on web
4. Projectables
5. NetBeans to demonstrate Free Design

I. Introduction

A. Today we will begin looking at creating and using graphical user interfaces (GUI's) in a program. This is a large subject, but after this series of lectures you should be able to create and use simple GUI's.

1. For iteration 1 of the team project, the GUI has been provided for you - you will just need to make some small additions for the "Return" and "Renewal" use cases.
2. But for iterations 2 and 3, you will need to add to the GUI provided for you.
3. This series of lectures will involve a lot of technical details concerning the Java libraries - which you will need to use both in lab and on your team project. There are a couple of important abstract concepts that you should also grasp from these lectures:

a) The overall structure of a GUI toolkit - of which the Java structure is just one, but all have some similar characteristics.

b) The notion of event-driven programming, which is a fundamental concept both with GUIs and in embedded systems.

B. A GUI performs two major tasks:

1. It displays information (graphics, text, and controls) on the screen.
2. It responds to user actions such as typing or clicking the mouse button.

II. **Introduction to Java GUIs**

A. One of the distinctive features of Java is its built-in support for implementing graphical user interfaces. This stands in contrast to the situation in some other languages where one uses a separate GUI toolkit on top of the language.

In Java, 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.

1. The classes comprising the awt reside in the package `java.awt`, and those comprising Swing reside in the package `javax.swing`.

a) To use Swing in a program, one normally includes the statement (PROJECT statements and fully-qualified forms)

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

and might also need

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

b) In addition, it may be necessary to import one or more *subpackages* - e.g.

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

(This package contains classes that are used for responding to user input. GUI-related actions performed by the user - e.g. clicking the mouse

button - result in the creation of special objects called *events* that the program can respond to. Both awt and Swing make use of these classes.)

c) Alternately, one can use fully-qualified names for the relevant classes - e.g.

```
javax.swing.JButton okButton =
new javax.swing.JButton("OK")
```

2. The awt and Swing are quite large - consisting of 98 classes plus 16 interfaces in the awt package and over 100 classes plus 25 interfaces in the Swing package in JDK 1.8 (and more in later versions), plus 11 subpackages of awt and 17 of swing. each with additional classes. We will only give a first introduction to them now, focussing on Swing (though we will also discuss an awt subpackage that Swing also uses).

3. Many of the visible components that are part of the Swing package have names that begin with capital J - e.g. JButtton, JLabel, etc.

   a) The J stands for the fact that the component is implemented by code written in Java.

   b) In contrast, awt components typically use the "native" components of the underlying platform.

   c) Actually, it is not uncommon to find that there is a "non-J" awt version of a component as well as a swing version - e.g. Button (awt) vs JButton (swing), etc - but this is not always true. The J is used even when there is no corresponding awt component.

   d) One important rule is to <u>never</u> mix awt and swing visible components in the same GUI! [However, swing makes some use of awt classes like LayoutManagers, which are not themselves visible. That's ok and unavoidable].

B. One of the fundamental classes in the swing package is the class `JComponent`. This class is the root of a hierarchy of classes that represent things that users can see in windows:

1. Subclasses representing individual GUI components - including five we will briefly introduce

   a) `JLabel`
   b) `JTextField`
   c) `JTextArea`
   d) `JButton`
   e) `JComboBox`

   The first of these is an output component - i.e. its only use is for displaying information for the user. The remaining four are primarily input components, though the second and third can also be used for output.

2. Containers - components that can themselves hold other components, and provide for physically arranging them through *layout managers*.

3. It is also possible to create one's own custom components - though this is beyond ouir current discussion.

C. We will demonstrate the various components using Dr. Java.

Setup - select Interactions pane, then type:

```
import java.awt.*;
import javax.swing.*;
JFrame f = new JFrame();
Container p = f.getContentPane();
p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
f.show();
```

D. A `JLabel` is a component that displays text on the screen.

1. The text in a label cannot be edited by the user.

2. A label is created by code like the following:

   ```
   JLabel prompt = new JLabel("Hello");
   ```

   Note that the constructor takes a parameter that specifies what the `JLabel` is to display.

   DEMO: the above code, then

   ```
   p.add(prompt);
   f.pack();
   ```

3. It is also possible to create a `JLabel` without specifying any text, and then specify the text later - e.g.

   DEMO (pause before final line)

   ```
   JLabel result = new JLabel()
   p.add(result);
   f.pack();
   ...
   result.setText("The answer is 42");
   f.pack();
   ```

E. A `JTextField` is a component that displays editable text on the screen.

1. In contrast to a `JLabel`, the text that is displayed in a `JTextField` can be edited by the user. The library class provides support for normal editing operations like placing the cursor, inserting and deleting characters, and cut and paste. (The program can also disable user editing and re-enable it later if desired.)

2. A `JTextField` is normally created by code like the following

   ```
   JTextField nameIn = new JTextField(10);
   ```

   where the integer specifies the number of characters to be displayed. (This is not an upper limit on the number of characters that can be typed, since the field will scroll if necessary.)

   DEMO: the above code, then

   ```
   p.add(nameIn);
   f.pack();
   ```

Note: it is also possible to specify the initial contents for the text field as a String. In this case, the size does not need to be specified, since it can be inferred from the initial contents; however, if a larger size is desired, it can be specified explicitly as well.

DEMO:

```
JTextField addressIn = new JTextField("Address");
p.add(addressIn);
f.pack();

JTextField cityIn = new JTextField("City", 40);
p.add(cityIn);
f.pack();
```

(Note how all components are stretched to match width required by widest - a consequence of the particular layout manager used.)

3. It is possible to access the current contents of a `JTextField` (i.e. whatever the user has typed in it) by using its `getText()` method.

   DEMO: put some text in the field, then

   ```
   nameIn.getText()       // No semicolon
   ```

F. A `JTextArea` is a text component that has multiple lines. One form of constructor allows specifying the size as rows and columns.

DEMO:

```
JTextArea area = new JTextArea(4, 40);
area.setAlignmentX(0)
p.add(area);
f.pack();
```

As with a `JTextField`, it is possible to use `getText()` to get the contents of a text area. (The result will be a single long string, without newlines dividing the lines).

DEMO: Fill in the four lines, then

```
area.getText()      [ no semicolon ]
```

G. A `JButton` is a component that a user can click to request that some particular action occur.

1. A `JButton` is typically constructed as follows:

   ```
   JButton ok = new JButton("OK");
   ```

   where the string specified is the name that appears inside the button

   DEMO: the above code, then

   ```
   p.add(ok);
   f.pack();
   ```

2. When we talk about GUI events later in the lecture we will talk about how to associate an action with a button.

H. A `JComboBox` is a component that a user can click to request that some particular action occur. We will use a demonstration program based on one created by the authors of a former textbook

   DEMO: ComboBoxDemo

1. Constructing a `JComboBox` is more complex than constructing other types of component, because one must specify the various values to be listed as well as creating the component, and may also specify an initial value

   PROJECT: Code that constructs the combo box

```
//construct combo box

speedChoice = new JComboBox();

// Add 3 entries
speedChoice.addItem ( "Slow" );
speedChoice.addItem ( "Medium" );
speedChoice.addItem ( "Fast" );

// Display "Medium" initially
speedChoice.setSelectedItem( "Medium" );

// this class is listener
speedChoice.addActionListener ( this );
```

2. Again, when we talk about GUI events later in the lecture, we will learn how the program can respond to events generated when an value is selected.

III. Introduction to GUI Containers

A. In the world of GUIs, a container is a special kind of component whose basic task is to hold and position other components. (The term "container" is used for other things in a different context)

1. Top level windows (`JFrames`) and applets (`JApplets`) have a container called the *content pane* that holds their actual contents.

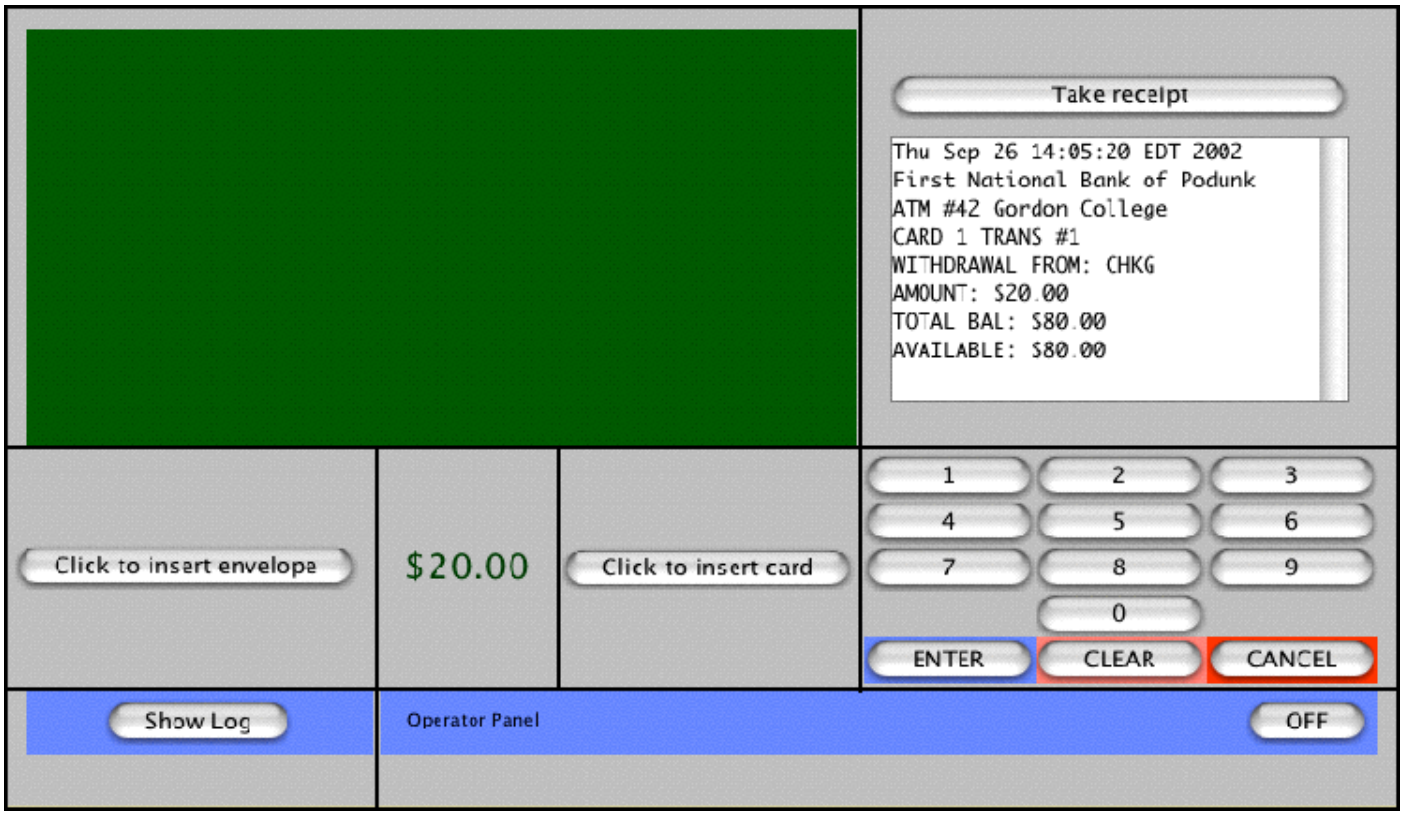
2. Another kind of container is a `JPanel`, which can be used to group components in another container - a sort of “window within a window”.

B. An overall GUI often consists of a container that includes other containers for grouping the components.

Example: The ATM system

1. Demo operation via link on course web site

2. Project panel structure



C. A key task of a container is to manage the layout of its components - i.e. where each component appears on the screen, and how much screen space is allocated to it.

1. The standard way to do this is through a special object associated with the container called a *Layout Manager*. The `java.awt` package includes a number of different kinds of layout managers that implement different policies. The `java.swing` package defines a couple more. Layout managers can do some very sophisticated layout work, but they are complicated to use. We will look at some of the rudiments of using them shortly.

2. An alternative is called *absolute positioning* - in which we explicitly specify the position and size for each component.

   a) This is a much simpler approach in simple cases.

b) It is not, however, the recommended approach for most programs for two reasons:

(1)Absolute positioning is somewhat dependent on details about the underlying platform and display device used to show the GUI, whereas the standard layout manager objects handle this automatically. As a result, it is not uncommon to get a complex GUI looking good on one platform, only to have text cut off or alignments messed up on a different platform.

(2)Absolute positioning is not responsive to changes in the size of a window resulting from resizing by the user, but the standard layout managers handle this as well.

c) Here is a simple example of the use of absolute positioning. (We will also use this program as a demonstration for the lecture section on events)

DEMO
PROJECT: GUIEventsDemo.java constructor

```
public GUIEventsDemo()
{
        // For simplicity, we'll use absolute positioning
        getContentPane().setLayout(null);
        setSize(WIDTH, HEIGHT);
        setResizable(false);

        numberPrompt = new JLabel("Please enter a positive number");
        getContentPane().add(numberPrompt);
        numberPrompt.setBounds(50, 50, 200, 20);

        numberInput = new JTextField(20);
        getContentPane().add(numberInput);
        numberInput.setBounds(270, 50, 80, 20);

        computeButton = new JButton("Compute square root");
        getContentPane().add(computeButton);
        computeButton.setBounds(100, 150, 200, 20);
```

```
        result = new JLabel("", SwingConstants.CENTER);
        getContentPane().add(result);
        result.setBounds(50, 250, 300, 20);

        // Prepare to handle mouse click on the compute button

        computeButton.addActionListener(this);
    }
```

D. NetBeans supports a free design mode, in which NetBeans actually creates the appropriate layout managers. We will work with this in a later lab, but for now let's do a simple demo.

1. Create a NetBeans project called Design Demo (Ant) (create main class not checked)
2. Create a Swing JFrame form
3. Add a Label: change text to This says something and name to `somethingLabel`

   (Note that NetBeans calls this a Label, though it actually uses the class JLabel. The same is true for other widgets)
4. Add a TextField: change text to empty and name to `inputField`. Drag to make big.
5. Add a Button: change text to OK and name to `okButton`
6. Run it.

   Note that it doesn't do anything yet
7. Add an event handler to the button so that it changes the label text to "You typed" + the field contents

   DEMO

   This leads into a topic we will consider shortly: event handling. But first, we want to spend some time discussing the use of layout managers.

IV. **Layout Managers.**

The sophisticated way to lay out a container is to make use of a LayoutManager. The awt package defines five kinds of layout manager; swing defines three more general ones plus various specialized ones used by different kinds of component.

DEMO: ComponentAndLayoutDemo

A. A FlowLayout lays out a container by "flowing" components across the width, and then going to a second row if necessary.

1. DEMO - note originally all in one line, then resize and show effect

2. A `FlowLayout` is specified by using one of the following

```
container.setLayout(new FlowLayout());
container.setLayout(new FlowLayout(align));
container.setLayout(new FlowLayout(align, hgap, vgap));
```

[Where align is one of `FlowLayout.LEFT`, `FlowLayout.RIGHT`, or `FlowLayout.CENTER`, hgap and vgap specify spacing between adjacent components horizontally and vertically]

3. A distinctive feature of a `FlowLayout` is that it gives each component exactly the amount of space it needs, and no more. It doesn't "stretch" components the way some other layouts do

DEMO: Contrast with `BorderLayout` example

B. A `BorderLayout` lays out a container in terms of five positions, designated `North, South, East, West,` and `Center`.

1. A `BorderLayout` is specified by using one of the following:

```
container.setLayout(new BorderLayout());
container.setLayout(new BorderLayout(hgap, vgap));
```

2. When a component is added to a container that uses a border layout, special form of the add method is used, in which the second parameter is one of `BorderLayout.NORTH, BorderLayout.EAST, BorderLayout.SOUTH, BorderLayout.WEST` or `BorderLayout.CENTER`..

   Example: Show code for adding components to `BorderLayout` version of demo

3. A consequence of this is that a container that uses a `BorderLayout` may directly show only five components - though any of these may be a Panel that itself contains several components.

4. A `BorderLayout` "stretches" individual components - e.g. the North and South components are stretched to match the bigger of the North, South, or combined width of the West, Center, and East; the West, Center and East components are stretched to match the height of the biggest of the three.

   NOTE in demo; demo resizing

C. A `GridLayout` lays out components on a grid, whose size is specified when the layout is constructed.

1. A `GridLayout` is specified by using one of the following::

   ```
   container.setLayout(new GridLayout(rows, cols));
   container.setLayout(new GridLayout(rows,cols,hgap,vgap));
   ```

   [Either rows or cols - but not both - can be zero - which means "use as many as are needed". The gaps specify the amount of space allowed between adjacent cells.]

2. When components are added to the container, they are placed in cells by filling the first row, then the second, then the third ...

3. All the grid cells will be of the same size, determined by the component with the greatest width and the component with the greatest height. All other components will be "stretched" to fill their cell

   NOTE IN DEMO; demo resizing

D. A `BoxLayout` can be used to lay out components in either a single vertical or horizontal line. It is similar to `FlowLayout`, except that the components remain in a single line when it is resized.

1. A `BoxLayout` is specified by using:

   ```
   BoxLayout layout = new BoxLayout(container, axis);
   ...
   container.setLayout(layout);
   ```

   [where the axis is typically either `BoxLayout.X_AXIS` or `BoxLayout.Y_AXIS`]

2. When components are added to the container, they are placed left to right or top to bottom in the order in which they are added.

   DEMO, including resizing

E. The most sophisticated layout manager by far is the `GridBagLayout`.

1. Like the `GridLayout`, the `GridBagLayout` lays out components in cells on a grid. However, each row in the grid can have a different height, and each column a different width - as determined by the highest or widest component in any given row/column.

2. Each component added to a container that uses this `LayoutManager` has a constraints object that specifies things like which cell (or cells) it goes in; whether it is to be stretched horizontally or vertically to fill its cell; where it is positioned in its cell (corner, side, center) if it is smaller than the cell, etc.

3. We won't discuss the details of using this class - but it is instructive to look at the example.

   DEMO, including resizing

4. Netbeans provides good support for using this kind of layout as well.

F. A `CardLayout` shows just one component at a time.

1. A `CardLayout` is specified by using one of the following

```
CardLayout layout = new CardLayout());
CardLayout layout = new CardLayout(hgap, vgap));
...
container.setLayout(layout);
```

   [where the gaps specify space around each component]

2. Components are added to the container using a form of the add method in which the second parameter is a `String` giving a name to the component.

3. The layout object itself supports methods that allow the program to specify which component is to be shown:

```
first(container);
last(container);
previous(container);
next(container);
show(container, name);
```

   PROJECT while loop in demo program that shows a different card every five seconds

```
while (true)
{
   synchronized(cardLayoutFrame)
   {
         try
         { cardLayoutFrame.wait(5000); /* Delay 5 seconds */ }
         catch (InterruptedException e)
         { }
   }
   cardLayout.next(cardLayoutFrame.getContentPane());
}
```

G. Panels.

1. Sometimes, in constructing a GUI, it is desirable to group several components into a single entity for layout. There are several reasons why this might be the case

   a) We want to ensure that they stay together, even if the overall window in which they appear is resized

   b) We want to use a different layout for positioning them relative to one another than is used for the overall layout

   c) We want to protect a component from being "stretched" by a layout manager.

2. For such purposes, we can make use of a kind of container known as a `JPanel`. A `JPanel` is a container that has its own layout, but can be placed in another layout.

V. Event-Driven Programming with GUIs

A. We said at the outset that GUI's perform two basic tasks: displaying information and responding to user input. We now turn to the handling of the second task.

B. Any windowing operating system has a software component that responds to changes in the position of the mouse, pressing or releasing the mouse's button(s), and pressing of keys. Each such action by the user constitutes an *event*, which this component of the OS *delivers* to the appropriate application (namely the application that owns the window that the cursor is over when the event occurs.) At this point, further processing of the event is up to the application.

DEMO: GUIEventsDemo

PROJECT code for adding event listener and actionPerformed method

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

public class GUIEventsDemo extends JFrame implements
ActionListener
{
   /** Constructor - create the GUI
    */
    public GUIEventsDemo()
    {
         ...
        // Prepare to handle mouse click on the compute button

        computeButton.addActionListener(this);
    }

    /** Method invoked when the user clicks the compute button
     */
    public void actionPerformed(ActionEvent e)
    {
        // Get the number the user typed:

        String numberTyped = numberInput.getText();

        // Convert it to a float - code borrowed from Wu
        Double doubleObj = new Double(numberTyped);
        double value = doubleObj.doubleValue();

        // Compute its square root and put it in the result label

        double sqrt = Math.sqrt(value);
        result.setText("Square root is " + sqrt);
    }
```

1. In the case of Java, the Java awt provides a standard mechanism for handling events that any program can build on.

    a) This is used by both Swing and awt (since Swing ultimately is built on top of awt)

b) To use the event handling mechanism, a program must import the package `java.awt.event`.

2. SHOW in projected code

3. When a Java program receives an event, the Java library delivers it to the appropriate GUI component - e.g. to the `JButton` object if the mouse is over a button; to the `JTextField` object if the mouse is over a text field, etc.

   a) A given type of component may handle certain types of events on its own - e.g. a key pressed event that is delivered to a text field object causes the character that was typed to be inserted in the text at the appropriate point.

   b) User-written software may also express an interest in handling a particular type of event by *registering* an *event listener* with the component. When an event listener is registered, and the appropriate type of event occurs, the event listener is activated to responds to it.

      Events that the component is not interested in and that have no registered listeners are simply ignored. For example, every mouse movement results in an event, but the vast majority of them are ignored. (One could register an interest in mouse movements, however, if one wanted to highlight some component on the screen when the mouse was moved over it.)

   c) Java has its system for classifying types of events. We talk primarily about one type, but will mention a couple of others as well.

      (1)An `ActionEvent` is created whenever a user does something that typically calls for an active response from the program - e.g.

         (a)Clicks a button

         (b)Presses return while typing in a text field

(c)Chooses a selection in a combo box

(d)Chooses a menu item
etc.

(2)A `KeyEvent` is created whenever a user presses a keyboard key.

(3)A `ChangeEvent` is created whenever a user changes the value of a slider.

(4)A `MouseEvent` is created for various mouse actions - pressing, releasing, clicking, moving, dragging, entering or leaving a component. (The various methods of WindowController in object draw such as onMousePress() provide a simplified way of working with these)

C. To register an event listener with a component, one uses a listener object that implements the appropriate interface. In the case of an `ActionEvent`, a listener object must

1. Be declared as implementing `ActionListener`. `ActionListener` is an *interface* in Java - a specification for behavior that all objects that implement the interface must have. In this case, the necessary behavior is having an `actionPerformed()` method that handles an action event in an appropriate way.

2. Have a method with the following signature:

   ```
   public void actionPerformed(ActionEvent e)
   ```

3. Be registered with the component. This is done by some code (typically the constructor for the GUI) sending an `addActionListener` message to the component, passing as a parameter the listener object.

   PROJECT: show each of the above in projected code.

Note that the program only deals with ActionEvents; the JTextField object handles KeyEvents without the program having to deal with them.

D. When an event occurs for which there is an appropriately registered listener, a suitable method of the listener is called. In the case of an ActionEvent, this is the `actionPerformed()` method of the listener object. The actual event is represented as an object that is passed as a parameter to this method. (The event object contains information that varies from type of event to type of event, but typically includes the precise coordinates where the cursor was when the event occurred, details about any modifier keys that were pressed, etc.) The `actionPerformed()` method is responsible for doing what needs to be done to respond to the event.

1. PROJECT: code for `actionPerformed()` in `GUIEventsDemo`

2. If we also wanted the user to be able to initiate computation by clicking return after typing in the text input box, we could add the following code to the constructor, making the applet be a listener for action events emanating from *either* the text field *or* the button.

   ```
   numberInput.addActionListener(this);
   ```

   In this case, an action event emanating from either component would activate the applet's `actionPerformed()` method. If the applet needed to know which component was the source of the event, it could find out by examining the `ActionEvent` object passed as a parameter. In this case, though, it doesn't - we need to do exactly the same thing in either case.

   DEMO: Show before nothing happens when we press return in input field; add line at end of main method:

   ```
   numberInput.addActionListener(this);
   ```

show changed behavior when pressing return

3. Note how the action listener reads the number typed by the user as a string and then uses the "wrapper class" approach to convert it to a number that can be used in computation. It is instructive to see what happens when a "bad" value is typed.

   DEMO: with -1 as input

   DEMO with abc as input

E. We can now summarize how a GUI program deals events

1. The main / initialization code of such a program typically sets up the graphical user interface and then terminates. This includes creating the needed components and - if appropriate - registering event handlers for them.

   Example: PROJECT excerpts from `GUI` class for Library project (automatically generated by NetBeans) showing adding action listener to save item in initComponents() and code for actionPerformed method that is called.

   All further computation takes place as the result of user gestures on various components.

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 one chooses one of the menu items, an ActionEvent object is created. The item that was clicked on is the source of the event.

3. When an event occurs, the the appropriate registered listener method of the source object is called. (If there is no such method, the event is ignored)

   EXAMPLE: When the user chooses a menu item, the `actionPerformed()` method of the registered ActionListener is called.

F. 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. We can use the getSource() method of the event object, and then compare it to known sources.

   DEMO MultipleEvents1.java

   PROJECT Code

   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)

```
public void actionPerformed(ActionEvent event)
{
    Object source = event.getSource();
      if (source == redButton)
           getContentPane().setBackground(Color.red);
      else if (source == greenButton)
           getContentPane().setBackground(Color.green);
      else if (source == blueButton)
           getContentPane().setBackground(Color.blue);
      repaint();
}
```

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. One way to do this is with anonymous classes, each created at the place where it is needed

   DEMO MultipleEvents2.java

   PROJECT Code

```
redButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event)
   {
        getContentPane().setBackground(Color.red);
       repaint();
   }
   });
buttonPanel.add(redButton);

JButton greenButton = new JButton("Green");
greenButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event)
   {
              getContentPane().setBackground(Color.green);
       repaint();
   }
   });
buttonPanel.add(greenButton);

JButton blueButton = new JButton("Blue");
blueButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event)
   {
        getContentPane().setBackground(Color.blue);
       repaint();
   }
   });
buttonPanel.add(blueButton);
```

   a) 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

b) Note that the classes we are creating are *local* - they are declared inside a method (just like local variables are).

c) 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.

d) 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!

e) 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 ");").

f) When using the Netbeans facility for adding Events to objects, the IDE takes care of all this

DEMO - Free Design Demo Project - add a new button called "Clear" that that clears the text in the label. (Specify text as " " to avoid repacking the frame).

SHOW generated code for adding ActionListener

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

DEMO MouseEvents.java

PROJECT excerpts from code

```
redButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event)
   {
         getContentPane().setBackground(Color.red);
       repaint();
   }
   });
buttonPanel.add(redButton);

JButton greenButton = new JButton("Green");
greenButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event)
   {
              getContentPane().setBackground(Color.green);
       repaint();
   }
   });
buttonPanel.add(greenButton);

JButton blueButton = new JButton("Blue");
blueButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event)
   {
         getContentPane().setBackground(Color.blue);
       repaint();
   }
   });
buttonPanel.add(blueButton);
```

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

H. We have introduced event-driven programming in the context of GUIs, but the event-driven approach is also used in many other places - e.g. embedded control systems.

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. Swing actually allows a program to have <u>two</u> different kinds of menus (though rarely would a single program have both, except for demo purposes)

1. "awt" menus, that follow the conventions of the native platform (e.g. on the Mac, an awt menu appears at the top of the screen)

2. "swing" menus that follow the conventions of swing (always at the top of the window)

C. Either way, the basic approach is the same

1. We create a menu bar object - class MenuBar or JMenuBar

2. We create menus - class Menu or JMenu. - and add each to the menu bar

3. We create menu items - class MenuItem or JMenuItem - and add each to its menu.

4. We add an action listener to each menu item

   DEMO MenuOptionsDemo

   PROJECT code

   a) 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.

   b) The use of two different kinds of menubars in a single program is not at all good practice. It is only done here for demonstration purposes!