CS112 Lecture: Graphical User Interfaces

Last revised 3/19/08

Objectives:

1. To introduce the notion of a "component" and some basic Swing components (JLabel, JTextField, JTextArea, JButton, JComboBox, JSlider)
2. To introduce the rudiments of using containers, with absolute positioning and the basic layout managers.
3. To introduce use of event-driven programming with GUI widgets (including with multiple event sources)
4. To introduce inner classes
5. To introduce creating and using menus

Materials:

1. Dr. Java to demonstrate individual operations
2. BlueJ project with demo programs from text: ButtonsBallController, MenuBallController, SliderBallController
3. BlueJ project with my demo programs: Component and LayoutDemo, JPanelDemo, GUIEventsDemo, MultipleEvents1/2/3/4, MouseEvents, MenuOptionsDemo
4. Code for AddressBookGUI for project 3

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, which we will continue to develop in future courses. But after this series of lectures you should be able to create and use simple GUI's.

B. The objectdraw package that the book uses provides some graphical facilities. In fact, it is built on the foundation of general Java facilities, while "hiding" some of the details that can be quite confusing, especially to beginners. In this lecture, we begin to explore the foundation on which objectdraw is built.

C. 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 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 - as the book does - 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.*;
   ```

2. 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.)

3. The awt and Swing are quite large - consisting of 90 classes plus 17 interfaces in the awt package and 120 classes plus 24 interfaces in the Swing package in JDK 1.5, plus several subpackages with additional classes. We will only give a first introduction to them now.
4. Many of the visible components that are part of the Swing package have names that begin with capital J - e.g. JButttton, 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 never 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 six we will briefly introduce

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

   The first of these is an output component - i.e. its only use is for displaying information for the user. The remaining five 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 - e.g. a `DrawingCanvas` used by objectdraw is actually a special kind of `JComponent`.

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();
   ```

   (Whenever a new component is added to a GUI, it is necessary to lay it out afresh to take into consideration the additional space needed. The book uses `validate()` for this - which is a method that can be used with any container. In these examples, we will use `pack()`, which can only be used with frames. It does what `validate()` does, plus it resizes the frame to the right size to show everything in it.)

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 learn how the program can respond to events generated when the button is clicked. For now, we'll look at a demo program put together by the authors of the book., that uses three buttons.

   DEMO: ButtonsBallController

   SHOW: Code that contstructs the buttons

H. A `JComboBox` is a component that a user can click to request that some particular action occur. Again, we will use a demonstration program prepared by the book's authors.

DEMO: MenuBallsController

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:

   SHOW: Code that constructs the combo box

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.

I. A `JSlider` is a component that a allows a user to choose a value within a range by manipulating a slider. Again, we will use a demonstration program prepared by the book's authors.

DEMO: SliderBallsController

1. When one constructs a slider, one specifies an orientation (horizontal or vertical, plus minimum, maximum, and initial values.

   SHOW: Code that constructs the slider

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

III. Introduction to Containers

A. A container is a special kind of component whose basic task is to hold and position other components.

1. Top level windows (`JFrames`) and applets (`JApplets`) have a container called the *content pane* that holds their actual contents. (Note that the class `WindowController` used by objectdraw is actually a special kind of `JApplet`.)

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. Container objects have a method called `add`, which can be used to add any component (including another container) to the container. Once a component has been added to a container, it will be displayed inside that container. (It is an error, and can cause weird run time problems, to add the same component to more than one container, or to add it twice to the same container.)

Example: We've done this with various components in our Dr. Java demos

C. Absolute Positioning of Components

1. One of the tasks 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.

2. 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.

3. The approach to layout used in the objectdraw package, and that you used for Project 2, 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) To use absolute positioning, you must do two things:

(1) Set the layout manager for the container to null.

In the case of the constructor for a `JFrame` or `JApplet`:

```
getContentPane().setLayout(null);
```

Otherwise, it is done by

```
whateverContainer.setLayout(null);
```

(2) Invoke the setBounds method on each component *after* adding it to the container:

```
whateverComponent.setBounds(x, y, w, h);
```

Where x and y are the coordinates of the upper left-hand corner of the component relative to its container, w is the width, and h is the height.

(3) Of course, for project 2 this is precisely what you did. (The first operation - setting the layout manager to null - was done by code I supplied for you.) The objectdraw library does this behind the scenes.

D. 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

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

a) DEMO - note originally all in one line, then resize and show effect

b) 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]

c) 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

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

a) A BorderLayout is specified by using one of the following:

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

b) 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

c) 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.

d) 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

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

a) 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.]

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

c) 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

4. 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.

a) 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`]

b) 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

5. The most sophisticated layout manager by far is the GridBagLayout.

a) 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.

b) 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.

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

DEMO, including resizing

6. A CardLayout shows just one component at a time.

a) 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]

b) 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.

c) 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);
```

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

E. 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.

3. An example JPanelDemo.

IV. 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. The objectdraw library defines methods like `onMousePress()` in the WindowController class, which "hide" many of the details of responding to user input in programs. At this point, we are going to shift to working directly with the underlying mechanism, which is much more flexible.

C. 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.

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

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

2. 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.

   c) 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.)

d) 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)

D. 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.

Example: Show each of the above in `GUIDemo`. Note that the program only deals with ActionEvents; the JTextField object handles KeyEvents without the program having to deal with them.

E. 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. *Example:* Show code for `actionPerformed()` in `GUIEventsDemo`

2. *Note:* 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: Run from commandline - `appletviewer GUIEventsDemo.html` with -1 as input

   DEMO with abc as input - show what happened

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

   Example: GUI class for Project 3 does this. SHOW AddressBookGUI.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.

G. 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. (This is the approach developed in the text).

      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 ");").

SHOW: Use of inner classes in `AddressBookGUI` for project 3

H. 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

V. 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, SHOW - MenuOptionsDemo

   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.