Swing GUI

Thanks to Nick Parlante for much of this handout

OOP GUI Systems

OOP Drawing vs Imperative Drawing

- In simple "imperative" drawing, you create a canvas of some sort, and start drawing on it.
- Most OOP drawing systems do not work that way.
- We will have objects that correspond to what's on screen. These objects are sent a "draw yourself" message when they should draw.
- So to get something on screen, we create an object that knows how to draw itself and install it on screen.

Library Class Hierarchy

- There is a large, pre-built inheritance hierarchy of classes for common problems -- drawing, controls, windows, scrolling.... These classes are engineered to work together and share a broad set of working assumptions (i.e. to work with these classes, you will need to understand their design a little).

System: Event -> Notifications (Swing Thread)

- There is a background "system" that manages the basic bookkeeping and orchestration of windows and events. AKA "the system"
- Once the frame is on screen we have "user events" -- clicking, typing, ... events that happen in real time.
- The system manages a queue of user events as they happen (realtime), and translates them to "notification" messages sent, one at a time, to the appropriate GUI objects. The GUI objects draw themselves, react to clicks, etc.
- The System has its own thread, known as the "swing thread" or the "event dispatch thread". Notifications sent to your code -- telling it to draw, telling it that a button has been clicked -- are run on the swing thread.

Programming Tasks...

1. Instantiate Library Classes (easy)

- Many tasks are as simple as constructing and installing system classes -- windows, buttons, labels, etc.
- This is the pretty easy -- requires some reading of the library class docs
- Pull a library object "off the shelf"

2. Subclass Library Classes (harder)

- To introduce custom behavior, subclass off a library class and use overriding to insert your custom code
- This is a trickier programming problem -- you need a deeper understanding of the superclass implementation in order to do the override "in harmony" with its design. In general, the designer of the subclass is responsible for understanding the superclass design, so that the design of the subclass fits in with the superclass.
- At runtime, the code relies on the "pop-down" feature of overriding, so that our little bit of subclass code gets called at the right moment.

- e.g. Subclass off `JButton` so it beeps when clicked
- e.g. Subclass off `JComponent` and override the `paintComponent()` method to insert your own drawing code (but keep the `JComponent` notions of geometry, painting schedule, etc.)

Java GUI History

AWT vs. Swing/JFC

- AWT
  - Abstract Windowing Toolkit
  - AWT provides simple GUI components. Not as rich as Swing.
  - Had some implementation problems early on (1996)
  - AWT drawing uses "native peers" -- creating an AWT button creates a native peer (Unix, Mac, Win32) button to put on screen, and then tries to keep the AWT button and the peer in sync.
  - Advantage: an AWT app has the "native" appearance for buttons etc. since there are in fact native buttons on screen.
  - Disadvantage: the peer design is difficult, keeping the Java object and its peer in sync, and acting the same for all cases on all platforms.
- Swing
  - Also known as JFC
  - Implemented **in Java** -- its the same bytecode running on all platforms -- in that way, Swing really acts the same on all platforms. Builds on only the simplest AWT classes -- frame, etc.
  - Swing has 10x more classes, depth, and functionality than AWT
  - Swing has pluggable look-and-feel feature where buttons, etc can **look** like native ones for that platform. The look and feel is mostly coded in java, and updating it so it looks native is a chore for the Swing maintainers.
- Most recent Java implementations call down to the native OS for rendering the button (e.g. Windows, Mac), so the pixels of the buttons etc. really look right, while doing the logic of the GUI in java.

SWT -- Standard Widget Toolkit

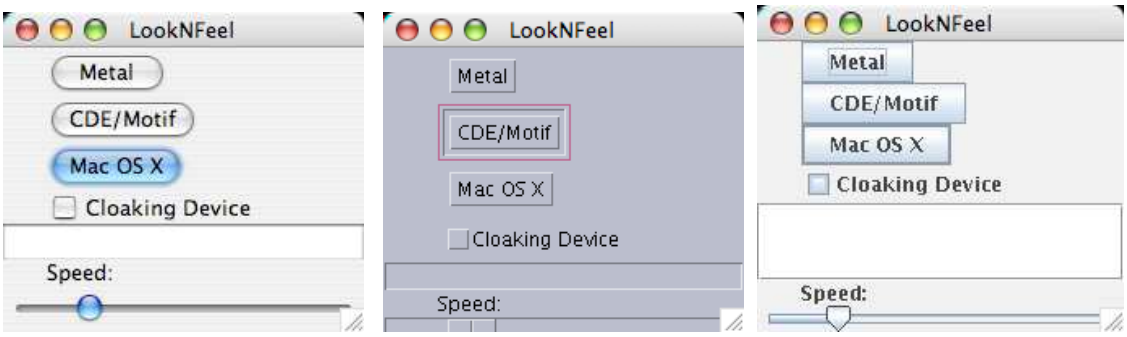
- IBM's Eclipse project uses its own SWT layer, which is similar to the old AWT, but with a newer design. Has the problem of keeping the peers up to date, but has AWT's advantage of looking better, and being quicker.
- SWT provides some competition to Swing, but Swing is by far the dominant GUI system used.

AWT vs. Swing classes

- Some old AWT classes are still used, but mostly we will use the modern Swing versions.
- e.g. AWT Component is the superclass of `JComponent`

Swing -- Flexible Look And Feel

- Swing's "look and feel" feature can adjust the components to take on different looks. Your java code does not need to do anything. The swing components have this capability automatically.
- This feature is needed so that a java program can -- chameleon like -- take on the "native" appearance of the OS where it is running. That's the theory anyway, although users may be sophisticated enough now that apps that look different may be ok.



- To change the look and feel dynamically, see `UIManager.setLookAndFeel()`.
- At the top of `main()`, you can call `setLookAndFeel()` to tell Swing to try to use the "native" look for whatever platform it is running on (example below)
- By default on most platforms, Swing uses a sort of plain, non-platform-specific look and feel called "metal", but typically the native one looks better.

Theme: Things Draw Themselves

- We will have objects that draw themselves -- labels, buttons, etc.
- The system sends components "draw yourself" notifications as needed

Theme: Layout Manager

- A "layout manager" will arrange the size and position of the things on screen.
- We often do not specify the exact x,y where a component should be on screen r its exact size. The LayoutManger does those things.

JComponent

- The Swing superclass of things that draw on screen.
- Defines the basic notions of geometry and drawing -- details below
- Things that appear on screen are generically called "components"

JLabel

- Built in `JComponent` that displays a little text string
- `new JLabel("Hello there");`

JFrame

- A single window
- Has a "content pane" `JComponent` that contains all components in the frame.
- Send `frame.getContentPane()` to get the content pane.
- By default, closing a frame just hides it. See the code below so that closing a frame actually quits the application

Content Pane / Layout Manager

- Use the `add()` message to add components to the content pane.
- Content pane uses a "Layout Manager" to size and position its components
- (Java 5) Frame has a convenience `add()` and `setLayout()` that go to its content pane.

Serializable Warning

- Eclipse gives a warning for a `JFrame` subclass about serialization -- you can ignore this warning. It only applies when serializing out a `JFrame`, which we never do.
- A simple subclass of `JFrame` that puts 3 labels and a button in its content pane.

JComponent

JComponent Basics

- `Drawable`
  - The superclass of all drawable, on screen things
  - Has a **size** and **position** on screen -- defining a "bounds" rectangle
  - Has a **parent** -- the component that contains it
  - Draws itself, within its bounds
  - The word "component" is generally used to refers to any sort of `JComponent`
  - **Warning**: do not manipulate `x, y, width` etc. directly -- these are controlled up in the `JComponent` superclass
- 227 public methods
  - Go glance at the method documentation page for `JComponent` to get a sense of what's there
- Class Hierarchy
  - `JComponent's` position in the inheritance hierarchy:
    `Object` -- (AWT)`Component` -- (AWT)`Container` -- `JComponent`
  - There are few times the AWT classes, intrude, but mostly we'll try to conceptually collapse everything down to `JComponent`.

Layout Managers

Visual Hierarchy

- Components are placed inside other components which form a nesting hierarchy from outer to inner components.
- Frames are the outermost component.
- We might have Frame : content : JPanel : Button1, Button2
- This is called the **visual hierarchy**
- This hierarchy is constructed at runtime, and may change over time.
- Contrast this to the compile time **class hierarchy** between the classes `JComponent`, `JPanel`, `JButton`, ... -- it's easy to get the two hierarchies mixed up

Component Z-Order / Transparency

- There is a back-to-front order of the components
- Each container is "behind" the components it contains
- Where a component is transparent, whatever is behind it shows through.

Layout Manager Theory

- Like HTML -- policy, not exact pixels
- 1. Don't set explicit (pixel) sizes or positions of things
- 2. The layout managers knows the "intent" (policy) of the layout
  - e.g. vertical list
- 3. The layout manager applies the intent to figure the correct size on the fly

- Good: the GUI can work, even though different platforms have fonts with slightly different metrics
- Good: window re-sizing works (the layout manager policy guides how it fits components in to the new window size)
- Good: internationalization (aka "i18n") -- layouts can adjust as the widths required for labels and buttons change for different languages
- Bad: new paradigm, can be unwieldy when you just want to say where things are.
- In Java 6, there is a new GroupLayout to work with GUI editor/generator tools, so you can "draw" your layout. Someday, this should enable good "draw the UI" tools.

Size Do/Don't -- setSize() vs. setMinimumSize() etc.

- Don't call `setSize()` -- the layout manager controls that
- Do call one or more of `setMinimumSize()`, `setMaxiumumSize()`, `setPreferredSize()` -- do this to register a preference **before** the layout manager lays everything out (e.g. before `pack()`/`setVisible()` is called).

Flow Layout

- Arranges components left-right, top-down like text.
  - `panel.setLayout(new FlowLayout())`

Box Layout

- Aligns components in a line -- either vertically or horizontally
- Can set an existing `JPanel` to use a `Box` layout...
  - `panel = new JPanel()`
  - `panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));`
- There are older convenience methods `Box.createVerticalBox()` and `Box.createHorizontalBox()`. In rare cases, the boxes created this way will not erase things correctly, so they are no longer recommended. Creating a `JPanel` and giving it a `BoxLayout` avoids this problem.
- Use `Box.createVerticalStrut(pixels)` to create a little spacer component that be added to the box between components.

Border Layout

- Main content in the center
  - e.g. the spreadsheet cells
  - Window size changes mostly go to the center
- Decorate with 4 things around the outsize -- north, south, east, west
  - e.g. the controls around the spreadsheet cells
- 2nd parameter to `add()` controls where things go
  - `border.add(comp, BorderLayout.CENTER);  // add comp to center`

Nested JPanel

- `JPanel` is a simple component that you can put other components in
- Use to group other components -- put them both in a JP©anel, and put the `JPanel` where you want
- If you want to control the size taken up by a group of elements, put them in a `JPanel` and `setPreferredSize` on the panel
- e.g. group a label with a control
- e.g. set the layout of the panel to vertical box, put lots of buttons in it, put the panel in the `EAST` of a border layout

Standard Frame Initialization

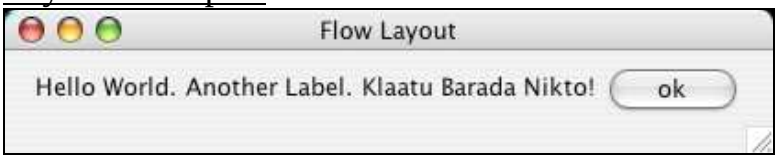
- Typically you create a frame and add components to it
- The `pack()` call tells the layout manager to size and position everything after all the components are added
- By default, the "close" box on a frame just hides it. However, the `setDefaultCloseOperation()` can program various actions when the frame is closed, such as exiting the whole program.
- Last, `setVisible(true)` brings the frame on screen
- Often the last three lines of a frame setup look like...

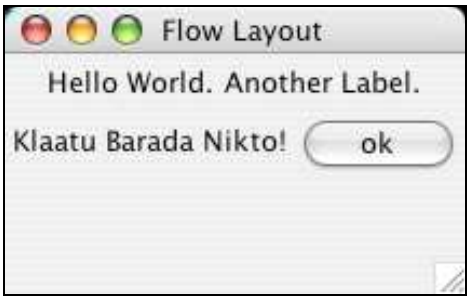
```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.pack();
frame.setVisible(true);
```

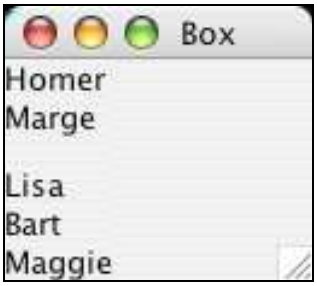
add() and revalidate()

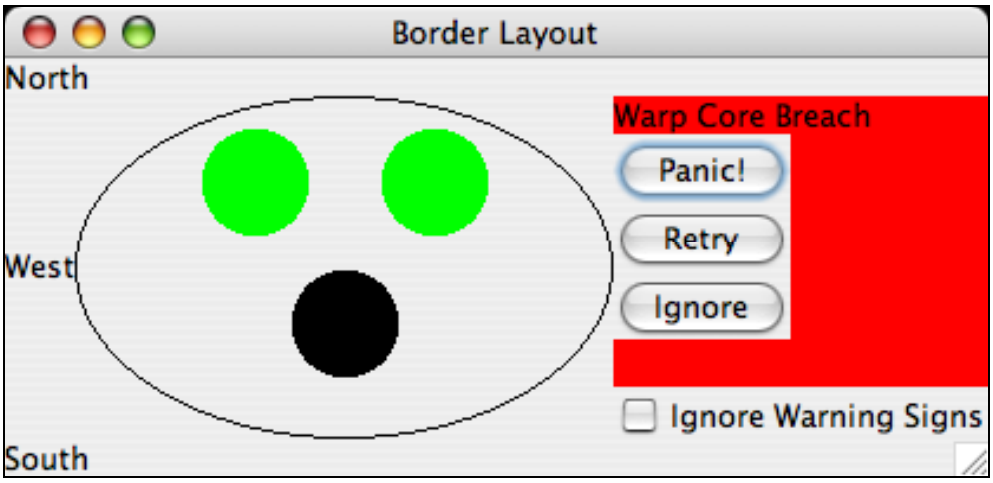
- Normally you `add()` things to the frame, and then a final `pack()`/`setVisible()` lays it out and puts it on screen. If, later on, you `add()` or `remove()` to change the structure of what components are inside other components, call `revalidate()` to trigger the layout manager to lay things out again (`revalidate()` is for Swing components, or use `validate()` for older AWT components). This is not necessary for simple size/width/height changes which work automatically.

Layout Examples










```
// Layouts.java
/*
 Demonstrates some basic layouts.
*/
import java.awt.*;
import javax.swing.*;

public class Layouts {

    public static void main(String[] args) {
        // Here we setup each frame right from main() --
        // alternately, could do setup in the ctor of each frame.

        // GUI Look And Feel
        // Do this incantation at the start of main() to tell Swing
        // to use the GUI LookAndFeel of the native platform. It's ok
        // to ignore the exception.
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (Exception ignored) { }


        // ------
        // 1. Flow Layout
        // Flow layout arranges Left-right top-bottom, like text
        JFrame frame1 = new JFrame("Flow Layout");
        frame1.setLayout(new FlowLayout());

        // Use frame.add() to install components
        frame1.add(new JLabel("Hello World."));
        frame1.add(new JLabel("Another Label."));
        frame1.add(new JLabel("Klaatu Barada Nikto!"));
        frame1.add(new JButton("ok"));

        // Force the frame to size/layout its components
        frame1.pack();
        frame1.setVisible(true);



        // ------
        // 2. Box Layout
        JFrame frame2 = new JFrame("Box Layout");

        // Create a component with vertical Box layout,
        // and install it in the frame
        JComponent content2 = new JPanel();
        content2.setLayout(new BoxLayout(content2, BoxLayout.Y_AXIS));
        frame2.setContentPane(content2);

        // add a few components
        frame2.add(new JLabel("Homer"));
        frame2.add(new JLabel("Marge"));
```

```
        // add a little spacer
        frame2.add(Box.createVerticalStrut(12));

        frame2.add(new JLabel("Lisa"));
        frame2.add(new JLabel("Bart"));
        frame2.add(new JLabel("Maggie"));

        frame2.pack();
        frame2.setVisible(true);

        // ------
        // 3. Border Layout + nested box panel
        JFrame frame3 = new JFrame("Border Layout");

        // Border layout
        frame3.setLayout(new BorderLayout());

        // Add labels around the edge
        frame3.add(new JLabel("North"), BorderLayout.NORTH);
        frame3.add(new JLabel("West"), BorderLayout.WEST);
        frame3.add(new JLabel("South"), BorderLayout.SOUTH);

        // Add a FaceComponent in the center (draws as sort of face in a rect)
        frame3.add(new FaceOld(200, 200), BorderLayout.CENTER);

        // Create a little vertical box JPanel
        // and put it in the EAST with our controls.
        // Make the panel RED. It is front of the window content (light gray)
        // but behind the JButtons
        JPanel panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
        panel.setBackground(Color.RED);
        panel.add(new JLabel("Warp Core Breach"));
        panel.add(new JButton("Panic!"));
        panel.add(new JButton("Retry"));
        panel.add(new JButton("Ignore"));

        panel.add(Box.createVerticalStrut(20));  // 20 pixel spacer

        panel.add(new JCheckBox("Ignore Warning Signs"));

        frame3.add(panel, BorderLayout.EAST);

        frame3.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame3.pack();
        frame3.setVisible(true);
    }
}
```

Swing Controls/Listeners

Listener Pattern

- In many places, Java uses the "listener" pattern to allow classes to notify each other about events.
- The "patterns" community seeks to identify and give names to common coding patterns. In that community, this pattern is called "Observer/Observable". It's a commonly used pattern.
- For controls, we will use listeners to "wire up" the control to an object that wants to know when the control is clicked.

Anonymous Inner Class Recap

- An "anonymous" inner class is a type of inner class created on the fly in the code with a quick-and-dirty syntax.
- Convenient for creating small inner classes -- essentially these will play the role of callback function pointers as we'll see below.

- As a matter of style, the anonymous inner class is appropriate for small sections of code. If the class requires non-trivial ivars or methods, then a true inner class is a better choice.
- When compiled, the inner classes are given names like `Outer$1`, `Outer$2` by the compiler.
- An anonymous inner class cannot have a ctor. It must rely on the default constructor of its superclass.
- An anonymous inner class does not have a name, but it may be stored in a Superclass type pointer. The inner class has access to the outer class ivars, as usual for an inner class.
- The anonymous inner class does not have access to local stack vars from where it is declared, unless they are declared final.

Controls and Listeners

Control Source-Listener Theory

- Source
  - Buttons, controls, etc.
- Listener
  - An object that wants to know when the control is operated
- Notification message
  - A message sent from the source to the listener as a notification that the event has occurred.
  - The listener puts the code they want to run in the notification method

1. Listener Interface

- `ActionListener` interface
- Objects that want to listen to a `JButton` must implement the `ActionListener` interface. `ActionListener` defines the message `actionPerformed()` which is the notification that the button sends when clicked.

```
// ActionListener.java
public interface ActionListener extends EventListener {

    /**
     * Invoked when an action occurs.
     */
    public void actionPerformed(ActionEvent e);

}
```

2. Notification Method

- The notification message is protoyped in the `ActionListener` interface.
- The listener implements this notification method to do whatever they want to do when the control sends the notification. (The notification is sent on the swing thread).
- The `ActionEvent` parameter includes extra information about the event in case the listener cares -- a pointer to the source object (`e.getSource()`), when the event happened, modifier keys held down, etc,

```
public void actionPerformed(ActionEvent e) {
    // code that runs when the control is clicked or whatever
}
```

3. source.addXXX(listener)

- To set up the listener relationship, the listener must register with the source by sending it an add listener message.

- e.g. `button.addActionListener(listener)`
- The listener must implement the `ActionListener` interface
  - i.e. it must respond to the message that the button will send

4. Event -> Notification

- When the action happens (button is clicked, etc.) ...
- The source iterates through its listeners
- Sends each the notification (the notification is done on the swing thread)
- e.g. `JButton` sends the `actionPerformed()` message to each listener

Using a Button and Listener

- There are 3 ways, but technique (3) below is the most common...

1. Component implements ActionListener

- The component or frame could implement the interface (`ActionListener`) directly, and register `this` as the listener object. This is simple and avoids the whole concept of inner classes. This is the way I used to do it in CS106A -- have the Frame listen to all the buttons.

```
class MyComponent extends JComponent implements ActionListener {
    ...
    ...
    public MyComponent() { // in the JComponent ctor
        button.addActionListener(this);
        ...
    }


    ...
    public void actionPerformed(ActionEvent e) {
        // do something
    }
```

2. Create an inner class to be the listener

- Like the `ChunkIterator` strategy.
- Create a `MyListener` inner class that implements `ActionListener`
- Create a new `MyListener` object and add it via button `addXXX(listener)`
- This works fine, but is not the most common technique.

```
// in the JComponent ctor
ActionListener listener = new MyActionListener();
button.addActionListener(listener);
```

3. Anonymous inner class

- Variant of above technique.
- Create an "anonymous inner class" that implements the listener interface
- Like an inner class (option 2 above), but does not have a name
- Can be created on the fly inside a method

```
        button = new JButton("Beep");
```

```
        panel.add(button);
        button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    Toolkit.getDefaultToolkit().beep();
                }
            }
        );
```

Listener Switch Logic

- It is possible to use a single listener object for multiple controls
- In that case, the listener can test `e.getSource()` to see which control was the source of the notification.
- If we have one listener per control, we won't need that logic

Button Listener Example


```
/*
 Demonstrates bringing up a frame with a couple of buttons in it.
 One button uses a named inner class listener, and one
 uses an anonymous inner class listener
*/
public class ListenerFrame extends JFrame {
    private JLabel label;


    /*
     When the Yay button is clicked, we append a "!" to
     the JLabel. This inner class expresses that code.
     */
    private class YayListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {

            String text = label.getText();
            label.setText(text + "!");
            // note that we can access ivars like 'label'
        }
    }

    public ListenerFrame() {
        super("ListenerFrame");
        setLayout(new FlowLayout());

        /*
         Example 1 --
         Create a Yay button and a label in a little panel.
         Connect the button to a YayListener.
        */
        JPanel panel = new JPanel();
        add(panel);  // Add the panel to the frame content

        // Add some things to the panel
        JButton yay = new JButton("Yay!");
        label = new JLabel("Woo Hoo");
        panel.add(yay);
        panel.add(label);
```

```
        ActionListener listener = new YayListener();    // create listener
        yay.addActionListener(listener);                // register it with button



        /*
         Example 2 --
         Create a button that beeps.
         Similar effect to above, but does it all in one step
         using an anonymous inner class.
         */
        JButton beep = new JButton("Beep!");
        add(beep);

        beep.addActionListener(
                // Create anon inner class ActionListener on the fly
                new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        label.setText("Beep!");
                        System.out.println("beep!");
                        Toolkit.getDefaultToolkit().beep();

                        // Can access outer ivars like "label" here.

                        // but not stack vars like "panel" and "beep"
                        // (unless they are final)
                        // beep.setEnabled(false);  // no compile without "final"

                        // What exceptions look like on the Swing thread
                        //String a = null;
                        //a.length();
                    }
                }
        );

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
        new ListenerFrame();
    }
}
```

Misc Listeners

JCheckBox

- Uses `ActionListener`, like `JButton`
- Responds to `boolean isSelected()` to see if it's currently checked

JSlider

- `JSlider` -- component with min/max/current int values
- `JSlider` uses the `StateChangeListener` interface -- the notification is called `stateChanged(ChangeEvent e)`
- Use `e.getSource()` to get a pointer to the source object
- `JSlider` responds to `int getValue()` to get its current value

JTextField

- Implements `ActionListener` like `JButton`
- Triggers when the user hits return
- Then use `getText()` / `setText()`
- Also supports a more complex `DocumentListener` interface that signals individual edits on the text

Listener Strategy

- The technique shown above.
- Get notifications from the button, slider, etc. at the time of the change

Poll Strategy

- Polling technique -- do not listen to the control. Instead, check the control's value at the time of your choosing
- Polling is simpler if you can get away with it.
- e.g. `checkbox.isSelected()`
- Avoids having two copies of the control's state -- just use the one copy in the control itself.
- Polling does not work if you need to do something immediately on control change, since you want to hear of the change right when it happens.