Swing, Part 1 of 5

Christopher Simpkins
chris.simpkins@gatech.edu

Outline

- Hello, Swing!
- Event-driven programming
- Hello, buttons!
- The observer pattern in Swing

Hello, Swing

Here's a minimal Swing program:

```
import javax.swing.JFrame;

public class HelloSwing {

    public static void main(String[] args) {
        JFrame mainFrame = new JFrame("Hello, Swing!");
        mainFrame.setSize(640, 480);
        mainFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        mainFrame.setVisible(true);
    }
}
```

See HelloSwing.java and the API documentation for JFrame

javax.swing.JFrame

Almost all Swing programs use a `JFrame` for the main GUI window.

Here's a high-level recipe for using `JFrame`:

- Instantiate a `JFrame`, passing a title to the constructor

```
JFrame mainFrame = new JFrame("Hello, Swing!");
```

- Set `JFrame`'s initial size (in the next lecture we'll see how to do this in a general way)

```
mainFrame.setSize(640, 480);
```

- Specify what to do when user clicks "x" button on title bar

```
mainFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

- Add components to the `JFrame` and wire them to listeners
- Display the `JFrame`

```
mainFrame.setVisible(true);
```

Today we're using and customizing a `JFrame` object from within another class. Next lecture we'll create custom subclasses of `JFrame`.

Event-Driven Programming

So far we've done structured sequential programming where the order of execution is controlled by the programmer. GUIs use event-driven programming:

- User is presented with options.
- User actions (and other actions) fire events.
- Event handlers execute in response to events.
- Order of execution is controlled by the order of events, which the programmer does not know in advance.

Hello, buttons!

```
public class HelloButtons {
    public static void main(String[] args) {
        JFrame mainFrame = new JFrame("Hello, buttons!");
        mainFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        mainFrame.setLayout(new FlowLayout());

        JButton exitButton = new JButton("Exit");
        exitButton.addActionListener(new ExitListener());

        JLabel counterLabel = new JLabel("Count: 0");
        JButton counterButton = new JButton("Increment count");
        counterButton.addActionListener(
            new CountListener(counterLabel));

        mainFrame.add(exitButton);
        mainFrame.add(counterButton);
        mainFrame.add(counterLabel);
        mainFrame.setSize(300, 275);
        mainFrame.setVisible(true);
    }
}
```

See HelloButtons.java, ExitListener.java and CountListener.java

The Observer Pattern in Swing

Three particpants in the observer pattern:

- An event publisher that fires events
- An event object that represent the event
- Event handlers that subscribe to event publishers and receive event objects

Practically speaking, firing an event means calling a method on event listeners. Let's look at a concrete example.

An Event Publisher: `javax.swing.JButton`

In `HelloButtons.java` we set up an exit button like this:

```
JButton exitButton = new JButton("Exit");
exitButton.addActionListener(new ExitListener());
```

- `JButton`'s `addActionListener` method takes an object that implements the `java.awt.event.ActionListener` interface

java.awt.event.ActionListener

```
public interface ActionListener extends EventListener {

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

}
```

- Event listeners implement the `ActionListener` interface, which includes only one method.

ExitListener

Here's the complete definition for `ExitListener`:

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ExitListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
```

- Any time the exit button is pressed, `ExitListener`'s `actionPerformed` method is called

CountListener

Here's the complete definition for CountListener.java (minus imports):

```
public class CountListener implements ActionListener {
    private JLabel countLabel;
    private int count;
    public CountListener(JLabel countLabel) {
        this.countLabel = countLabel;
        count = 0;
    }
    public void actionPerformed(ActionEvent e) {
        count++;
        countLabel.setText("Count: " + count);
    }
}
```

- `CountListener` keeps a reference to a `JLabel` and an `int` to hold the count
- When its `actionPerformed` method is called, it updates the count and (re-) sets the text on its `JLabel` with the new count

Three objects cooperating: a `JButton`, a `JLabel`, and a `CountListener` (which is-a `ActionListener`) to tie them together.

Closing Thoughts

- Event-driven GUI programming requires a shift in thinking. Putting the user in control means you have to work harder to
  - handle order dependencies, e.g., by disabling buttons until certain actions are taken, and
  - guide the user, e.g., by following UI guidelines to maximize familiarity.
- JavaFX is the future, but learning Swing first makes sense because
  - Swing concepts are present in JavaFX,
  - JavaFX is much harder to set up and debug (so Swing is better for beginners), and
  - our goal here is to teach the basics of event-driven GUI programming.