COMP241
Software Engineering Development
Lecture 14: GUIs & Event Handling 2

Mark Hall

Readings: Horstmann Chap 4

- Inner classes
  - Anonymous inner classes
- Containers as event listeners
- More event handling
  - Handling mouse events
  - Adapter classes
  - Processing text input

DEPARTMENT OF COMPUTER SCIENCE
TARI RUROHIKO

Handling action events for two different buttons

- Solution—*inner classes*
- Classes declared within the scope of an enclosing class
  - Can use **all** the methods and variables of the outer class, **even the private ones**

Inner classes

- Simple inner class:

```
class MyOuterClass {

  class MyInnerClass {
    void go() {
    }
  }
}
```

Inner class is fully enclosed by outer class

- Inner class using outer class variable:

```
class MyOuterClass {

  private int x;

  class MyInnerClass {
    void go() {
      x = 42;
    }
  } // close inner class
} // close outer class
```

Use 'x' as if it were a variable of the inner class!

Inner classes

- An inner class instance must be tied to an outer class instance*
  - *An **inner** object must be tied to a specific **outer** object on the heap*
    - Make an instance of the outer class
    - Make an instance of the inner class by using the instance of the outer class
    - The outer and inner objects are now linked

*Exception: inner class declared as **static**

How to make an instance of an inner class

```
class MyOuter {

  private int x;

  MyInner inner = new MyInner();

  public void doStuff() {
    inner.go();
  }

  class MyInner {
    void go() {
      x = 42;
    }
  } // close inner class
} // close outer class
```

Make an instance of the inner class.

Call a method on the inner class

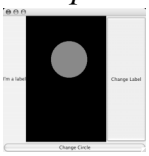
How to make an instance of an inner class

- You can instantiate an inner class from code running outside the outer class
  - Have to use a special syntax

```
class foo {
  public static void main(String [] args) {
    MyOuter outerObj = new MyOuter();
    MyOuter.MyInner innerObj = outerObj.new MyInner();
  }
}
```

Returning to the two button example

```
public class TwoButtons {
  JFrame frame;
  JLabel label;

  public static void main(String [] args) {
    TwoButtons gui = new TwoButtons();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JButton labelButton = new JButton("Change label");
    labelButton.addActionListener(new LabelListener());

    JButton colorButton = new JButton("Change Circle");
    colorButton.addActionListener(new ColorListener());

    label = new JLabel("I'm a label");
    MyDrawPanel drawPanel = new MyDrawPanel();

    frame.getContentPane().add(BorderLayout...
    ... // remainder of gui setup stuff
  }

  ... // continued on next slide
```



Instead of passing *this* to the button's listener registration method, pass a new instance of the appropriate listener class.

Two buttons example

```
// continued from previous slide

class LabelListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    label.setText("Ouch!");
  }
} // close inner class

class colorListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    frame.repaint();
  }
} // close inner class

} // close TwoButtons
```

Now we get to have two ActionListeners in a single class.

The inner class gets to use the 'frame' instance variable without having an explicit reference to the outer class object.

Anonymous Inner Classes

- An entity is *anonymous* if it doesn't have a name
  - In a program, something that is used once usually doesn't need a name
    - Eg. A static label:

      ```
      add(new JLabel("x = "));
      ```
  - If the `Jlabel`'s text does not need to change then we do not need a reference to it—it is an *anonymous* object
- Inner classes often give rise to a similar situation

Anonymous Inner Classes

- The `LabelListener` and `ColorListener` inner classes are only instantiated once in the `TwoButton`'s `go()` method; after that they are never used again
- In Java, it is possible to define an *anonymous inner class* if all you ever need is a single object of that class
- Anonymous inner classes are typically used quite extensively in gui event handling code

```
public class TwoButtons {
  JFrame frame;
  JLabel label;

  public static void main(String [] args) {
    TwoButtons gui = new TwoButtons();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JButton labelButton = new JButton("Change label");
    labelButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        label.setText("Ouch!");
      }
    });

    JButton colorButton = new JButton("Change Circle");
    colorButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        frame.repaint();
      }
    });
    ... // remainder of gui setup stuff
  }
  ... // continued on next slide
```

Now we don't need the LabelListener or ColorListener classes at all!

Common Error: forgetting to attach a Listener

- You run your program and find that your buttons (and other UI components) seem dead
  - Probable cause—you've programmed the listener class and the event handler action but have forgotten to attach it to the event source
- Using anonymous inner classes can help here because you typically declare and instantiate the listener in the statement that adds it to the event source

Productivity Hint: don't use a Container as a Listener

- We've made use of inner classes for event listeners
  - This approach works for many different event types
  - Is simple and intuitive once you master the technique
- It is bad practice to bypass separate event listener classes and turn a Container (such as a `JPanel`) into a Listener (like we did for the simple examples last lecture)
  - E.g. a subclass of `JPanel` implements `ActionListener` for buttons (`actionPerformed` method now becomes part of the `JPanel`)
  - Bad because it separates the button definition from the button action
  - Bad because it doesn't ***scale*** well—for multiple buttons the `actionPerformed` method must investigate the event source
    - Leads to messy, hard to understand code

More event handling: Mouse Example

- Write a program to spy on `MouseEvents` and print them to the console as they occur. Three classes are involved:
  - *The event source*—the component that generates the mouse event and that manages the listeners
    - We will use a **`JPanel`** for this
  - *The listener class*—for mouse clicks we need a class that implements the `MouseListener` interface
    - Contains methods that are called when the mouse button is depressed, released etc.
  - *The event class*—`MouseEvent`
    - Each listener method takes a `MouseEvent` parameter that tells you details about the event (e.g. *x* and *y* position of the mouse pointer)

```
Mouse entered. x = 239 y = 2
Mouse pressed. x = 226 y = 61
Mouse released. x = 226 y = 61
Mouse clicked. x = 226 y = 61
Mouse pressed. x = 196 y = 108
Mouse released. x = 196 y = 108
Mouse clicked. x = 196 y = 108
Mouse exited. x = 303 y = 176
Mouse entered. x = 289 y = 184
Mouse exited. x = 318 y = 188
. . .
```

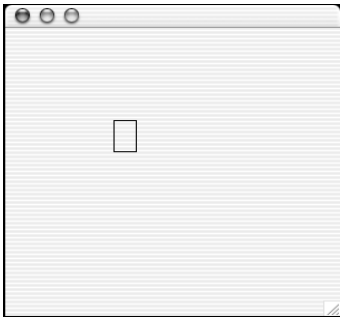
```
import java.awt.event.*; // events/listeners live here

public class MouseSpy implements MouseListener {
  public void mousePressed(MouseEvent e) {
    printEvent(e, "Mouse pressed");
  }
  public void mouseReleased(MouseEvent e) {
    printEvent(e, "Mouse released");
  }
  public void mouseClicked(MouseEvent e) {
    printEvent(e, "Mouse clicked");
  }
  public void mouseEntered(MouseEvent e) {
    printEvent(e, "Mouse entered");
  }
  public void mouseExited(MouseEvent e) {
    printEvent(e, "Mouse exited");
  }
  private void printEvent(MouseEvent e, String type) {
    System.out.println(type + ". x = " + e.getX()
              + " y = " + e.getY());
  }
}
```

MouseListener methods that we must implement

```
// MouseExample application - makes a JPanel, adds a
// MouseSpy listener to the panel and adds the JPanel
// to a JFrame
import javax.swing.JPanel;
import javax.swing.JFrame;

public class MouseExample {
  public static void main(String [] args) {
    JPanel myPanel = new JPanel();
    myPanel.addMouseListener(new MouseSpy());
    JFrame myFrame = new JFrame();
    myFrame.setContentPane(myPanel);
    myFrame.setSize(300, 300);
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    myFrame.setVisible(true);
  }
}
```

Processing Mouse Input

- In real programs you will want to actually do something useful when the user presses the mouse
- Mouse example 2
  - Write a program that moves a rectangle to the mouse press position



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

public class MouseExample2 extends JPanel {
  // Rectangle – x, y, width, height
  private Rectangle mBox = new Rectangle(100, 100, 20, 30);

  public MouseExample2() {

    // inner class for handling mouse press events
    class MousePressListener extends MouseAdapter {
      public void mousePressed(MouseEvent e) {
        // reset the coordinates of mBox
        mBox.setLocation(e.getX(), e.getY());
        repaint(); // ask the JPanel to refresh itself
      }
    } // end MousePressListener

    // Add a MousePressedListener to this JPanel
    addMouseListener(new MousePressListener());
  }

  . . . // Continued on next slide
```

```
  ... // continued from previous slide
  public void paintComponent(Graphics g) {
    // first let the superclass erase the old contents
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    g2.draw(mBox); // now draw our box
  }
}
```

MouseExample2

- `MouseExample2` introduces the use of an adapter class (`MouseAdapter`)
  - We extended **`MouseAdapter`** when implementing our mouse listener
    - `MouseAdapter` is an *abstract* class, so we have to extend it
    - `MouseAdapter` provides **empty** implementations of `MouseListener` methods; we can just override the methods that we're interested in

MouseExample2

- `MouseExample2` extends the functionality of `JPanel`
  - We overide the **`paintComponent`** method in `JPanel` in order to render the `Rectangle`
  - Notice that we also call `JPanel`'s `paintComponent` method so that the panel gets cleared between updates
- After processing a mouse pressed event we call `repaint()`
  - This tells the component to repaint itself at the next convenient moment
  - There is a `paint()` method but we should never call this directly—the window manager will call this for us with an appropriate `Graphics` object

Processing Text Input

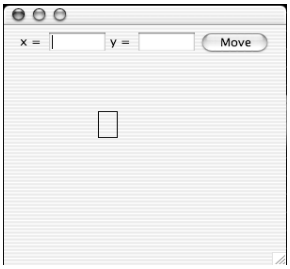
- Most graphical programs collect text input through *text fields*
- The Java Swing GUI libraries have a `JTextField` class for text input
  - When you construct a text field, you supply the width (approx number of characters)

    ```
    JTextField mXField = new JTextField(5);
    ```
  - You can type additional characters, but then part of the content of the field becomes invisible
- You will want to *label* each text field
  - Use a `JLabel`:

    ```
    JLabel mXLabel = new JLabel("x = " );
    ```
- Finally, you want to give the user an opportunity to enter information in all text fields before processing it
  - Need a button that the user can press to indicate that the input is ready for processing

Processing Text Input

- `TextInputExample`
  - Similar to `MouseExample2`, but has text fields to allow the user to set the *x* and *y* coordinates of the box




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

public class TextInputExample extends JPanel {
  // x, y, width, height
  private Rectangle mBox =
    new Rectangle(100, 100, 20, 30);

  // text fields for the x and y coordinates
  private JTextField mXField = new JTextField(5);
  private JTextField mYField = new JTextField(5);

  // a button to allow the user to update the
  // rectangle location
  private JButton mMoveButton =
    new JButton("Move");

  . . . // continued on next slide
```

```
  . . . // continued from previous slide

  public TextInputExample() {
    super();

    // install an ActionListener to move the rectangle
    mMoveButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // reset the coordinates of mBox
        int x = Integer.parseInt(mXField.getText());
        int y = Integer.parseInt(mYField.getText());
        mBox.setLocation(x, y);
        repaint(); // ask the JPanel to refresh itself
      }
    });

    // set up the panel
    add(new JLabel("x = ")); add(mXField);
    add(new JLabel("y = ")); add(mYField);
    add(mMoveButton);
  }

  ... // paintComponent and main method ommitted
```