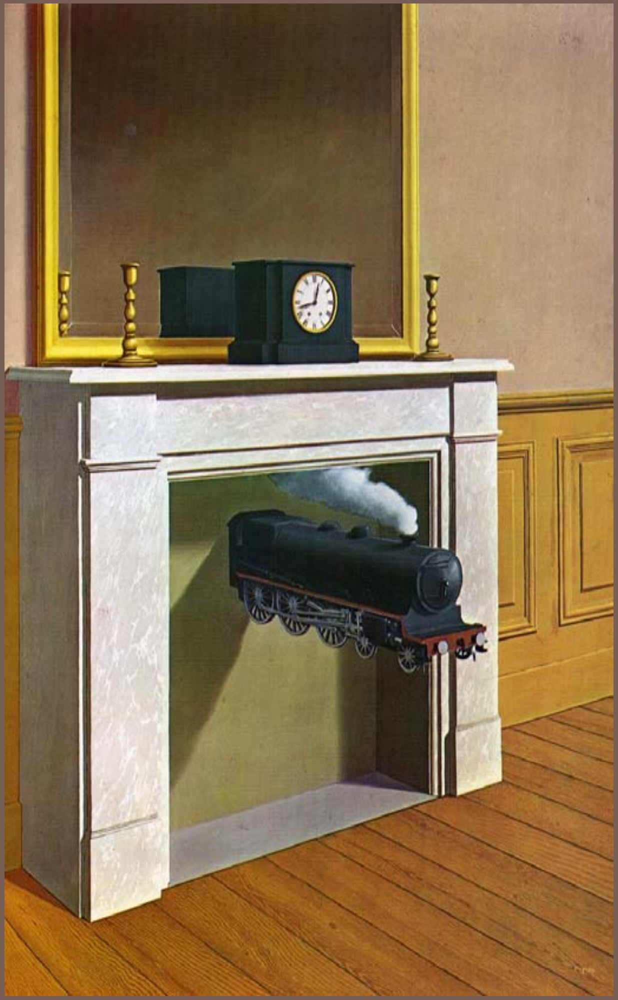


GUI DYNAMICS

Lecture 11

CS2110 – Fall 2009

GUI Statics and GUI Dynamics

- Statics: what's drawn on the screen
  - Components
    - buttons, labels, lists, sliders, menus, ...
  - Containers: components that contain other components
    - frames, panels, dialog boxes, ...
  - Layout managers: control placement and sizing of components

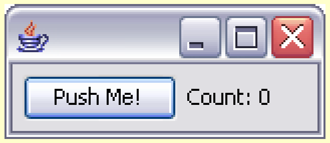
- Dynamics: user interactions
  - Events
  - button-press, mouse-click, key-press, ...
  - Listeners: an object that responds to an event
  - Helper classes
  - Graphics, Color, Font, FontMetrics, Dimension, ...

Dynamics Overview

- Dynamics = causing and responding to actions
  - What actions?
    - Called *events*: mouse clicks, mouse motion, dragging, keystrokes
    - We would like to write code (a *handler*) that is invoked when an event occurs so that the program can respond appropriately
    - In Java, you can intercept events by providing an *object* that "hears" the event – a *listener*
- What objects do we need to know about?
  - *Events*
  - *Event listeners*

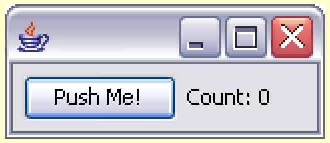
Brief Example Revisited

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

public class Intro extends JFrame {

   private int count = 0;
   private JButton myButton = new JButton("Push Me!");
   private JLabel label = new JLabel("Count: " + count);

   public Intro() {
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setLayout(new FlowLayout(FlowLayout.LEFT)); //set layout manager
      add(myButton); //add components
      add(label);
      label.setPreferredSize(new Dimension(60, 10));

      myButton.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            count++;
            label.setText("Count: " + count);
         }
      });

      pack();
      setVisible(true);
   }

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



Brief Example Revisited

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

public class Intro extends JFrame {

   private int count = 0;
   private JButton myButton = new JButton("Push Me!");
   private JLabel label = new JLabel("Count: " + count);

   public Intro() {
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setLayout(new FlowLayout(FlowLayout.LEFT)); //set layout manager
      add(myButton); //add components
      add(label);
      label.setPreferredSize(new Dimension(60, 10));

      myButton.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            count++;
            label.setText("Count: " + count);
         }
      });

      pack();
      setVisible(true);
   }

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



The Java Event Model

- Timeline
  - User or program does something to a component
    - clicks on a button, resizes a window, …
  - Java issues an *event object* describing the event
  - A special type of object (a listener) “hears” the event
    - The listener has a method that “handles” the event
    - The handler does whatever the programmer programmed

- What you need to understand
  - *Events*: How components issue events
  - *Listeners*: How to make an object that listens for events
  - *Handlers*: How to write a method that responds to an event

Events: How your application learns when something interesting happens

- Basic idea: You register a listener and Java calls it
- The argument is an "event": a normal Java object
  - Events are normally created by the Java runtime system
  - You can create your own, but this is unusual
  - Normally events are associated with a component
  - Most events are in java.awt.event and javax.swing.event
  - All events are subclasses of AWTEvent

- ActionEvent
- AdjustmentEvent
- ComponentEvent
- ContainerEvent
- FocusEvent
- HierarchyEvent
- InputEvent
- InputMethodEvent
- InvocationEvent
- ItemEvent
- KeyEvent
- MouseEvent
- MouseWheelEvent
- PaintEvent
- TextEvent
- WindowEvent

Types of Events

- Each Swing Component can generate one or more types of events
  - The type of event depends on the component
    - Clicking a **`JButton`** creates an **`ActionEvent`**
    - Clicking a **`JCheckbox`** creates an **`ItemEvent`**
  - The different kinds of events include different information about what has occurred
    - All events have method **`getSource()`** which returns the object (e.g., the button or checkbox) on which the Event initially occurred
    - An **`ItemEvent`** has a method **`getStateChange()`** that returns an integer indicating whether the item (e.g., the checkbox) was *selected* or *deselected*

Event Listeners

- **`ActionListener`**, **`MouseListener`**, **`WindowListener`**, …

- Listeners are Java interfaces
  - Any class that implements that interface can be used as a listener

- To be a listener, a class must implement the interface
  - Example: an **`ActionListener`** must contain a method **`public void actionPerformed(ActionEvent e)`**

Implementing Listeners

- Which class should be a listener?
  - Java has no restrictions on this, so *any* class that implements the listener will work

- Typical choices
  - Top-level container that contains whole GUI

    ```
    public class GUI implements ActionListener
    ```
  - Inner classes to create specific listeners for reuse

    ```
    private class LabelMaker implements ActionListener
    ```
  - Anonymous classes created on the spot

    ```
    b.addActionListener(new ActionListener() {...});
    ```

Listeners and Listener Methods

- When you implement an interface, you must implement all the interface's methods
  - Interface **`ActionListener`** has one method:
    - **`void actionPerformed(ActionEvent e)`**
  - Interface **`MouseListener`** has five methods:
    - **`void mouseClicked(MouseEvent e)`**
    - **`void mouseEntered(MouseEvent e)`**
    - **`void mouseExited(MouseEvent e)`**
    - **`void mousePressed(MouseEvent e)`**
    - **`void mouseReleased(MouseEvent e)`**
  - Interface **`MouseMotionListener`** has two methods:
    - **`void mouseDragged(MouseEvent e)`**
    - **`void mouseMoved(MouseEvent e)`**

Registering Listeners

- How does a component know which listener to use?
- You must *register* the listeners
  - This connects listener objects with their source objects
  - Syntax: `component.addTypeListener(Listener)`
  - You can register as many listeners as you like

- Example:

```
b.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      count++;
      label.setText(generateLabel());
   }
});
```

Example 1: The Frame is the Listener

```
import javax.swing.*; import java.awt.*; import java.awt.event.*;
public class ListenerExample1 extends JFrame implements ActionListener {
    private int count;
    private JButton b = new JButton("Push Me!");
    private JLabel label = new JLabel("Count: " + count);
    public static void main(String[] args) {
        JFrame f = new ListenerExample1();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(200,100);
        f.setVisible(true);
    }
    public ListenerExample1() {
        setLayout(new FlowLayout(FlowLayout.LEFT));
        add(b); add(label);
        b.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e) {
        count++;
        label.setText("Count: " + count);
    }
}
```

Example 2: The Listener is an Inner Class

```
import javax.swing.*; import java.awt.*; import java.awt.event.*;
public class ListenerExample2 extends JFrame {
    private int count;
    private JButton b = new JButton("Push Me!");
    private JLabel label = new JLabel("Count: " + count);
    class Helper implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            count++;
            label.setText("Count: " + count);
        }
    }
    public static void main(String[] args) {
        JFrame f = new ListenerExample2();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(200,100); f.setVisible(true);
    }
    public ListenerExample2() {
        setLayout(new FlowLayout(FlowLayout.LEFT));
        add(b); add(label); b.addActionListener(new Helper());
    }
}
```

Example 3: The Listener is an Anonymous Class

```
import javax.swing.*; import java.awt.*; import java.awt.event.*;
public class ListenerExample3 extends JFrame {
    private int count;
    private JButton b = new JButton("Push Me!");
    private JLabel label = new JLabel("Count: " + count);
    public static void main (String[] args) {
        JFrame f = new ListenerExample3();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(200,100); f.setVisible(true);
    }
    public ListenerExample3() {
        setLayout(new FlowLayout(FlowLayout.LEFT));
        add(b); add(label);
        b.addActionListener(new ActionListener() {
            public void actionPerformed (ActionEvent e) {
                count++;
                label.setText("Count: " + count);
            }
        });
    }
}
```

Adapters

- Some listeners (e.g., **`MouseListener`**) have lots of methods; you don't always need all of them
  - For instance, you may be interested only in mouse clicks
- For this situation, Java provides *adapters*
  - An *adapter* is a predefined class that implements all the methods of the corresponding Listener
    - Example: **`MouseAdapter`** is a class that implements all the methods of interfaces **`MouseListener`** and **`MouseMotionListener`**
  - The adapter methods *do nothing*
  - To easily create your own listener, you *extend* the adapter class, *overriding* just the methods that you actually need

Using Adapters

```
import javax.swing.*; import javax.swing.event.*;
import java.awt.*; import java.awt.event.*;
public class AdapterExample extends JFrame {
    private int count; private JButton b = new JButton("Mouse Me!");
    private JLabel label = new JLabel("Count: " + count);
    class Helper extends MouseAdapter {
        public void mouseEntered(MouseEvent e) {
            count++;
            label.setText("Count: " + count);
        }
    }
    public static void main(String[] args) {
        JFrame f = new AdapterExample();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(200,100); f.setVisible(true);
    }
    public AdapterExample() {
        setLayout(new FlowLayout(FlowLayout.LEFT));
        add(b); add(label); b.addMouseListener(new Helper());
    }
}
```

Notes on Events and Listeners

- A single component can have many listeners

- Multiple components can share the same listener
  - Can use **`event.getSource()`** to identify the component that generated the event

- For more information on designing listeners, see **`http://java.sun.com/docs/books/tutorial/uiswing/events/`**

- For more information on designing GUIs, see **`http://java.sun.com/docs/books/tutorial/uiswing/`**

Aside: On *Anonymous Classes*

- An amazingly powerful idea
  - In effect, you can create an object, or a static class in one "context" where it can see the variables and methods of its creating class
  - Then pass it to some other context entirely and invoke it, perhaps much later. *It can still access the variables and methods it was able to see when it was created even if the context that created it is no longer active!*
  - Sometimes called a *closure* in the programming languages community

Why are anonymous classes valuable?

- Precisely because they "remember" the context in which they were created
  - Value variables are copied
  - Reference variables: the reference is retained
- Let's see why this benefits us by revisiting an example we used on Tuesday

FlowLayout Example from Tuesday

```java
class S1GUI {
   public class ListenerExample1 extends JFrame {
    private int count;
    private JButton b = new JButton("Push Me!");
    private JLabel label = new JLabel("Count: " + count);

   public S1GUI() {
      JFrame f = new ListenerExample1();
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      f.setSize(500, 200);
      f.setLayout(new FlowLayout(FlowLayout.LEFT));
      for (int b = 1; b < 9; b++)
         f.add(new JButton("Button " + b));
      f.setVisible(true);
   }
}
```

… now with a ButtonClick handler

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

public class Statics1 {
   public static void main(String[] args) {
      new S1GUI();
   }
}

class S1GUI {
    public class ListenerExample1 extends JFrame implements ActionListener {
       public int count;
       public JButton b = new JButton("Push Me!");
        public JLabel label = new JLabel("Count: " + count);
    }

   public S1GUI() {
      JFrame f = new ListenerExample1();
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CL[illegible]
      f.setSize(500, 200);
      f.setLayout(new FlowLayout(FlowLayou[illegible]));
      for (int b = 1; b < 9; b++)
      {
         Jbutton myButton = new [illegible]utton("Button " + b);
            myButton.addAct[illegible]onListener(new ActionListener() {
            public v[illegible]id actionPerformed (ActionEvent e) [illegible]
                f.count++;
                f.label.setText("[" + b + "]: Count " + count);
            }
          });
         f.add(myButton);
      }
      f.setVisible(true);
   }
}
```

The anonymous inner method can access the fields of the Jframe….

… and even the variables that were active when the class was instantiated!

Whoa! What was "b" doing?

- Inside the inner method, b is acting like a parameter
  - In fact Java makes a copy of b, which is why it retains the value it had when the anonymous class was created via new (otherwise everyone would think b = 9!)
  - Java also makes copies of pointers to objects referenced in the method such as "this" and "label", which is why it can access "count" (which "means" this.count) and why it can call label.setlabel().
- Once you get the idea it all makes a lot of sense
  - And this code is very easy to read, too…

… But you can also take these things one step too far

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

public class Statics1 {
   public static void main(String[] args) {
      new S1GUI();
   }
}

class S1GUI {
    public class ListenerExample1 extends JFrame implements A
       public int count;
       public JButton b = new JButton("Push Me!");
       public JLabel label = new JLabel("Count: " + co
   }

   public S1GUI() {
      JFrame f = new ListenerExample1();
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      f.setSize(500, 200);
      f.setLayout(new FlowLayout(FlowLayout.LEFT));
      for (int b = 1; b < 9; b++)
      {
         f.add((new JButton(“Button " + b)).addActionListener(new ActionListener() {
            public void actionPerformed (ActionEvent e) {
                f.count++;
                f.label.setText("[" + b + "]: Count " + count);
            }
          });
      }
      f.setVisible(true);
   }
}
```

Debatable whether this code is at all comprehensible but it certainly is compact!

GUI Drawing and Painting

- For a drawing area, extend **`JPanel`** and override the method **`public void paintComponent(Graphics g)`**
- **`paintComponent`** contains the code to completely draw *everything* in your drawing panel
- Do not call **`paintComponent`** directly – instead, request that the system redraw the panel at the next convenient opportunity by calling **`myPanel.repaint()`**
- **`repaint()`** requests a call **`paintComponent()`** "soon"
  - **`repaint(ms)`** requests a call within ms milliseconds
    - Avoids unnecessary repainting
    - 16ms is a reasonable value

Java Graphics

- The **`Graphics`** class has methods for colors, fonts, and various shapes and lines
  - `setColor(Color c)`
  - `drawOval(int x, int y, int width, int height)`
  - `fillOval(int x, int y, int width, int height)`
  - `drawLine(int x1, int y1, int x2, int y2)`
  - `drawString(String str, int x, int y)`
- Take a look at
  - **`java.awt.Graphics`** (for basic graphics)
  - **`java.awt.Graphics2D`** (for more sophisticated control)
  - The 2D Graphics Trail: `http://java.sun.com/docs/books/tutorial/2d/`
  - examples on the web site