Lecture 14

GUIs, Part IV

14

- Homework 4 due Wednesday
- Quiz #3 next Monday
- Midterm Monday, November 7

UNM LOS ALAMOS

CS251L Intermediate Programming

Software Interaction

- In the early days of programming, you submitted a program (and some input data) to a mainframe and your program was scheduled to run.
- When the mainframe had time to run your program, it would return to you some output data. If your program didn't work correctly, you did the whole process again.
- Later, "interactive computing" meant that you had a session on a terminal connected to some large computer. This allowed you to interact with the computer in real time.

Software Interaction

- The program running on the server was still in control of the human-machine interaction. The computer would ask you for some input, and you would respond. Most programs were only capable of taking one type of input at a time.
  - For example: a program would ask you to enter your name, and wouldn't do anything else until you complied.
- When GUIs were invented, they allowed the user to decide what order to enter data and perform operations. Instead of playing a game of "20 Questions" with the user, they displayed a variety options to the user, and allow the user to drive the interaction with a program.

Software Interaction

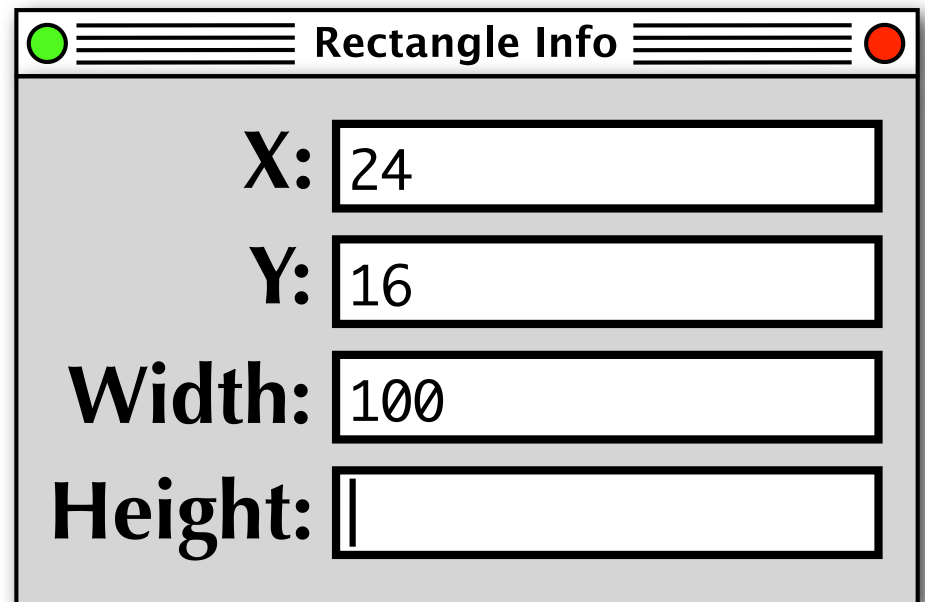
- When a program is only ready to take one input value at a time and imposes a sequence of operations on the users, it is called modal. Modality occurs when only a small fraction of the program's functions are available to the user at a given time.
- When a program lets the user control the order of a process, it is called modeless. Some modality still occurs in modern programs (ex: "wizards").

Modality

Modal:

```
Enter X Value:
>24
Enter Y Value:
>16
Enter Width:
>100
Enter Height:
>|
```

Non-modal:



Events

- GUIs allow a user to interact with any component on the screen at any time, so components have to be ready to respond to the user at any time.
- The primary devices the user has to communicate with a program are the keyboard and the mouse.
- When the user types a key, releases a key, moves the mouse, clicks the mouse button, etc., information about the input event is sent to your program.

Events

- This information includes:
  - When the event happened (used for double-click detection and key repeat processing)
  - Where the event happened on the screen (for mouse event, to help determine what was clicked on)
  - What state the modifier keys were in (Shift, Control, etc.; up or down) when the event happened

Events

- In the early days of GUI programming, when your program got a mouse or keyboard event, it executed a long series of if-then or switch statements to determine which window the event should go to, which component, what response, etc.
- This “event dispatcher” often spanned many pages of code and had to be modified in order to add or change any components on the screen.
- Object-oriented frameworks have done away with the need to write this code.

Events

- AWT and Swing do much of the work to figure out where events go.
  - AWT uses the container hierarchy information and/or keyboard focus information to decide where to send events.
  - Standard Swing components do some processing of events and generate higher level events in response.
- Keyboard and mouse events are the lowest level events in the GUI framework.
- After a JFC component receives a low-level event, it often emits a higher level event given more abstract information about the significance of the low-level event.

Events

- There are higher level events for: button clicked, combo box changed, item in list selected, etc.
- You can decide to process the low level events if you need fine-grain control over how user input is handled, or you can process the high level events and let the framework do most of the work.
- When you write your own components (like our component that displays a Drawing in its content space), you have to handle the low level events yourself.

Events and Listeners

- A running application generates a large quantity of events (as the mouse is moving around, passing over components; as keys or mouse buttons are pressed; as windows are moving or being resized; etc.)
- Also, a typical application contains hundreds or thousands or more components.

Events and Listeners

- If every component were receiving and processing every event, there would be (thousands of components × thousands of events) millions of event processing method calls per second. The application would keep the CPU at 100% utilization every time the mouse was touched. Most of this processing would be unproductive (each component in turn analyzing the event and deciding not to do anything).

Events and Listeners

- To reduce this waste of compute resources, Java uses a publish-and-subscribe model for event distribution. Each component "subscribes" to the event category it need to process, and generated events are sent only to subscribers.
- There are three objects involved in this publish-subscribe "transaction":
  - The sender of the event (usually the Java runtime, or a Swing component)
  - The event object (contains details about the event)
  - The listener to the event (some class that wants to respond when an event occurs).

Events and Listeners

- After a listener subscribes to an event "source," it starts receiving event objects whenever something happens. It can later unsubscribe in order to stop receiving events.
- For your listener to receive events, it implements an interface that is defined for that event type. This interface defines methods that the event sender will call whenever the event occurs. Each listener method accepts a single argument which is the event object.
- For some listeners, there are several methods in the listener interface and the sender will pick one depending on the kind of action that triggered the event.

Mouse Listener

```
package java.awt.event;

import java.awt.event.MouseEvent;

public interface MouseListener
{
  /** Called after mouseReleased() when mouse pressed and
      released at same spot */
  public void mouseClicked(MouseEvent e);

  /** Called when mouse button pressed */
  public void mousePressed(MouseEvent e);

  /** Called when mouse button released */
  public void mouseReleased(MouseEvent e);

  /** Called when mouse moved into a component's region */
  public void mouseEntered(MouseEvent e);

  /** Called when mouse moves out of a component's region */
  public void mouseExited(MouseEvent e);
}
```

New package

This interface is part of AWT. If you implement this interface in your class, your class can then subscribe to mouse events for some component(s).

JFrame

Accessors on `java.awt.event.MouseEvent`:

Method	Description
`getSource()`	Get the object that sent the event (usually a component).
`getX()`	Get the relative mouse X-coordinate (relative to the component the event came from).
`getY()`	Get the relative mouse Y-coordinate
`getButton()`	Get which mouse button was affected: `MouseEvent.BUTTON1`, `MouseEvent.BUTTON2`, or `MouseEvent.BUTTON3`.
`getClickCount()`	Get the number of times the mouse button was clicked.
`getWhen()`	Get the timestamp when the click occurred, in milliseconds

Using Mouse Listener

```
public class SampleMouseListener implements MouseListener
{
  private JComponent componentToListenTo;

  public SampleMouseListener(JComponent componentToListenTo)
  {
    this.componentToListenTo = componentToListenTo;
    // Subscribe to mouse events, must pass a MouseListener argument
    this.componentToListenTo.addMouseListener(this);
  }

  public void stopListening()
  {
    this.componentToListenTo.removeMouseListener(this);
  }

  public void mouseClicked(MouseEvent e) { } // Ignoring for now

  public void mousePressed(MouseEvent e)
  {
    System.out.println("The mouse was just pressed on the component");
  }

  public void mouseReleased(MouseEvent e)
  {
    System.out.println("The mouse was just released");
  }

  public void mouseEntered(MouseEvent e) { } // Ignoring for now

  public void mouseExited(MouseEvent e) { } // Ignoring for now
}
```

Mouse Motion Listener

```
package java.awt.event;

import java.awt.event.MouseEvent;

public interface MouseMotionListener
{
  /** Called when the mouse moves within a component's region */
  public void mouseMoved(MouseEvent e);

  /** Called when the mouse moves in a component's region while
      a button is down */
  public void mouseDragged(MouseEvent e);
}
```

This interface contains additional mouse methods that are less-commonly needed. It is broken out so you don’t have to implement all 7 methods in order to handle simple mouse activities.

Using Mouse Motion Listener

```
public class SampleMouseListener implements MouseListener, MouseMotionListener
{
  private JComponent componentToListenTo;

  public SampleMouseListener(JComponent componentToListenTo)
  {
    this.componentToListenTo = componentToListenTo;
    // Subscribe to mouse events, must pass a MouseListener argument
    this.componentToListenTo.addMouseListener(this);
    this.componentToListenTo.addMouseMotionListener(this);
  }

  public void stopListening()
  {
    this.componentToListenTo.removeMouseListener(this);
    this.componentToListenTo.removeMouseMotionListener(this);
  }

  /** Called when the mouse moves within a component's region */
  public void mouseMoved(MouseEvent e)
  {
    System.out.println("The mouse was moved to " + e.getX() + "," + e.getY());
  }

  /** Called when the mouse moves within a component's region while a button is down */
  public void mouseDragged(MouseEvent e)
  {
    System.out.println("The mouse was dragged to " + e.getX() + "," + e.getY());
  }

  // .. Other methods are still here, but not shown
}
```

Listener Summary

- Every `JComponent` has methods `addMouseListener(MouseListener)`, and `removeMouseListener(MouseListener)`. Any class that implements `MouseListener` can be an argument to these calls.
- Every `JComponent` has methods `addMouseMotionListener(MouseMotionListener)`, `and removeMouseListener(MouseMotionListener)`. Any class that implements `MouseMotionListener` can be an argument to these calls.

Listener Summary

- Any class can be a listener by implementing the interface and declaring the proper methods—either the component itself (it can listen for its own events) or another class.
- If you define your own component, good modularity usually means handling events in the component class so that all the required behavior to make the component work is in one module.

In-class lab exercise #6a

Studying Mouse Events

Add a MouseListener and a MouseMotionListener to your DrawFrame.DrawComponent (from lab 3 or 4) and observe how it works.

6a Studying Mouse Events

Add a simple mouse and mouse motion listener to your **DrawFrame**:

1. Use the sample code in **Mouse.include** to save time.
2. Add the 5 **MouseListener** methods and the 2 **MouseMotionListener** methods to your **DrawFrame.DrawComponent** class. Each of these is predefined to just print the **MouseEvent** to the console for examination.
3. Add **MouseListener** and **MouseMotionListener** to the implements clause of the **DrawComponent**.
4. In the **DrawComponent** constructor, subscribe it to both types of events (**addMouseListener(this)**, **addMouseMotionListener(this)**).
5. Run the program and look for the following things:
   - 5.1. The mouse event type is printed (**MOUSE_DOWN**, **MOUSE_MOVED**, etc.)
   - 5.2. The relative coordinates, which are **(0,0)** at the top left of the component.
   - 5.3. Try clicking buttons without moving the mouse. Notice when **MOUSE_CLICKED** is and is not received.
   - 5.4. Try holding down modifier keys on the keyboard (Shift, Control, etc.)

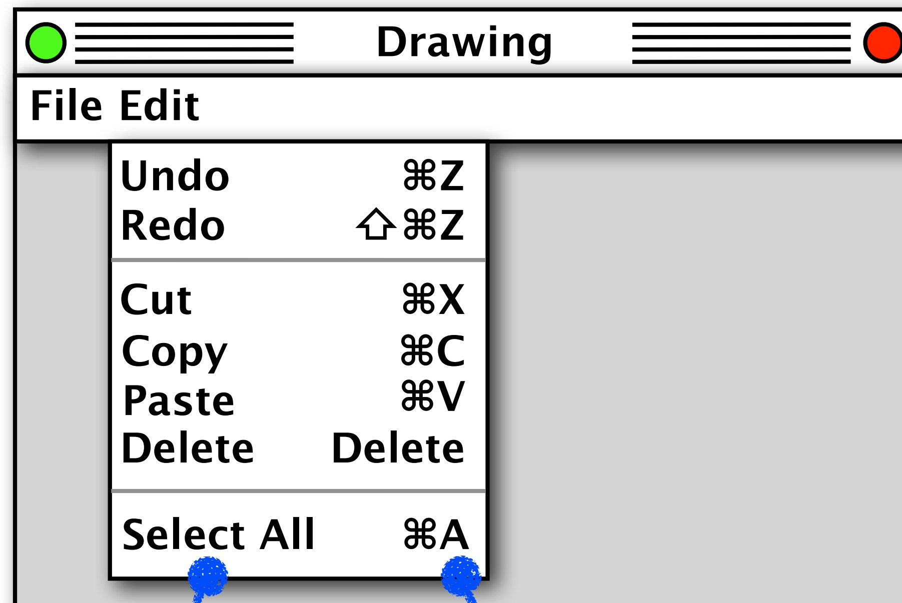
Menus

- To add menus to your applications requires creating three types of components: menu items, menus, and a menu bar.
- The menu bar is in **`javax.swing.JMenuBar`**.
  - The constructor for this class takes no arguments.
  - You add menus later.
  - You add the menu bar to a window by calling **`JFrame.setJMenuBar(JMenuBar)`**.
  - You can do this before or after building the menus.

Menus

- You create a menu by creating an instance of **`javax.swing.JMenu`**.
  - The constructor takes one argument, which is the name of the menu as it appears on the screen.
  - You add the menu to the menu bar by calling **`JMenuBar.add(JMenu)`**.
- To add commands to your menus, you create an instance of **`javax.swing.JMenuItem`**.
  - The constructor takes an argument, which is the name of the menu item as it appears onscreen.
  - You add the menu item to the menu by calling **`JMenu.add(JMenuItem)`**.
  - You can add a separator to a menu (to group related options) by calling **`JMenu.addSeparator()`**.

Menus

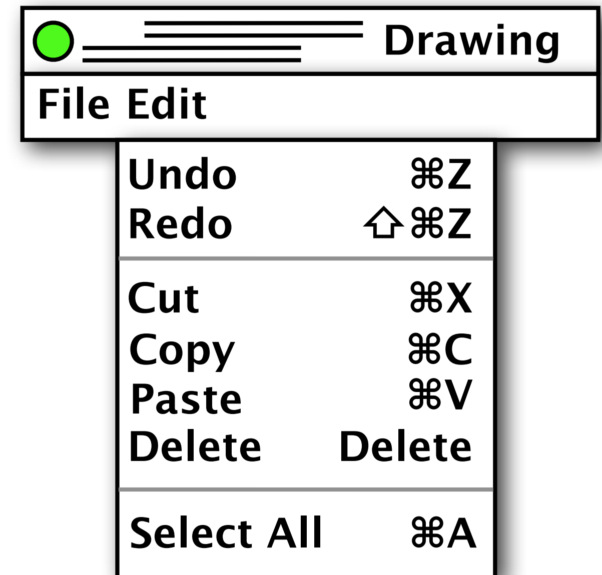


Menu Item

Keyboard Accelerator
(with modifier keys)

Creating Menus

```
JMenuBar menuBar = new JMenuBar();
myFrame.setJMenuBar(menuBar);

// ...

JMenu editMenu = new JMenu("Edit");
menuBar.add(editMenu);

this.undoMenuItem = new JMenuItem("Undo");
editMenu.add(this.undoMenuItem);

this.redoMenuItem = new JMenuItem("Redo");
editMenu.add(this.redoMenuItem);

editMenu.addSeparator();

// ...
```



Action Events

- Most components handle the `MouseEvents` internally and generate higher level events that describe what happened "semantically."
  - This saves you the trouble of having to implement the "feel" (of "look and feel") of each component.
  - This ensures that the behavior of the components will be standard across different applications.
- Certain components that are supposed to send "commands" to the application when clicked send `ActionEvents`. Components that do this include `JButtons` and `JMenuItems`.
- If you want to know when a button is clicked, you attach an `ActionListener` to it.
  - `ActionListener` only defines one method.

Using Action Listener

```
public class MyFrame extends JFrame implements ActionListener
{
  private JButton button1;
  private JButton button2;

  public MyFrame()
  {
    super("My Frame with 2 Buttons");
    // ... Layout code
    this.button1.addActionListener(this);
    this.button2.addActionListener(this);
  }

  public void actionPerformed(ActionEvent event)
  {
    System.out.println("One of my buttons was clicked");
  }
}
```

Clicking either button will cause **`actionPerformed()`** to be called. How do we tell which button was clicked?

Using Action Listener

```
public class MyFrame extends JFrame implements ActionListener
{
  private JButton button1;
  private JButton button2;

  public MyFrame()
  {
    super("My Frame with 2 Buttons");
    // ... Layout code
    this.button1.addActionListener(this);
    this.button2.addActionListener(this);
  }

  public void actionPerformed(ActionEvent event)
  {
    Object source = event.getSource();
    if (source == this.button1)
      doThing1();
    else if (source == this.button2)
      doThing2();
  }
}
```

It's usually more convenient to put the actual implementation in separate methods so you can call them programmatically.

Window Events

- You might have noticed when you create your own JFrame that the JVM does not exit when you close the window.
- Most applications have multiple windows, and don't exit when any one of them is closed, so Java leaves it to the programmer to handle what happens when a window is closed.

Window Listener

```
package java.awt.event;

import java.awt.event.WindowEvent;

public interface WindowListener
{
  /** Called when a window comes to the front */
  public void windowActivated(WindowEvent e);

  /** Called when window moves behind another window */
  public void windowDeactivated(WindowEvent e);

  /** Called when a close control is clicked */
  public void windowClosing(WindowEvent e);

  /** Called when the window has closed */
  public void windowClosed(WindowEvent e);

  /** Called when the window is first displayed */
  public void windowOpened(WindowEvent e);

  /** Called when the window is minimized */
  public void windowIconified(WindowEvent e);

  /** Called when the window is unminimized */
  public void windowDeiconified(WindowEvent e);
}
```

Window Events

- To handle window events, you implement **`WindowListener`**, and declare the 7 methods. Then you call **`addWindowListener(wl)`** on the **`JFrame`** to start receiving events.

Window Events

- **JFrame** has 4 default behaviors in response to the user clicking the close box of the window. These are controlled by **JFrame.setDefaultCloseOperation(op)**:
  - **JFrame.DO_NOTHING_ON_CLOSE**: calls **windowClosing(e)** on listeners. Listeners will call **dispose()** if they really want to close the window.
  - **JFrame.HIDE_ON_CLOSE**: just makes the window invisible. (**setVisible(false)**).
  - **JFrame.DISPOSE_ON_CLOSE**: calls **windowClosing(e)** and **windowClosed(e)** on listeners. Destroys the window.
  - **JFrame.EXIT_ON_CLOSE**: Terminates the JVM by calling **System.exit(0)**.

Menu Interaction

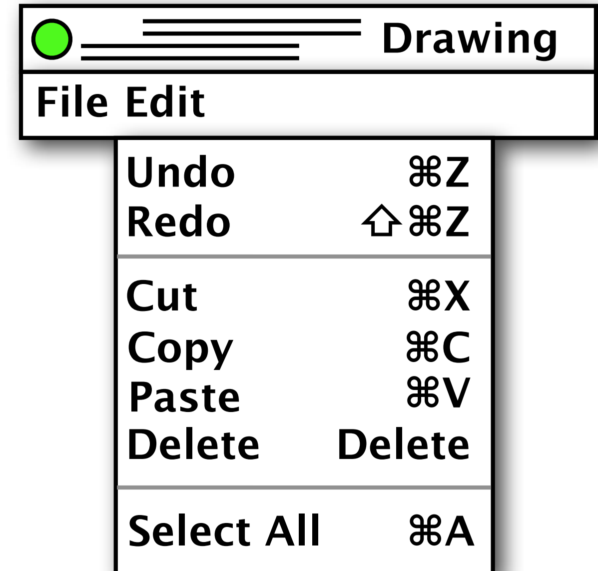
- Menu items behave like buttons, so you add an **`ActionListener`** to the menu item to get notifications that it was selected.
  - The source (**`ActionEvent.getSource()`**) gives the object clicked by the user, whether a **`JButton`** or **`JMenuItem`**.
- You can add a keyboard equivalent to a menu item using the following code **`JMenuItem.addAccelerator (javax.swing.KeyStroke.getKeyStroke (java.awt.event.KeyEvent.VK_N, java.awt.event.InputEvent.CTRL_MASK))`**.
  - This sets Control-N as the keyboard code to activate this menu item.

Menu Interaction

- Compare your program with popular applications to help determine appropriate menu options and arrangement.

Creating Menus

```
import java.awt.event.KeyEvent;
import java.awt.event.InputEvent;
import javax.swing.KeyStroke;

// ...

JMenuBar menuBar = new JMenuBar();
myFrame.setJMenuBar(menuBar);

// ...

JMenu editMenu = new JMenu("Edit");
menuBar.add(editMenu);

this.undoMenuItem = new JMenuItem("Undo");
this.undoMenuItem.addActionListener(this);
// Using Mac style cloverleaf modifier
this.undoMenuItem.setAccelerator(KeyStroke.getKeyStroke
(KeyEvent.VK_Z, InputEvent.META_MASK));
editMenu.add(this.undoMenuItem);

this.redoMenuItem = new JMenuItem("Redo");
this.redoMenuItem.addActionListener(this);
// Using Windows style control modifier
this.redoMenuItem.setAccelerator(KeyStroke.getKeyStroke
(KeyEvent.VK_Z, InputEvent.SHIFT_MASK + InputEvent.CTRL_MASK));
editMenu.add(this.redoMenuItem);

editMenu.addSeparator();

// ...
```



In-class lab exercise #6b

Creating Menus

Add some menus to your window using JMenuBar, JMenu, and JMenuItem.

6b

Creating Menus

Add menus to your **DrawFrame**:

1. Suggested menus are "File" and "Create".
2. For the File menu, you should have commands at least for "Open", "Save" and "Quit".
3. For the Create menu, you should have commands for "Line", "Rectangle", "Ellipse", and "Polygon".
4. Each menu item should have an **ActionListener** registered for the frame.
5. In your **actionPerformed(ActionEvent)**, test for which menu was activated and branch out to an empty method body (or perhaps print to the console a message like "Open menu selected")
6. Want your widgets to look like Windows/Mac/etc? Paste the following into your main() method:

```
try
{
  UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
}
catch (Exception e)
{
  e.printStackTrace();
}
```