CompSci 230
Software Construction

Swing 1 S1 2015

Authors: Tim Vaughan (Theme B lecturer, S2 2014), Clark Thomborson

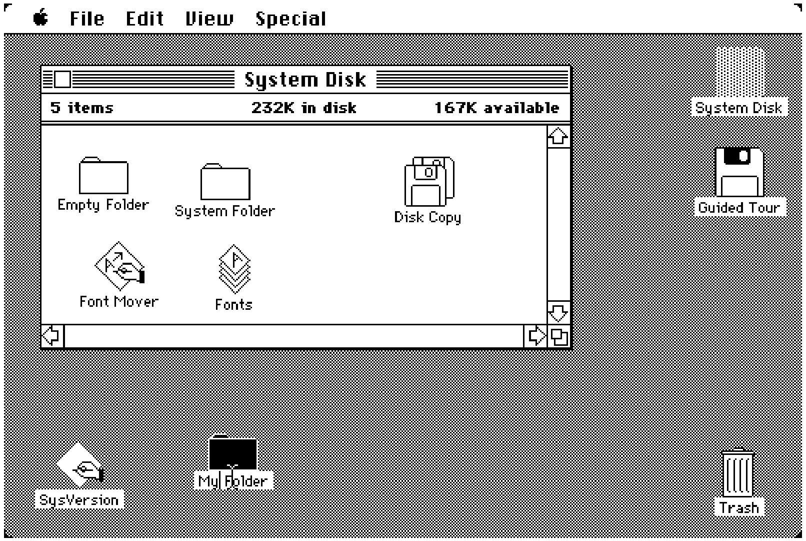
Learning Goals

- You will gain a high-level understanding of GUI Frameworks which is
  - Sufficient to get you started on Assignment 2 (in Swing)
  - Provides a foundation for our subsequent lectures (after break) on some of the most-important features of AWT and Swing.

History of Graphical User Interfaces (GUIs)

- In the beginning was the Command Line Interface (CLI)
- The first GUI was developed at Xerox PARC in the early 70s.
  - Desktop metaphor, mouse & keyboard, windows, menus, buttons, …
  - Xerox Alto (1973-), Star (1981-).
  - Not a commercial success, but is the basis for all subsequent GUIs.
- First commercially-successful GUI on personal computers:
  - Apple Macintosh (1984-).
- The X Window System (version 11, released 1987) ran on many platforms including Unix workstations, PCs, Macs.
  - "… an architecture-independent system for remote graphical user interfaces and input device capabilities.
  - "Each person using a networked terminal has the ability to interact with the display with any type of user input device." [Wikipedia]
- Windows 3.0 (1990-)
  - This was Microsoft's first successful GUI-based OS.

1960s mouse (Engelbart)

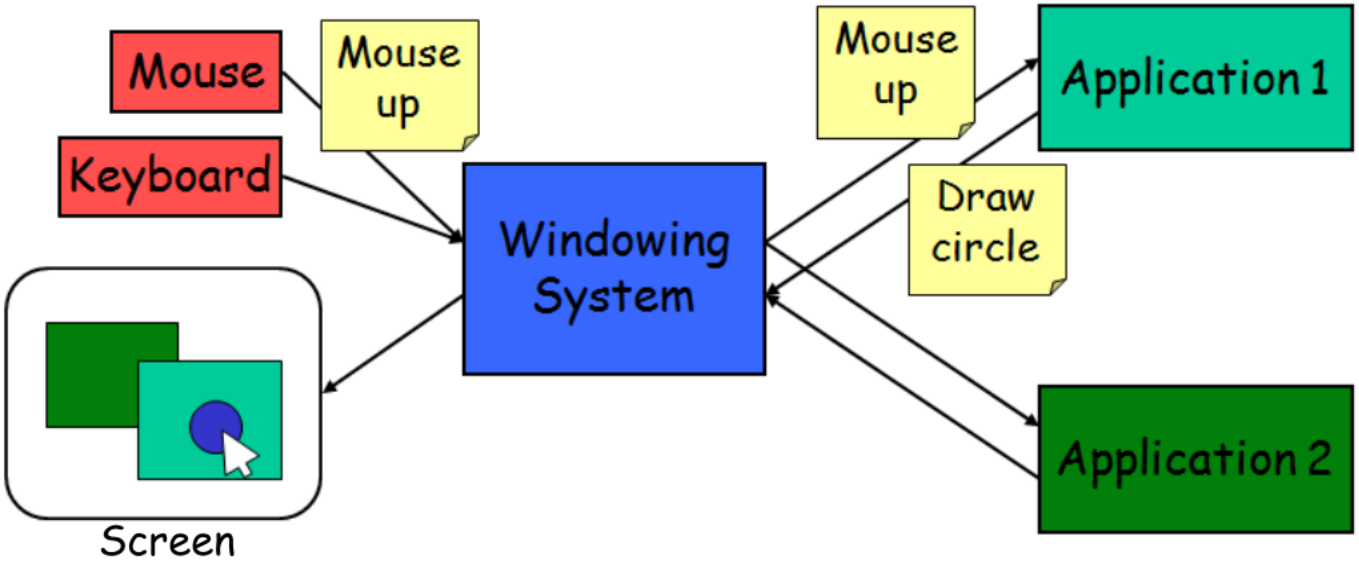


WIMPs

- “In a WIMP system:
  - A **window** runs a self-contained program, isolated from other programs that (if in a multi-program operating system) run at the same time in other windows.
  - An **icon** acts as a shortcut to an action the computer performs (e.g. execute a program …).
  - A **menu** is a text or icon-based selection system that selects and executes programs or tasks.
  - The **pointer** is an onscreen symbol that represents movement of a physical device [which] the user controls to select icons, data elements...” [Wikipedia]
- Typical design (from PARC)
  - Windowing system: handles low-level input/output (possibly over a network)
  - Window Manager: takes care of placement and appearance of windows
  - GUI Framework/Toolkit: software library, eases programmer’s burden.
    - Icon/Widget Graphic: object with functionality e.g. button, toolbar
    - Window Container: holds widgets and nested containers.
    - Events/messages: How windows communicate

Windowing System

- Manages input and output devices
  - e.g. graphics cards, screens, mice, keyboards,
- Sends input events from input devices to apps,
  - Receives and processes drawing commands from apps.
- May interact with remote applications.
  - X11 (1987-), Microsoft Remote Desktop Connection (1997-), Apple Remote Desktop (2002-).

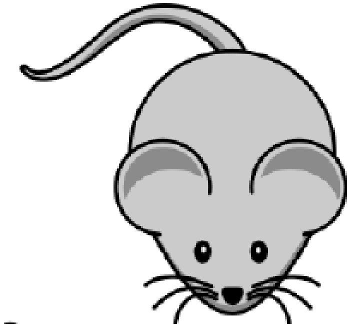


GUI Input Events

- Primitive Pointer Events
  - Mouse Moved
  - Mouse Down
  - Mouse Up

- Primitive Keyboard Events
  - Key down
  - Key up

- Complex Pointer Events
  - Click: mouse down, mouse up
  - Double Click: two clicks within a certain time
  - Enter: mouse moves into a region
  - Leave: mouse moves out of a region
  - Hover: mouse stays in a region for a time
  - Drag and Drop: mouse down, mouse moved, mouse up



Event Handlers

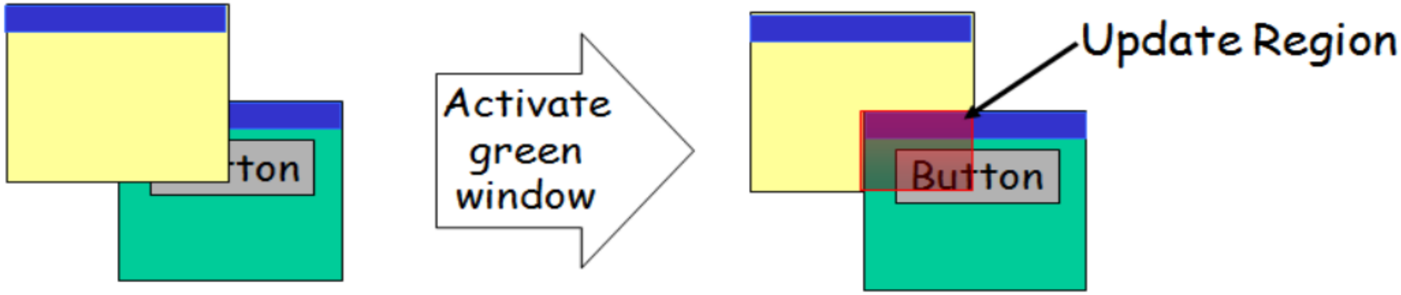
- **Input events** are routed through the windowing system, and then the GUI framework, to an **event listener** (a.k.a. **event handler**) of a **widget** (a.k.a. Swing Component, JavaFX control, ActiveX control, …)
  - Keyboard and mouse events are sent to the active ("**focus**") window.
    - Focus is usually selected by the user, but may be forced by the OS.
    - Within a window, a mouse event is usually routed to the widget that is displayed at the position of the mouse.
- Widget methods (of an appropriate type-signature) must be registered as event handlers with the GUI framework – otherwise no events will be routed to them.
  - **Handler registration**: A reference to an event-handling method is passed as an argument, in a method call to an event dispatcher.
  - **Handler callback**: an event dispatcher invokes a registered handler.
- App developers write event handlers which invoke application logic.
  - A mouse-click event could be handled by a "Save As" button. This handler method might enter a file-write task on a work-queue, then exit.
  - Event handlers should never perform lengthy computations.

“Painting” of Widgets

- Widgets have a visual representation.
  - Widgets must define (or inherit) a `paint()` method, then register it as a paint-event handler.
  - When it is invoked, a paint() method should render (or “paint”) its widget on the display – by sending commands to the windowing system. A widget is not visible to the user until it is rendered.
  - Paint events (a.k.a. update events) are dispatched to paint-event handlers through the GUI framework.
- Containers also have `paint()` methods.
  - A GUI container holds widgets and other GUI containers.
  - A container’s paint-event handler, when invoked, dispatches paint events to all visible widgets in the container.
  - Developers rarely have to write `paint()` methods for containers – the implementations in the GUI framework should dispatch paint events to anything that is inserted into a GUI container using its `add()` method.
- System-triggered paint events:
  - Widgets must be rendered whenever the display window is resized or its visible area is changed in some other way (e.g. because of window movement).
- Model-triggered paint events:
  - The GUI framework will generate paint events whenever the user-visible state of a widget is changed
  - For example, if a tick-box or menu-item has been selected, some text has been typed into a textbox, or a widget’s setter is invoked by a developer’s code, this “change of model” will trigger a paint event.
    - Goal: “the view should always correspond to the model”.
    - Developers can “read the model” by querying the state of a widget (using its getters).

Repaints and invalidations

- Developers can invoke the `repaint()` method of a widget or container.
  - This is a "nice" way to request a paint-event.
  - Repaint events are queued, and are coalesced – so that repaints cause at most 100 paint-events per second per widget.
- Developers should not (in general) throw invalidation events nor should they invoke invalidate() methods.
  - The GUI framework throws invalidation events at all currently-visible containers, whenever "their" region of the display must be repainted because of window movements and resizings.
  - The GUI framework's default invalidation-handler for a container will throw paint events at its contained widgets and its nested containers.
    - Widgets and containers that don't overlap the invalidated region do not receive paint() events from an invalidation: this is an important optimisation.



A Simple Swing App

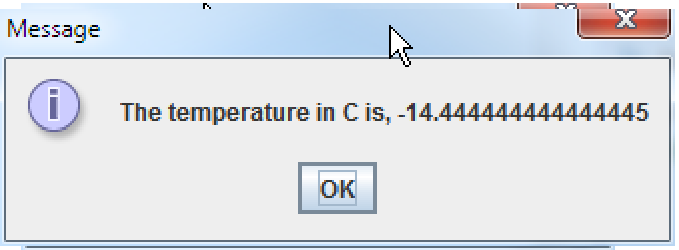
```
import javax.swing.*;

public class TempConvGUI {

    public static void main(String[] args) {
        String fahrString;
        double fahr, cel;

        fahrString = JOptionPane.showInputDialog("Enter the temperature in F");
        fahr = Double.parseDouble(fahrString);
        cel = (fahr - 32) * 5.0/9.0;

        JOptionPane.showMessageDialog(null,"The temperature in C is, " + cel);
    }

}
```

JOptionPane inherits from awt.Component.

A static method of the JOptionPane class. Instantiates and paints a container with several widgets; waits for the user to click OK.

The user-modified portion of the GUI model is returned as a String.

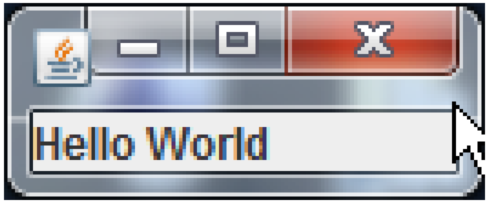
Another static method of the JOptionPane class. Instantiates and paints a container with several widgets; waits for the user to click OK.



Message

The temperature in C is, -14.444444444444445

HelloWorldSwing



Hello World

```
import javax.swing.*;

public class HelloWorldSwing {

  private static void createAndShowGUI()
  {
    //Create and set up the window.
    JFrame frame = new
      JFrame("HelloWorldSwing");
    frame.setDefaultCloseOperation(
      JFrame.EXIT_ON_CLOSE);

    JLabel label = new
      JLabel("Hello World");
    frame.getContentPane().add(label);

    //Display the window.
    frame.pack();
    frame.setVisible(true);
  }
```

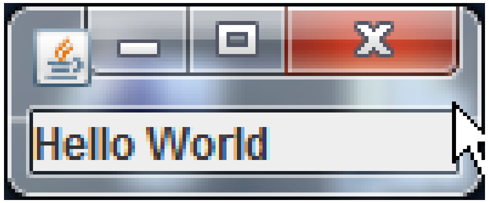
```
public static void main(String[] args) {
  // Schedule a job for the event-
  // dispatching thread: creating and
  // showing this application's GUI.
  javax.swing.SwingUtilities.
    invokeLater(new Runnable() {
      public void run() {
        createAndShowGUI();
      }
  });
}
}
```

JFrame inherits from awt.Component.

We add a JLabel to our JFrame instance.

The initial size of our frame is just large enough to display all of its widgets.

HelloWorldSwing



```
import javax.swing.*;

public class HelloWorldSwing {

  private static void createAndShowGUI()
  {
    //Create and set up the window.
    JFrame frame = new
      JFrame("HelloWorldSwing");
    frame.setDefaultCloseOperation(
      JFrame.EXIT_ON_CLOSE);

    JLabel label = new
      JLabel("Hello World");
    frame.getContentPane().add(label);

    //Display the window.
    frame.pack();
    frame.setVisible(true);
  }
```

```
public static void main(String[] args) {
  // Schedule a job for the event-
  // dispatching thread: creating and
  // showing this application's GUI.
  javax.swing.SwingUtilities.
    invokeLater(new Runnable() {
      public void run() {
        createAndShowGUI();
      }
  });
 }
}
```

The main() thread exits normally, but another thread executes the run() method in an anonymous class.

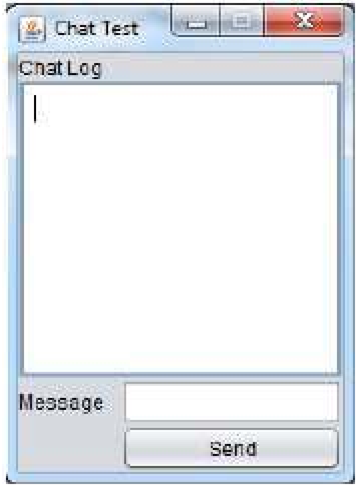
A frame doesn't have to be visible! The widgets in an invisible window will respond to method-calls.

The GUI Event Loop

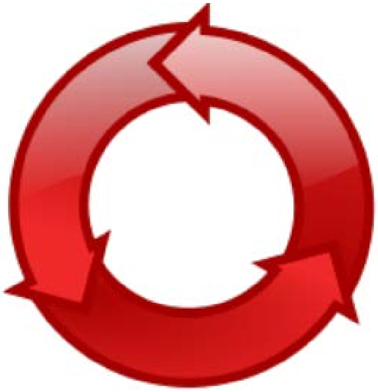
“Inversion of Control”

1. Application is started in its main().
2. Widgets are instantiated; their event-handlers are registered.
3. Event loop is started.
   a) Usually `main()` is terminated at this point. The GUI Framework is now in control! However the developer “sets the stage” in steps 1 and 2, so that the actors (the widgets and other objects) will respond appropriately to incoming events.

“Event Driven Programming”



4. The GUI Framework waits until there’s something (e.g. a mouse-click report from the Windowing System) in its event queue.
5. The GUI Framework’s event-dispatcher removes an event from the event queue, dispatches it to the appropriate handler, and returns to step 4.
   a) Most input events from the Windowing System will cause a cascade of internal events to occur within the GUI Framework, because many event-handlers will put additional events on the event queue.



Window Manager

- Definition by functionality: A window manager is any software which…
  - Controls the placement and appearance of all windows (but not the window contents) on all window-level operations (open, close, minimize, maximize, move, resize)
    - While relying on the application (which is probably running a GUI Framework) to paint a window's contents *after* the Window Manager has determined its position and visibility; and which
  - Is directly involved in starting and stopping GUI apps, and in handling window-focus events.
    - Note that these events determine which app is responsible for determining what should be displayed in a window..
- This definition is not entirely satisfactory, because the functionality of a window manager (as defined above) may be delivered (at least in part) by software which delivers many other functions.
  - In Windows computers, the window-management software is integrated with the operating system, so the window manager is better described as a "cluster of features" in the OS than as a distinct software component within the OS.
    - In Apple's OS X, different windowing systems may control different "layers" of the display, and you could be running a different window manager on each layer. Layer management is handled by the OS, which dispatches events to the window manager on affected layers.
  - The interface between a Windowing System and a Window Manager is somewhat arbitrary.
    - A window manager which enforces a standard "look and feel" by using only low-level graphic primitives, rather than using higher-level primitives provided by native-code OS libraries such as the Win32 GUI API, is doing "some of the work" that a Windowing System could do.
    - Note: a Windowing System may also provide widgets for a GUI Framework, see e.g. Eclipse's SWT.
  - Any GUI Framework which can handle many applications simultaneously, and which doesn't rely on an OS for its "internally-managed windows", is difficult to distinguish from a Window Manager.

Summary

- Concepts:
  - Window Manager, GUI Framework, Windowing System
    - As stack of (vaguely specified) functions, listed here from "high level" to "low level"
  - Event-driven programming, inversion of control
    - A new way to think about programming?
    - The job of main() is to "set the stage". During the actual "performance", the GUI Framework's event-dispatch loop controls "what happens next". Handlers "respond" to events by pushing other events onto the event queue, and not by directly invoking other methods.
  - GUI Containers and Widgets
    - The state of a widget is its portion of the "model", and its `paint()` method should update the user's "view" of this state – so that the view is (nearly) always consistent with the model.
      - □ Anything which changes the user-relevant state of a widget should cause a paint().
    - Developers don't invoke `paint()` directly in their code, unless they're implementing custom widgets!
      - □ The GUI Framework should generate paint-events at appropriate times, e.g. after `repaint()` is invoked by an event-handler, or a Window Manager advises of an invalidated region on the display.

Learning Goals: Review

- You will gain a high-level understanding of GUI Frameworks which is
  - Sufficient to get you started on Assignment 2 (in Swing)
  - Provides a foundation for our subsequent lectures (after break) on some of the most-important features of AWT and Swing.