Event Handling

How to manage event-code binding

- Switch Statement Binding
- Inheritance Binding
- Event Interfaces & Listeners
- Delegate Binding

Event Dispatch vs. Handling

- **Event Dispatch** phase addresses:
  - Which window receives an event?
  - Which widget processes it?
    - Positional dispatch
      - Bottom-up dispatch
      - Top-down dispatch
    - Focus dispatch

- **Event Handling** attempts to answer:
  - After dispatch to a widget, how do we bind an event to code?

Event-to-Code Binding

Key Question: How do we design our GUI architecture to enable application logic to interpret events once they've arrived at the widget?

- Design Goals:
  - Easy to understand (clear connection between each event and code that will execute)
  - Easy to implement (binding paradigm or API)
  - Easy to debug (how did this event get here?)
  - Good performance

Code-Binding Mechanisms

- Event Loop & Switch Statement Binding
- Inheritance Binding
- Event Interfaces & Listeners
- Delegate Binding

Event Loop and Switch Statement Binding (X11)

- All application events are consumed in one event loop (not by the widgets themselves)
- Outer switch statement selects window and inner switch selects code to handle the event
- Used in Xlib, Apple System 7, and, until recently, Blender

```
while( true ) {
    XNextEvent(display, &event); // wait next event
    switch(event.type) {
    case Expose:
       // ... handle expose event ...
       cout << event.xexpose.count << endl;
        break;
    case ButtonPress:
    // ... handle button press event ...
       cout << event.xbutton.x << endl;
        break;
    ...
```

WindowProc Binding (MS Windows)

- Each window registers a WindowProc function (Window Procedure) which is called each time an event is dispatched
- The WindowProc uses a switch statement to identify each event that it needs to handle.
  - There are over 100 standard events…

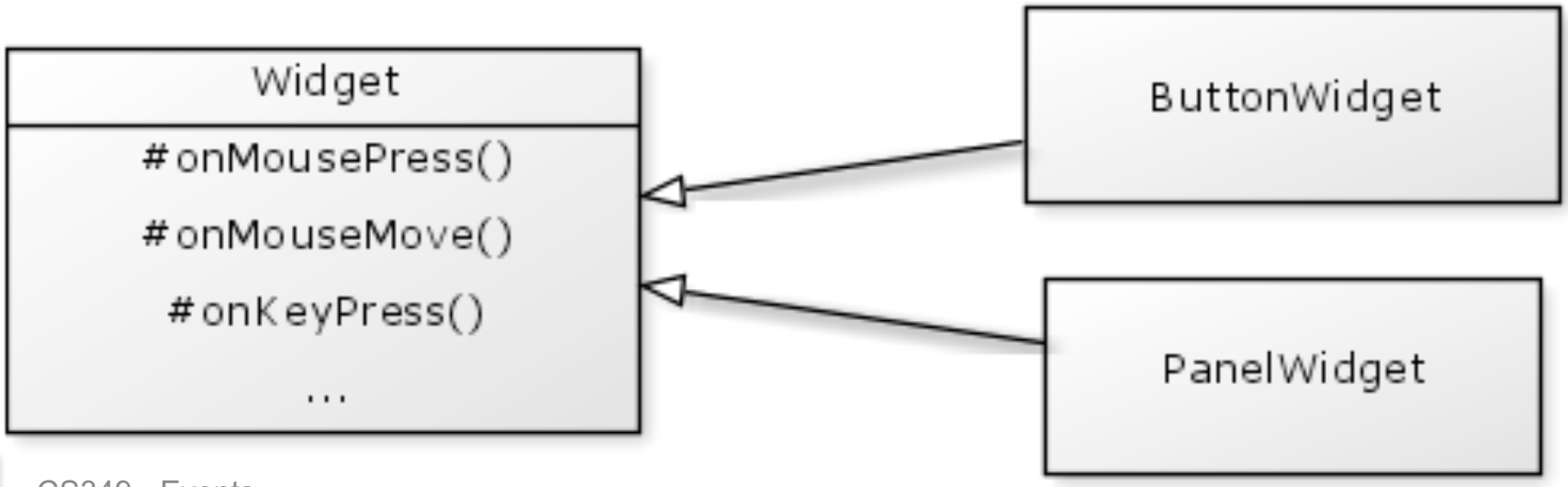
```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg,
                            WPARAM wParam, LPARAM lParam) {
  switch (uMsg) {
    case WM_SIZE: {
      int width = LOWORD(lParam); // low-order word.
      int height = HIWORD(lParam); // high-order word.
      // Respond to the message:
      OnSize(hwnd, (UINT)wParam, width, height);
    }
    break;
```

Code-Binding Mechanisms

- Event Loop & Switch Statement Binding
- Inheritance Binding
- Event Interfaces & Listeners
- Delegate Binding

Inheritance Binding (Java 1.0, OSX)

- Event is dispatched to an Object-Oriented (OO) widget
  - OO widget inherits from a base widget class with all event handling methods defined *a priori*
    - `onMousePress`, `onMouseMove`, `onKeyPress`, etc
  - The widget overrides methods for events it wishes to handle. e.g. Java 1.0, NeXT, OSX
    - Each method handles multiple related events



Inheritance Problems

1. Each widget handles its own events, or the widget container has to check what widget the event is meant for
2. Multiple event types are processed through each event method: complex and error-prone (just a switch statement again)
3. No filtering of events: performance issues (e.g. with frequent events, like mouse-move events)
4. It doesn't scale well: How to add new events?
   - e.g. `penButtonPress`, `touchGesture`, ….
5. Muddies separation between GUI and application logic: event handling application code is in the inherited widget
   - Take-home point: Use inheritance for extending class functionality, not for binding events

Code-Binding Mechanisms

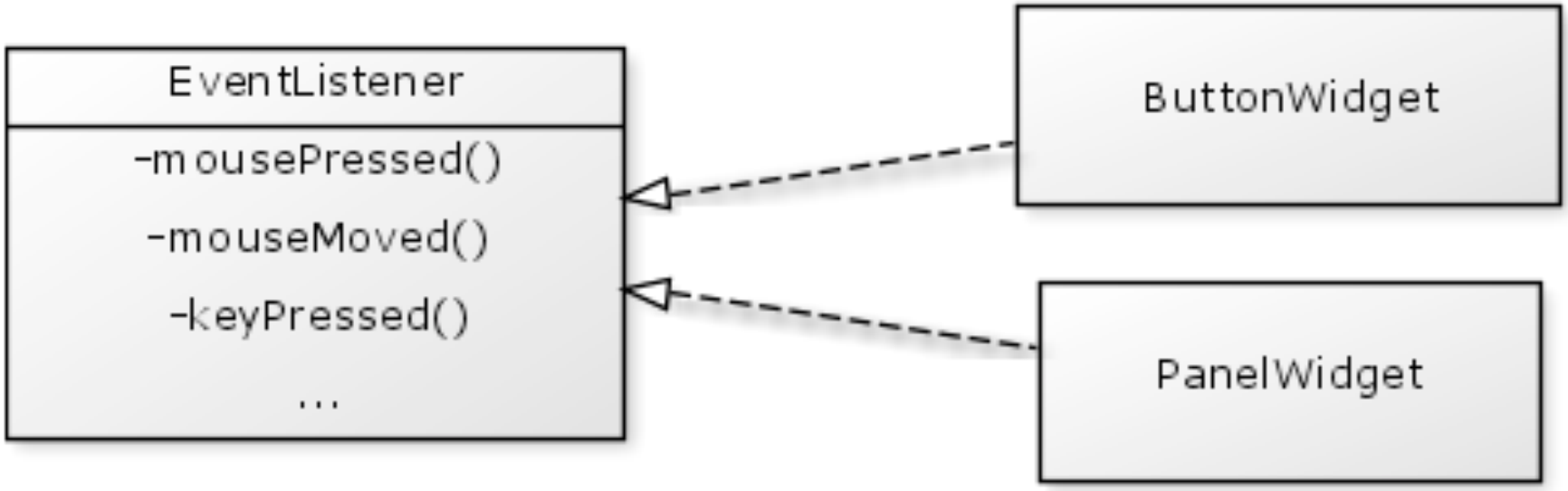
- Event Loop & Switch Statement Binding
- Inheritance Binding
- Event Interfaces & Listeners
- Delegate Binding

Event Interfaces

- Rather than subclass widget, define an *interface* for event handling
- Here, an *interface* refers to a set of functions or method signatures for handling specific types of events
- For example, in Java, can define an interface for handling mouse events
- Can then create a class that implements that interface by implementing methods for handling these mouse events

Listener *Interface* Binding (Java)

- Widget object implements event "listener" interfaces
  - e.g. `MouseListener`, `MouseMotionListener`, `KeyListener`, ...
- When event is dispatched to widget, the relevant listener method is called
  - `mousePressed`, `mouseMoved`, ...



```
public class MyAwesomePanel
             extends JPanel
             implements MouseMotionListener  {

   MyAwesomePanel() { }

   public void mouseDragged(MouseEvent e) {
      x = e.getX();
      y = e.getY();
      // Do everything else here
      // Assume that we need to repaint afterwards
      repaint();
   }

   public void mouseMoved(MouseEvent e) {
      // Empty body, forced to implement b/c of interface
   }
}
```

Interface Better, But Still Problems

Improvements:

- Each event type assigned to an event method
- Events are filtered: only sent to object which implements interface
- Easy to scale to new events: add new interfaces

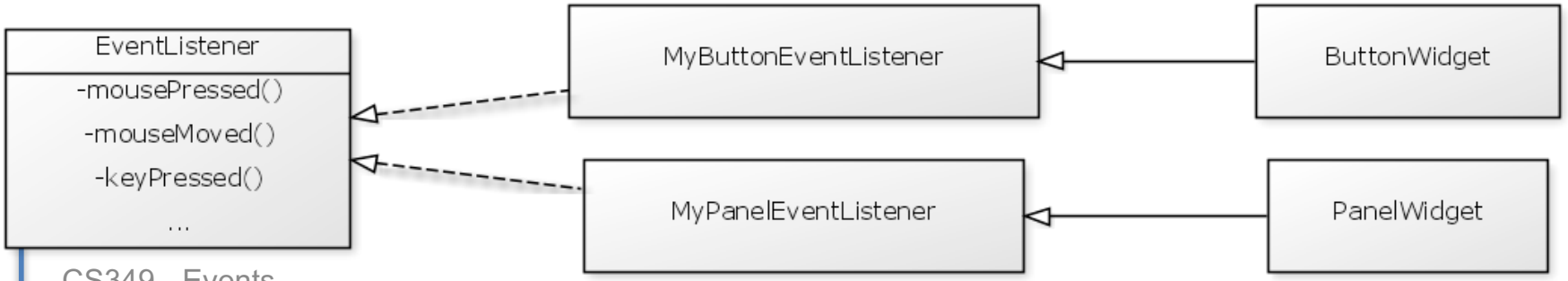
  e.g. `PenInputListener`, `TouchGestureListener`

Problems:

1. Each widget handles its own events, or widget container has to check what widget the event is meant for (i.e. no mediator)
2. Muddies separation between GUI and application logic: event handling application code is in inherited widget

Listener *Object* Binding (Java 1.1)

- Widget object is associated with one or more event listener *objects* (which implement an event binding interface)
  - e.g. `MouseListener`, `MouseMotionListener`, `KeyListener`, ...
- When event is dispatched to a widget, the relevant listener object processes the event with implemented method: `mousePressed`, `mouseReleased`, ...
- application logic and event handling are decoupled



MouseListener

```
public class MyImportantPanel extends JPanel  {

 MyImportantPanel() {
     this.addMouseMotionListener(new MyPanelListener());
  }

  // inner class listener
  class MyPanelListener implements MouseMotionListener {
    public void mouseDragged(MouseEvent e) {
      x = e.getX();
      y = e.getY();
      // Make some meaningful change to app
      repaint();
    }
    public void mouseMoved(MouseEvent e) { /* no-op */ }
```

Listener Adapter Pattern

- Many listener interfaces have only a single method
  - e.g. `ActionListener` has only `actionPerformed`
- Other listener interfaces have several methods
  - e.g. `WindowListener` has 7 methods, including `windowActivated`, `windowClosed`, `windowClosing`, ...
- Typically interested in only a few of these methods. Leads to lots of "boilerplate" code with no-op methods, e.g.

```
void windowClosed(WindowEvent e) { }
```

- Each listener with multiple methods has an Adapter class with no-op methods. Simply extend the adapter, overriding only the methods of interest.

MouseMotionAdapter

```
public class AdapterEvents extends JPanel  {

   AdapterEvents() {
       this.addMouseMotionListener(new MyListener());
   }

   class MyListener extends MouseMotionAdapter {
      public void mouseDragged(MouseEvent e) {
         x = e.getX();
         y = e.getY();
         // Do something meaningful here
         repaint();
      }
   }
```

Code-Binding Mechanisms

- Event Loop & Switch Statement Binding
- Inheritance Binding
- Event Interfaces & Listeners
- Delegate Binding

Delegate Binding (.NET)

- Interface architecture can be a bit heavyweight
- Can instead have something closer to a simple function callback (a function called when a specific event occurs)
- Delegates in Microsoft's .NET are like a C/C++ function pointer for methods, but they:
  - Are object oriented
  - Are completely type checked
  - Are more secure
  - Support multicasting (able to "point" to more than one method)
- Using delegates is a way to broadcast and subscribe to events
- .NET has special delegates called "events"

Using Delegates

1. Declare a delegate using a method signature

```
public delegate void Del(string m);
```

2. Declare a delegate object

```
Del handler;
```

3. Instantiate the delegate with a method

```
// method to delegate (in MyClass)
public static void MyMethod(string m) {
    System.Console.WriteLine(m);
}
handler = myClassObject.MyMethod;
```

4. Invoke the delegate

```
handler("Hello World");
```

Multicasting

- Instantiate more than one method for a delegate object

```
handler = MyMethod1 + MyMethod2;
handler += MyMethod3;
```

- Invoke the delegate, calling all the methods

```
handler("Hello World");
```

- Remove method from a delegate object

```
handler -= MyMethod1;
```

- What about this?

```
handler = MyMethod4;
```

Events in .NET

- Events are delegates that are designed for event handling
  - Delegates with restricted access
  - Declare an event object instead of a delegate object:

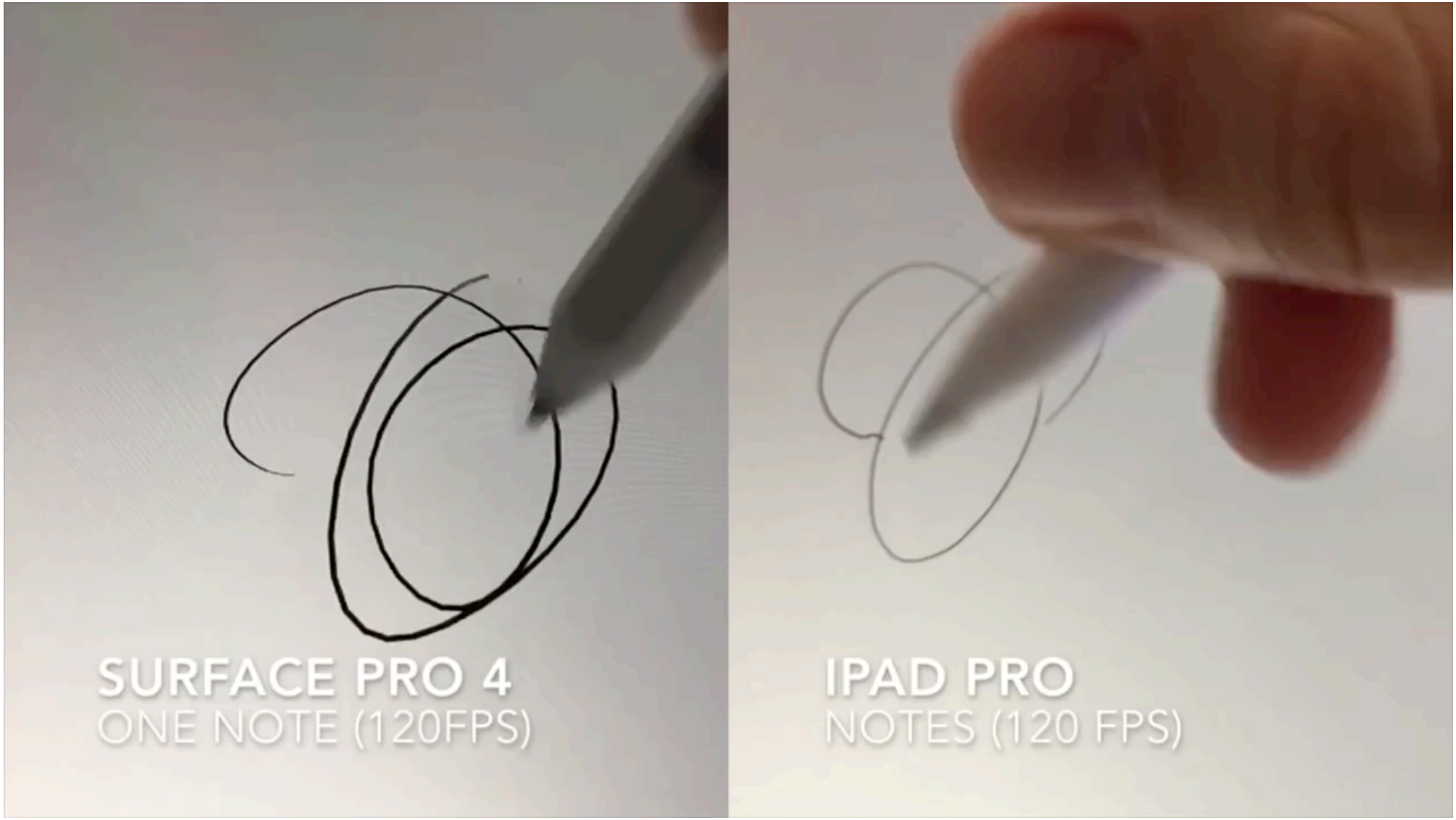
```
public delegate void Del(string message);

event Del handler;
```

- "event" keyword allows enclosing class to use delegate as normal, but outside code can only use the -= and += features of the delegate
- Gives enclosing class exclusive control over the delegate

Events for High Frequency Input

- Pen and touch generate a higher number and frequency of events than normal hardware
  - pen motion input can be 125Hz or higher
  - multi-touch hardware can generate many simultaneous contact/move events for all fingers
  - pen sensor is much higher resolution than display
- This is often faster than an application can handle it!
- Not all events are guaranteed to be delivered individually
  - All penDown and penUp, but may skip some penMove events
  - Event object includes array of “skipped” penMove positions
  - Android does this for touch input
- For things like pen input, use other methods to grab these events because of their high rate of generation



Surface Pro 4 vs iPad Pro pencil tracking
https://www.youtube.com/watch?v=pK41eAYNLu4