Event Binding

Different Approaches

Global Hooks

Event Dispatch vs. Event 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 answers:
  - After dispatch to a widget, how do we **bind** an event to code?

Event-to-Code Binding

- 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

Approaches

- Event loop "manual" binding
- Inheritance binding
- Listener Interface binding
- Listener Object binding
- Listener Adapter binding
- Delegate binding (C#)

Event Loop and Switch Statement Binding

- All events consumed in one event loop (not by widgets)
- Switch selects window and code to handle the event
- Used in Xlib and many early systems

```
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 Variation

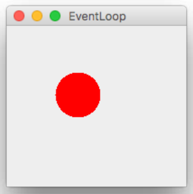
- Each window registers a WindowProc function (Window Procedure) which is called each time an event is dispatched
- The WindowProc uses a switch statement to identify each event that it needs to handle.
  – There are over 100 standard events...

```
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;
```

Java Event Queue

- Java has an event queue, can use it like an event loop
- Available from java.awt.Toolkit:
  - Toolkit.getDefaultToolkit().getSystemEventQueue()
- java.awt.EventQueue has methods for:
  - Getting current event, next event
  - Peeking at an event
  - Replacing an event (push())
  - Checking whether current thread is dispatch thread
  - Placing an event on the queue for later invocation

EventLoop.java

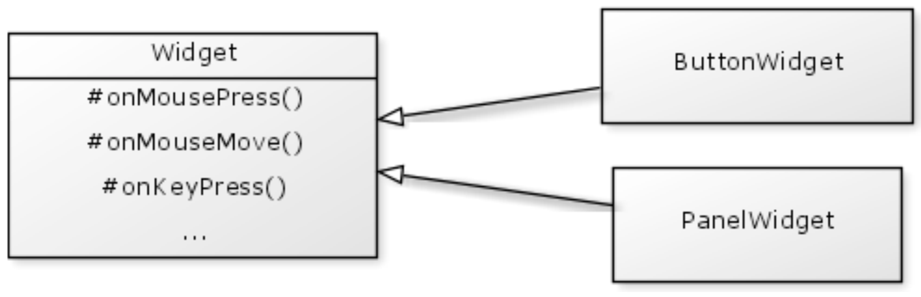


```
public class EventLoop extends JPanel  {
   EventLoop() {
      EventQueue eq = Toolkit.getDefaultToolkit().
                                         getSystemEventQueue();
        eq.push(new MyEventQueue());
   }
   public static void main(String[] args) {
      EventLoop panel = new EventLoop();
      ...
   }
   private class MyEventQueue extends EventQueue {
      public void dispatchEvent(AWTEvent e) {
         if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
           MouseEvent me = (MouseEvent)e;
            x = me.getX();
            y = me.getY();
         }
         repaint();
         super.dispatchEvent(e);
      }
  }
```

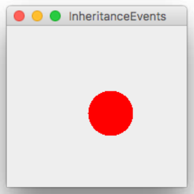
Inheritance Binding

- 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
  - Each method handles multiple related events
- Used in Java 1.0



InheritanceEvents.java



```
public class InheritanceEvents extends JPanel  {
   public static void main(String[] args) {
      InheritanceEvents p = new InheritanceEvents();
      ...
      // enable events for this JPanel
      p.enableEvents(MouseEvent.MOUSE_MOTION_EVENT_MASK);
   }

   protected void processMouseMotionEvent(MouseEvent e) {
      // only detects button state WHILE moving!
      if (e.getID() == MouseEvent.MOUSE_DRAGGED)
         colour = Color.RED;
      else
         colour = Color.GRAY;
      x = e.getX();
      y = e.getY();
      repaint();
   }
```

Inheritance Problems

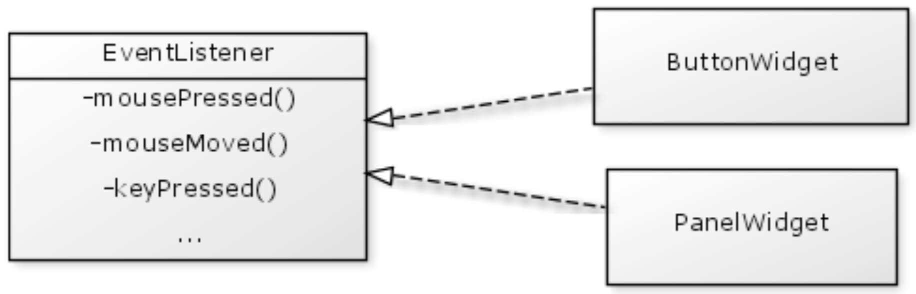
- Each widget handles its own events, or the widget container has to check what widget the event is meant for
- Multiple event types are processed through each event method
  – complex and error-prone, just a switch statement again
- No filtering of events: performance issues
  – consider frequent events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

Event Interfaces

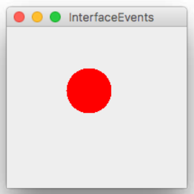
- Define an Interface for event handling
  – collection of method signatures for handling specific events
  – e.g. an Interface for handling mouse events
- Can then create a class that implements that interface by implementing methods for handling these mouse events

Listener Interface Binding (Java)

- Widget object implements event “listener” Interfaces
  - e.g. MouseListener, MouseMotionListener, KeyListener, …
- When event is dispatched, relevant listener method is called
  - mousePressed, mouseMoved, …



InterfaceEvents.java



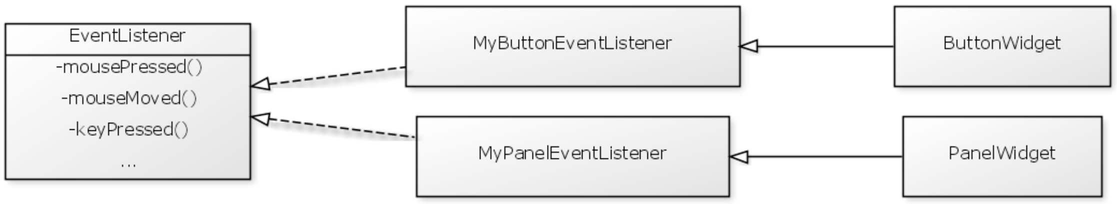
```
public class InterfaceEvents extends JPanel
                        implements MouseMotionListener {
   public static void main(String[] args) {
      InterfaceEvents panel = new InterfaceEvents();
       ...
   }

   InterfaceEvents() {
      this.addMouseMotionListener(this); // add listener
   }

   public void mouseDragged(MouseEvent e) {
      x = e.getX();
      y = e.getY();
      repaint();
   }
   public void mouseMoved(MouseEvent e) { /* no-op */  }
```

Listener Interface Better, But Still Problems

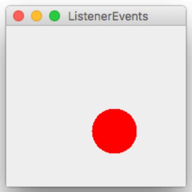
- Improvements:
  - Each event type assigned to an event method
  - Events are filtered: only sent to object implementing interface
  - Easy to scale to new events, just add new interfaces
    e.g. PenInputListener, TouchGestureListener
- Problems:
  - Each widget handles its own events, or widget container has to check what widget the event is meant for (i.e. no mediator)
  - 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



ListenerEvents.java

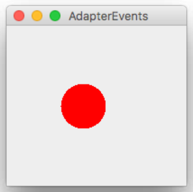


```
public class ListenerEvents extends JPanel  {
  public static void main(String[] args) {
      ListenerEvents panel = new ListenerEvents();
      ...
  }

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

  // inner class listener
  class MyListener implements MouseMotionListener {

    public void mouseDragged(MouseEvent e) {
      x = e.getX();
      y = e.getY();
      repaint();
    }

    public void mouseMoved(MouseEvent e) { /* no-op */ }
```

Listener Adapter Binding

- 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.

AdapterEvents.java



```
public class AdapterEvents extends JPanel  {
   public static void main(String[] args) {
      AdapterEvents panel = new AdapterEvents();
      ...
   }

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

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

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 message);
```

2. Declare a delegate object

```
Del handler;
```

3. Instantiate the delegate with a method

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

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 a delegate 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
- Outside code can’t wipe out delegate list, can’t do this:

  ~~`handler = MyMethod4;`~~

- Can have anonymous delegate events (similar to Java style):

```
b.Click += delegate(Object o, EventArgs e) {
      Windows.Forms.MessageBox.Show("Click!");  };
```

Global Event Queue “Hooks”

- An application monitors BWS events **across all applications**
- Can also inject events **to another application**
- This can be a very useful technique
  - examples?
- This can be a security issue
  - examples?

- Take a look at jnativehook
  - library to provide global keyboard and mouse listeners for Java.
  - https://github.com/kwhat/jnativehook/

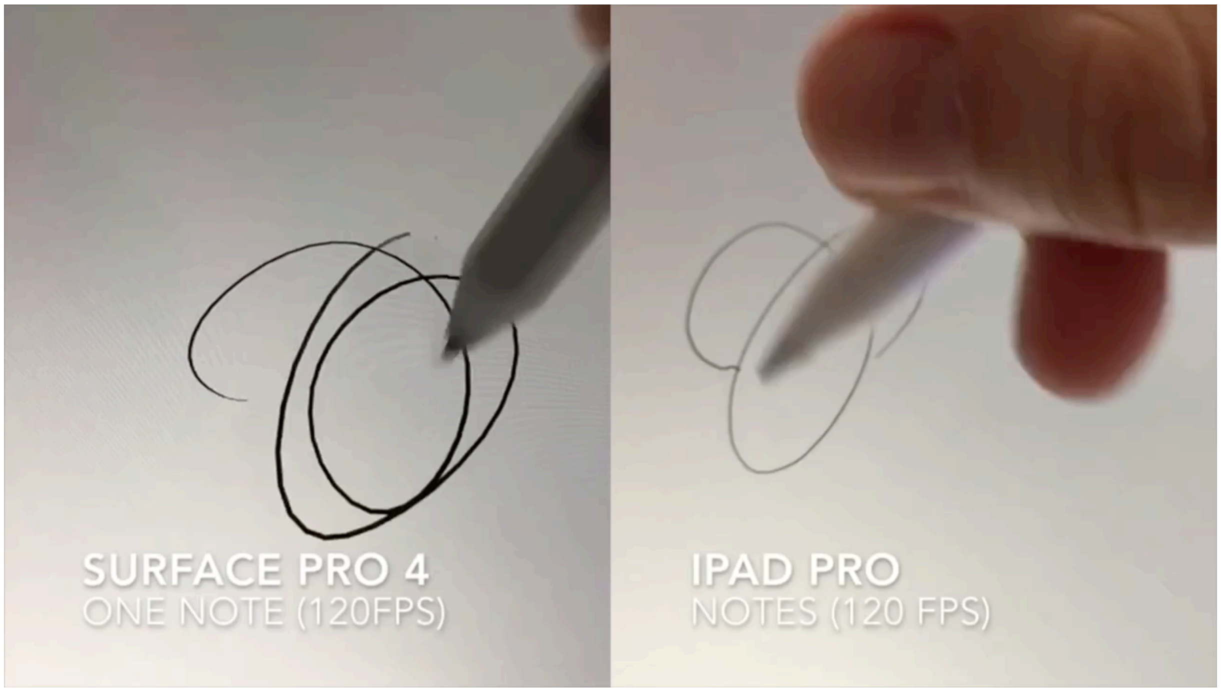
Global Hooks for Awareness

- Some application monitor level of “activity” using global hooks
- When activity drops, can do something
  - IM client: set state to “away”
  - Screensaver: start screensaver

Global Hooks in Tap-Kick-Click research prototype
– https://youtu.be/pqycjWHoI2w

Events for High Frequency Input

- Pen and touch generate many high frequency events
  - pen motion input can be 120Hz or higher
  - pen sensor is much higher resolution than display
  - multi-touch generates simultaneous events for multiple fingers
- **Problem:** These events often too fast for application to handle
- **Solution:** Not all events delivered individually:
  - e.g. all penDown and penUp,
    but may skip some penMove events
  - Event object includes array of "skipped" penMove positions
  - (Android does this for touch input)



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