CS 349
Events

Byron Weber Becker
Spring 2009

11-May-2009 Announcements

- New students in the course need to get in touch with me ASAP.

Events Defined

- Event: noun: *a thing that happens, especially one of importance* example: the media focused on events in the Middle East
- Event: a structure used to notify an application of an events occurrence
- Examples:
  - Keyboard (key press, key release)
  - Pointer Events (button press, button release, motion)
  - Window crossing (enter, leave)
  - Input focus (gained, lost)
  - Window events (exposure, destroy, minimize)
  - Timer events

Events: Why do we need them?

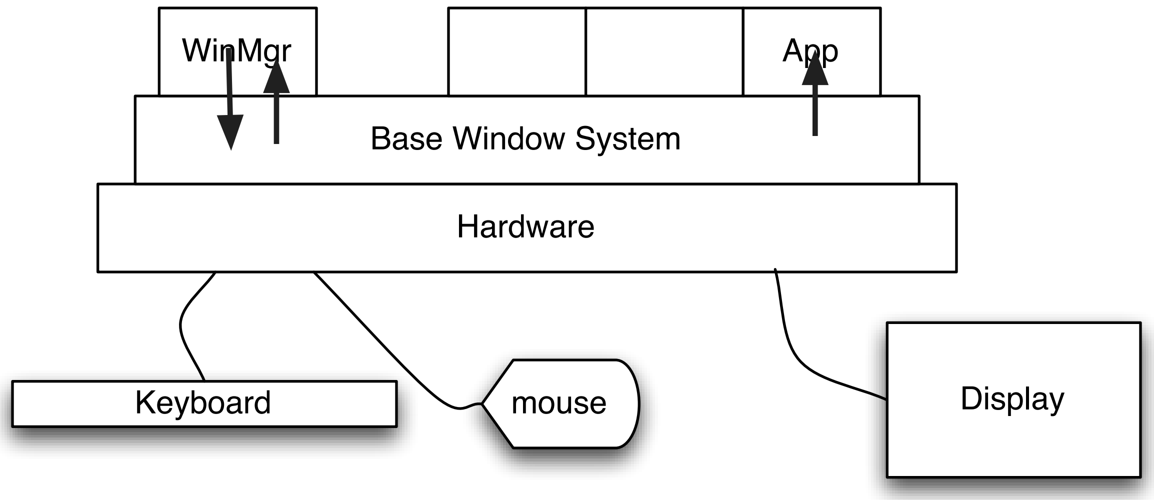
- Users have lots of options in a modern interface
- Need a uniform, well-structured, way to handle them
- Need to be able to handle *any* event, including those that aren’t appropriate given the current state of the app
  - eg: clicking on a button that is currently disabled

Role of the Base Window System

- Collect event information
- Put in a known structure
- Order the events by time
- Decide to which application/window the event should be dispatched.
- Deliver the event.

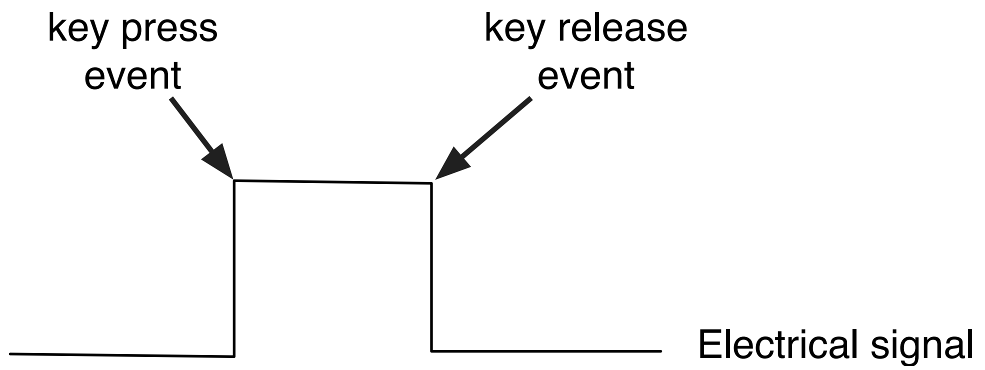
BWS: Collecting Events

- Some events come from the user via the underlying hardware; some from the window manager.



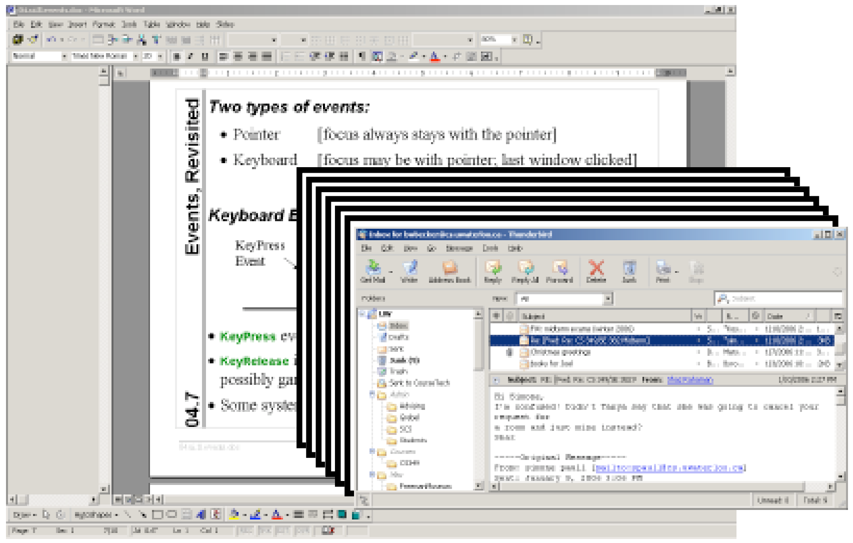
BWS: Collecting Events

- Key events:
  - Key press: put char on screen
  - Key release: usually ignored except to tell if key is being held for auto-repeat purposes
  - Scan codes
- Mouse button events:
  - press and release *are* differentiated
- Mouse motion events:
  - mouse moved
  - mouse dragged



BWS: Collecting Events

- Damage Events
  - May be a long sequence of damage events
  - Responding to them all can bog the system down
  - X includes a field indicating a minimum number that are yet to come. Ignore damage event unless that field is 0.



Events, Revisited

Two types of events:

- Pointer [focus always stays with the pointer]
- Keyboard [focus may be with pointer; last window clicked]

Keyboard E

KeyPress Event

- KeyPress ev
- KeyRelease possibly ga
- Some syste

04.7

BWS: Known structure : X

- X uses a C union

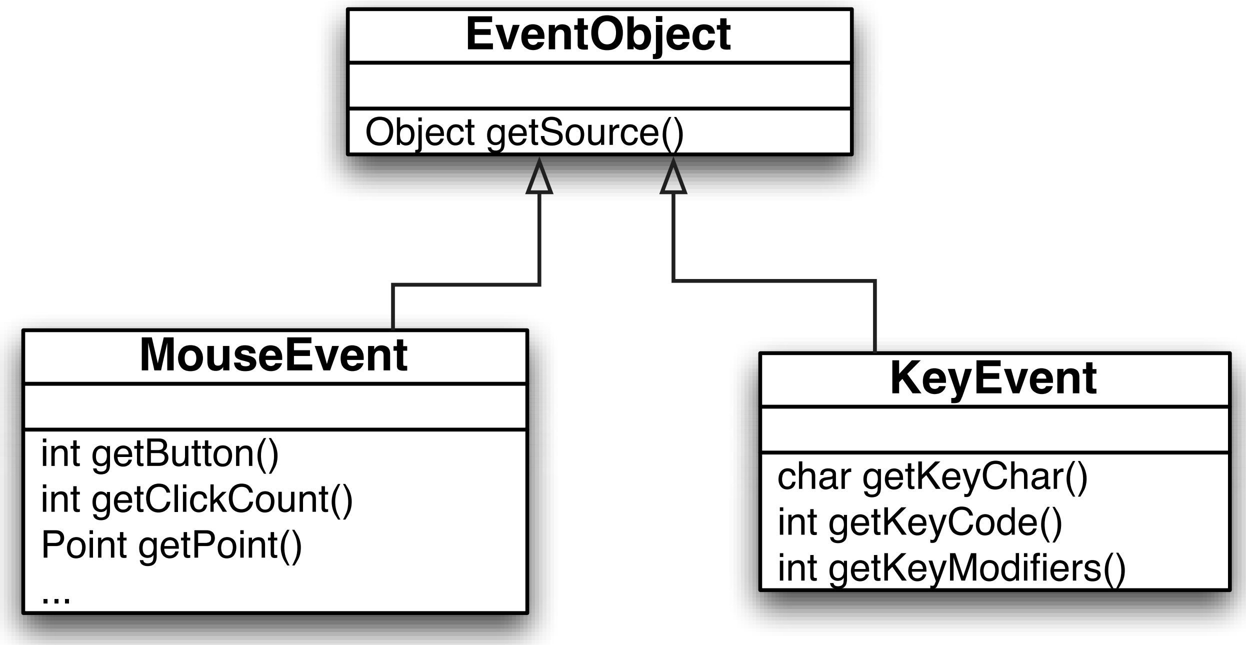
```
typedef union {
   int type;
   XKeyEvent xkey;
   XButtonEvent xbutton;
   XMotionEvent xmotion;
   // etc.
}
```

- Each structure contains at least the following

```
typedef struct {
   int type;
   unsigned long serial;
   Bool send_end;   // from SendEvent request?
   Display* display;
   Window window;
} X___Event
```

BWS: Known Structure: Java

- Java uses an inheritance hierarchy
- Each subclass contains additional information, as required (not shown)



BWS: Order by Time

- Use an Event Queue to maintain a list in order.
- In X, applications get the next event with
  - XNextEvent(Display* display, XEvent* evt)
    - Gets and removes the next event in the queue. If empty, it blocks until another event arrives.
  - XWindowEvent(dispaly, w, event_mask, event_return)
    - Searches the queue for an event for the specified window and of the specified type. Blocks if there isn't one.
- Blocking isn't good if you have animations or other stuff happening

```
XEvent event;

while (true)
{  XNextEvent(xinfo.display, &event);
   switch (event.type)
   {  case Expose:
         if (event.xexpose.count == 0)…
         break;

      case ButtonPress:
         // handle event
         break;

   }
   repaint(…);
}
```

```
XEvent event;

while (true)
{  if (XPending() > 0)
   {   XNextEvent(xinfo.display,
                          &event);
      switch (event.type)
      {  case Expose:
            if (event.xexpose.count == 0)…
            break;

         case ButtonPress:
            // handle event
            break;

         case TimerEvent:
             doAnimation(…);
             break;
      }
   }
   sleep(…);
   serviceTimers();
}
```

BWS: Event Dispatch

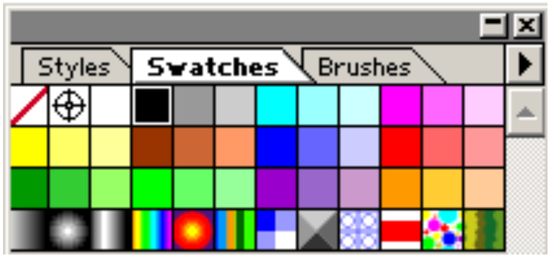
- Two approaches
  - Positional dispatch
    - Bottom-up dispatch
    - Top-down dispatch
  - Focus dispatch

BWS: Dispatch: Positional

- Basic strategy: Send input to the component the mouse is over.
- Primary issue: Components can overlap, so which one should receive the event?
  - Bottom-up
  - Top-down

BWS: Positional: Bottom-up

- Leaf node in the interactor tree receives the event
  - Can process the event itself
  - Send the event to its parent (who can process it or send to its parent...)



- Why send to its parent?
  - Example: A palette of colour swatches may implement the colours as buttons. But palette as a whole needs to track the currently selected colour. Easiest if the palette deals with the events.
- Strategy is applicable architectures where everything the BWS knows about each component.

BWS: Positional: Top-down

- Highest level node in the interactor tree receives the event.
  - Can process the event itself.
  - Can pass it on to a child component.
- In bottom-up dispatch, the BWS does this to determine which of the leaf nodes to deliver the event to.
- Advantage of top-down: Allows common activities like event logging
  - Wrap the top-level component with a proxy component
  - Proxy component logs each event as it arrives, then dispatches it to the appropriate child component.
  - Can then replay the events to replay a user's interaction with the system.

BWS: Positional Dispatch Limits

- Positional dispatch is sometimes inappropriate.
  - Send keystrokes to scrollbar if mouse over the scrollbar?
  - Mouse starts in a scrollbar, but then moves outside the scrollbar. Send the events to the adjacent component?
  - Mouse button press event in in one button component but release is in another. Each button gets one of the events?
- Conclusion: Sometimes we need to give one control the "focus".

BWS: Dispatch: Focus

- One control has the focus
  - Events go to that control, regardless of the mouse position.
- Need to pay attention to focus for both keyboard and mouse events:
  - Mouse down on a button, move off of it, release (mouse focus)
  - Click on a text field, move mouse off, start typing (keyboard focus)
- Only one control should have control at a time
- Need to gain and lose focus at appropriate times
  - Transfer focus on a user click
  - Transfer focus on a tab

University of Waterloo

BWS: Dispatch: Focus

- Even though a component has focus, it should not necessarily receive *every* event:
  - Need to be able to click on another control to change focus
  - Paint/damage events not necessarily associated with the component that has focus
    - Example: moving a slider has an effect on some other component which is repainted
- Conclusions:
  - Mouse-down events should go to the component under the cursor
  - *Input* events go to the component with focus; other events may go elsewhere.
  - Often helpful to have an explicit focus manager in a container component to manage which component has the focus.
  - Need to explicit handle focus issues in A1.

BWS: Dispatch: Accelerator Keys

- Accelerator Keys provide two ways to invoke the same functionality:
  - Via the keyboard: Seems natural to send them to the component with keyboard focus
  - Via the menu: Click on the menu and *it* gets the focus. So now who gets the command?
- Have two places in code that do the same thing?
- Alternative:
  - Menus register keyboard accelerators with specific menu items.
  - The GUI toolkit intercepts accelerators and forward to the appropriate menu to be handled

BWS: Event Delivery

- An event happens…
- The toolkit decides which component it should be dispatched to.
- Now, how do we actually deliver it? How do we structure our GUI architecture to deliver the event information to the code that should handle it?
- Lots of approaches -- X and a case statement is just one. Criteria:
  - Easy to bind event to code
  - Clean, easy to understand what happened and why
  - Good performance

```
XEvent event;

while (true)
{  XNextEvent(xinfo.display, &event);

   switch (event.type)
   {  case Expose:
         ________________________
         break;

         case ButtonPress:
         ________________________
         break;

         case KeyPress:
         ________________________
         break;
   }
}
```

The event loop is (nearly) always the same. It's only the processing in reaction to each event that changes. Can we factor out the event loop?

BWS: Delivery Options (1/3)

- Nested Case Statements (X, Original Mac)
  - Usually used nested case statements (as above). The outer case statement to select the window and the inner case to select the code to handle the event. (Both hide the outer case statement in the BWS.)
- Event Tables (GIGO from Sun)
  - Each window has an event table, consisting of addresses of C procedures that should be called for a specific event. Index the table based on event type and call the procedure found there. Fill tables with default values.
- Callbacks (Xtk, Motif)
  - Similar to event tables, but distributed to individual components.

BWS: Delivery Options (2/3)

- WindowProc: MS Windows
  - Each window has just one callback, called a *WindowProc*. The WindowProc uses a case statement to identify each event that it needs to handle. There are over 100 standard events. Rather than handling all of them, it can delegate to another WindowProc.
- Subclassing: Java 1.0 (Mac OS X?)
  - BWS directs events to the component in which it occurs. That component inherits from an abstract class, overriding any methods it needs to modify to handle the event.
- Listeners: Java 1.1 and later
  - Register objects implementing a specific interface with the component. Appropriate method in each object is called when event occurs.

BWS: Delivery Options (3/3)

- Delegates: .NET
  - Only one of the methods in a listener’s interface is called. Why not provide just a method?

BWS: Delivery: Inheritance

```
class Button
{ int x, y;
  String label;
  boolean enabled;
  …

  void mouseDown(…)
  {}

  void mouseUp(…)
  {}

  void keyPressed(…)
  {}
```

```
class MyButton
        extends Button
{ LSystem lsys;
  …

  void mouseUp(…)
  { if (lsys.started)
    { lsys.pause();
      super.label =
       “Start”;
    } else
    { lsys.start();
      super.label =
        “Pause”;
}
```

13-May-2009 Announcements

- Please feel free to edit the wiki with questions, tips, interesting UI stuff, etc.
  - But… please use the summary field when you update to help us see what changed.
- Office hours for Jaime and Matt are now posted on the course web page.
- mono (open source version of .NET) is installed in the VM, but doesn't work with assemblies (ie: -r:System.Windows.Forms.dll). Kudos to anyone who can figure out why.

BWS: Delivery: Listeners

- Use the Strategy design pattern to factor out the behaviour unique to a particular UI component.
- Provide the component with one or more objects implementing a particular interface (set of methods). When the event occurs, the relevant method in the listener objects are called.

BWS: Delivery: Listeners 1

```
public interface ActionListener extends EventListener
{ void actionPerformed(ActionEvent e);
}

public class ActionEvent extends …
{ // lots of fields omitted
  int getModifiers()…     // modifier keys down?
  long getWhen()…         // when did it happen?
  ...
}
```

BWS: Delivery: Listeners 2

```
public class JButton extends...
{  List<ActionListener> listeners;

   void addActionListener(ActionListener l)
   {  listeners.add(l);
   }

   void processActionEvent(ActionEvent ae)
   {  for(int i=0; i<listeners.length(); i++)
      {  listeners.get(i).actionPerformed(ae);
      }
   }
}
```

Listeners 3a: top-level class

```
class MyActionListener implements ActionListener
{  private LSystem lsys;
   ...
   public void actionPerformed(ActionEvent ae)
   {  if (lsys.isPaused())    lsys.play();
      else                    lsys.pause();
   }
}
public class MyComponent extends …
{  private JButton playPause = new JButton(…);
   private LSystem lsys;
   public MyComponent(…)
   {  playPause.addActionListener(
         new MyActionListener(lsys));
}
```

Listeners 3b: nested class

```
public class MyComponent extends …
{  private JButton playPause = new JButton(…);
   private LSystem lsys;
   public MyComponent(…)
   {  playPause.addActionListener(
         new MyActionListener());

   class MyActionListener implements ActionListener
   {  public void actionPerformed(ActionEvent ae)
      {  if (lsys.isPaused())   lsys.play();
         else                   lsys.pause();
      }
   }
}
```

University of Waterloo

Listeners 3c: anon inner class

```
public class MyComponent extends …
{  private JButton playPause = new JButton(…);
   private LSystem lsys;
   public MyComponent(…)
   {  playPause.addActionListener(
            new ActionListener() {
              public void actionPerformed(ActionEvent ae)
              {  if (lsys.isPaused())   lsys.play();
                 else                   lsys.pause();
              }
            });
   }
}
```

Listeners 3d: mix listener & class

```
public class MyComponent extends …
             implements ActionListener
{  private JButton playPause = new JButton(…);
   private LSystem lsys;
   public MyComponent(…)
   {  playPause.addActionListener(this);

   public void actionPerformed(ActionEvent ae)
   {  if (lsys.isPaused())   lsys.play();
         else                lsys.pause();
   }
}
```

- Quick and dirty; often found in Web-based examples
- Not as desirable as other approaches

Inheritance vs. Listeners 1

- Delivering events by overriding methods (inheritance) leads to a huge class tree or convoluted code
  - Every button must be subclassed to respond to clicks
  - Everything else about the button remains the same
  - Alternative: overridden methods handle include a switch to include code for many different button instances
- Inheritance does not lend itself to maintaining a clean separation between the application model and the GUI.
- No filtering of events; every event is delivered, resulting in performance issues

Inheritance vs. Listeners 1

- Listener approach factors out the behaviour that is unique to each application
  - Application provides an object implementing the particular listener interface and the code needed for a particular button
- This is a common approach in UI toolkits:
  - Delegate customizable, application-specific functionality to configurable run-time objects.
  - Next step?

Listeners: Adapter pattern

- Many listener interfaces have only a single method; others have more.
  - WindowListener has 7, including
    - windowActivated(WindowEvent e)
    - windowClosed(WindowEvent e)
    - windowClosing(WindowEvent e)
- Typically interested in only a few of these methods. Leads to lots of "boilerplate" code.
- Each listener with multiple methods has an adapter with null implementations of each method. Simply extend the adapter, overriding only the methods of interest.

Adapter

```
JFrame f = new JFrame();

f.addWindowListener(new WindowListener() {
    public void windowClosed(…) {}
    public void windowClosing(…) {
        System.exit(0);
    }
    public void windowActivated(…) {}
    // and 3 others
});

// Compare to:
f.addWindowListener(new WindowAdapter() {
    // Just override the method we’re interested in
    public void windowClosing(…) {
        System.exit(0);
    }
    });
```

BWS: Delivery: Delegates & .NET

- Designed by Microsoft
- Allegedly intended to be cross-platform, but architecture, conventions clearly rooted in Windows
  - Example: Very easy to use native libraries compared to Java (using P/Invoke), but mechanisms not designed with cross-platform use in mind (no generic method of loading dynamic libraries)
- But, still a number of significant improvements in basic architecture of the VM, core system, and C# language
  - Many improvements noteworthy for building GUIs

C# and .NET

- C# and .NET architecture very, very Java-esque, but with more syntactic sugar
  - And more liberal use of Capital Letters!
- Once you know Java and Swing, C# is easily learned
- Example:
  - Java: System.out.println(“CS 349 is the best class ever!”);
  - .NET: System.Console.WriteLine(“No, SE 382 is the best ever!”);

.NET + Mono

- Mono an open source implementation of C# and .NET by Novell
  - GPL, LGPL, and MIT licenses
- Mono 2.0.1 includes WinForms compatibility (basic GUI system in .NET)
  - Most basic .NET GUIs will work in Mono
- Mono also has bindings to GTK through GTK#
- Mono 2.0.1 installed in your VM
  - PATH already set up to execute mono binaries

Using Mono

- gmcs is the mono compiler

```
gmcs myfile.cs -r:System.Windows.Forms.dll
  -r:System.Drawing.dll
```

- This will compile myfile.cs to myfile.exe
  - (Yes, ".exe", even on Linux – remember the "not really designed to be cross-platform bit")
- The "-r:" switches tell the system to include particular assemblies (libraries)
- System.Windows.Forms.dll not currently working. Researching...

Running Mono programs

- To run:

```
mono myfile.exe
```

Responding to Events in .NET

- Rather than listeners, C#/.NET uses *delegates*
- Delegates an elegant form of broadcasting/subscribing to events

Delegates

- Three components:
  1. Definition of a delegate type
  2. Declaration of a delegate instance
  3. One or more methods assigned to the delegate

1. Definition of delegate type defines a *method signature*
2. Delegate instance maintains a list of references to methods with that method signature
3. Delegate instance can then be invoked to call those methods

Delegates Example

```
using System;
using System.IO;

public delegate void Logger(string s);

public class DelegateDemo
{ static StreamWriter LogFile;

  public static void FileLogger(string s)
      { LogFile.WriteLine("Error: " + s); }

  public static void StdErrLogger(string s)
      { System.Console.Error.WriteLine("Error: " + s); }

  public static void Main() {
      LogFile = new FileInfo("Log.txt").AppendText();
      Logger log = null;
      log += FileLogger;
      log += StdErrLogger;
      log += (s) => System.Console.WriteLine("Error: " + s);

      log("Oops!");
      log("Oh, no!");
      LogFile.Close();
  }
}
```

Delegates Example

- `(s) => System.Console.WriteLine(s);` is a lambda expression
- `d` will refer to FileLogger, StdErrLogger, and the lambda expression
- Will invoke *all* of the methods when called
- The result (if there is one) returned is the result of the last method added to the delegate
- Methods can be removed using the `-=` syntax

Events in .NET

- Events in .NET are an extension of delegates
- Declare an “event” instance instead of a “delegate” instance:

```
public event Logger d;
```

- “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 (e.g., “myObject.d = null”)

```
using System;
public delegate void Logger(string s);

public class MyClass {
    // add/remove "event" to prove the point
    public event Logger log = null;
}

public class OtherClass
{
   public static void StdErrLogger(string s)

  { System.Console.Error.WriteLine("Err:"+s); }
    public static void Main() {
        MyClass c = new MyClass();
        c.log += StdErrLogger;  // allowed
        c.log = StdErrLogger;   // not allowed
        c.log("Oops!");         // not allowed
    }
}
```

Example Use of Delegates

```
using System;
using System.Windows.Forms;

public class HelloWorld : Form
{
    private void HandleClick(object source, EventArgs args)
    {   System.Console.WriteLine("Got button click.");
    }

    public HelloWorld()
    {   Button b = new Button();
        b.Text = "Click Me!";
        b.Click += HandleClick;
        this.Controls.Add(b);
    }

    public static void Main()
    {   Application.Run(new HelloWorld());
    }
}
```

University of Waterloo

Standard .NET Event Pattern

- When creating your own custom components, you define custom events thusly:

1. Define event data to be passed to event handlers by sub-classing System.EventArgs
2. Define public event instance using generic EventHandler class, with EventArgs class defined in step 1 above
3. Create protected virtual method for firing event, prefixed with "On" (e.g., "OnAreaExposed")

1. Define Event Data Class

Sub-class System.EventArgs to define event data:

```
public class BoundsEventArgs : System.EventArgs
{
    public readonly double x;
    public readonly double y;
    public readonly double width;
    public readonly double height;

    public BoundsEventArgs(double x, double y,
                           double width, double height)
    {
        this.x = x;
        this.y = x;
        this.width = width;
        this.height = height;
    }
}
```

2. Define public event instance

- Define event instance using the generic defined in System

```
public delegate void EventHandler<TEventArgs> (object
source, TEventArgs e) where TEventArgs : EventArgs;
```

(“object” refers to *who* is broadcasting the event)

- So your event instance would be:

```
public event EventHandler<BoundsEventArgs> AreaExposed;
```

3. Create protected Method to Fire Event

In your class:

```
public class MyClass :
{

  public event EventHandler<BoundsEventArgs> AreaExposed;

  // By convention, “On” + name of event (AreaExposed)
  protected virtual void OnAreaExposed(BoundsEventArgs args)
  {
     if (AreaExposed != null) AreaExposed(this, args);
  }

}
```

Event Queues Revisited

- Java uses listeners to inform components
  - Does it still have an event loop?

Java Event Queue

- Available from java.awt.Toolkit:
  - `Toolkit.getDefaultToolkit().getSystemEventQueue()`
- java.awt.EventQueue
  - 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

Awareness Systems

- Latching into event queue allows applications which are more “aware”: Can change behavior based on degree of activity in the interface
- Provides more nuanced types of interaction
- Some examples of this?

University of Waterloo

Awareness Systems

- IM clients, screensavers can make use of raw event queue by monitoring “activity”
- When activity drops, can do something
  - IM client: Set state to “away”
  - Screensaver: Start screensaver
- Issue: Windows allows *any* application access to global event queue through windows hooks
  - Implications with this?

Security

- Open access to global event queue is an enormous security risk
- Enables keyboard loggers, without user's awareness
  - Trivial to implement

Event Queues and New Input

- Event queue most suited for discrete, low frequency events
  - Breaks down for rich sensor input of high frequency or high bandwidth
  - Example: Pen input

Reference for Java Events

- Great reference on the rationale behind, and design of, Java's event model
  - http://java.sun.com/j2se/1.3/docs/guide/awt/designspec/events.html

Course Roadmap

- How events get delivered to application
- How application delivers events to components
- How components receive and act on events
- Next up:
  - How to design *usable* components (user’s perspective)
  - How to design components to promote reusability and extensibility