

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
 - 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          class MyButton
{ int x, y;           extends Button
  String label;      { LSystem lsys;
  boolean enabled;   ...
  ...
  void mouseDown(...) { if (lsys.started)
  {}                 { lsys.pause();
                    super.label =
                    "Start";
                    } else
                    { lsys.start();
                    super.label =
                    "Pause";
                    }
                    }
  void mouseUp(...)
  {}
  void keyPressed(...)
  {}
}
```



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();
    }
  }
}
```



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



Why is it not as desirable? Because the event handling becomes interwoven with the classes' public interface. Instead, want to contain the event handling in a logically clumped unit

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

- .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 and recently taken over by Xamarin
 - 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
- Now at version 2.10

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
- `log` 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()); }
}
```



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 behaviour based on degree of activity in the interface
- Provides more nuanced types of interaction
- Some examples of this?

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 best 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:
 - GUI toolkits