Event-Driven Programming

What is event-driven programming?

- progam structured in terms of *events*
- procedural programming uses top-down flow of control
  - starts from main and proceeds step by step
- event-driven programming, code fragments are associated with events, and invoked when the events occur
  - decouples order of execution
  - don't have to deal with the order of events, which helps a lot when the order is unknown

Example

```
// in MouseSpy.java
public class MouseSpy implements MouseListener {

   public void mouseClicked( MouseEvent event ) {
      System.out.println( “Mouse clicked. x = “
         + event.getX() + “ y = “ + event.getY() );
   }
   // implementations omitted for brevity
   public void mousePressed( MouseEvent event ) { ... }
   public void mouseReleased( MouseEvent event ) { ... }

   public void mouseEntered( MouseEvent event ) {
      System.out.println( “Mouse entered. x = “
         + event.getX() + “ y = “ + event.getY() );
   }

   public void mouseExited( MouseEvent event ) {
      System.out.println( “Mouse exited. x = “
         + event.getX() + “ y = “ + event.getY() );
   }
}

// in MouseSpyApplet.java
public class MouseSpyApplet extends Applet {
   public MouseSpyApplet() {
      MouseSpy listener = new MouseSpy;
      addMouseListener( listener );
   }
}

// in MyAppletPage.html
<applet code=”MouseSpyApplet.class” width=”400” height=”300”>
   If you are reading this, you need a Java capable browser!
</applet>
```

Applications of Event-Driven Programming

- GUI
  - modern GUI environments are event-driven
  - events occur when the user does something:
    - move the mouse
    - click
    - press a key
    - minimize/maximize windows
    - ...
- event driven I/O
  - especially networking, where I/O is very slow
  - events occur:
    - when connections are made
    - when ready to send
    - when data arrives
- Lego Mindstorms :-)

I/O Comparison

```
// procedural I/O
main {
   Socket s = new Socket()
   while ( s.isConnected() ) {
      s.recv( recvBuffer )
      do something with recv'd data
      put some more data in the send buffer
      s.send( sendBuffer )
   }
}

// event driven I/O
class SocketListener {
   onRecv( Socket s ) {
      s.recv( recvBuffer )
      do something with recv'd data
   }

   onSend( Socket s ) {
      put some more data in the send buffer
      s.send( sendBuffer )
   }

   onDisconnect() {
      exit program
   }
}

main {
   Socket s = new Socket()
   SocketListener sl = new SocketListener()
   registerSocketListener( sl, s )

   runEventLoop()
}
```

Participants in an OO Event Driven System

- events
  - represent occurences, changes of state, or requests which a program might need to handle
  - store all event-instance specific information
  - e.g. MouseEvent, stores x,y coords.
- handlers
  - objects that respond to events
  - register to be notified
  - handlers are usually stateless
  - e.g. MouseSpy
- event sources
  - objects that generate events
  - handlers specify which event source they want to use

Design of event driven programs

- application layer
  - core classes that provide underlying functionality
  - independent of interface and event handling
- translation layer
  - event handlers
  - interact with the application layer and presentation layer
  - typically getting and setting properties in each layer so that the presentation reflects the current application state
- presentation layer
  - the visual appearance of the app
  - buttons, frames, menus, etc.

Java considerations: inner classes

- event handlers are usually implemented as inner classes
- convenient to have direct access to application class
- results in a strong coupling between application and translation layers (bad)
- less bad than making application layer classes event handlers, inner classes can evolve into standalone classes

Java considerations: event adapters

- many event handler (listener) interfaces include methods that you won't need (e.g. MouseListener)
- event adapters are abstract base classes which provide empty implementations of the listener interface methods, so that you don't have to.
- can lead to confusing errors when you don't override properly, because compiler won't catch the error

Example of an event-driven program

```
//: c13:TrackEvent.java
// From 'Thinking in Java, 2nd ed.' by Bruce Eckel
// www.BruceEckel.com. See copyright notice in CopyRight.txt.
// Modified for CSC326 by Neil Gower.
// Show events as they happen.
// <applet code=TrackEvent
//  width=700 height=500></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class TrackEvent extends JApplet {
  HashMap h = new HashMap();
  String[] event = {
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited","mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
  };
  MyButton
    b1 = new MyButton(Color.blue, "test1"),
    b2 = new MyButton(Color.red, "test2");
  class MyButton extends JButton {
    void report(String field, String msg) {
      ((JTextField)h.get(field)).setText(msg);
    }
```

```
FocusListener fl = new FocusListener() {
  public void focusGained(FocusEvent e) {
    report("focusGained", e.paramString());
  }
  public void focusLost(FocusEvent e) {
    report("focusLost", e.paramString());
  }
};
KeyListener kl = new KeyListener() {
  public void keyPressed(KeyEvent e) {
    report("keyPressed", e.paramString());
  }
  public void keyReleased(KeyEvent e) {
    report("keyReleased", e.paramString());
  }
  public void keyTyped(KeyEvent e) {
    report("keyTyped", e.paramString());
  }
};
MouseListener ml = new MouseAdapter() {
  public void mouseClicked(MouseEvent e) {
    report("mouseClicked", e.paramString());
  }
  public void mouseEntered(MouseEvent e) {
    report("mouseEntered", e.paramString());
  }
  public void mouseExited(MouseEvent e) {
    report("mouseExited", e.paramString());
  }
  // have not overridden mousePressed() and mouseRelease(), because
  // we are not interested in those events
};
```

```
    MouseMotionListener mml =
      new MouseMotionListener() {
      public void mouseDragged(MouseEvent e) {
        report("mouseDragged", e.paramString());
      }
      public void mouseMoved(MouseEvent e) {
        report("mouseMoved", e.paramString());
      }
    };
    // constructor
    public MyButton(Color color, String label) {
      super(label);
      setBackground(color);
      addFocusListener(fl);
      addKeyListener(kl);
      addMouseListener(ml);
      addMouseMotionListener(mml);
    }
  } // end of class MyButton
  public void init() {
    Container c = getContentPane();
    c.setLayout(new GridLayout(event.length+1,2));
    for(int i = 0; i < event.length; i++) {
      JTextField t = new JTextField();
      t.setEditable(false);
      c.add(new JLabel(event[i], JLabel.RIGHT));
      c.add(t);
      h.put(event[i], t);
    }
    c.add(b1);
    c.add(b2);
  }
  public static void main(String[] args) {
    Console.run(new TrackEvent(), 700, 500);
  }
} ///:~
```

Implementing an event-driven system

- for many applications, you can just use the OS's event system, or an existing framework (like Swing or AWT in Java)
- to appreciate what's going on, we'll look at how such a framework is constructed
- key steps
  - find out who will handle the events
  - watch for events
  - call the handlers when they occur

The event queue

- as events occur, they are placed in a queue to be dispatched by the event loop
- noticing events may be part of the event loop, or tied into a lower level event-driven system
- e.g. select() is a method that blocks until one of several file handles is ready for reading

The event loop

- the main thread(s) of the program
- when events appear in the queue, the event loop dispatches them
  - dequeue next event
  - look up handler(s) for that event
  - call handler's event handling method
- this loop must iterate quickly enough for events to appear to be handled instantaneously

Caution!

- event handling methods must be short, otherwise the app becomes unresponsive
- a multi-threaded event dispatcher can sometime reduce the severity this problem (at the cost of adding the complexity of concurrent programming)
- general solution is to not to block in the handler
  - create a new thread to do time consuming processing
  - and/or trigger an event when processing is done
  - this can make event-driven code harder to understand

Registering handlers

- need to provide an interface for handlers to declare that they want to be notified of a type of event
- define how handlers:
  - specify what kinds of events they are interested in
  - specify the source of the events
  - specify the object and method to call when the event occurs
- store references to the handlers, associating them with the event type they are interested in, and the event source
- typically, this is done via an addXXXHandler() method

Deciding on the handler interface

- events types can be distinguished by subclass
- handlers can implement an interface, or even supply a java.lang.reflect.Method
- handlers are passed an event object, which contains *all* of the information about that event
- handler methods do not typically return values, because the event dispatcher is not interested in application specific information