Events

Most Java programs …

- Most Java programs incorporate a graphical user interface;
  the user controls the program via this interface.
- This leads to a different style of program:

an "event driven" program

A *long* time ago,

- Program structures were simple:

```
ScienceProgram
    Initialize
    Fill array with numbers
    Compute transform of array
    Calculate properties of
        transformed array
    Print computed properties
```

```
BusinessProgram
    Initialize
    repeat
        read next record
        process record
        print record info
        update totals
    until end of file
    print totals
```

Input-compute-output

- In the days when compute power was limited, "user-friendliness" meant getting the results back in two days rather than two weeks.
- Programs ran in "batch mode" reading previously prepared data sets.
  - No interaction was possible.
  - The data had to be in a predefined sequence.
- The flow of control was fixed.
- Data were generally in the form of simple records or arrays.
- Programming was in many respects simpler.

MenuSelect

- By the 1970s, timeshare systems were common.
- Users worked interactively at terminals.
- The archetypical program now was:

```
MenuSelect
  Display menu showing choice of processing options
  prompt user to enter identifier of required option
  switch on entered id
case 1      DoProcessingOptionOne() break
case 2      DoProcessingOptionTwo() break
     …
```

MenuSelect

- Still overly computer- or program-centric.
- The program is "in charge"
  - it prompts, the user responds.
  - Once an option has been selected, the user must complete the actions required, providing data in a prescribed order.
- Having "the program in charge" simplifies coding.
  - The order of processing is defined – "get first input, do this, get next two inputs, now do this, get fourth input, now process, print".
  - There is only one logical path through such a program.
  - The programmer knows that all necessary data will have been entered prior to the processing step.

Program-centric versus User-centric

- Inflexible structures may suit computers, they don't suit users. Users:
  - make mistakes
  - want to change their mind
  - want to skip bits of the processing
  - generally behave like erratic, emotional, irrational *humans*.
- If computer resources are limited, users have to accept the rules.
- For a long time, computer resources were limited; even in the 1970s, most software development still focussed on "efficient use of the computer".

Xerox-Parc

- There was an exception to the predominant Computer-centric view.
- Researchers in Xerox's extrapolated the graphs of costs and predicted that by the 1980s computers would be cheap enough to go on the desks of secretaries.
- Xerox's business was "office automation" so naturally it wanted to know how these ubiquitous computers would be employed.
- How should these machines be used so as to maximize "*efficient work by people as aided by their personal computers*"?

Focussing on the user (+ PC) ..

- Xerox's group included psychologists, graphics designers, managerial experts as well as programmers.
- Together they developed a variety of new concepts and tools.
- Although never effectively exploited commercially by Xerox, their work forms the basis for virtually all modern computing.

Xerox contributions

- Smalltalk programming language
  - created to achieve main goal
- Main goal:

  make computer power accessible to anyone who needs it
- Increased accessibility required
  - reworking the interface between user and the running program
  - exchanging their roles in regard to control.

Graphic user interface

- Users must be able to
  - "see" their data
  - "see" what operations can be done on the data
  - select operations as they wished,
  - complete subtasks in whatever order they found convenient,
  - "see" the effects of their operations through changes to the displayed data.
- Need high bandwidth graphics
  - You must draw a picture of the data elements.
- You must provide a means whereby the user can "*pick a data object and give it a command*".

"Pick an object, give it a command"

- "Pick an object, give it a command" – it was a kind of leitmotif that ran through all the work at Xerox.
- *in the programs ran on graphic displays:*
  - data objects were represented by iconic images
  - you picked an object using a cursor (controlled by mouse or joystick) and pressing a select button on the pointer control
  - you selected a command from a visual menu
- *in the programming language Smalltalk:*
  - everything (even numbers, even bits of code) was an "object"
  - a statement in Smalltalk identified an object and then specified the command the object was to obey

A prototype standard “GUI” interface

- The Xerox experiments also lead to much of the now standard GUI components:
  - screen area divided into logical windows
  - windows with standard structure – frame border, resizing element, iconic/full state selection, movable
  - menu
    - “pop up” or “pull down” when mouse button activated in appropriate way
  - scrollers and scrollbars
    - a window will have a fixed maximum area,
    - the data that are to be displayed may require a much larger area
    - so controls that “move the window” over the data
  - “action buttons”, “checkboxes”, “radio button clusters”,

GUIs

- Xerox’s model for effective human computer interaction was popularized:
  - first through the Macintosh
  - then through Windows and other systems like various "X" based graphics user interfaces developed for Unix.
- The Xerox project also resulted in a new programming model becoming the standard.
- This new model is necessary to support the ubiquitous "user friendly graphical user interface".

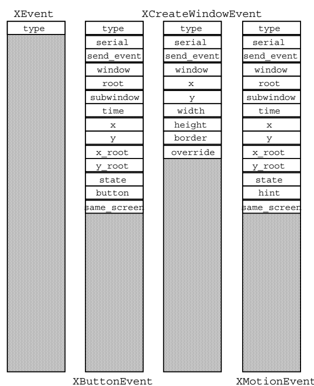
Main event loop

- The standard program of today involves code something like the following:

```
Run
  repeat
     Wait for arrival of an "event" that represents
          a command entered by the user
     decode the event, identifying the command and
          the (program) object that is to obey
          the command
     tell object to obey the command
  until the "application object" receives
               a "Quit" command
```

Events

- "events"
  - start as simple record structures (C structs)
  - created by a combination of code in the operating system and the executing program
    - operating system responds to an interrupt caused by something like the user clicking a mouse button or keying a character.
    - operating system passes such details on to a run-time support routine linked into the application.
    - This routine fills in an event data structure
      - "Left-button down event, time is ..., cursor's screen coordinates are ..., ..."
      - "Key-press event, key is Q, modifier keys Command and Shift, time is ...".
  - may get processed immediately, usually queued up for later processing.



Events

- Next step involves code that is standardized for any particular platform.
  - That mouse button click, where did it occur?
    - If in the menu-bar, then the corresponding menu-bar object should display the complete menu.
    - If in the frame of partially obscured window, then that window should be moved to the front of the stacking order so that it becomes completely visible.
    - If the mouse click was in an action button, the user is entering a command for the corresponding action to be performed.
- Code involves
  - working out which window, subwindow or other graphic element was activated and hence identifying the appropriate object.
  - using other information in the event structure to determine the command that this object is to obey.

Main event loop? Well, OK;
but how does the program work?

- Programs are still about manipulating data, using coded up algorithms to convert inputs to outputs.
- You may still get "input-compute-output"
  - enter some data
  - invoke a chosen processing function
  - (wait)
  - results created in memory
  - picture of results drawn on screen
- But the "compute" step more likely to involve host of small processing phases, some optional.
- Partial results may be interesting and may need to be displayed.

Incremental processing

- Can enter all data and then process, but more typically:
  - enter a little data
  - partially process to check things going right
  - edit and change previous data that now appear incorrect and add some more
  - partially process to check things going right
  - edit and change ...
- Examples:
  - you coding your C++ assignment on an Integrated development environment (edit, compile, edit, compile, ...)
  - you entering text in word processor, invoking spelling checks or print preview options
  - chemist entering a structure, invoking model builder to check sterochemistry,
  - …

Idling CPU

- In olden times, efficient user of CPU determined design; now it's efficient work by user.
- CPU often idling
  - pick up command
  - invoke processing function
  - redraw display
  - wait for next command

  - time spent processing and redrawing much less than time spent sitting waiting for user to enter next command

Where is that main event loop?

- If you were to write code for Macintosh at Toolbox level, *you* would have to write code for the main event loop.
- On Windows, and with the X- toolkits based on Xt- (Motif, OLIT, Athena), the main event loop itself is in code provided by the library
  - instead of coding the actual switch statements that call your routines, you "register" your routines (associating them with corresponding user interface elements)
  - system code will make "call backs" to your routine when the user interface element is involved in an event
- In Java? Well, it is partly in virtual machine itself , but mostly in classes from graphics components (plus some in the Runtime and System objects).

Handling events

- Mouse clicks –
  - location on screen matched to areas representing different controls, menus, or drawings of parts of data
  - meaning
    - selection of "target"
    - selection of operation to apply to target
- Keystrokes –
  - meaning
    - more data to be collected by "something" that is building a textual data structure
    - or, if "control key" combination, selection of operation to apply to target

Handling events

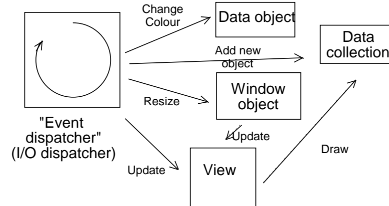
- Keystrokes and mouse clicks represent user input and are the primary events.
- Can get others. Some are obvious extensions of keystroke and mouse click e.g. messages over network defining actions by another user with a collaborating program.
- Other more subtle
  - "activate" event
    - follows user action that changes primary window
  - "update" event
    - essentially a message telling a View to arrange to redraw itself
- Again, need to identify "target" data structure involved in such an event.

Maintaining “target” data structures ...

- *Can* be done in essentially ad hoc manner
  - global pointer to “current text buffer”
  - array of global pointers to “window structures” – defining their areas so can identify which contains cursor on mouse click
  - global integers representing data like position of scrollable panes, amounts of travel in scrollbars
  - …
- *Can be done that way.*
- Was done that way in early Macintosh and Windows programs.
- An ad hoc way is not a good way!

Xerox’s model for program structure

- Xerox provided a way of structuring these event handling programs.
  - Smalltalk
  - using objects



I/O dispatcher command handlers

- Xerox’s group invented a systematic OO approach.
- The “main event loop” would be the “run” function of some i/o dispatcher object
  - pick up event
  - decode
  - turn into a “message” or “command” object
  - send to appropriate handler
- The visual display structure would be built from some objects that were instances of “standard” window and view classes.
- The data manipulated by the program would consist of objects held by some principal collection object.

Model-View-Control

- Xerox formalized their approach in one of the earliest “*design patterns*”.
- Model
  - the collection of objects that represent the data that the program is manipulating
    - principle object holds one or more collections of other data
    - collection objects (standard lists, dynamic arrays etc)
    - application specific data
- View
  - display areas,
    - area of screen (window), “graphics context”, coordinate system
    - link to data in model
- Control
  - i/o dispatcher and assistants

Design Patterns

- Learning about “Design Patterns” is an important aspect of your education in programming, but one that receives too little attention in our current subjects.
- Issue of reuse
  - functions
  - simple concrete classes
  - design patterns
  - frameworks

Design patterns

- *Proven solutions to frequently occurring problems*
- *Solutions that involve*
  - *instances of several classes*
  - *instances that work together in regularized manner*
  - *code defined for these interactions is widely applicable*
- *Often can use inheritance to specialize for particular application*

Design Patterns

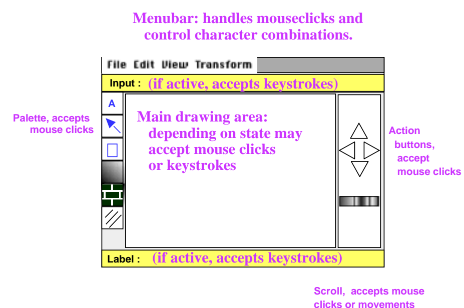
- Read
  - E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns", Addison-Wesley, 1995.
  - F. Buschman and **lots of others!**, "Pattern Oriented Software Architecture – A System of Patterns", Prentice-Hall, 1996.

Strategies for handling events

- "Chain of command" pattern
  - as in most of C++ framework class libraries and Java 1.02
- "Dependency" (observer) pattern
  - inspiration for Java 1.1 mechanism

Chain of command

- We focus initially on user initiated commands as in some standard GUI program
  - event handling will start with keystroke or click of mouse button
- Our task:
  - from data in event, we must work out which program object should be involved and what it should do



Sorting out events

- Low level event structure:
  - mouse
    - coords
    - time
    - maybe things like control, shift key held down
  - keyboard
    - character, with modifiers
- Mouse events slightly easier, coords often sufficient to identify appropriate handler object

Mouse events

- OS may know about subwindows (this is case with X-)
- Event is tagged with id of subwindow
- Each control, menu-bar, view is separate window.
- If on something like Mac where really only one window, code in application has to find smallest enclosing 'view'.

Sorting out events

- Click in menu bar?
  - MenuBarObject, HandleMouse
    - if mouse click in menu[i] title,
      menu[i] HandleMouse
- Click in main drawing area?
  - If drawing tool selected, create new drawing object; drawing object track mouse.
- Click on tool palette?
  - Change currently selected tool.

Sorting out events

- Palette and main drawing area are naturally application specific views that expect to handle mouse clicks in some programmer defined manner.
- What about clicks on action buttons?

Subclass the buttons?

- Could try this:

```
class MyNorthButton  extends Button {
   public MyNorthButton(Data linkeddatastruct)
       { fData = linkeddatastruct; }
   public void OnClick() {
         fData.GoNorth();
         }
}
class MyEastButton extends Button {  }
class MySouthButton extends Button { }
```

- could! But it really is a bit messy.

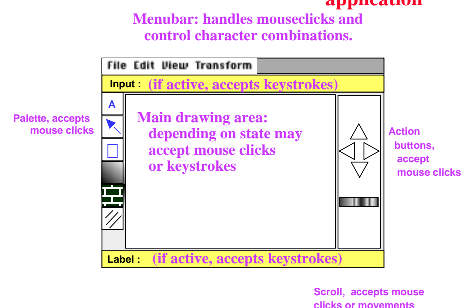
Centralising event handling

- A solution involving numerous special subclasses of controls was never popular.
  - design just didn't seem particularly coherent;
  - control too distributed;
  - maintenance and extension difficult;
    - think of hassles if can change the principal data structure by loading a file ...
- General feeling was should centralise event handling.

Centralising event handling

- Approach
  - button doesn't do anything with click apart from reporting to "next handler" that it has been clicked
  - for a GUI element like a button, "next handler" would be enclosing view or window

Buttons enclosed in 'pane', enclosed in window, owned by document, created by application



Centralising event handling

- Approach
  - so button's "*I've been clicked*" message goes to pane
  - pane doesn't know what to do, so it passes message on to enclosing window
  - window doesn't know what to do, so passes message to document
  - document has code

    if north button pressed, data do GoNorth else ...

Not just the mouse clicks on controls ...

- The same "chain of command" (buck passing?) is used to deal with menu requests.
- When a menu is activated, the menubar object creates a "menu_event" and gives it to the "target" object (the one at the head of the command chain).
- A menu_event gets passed up the chain until it reaches the object that handles that request.

Not just the mouse clicks on controls ...

- Keystrokes are handled similarly (except those with control characters which go straight to the menubar object)
- An event "*the user typed k*" is passed to the target object, if this is not interested in handling keystrokes it passes the event up the command handler chain.

Target object

- Mouse clicks on controls automatically define a target (the object that gets first go at handling an event).
- The target for the command chain must be set explicitly by the program
  - when mouse is clicked in (or maybe moves over) a text field, that field becomes the target ("acquires keyboard focus")

Command chain

- The organisation of the chain of handler objects is largely automatic
  - each GUI element will identify an enclosing GUI element as its next handler;
  - top-level GUI element, 'framewindow', will identify some application or document object as next handler
- Command chain approach widely utilized

Command chain

- The command chain must end at some overall "boss" class (*the buck stops here!*).
- In C++ frameworks (OWL, MFC etc) there will be an Application object and a Document object - the Document represents the end of the chain for most events.
- In Java1.02 applets, an instance of an Applet class will be end of the chain; Java1.02 applications using GUIs will typically have a Frame class at the end of the chain.

Java 1.02
command chain

- In Java, most of the objects in the command chain will be instances of subclasses of component
- Component defined a "handleEvent" method - this identified the event type
  - Action event from a typical control,
  - List event picking from list,
  - Window events,
  - Mouse movement events,
  - key press event
  - mouse down events
  - handleEvent() dispatched these to auxiliary functions (but could be replaced in special cases)

Java 1.02
Event handling

- Component also defined
  - action() for action events
  - mouseDown(), mouseDrag(), mouseEnter() ...
  - keyDown()
  - gotFocus(), lostFocus()
  - ...
- These returned boolean; true if event had been handled; false meant event should be passed up the control handler chain.

Java 1.02
Event handling

- In Java 1.02, the way you would handle those buttons from earlier example would be:
  - your Frame or Applet would define action()
    - action(Event evt, Object what)
      - from arguments you would find out which button had initiated event, hence would invoke appropiate method of data object

Command chain

- Command chain well established approach.
- But
  - somewhat inefficient
    - quite a lot of processing done at each stage as an event is passed up chain until it reaches object that deals with the event
  - restricting?
    - event handlers must be "components"?

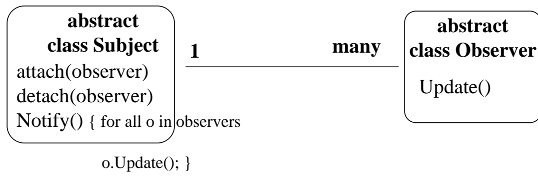
Java 1.1

- When revising initial release of Java, Sun's designers decided to look for ***alternative*** to command chain.
- Another well known "design pattern" is "dependency" ('observer')
  - if an object is changed, it may have to notify others
  - "dependency" pattern represent a standardized approach to this situation

Dependency pattern (aka Observer)

- *Intent*
  - define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and update automatically.
- *"classic example"*
  - Value cell in spread sheet
    - dependents include cells related via formulae
    - views (spreadsheet view, pi-chart view, ...)
    - document (needs to know data changed)

structure for pattern



In real program, have instance of some class that implements "Subject" interface; e.g. a spreadsheet cell.
Each cell related to it by a formula registers as an observer.
Spreadsheet document also registers as observer.
When cell value changed; all registered observers are notified.
Observers can ask for new value while doing their updates.

advantages

- subject and observer(s) loosely coupled
- subject does not need to know anything of observers (except that they can be told to update themselves)

implementation

- Can be trivial:
  - each subject object has a List<observers>
  - really do say

```
for each observer o in list do
  o.Update(...)
```

- Many more sophisticated implementation strategies known (typically involve a change manager object that looks after dependencies and optimizes update sequences)

Adapted to suit Java 1.1

- When an object experiences something which would have lead to creation of an event, now send a notification to "listeners" (dependents/observers) that have registered an interest.
- Not quite the simple Subject-Observer configuration.
- *Java defines an entire type hierarchy.*

ActionEvents

- ActionEvents -
- Generated by Button, List, MenuItem, TextField objects.
- These classes have a method
  `addActionListener(ActionListener l)`
- ActionListener --- interface
  `actionPerformed(ActionEvent e)`

ActionListener is a kind of "observer"

ActionEvents

- ActionEvent object passed to ActionListener
  - can return name associated with action command (eg button name)
  - can return details of any modifier keys
  - ...

AdjustmentEvents

- AdjustmentEvents - user moving a scroll setting
- Generated by Adjustable objects (only standard one is Scrollbar).
- These classes have a method
  `addAdjustmentListener(AdjustmentListener l)`
- AdjustmentListener --- interface
  `adjustmentValueChanged(AdjustmentEvent e)`

AdjustmentListener is a kind of "observer"

AdjustmentEvents

- AdjustmentEvent
  - getValue *what is the current scroll setting?*
  - getAdjustable *returns reference to (scroll bar object) that was changed*
  - getAdjustmentType *UNIT_INCREMENT, ...*

ItemEvents

- ItemEvents - user making a choice?
- Generated by Checkbox, Choice, List, ... objects.
- These classes have a method
  `addItemListener(ItemListener l)`
- ActionListener --- interface
  `itemStateChanged(ItemEvent e)`

Mouse(Motion)Events

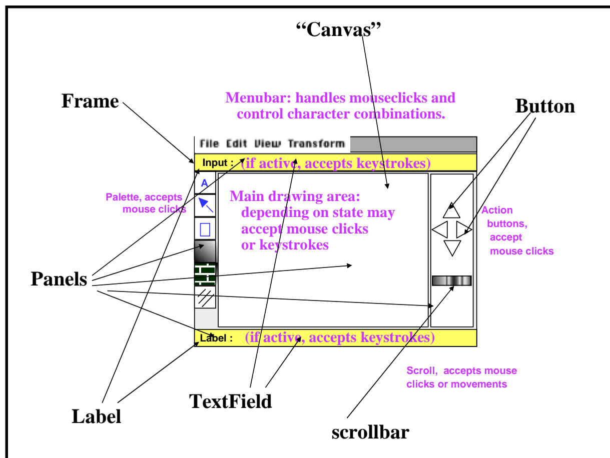
- MouseEvents - user drawing or selecting?
- Generated by any Component .
- Component has method
  ```
  addMouseMotionListener(
              MouseMotionListener l)
  ```
- MouseMotionListener --- interface
  ```
  mouseDragged(MouseEvent e)
  mouseMoved(MouseEvent e)
  ```

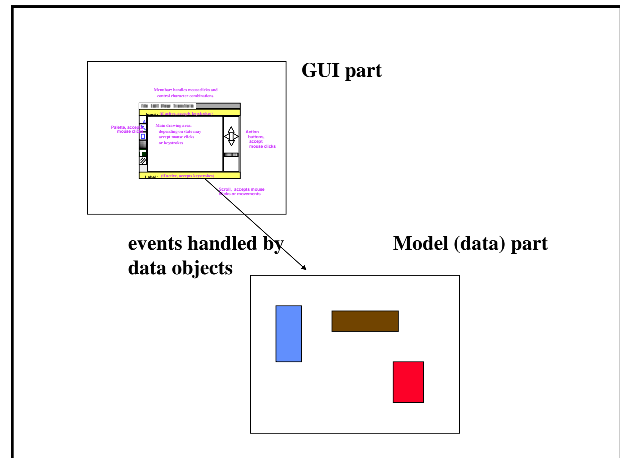
Working with Java 1.1+ (GUI) events

- Sketch your user interface, identifying controls and other interactive elements and noting the types of "Event" they may generate.
- Decide on a "model" (group of objects that will represent your data)
- Decide which data objects will be affected by activation of particular controls.

Working with Java 1.1+ (GUI) events

- Make certain that classes (whose instance will handle specific kinds of event) do implement appropriate Listener interface(s).
- Sort out how to connect GUI components to the objects that will handle the events that they generate
  - these handler objects must be registered as "listeners" for the GUI elements.





Java 1.1+ style

- Objects (some of them)
  - GUI: Frame (or Applet), Panels (for palette, and holding controls), specialized Canvas subclasses (for palette & main drawing area), standard Buttons, and Scrollbar
  - Data: PaletteObject and DataObject (collection of individual data items)
- PaletteObject would be a *MouseListener*, and would be "observing" the Canvas used to display tool palette - when click on this Canvas, PaletteObject immediately notified

Java 1.1 style

- Principal data object would
  - be a MouseListener, MouseMotionListener, ActionListener, KeyListener!
- It would "observe"
  - the control buttons (as an ActionListener)
  - its Canvas (as MouseListener, MouseMotionListener etc)

Java 1.1 style

- If motion buttons clicked, data object gets notified.
- It updates coordinates of its constituent data items.
- It then tells its Canvas to "repaint()"; this will result in a call to "update()" being scheduled shortly.

(note on update(), paint() etc)

- All the frameworks (C++ things like OWL, ET++, MFC; Java; Delphi) have a similar model of how to handle graphics output.
- When a data structure is changed, it does not immediately try to redraw its representation in affected "views".
- Instead, it tells the views that they are "invalid" (Java's repaint() call).

(note on update(), paint() etc)

- Invalidation of a view results in the framework code (associated with the main event loop) scheduling an "update event".
- It is the "update event" that results in the new display
  - "update event" reaches view
  - view clears itself
  - view sets coordinate frameworks etc
  - view tells data structure to draw itself (passing a graphics context if needed)

(note on update(), paint() etc)

- Mechanism allows a framework to perform some optimisations; often several "invalidate" requests can be dealt with by single update.
- Java employs this standard approach
  - change data object
  - data object tells any related display Component to repaint() (can specify area affected, allows further optimisations)
  - repaint() results in framework code scheduling update
  - Component gets update() call, this calls paint()
  - Component.paint() should get data object to draw

"update events", "activate events"

- Most events received by GUI style programs represent explicit user actions like mouse clicks and key presses.
- But also have "standard" things like:
  - "update events" - created by framework when view invalid
  - "activate events" - created by framework when a window changes state (opened, iconified, reshown, partially hidden, reexposed, closed)

Java's WindowEvents

- Java has its own equivalents of the "standard" update and activate events.
  - its updates are essentially internal, they are not even defined as an explicit event class
  - activates are just one aspect of Java's more general WindowEvent

Java's WindowEvents

- Need some form of WindowListener - an object that dealt with the WindowEvent that is generated when the user closes a window
  - a WindowListener object can simply call System.exit(...) to terminate a program
  - *WindowListener is an "interface"*, we can implement it in any class we want; so it is largely our choice as to which object actually deals with closing a window

Java event handling example

Simple example

- Program - illustrates events by updating and then redrawing a representation of a data structure.
- Data structure
  - a set of rectangles
- Display
  - a canvas where a diagram of the data is drawn
  - a set of four button controls
  - a framewindow

Swing or AWT?

- *There are two main GUI libraries for Java:*
  - *abstract windows toolkit*
    - *"peer" implementations on Mac, PC, Xwindows*
  - *swing*
    - *(built on top of awt)*
    - *different implementation - no platform specific peer classes, but its implementation classes can mimic display styles of different platforms*

Swing or AWT?

- *Swing has*
  - *classes that behave similarly to, or identically to AWT classes (JFrame for Frame, JApplet for Applet, JButton for Button, …)*
  - *lots of additional classes for more elaborate displays (styled text, display of tree structures etc)*

Swing or AWT?

- *Swing advantages?*
  - *Many users think it looks better*
  - *Extra classes very useful in more advanced applications*
- *Swing disadvantages?*
  - *Does seem slower, particularly at start up*
  - *Many Browsers have Java implementations that provide only AWT*
  - *and threading complications!*

Swing or AWT?

- *Our examples:*
  - *Mix: some AWT, some swing*
- *Your choice for now*
  - *Whatever the assignment specifies!*
- *Your choice in future?*
  - *More likely to be swing than awt (is richer, you will learn how to handle the problems related to threading issues)*

Swing or AWT?

- *No real problems relating to use of one library or the other*
  - *for AWT Button:*
    - *import java.awt.*; import java.awt.event.*;*
    - *create button as an instance of class Button*
  - *for Swing JButton:*
    - *import the same awt files, also import javax.swing.*;*
    - *create button as an instance of class JButton*

Swing or AWT?

- *In some cases, functionality is different*
  - *example "text areas" (display of blocks of optionally editable text)*
    - *AWT provides a scrollable text area*
    - *Swing's text area does not in itself supply scrolling*
      - *so create a JScrollPane*
      - *create the JTextArea*
      - *put the JTextArea in the JScrollPane*
      - *continue as if had a scrollable text area*

Another swing issue …

- When awt built (~1995) the code used locking to prevent the structures that represent parts of a GUI from being changed by one thread while they were being drawn by another.
- Locks are slow and expensive (have to get down into C code in the program that runs the Java Virtual Machine)
- Swing libraries tried to reduce locking – but that leaves code 'thread unsafe'

swing and threads

- swing interface objects should only be used by the GUI thread (single threaded)
- Usually OK
  - build the interface by putting all the parts together
  - "show" it, starting GUI thread that then runs
- But …

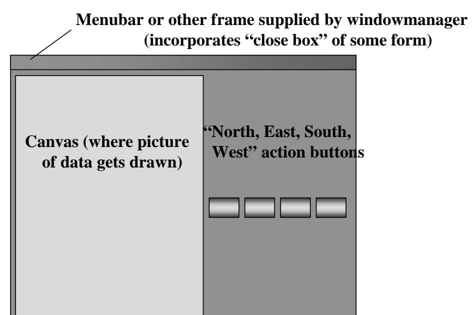
swing and threads

- Sometimes there can be problems
  - Most likely to occur if program wants to change interface by adding or removing GUI components.
  - Program could mess up screen or even crash
- Complex mechanism exists for getting GUI thread to do any changes to GUI components.
- You have been warned –
  - you won't have problems with simple things in ITCS213
  - you could have problems if you try to build complex dynamic swing interfaces without dealing with these threading issues

Swing or AWT?

- *What should you be using in future¶?*
- *Whatever best suits the immediate needs!*

Display sketch



Events from GUI

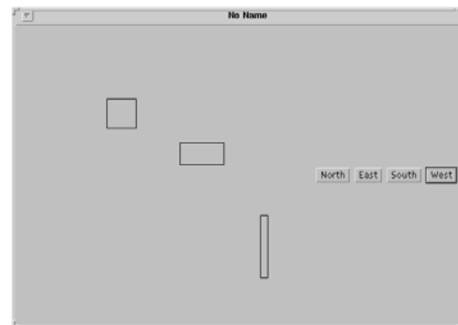
- Frame
  - will generate WindowEvents
    - must handle “window closing”, could possibly respond to others
- Buttons
  - will generate ActionEvents

Data objects

- Main DataObject
  - owns a collection (Vector) of rectangles
  - paints rectangles on a given Graphics object
  - should respond to ActionEvents used to move its rectangles
  - should “implement ActionListener”

Data objects

- Something to look after display interface
  - owns Canvas, Frame etc
  - responds to window event for closing window
  - implements WindowListener



classes

- Display:
  - Listens for window closing events
  - Owns some buttons etc
- DataObject
  - Registered as “listener” for those buttons
- MyCanvas
  - Forwards “draw” requests to data object
- EventDemo1
  - Provides the main( ) function

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
class DataObject implements ActionListener { }
class MyCanvas extends Canvas { ... }
class Display implements WindowListener { ...}
public class EventDemo1 {
  public static void main(String args[])
  {
   ...
  }
}
```

```
class DataObject implements ActionListener {
   public DataObject() { ... }
   public void LinkToCanvas(Canvas c) { fCanvas = c; }
   public void paint(Graphics g) { ... }
   public void actionPerformed(ActionEvent e) { ... }

   private Vector fCollection = new Vector();
   private int fX0;
   private int fY0;

   private Canvas fCanvas;

}
```

```
public DataObject() {
   Rectangle r1;
   r1 = new Rectangle(10, 20, 40, 40);
   fCollection.addElement(r1);
   r1 = new Rectangle(110,80, 60, 30);
   ...
   fCollection.addElement(r1);
}
public void paint(Graphics g) {
   Enumeration e = fCollection.elements();
   while(e.hasMoreElements()) {
        int x; int y;
        try {
                Rectangle r = (Rectangle) e.nextElement();
                x = fX0 + r.x; y = fY0 + r.y;
                g.drawRect(x, y, r.width, r.height);
                }
        catch (NoSuchElementException ns) { }
        }
}
```

```
public void actionPerformed(ActionEvent e)
{
   String s = e.getActionCommand();
   if(s.equals("North")) { fY0 -= 5; fCanvas.repaint(); }
   else
   if(s.equals("East")) { fX0 += 5; fCanvas.repaint(); }
   else
   if(s.equals("South")) { fY0 += 5; fCanvas.repaint(); }
   else
   if(s.equals("West")) { fX0 -= 5; fCanvas.repaint(); }

}
```

```
class MyCanvas extends Canvas {
   public MyCanvas(DataObject d)
    { fData = d; setSize(400, 400); }

   public void paint(Graphics g) { fData.paint(g); }

   private DataObject fData;
}
```

```
class Display implements WindowListener {
public void windowClosing(WindowEvent e)
              { System.exit(0); }
public void windowClosed(WindowEvent e) { }
public void windowIconified(WindowEvent e) { }
public void windowOpened(WindowEvent e) { }
public void windowDeiconified(WindowEvent e) { }
public void windowActivated(WindowEvent e) { }
public void windowDeactivated(WindowEvent e) { }

   public Display(DataObject d) { ... }
   public void show() { fFrame.show(); }

   private Frame fFrame;
   private MyCanvas fCanvas;

}
```

Java listener classes

- WindowListener has 7 different methods to handle different types of window event
- Could have had one public method
  - this would have tested and invoked appropriate processing functions as needed;
- Java library designers decided to make the different processing functions part of the interface

Java listener classes

- Classes like Display that implement WindowListener have to provide all seven methods.
- Note: adapter classes that can avoid need for all those empty functions (covered in detail in “Core Java” text, will discuss briefly later)

```
public Display(DataObject d)
{
        fFrame = new Frame();
        fFrame.setLayout(new FlowLayout());
        fCanvas = new MyCanvas(d);
        d.LinkToCanvas(fCanvas);
        fFrame.add(fCanvas);
        Button b;
        b = new Button("North");
        b.addActionListener(d);
        fFrame.add(b);

        b = new Button("East");
        b.addActionListener(d);
        fFrame.add(b);
        ...
```

```
public Display(DataObject d)
{
        fFrame = new Frame();

        ...

        b = new Button("South");
        b.addActionListener(d);
        fFrame.add(b);

        b = new Button("West");
        b.addActionListener(d);
        fFrame.add(b);

        fFrame.pack();
        fFrame.addWindowListener(this);
   }
```

```
public class EventDemo1 {


   public static void main(String args[])
   {
   DataObject fData;
   Display fDisplay;
        fData = new DataObject();
        fDisplay = new Display(fData);
        fDisplay.show();
   }
}
```

EventDemo1

- OK?
- ***Where is that event loop?***

Event loop

- EventDemo1.main() finished
- But because we’ve created display structures, control ends up in main event loop provided by Java system.
- It now “runs the program”.

Events and your program

- Makes coding slightly odd!
- *Your code doesn't appear to define overall flow of control.*
- You have a main() (or an Applet.init()) function that creates principal data structures and display
- You have a variety of functions scattered through your classes that will get called when specific events occur.

EventListeners

EventDemo1 as event handler example- 1

- Simplified, but fairly typical
- It defined a graphical interface
  - Some components are inherently sources of events
    - Main window (Frame) itself – things like closing event
    - Buttons

EventDemo1 as event handler example- 2

- Define some of our own classes as "event handlers"
  - Add implements XListener to class declaration
    - WindowListener
    - ActionListener

```
class DataObject implements ActionListener
```

- Define implementations of the necessary methods in our own classes
  - windowClosed, windowIconified etc
  - actionPerformed

EventDemo1 as event handler example- 3

- In the initialization code, link the event sources to the listeners that will handle the events

```
public Display(DataObject d)
{
  ...
  b = new Button("South");
  b.addActionListener(d);
```

EventDemo1 as event handler example- 4

- Program displays its graphical interface
- Main thread terminates!
- Program continues with a thread handling graphics
  - This thread responds when user interacts with a GUI element and causes event to be generated
    - Click on button

Key features

- GUI incorporating event source components
- Application classes defined with event handling functionality (implements Listener)
- Instances of application classes that are to handle events are linked to event sources (addActionListener() etc)