Events

Event-driven programming

Events

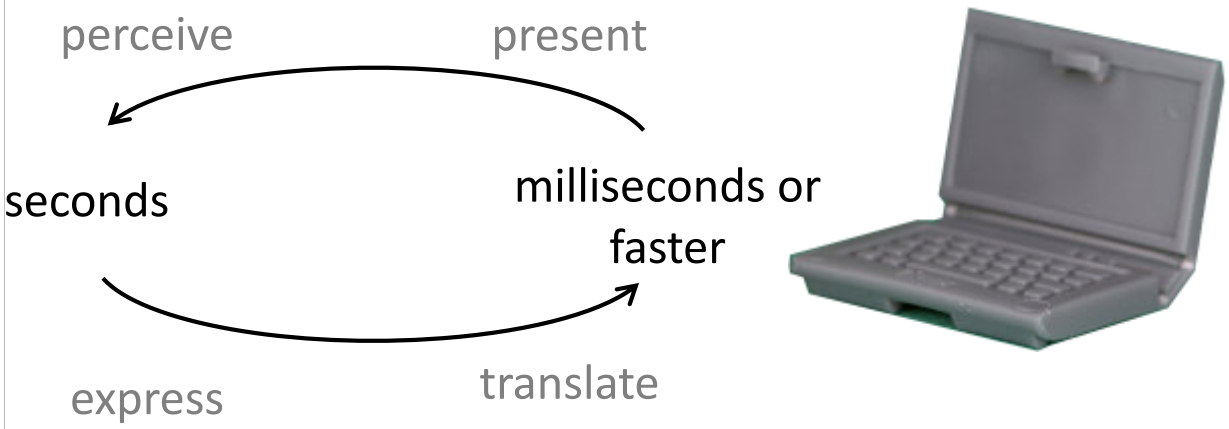
Event loop

UI thread

Event Driven Programming

User

Interactive System

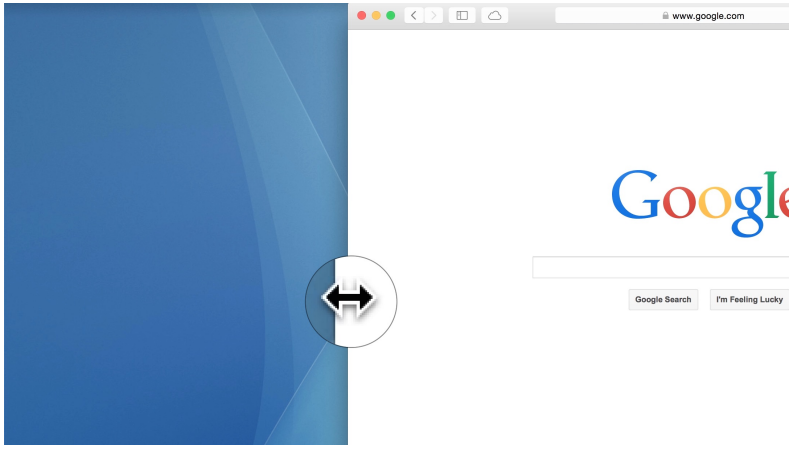


Event-driven programming is a programming paradigm that bases program execution flow on events, such as user actions (e.g. mouse-click, or key-press), or system events (e.g. timer firing).

Event Driven Programming

Very little happens unless the something else happens:

- user presses a key, moves the mouse, …
- window is resized, closed, covered …
- certain time passes (i.e. timer fires)
- (file changes, network connection, database updates, order arrives, sensor is triggered, …)



Event

- English:
  - An observable occurrence, often extraordinary occurrence
- User Interface Architecture:
  - A message to notify an application that something happened
- Examples (*exact* event name will vary by toolkit):
  - Keyboard (key press, key release)
  - Pointer Events (button press, button release, mouse move)
  - Window crossing (mouse enter, mouse leave)
  - Input focus (focus gained, focus lost)
  - Window events (window exposed, window destroy, window minimize)
  - Timer events (tick)

Java Events

- Events are defined in the java.awt.event package.
- Events include generic input events (e.g. mouse and keyboard) and specialized widget events (e.g. CaretEvent for cursor, or Timer ticks).
- As a developer, you can also define your own event-types.

Common Events

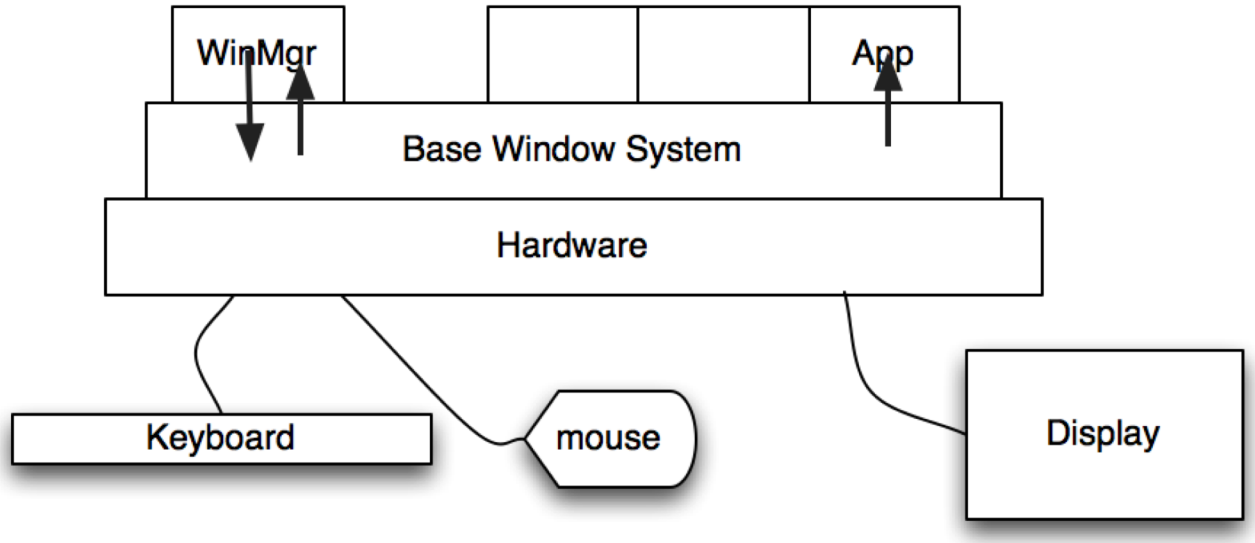
Input Device	Event	Class Name
Keyboard	A key is pressed or released	KeyEvent
Mouse	A mouse button is pressed or released, or the mouse is moved	MouseEvent
Mouse Wheel	The mouse wheel is changed	MouseWheelEvent
*	Focus gained or lost for any component	FocusEvent
*	Change in components visibility, size or position	ComponentEvent
Window	Window opened, closed, minimized, maximized	WindowEvent

https://docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html

Role of the Window System

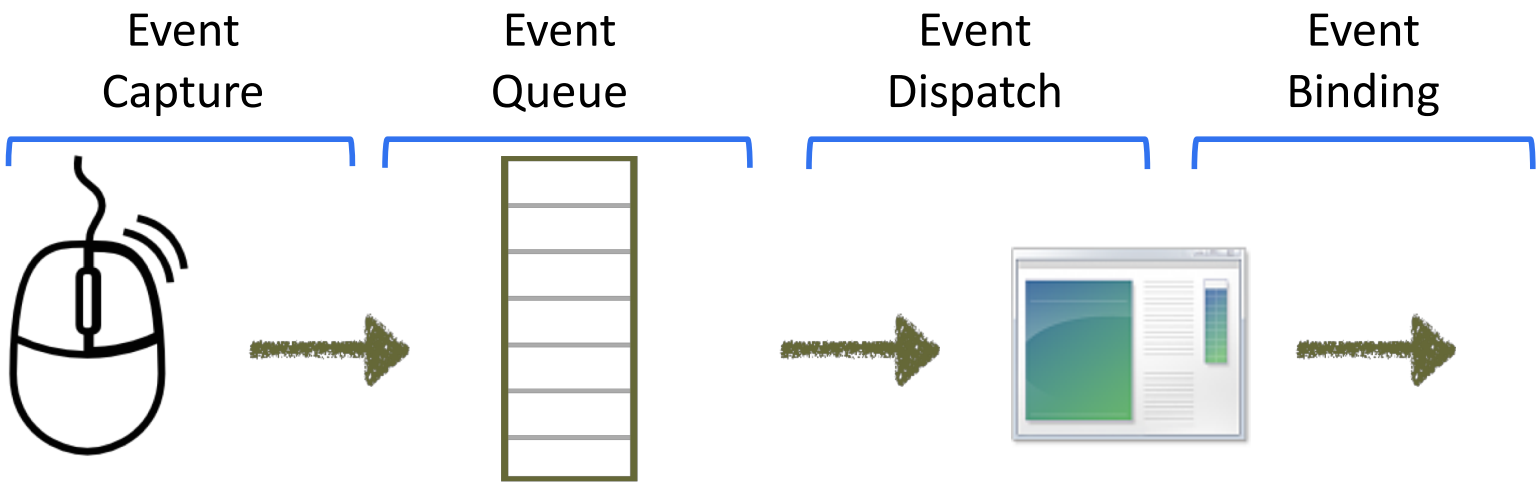
The Window System needs to package events and pass them to the appropriate application (i.e. each application handles its own events). It performs these steps:

1. Collect event information from user action, or underlying hardware
2. Put relevant information in a known event structure
3. Order the events, by time, in a queue
4. Deliver the events from the queue to the correct application



Event Architecture

- This can be modelled as a pipeline:
  - **Capture** and **Queue** low-level hardware events,
  - **Dispatch** events to the correct window and widget
  - **Bind each** event to the correct widget code that will process it



Event Loop

This pipeline is handled by an an **event loop** that continuously checks for new events, and performs these steps.

All modern GUIs have an event loop – either in the application, or in some other framework that manages events on behalf of the application.

```
while
    get next event
    handle the event:
        if the event was a quit message
            exit the loop
        else
            do something based on the type of event
     draw to the screen
loop
```

Event Loop

- Iterates through the event queue, and dispatches events
  - Serves as a low-level mechanism for event dispatch
- We want dispatch to the application window, which
  - triggered the event, or
  - is in the foreground/accepting events

- The application *may* have it's own event queue as well
  - e.g. Xwindows applications typically pull from the BWS event queue and manages them immediately.
  - e.g. Java applications don't normally manipulate an event queue, but rely on the JRE to queue, manage and dispatch events.

Event Driven Programming

Xwindows is an example of a toolkit that requires applications to handle their own event loop. We'll walk through setting up the event loop in XWindows.

We will demonstrate code to:

1. Register to receive events (i.e. register types of events)
2. Receive and interpret those types of events
3. Update program content based on event
4. Redraw the display to communicate to user what changed, and provide feedback.

Other toolkits and frameworks provide similar functionality.

Xwindows Event Loop

eventloop.min.cpp

```
// select which events to monitor
XSelectInput(dis, win, PointerMotionMask | KeyPressMask);
...
XEvent event;                          // save the event here

while( true ) {                        // event loop

   XNextEvent( display, &event );  // block waiting for an event

   switch( event.type ) {

     case MotionNotify:              // mouse movement event
        // handle here …
        break;

     case KeyPress:                  // key press event
        // handle here …
        exit(0);                       // exit event loop
        break;
   }
}
```

Selecting Input Events to “listen to”

```
// Tell the base window system what input events you want.
XSelectInput( display, window,
              ButtonPressMask | KeyPressMask |
              ButtonMotionMask );
```

- Defined masks:

```
NoEventMask, KeyPressMask, KeyReleaseMask,
ButtonPressMask, ButtonReleaseMask, EnterWindowMask,
LeaveWindowMask, PointerMotionMask,
PointerMotionHintMask, Button1MotionMask,
Button2MotionMask, ..., ButtonMotionMask,
KeymapStateMask, ExposureMask, VisibilityChangeMask,
...
```

- See
  - http://www.tronche.com/gui/x/xlib/events/types.html
  - http://www.tronche.com/gui/x/xlib/events/mask.html

Event Structure: Union

- X uses a union

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

- Each structure contains at least the following

```
typedef struct {
    int type;
    unsigned long serial;  // sequential #
    Bool send_end;   // from SendEvent request?
    Display* display; // display event was read from
    Window window; // window which event is relative to
} XKeyEvent;
```

Responding to Events (blocking)

```
while( true ) {
    XNextEvent(display, &event); // wait for 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;
    case MotionNotify:
       // ... handle event ...
       cout << event.xmotion.x << endl;
       break;
    }
    repaint( ... ); // call my repaint function
}
```

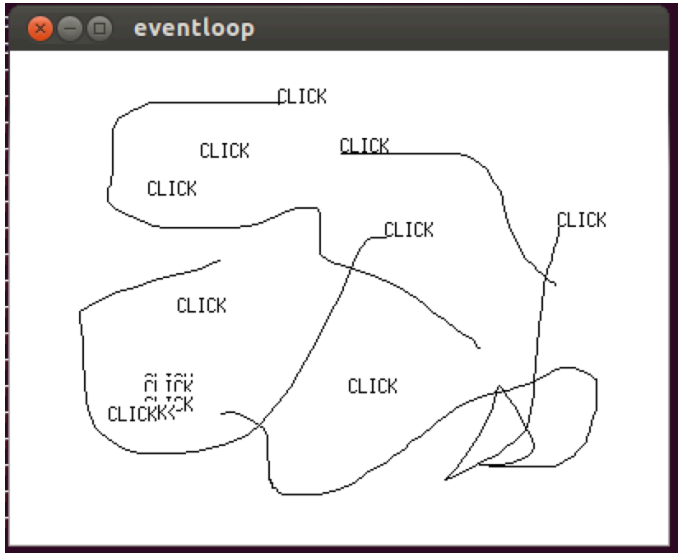
Code Review: eventloop.cpp

- XSelectInput
- XNextEvent
- event loop
- Notes:

  `KeyPress and XLookupString`

  – character vs. scan codes

  – Uses Displayables



Java Event Loop

Java UI apps typically have two main threads:

- The main application thread starts when the application is launched and runs the main method. Console apps only have one thread.
- If it's a UI app (with AWT or Swing classes), a second thread is launched to execute the UI code.
  - Called: Event-handling thread, Swing-thread or UI thread

Java Swing is a considered a single-thread toolkit, because all UI code needs to execute on this UI thread.

- Events must be handled on that thread as well
- This is why we still need an event loop in Java! We can only process one event at a time, in this single thread.
- We must be careful to not block while doing this processing.

Sidebar: How To Properly Launch a Swing UI App

- Remember there's a **main thread** and a **UI thread**
- For this reason, you may see something like this in Swing code.
- The SwingUtilities.invokeLater() method places a task on the Swing event queue, to be executed in order.

```
public static void main(String[] args) {
   SwingUtilities.invokeLater(new Runnable() {
      public void run() {
         new createAndShowGUI();
      }
   });
}
```

https://bitguru.wordpress.com/2007/03/21/will-the-real-swing-single-threading-rule-please-stand-up/

Java Event Loop (hidden)

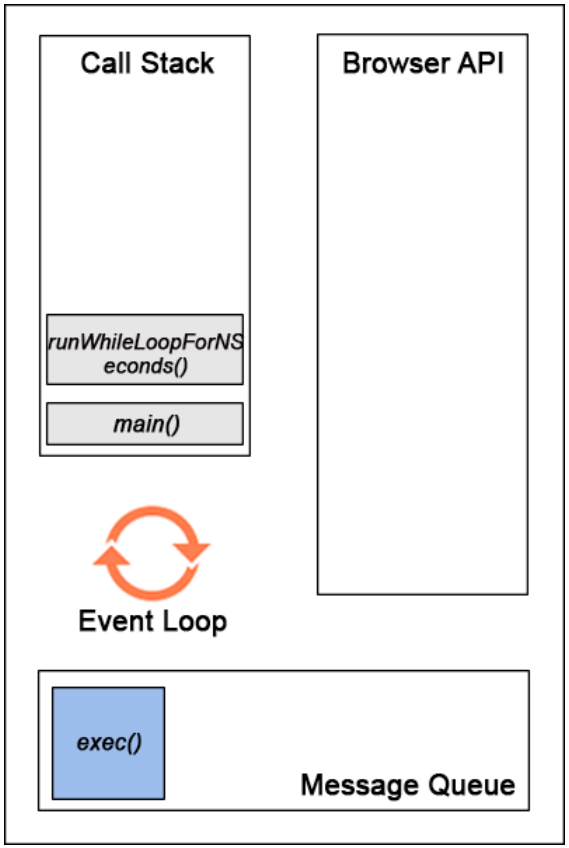
EventLoop.java

```
public class EventLoop extends JPanel  {

    EventLoop()
            throws InterruptedException, InvocationTargetException  {

        EventQueue.invokeAndWait(new Runnable() {
            public void run() {
                EventQueue eq = Toolkit.getDefaultToolkit().getSystemEventQueue();
                // replace the current event queue with MyEventQueue
                eq.push(new MyEventQueue());
                System.out.println("Run");
            }
        });
    }
```

```
    // kind of like an event loop
    private class MyEventQueue extends EventQueue {
        // mouse events come in here
        public void dispatchEvent(AWTEvent e) {
            //System.out.println("dispatchEvent " + e.getID() );
            if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
                MouseEvent me = (MouseEvent)e;
                x = me.getX();
                y = me.getY();
                System.out.println("(" + x + "," + y + ")");
                repaint();
            }
            super.dispatchEvent(e);
        }
    }
```

Javascript Event Loop

```
1  function main(){
2    console.log('A');
3    setTimeout(
4      function exec(){ console.log('B'); }
5    , 0);
6    runWhileLoopForNSeconds(3);
7    console.log('C');
8  }
9  main();
10 function runWhileLoopForNSeconds(sec){
11   let start = Date.now(), now = start;
12   while (now - start < (sec*1000)) {
13     now = Date.now();
14   }
15 }
16 // Output
17 // A
18 // C
19 // B
```



Browser-managed event loop

https://www.youtube.com/watch?v=8aGhZQkoFbQ @10:35, 11:49

https://www.student.cs.uwaterloo.ca/~cs349/videos/what_the_heck_is_the_event_loop_anyway.mp4