HCI System

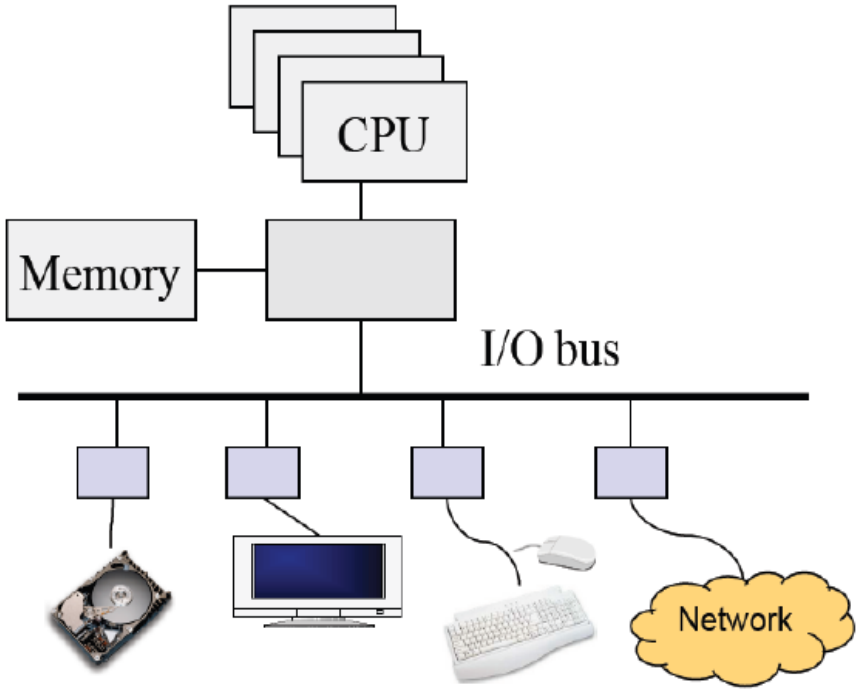
Review: Computer Architecture

Compute hardware

- CPU and caches
- Chipset
- Memory

I/O Hardware

- I/O bus or interconnect
- I/O controller or adaptor
- I/O device



Two types of I/O

- Programmed I/O (PIO)
  - CPU does the work of moving data
- Direct Memory Access (DMA)
  - CPU offloads the work of moving data to DMA controller

I/O or interactivity at the low/system level

- I/O devices:
  - Communication devices
    - Input only (keyboard, mouse, joystick, light pen) / Output only (printer, visual display, voice synthesizers…)
    - Input/output (network card…)
  - Storage devices
    - Input/output (disk, tape, writeable optical disks…)
    - Input only (CD-ROM…)
- Every device type/model is different: input/output/both, block/char oriented, speed, errors, …
- Main tasks of I/O system:
  - Present logical (abstract) view of devices
    - Hide details of hardware interface
    - Hide error handling
  - Facilitate efficient use: Overlap CPU and I/O
  - Support sharing of devices
    - Protection when device is shared (disk)
    - Scheduling when exclusive access needed (printer)

How does the CPU talk to devices?

(+ S/W too)

Device controller: Hardware that enables devices to talk to the peripheral bus

Device controller allows OS to specify simpler instructions to access data

Example: a disk controller

- Translates "access sector 23" to "move head reader 1.672725272 cm from edge of platter"
- Disk controller "advertises" disk parameters to OS, hides internal disk geometry
- Most modern hard drives have disk controller embedded as a chip on the physical device

Polling- vs. Interrupt-driven I/O

Polling

- CPU issues I/O command
- CPU directly writes instructions into device's registers
- CPU busy waits for completion

To carry out I/O tasks, the processor enters a loop, continually checking the I/O device status whether it is ready, and incrementally accomplishes the I/O task.

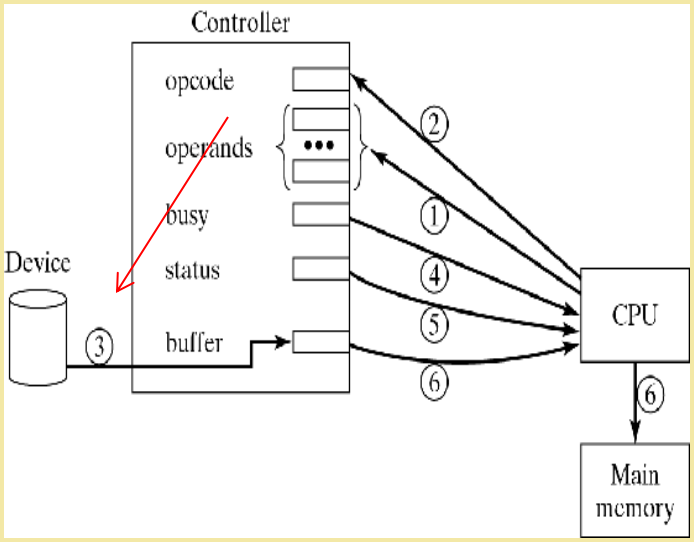
This form of an I/O is deficient in supporting asynchronous user-driven (anytime) I/O and wastes CPU time blocking other non-I/O processes to move on …

(well … see later slides as how OS can help)

Programmed I/O with Polling

- Driver operation to **input** sequence/blocks of characters:

```
i = 0;
do {
    write_reg(opcode, read);
    while (busy_flag == true) {…??...}; //waiting
    mm_in_area[i] = data_buffer;
    increment i;
    compute;
} while (…)
```



Programmed I/O with Polling

- Driver operation to **output** sequence/blocks of characters:

```
i = 0;
do {
    compute;
    data_buffer = mm_out_area[i]; /*put data first */
    increment i;
    write_reg(opcode, write); /* issue command */
    while (busy_flag == true) {…??...};   /* busy wait here*/
  } while (data_available)
```

Busy flag checking can come first
And writing to the reg could be done only after ready

As soon as device becomes ready to serve
The buffered data will be transferred to the device

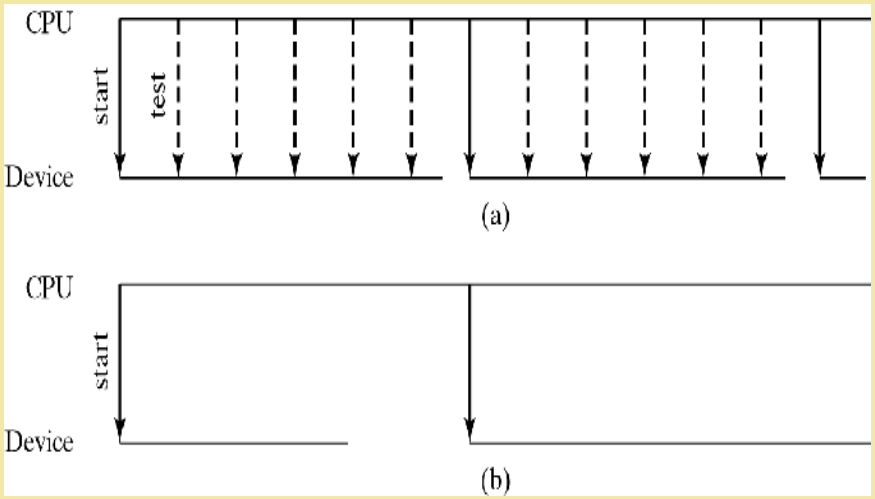
But computation do not proceed until data buffer tranfers out

Programmed I/O with Polling

- What to do while waiting?
  - Normally, Just Idle (busy wait)
  - Some other computation (what?) and repeatedly poll?
    - How frequently to poll?
      - Device may remain unused for a long time

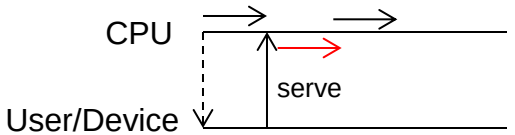
Coordination is difficult

Have some mechanism for
device to actively say something
Instead of CPU just asking around



Interrupt-driven I/O

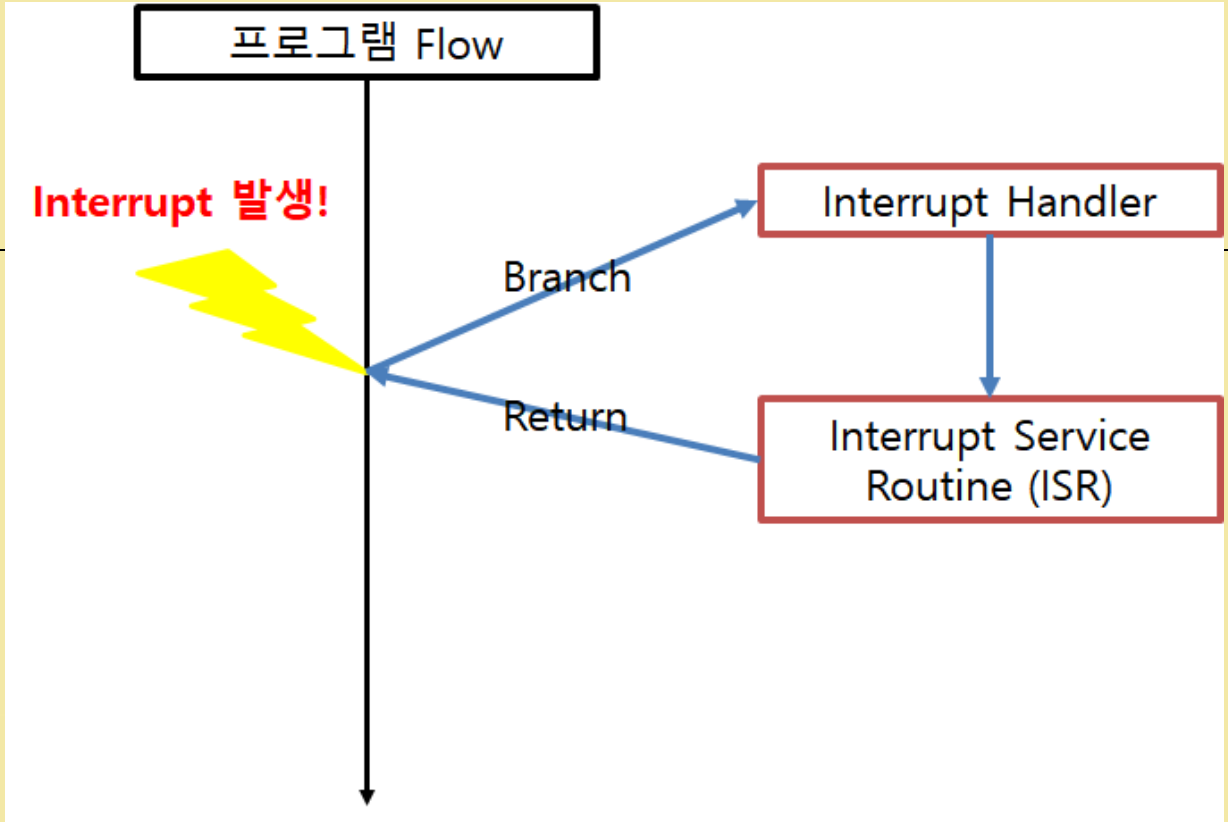
- CPU issues I/O command ?
- CPU directly writes instructions into device's registers (and continue until …)
- CPU continues operation until interrupt



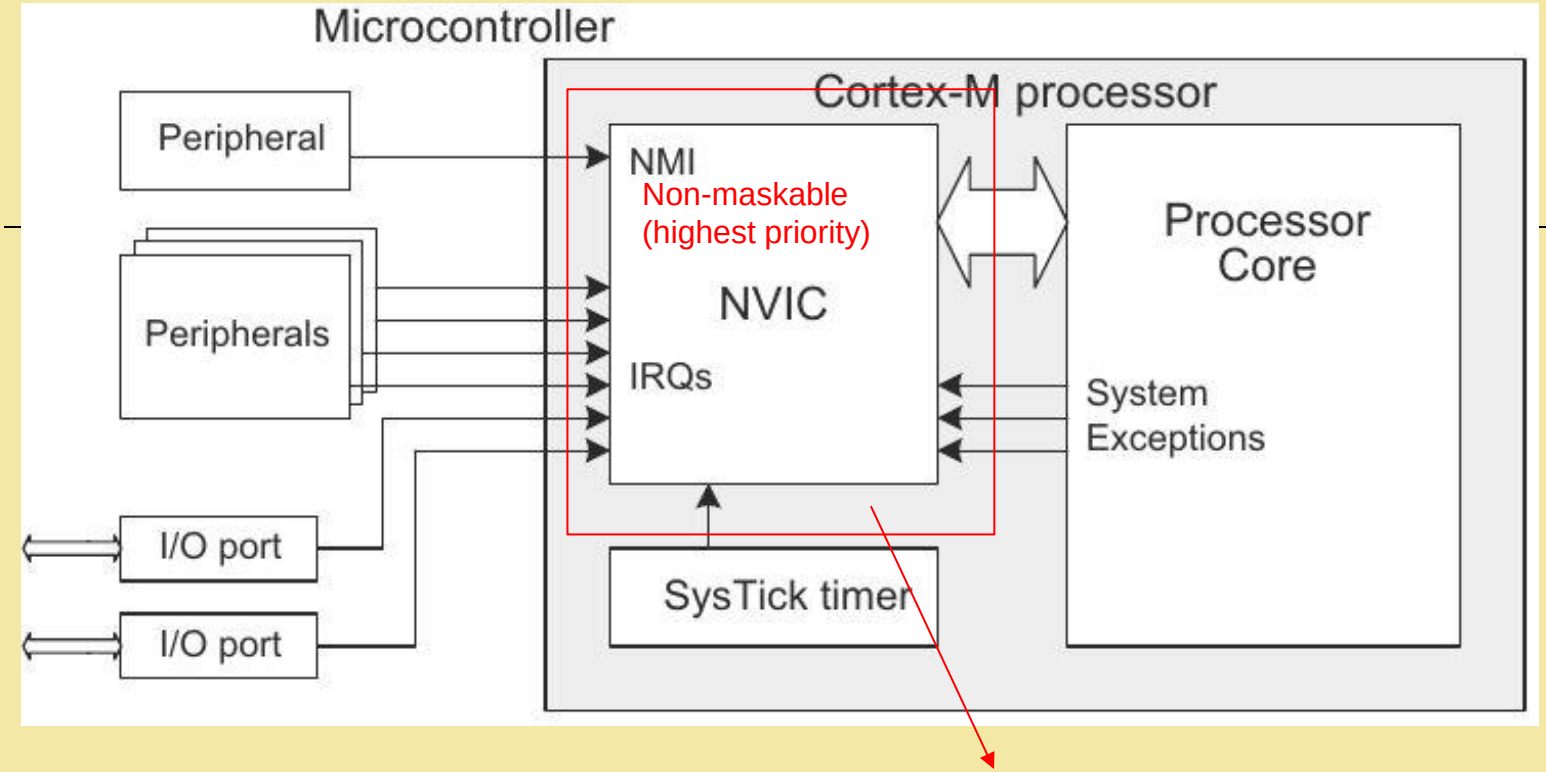
Device signals when ready
Other times, CPU continues

- Interrupt
  - "Signal" to the processor indicating an (usually I/O) event has occurred and requires to be handled.
  - Mechanism
    - An interrupt signal is interpreted so that the address of its handler procedure can be looked up and be executed while suspending the on-going process for a moment.
    - After the handler is finished, the suspended process resumes again.
    - An arrival of an interrupt is checked at a very fast rate, as part of the machine instruction "Fetch-Decode-Execute-Check Interrupt" cycle.
    - This practically makes the processor "always" listening to the incoming events and being ready to serve them.

At a higher level, the I/O operation is often described in terms of "events" and "event handlers," which are abstractions of the lower level interrupt mechanism.
Such an abstraction makes the understanding of the event handling and interactive program development much easier.



After the CPU acks the interrupt, an interrupt controller (PIC) will send the interrupt number back to the CPU. CPU then uses that interrupt number as an index into the interrupt vector table to find the appropriate handler and then continues from there



Interrupt controller
(may be separate outside the processor)

- Manage many interrupts
  - Group them (see later USB example)
  - Priority

x

Offset into table

Table @ 1000

Table 63. Vector table for other STM32F10xxx devices

e.g. Offset = (x) * 4

Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000_0000
-	-3	fixed	Reset	Reset	0x0000_0004
-	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000_0008
-	-1	fixed	HardFault	All class of fault	0x0000_000C
-	0	settable	MemManage	Memory management	0x0000_0010
-	1	settable	BusFault	Prefetch fault, memory access fault	0x0000_0014
-	2	settable	UsageFault	Ur state	0x0000_0018
-	-	-	-	Reserved	0x0000_001C - 0x0000_002B
-	3	settable	SVCall	System service call via SWI instruction	0x0000_002C
-	4	settable	Debug Monitor	Debug Monitor	0x0000_0030
-	-	-	-	Reserved	0x0000_0034
-	5	settable	PendSV		0x0000_0038
-	6	settable	SysTick	System tick timer	0x0000_003C
0	7	settable	WWDG	Window watchdog interrupt	0x0000_0040
1	8	settable	PVD	PVD through EXTI Line detection interrupt	0x0000_0044
2	9	settable	TAMPER	Tamper interrupt	0x0000_0048
3	10	settable	RTC	RTC global interrupt	0x0000_004C

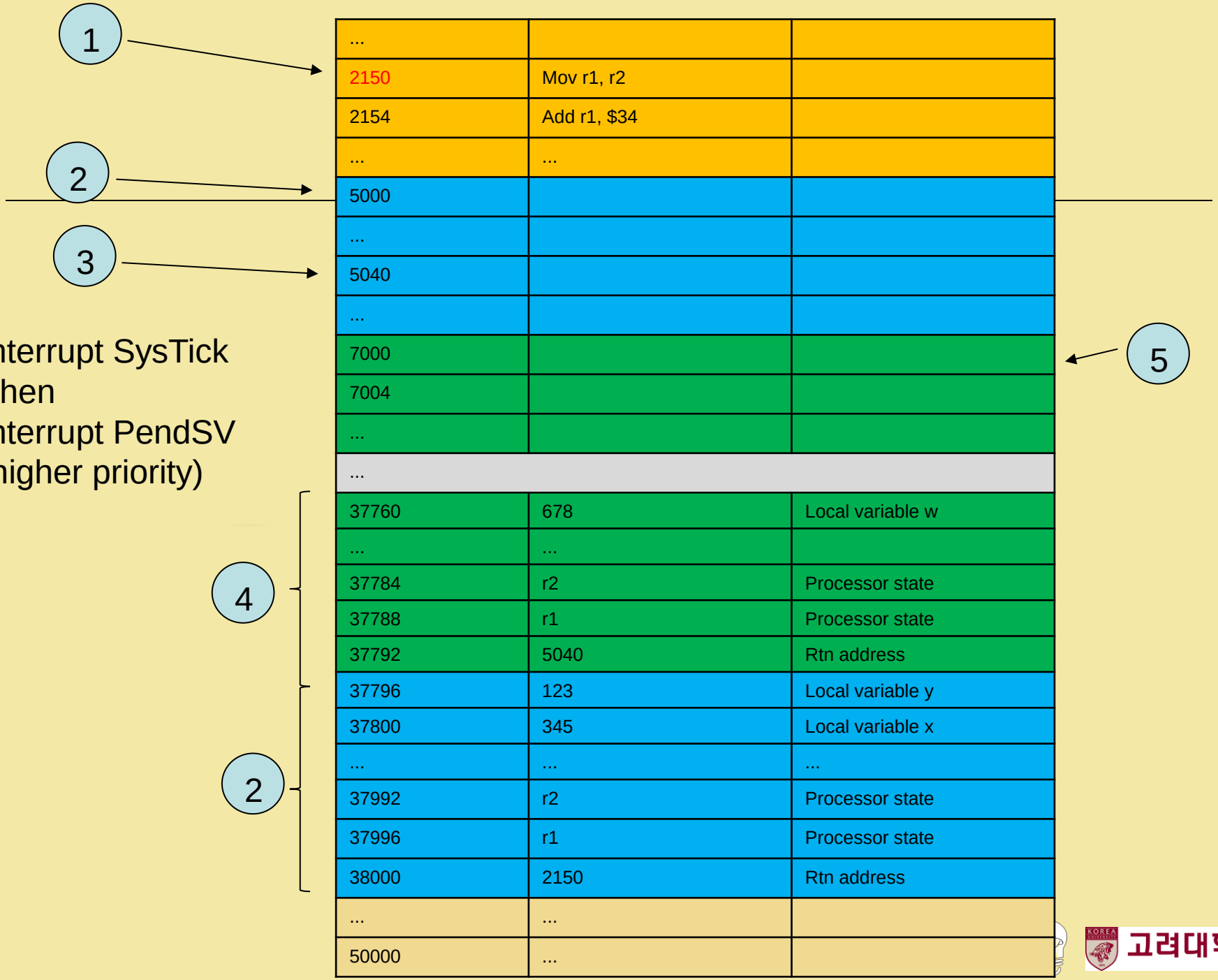
(1) * 4

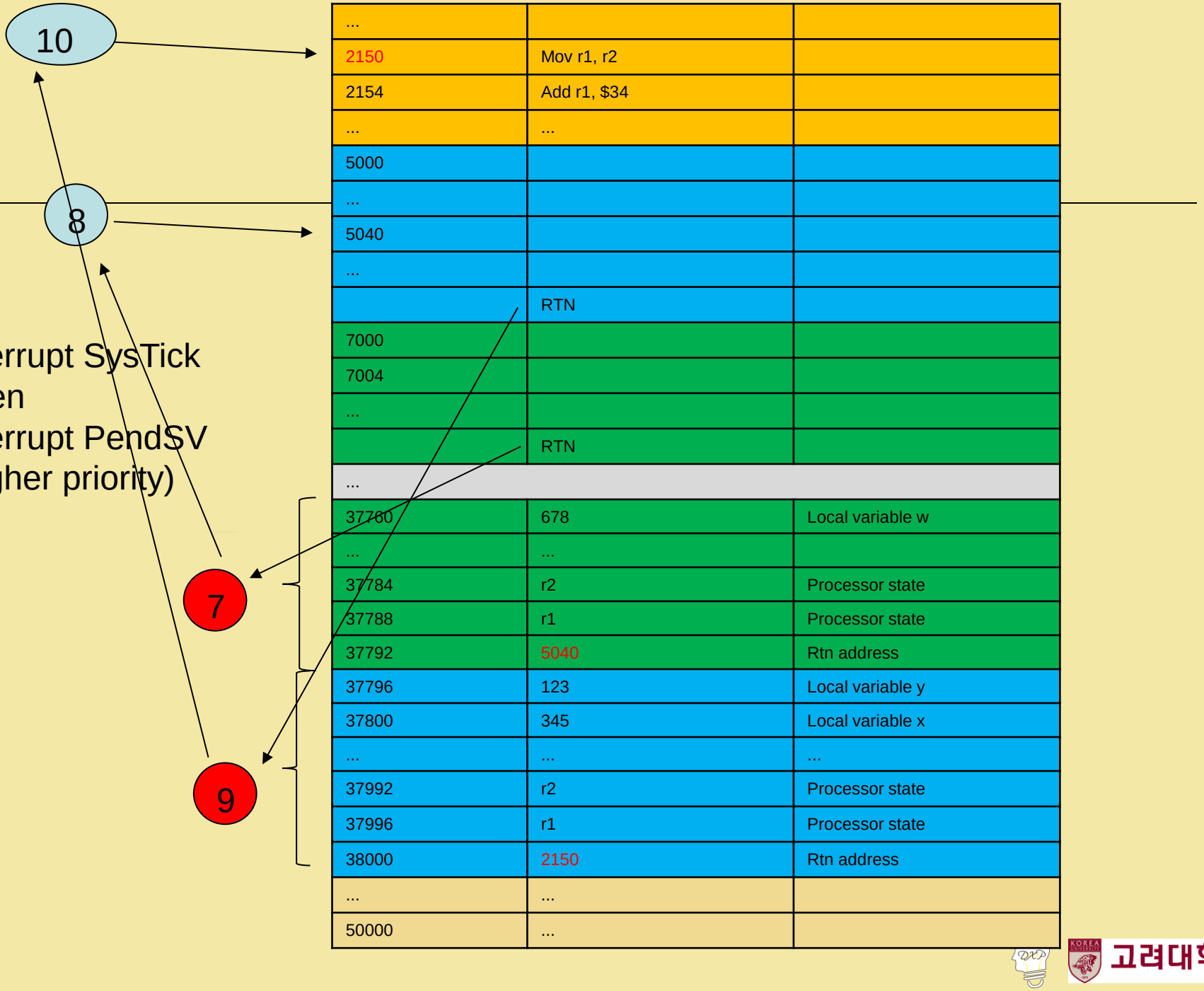
(6) * 4 = 24 = 18_F

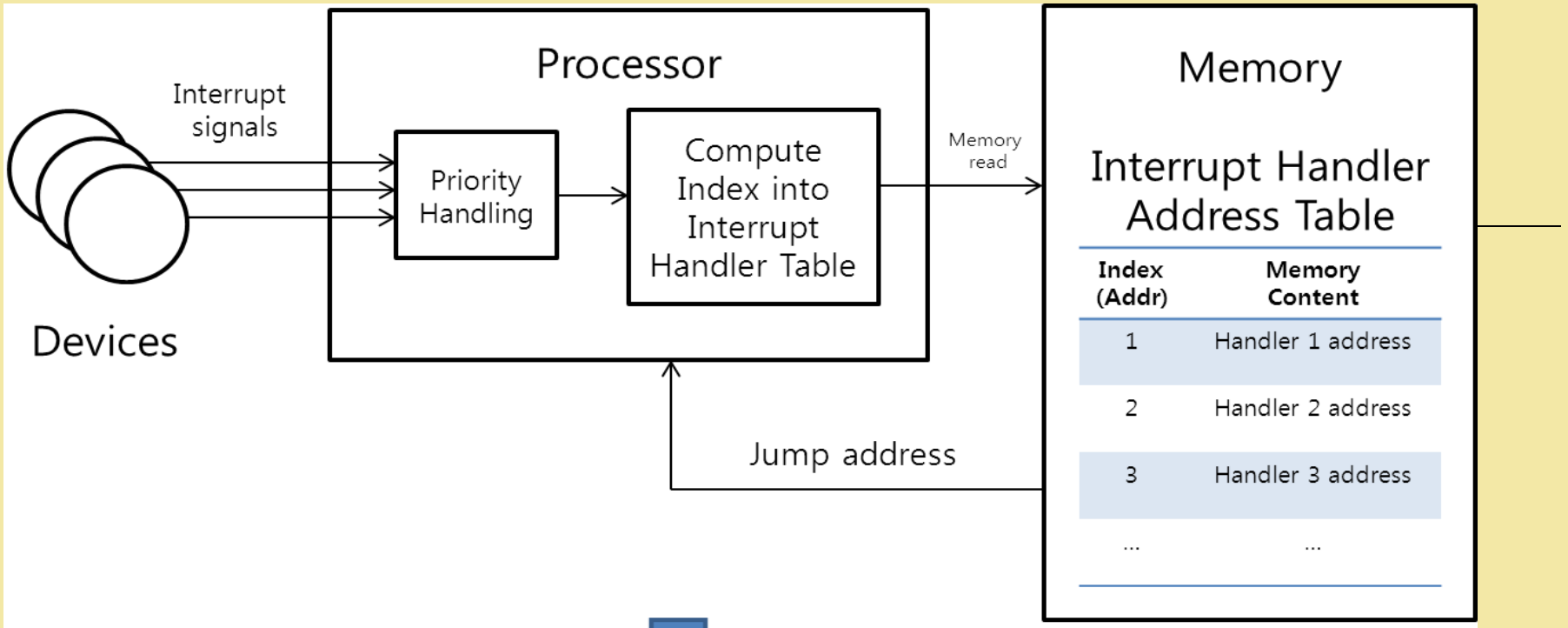
(14) * 4 = 56 = 38_F

...	
1000	
...	
1038	5000
103C	7000
...	
...	
...	
5000	
5004	
5008	

고려대학교


1
2
3
4
5
2
...
2150 Mov r1, r2
2154 Add r1, $34
...
5000
...
5040
...
7000
7004
...
...
37760 678 Local variable w
... ...
37784 r2 Processor state
37788 r1 Processor state
37792 5040 Rtn address
37796 123 Local variable y
37800 345 Local variable x
... ...
37992 r2 Processor state
37996 r1 Processor state
38000 2150 Rtn address
... ...
50000 ...
Interrupt SysTick
Then
Interrupt PendSV
(higher priority)


10
8
7
9
Interrupt SysTick
Then
Interrupt PendSV
(higher priority)
...
2150 Mov r1, r2
2154 Add r1, $34
...
5000
...
5040
...
RTN
7000
7004
...
RTN
...
37760 678 Local variable w
... ...
37784 r2 Processor state
37788 r1 Processor state
37792 5040 Rtn address
37796 123 Local variable y
37800 345 Local variable x
... ...
37992 r2 Processor state
37996 r1 Processor state
38000 2150 Rtn address
... ...
50000 ...



Abstraction

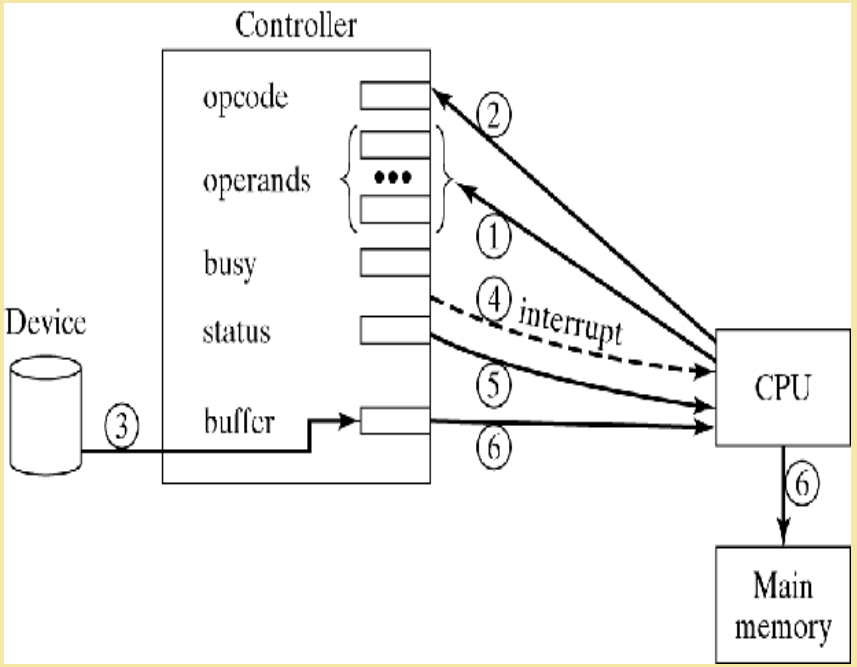
Event id	Handler Table
1	Handler 1 address
2	Handler 2 address
3	Handler 3 address
...	...

Other uses of interrupts: exceptions

- Division by zero, wrong address
- System calls (software interrupts/signals, trap)
- Virtual memory paging

Programmed I/O with Interrupts (?)

```
i = 0;
do {  write_reg(opcode, read);
⇨     while (busy_flag == true) {…};  //active wait
        mm_in_area[i] = data_buffer;
        increment i;
        compute;
} while (…)
```



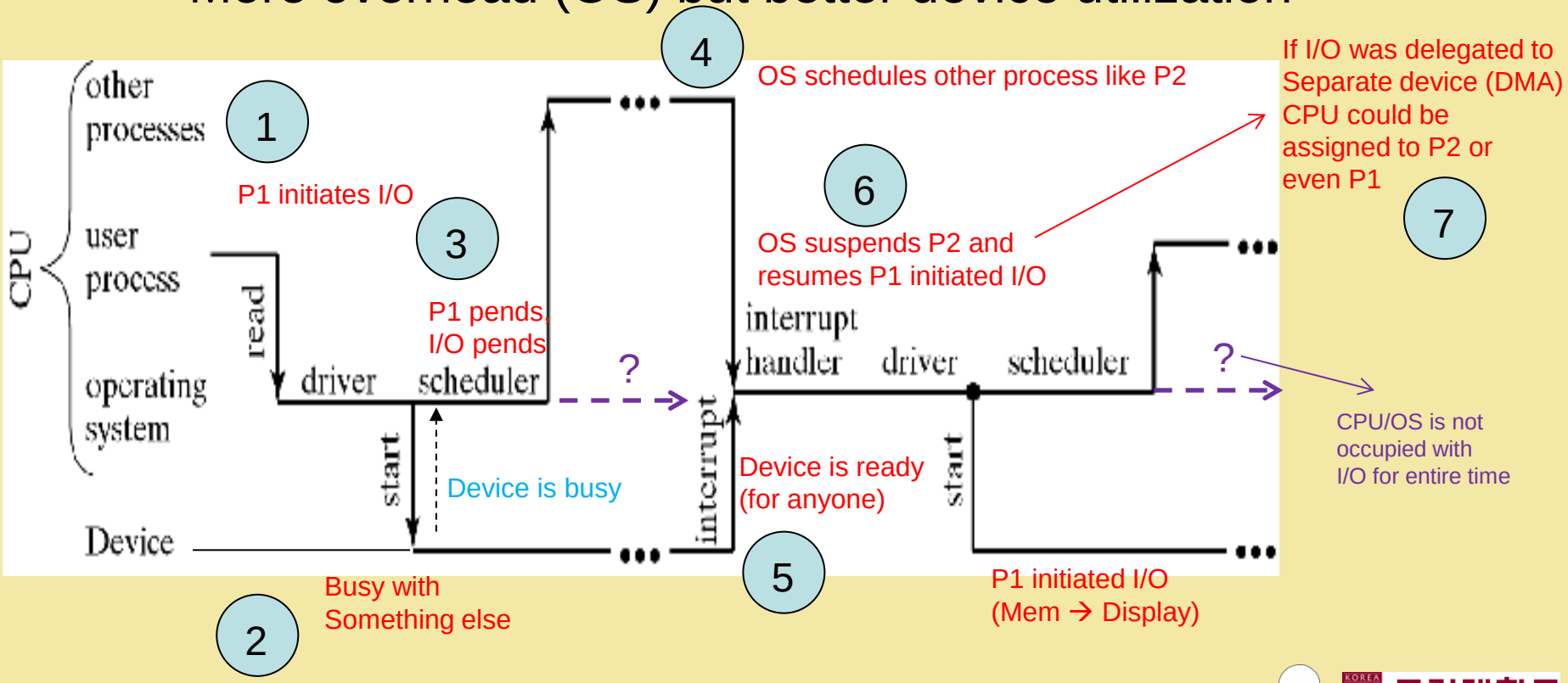

```
i = 0;
do {  write_reg(opcode, read);
⇨       block to wait for interrupt;
          mm_in_area[i] = data_buffer;
          increment i;
          compute;
} while (…)
```

what can OS do for you?

Really same as programmed io (if just blocking …)

Programmed I/O with Interrupts

- I/O with interrupts: more complicated, **involves OS**
- More overhead (OS) but better device utilization



Sync vs Asynchronous I/O

As seen at the OS level

Muilti-processing
Concurrent processing

Synchronous I/O

- read() or write() will block a user process until its completion
- OS overlaps synchronous I/O with another process

Well only if
Operations after I/O
are non dependent …

Asynchronous I/O

- read() or write() will not block a user process
  - returns -1, sets error code EAGAIN or EWOULDBLOCK
- user process can do other things before I/O completion
- can determine if device is ready with select() / poll() / epoll()
- Make asynchronous with O_NONBLOCK option on open() or later via fcntl()

Like printf …
What if,
z = read_device (X, 100)?

Hide the detailed interrrupt mechanism

고려대학교

Programmed Input Device

User driven I/O

Device controller

- Status registers
  - ready: tells if the host is done
  - busy: tells if the controller is done
  - int: interrupt
  - …
- Data registers

A simple mouse design

- When moved, put (X, Y) in mouse's device controller's data registers 0
- Interrupt CPU 1

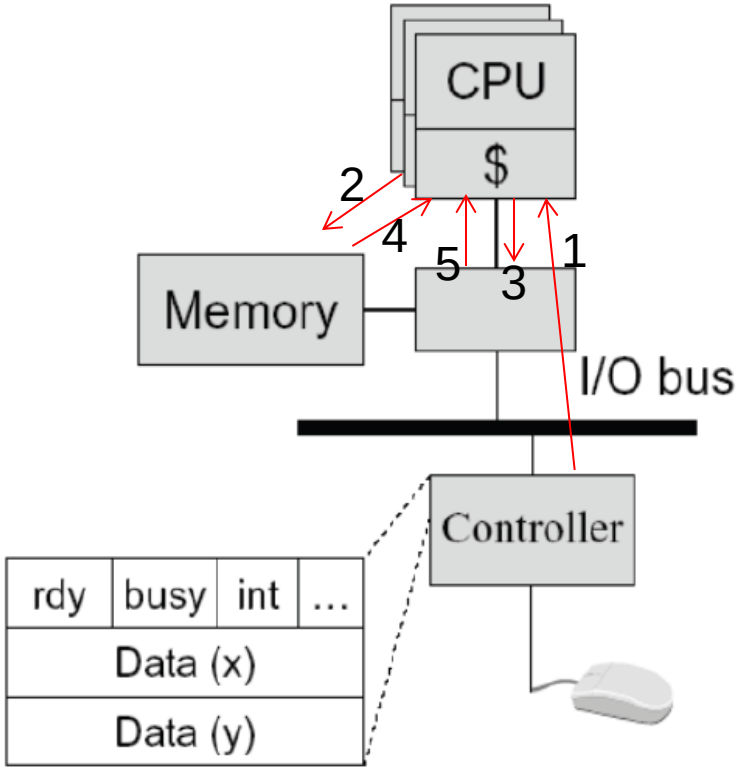
Input on an interrupt

2 CPU saves state of currently-executing program

- Reads values in X, Y registers 3

3.1 Sets ready bit

- Wakes up a process/thread or execute a piece of code to handle interrupt 5



4 Handler Rtns/ Restore

- Process Blocking
  - e.g. request for some service and waiting for a reply
- Synchronous programming
  - Waits until not blocking anymore (until the service comes back with resutls
  - Do things one at a time
- Asynchronous programming
    - Let the wait go on / Do some other unrelated stuff
    - Alternative: Create another thread and go parallel or concurrent
      - Javascript has only one thread – need some added mechanism

Device Management (OS)

- Device-independent techniques:
  - Buffering/caching, Queuing
    - Allows asynchronous operation of producers and consumers
    - Allows different granularities of data
    - Consumer or producer can be swapped out while waiting for buffer fill/empty
  - Error Handling
    - Repair/reinstall program
    - Transient SW/HW errors : Error detecting/correcting codes
  - Device Scheduling/Sharing

Device drivers

Manage the complexity and differences among specific types of devices (disk vs. mouse, different types of disks …)

Special driver for each device type or group of them (eg, USB)

Typical Device Driver Design

Operating system and driver communication

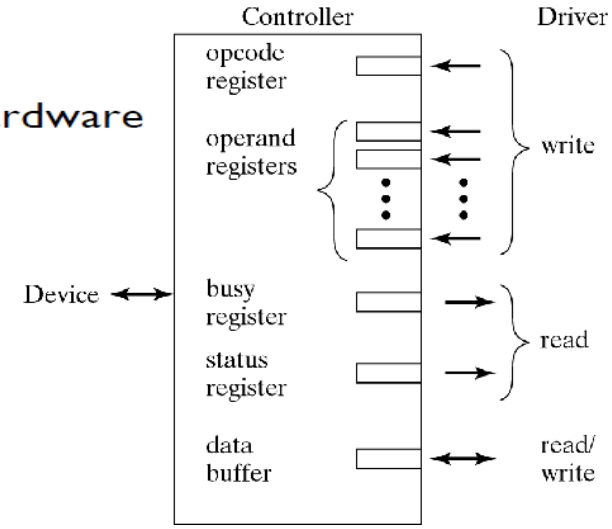
- Commands and data between OS and device drivers

Driver and hardware communication

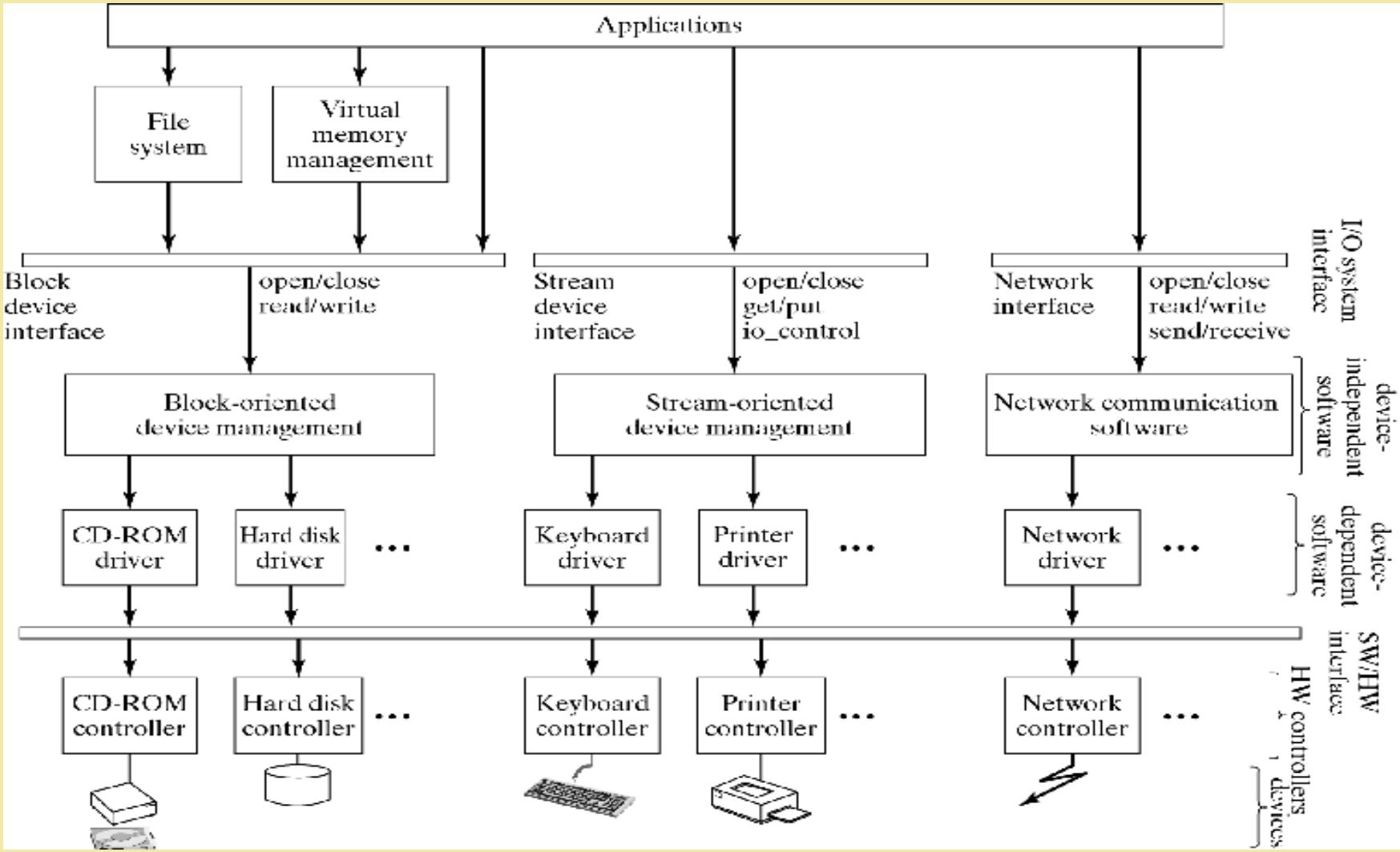
- Commands and data between driver and hardware

Driver responsibilities

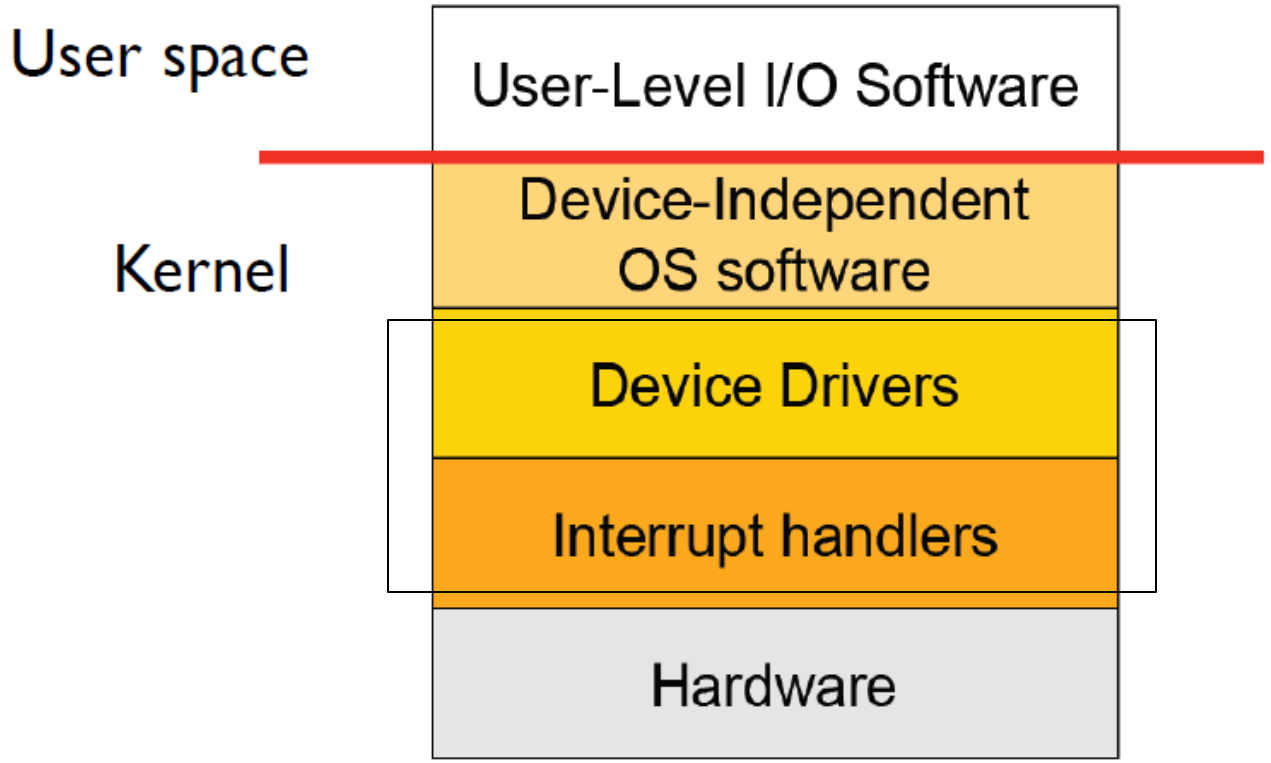
- Initialize devices
- Interpreting commands from OS
- Schedule multiple outstanding requests
- Accept and process interrupts
- Manage data transfers



Hierarchical Model of the I/O System



I/O Software Stack



UI Layer

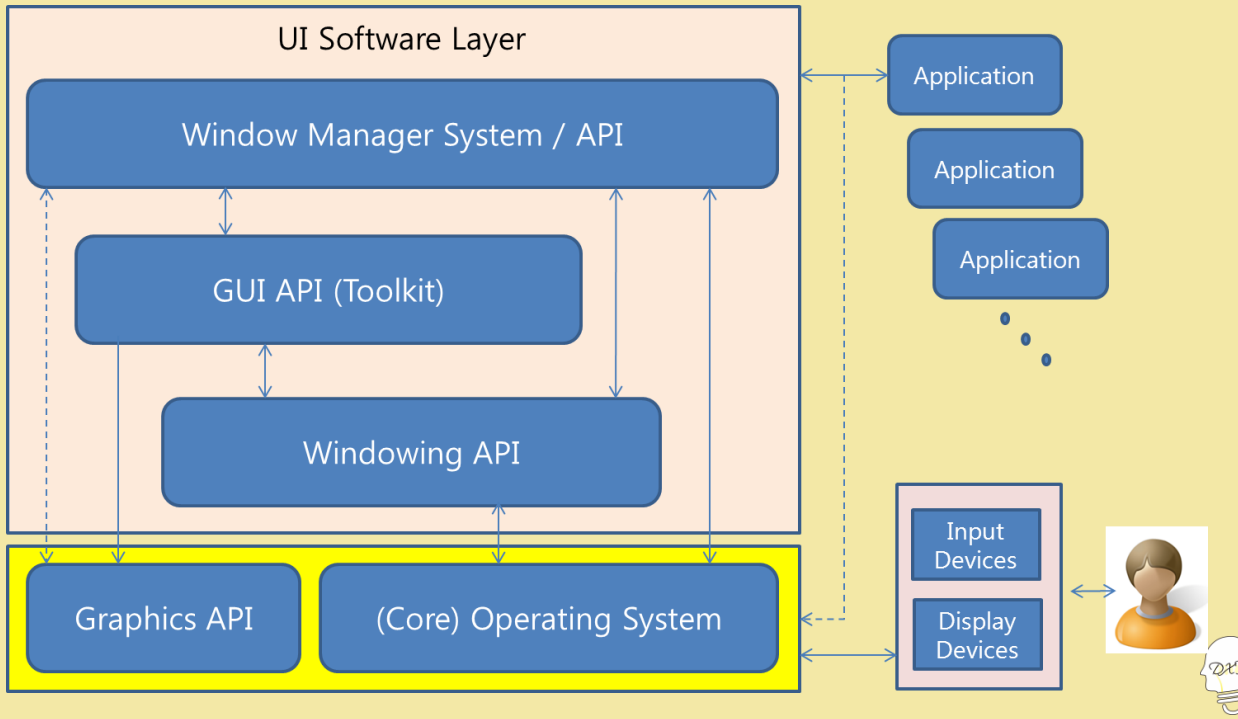
- Set of software layers that operate
  - Above the core operating system and underneath the application
  - Encapsulate and expose system functions for
    - Fluid input and output (I/O),
    - Facilitation of development of I/O functionalities - API or toolkit
    - On-line or run-time management of graphical applications (as windows/GUI)
      - In the form of separate application often called the window manager
  - Most interfaces are 2D/3D graphic based
      - Graphics subsystem is a must like OpenGL, …

Window (Layer) system

- Window
  - Rectangular portion of the screen associated with a given application is used as a space and channel for interacting with the application.
  - Not just physical window itself but manifestation of thread/process
- Window system
  - Provides a set of s/w services that allow programs to create organize content of windows (developer's perspective)
  - Windows API – basic windowing function (create/destroy, deactivate window, etc.)
  - Allows control (user perspective)
  - Basic window fns and management control look and feel of interface
    - Managed as hierarchy of UI resources
      - » Root window and its children objects/windows
      - » Note that one app can have several root windows
      - » Usually within one app but things can go across apps (cut and paste)

Window (Layer) manager

- A separate program for users
- Background running
  - Manifestation of the OS
- Based on window system and graphic API



Interactive Software Architecture:
Events,UI Objects and Event Handlers

- UI object
  - Window (or layer)
  - Buttons, menus, icons, forms, dialog boxes, text box, etc. (referred to as “GUI objects” or “widgets”)
  - GUI based interactive applications would have a
    - “Top window”
    - with all other UI objects or widgets to be logically and/or spatially subordinate to it
- Current operating systems mostly supporting concurrency, and as such, separate windows/widgets for concurrent applications can co-exist, overlap with one another, switched for the current focus.
  - When there are multiple windows (and one mouse/keyboard), the user carries out an action to designate the active or current window *in focus* to which the input events will be channeled.
  - Two major methods for focusing are:
    - Click-to-type: the user has to explicitly click on the window before making input into it (regardless of mouse position, last object that was clicked on will be the one in focus)
    - Move-to-type: the window for which the mouse cursor is over becomes in focus.

Interactive Software Architecture:
Events, UI Objects and Event Handlers

- While not all UI layers are modeled and implemented in an object oriented fashion, many recent ones are.
- Thus we can think of generic or abstract object classes for a window and other UI objects and widgets and organized hierarchically
- Moreover, we can designate the background screen space as the default “root” system window (which becomes automatically activated upon system start) onto which children application windows and GUI elements (e.g. icons, menus) are put on.
- The background also naturally becomes the top window for the window manager process.

Interactive Software Architecture:
Events, UI Objects and Event Handlers

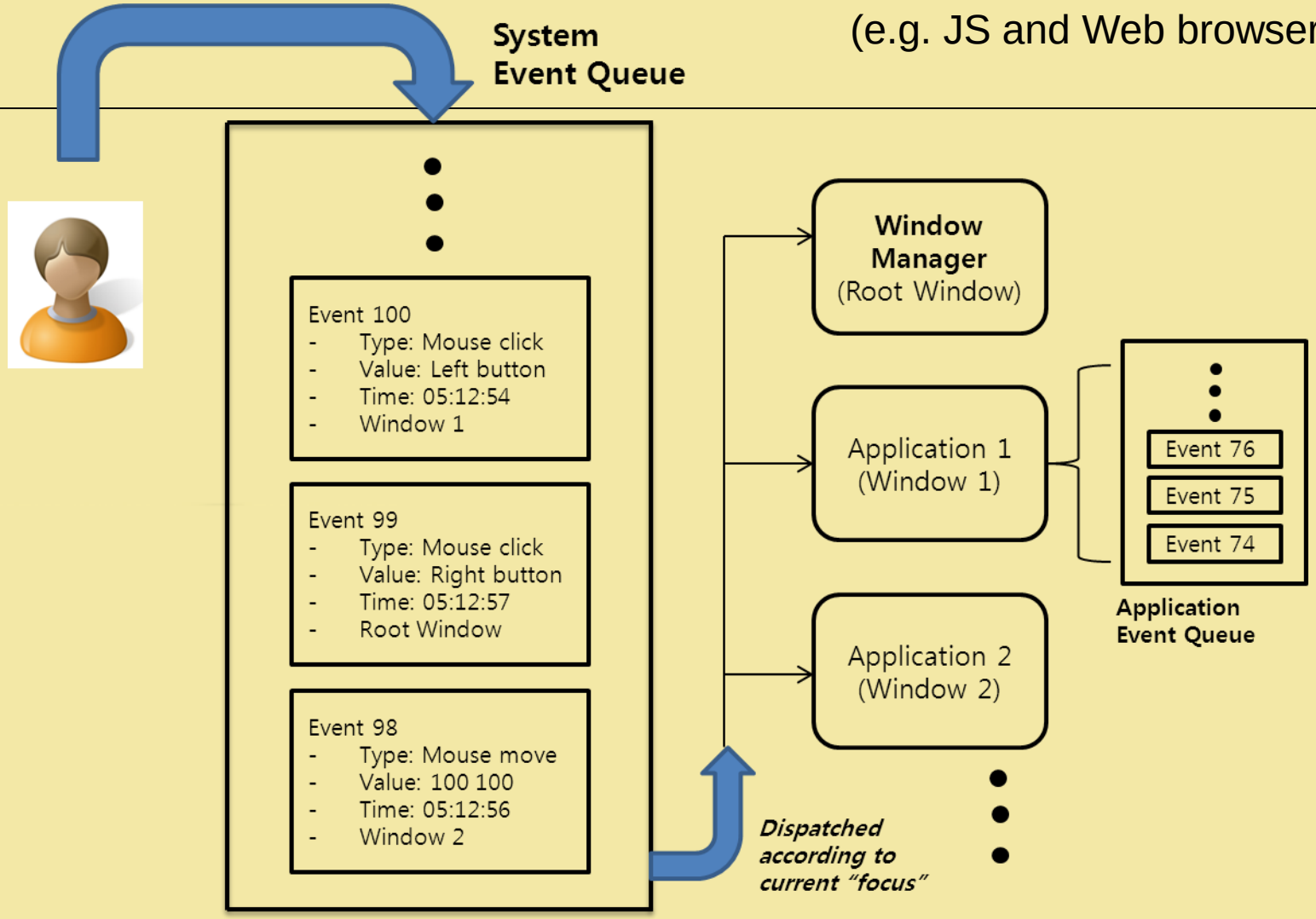
- Whether it is the root (background) window, application (top) window or GUI widget, as an interaction channel or object, it will receive input from a user through input devices such as the keyboard, mouse, etc.
- The physical input from the user/devices is converted into an "event" (e.g. by the device drivers and operating system)
  - Simply, data containing information about the user intent or action.
    - Aside from the event value itself (e.g. which key was pressed)
    - Additional information such as its type, a time stamp, the window to which it was directed, coordinates (e.g. in case of a mouse or pen event)
- Various events are put into a queue by the operating system (or the windowing system) and "dispatched" (or dequeued) e.g. according to the current focus) and processed by the target window specific processes or programs.

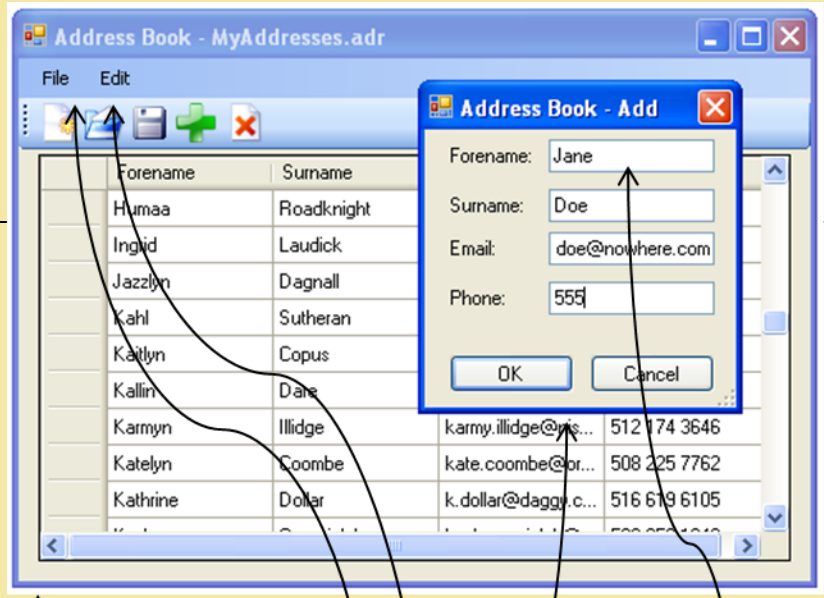
Interactive Software Architecture:
Events, UI Objects and Event Handlers

- Events not necessarily correspond exactly just to the individual physical input. The raw inputs may be filtered to extract and processed to form “meaningful” inputs from the current “context.”
  - For instance, a sequence of raw inputs may form a meaningful event such as “double-click,” commands with modifiers (e.g. alt-ctrl-del), “mouse-enter/exit” (e.g. detection of mouse cursor leaving a particular window).

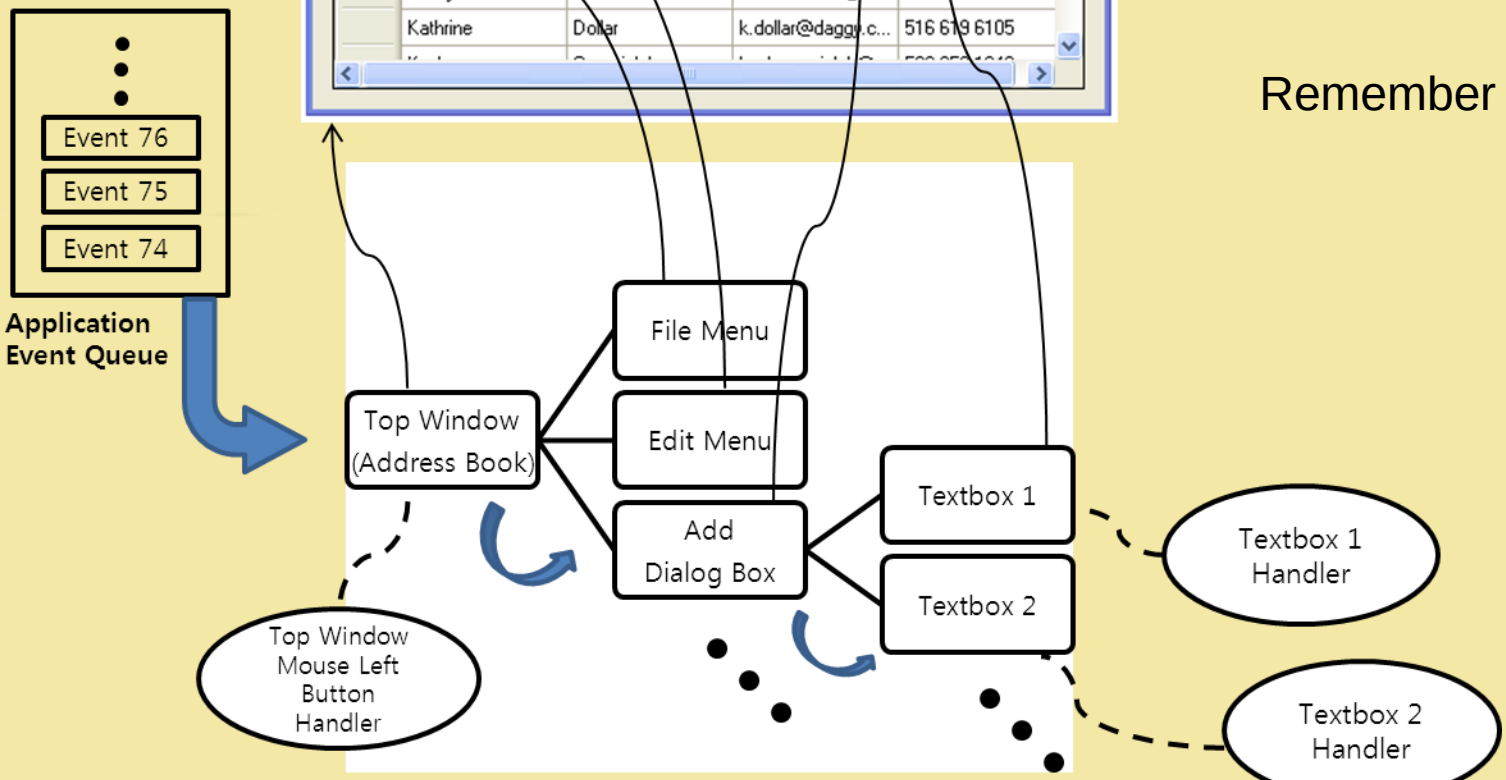
- “Two tier” event queuing system
  - System-level event queuing system that dispatches the events at the application level (which would typically be associated with its own top-level window).
  - Each application or process also typically manages its own event queue and dispatching them to its UI objects.
  - The proper event is “caught” by the UI object as it traverses down the application’s hierarchical UI structure e.g. from top to bottom (there are other methods as well).
  - Then the event handler (also sometimes called the callback function) associated with the UI object is activated in response to the event that is “caught.”

Note application is built around some framework (e.g. JS and Web browser)





Remember DOM?

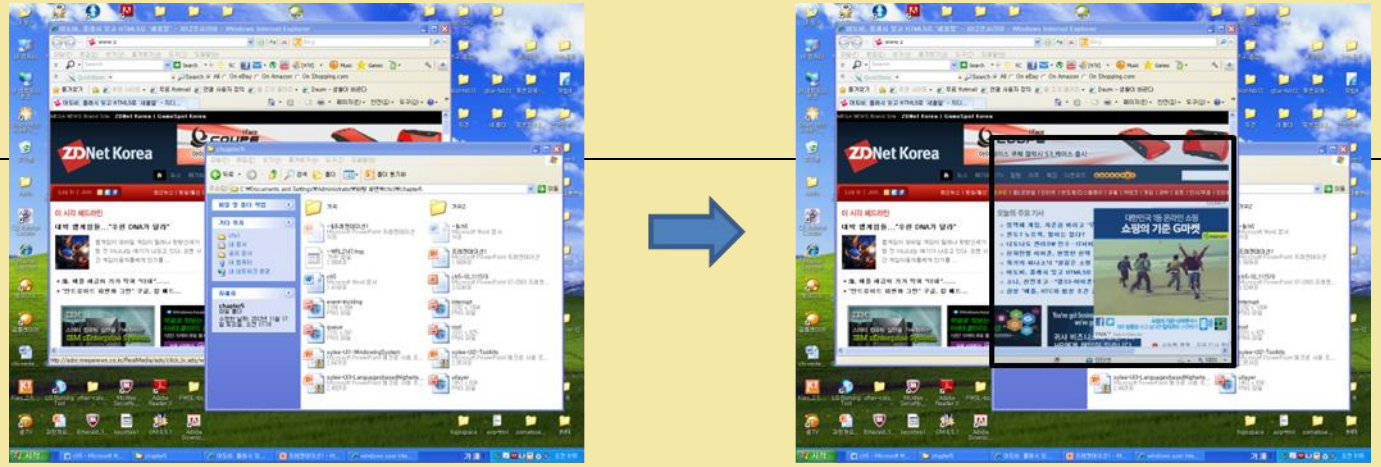


고려대학교

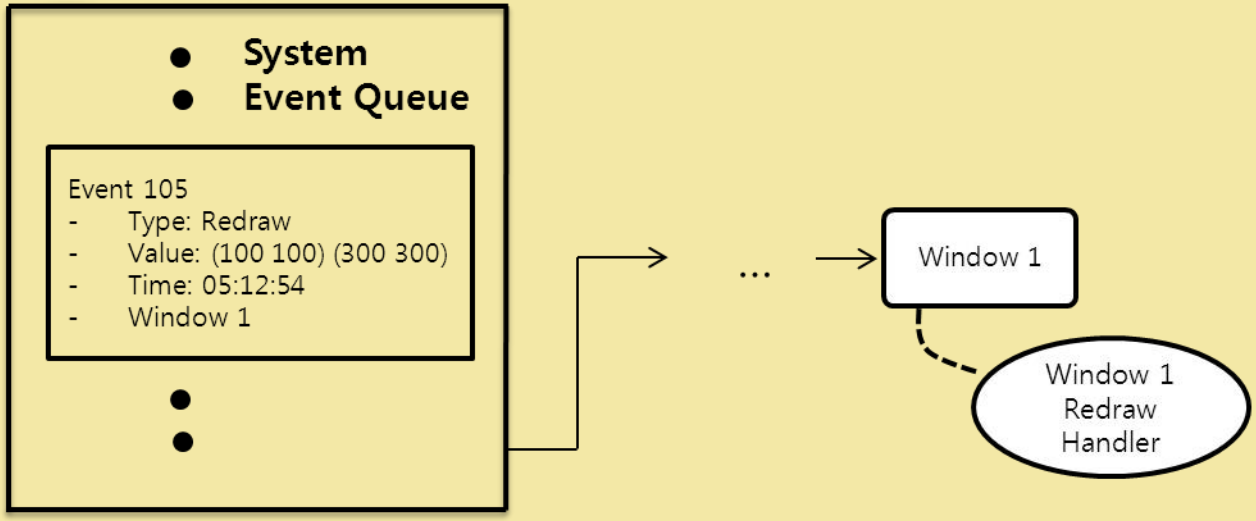
Interactive Software Architecture:
Events, UI Objects and Event Handlers

- The events do not have to necessarily be generated externally by the interaction devices, but sometimes internally for special purposes (these are sometimes called the **pseudo-events**).
  - For instance, when a window is resized, in addition to the resizing itself, the internal content of the window must be redrawn and the same goes for the other windows occluded or newly revealed by it (after resizing).
  - Special pseudo-events are enqueued and conveyed to the respective applications/windows.
  - In the case of resizing/hiding/activating and redrawing of windows, it is the individual application's responsibility, rather than the window manager's, to update its display contents because only the application has the knowledge as how to update its content.
  - Thus a special "redraw" event is sent to the application with information of which region is to be updated.
  - The window might need to be redrawn not because of the window management commands such as resizing and window activation but due to the needs of the application itself.
  - The application itself can generate pseudo-special events for redrawing parts of its own window.

- More generally, UI objects can generate pseudo-special events for creating chain effects e.g. when a scroll bar is moved, both the window content and the scroll bar position have to be updated.



Window 1 in the back exposed to the front



Event driven Architecture

- Event driven program structure generally takes the form of the following structure.
  - The first initialization part of the program creates the necessary UI objects for the application and declares the event handler functions and procedures for the created UI objects.
  - Then, the program enters a loop that automatically keeps dequeing an event from the application event queue, and invoking the corresponding handler (i.e. dispatching the event).
  - Often the development environment hides this part of the program so that the developer does not have to worry about such system level implementations.
  - However, depending on the development toolkits, the user may have to explicitly program this part as well.

Event driven Architecture

Initialize → Define root window; Load info about app; Set things up (not necessarily drawn) → There are different set-ups

While (not time to quit)

{ Get next event E Main loop

Dispatch event E

}

Event handling code is distributed

- **Event queue (usually one for each process)**
  - **Window system provides routine to get next event from the queue**
  - **(OR APP FRAMEWORK REALLY … JAVA, JS/WEB Browser, …)**
  - **But usually main loop just removes events from queue and dispatches for processing**
  - **Filtering may be needed (e.g. to distinguish between "Ctrl-t" and "t")**

- **Event types**
  - **Input: Mouse button, Double Click, Modifiers (Ctrl-Shft), Keyboard, Mouse enter/leave, …**
  - **Windowing: Creation/Destruction, Open/Close, Resize, …**
  - **Redrawing / window refresh (e.g. Handling newly exposed portion of window)**

1. initialize application

- define UI objects
- define event handlers
- …

2. run "system_loop"

```
system_loop ( ) {
 while (event != QUIT) {
  get next event from application queue
  find the target_object by traversing down the UI object tree
  invoke corresponding event handler for this target_object
  refresh screen and send commands to aural/haptic device
  }
} /* system provided event processing loop */
```

1. initialize application

- define UI objects
- define event handlers
- …

2. while (event != QUIT) {

get next event from application queue

find the *target_object* by traversing down the UI object tree

switch (event) {

case mouse_right_button:

/* compute for response behavior to this event for this object */

/* compute for new or changes in visual/aural/haptic/tactile output */

target_object.mouse_right_button_handler (…)

case mouse_left_button:

target_object.mouse_left_button_handler (…)

…

} /* event dispatching */

refresh screen and send commands to aural/haptic device

} /* user implemented event processing loop */

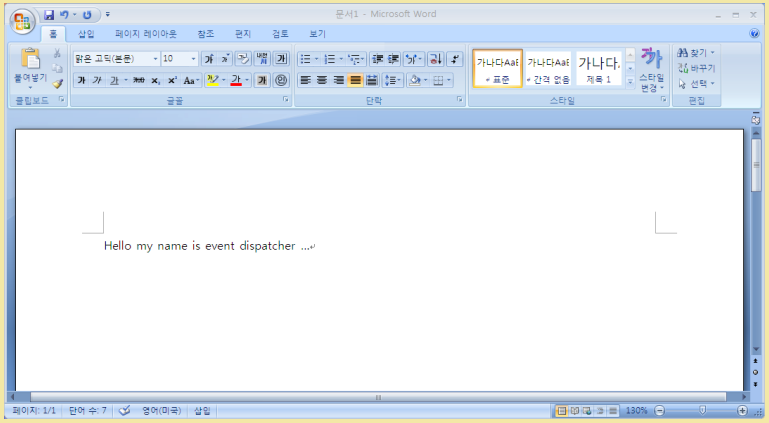
OO Event Dispatching

- For easier development and less debugging time? (or more control)
  - Developers have to sort out types of events and use “case” statements …
- Handle issues of passing messages between independent objects
- Define abstract class for event handling (Event handling class)
  - SetCanvas, SetBounds, GetBounds (setting the areal extent for this event handler)
  - MouseDown, MouseUp, MouseMove, KeyPress
  - MouseEnter, MouseLeave
  - Redraw …

```
Init
While (not time to quit) {
  Get next event E
  W = window to which E is directed;
  WEH = W.myEventHandler;
  switch (E.EventKind) {
     case RedrawEvent: WEH→Redraw(E.DamagedRegion);
     case MouseDownEvent: WEH→MouseDown(E.WhichButton, E.MouseLoc, …);
     …
    }
…
```

Hierarchical OO GUI Toolkit

- Usually a top application class
  - Run method → the infinite loop
  - Init method → Initialization part
  - Quit method → Clean things up upon destruction of application



- Hierarchical UI objects
  - Top window
    - Title bar
    - Horizontal scroll bar
    - Vertical scroll bar
    - Contents area
      - Tool bar
        - Other subwindows
      - Text editing window
    - Grow box
    - …

Remember DOM?

고려대학교

Dispatching events (more details)

- **Basic model:**
  - **Top down search for bottom most object in the UI hierarchy**
  - **Then the event is processed at the bottom most object and that would be it …**
    - **E.g. Select scroll bar, its parent window does not get the event**

- **But suppose you want to use scroll as only indication of the location of your cursor**
  - **Click on scroll does nothing / if I move my cursor in the parent content window, scroll just responds and move**
  - **Event on the parent should (like moving cursor in it) control the child scroll object**
    - **Propagate event downward --> how?**
- **How about if I click on scroll, the parent window should change accordingly?**
  - **Propagate event upward --> how?**
- **Javascript: Bubble and Capture (Event goes down the hierarchy then up)**
  - **Can be caught to be processed at any level**
- **IOS UIKit: Handler (Responder) chain**

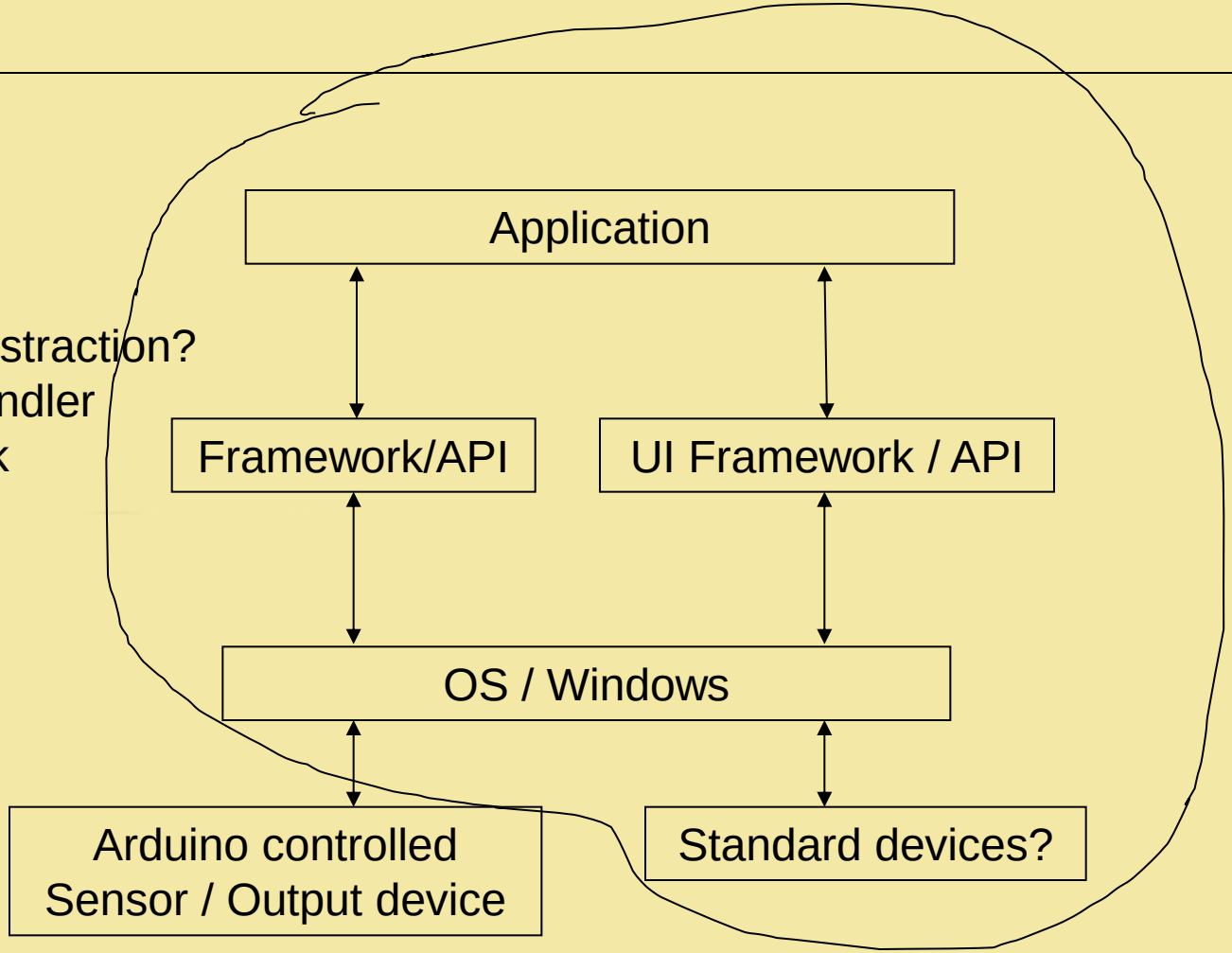
Output Synchronization?

- Usually output modality is visual only
  - E.g. Redraw screen → The event is handled by call to a lower “visual” renderer (by the handler)
- What if we have several modalities: Aural, Haptic, Tactile, …
  - Need Aural, haptic and tactile renderer as part of the event hander
- Handler will look like:

```
…
Update_screen();
Play_sound(“mysound.mp3”, …);
Generate_force (2N, …);
…
```

- Calls are made in sequence … Will they be synchronized? (played at once ?)
  - Maybe, if all of them are fast enough
  - Is this true on all systems?

External modules?



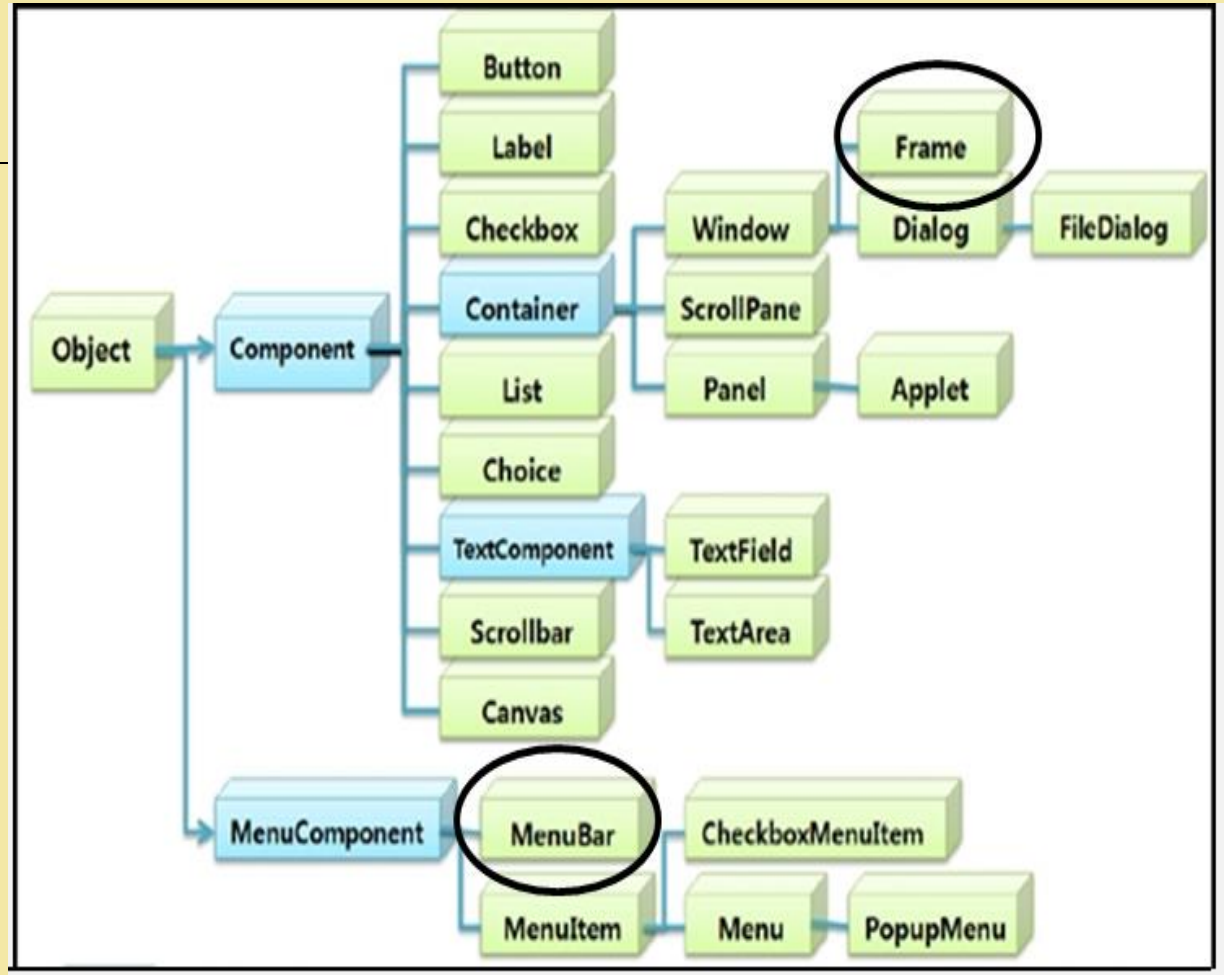
Similar abstraction?
events/handler
framework

This is really another computer (Arduino) …
But we can make it look computer to device instead of computer to computer

UI Toolkits / API

- Library of pre-composed UI objects (which would include event handlers as part of them)
- Predefined set of events which are defined and composed from the lower level UI software layer or the UI execution framework.
- Abstracts out the system details of handling events
  → Programming for interactive software becomes easier more convenient.
- Take the controllable graphical form on a screen, called as the widget (i.e. window gadget).
- Typical widget set includes the menus, buttons, text boxes, images, etc.
  - Just singular or composite (made up of several UI objects in themselves).
- The use of a toolkit also promotes the creation of an interface with a consistent look, feel and mechanism (consistency)

- We examine how events are defined, UI objects created, event handlers specified, and the interface (developed this way) combine with the core functional part of the program.



Overall UI object hierarchy

Java

- Offers a library of object classes called the AWT (Abstract Window Toolkit)
  - Classes useful for creating 2D UI and graphic objects.

- Part of the UI execution framework for Java which handles a (large) subset of interaction events, called the “AWTEvents”
  - In general, the interaction events are sent to Java programs which are captured, abstracted and stored as “EventObject.”
  - The AWTEvents are descendants of the EventObjects which cover most of the useful user interaction events (such as mouse clicks, keyboard input, etc.).

- The AWT framework will map the AWTEvent to the corresponding AWT UI object/handler.

Java: 2 ways of handling events (1)

- Overriding the (predefined) callback methods of the interactive applet object for the events.
- AWTEvent “types” and the corresponding callback methods that can be overridden for customized event handling.

AWTEvent type	Callback Function
mouseDown	mouseDown (Event evt, int x, int y)
mouseUp	mouseUp (Event evt, int x, int y)
mouseEnter	mouseEnter (Event evt, int x, int y)
mouseExit	mouseExit (Event evt, int x, int y)
mouseDrag	mouseDrag (Event evt, int x, int y)
gotFocus	gotFocus (Event evt, Object x)
lostFocus	lostFocus (Event evt, Object x)
keyDown	keyDown (Event evt, int key)
keyUp	keyUp (Event evt, int key)
action	Action (Event evt, Object x)

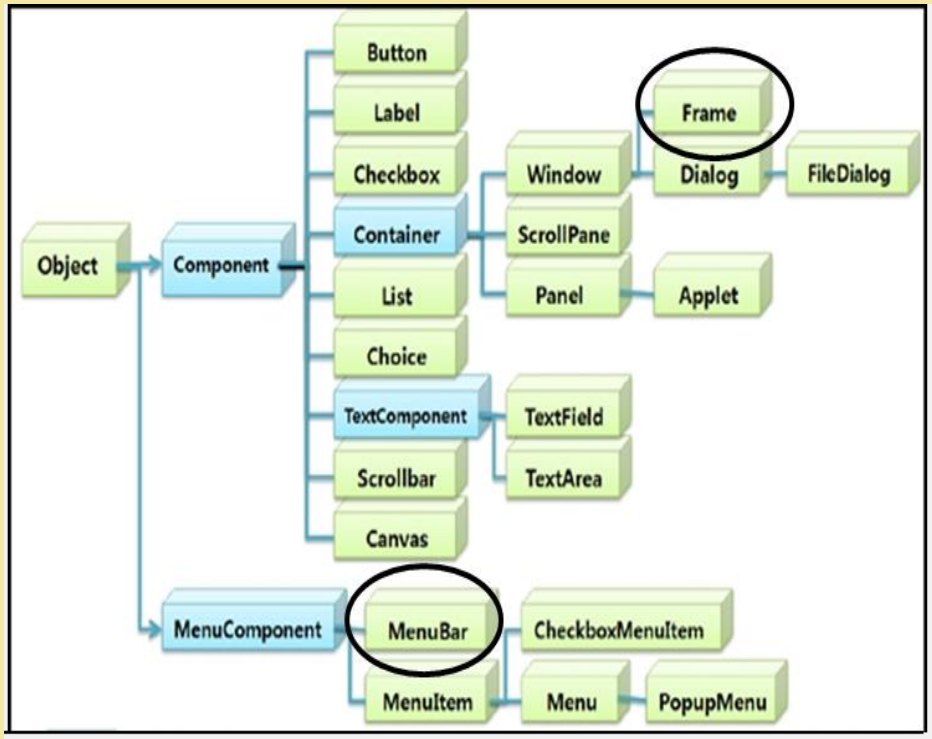
```
import java.awt.*;
import java.applet.*;

public class My_Applet extends Applet
{
        ...

        // Overridden methods for event handling
        public boolean mouseEnter( Event evt, int x, int y)
        {
                draw_object (x, y);  // draw object at mouse clock position

                // Signal we have handled the event
                return true;
        }

        public boolean mouseMove( Event evt, int x, int y)
        {
                repaint()               // repaint whole screen
                draw_cursor (x, y);     // draw new cursor

                // Signal we have handled the event
                return true;

        }

        ...

}
```

An interactive applet with callbacks for mouse events. When the mouse click is entered, the applet draws an object at the click position. When the mouse moves, the whole applet is repainted and a new cursor is drawn at the newly moved position.

Java: 2 ways of handling events (2)

- Individual AWT UI object is to be registered with an event listener that waits for and respond the corresponding event

- The “Component” is the most bare and abstract UI class from which other variant UI objects derive from.
  - For instance, a window, button, canvas, label and others are descendants (subclasses) from the Component class.
  - The “Window” class has further subclasses such as the “Frame” and “Dialog.”
  - Each class has basic methods, e.g. a window has such like resizing, adding sub-elements to it, setting its layout, moving to a new location, showing or hiding, etc.



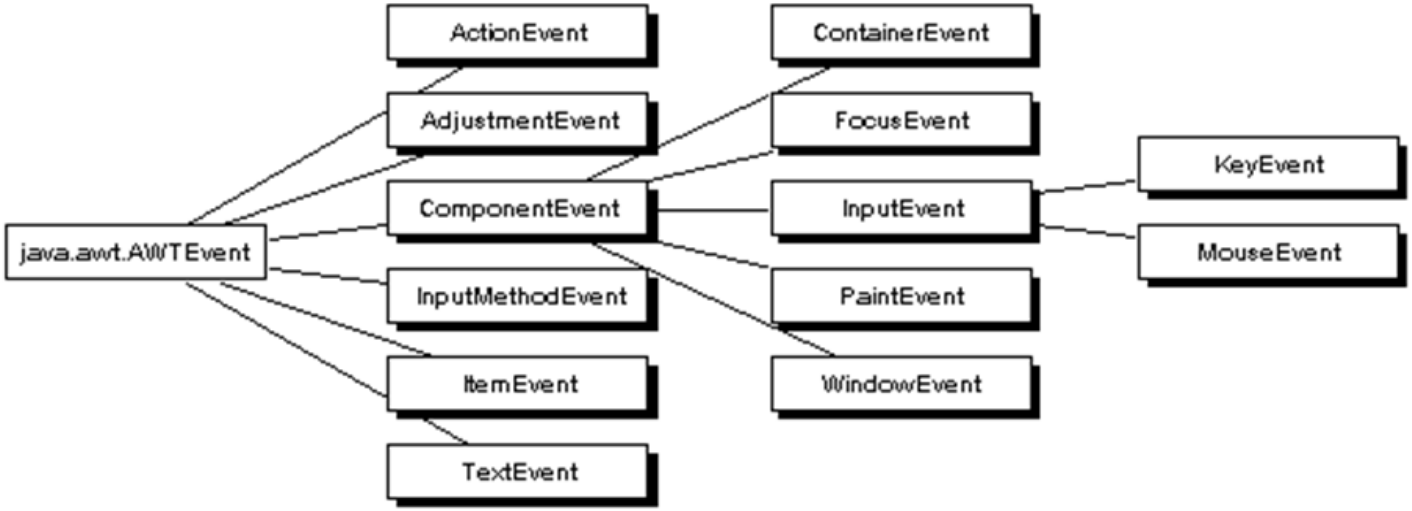

```
/* create a frame, a window with a menu bar */
my_frame = new Frame("my frame");

my_frame.show();                    /* display it */
my_frame.resize( ... );             /* resize it  */

/* create a menu bar */
menuBar mb = new MenuBar();

/* set the menubar for the frame */
my_frame.setMenuBar(mb);

...
```

Overall UI object hierarchy and an example of the codes for creating a frame (a window with a menu bar) and setting some of its properties by calling of such methods.

Events

- All the events derive from an abstract “EventObject” and offer basic methods for retrieving the object associated with the event and accessing event type and id.
- Descendant event classes possess additional specific attributes and associated methods for accessing the values.
  - For instance, the “KeyEvent” has a method called “getkeyChar” that returns the value of the keyboard input
  - “MouseEvent” has a method called the “getPoint()” and “getClickCount()” that return the screen position data at which the mouse event occurred and the number of clicks.

- For more detailed information, the readers are referred to the reference manual for Java AWT [ref].



Event Class	Description / Examples
ActionEvent	Button press, double click on an item, selection of a menu item
AdjustmentEvent	Scroll bar movement
ComponentEvent	Hiding/revealing a component, component movement and resizing
FocusEvent	Component gaining or losing a focus
KeyEvent	Keyboard input
ItemEvent	Check box selection/deselection, menu item selection,
MouseEvent	Mouse button, mouse moving, mouse dragging, mouse focus
TextEvent	Text entry
WindowEvent	Window opening and closing, window de/activation, de/iconification

Java Listeners

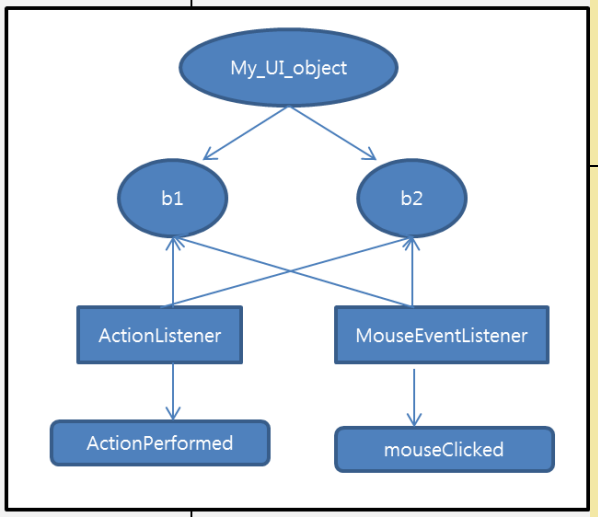
- The event handlers in Java AWT are known as the “listeners.”
- Background process that listens for the associated events for a given UI object and responds to them.
- It is an abstraction of the event processing loop (we have already discussed) assigned to a UI component for each various input events.
- Thus the listeners must be “registered” for various events that can be taken up by the given UI object.
- As a single UI object may be composed of several basic components and potentially receive many different types of input events, and as such listeners for each of them will have to be coded and registered.
- Such a UI object is modeled as a collection of listeners through the Java “implementation interface” construct
- To be precise, a “listener” is differentiated from a simple callback function because it is a process that waits for and react to the associated event, while a callback function is just the procedure that reacts to the event.

Event listeners

- Just like the event component hierarchy, the event listener interface also is structured correspondingly.
- A UI object, possibly composed of several basic UI components, is designated to react to different events by associating the corresponding listeners with the UI object by the “interface implementation” construct.
- That is the UI object is declared to “implement” the various necessary listeners and in the object initialization phase, the specific components are created and listeners registered.
- The class definition will thus include the implementation of the methods for the registered listeners.

Event Class	Corresponding Listener Description	Samples of Derived Methods
ActionEvent	ActionListener	actionPerformed
AdjustmentEvent	AdjustmentListener .	adjustmentValueChanged
ComponentEvent	ComponentListener	componentHiddencomponentMoved componentResized
FocusEvent	FocusListener	focusGained focusLost
KeyEvent	KeyListener	keyPressed keyReleased keyTyped
ItemEvent	ItemListener	itemStateChanged
MouseEvent	MouseListener	mouseClicked mouseEntered mouseExited mousePressed mouseReleased
MouseMotionEvent	MouseMotionListener	mouseDragged mouseMoved
WindowEvent	WindowListener	windowOpened windowClosed windowActivated windowDeactivated windowDeiconified windowIconified

```
public class my_UI_object extends Applet
implements ActionListner, MouseListener {

        public void init () {
         /* this object has two components */
         Button b1 = new Button("One");
         Button b2 = new Button("Two");

         /* register different event listeners */
         b1.addActionListener (this);
         b2.addActionListener (this);

         b1.addMouseEventListener(this)
         b2.addMouseEventListener(this)

         add(b1);
         add(b2);
        }

        /* The listener codes */
        public void actionPerformed (ActionEvent ae) {

         /* specify actions to for ActionEvent (button press)      */
         /* access ae and find out which button is pressed         */
         /* make proper response  depending on which button        */
         ...
        }

        public void mouseClicked (MouseEvent me) {


         /* specify actions to for MouseEvent (mouse click)        */
         /* access me and find out which kind of mouse event       */
         /* make proper response  depending on which button        */
         ...
        }

        ...
}
```



An example of a UI object as an event listener implementation with some of its associated methods is shown in Figure. That is, "my_UI_ojbect" is declared as an extension of an "Applet" and at the same time as an implementation of two listeners (reacting to three types of events): the ActionEvent, and MouseEvent. It also has two button components created in the init(). Two listeners are also registered to each of the buttons (b1 and b2) in the same init() method. The method implementations follow.

```html
<div class="parent">
  <div class="child"></div>
</div>
```

```js
parent.addEventListener("click", () => {
  console.log("Parent")
})

child.addEventListener("click", () => {
  console.log("Child")
})
```

Click in child --> Print "Child"

AND

Print "Parent"

Event Bubbles back up
(all the way to top of DOM)

고려대학교

```
parent.addEventListener("click", () => {
  console.log("Parent Bubble")
})

parent.addEventListener(
  "click",
  () => {
    console.log("Parent Capture")
  },
  { capture: true }
)

child.addEventListener("click", () => {
  console.log("Child Bubble")
})

child.addEventListener(
  "click",
  () => {
    console.log("Child Capture")
  },
  { capture: true }
)
```

But actually, before bubbilng up from the child …

~~It starts from the top and goes down,~~ then bubbles up …

Propagation can be stopped too

Still difficult to delegate control to someone other than who is in the other part of the UI tree ...

```
Parent Capture
Child Capture
Child Bubble
Parent Bubble
```

고려대학교

Android UI Framework/Toolkit

- Android is based on Java and so is the UI framework
- Android UI Toolkit includes a declarative method for UI development
- Events in Android can take a variety of different forms, but are usually generated in response to bare and raw external actions, such as touch and button input.
- Multiple or composite “higher” level events may also be internally recognized and generated such as touch gestures (e.g. flick, swipe) or virtual keyboard inputs.
- The Android framework maintains an event queue into which events are placed as they occur. Events are then removed from the queue on a first-in, first-out (FIFO) basis.
  - Input event is passed to the “View” either by the location on the screen where the touch took place or by the current focus (event notification)
    - UI object classes in Android derive from what is called the “View” object
  - View is also passed a range of information (depending on the event type) about the nature of the event (e.g. coordinates of the point of contact)

Handlers in Android

- Similarly to the case of Java AWT, there are two major ways to define the reactive behavior to these events
  - Override the default callback methods of the “View” interactive class object for input events.
  - The second method is to associate an event listener with the “View” object.”
- The Android View class, from with all user interface components are derived, contains a range of event listener interfaces, each of which contains an abstract declaration for a callback method.
- In order be able to respond to an event of a particular type, a view must register the appropriate event listener and implement the corresponding callback.
  - For example, if a button is to respond to a click event, it must both register the View.onClickListener event listener (via a call to the target view’s setOnClickListener() method) and implement the corresponding onClick() callback method.
  - When a “click” event is detected on the screen at the location of the button view, the Android framework will call the onClick() method of that view when that event is removed from the event queue.

```
public class My_Activity extends Activity {

   /** Called when the activity is first created */
   @Override
   public void onCreate (Bundle savedInstanceState) {
       ...
       /* create a view object */
       View vw = new View(this);

       /* associate a listener */
       vw.setOnTouchListener (my_touchListener);

       ...
   }

   /* extend the listener class and implement the handler method */
   class MyTouchListenerClass implements View.OnTouchListener {

       public boolean onTouch (View v, MotionEvent event) {
         ...
         if(event.getAction() == MotionEvent.ACTION_DOWN) {
           return true;
         }
         return false;
       }
   }

   /* create an instance of MyTouchListenerClass */
   MyTouchListenerClass my_touchListener = new MyTouchListenerClass();

  ...

}
```

Registering and implementing an event listener by implementing the event listener itself and associating it with the view object (which is often part of the Android "Activity" object)

In this example, "My_Activity" creates a view object called the "vw" and also creates, "MyTouchListenerClass" by extending and overriding the "View.OnTouchListener." "my_touchListener" is instantiated and registered to the view object "vw."

```
public class My_Activity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        MyView vw = new MyView(this);
        vw.setOnTouchListener(vw);
        ...
    }

    protected class MyView extends View implements View.OnTouchListener {
        ...

        public boolean onTouch (View v, MotionEvent event) {
            // TODO Auto-generated method stub
            return false;
        }
    }

    ...
}
```

Registering and implementing an event listener by having the "view" object implement the event listener

"My_Activity" creates "MyView" class by implementing the "View.OnTouchListner" and instantiates a "MyView" object "vw." The listener is registered by calling the "setOnTouchListener" on itself.

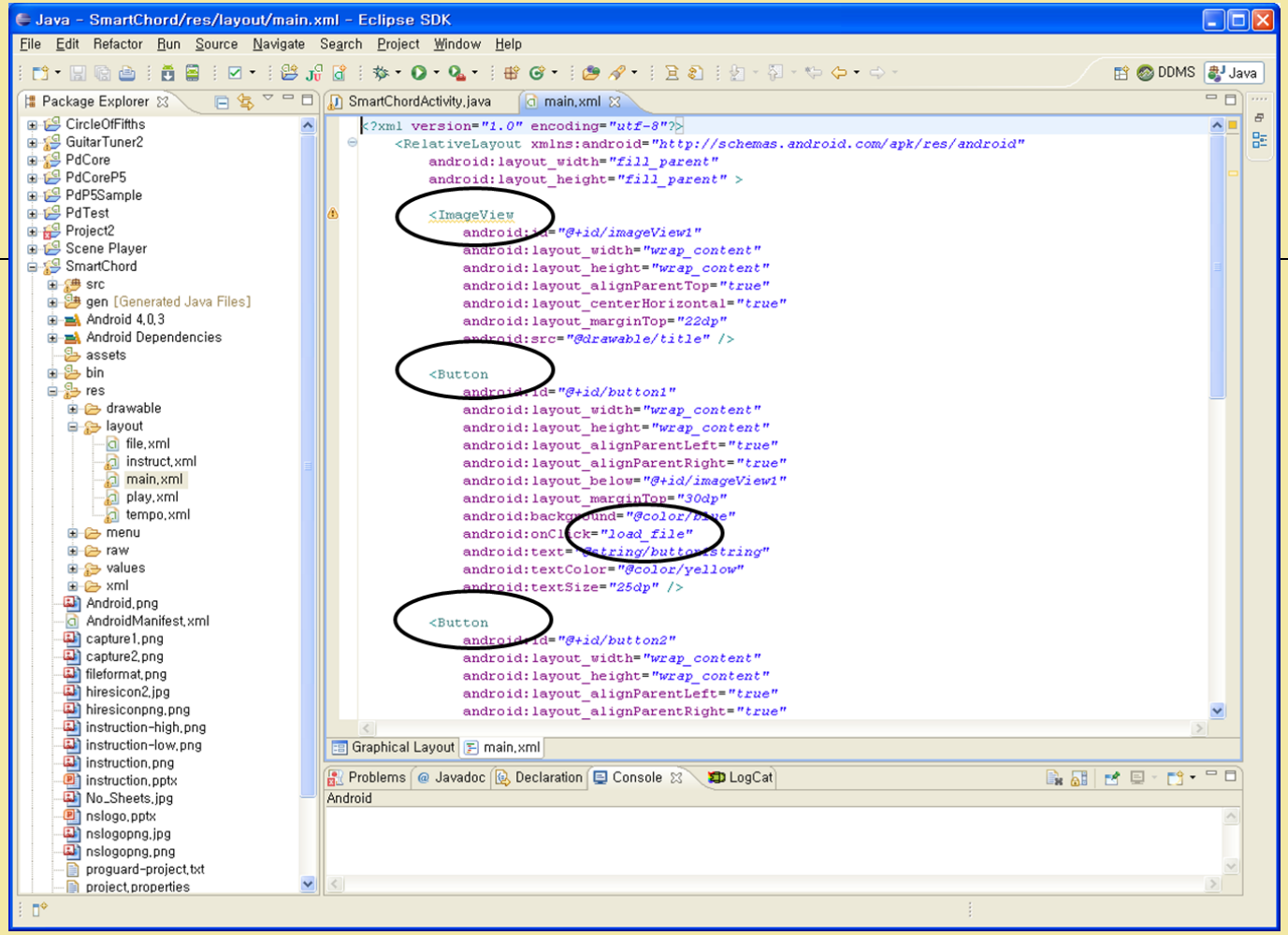
```
public class My_Activity extends Activity implements View.OnTouchListener {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        View vw = new View(this);
        vw.setOnTouchListener(this);
        ...
    }

    public boolean onTouch (View v, MotionEvent event) {
        if(event.getAction() == MotionEvent.ACTION_DOWN) {
            return true;
        }
        return false;
    }

    ...
}
```

Registering and implementing an event listener by having the top-most "Activity" object (which houses various view objects as its parts) implement the event listener.

"My_Activity" implements the View.OnTouchListener and thus includes and overrides the "onTouch" handler.

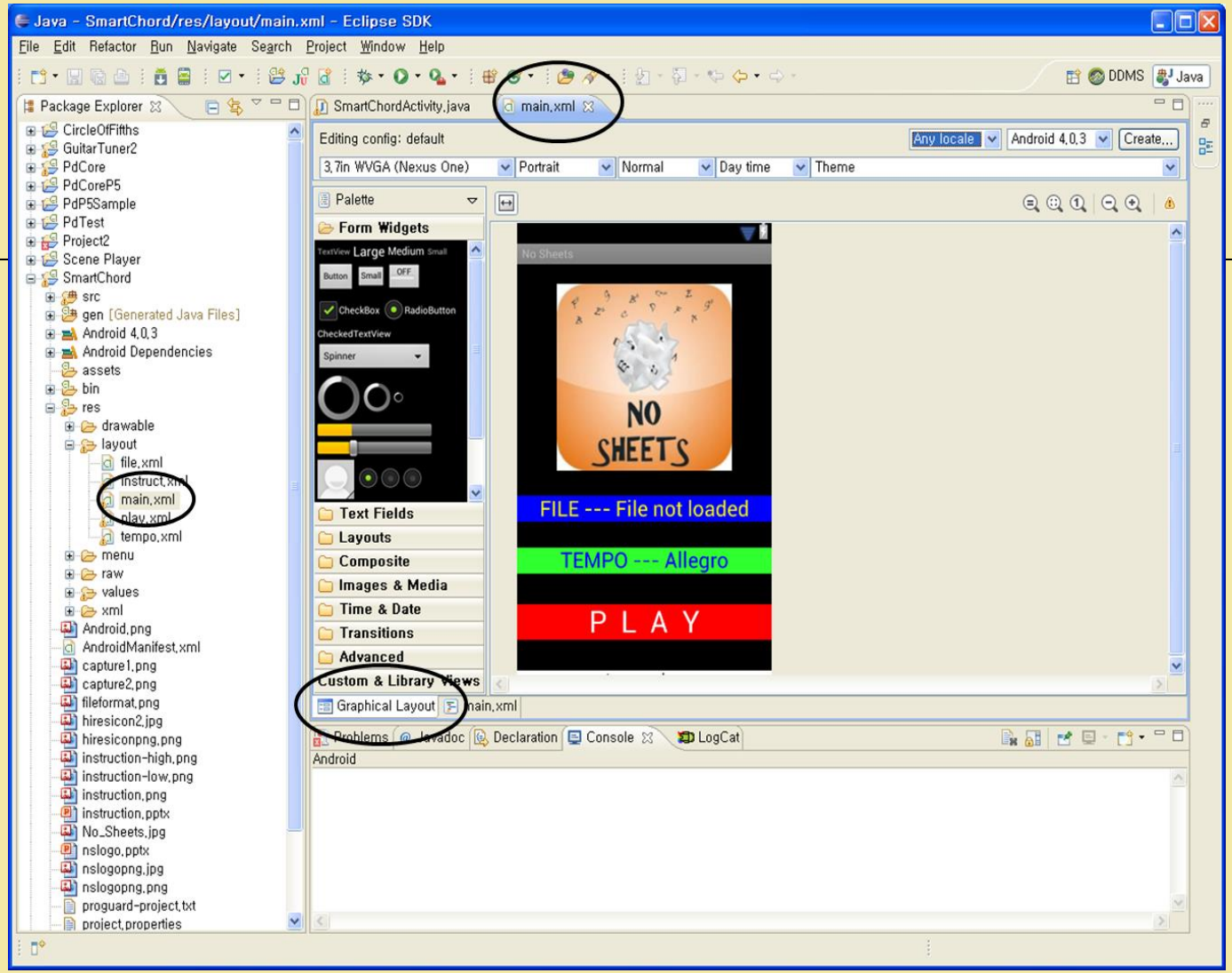
Declarative/Graphical method

- Android UI framework also provides a “declarative” method for specifying the UI.
- The form of the UI can be “declared” using a mark-up language.
- Through a development tool such as Eclipse, the UI can be built through direct graphical manipulation as well.
- In Eclipse, the de facto development tool for Android applications, any methods can be used and three methods can be used even simultaneously.
- The corresponding UI to the declarative UI specification can be displayed in the graphic form and be manipulated.
  - When a UI component is added, deleted, attribute values changed, such actions are reflected back to the respective representations, graphic or declarative.
  - E.g. the declarative specification names the event handler and other attribute values for the components in more exact terms.
  - The handler code is implemented in the corresponding programmatic representation part.

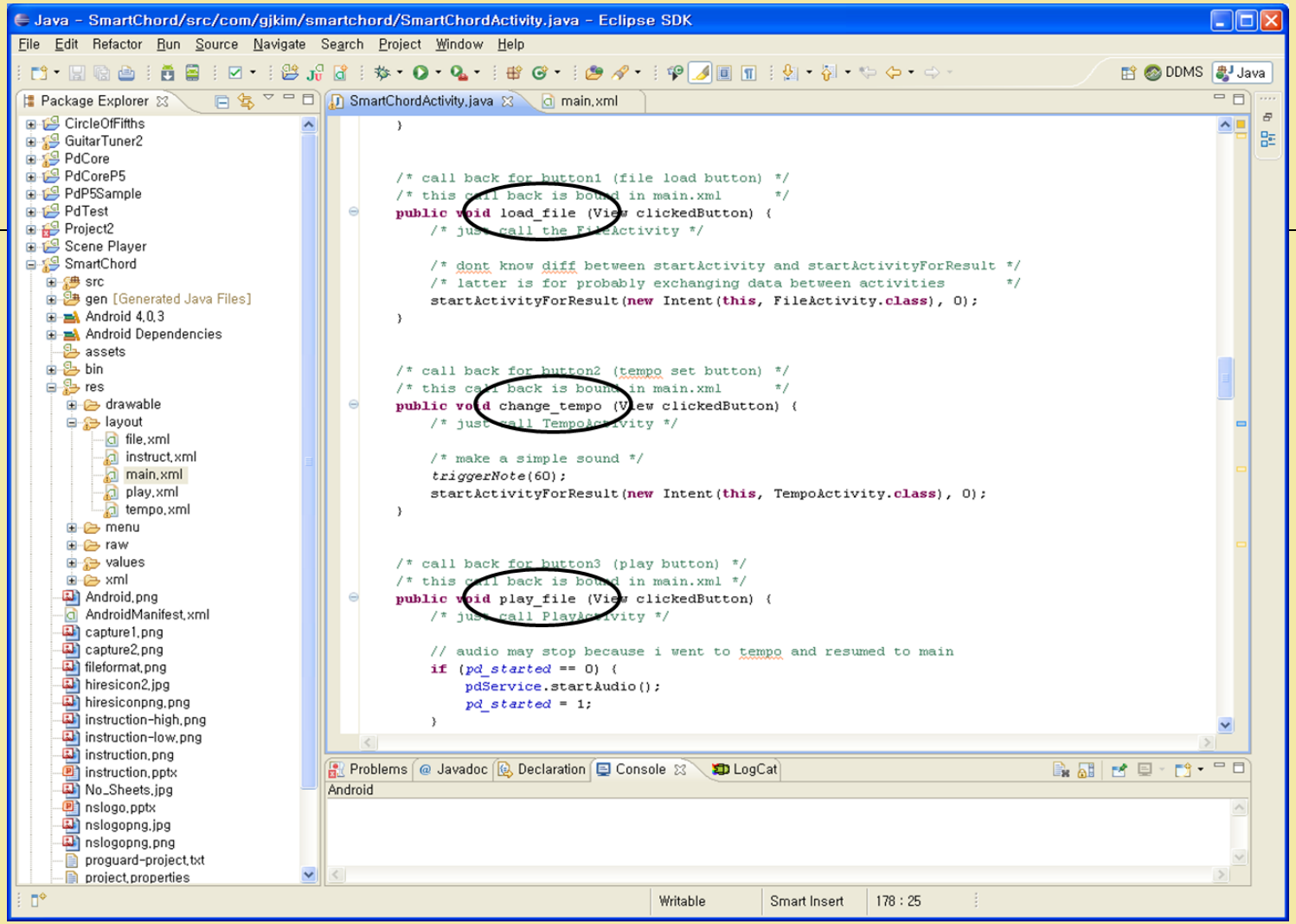


An example of a declarative specification of the UI (for the “No Sheets” application (see Chapter X). The declarative specification is saved in an XML file with element constructs for UI components such as image, buttons and etc. Note that the file names the elements (for later referral in the program) and one of the attribute of the UI object is the UI handler name (e.g. the encircled “load_file”).

고려대학교



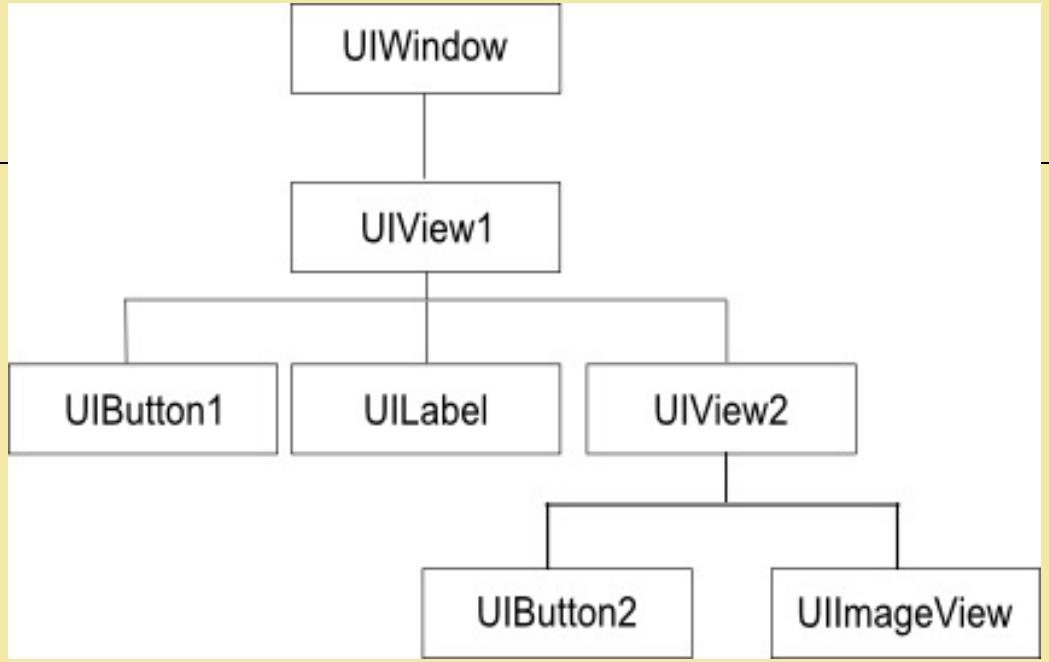
An example of a graphic specification of the UI (for the “No Sheets” application (see Chapter X). The graphically displayed UI is consistent with the corresponding declarative representation (saved in the “main.xml”).



An example of a programmatic specification of the UI (for the “No Sheets” application (see Chapter X). Above shows the code implementation for the UI handers (e.g. “load_file”) defined in the declarative specification.

iOS UI Framework and Toolkit

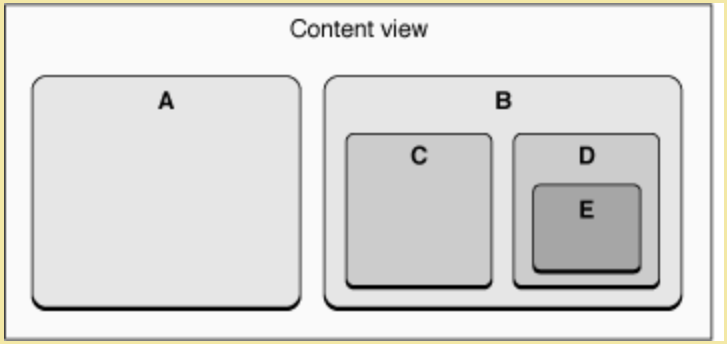
- Three major types of “discrete” events for iOS:
  - Multi-touch / Motion
  - Remote Control (discrete events from external device such as remote controlled headphones).
- The iOS generates low level events when users touch views of an application.
- The application sends these discrete events as UIEvent objects, as defined by the UIKit framework, to the view (i.e. specific UI component) on which the touches occurred.
- The view analyzes the touch event and responds to them.
- Touch events can be combined to represent higher level gestures such as flicks and swipes.
- The given application uses the UIKit classes for gesture recognition and responding to such “recognized” events.
- For continuous stream of sensor data such as those from accelerometers or gyroscopes, a separate “Core Motion” framework is used.
- Nevertheless, the mechanism is still similar in the sense that that sensor data, abstract event or recognized event is conveyed from iOS to the application then to the particular view according to the hierarchical structure of the application UI.

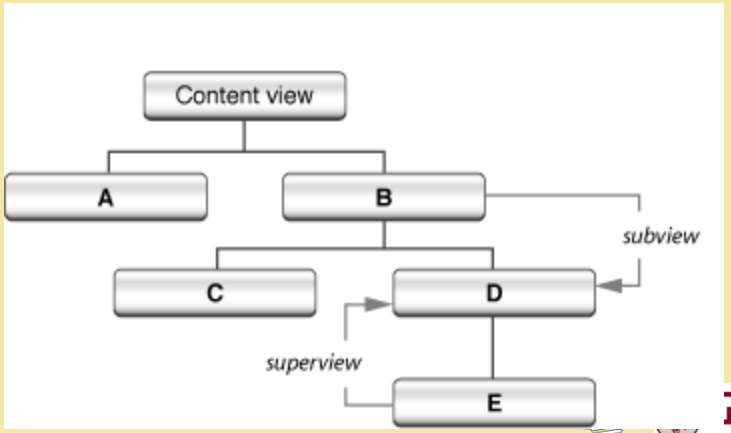


iOS UIKit object hierarchy example

UI Framework Mechanism

- When users touch the screen of a device, iOS recognizes the set of touches and packages them in a UIEvent object that it places in the active application's event queue.
- If the system interprets the shaking of the device as a motion event, an event object representing that event is also placed in the application's event queue.
- The singleton UIApplication object managing the application takes an event from the top of the queue and dispatches it for handling.
- Typically, it sends the event to the application's key window (the window currently in focus for user events) and the window object representing that window sends the event to an initial object for handling.





UI object hierarchy

UI Framework Mechanism

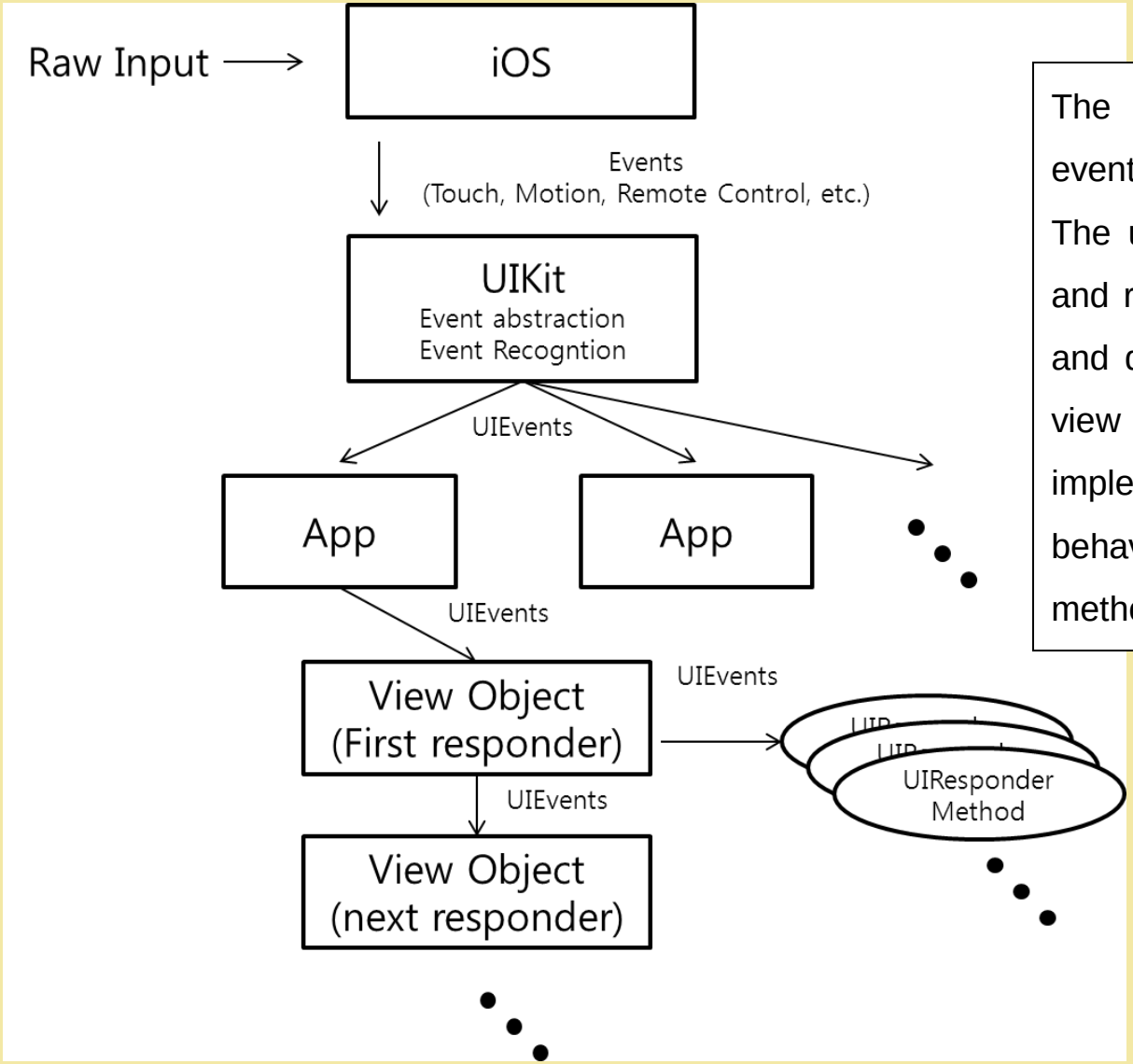
- That object is different for touch events and motion events.
  - For touch events, the window object uses hit-testing and the “responder” chain to find the view to receive the touch event.
    - In hit-testing, a window calls hitTest:withEvent: on the top-most view of the viewhierarchy; this method proceeds by recursively calling pointInside:withEvent: on each view in the view hierarchy that returns YES, proceeding down the hierarchy until it finds the subview within whose bounds the touch took place. That view becomes the hit-test view. If the hit-test view cannot handle the event, the event travels up the responder chain until the system finds a view that can handle it.
  - For Motion and Remote Control events, the window object sends each shaking-motion or remote-control event to the first responder for handling.
- Although the hit-test view and the first responder are often the same view object, they do not have to be the same.

Responder (Handler)

- A responder object is an object that can respond to events and handle them.
- UIResponder is the base class for all responder objects, also known as, simply, responders.
- It defines the programmatic interface not only for event handling but for common responder behavior.
- UIApplication, UIView, and all UIKit classes that descend from UIView (including UIWindow) inherit directly or indirectly from UIResponder, and thus their instances are responder objects.
- The first responder is the responder object in an application (usually a UIView object) that is designated to be the first recipient of events other than touch events.
- A UIWindow object sends the first responder these events in messages, giving it the first shot at handling them.

Responder (Handler)

- If the first responder or the hit-test view doesn't handle an event, UIKit may pass the event to the next responder in the responder chain to see if it can handle it.
- The responder chain is a linked series of responder objects along which an event is passed.
- It allows responder objects to transfer responsibility to other, higher-level objects.
- An event proceeds up the responder chain as the application looks for an object capable of handling the event.
- The response behavior itself is implemented by the responder objects (i.e. the UI components such as the window, button, slider, etc.).
- For instance, a responder object methods for "Multi-Touch" events include the following:
  - (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event : when one or more fingers touch down on the screen
  - (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event : when one or more fingers move
  - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event : when one or more fingers lift up from the screen
  - (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event : when the touch sequence is cancelled by a system event (such as an incoming phone call).



The event processing flow and the event driven object behavior structure. The user input is captured, abstracted and recognized by the UIKit framework and queued into the proper application view objects (or responders) who implement the particular response behaviors using the UIResponder methods.

Homework circa 2020

Implement a tree data structure using Javascript. Tree data structure should be a tree of simple nodes with id number, arbitrary number of children. Set of basic methods/procedures such as: create_root, add_node, delete_node, search_node, etc.

Extend the node structure to contain a “primitive” graphic element information like a circle, line, rectangle, point, etc. We will call these “graphic nodes. Different graphic elements will need different attributes (only include the very basic ones). Example graphic node might look like:

Node1234	
Type	Line
Start point	(100, 100)
End point	(200, 200)
Pointer parent	Node123
Pointer to children list	Node12345

Homework circa 2020

Modify the tree ADT so that the graphic UI nodes (window, button, text box, label (text box without border), check box, etc.) can be managed (be able to create/add/delete/search different types of graphic UI nodes). For instance, you might define a GUI APIs such as “tree_node create_window (ID, position_x, position_y, width, height, title)” that can be called from the script to return a node addable to the tree.

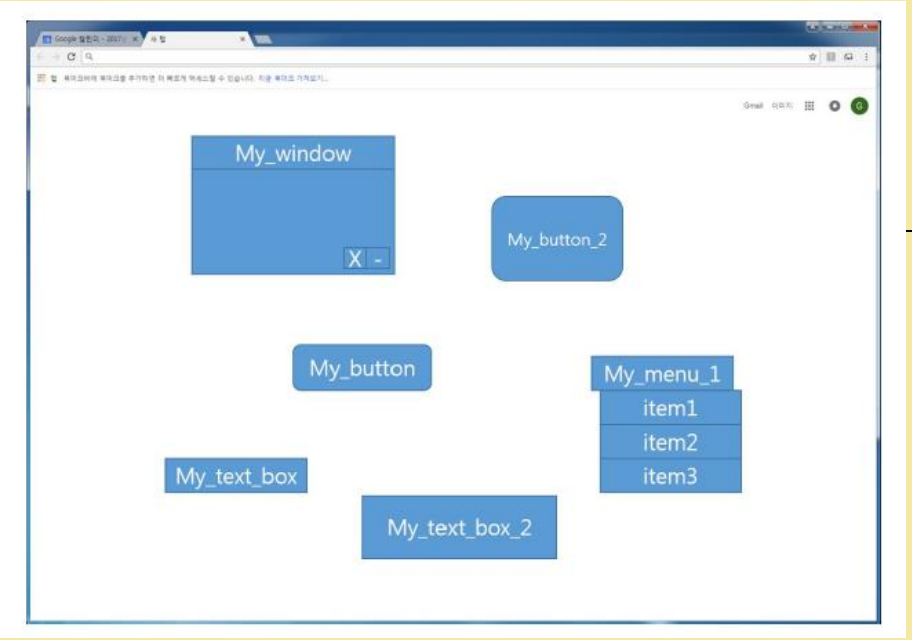
The GUI objects can be thought to be made of as combinations of the primitive graphics elements (e.g. button = rectangle + text, window = rectangle + rectangle). The corresponding node will have to have attributes as necessary (e.g. window title, background color, text color, (just the basics). An example GUI node might now look like this:

Homework circa 2020

Node1234	
Type	Elliptic Button
Position	(100, 100)
Height	10
Width	30
Text	“Press”
Color	Blue
Pointer parent	Node123
Pointer to children list	Node12345

Node1234	
Type	Window
Position	(100, 100)
Height	100
Width	50
Title	“My Window”
BG Color	White
Pointer parent	Node123
Pointer to children list	Node12345

Create an example of hierarchy of GUI objects and print their structure recursively (e.g. in depth first or breath first manner).



Render them using HTML Canvas …
(or other graphics package)

Further extend the tree-graphics program in the following way. Write and demonstrate a routine that takes a mouse input and identifies the object that is clicked (e.g. print message or highlight the object, see below). If there are multiple objects that correspond to the location of the mouse click, the lowest in the tree must be identified. In order to do this, one will have to define the rectangular areal dimension of the GUI object. When there are multiple overlapping GUI objects, the lowest level object should be the one selected. Also note that the objects are now interactable.

Development Process/Methodology

- Interactive application
  - Core functions
  - UI
- A **development framework** refers to a modular approach for interactive program development where the core computational and interface parts are developed in a modularized fashion and combined in a flexible manner.
- Such a development framework is often much based on the UI toolkit which provides the abstraction for the interface parts.
- For one, the framework allows the concept of plugging in different interfaces for the same model computation and easier maintenance of the overall program.

Model

- Part of the application corresponds to the computation (e.g. realized as objects) that deals with the underlying problem or main information or data of the application.
  - Information that the application is trying to manipulate
  - This is the data representation of the real world objects in which the user is interested
- For all practical purposes, once in place, “Model” of the application tends to be stable and unchanging.
  - For instance, in an interactive banking application, the “Model” will be parts of the program that maintains the balance, computes the interest, make wire transfers and etc.
- The Model has no knowledge of how the central information will be presented to the user (output), or how the transactions (input) are handled.

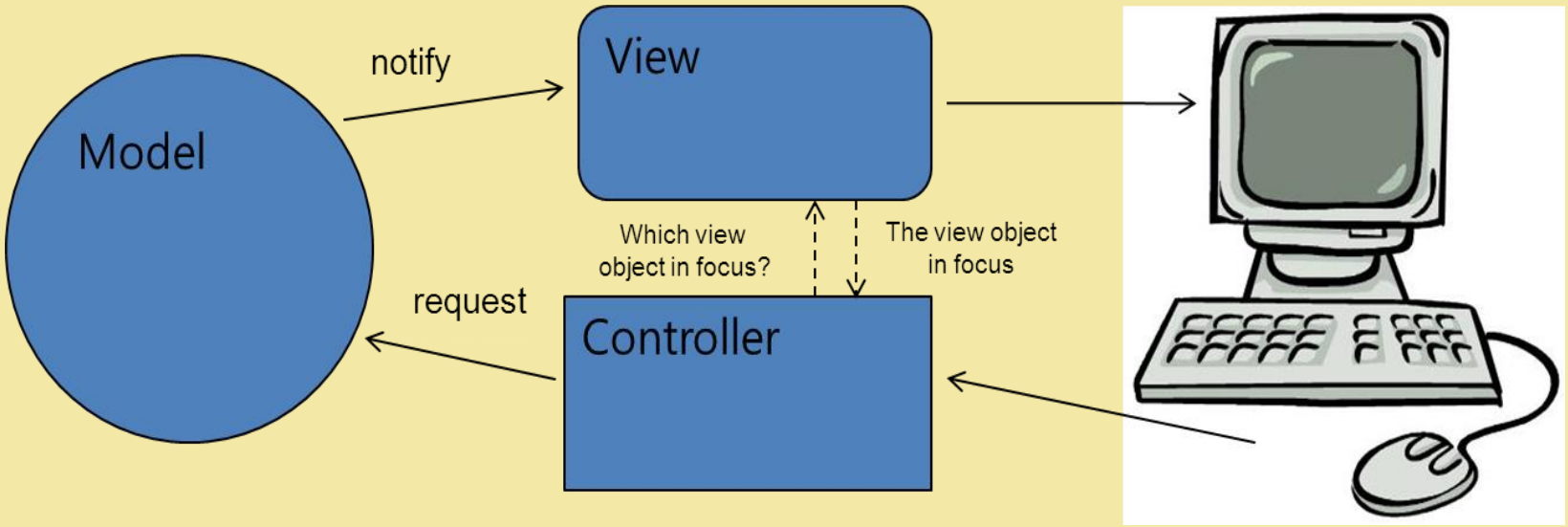
View

- Part of the application corresponds to the implementation for output and presentation of data.
  - Implements a display of the model
  - In modern GUI based interfaces, the implementation will typically be consisted of widgets.
    - For instance, views might be windows and widgets that display the list of transactions and balance of a given account in a banking application, or playing of a background audio clip depending on the score level for a game.
- As a whole, there may exist multiple views for a single application (or model).
  - For instance, there could be different view implementations for different display platforms or user groups (e.g. 17 inch monitor, 10 inch LCD, HD resolution display, display with vibro-tactile output device, young users, elderly users). Note that the output display does not necessarily have to be visual.
  - Anytime, the model is changed each view of that model must be notified so that it can change the display representation of the model on the screen
  - Region of the screen that is no longer consistent with the model is called “damaged” (reporting of damaged portion is very important)
    - Often times it is be too tedious to update just the damaged part of the display upon change of information in the model. The practical approach is to redraw the entire content of the “smallest” widget that encompasses the damaged region or redrawing the entire window.

Controller

- Part of the application corresponds to the implementation for manipulating the view (in order to ultimately manipulate the (internal) model.
  - It takes external inputs from the user, interprets and relays them to the model.
  - Receives all of the input events from the user and decides what they mean and what should be done
  - Controller thus practically takes care of the input part of the interaction.
  - It uses the interaction framework or underlying operating system to achieve this purpose (while the view is mostly independent from the operating system or platform).
- In order to find the object in focus (which visual object is to be manipulated on behalf of the model), the controller must communicate with the view objects (UI Framework stuff).
- In addition, the controller might also change the content of the display without changing the model sometimes.
  - For instance, if the user wanted to simply change the color of a button (e.g. for UI customization purpose), the controller can directly communicate with the view to achieve this effect.
- Once the object in focus is identified, the corresponding event handler would be invoked.
- The controller will only **“relay” a query** or message for certain change or manipulation to happen to the model rather than making the change itself.

Model-View-Controller Architecture



Model-View-Controller Architecture

- The MVC approach was first proposed as a computational architecture for interactive programs (rather than a methodology) by the designers of the programming language called SmallTalk.

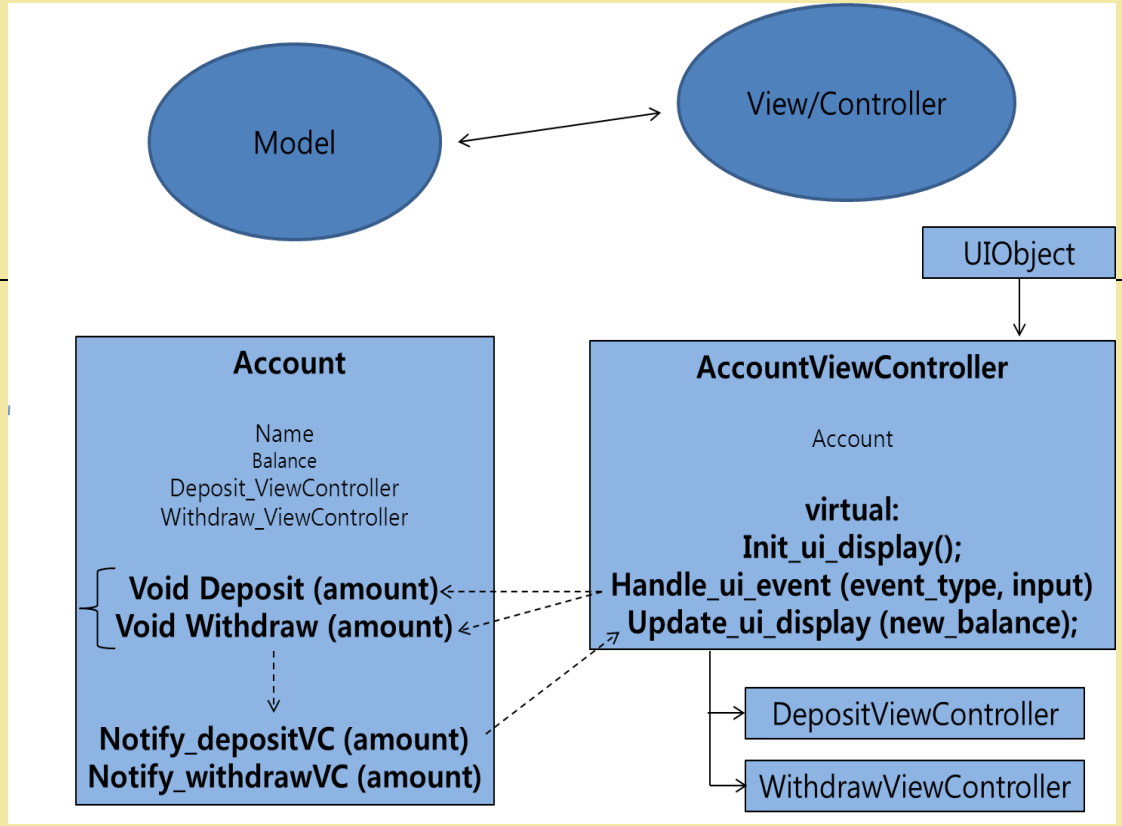
- The MVC architecture or development methodology makes it much easier, particularly for large scale systems, quickly explore and implement and modify various different user interfaces (view/controller) for the same core functional model. This is based on the famous software engineering principle, the separation of concern.
- For small systems, some of them may be merged but generally not a good idea
- In many application architectures, the view and controller may be merged into one module or object, because they are so tightly related to each other (View-Controller).
  - For instance, a UI button object will be defined by display parameters such as its size, label (e.g. which comes from the model), color, and also the event handler which makes invokes the methods on the model for change or manipulation.
- Multiple models and multiple views possible

Visual C++

- View/Document Architecture
- Drawing is handled by the CDC class (device context)
- CView ~ corresponds to View
- CDocument ~ corresponds to Model

Example: Bank Application (M-VC)

- Model maintains the balance for a user who can make deposits or withdrawals through a computer user interface.
  - A class called “Account” maintains and has the customer name, balance and (pointers to) two view/controllers (one for displaying the balance and realizing the UI for making deposits, and the other for withdrawal). → *kind of two “sub” views*
  - The model has two core methods for maintaining the correct balance when a deposit (*Account::Deposit*) or withdrawal (*Account::Withdrawal*) is made.
  - These two methods use the *Notify_depositVC* and *Notify_withdrawalVC* to notify the corresponding view for updating the balance in the display.
- View and Controller parts are merged into one class, called the *AccountViewController*
  - which has a pointer to the corresponding model object (as the recipient of the notifications and model change queries).
  - This class is a subclass of a more general *UIObject* that is capable of housing constituent widgets and reactive behavior to external input.
  - “*DespositViewController*” and “*WithdrawalViewController*” implements two VCs




```
Account :: Deposit (amount) {
 ...
 // update balance by amount
 balance = balance + amount;

 Notify_DepositVC (balance);
 ...
}
```

```
Account :: Notify_DepositVC (new_balance) {
 ...

 Deposit_ViewController.update_UI_display (new_balance);
 ...


}
```

```
Account :: Withdraw (amount) {
 ...
 // update balance by amount
 balance = balance – amount;
 if balance < 0 then balance = 0;

 Notify_WithdrawVC (balance);
 ...
}
```

```
Account :: Notify_WithdrawVC (new_balance) {
 ...

 Withdraw_ViewController.update_UI_display (new_balance);
 ...


}
```

- The subclasses, among others, implement three important virtual methods
  - "*Init_ui_display*" : creating and initializing the display and UI objects within the view/controller
  - "*Update_ui_display*" : updating the display (invoked by the notification method from the model)
  - "*Handle_ui_event*" : handling the user input.

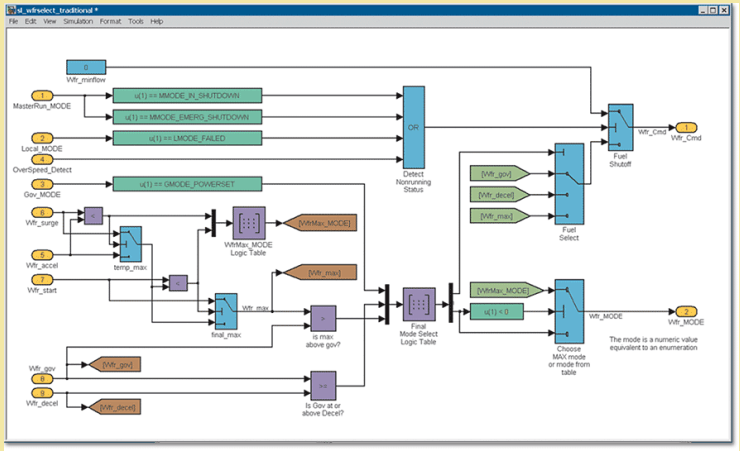
- "*Handle_ui_event*" method of the *DepositViewCotnroller* which interprets the user input (e.g. textual input of digits into integers) and invokes the model method to e.g. make a deposit by calling *my_model->Deposite(deposit_amount)*. Understand that this will eventually change the model and the view/controller will be notified to change its display (e.g. to show the proper amount of balance after the deposit). Although not shown, the "*WithdrawalViewController*" would be coded in a similar manner.

```
DepositViewController :: AccountViewController {

 ...

 // Implement the subclass specific virtual methods

 // create and initialize UI display (the view) for deposit
 void Init_ui_display ();

 // redraw the UI display with new_balance
 void Update_ui_display (new_balance);

 // handle events from user (the controller)
 void Handle_ui_event (event_type, input);

}
```

```
DepositViewController :: Handle_ui_event (event_type, input) {

 ...

 // get the model
 my_model = Get_model ();

 // if the user input is text input
 if (event_type == TEXT_INPUT) {
            // convert the text input to integer
            deposit_amount = convert_to_int (input);
            // make a request to model to deposit
            my_model->Deposit (deposit_amount);
 }
 ...
}
```

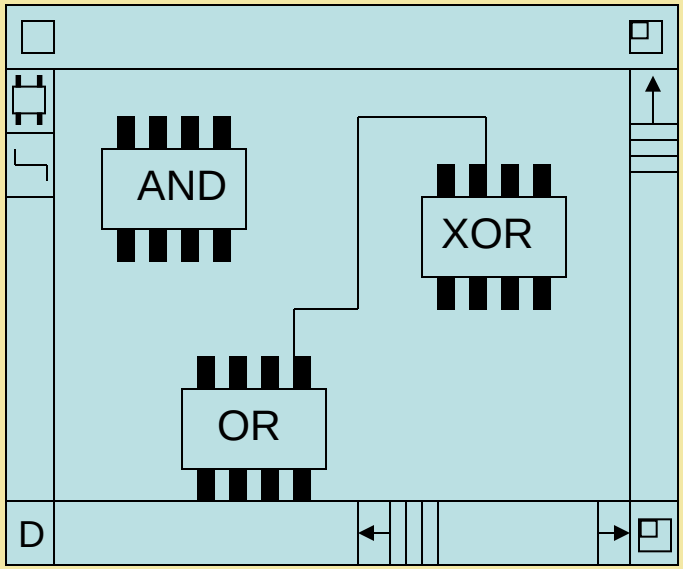
Example: Chip Layout Designer

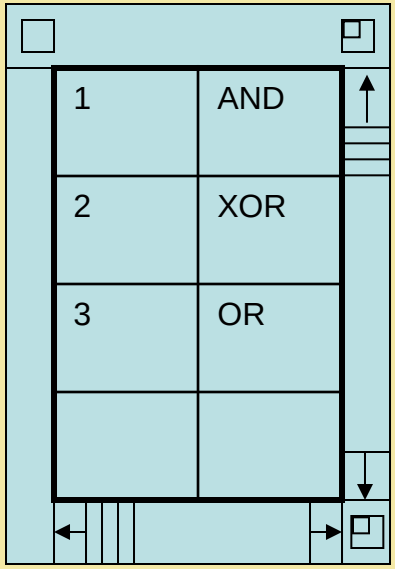
- Model
  - This is the data representation of the real world objects the user is interested
    - E.g. in a logic diagram program: Circuits, chips and wire classes
- View
  - Implements a visual display of the model
    - e.g. circuit view, parts list view, …
- Controller
  - Receives all of the input events from the user and decides what they mean and what should be done
    - E.g. the part that takes mouse down event and handling it properly



Logic Schematic Drawer

- Two windows
  - One for circuit layout
  - The other for showing a list of chips and names





Functional Model (Core): Circuit, Chips and Wires

- Circuit: captures the semantics of both the circuit view and the parts list view
  - Object being manipulated by both views
  - Consists of
    - Chips: Array of chip objects
    - Wires: Array of wire objects
    - Selected Chip: The index into “Chips” for the currently selected chip (-1 means no selection)
    - Selected Wire: The index into “Wires” for the currently selected wire (-1 similarly defined)

Circuit Object: Methods

- AddChip (CenterPoint): add a new chip to the circuit at the specified location
- AddWire(Chip1, Connector1, Chip2, Connector2): add a wire to the circuit
- SelectChip(ChipNum): Make the chip with specified index the selected one
- ~~MoveChip(ChipNum, NewCenterPoint)~~
- ChangeChipName(ChipNum, NewName)
- DeleteChip(ChipNum)
- SelectWire(WireNum)
- DeleteWire(WireNum)

- Notify views of any changes → See CircuitView Class
- Functionality of our application

Chip and Wire

- Chip
  - CenterPoint
  - Name

- Wire
  - Chip1
  - Connector1
  - Chip2
  - Connector2

CircuitView~~Controller~~ Class

- Circuit Class needs a general mechanism for notifying all of its views that some aspect of the circuit has changed.

- Define abstract class “CircuitView”
  - Circuit’s interface to any of its views
  - Combine view and controller functions in a single class
  - Must be subclass of WinEventHandler so that it can receive input and redraw events from the windowing system
  - Each of view will be subclasses of CircuitView and will inherit both the windowing system interface methods and view interface for the Circuit class

Viewer/Controller Implementation

- In this example, viewer/controller combined into a single class

- Views are subclass of CircuitView
- Passed to Circuit::AddView for attaching view to a particular model

- View
  - Actual redrawing in response to Redraw message
  - Responding to change notification message from the model
  - Handling of selection (translation of mouse location into some record of what should be selected)
  - Concerned with geometry of the presentation

- Controller
  - Translates user input into the methods of the functional model

CircuitView Class

- One attribute
  - myCircuit
    - Pointer to the model it represents
    - Must be specified when created

View handling events (from C or M)

- CircuitView::ChangeChip
- CircuitView::ChangeWire
- CircuitView::ChipMoved

e.g. Chip moved by C → V then its info updated in M … (or not … depends)

- When a view receives ChangeChip message, it will damage any region that display information about that chip.

- We have to create subclasses to CircuitView to handle specific situations
  - ChangeChip on part list and circuit behave differently …

View notification in Circuit Class

- Circuit Class needs to manage which views are attached to each model object
  - Circuit::AddView(View)
  - Circuit::RemoveView(View)

M notifying V

  - Circuit::ChangeChip
  - Circuit::ChangeWire
  - Circuit::ChipMoved
  - Same methods found on CircuitView class
  - Model codes can invoke these notifications methods without worrying about the views that need to be updated
  - Each of these Circuit methods will loop through the list of views in the circuit's view list and invoke the corresponding method on each view … e.g.

```
Void Circuit::ChangeChip (ChipNum)
        { For each V in the circuit's view list
          { V.ChangeChip(ChipNum) }
        }
```

M notifying all of the views

고려대학교

PartListView

- Chips are presented in their part number order
- ~~Each chip's presentation is exactly the same height~~

Basically for Redrawing

- Rectangle PartListView::ChipArea
  - This method will take a chip number and return the rectangle that bounds the area of that chip
  - PartListView is subclass of ClassView which is subclass of WinEventHandler, it has a GetBounds method that will return the bounding rectangle of the PartList window …
    - B = GetBounds()
    - Top = B.top + (ChipNum – 1) * ChipHeight
    - 위에서 몇 번째이니 앎으로 해당 칲 정보의 기하학적 위치/영역 알아냄

Change Notification

- No need for PartListView::ChangeWire and MoveChip

```
PartListView::ChangeChip()
{ myCanvas.Damage(ChipArea(ChipNum)) }
```

Do not redraw right away …
redraw may have to be coordinated
at the windows level (back to front)

PartListView: Redraw

- PartListView::Redraw

  For each chip C in the model

  { R = ChipArea(C)

  Draw info about chip C in R

  Draw horizontal line across the bottom of R

  }

  Draw bounding rectangle for the window

  Draw the vertical line

  OR

  Just redraw everything by window system …

Controller

- Handles input events. Its functionality is spread across the methods inherited from WinEventHandler that receive input events. These are e.g.
  - MouseDown
  - MouseMove
  - MouseUp
  - KeyPress
  - MouseEnter
  - MouseExit
  - …

Controller

PartListView::MouseUp(location, modifiers)
{myCircuit.SelectChip(WhichChip(location);}

PartListView::MouseDown(button, location, modifiers)
{myCircuit.SelectChip(WhichChip(location);}

Relaying events to the Model

PartListView::MouseMove(location, modifiers)
{if (modfiers show button is down)
{myCircuit.SelectChip(WhichChip(location));}
}

PartListView::KeyPress

- If there is no selected chip → return and do nothing
- Current character index is 0 and input character is backspace
  → do nothing
- Current character index is greater than zero and input character is a backspace
  → delete the indexed character from the string and invoke model's ChangeChipName method …
- …

Changing the display

- Simple: Just redraw the changed object
  - Does not work in general (other objects are often involved too)

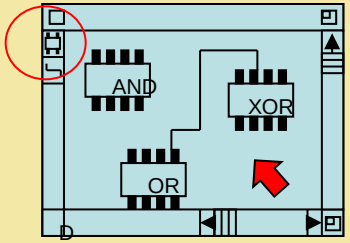
- Redraw everything from back to front → too slow?

Controller-View only

- Damage – Redraw technique
  - All windowing systems support a variant of the damage/redraw technique
  - A view can inform the windowing system when a region of a window needs to be updated
  - The windowing system will redraw the damaged region (or the whole screen space) in the back to front order (tied to the low level graphics level)

But what if the model wants to manage screen location of objects … (this will not work …)

Event Flow: Creation of a new chip (1)



- User already selected the chip icon on the left
- User places the mouse over some position where the new chip will go and press mouse button
- When mouse is pressed, the windowing system will identify which window should receive the event. The WinEventHandler that implements our circuit-view (and controller) will have its MouseDown method invoked (this is part of the controller)

  C → V
- Controller determines that it is in chip mode and inquires of the view as to whether the mouse is over an existing chip. If the mouse is not over an existing chip, the controller decides that new chip is to be created. It requests the view to start echoing a fact that it is creating a new chip. The MouseDown method then returns.

  C → V
- The user then adjusts where the chip will be placed by moving the mouse while holding down the mouse button. Each time the mouse moves, the windowing system will invoke the controller's MouseMove method. The controller will then have the view move the echoing rectangle to the new position.

Event Flow: Creation of a new chip (2)

- When the user finally decides that the chip is in the right position, the mouse button is released and the windowing system will invoke the MouseUp method on the view-controller. The controller will have the view remove the echoing rectangle from the screen, take itself out of chip positioning mode and invoke AddChip method on the circuit model, passing in the new position.

C → M

- When the model has its Add chip method invoked, it will add the new chip to its array of chips and will then go to the list of views that have been registered with this model. For each of these views, the model will invoke the appropriate methods to notify them that a new chip has been added.

M → V

- When the part list view receives notification that there is a new chip, it will inform the windowing system that the space at the bottom of the list is damaged and needs to be updated. Note that the part list view does not draw the new chip into the window at this point yet.

Pseudo event

Event Flow: Creation of a new chip (3)

- When the circuit view receives notification of the new chip, it will also inform the windowing system that the region where the new chip is to go is damaged.
- Note that even though the circuit view's controller initiated the request to create a new chip, the view still waits for notification from the model.
- Suppose that the model was enforcing some design constraints that would not allow chips to overlap each other. The model might return new alternative position, or disallow the creation.
- The view must accurately reflect what is in the model, even if it is different from what the view's own controller specified. Also note that circuit view must respond to notifications of new chips, no matter where such changes originate.

Event Flow: Creation of a new chip (3)

- When all views have been notified and performed their damage processing, the model returns from its AddChip method to the controller, which then returns from its MouseDown method, leaving the windowing system in control again.
- The windowing system determines there are damage requests pending and will respond to them. (The first damage request is from the part list view)
- The windowing system determines however that this portion of the part list window is completely obscured by some other window (e.g.).
  - In this case, the damage request is discarded because the damaged region is not visible. This is why the part list view or any other view only damages the changed area in response to notification of a model change, rather than drawing the changed information immediately.
  - The other damage request found by the windowing system is for the circuit view window. This is not obscured (e.g.) so the windowing system invokes the circuit view's Redraw method with the damaged area.
  - When the circuit view receives its Redraw message, it will look through all of the chips in the model and draw any chip that overlaps the damaged area. It will then look through all of the wires and do the same.
  - Several strategies are possible (e.g. just draw everything again or the part that changed only).

Pseudo event → V (Redraw)

고려대학교

Event Flow: Moving a chip (1) (Focusing on the damage/redraw mechanism)

- When mouse button goes down over the XOR chip (e.g.), the windowing system invokes the controller's MouseDown event
- The controller requests the view to select a chip and view returns the index of the XOR chip as the one selected. The controller then notifies the model of the selection by calling the model's SelectChip method.

C → V, C → M?

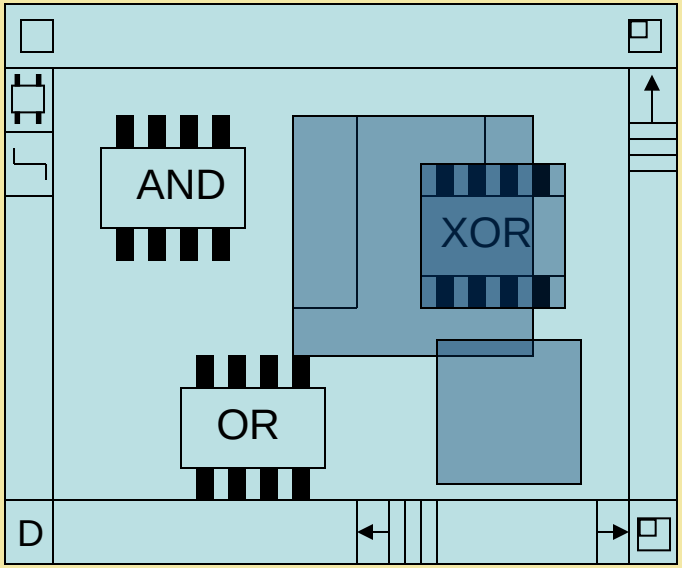
- The model's SelectChip method notifies all views registered with that model that the XOR chip has been selected. Each view then damages its presentation of the XOR chip. In the layout view, the rectangular region around the chip is damaged; in the part list view, the chip's name region is damaged.
- The controller then stores the fact that it is waiting to drag the chip to a new location and returns to the windowing system.
- The windowing system locates the entries for the damaged entries and invokes the Redraw methods on the appropriate views. These Redraw methods will draw the presentation of the XOR chip to show that it has been selected.

C → Pseudo event → V (Redraw)

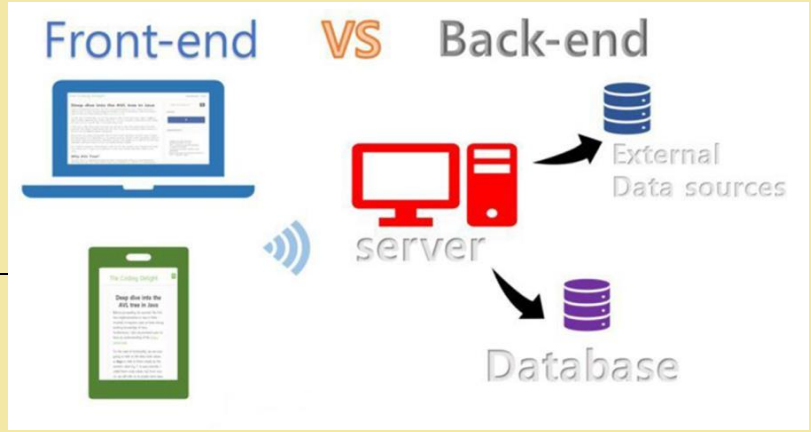
Event Flow: Moving a chip (2) (Focusing on the damage/redraw mechanism)

- The windowing system then waits for more input events. Since we are dragging the chip, the next input event will be a movement of the mouse. When each mouse movement is received by the windowing system, the system calls the MouseMove method on the circuit layout view. This method must echo the new location of the chip on the screen. C → V
- When the mouse button is released, the windowing system will send a MouseUp message to the controller. ~~The controller remembers that it is dragging a chip to a new location and invokes the model's MoveChip method.~~
- ~~The model will notify each view that the XOR chip has moved to a new location. The part list view will ignore this notice because its display does not involve the chip location.~~
- The circuit layout view has some work to do, however. The circuit layout view must not only move the chip but also the wires connected to it as well. When moving an object, we must damage both the old location and the new location. Because the wires are moving, the areas around the wires must also be damaged.

Pseudo event → V



Thought on MVC



- Front end – UI stuff
  - UI Framework (e.g. JAVA AWT, HTML/JS,…)
- Back end – Core function
  - Some other framework …
- What about functions that have to do with UI/UX?
  - Front or back end?
- What are some other methodologies:
  - React …? → still falls under the MVC
    - More about how to deploy front-end objects effectively or how to manage complex interactive behaviors of many objects …
    - More like methodology on front-end engineering …