Communication Patterns

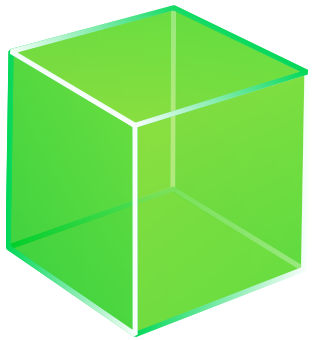
4/28’ 16

Communication Patterns

Every apps consists of multiple modules (objects) that need to communicate with each other to get the job done.

- Notifications
- Callback blocks/closures
- *Key-Value Observation KVO*
- Delegation
- Target-Action

Target-Action



UIEvent

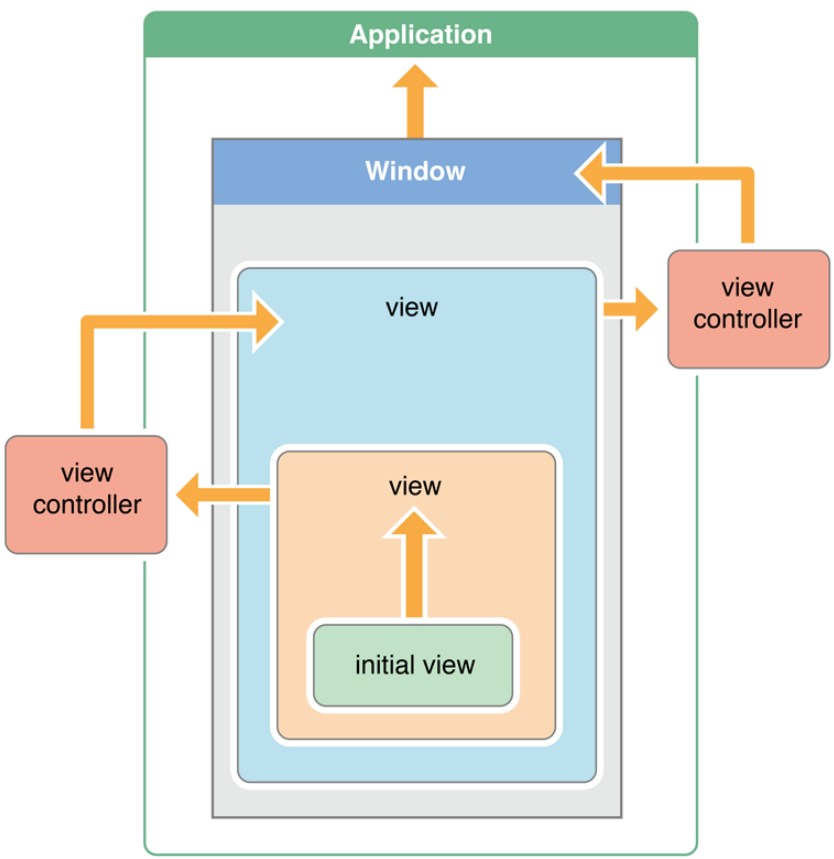
target

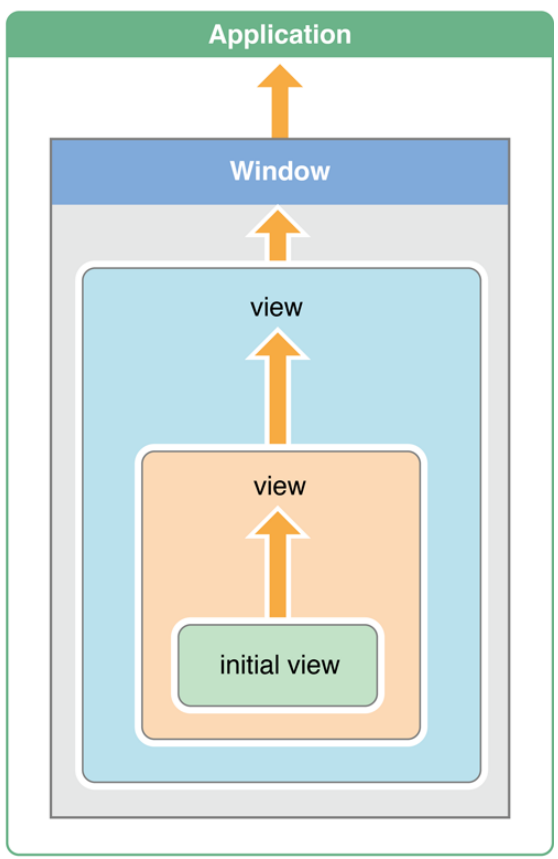
message passing

UIResponder

Responder Chain

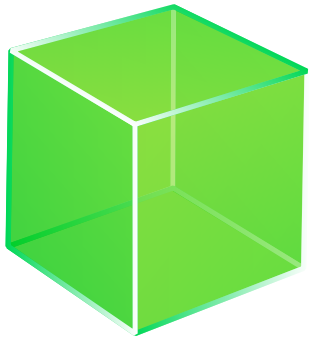
iOS





OS X

Delegation



UIViewController
<UIWebViewDelegate>

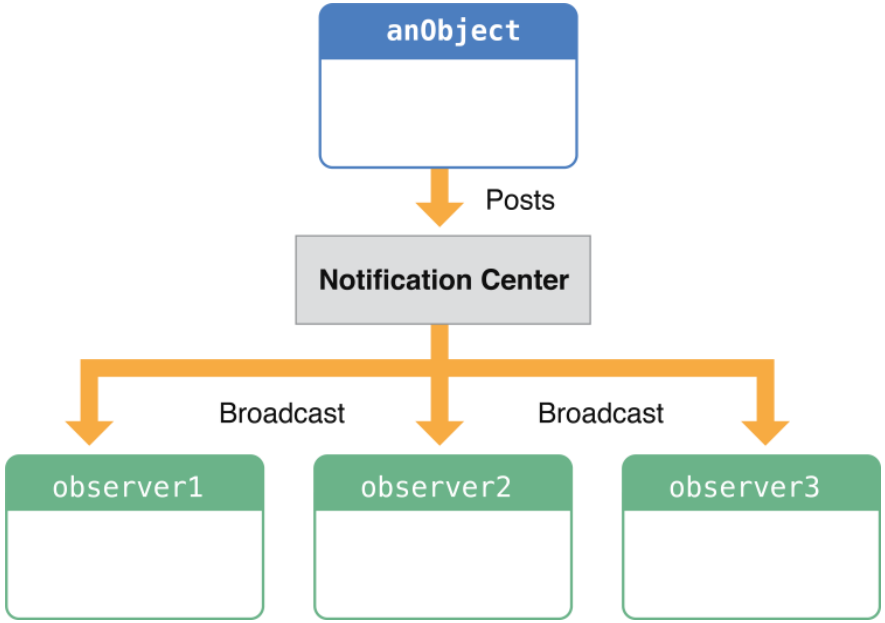
strong ref.

weak ref.

UIWebView

Notification Pattern

Notification Pattern



Notification Pattern

A notification encapsulates information about an event, including event name, sender, and a dict containing information of the event.

Objects that need to know about an event register with the notification center that it wants to be notified when that event emits.

When the event does happen, a notification is posted to the notification center, which broadcasts it to all registered objects.

Also called “Publish-Subscribe” pattern in other languages.

```
class MyClass: NSObject {

    override init() {
        super.init()
        NSNotificationCenter.defaultCenter().addObserver(self,
                                                         selector: #selector(MyClass.appDidBecomeActive(_:)),
                                                         name: UIApplicationDidBecomeActiveNotification,
                                                         object: nil)
    }

    deinit {
        NSNotificationCenter.defaultCenter().removeObserver(self,
                                                            name: UIApplicationDidFinishLaunchingNotification,
                                                            object: nil)
    }

    func appDidBecomeActive(notification: NSNotification) {
        print(notification.name, notification.object, notification.userInfo)
    }

}
```

Register to Notification Center

```
NSNotificationCenter.defaultCenter().addObserver(self,
    selector: #selector(MyClass.appDidBecomeActive(_:)),
    name: UIApplicationDidBecomeActiveNotification,
    object: nil)
```

Set the notification name and sender object you are interested in.

Both name and object can be nil, which means you don't care what kind of events and whoever sends it. (i.e., all events from all senders)

Use observer and selector to specify the receiving method.

The observer should be instances of classes, and the `#selector` syntax represents a method of a class.

Deregister from Notification Center

```
NSNotificationCenter.defaultCenter().removeObserver(self,
    name: UIApplicationDidFinishLaunchingNotification,
    object: nil)
```

Deregister from notification center when you are no longer interested in an event, or when you are no longer able to receive messages.

Usually put this code in **`deinit`** method of a class.

Post Notifications

```
let SomeDataUpdatedNotification = "SomeDataUpdatedNotification"

func updateSomeData() {
    // ... Data update logic here
    let n = NSNotification(name: SomeDataUpdatedNotification,
        object: dataSource,
        userInfo: ["success": true])
    NSNotificationCenter.defaultCenter().postNotification(n)
}
```

Notification names are usually constant global variable.

Using `let` to declare it at the global scope.

NSNotificationQueue

Notification center posts notifications synchronously.

So the sender would be blocked by receivers.

Use notification queue to post notifications asynchronously.

With `NSPostingStyle` to specify the time to post notifications.

Use coalescing options to ignore queued duplicated notifications.

Callback Closures/Blocks Pattern

Callback Closures/Blocks Pattern

```
typealias DataFetchHandler = (data: String?, error: Error?) -> Void
func fetchRemoteData(completion handler: DataFetchHandler) {
    // ... data logic goes here
    handler(data: "data string", error: nil)
}
```

node.js uses this pattern very much.

be careful about the object reference cycle when using this pattern.

Target-Action

Delegation

Callback blocks/closures

Notifications

Key-Value Observation, KVO

Easy to setup

But compiler cannot validate the setup for you.

Cannot carry extra information to the receiver and get return value from receivers.

Used by the OS to dispatch user interaction events.

Target-Action

Delegation

Callback blocks/closures

Notifications

Key-Value Observation, KVO

Very strict syntax, and compiler would check the conformation.
But also hard to setup

Easy to debug and trace.

Easy to pass information as arguments and get return value from the receiver.

Target-Action

Delegation

Callback blocks/closures

Notifications

Key-Value Observation, KVO

Easy to setup

Suited for one-off event/ response callback

Target-Action

Delegation

Callback blocks/closures

Notifications

Key-Value Observation, KVO

Each message can has multiple receivers

Cannot get return value.
The sender doesn't know the existence of receivers.

The sender takes the initiative to send messages.

Hard to debug and trace
But easy to setup and use

Target-Action

Delegation

Callback blocks/closures

Notifications

Key-Value Observation, KVO

Focuses on value changes

Cannot get return value from the receiver.

The sender doesn't know the existence of receivers.

The receiver takes the initiative to observe value changes.

Old-school and bad design API

objc.io - Communication Patterns
https://www.objc.io/issues/7-foundation/communication-patterns/

iOS Developer Library - Notification Programming Topics
https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Notifications/Introduction/introNotifications.html#//apple_ref/doc/uid/10000043-SW1

NSHipster - NSNotification & NSNotificationCenter
http://nshipster.com/nsnotification-and-nsnotificationcenter/

objc.io - Key Value Coding and Observing
https://www.objc.io/issues/7-foundation/key-value-coding-and-observing/

NSHipster - Key-Value Observing
http://nshipster.com/key-value-observing/