iPhone Application Programming
Lecture 4: Foundation Classes and Design Patterns

Nur Al-huda Hamdan
Media Computing Group
RWTH Aachen University

Winter Semester 2015/2016

http://hci.rwth-aachen.de/iphone

Media Computing Group | RWTH AACHEN UNIVERSITY

Learning Objectives

- Warp up swift: "easy to learn, hard to master"
  - Enumeration, error handling and ARC
- Cocoa Touch
  - Concurrency programming
  - Foundation classes
  - MVC
  - Notifications, KVO, target-action

Swift Built-in Types

- Swift's six built-in types

Named Types

- ✓ Protocols
- ✓ Structs
- ✓ Classes
- ✓ Enumerations

Compound Types

- ✓ Functions
- ✓ Tuples

Enumerations

- Represent a finite number of states
- There are two distinct types of enumerations in Swift
  - Raw value enumerations
    - Similar to Java or C enumerations
  - Associated value enumerations
    - Similar to tagged unions (e.g. in Haskell)

Raw Value Enumerations

- Much more powerful than C enumerations
  - Can have methods and initializers, can have extensions and can conform to protocols
- More flexible than Java enumerations
  - Can be defined over other underlying types (String, Character, all numeric types)

```
enum TrainClass: String, Stringifyable {
  case S = "S-Bahn"
  case RB = "Regionalbahn"
  case RE = "Regional-Express"
  case IC = "Intercity"
  case ICE = "Intercity Express"
  static let allCases = [S, RB, RE, IC, ICE]

  func onTime() -> Bool {
   if self == .S || self == .ICE {
     return true
   }
   return false
  }

 func stringify() -> String {
  return self.rawValue
  }
}
```

Associated Value Enumerations

- Every case represents a tuple type
  - Can be used as simple static Polymorphism
- Instantiate cases with values of the represented type
- You can mix the two enumeration types

```
enum Transport {
  case plane(String, Int)
  case train(TrainClass, Int)
  case bus(Int)
  case car(String, String, Int)
}

var myRide = Transport.train(.ICE, 11)
// GDL strike: change travel plans!
myRide = .car("AC", "X", 1337)

func canWork(onRide: Transport) -> Bool {
  switch onRide {
  case .train(let trainClass, let number):
    return trainClass == .ICE
  case .plane(_, _):
    return true
  default:
    return false
  }
}
```

Extensions

- Can extend Structs, Classes, Enumerations
- Can implement protocol requirements
- Can add functions, computed properties, nested types
- Can declare protocol conformance
- Cannot override existing functionality
- Often useful to clean up code structure

```
extension Temperature : CustomStringConvertible {
   var description : String {
     get {
       return (NSString(format:"%.2d", self.value) as
String) + self.unit.rawValue
     }
   }
}
```

Nested Types

- Nest enums, classes, and structs within the definition of a type
- Can have deep hierarchies
- To use a nested type outside definition scope, prefix its name with the name of the type(s) it is nested within

```
struct BlackjackCard {
     // nested Rank enumeration
    enum Rank: Int {
        case Two = 2, Three, Four, Five, Six, Seven, Eight,
Nine, Ten
        case Jack, Queen, King, Ace
         // nested Values structure
        struct Values {
            let first: Int, second: Int?
        }
        // computed property
        var values: Values {
            switch self {
            case .Ace:
                return Values(first: 1, second: 11)
            case .Jack, .Queen, .King:
                return Values(first: 10, second: nil)
            default:
                return Values(first: self.rawValue, second: nil)
            }
        }
    }
}

let c1 = BlackjackCard.Rank.Three.rawValue
let c2 = BlackjackCard.Rank.Ace.values.second
//Three deep
let newCard = BlackjackCard.Rank.Values(first: 10, second: nil)
```

Optional Chaining

- Accessing properties/methods on optionals
- If one of the optionals is nil, this syntax fails graceful (no run time error)
- If all optionals are set, the chain returns an optional (even if the object in request, e.g., subviews, is not optional)
  - self.window?.rootViewController?.view.subviews!
    //compiler error, subviews is not of type optional
- With subscripts dict?[someKey]

```
for subview in
(self.window?.rootViewController?.view.subviews)!
as [UIView]
 {
       //type casting the subview to UILable
      if let labelView = subview as? UILabel
      {
        let formatter = NSDateFormatter()
        formatter.timeStyle = .MediumStyle
         labelView.text =
formatter.stringFromDate(NSDate())
       }
 }
```

```
var dict:[String:String]?
dict?[someKey]

var someArray:[String]?
someArray?.insert(someValue, atIndex: someIndex)

//side notes
var dict:[String:String?]
dict = ["hi":"there", "bye":nil]
dict["hi"] // "there"
dict["bye"] // nil

var dict:[String?:String] //error keys cannot be optional
```

Type Casting

- Upcasting: casts an instance to its superclass type
  - instance `as` superclass (assumes it is always successful)
  - 0.1 `as Int //0` and 0.1 `as Double //0.1`
- Downcasting: casts an instance of a superclass to its actual subclass type
  - let object = instance `as`! subclass. Results in downcasts + force unwarp OR runtime error
  - if let object = instance `as`? subclass {…}. Results in downcasts or nil
- Object checking: checks if instance is of type subclass
  - instance `is` subclass `//true or false`

Access Control

- `private` entities are available only from within the source file where they are defined
- `internal` entities are available to the entire module that includes the definition (e.g. an app or framework target) ← the default case
- `public` entities are intended for use as API, and can be accessed by any file that imports the module, e.g. as a framework used in several of your projects
- Apply to classes, structures, and enumerations, properties, methods, initializers, and subscripts
- Global constants, variables, functions, and protocols can be restricted to a certain context

Custom Operators

- Operators can be declared at global scope
- Can have prefix, infix or postfix modifiers
- Infix operators have associativity and precedence values
- Operators are implemented as functions at global scope
- Be very conservative when overloading operators!

```
// ...this one maybe makes sense...
prefix operator ∑ {}
prefix func ∑(a: [Int]) -> Int {
   var accum = 0
   for value in a {
     accum += value
   }
   return accum
}
var myArray = [-2, 6, 0, 1]
let sum = ∑myArray

// ...this one surely not!
postfix operator ^-^ {}
postfix func ^-^(s: String) -> String {
   return s + " 😄"
}
let chatMessage = "Operator Overloading 4TW!"
print(chatMessage^-^)
```

Error Handling

- Errors values are of types that conform to the empty protocol ErrorType. The protocol indicates that a type can be used for error handling
- func canThrowErrors() throws -> Type to indicate the function throws errors
- Use throw someErrorOfErrorType (usually enums) to indicate an error
- Errors thrown inside a nonthrowing function must be handled inside the function
- Errors thrown inside a throwing function can propagate to the scope from which they were called

```
enum VendingMachineError: ErrorType {
    case InvalidSelection
    case InsufficientFunds(coinsNeeded: Int)
    case OutOfStock
}
```

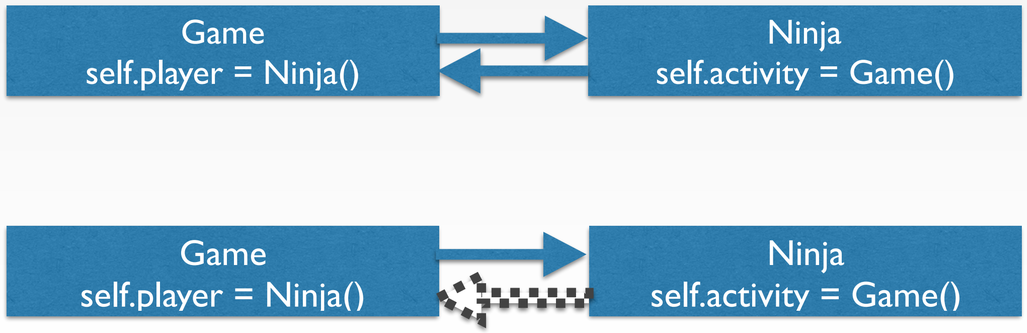
```
throw
VendingMachineError.InsufficientFunds
(coinsNeeded: 5)
```

Error Handling Techniques

- **do-try-catch**: When working with code that can throw errors, surround it with a do, call it with a try and handle it in one or more catch clauses
  - Write a pattern after catch to indicate what errors to handle, or no pattern to handle all errors
  - If errors are not handled in your catches or the surrounding scope, then runtime error
- **try?**: Convert an error to optional
  - if let x = try? someThrowingFunction() { return data } return nil `// x is optional. It is either set with returned value or in case of errors it's nil`
- **try!**: The error will not occur (or you'll get runtime error)
  - let x = try! someThrowingFunction()

Automatic Reference Counting

- ARC *automatically* frees up the memory used by class instances no longer needed
- ARC tracks how many properties, constants, and variables are currently referring to each class instance and deallocates the instance only if no active references exists
- Strong reference cycles prevent referencing instances from ever being deallocated, causing a memory leak
  - Use a (var) weak reference if that reference can ever be "no value", i.e., optional
  - For non optional references, use unowned



Ref A	Ref B	Solution
optional	optional	make one var weak optional
optional	non optional	unowned the non optional
non optional	non optional	unowned one and make the other implicitly unwrapped !

Strong Reference Cycles in Closures

- If a closure is called within a class instance and you use self.someProperty or self.someMethod() in the closure, self is then captured and a strong reference cycle is created
- Resolved with capture lists or unowned and weak references/variables

```
let alterAction = UIAlertAction(title: "Submit", style: .Default)
{ [unowned self] (action: UIAlertAction!) in
   self.doSomething()
}
```

Any and AnyObject

- Because of bridging, we see AnyObject and Any types when using Cocoa APIs
  - In objective-c an array can contain heterogeneous types, in swift only homogenous
- Swift provides two special type aliases for working with non-specific types:
  - AnyObject can represent an instance of any class type.
  - Any can represent an instance of any type at all, including function types.
    - Int, Double, Float, String, Array, and Dictionary are not objects

```
typealias funcDef = (Double) -> (Double)

var hetroArray1 = ["hello", 1, 0.1, true]//[NSObject]

var hetroArray2 = [AnyObject]()
hetroArray2.append(NSData())

var hetroArray3 = [Any]()
hetroArray3.append(hetroArray1)
hetroArray3.append(hetroArray2)
hetroArray3.append(funcDef)
```

Wrapping Up Swift

- To check for conditions early in your code
  - Assert: continue, crash (if in debug mode), or ignore (if in release mode)
  - Guard: continue, or execute exit code gracefully
- The nil-coalescing operator a ?? b
  - a != nil ? a! : b returns a unwrapped if not nil or b
- //MARK: Some Section Name
  - To mark sections of code in the symbol navigator

```
var data:NSData? =
"text".dataUsingEncoding(NSUTF8StringEncoding)

//Msg is printed and program crushes if the condition fails
assert(data?.length > 0, "no data, will exit!")
```

```
//Else code is executed if the condition fails. It must be
followed by return, break, continue or throw
guard let unwarpData = data where unwarpData.length > 0
else
{
        print("no data, will exit!")
        return //break or continue or throw
}
print("data arrived \(unwarpData)")
```

Cocoa Touch

Foundation Classes

- Numeric types: NSNumber
- Other types: NSValue
- Binary data: NSData. Used to read and write files
- NSString bridged to String comes with many methods to search and manipulate strings
- NSDate, NSDateFormatter and NSCalendar

```
let startOfHolidayComponents =
NSDateComponents()
startOfHolidayComponents.year = 2015
startOfHolidayComponents.month = 12
startOfHolidayComponents.day = 23
startOfHolidayComponents.hour = 8
startOfHolidayComponents.minute = 0
startOfHolidayComponents.second = 0

let startOfHoliday =
NSCalendar.currentCalendar().dateFromCompon
ents(startOfHolidayComponents)!

let formatter = NSDateFormatter()
formatter.dateStyle =
NSDateFormatterStyle.LongStyle
formatter.timeStyle = .MediumStyle

formatter.stringFromDate(startOfHoliday)
//December 23, 2015 at 8:00:00 AM
```

NSNumber

- Encapsulation of numerical values
- Provides compare: method to determine the ordering of two NSNumbers

```
+ numberWithBool:                    – boolValue
+ numberWithChar:                    – charValue
+ numberWithDouble:                  – doubleValue
+ numberWithFloat:                   – floatValue
+ numberWithInt:                     – intValue
+ numberWithInteger:                 – integerValue
+ numberWithLong:                    – longValue
+ numberWithShort:                   – shortValue
+ numberWithUnsignedChar:            – unsignedCharValue
+ numberWithUnsignedInt:             – unsignedIntegerValue
+ numberWithUnsignedInteger:         – unsignedIntValue
+ numberWithUnsignedLong:            – unsignedLongLongValue
+ numberWithUnsignedLongLong:        – unsignedLongValue
+ numberWithUnsignedShort:           – unsignedShortValue
```

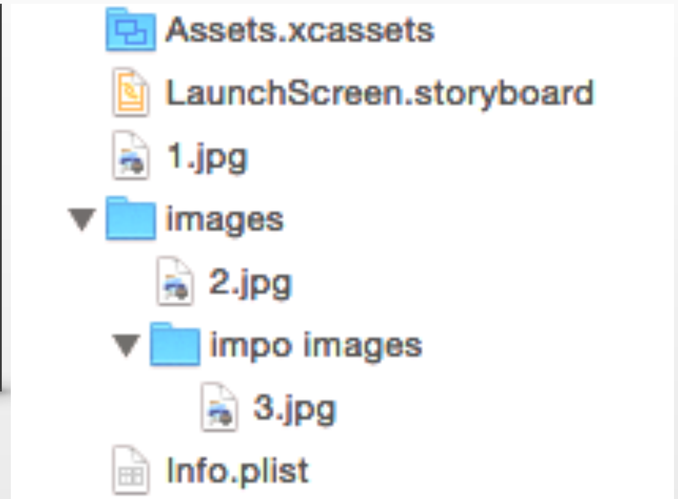
Bundels

The app bundle is read-only!

- Bundles encapsulate code and resources, and facilitate localization. Each bundle type has a defined structure
- An app bundle (.app) is a directory containing: the binary file (compiled version of your code), Info.plist, all the media assets of the app (e.g., icon and launch images), visual layout files, and metadata and security entitlements
- NSBundle.mainBundle() is where your resources reside. Use paths to access files in a bundle

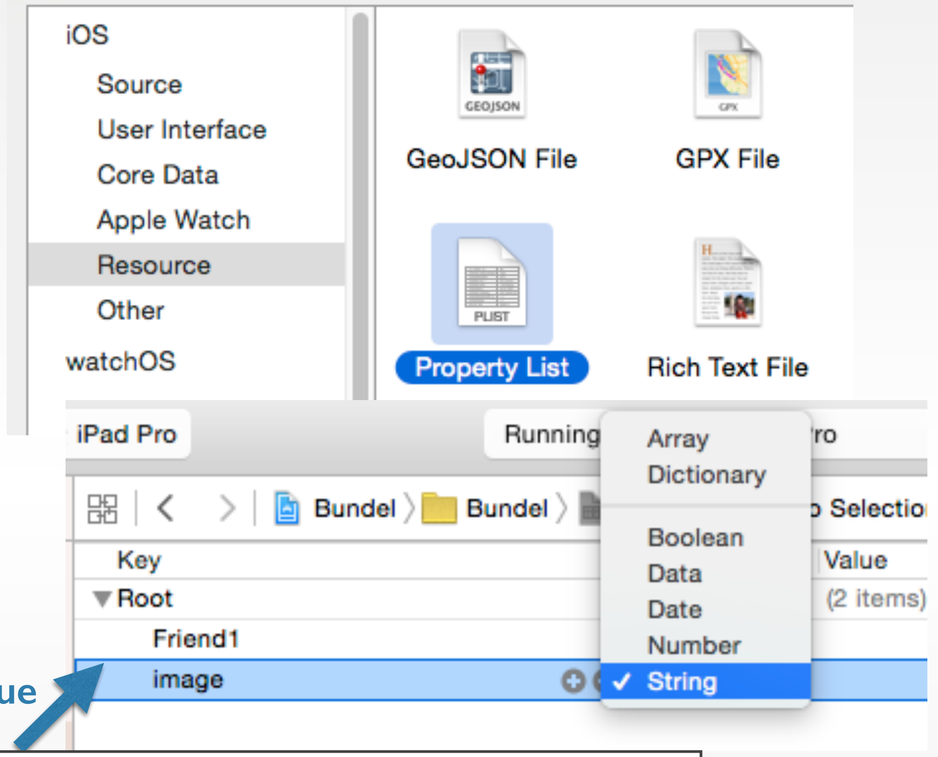
```
imageView.image = UIImage(named: “1.jpg") //no need to reference NSBundle

let targetPath1 : String? = NSBundle.mainBundle().pathForResource("1", ofType:
"jpg")
let targetPath2 = NSBundle.mainBundle().pathForResource("2", ofType: "jpg",
inDirectory: "images")
let targetPath3 = NSBundle.mainBundle().pathForResource("3", ofType: "jpg",
inDirectory: "images/impo images")
imageView.image = UIImage(named: targetPath2!) //targetPath1, targetPath3
```



Property Lists (plist)

- Property Lists offer a convenient way to store and retrieve simple structural data: basic types, binary data, date, and collections of allowed types
- Used to store small amounts of data few 100 KB. Mainly for app settings and app default data
- Can be directly read into a dictionary or array




```
//From app bundle
let path = NSBundle.mainBundle().pathForResource("Defaults", ofType: “plist”)!
var resultDictionary = NSMutableDictionary(contentsOfFile: path) //optional

//From documents directory
let documentsDirectory = NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMask, true)
[0]
let path = documentsDirectory.stringByAppendingString("Defaults.plist")
let fileManager = NSFileManager.defaultManager()
if(fileManager.fileExistsAtPath(path)) {resultDictionary = NSMutableDictionary(contentsOfFile: path)}
```

File Manager

- NSFileManager object lets you examine the contents of the file system and make changes to it
- Access files using their path = documents directory + unique file name *with* file extension (or UUID)
- More details in the Data Persistence lecture…

```
//Read from documents directory any file extension
let documentsDirectory =
NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMask, true)[0]
let documentPath = documentsDirectory.stringByAppendingString("Defaults.plist")
let fileManager = NSFileManager.defaultManager()
if(fileManager.fileExistsAtPath(documentPath)) {resultDictionary =
NSMutableDictionary(contentsOfFile: documentPath)}
//Write to documents directory
resultDictionary!.writeToFile(documentPath, atomically: false)
```

```
let path:String? = NSBundle.mainBundle().resourcePath! //all files (not embedded in folders) in the app
bundle
let fileManager = NSFileManager.defaultManager()

let items = try! fileManager.contentsOfDirectoryAtPath(path) //array of all files in path
```

Asset Catalogs

- Asset Catalogs provide an optimised way of importing and using images in iOS project
- App icons, launch image, image sets, other data files (not binary executables)
- Can access files in code by name directly
- Assets image sets come in the sizes 1x, 2x and 3x
  - A 1x image is just called its regular name, e.g., contactImage.png, contactImage@2x.png, (retina devices) contactImage@3x.png (retina HD devices). In code you only use contactImage.png and the system chooses the right image for the device

Asset	iPhone 6s Plus and iPhone 6 Plus (@3x)	iPhone 6s, iPhone 6, and iPhone 5 (@2x)	iPhone 4s (@2x)	iPad and iPad mini (@2x)	iPad 2 and iPad mini (@1x)	iPad Pro (@2x)
App icon (**required** for all apps)	180 x 180	120 x 120	120 x 120	152 x 152	76 x 76	167 x 167

Threads are better for code that must run in real time

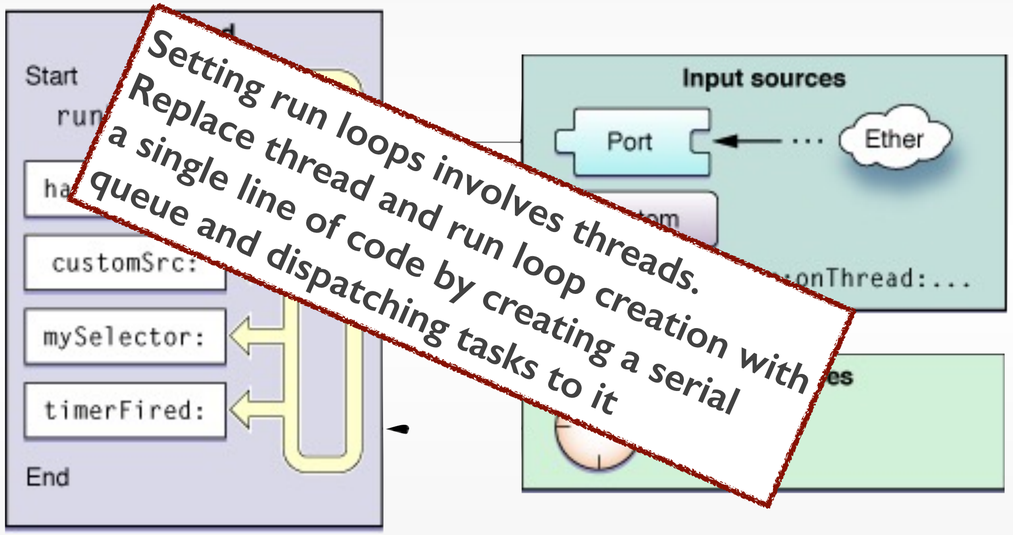
Threads

Threads are not scalable!

- Threads are one of several technologies for concurrency. In a non-concurrent app, only one thread "main thread" starts and ends with the app's main routine
  - Concurrent apps starts with one thread and adds more as needed to create additional execution paths independent of the main thread
- All user interface work must occur on the **main thread.** If you try to execute code on a different thread results are unpredictable
- Thread creating and management is not trivial (e.g., Foundation's NSThread)
  - For example, because threads of an app share the same memory space, threads must coordinate, or race conditions occur
- Apple encourages developers to migrate away from threads to newer technologies such as operation objects and Grand Central Dispatch (GCD)

Run Loops

- Threads can be associated with a run loop which process requests dynamically as they arrive
  - A thread enters a loop and uses it to run event handlers in response to incoming events
  - Your code provides the control statements (the while or for loop) that drives the run loop
  - Within your loop, a run loop object runs the event-processing code that receives events and calls the installed handlers
- The run loop of your app's main thread starts automatically. If you create other threads, you must configure the run loop and start it manually
- Run loops receive events from two different types of sources
  - Input sources deliver asynchronous events, e.g., msgs from another thread or app
  - Timer sources deliver synchronous events at a scheduled or repeating interval



Asynchronous Design Approach

- An asynchronous function does work behind the scenes to start a task running but returns before that task completes
  - Involves acquiring a background thread, starting the desired task on that thread, and then sending a notification to the caller (a callback function) when the task is done
- Grand Central Dispatch (GCD) is one technology for starting tasks asynchronously
  - Involves defining the tasks to execute and adding them to an appropriate dispatch queue
  - Takes care of creating the needed threads and of scheduling your tasks to run on those threads
- Operation queues are Objective-C objects that act very much like dispatch queues

Dispatch Queue (GCD)

- A dispatch queue executes tasks serially or concurrently in first-in, first-out order
- Dispatch queues scale well, better than sync. locks, cannot deadlock, better speed and energy, simple API
- The tasks you submit to a dispatch queue must be encapsulated in a function or a closure
- Unlike dispatch queues NSOperationQueue execute tasks based on their dependencies not FIFO
  - Tasks should be instances of NSOperation and use key-value observing (KVO) notifications for monitoring the progress of a task

```
let queue:dispatch_queue_t = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)

dispatch_apply(count, queue){i in
    print(i)
}
```

Dispatch Sources (GCD)

- Dispatch sources are used for processing specific types of system events asynchronously
- When a particular event occurs they submits a specific closure or function to a dispatch queue
- Dispatch sources can monitor the following types of system events: timers, signal handlers, descriptor-related events (files r/w related), process-related events, mach port events, custom events that you trigger

Grand Central Dispatch

- If you're accessing a remote resource (a server, or a file), accessing and processing data should be in a background thread (not the main thread)
- The power of GCD is that it takes away a lot of the hassle of creating and working with multiple threads
- GCD creates a number of queues, and places tasks in those queues depending on how important you say they are
- You can create your custom queues
- Four background queues that you can use, each has its own quality of service QoS level
  - As a developer, you should categorise your app tasks based on quality of service (QoS) classes. This ensures that your app is responsive and energy efficient
  - The system uses QoS information to adjust priorities such as scheduling, CPU and I/O throughput, and timer latency

Quality of Service

QoS Class	Usage	Focus	Duration
User-interactive	Operating on the main thread, refreshing the user interface, or performing animations.	Responsiveness and performance	Virtually instantaneous
User-initiated	Opening a saved document or performing an action when the user clicks something in the user interface	Responsiveness and performance	Nearly instantaneous, such as a few seconds or less
Utility	Downloading or importing data. Utility tasks typically have a progress bar that is visible to the user	Balance between responsiveness, performance, and energy	A few seconds to a few minutes.
Background	Indexing, synchronizing, and backups	Energy	Minutes or hours.

GCD and QoS Example

- dispatch_async() takes one parameter, then a closure to execute asynchronously. The parameter it takes is which queue you want to use
- Two functions: dispatch_get_global_queue() asks for a queue with a particular quality of service setting, and dispatch_get_main_queue() will use the main queue.

```
//background queue, USER_INITIATED (do not keep the user waiting)
dispatch_async(dispatch_get_global_queue(QOS_CLASS_USER_INITIATED, 0)) { [unowned self] in
    if let data = NSData(contentsOfURL: url) {
        //parse data
    }
    else {
        dispatch_async(dispatch_get_main_queue())
            { [unowned self] in
                let alert = UIAlertController(title: "Loading error", message: "There was a problem loading the data.",
preferredStyle: .Alert)
                alert.addAction(UIAlertAction(title: "OK", style: .Default, handler: nil))
                self.presentViewController(alert, animated: true, completion: nil)
        }//
    }
}
```

Is Concurrency Necessary?

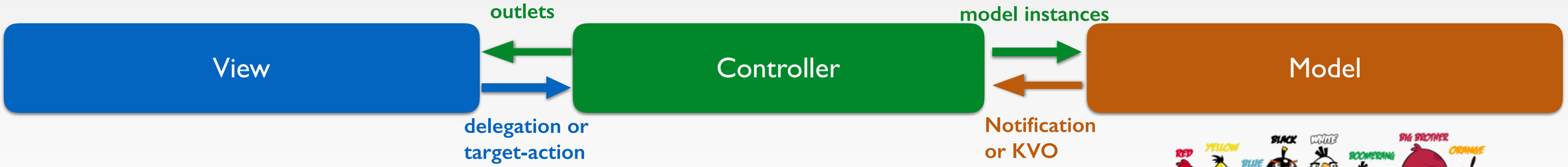
- Concurrency can improve the responsiveness of your code by ensuring that your main thread is responsive to user events
- It can improve the efficiency of your code to execute in less time
- It make your app more energy efficient
- However, it also adds overhead and increases the overall complexity of your code (writing and debugging)
- Done incorrectly, your app might become slower and less responsive
- I recommend reading Concurrency Programming Guide to be able to make the right "concurrency" decision for your app

MVC and Object-to-Object Communication

MVC Design Pattern

- An iOS app is a collection of objets
- MVC design pattern defines (a) the roles of objects in an app, (b) how they communicate with each other
  - Reusable, extensible, adaptable for different devices with different screen sizes
  - Cocoa Touch frameworks are based on MVC and require app custom objects to play one of the MVC roles

MVC



Interpret user actions and communicate changes to models

Decides how model data is displayed in the views

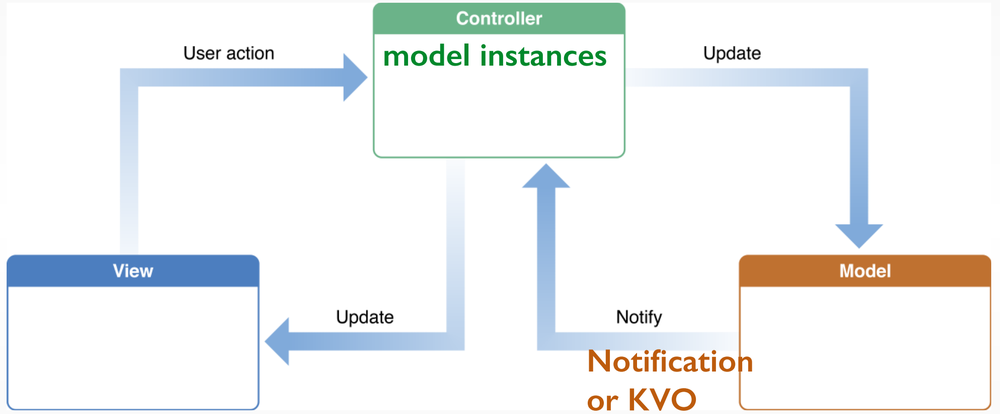
Each view controller is responsible of one screen of views

Representation of the models
Event handling (responds to users actions)

class RedBird:Bird
Attributes are properties let value =10
Behaviors/rules are methods func fly ()

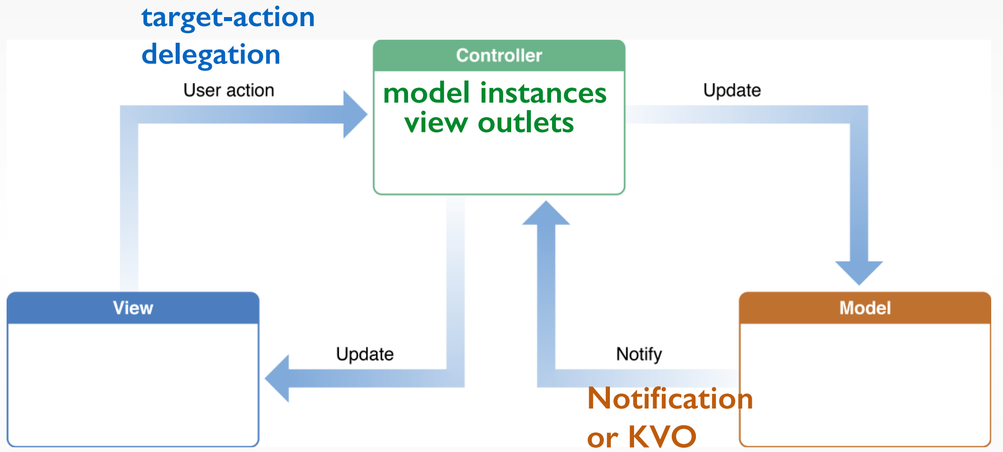
Models

- Model objects: data specific to an application and define the logic to manipulate that data
- UI independent
- No explicit connection to the view objects
- Uses Notification or KVO to notify the controller of, e.g., new data arriving from network
- Controller uses model instances to update them of user actions in views, e.g., remove a data entry



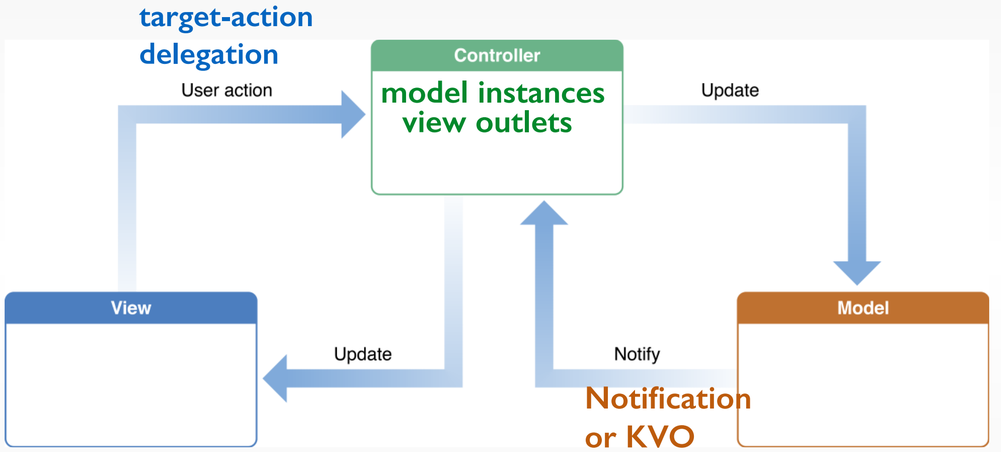
Views

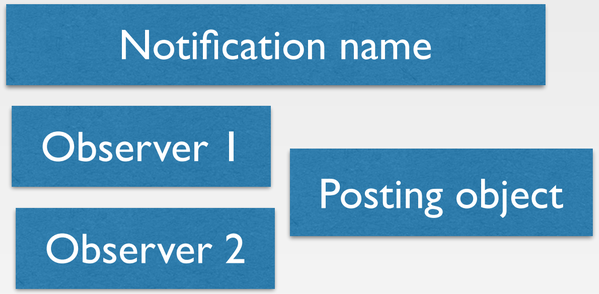
- View objects: an object that users can see. It can draw itself and respond to user actions
- No explicit connection to the model objects
- Send actions (blindly) to targets in the controller, e.g., a button was touched
- Use protocols to (blindly) synchronize with controller by invoking a method on a delegate/data source controller, e.g., the text view scroll position value, or what is data in this table cell



Controllers

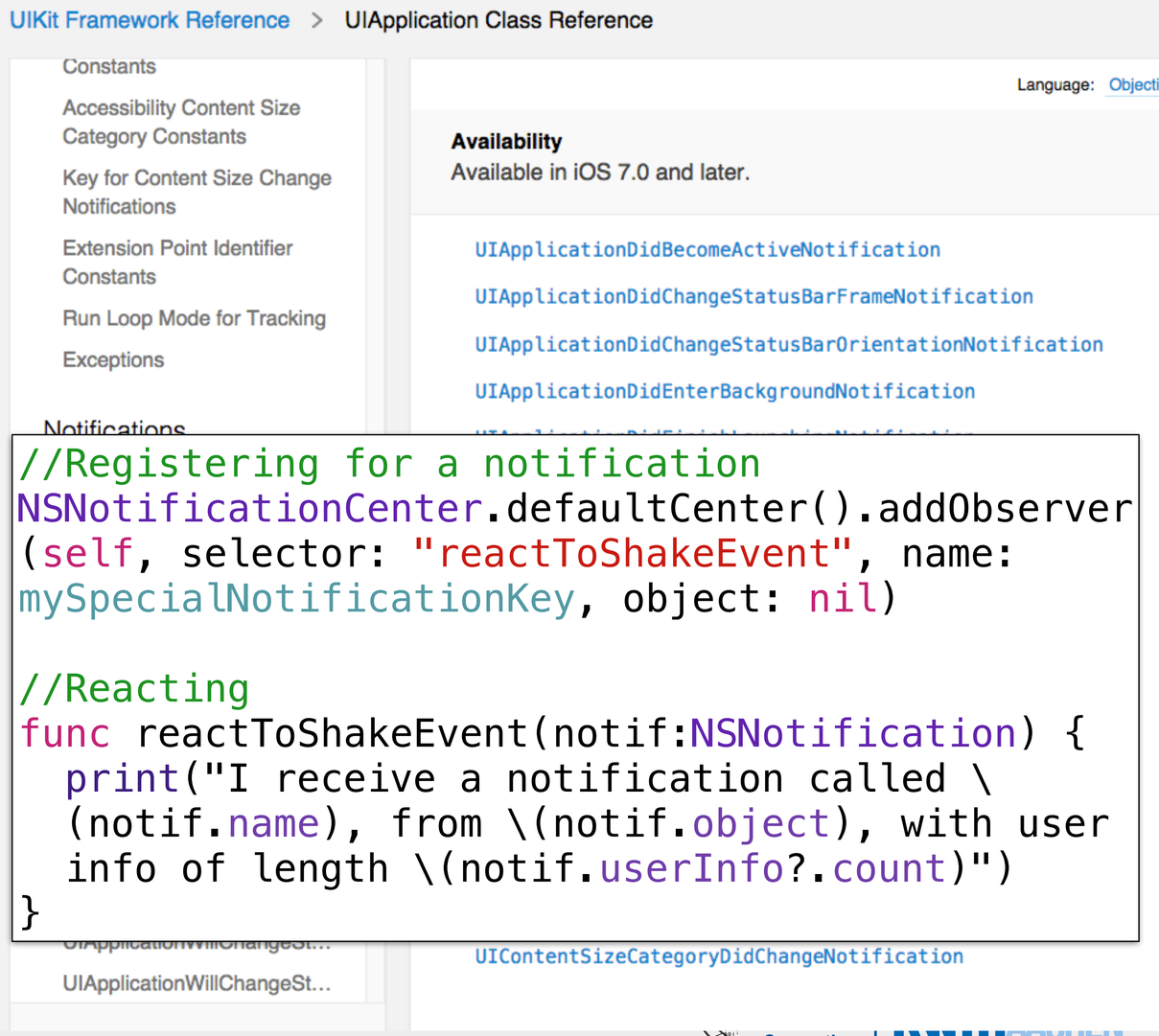
- Controller objects: mediate interaction between models and views
- In iOS each view controller is responsive for one screen of views
- Manages the presentation of views and the transition to any subsequent view in the app
- Display modal views, respond to low-memory warnings, and rotate views
- Interpret user actions and communicate changes to models
- Decides how model data is displayed in the views
- View controllers are typically the delegate or data source for many types of framework objects





Notifications

- When an event occurs, an object posts notification in a broadcasting fashion (doesn't know who wants it)
- An object (observer) registers itself to receive a notification (by name) for some event
- The observer implements a function to respond to the event
- The observer should remove itself if it's no longer listening for notifications (`deinit`, called when the object will dealloc)
- Application notifications are NOT <u>push notifcations</u>




```
//Registering for a notification
NSNotificationCenter.defaultCenter().addObserver
(self, selector: "reactToShakeEvent", name:
mySpecialNotificationKey, object: nil)

//Reacting
func reactToShakeEvent(notif:NSNotification) {
  print("I receive a notification called \
  (notif.name), from \(notif.object), with user
  info of length \(notif.userInfo?.count)")
}
```

Using Notifications

- Posting a notification requires defining a unique global string constant with the notification `name`
  - Should be added to the location that receives the event of interest, e.g., `applicationWillEnterForeground`

```
//Posting a notification
//Notification ahem on a global scope
let mySpecialNotificationKey = "CLKshakeEvent"

NSNotificationCenter.defaultCenter().postNotificationName(mySpecialNotificationKey,
object:self) //self (this class instance) is the one posting
```

Using Notifications

- Registration for a notification requires the notification `name`
  - `selector` is name of function that will be called
  - `object` is the one posting the notification
  - If notification `name` is nil, the notification center notifies the observer (`self`) of all notifications with an object matching `object`
  - If `object` is nil, the notification center notifies the observer of all notifications with the same notification `name`
  - The observer should implement the `selector`, which takes 0 or 1 argument of type `NSNotification` with
    - notification `name`; posting `object`; and `userInfo` (dictionary with additional relevant objects; can be nil)

```
//Posting a notification
//Notification ahem on a global scope
let mySpecialNotificationKey = "CLKshakeEvent"

NSNotificationCenter.defaultCenter().postNotific
ationName(mySpecialNotificationKey, object:self)
//self (this class instance) is the one posting
```

```
//Registering for a notification
NSNotificationCenter.defaultCenter().addObserver
(self, selector: "reactToShakeEvent", name:
mySpecialNotificationKey, object: nil)

//Reacting
func reactToShakeEvent(notif:NSNotification) {
  print("I receive a notification called \
  (notif.name), from \(notif.object), with user
  info of length \(notif.userInfo?.count)")
}
```

Using Notifications

- The observer should remove itself if it's no longer listening for notifications
  - When the observer is no longer referenced and thus deallocated from memory (in `deinit`)
  - When a notification is no longer relevant for an observer (you can put the same .`removeObserver` line anywhere in your code)

```
//Posting a notification
//Notification ahem on a global scope
let mySpecialNotificationKey = "CLKshakeEvent"

NSNotificationCenter.defaultCenter().postNotific
ationName(mySpecialNotificationKey, object:self)
//self (this class instance) is the one posting
```

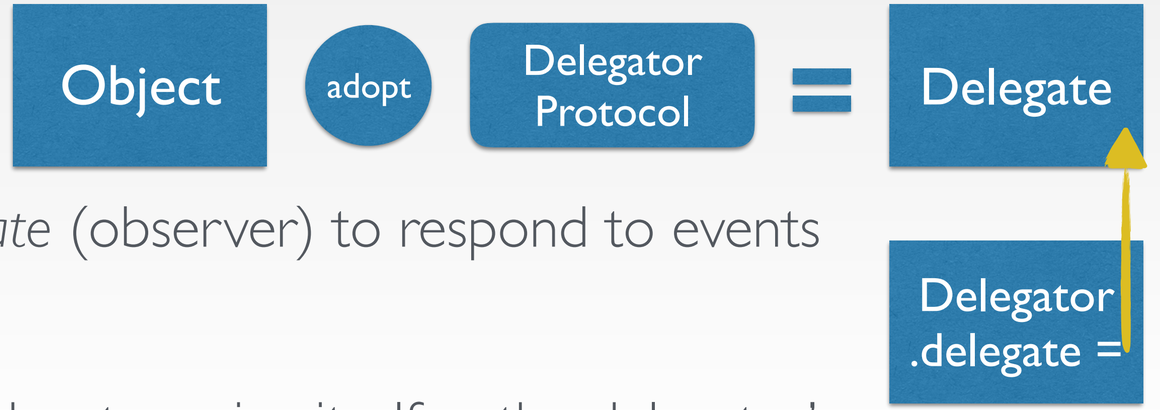
```
//Registering for a notification
NSNotificationCenter.defaultCenter().addObserver
(self, selector: "reactToShakeEvent", name:
mySpecialNotificationKey, object: nil)

//Reacting
func reactToShakeEvent(notif:NSNotification) {
  print("I receive a notification called \
  (notif.name), from \(notif.object), with user
  info of length \(notif.userInfo?.count)")
}
```

```
deinit {

NSNotificationCenter.defaultCenter().removeObserver(self)
}
```

Delegation



- Similar to notifications, delegation allows the *delegate* (observer) to respond to events on behalf of the *delegator* (posting object)
- Instead of registering for notifications, the delegate has to assign itself as the delegator's delegate and declare that will implement the *required* methods (*conform to protocol*)
- The delegator keeps a reference to the delegate and sends messages I *did* handle or *will* handle or *should* handle this event
- The main value of delegation is that it allows you to easily customize the behavior of several objects in one central object
- A delegate can be a data source for the delegator and respond to requests of data

KVO

- KVO is another mechanism from object-to-object combination, especially Model → Controller
  - The observed object should inherit from `NSObject`. The observed property should be `dynamic`, e.g., dynamic var myDate = NSDate()
  - The observing object should add itself as an observe for the property `addObserver(_:forKeyPath:options:context)`, override `observeValueForKeyPath`. When done observing, remove the observer `removeObserver(_:forKeyPath:context)`, e.g., in `deinint`

ViewController:WKNavigationDelegate

```
var webView: WKWebView!
Delegation
func webView(webView:WKWebView,
didFinishNavigation navigation:WKNavigation!) {
        title = webView.title
    }


KVO
 webView.addObserver(self, forKeyPath:
"estimatedProgress", options: .New, context: nil)
override func observeValueForKeyPath(keyPath:
String?, ofObject object:AnyObject?, change:
[String :AnyObject]?, context:
UnsafeMutablePointer<Void>) {
        if keyPath == "estimatedProgress" {
            progressView.progress =
Float(webView.estimatedProgress) }}
webView.removeObserver(self, forKeyPath:
“estimatedProgress")


Target-action
navigationItem.rightBarButtonItem =
UIBarButtonItem(barButtonSystemItem: .Refresh,
target: webView, action: "reload")
```

WKWebView

var estimatedProgress: Double

WKNavigationDelegate

optional func webView(_ webView:
WKWebView,
  didFinishNavigation navigation:
WKNavigation!)

Next Time

- The slides and playgrounds from this lecture will be uploaded to our website
- This week's reading assignment will be on the website today
- Come to the lab next week 30.11.2015 at 14:15 to catch up on some topics (it will be on iTune U)
- Next week we'll talk about UI Design principles and View and Navigation Controllers