

# iOS Dev Accelerator

## Week1 Day2

- Closures
- Accounts Framework
- Social Framework
- HTTP Status Codes
- Swift Switch Statements
- Concurrency

# Homework Review





Closures



# Closures

- Closures in Swift are similar to blocks in C and Objective-C and lamdas in other languages.
- Closures are an extremely important topic, so pay attention!!
- Apple uses closures/blocks in a significant portion of their API's, so if you don't understand what they are and what they do, everybody loses, and by everybody I mean you.

# So what is a closure?

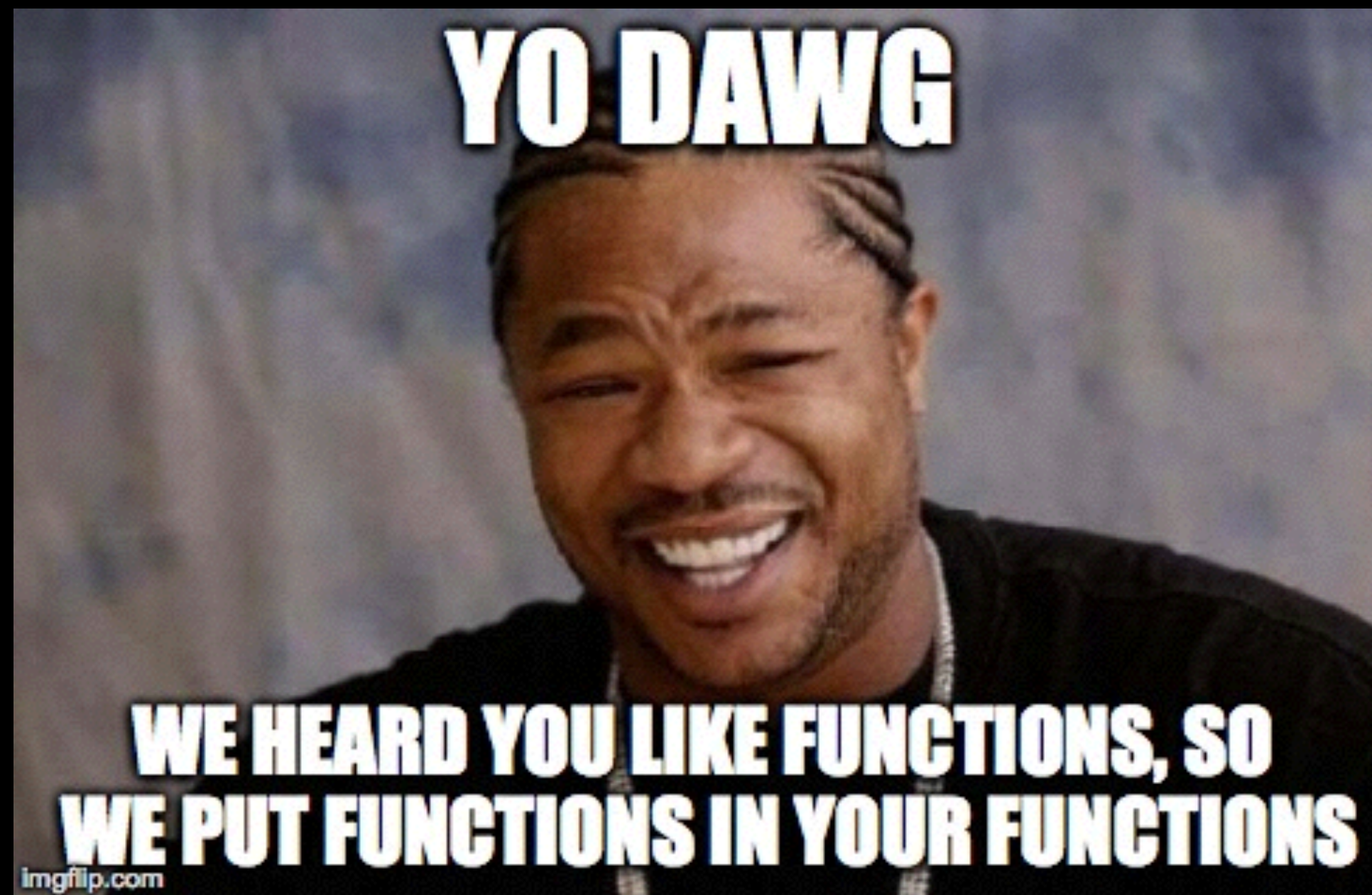
- “Closures are self-contained blocks of **functionality** that can be passed around and used in your code”
- But before we can really get into closures, we need to talk about what functions are.

# Functions

- “Functions are self-contained chunks of code that perform a specific task”
- Sounds pretty similar to closures, eh?
- Functions have names that tells the developer what the function does, and the developer uses that name to call the function.
- Methods are just functions associated with a type (class, struct, etc)
- **Functions have types, just like variables and constants.**

# Function Types

- A function's type consists of the function's parameters types and return type.
- This allows us to easily pass functions in as parameters to other functions, and also return functions as well.





# Function Types

- For example, the function:

```
func combineTwoStrings(a : String, b : String) -> String {  
    return a + b  
}
```

- Has a type of (String, String) -> String
- So we could actually store this function as a variable:

```
func combineTwoStrings(a : String, b : String) -> String {  
    return a + b  
}  
  
var combineStringFunction : (String, String) -> String = combineTwoStrings
```

- We can even pass the function in as a parameter to another function.
- All of this applies to methods to, which are just functions that are associated with a type.



Demo

# Back to closures

- Closures take three different forms:
- Global Functions
- Nested Functions
- Closure Expressions, which we will focus on today.

# Syntax

- The general of a closure expression is:

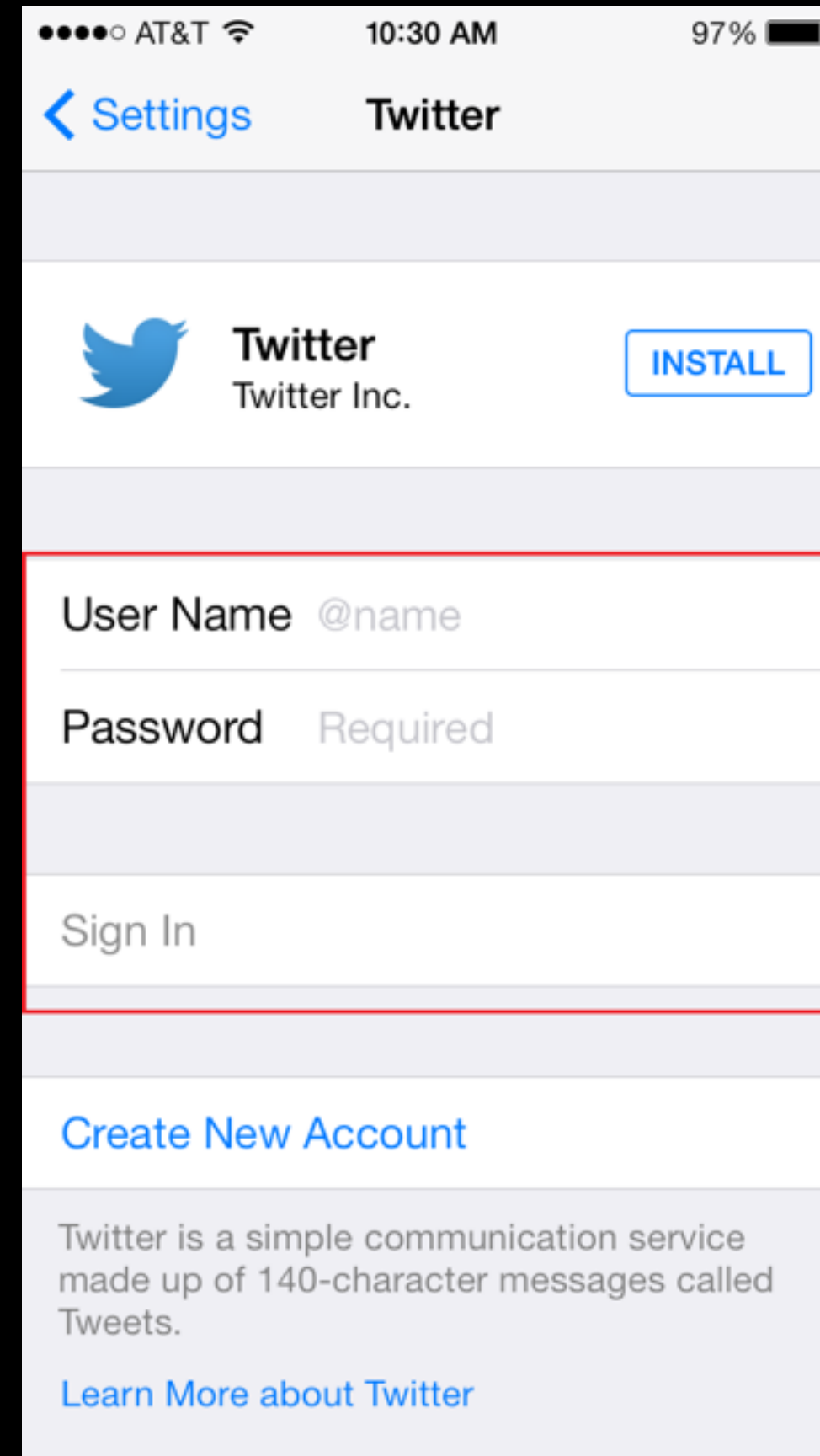
```
{ (parameters)  $\rightarrow$  return type in  
statements  
}
```



# Capturing Variables

- So why are are closures called closures?
- Closures can capture and store references to variables (and constants) from the context in which the closure was defined. This is known as *closing* over those variables.
- Each type of closure has specific rules about capturing:
  - Global functions: no capturing
  - Nested functions: captures values from the enclosing function
  - Closure expression: captures values from its surrounding context

Demo



# Accounts Framework



# Accounts Framework

- The Accounts framework gives your app access to a user's accounts stored in the 'Accounts database'.
- Each account stores credentials for a particular Service, like Facebook or Twitter.
- Currently there are 4 service types available: Facebook, Twitter, SinaWeibo, and TencentWeibo.

# Accounts Workflow

- Get a reference to the account store by instantiating an instance of `ACAccountStore`.
- Get a reference to the correct account type you are looking by calling `accountTypeWithIdentifier()` on your account store.
- Call `requestAccessToAccountsWithType(completion:)` on your account store.
- In the completion handler, check if access was granted, if its granted get an array of accounts back by calling `accountsWithAccountType()` on the accounts store.
- Prompt the user, asking which account they would like to use (or just pull object at index 0 from the array, assuming the user only has one account of that type)
- Success!





# Social Framework



# Social Framework

- “The Social framework provides a simple interface for accessing the user’s social media accounts”
- Prior to iOS6 there was only a Twitter framework. iOS6 introduced the Social framework which has support for Facebook, SinaWeibo, and TencentWeibo, in addition to Twitter.
- The social framework APIs focuses on two aspects of social integration: Requests and Composing (aka uploading).

# SLRequest

- To make a request with the Social framework, you use the class `SLRequest`.
- `SLRequest` is a easy way to configure and perform an HTTP request for one of the supported social services.
- `SLRequest` has an initializer that does most of the setup for you.
- Once the request is setup, you can fire it off with `performRequestWithHandler()`, which we will use with a closure expression!

Demo





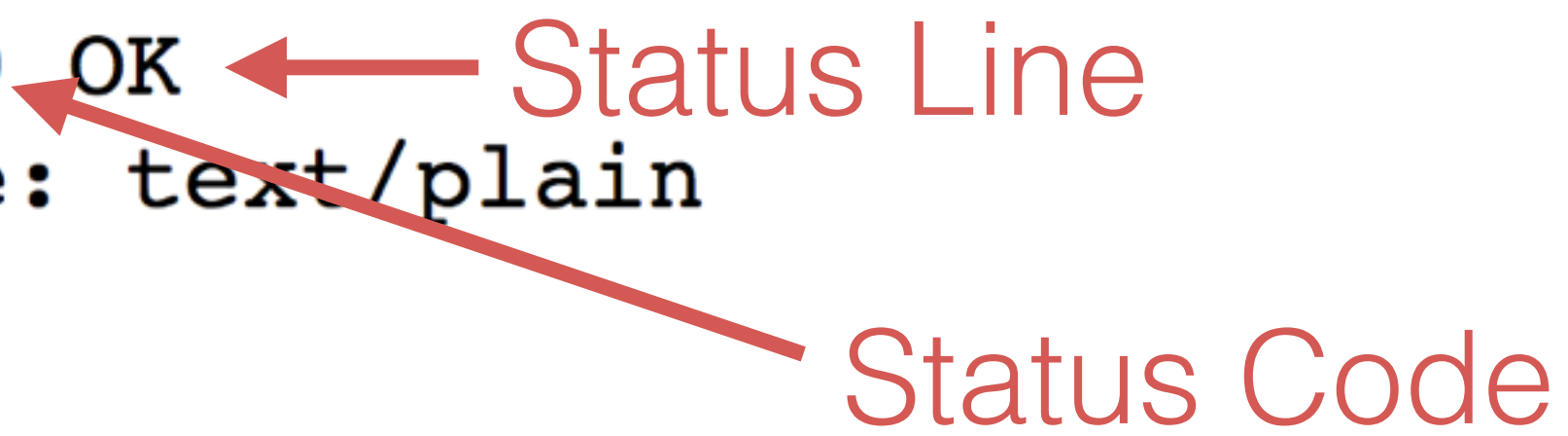
HTTP Status Codes



# HTTP Status Codes

- When a web server responds to a request, one of things that makes up its response is a status code:

```
HTTP/1.1 200 OK  
Content-Type: text/plain  
  
Hello World
```



← Status Line

Status Code

- We will learn more in depth about HTTP and HTTP methods later on.

# HTTP Status Codes

- So based on the status code in the response we get back, we can react accordingly in our code.
- Heres a general guide to the most common status codes an iOS app needs to watch for:
- 200 OK - standard response for a successful HTTP request
- 2XX - Most API services only use 200, but anything in the 200's means whatever you were trying to request worked.
- 400 Bad - bad request, most likely syntax error
- 401 Unauthorized - authentication was required but not provided or incorrect in request
- 403 Forbidden - Request was valid, but the server refuses to respond.
- 404 Not found - The requested resource was not found
- 429 Too Many Requests - rate limited
- 5xx Server Error - not your app's fault!
- **Keep in mind, its up to the server architects to decide which codes mean what, but they all generally all follow best practices.**

Demo



# Swift Switch Statements

# Switch statement

- “A switch statement considers a value and compares it against several possible matching patterns”
- Once it finds a pattern that matches, it runs a block of code for that case.
- Its like an if statement, but provides multiple potential states.

# Switch format:

```
switch (some value to consider) {  
  case (value 1):  
    (respond to value 1)  
  case (value 2),  
    (value 3):  
    (respond to value 2 or 3)  
  default:  
    (otherwise, do something else)  
}
```

# Switch statement

- Each possible case begins with the keyword `case`, and then the value for this case
- Each case must have at least one line of code that will execute if this specific case branch is chosen.
- Every switch statement in Swift must be exhaustive. Every possible value of the type being considered must be accounted for.
- If you have something like a number, where it would not be possible to have a case for every value, you can use the `default` keyword to specify a code block to run if all the other cases don't hit.

# Switch on single character

```
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    println("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l",
    "m",
    "n", "p", "q", "r", "s", "t", "v", "w", "x", "y",
    "z":
    println("\(someCharacter) is a consonant")
default:
    println("\(someCharacter) is not a vowel or a
        consonant")
}
// prints "e is a vowel"
```

# No fall through

- Once the first matching switch case is completed, the switch statement is finished.
- This is different from C and Objective-C, where you have to specify that when you intend to exit the switch statement by using the keyword break
- You can use the fall through keyword if you want to opt int to fall through behavior



# Range Matching

- “Values in a switch case can be checked for their inclusion in a range”
- We can use this for HTTP Status Code checking!

```
switch count {
case 0:
    naturalCount = "no"
case 1...3:
    naturalCount = "a few"
case 4...9:
    naturalCount = "several"
case 10...99:
    naturalCount = "tens of"
case 100...999:
    naturalCount = "hundreds of"
case 1000...999_999:
    naturalCount = "thousands of"
default:
    naturalCount = "millions and millions of"
}
```

Demo



Everyone the first time they try to comprehend concurrency programming

# Concurrency

# Concurrency Intro

- Concurrency is just a fancy word for multiple things happening at the same time.
- There are a number of different APIs an iOS developer can use to introduce concurrency to an app.



# History of Concurrency

- Before multi-core CPU's, the amount of work you could do on a computer was directly determined by the clock speed of the CPU.
- So instead of just increasing the clock speed, people realized you could just add more CPU cores to the chip.
- This way, the computer execute more instructions per second without necessarily increasing the clock speed.

# Using Concurrency

- Even if you have multiple cores, if the software you are using isn't designed to do multiple things at once, the cores go to waste.
- The traditional, 'old fashioned' way an app can use multiple cores is to create and manage multiple threads.
- So what are threads?



# Threads

- “Threads are a lightweight way to implement multiple paths of execution inside of an application”
- “At the system level, programs run side by side, with the system dolling out execution time to each program based on its needs and the needs of other programs”
- Each program can have multiple threads of execution.
- These threads are used to perform tasks simultaneously, or almost simultaneously. In a single core CPU, multithreading can give the appearance of multitasking by splitting up the tasks into slices and running the slices from different tasks one after the other.
- Using multiple threads in your app has 2 main benefits:
  - It can improve the perceived responsiveness of your app
  - It can improve the real-time performance of your app

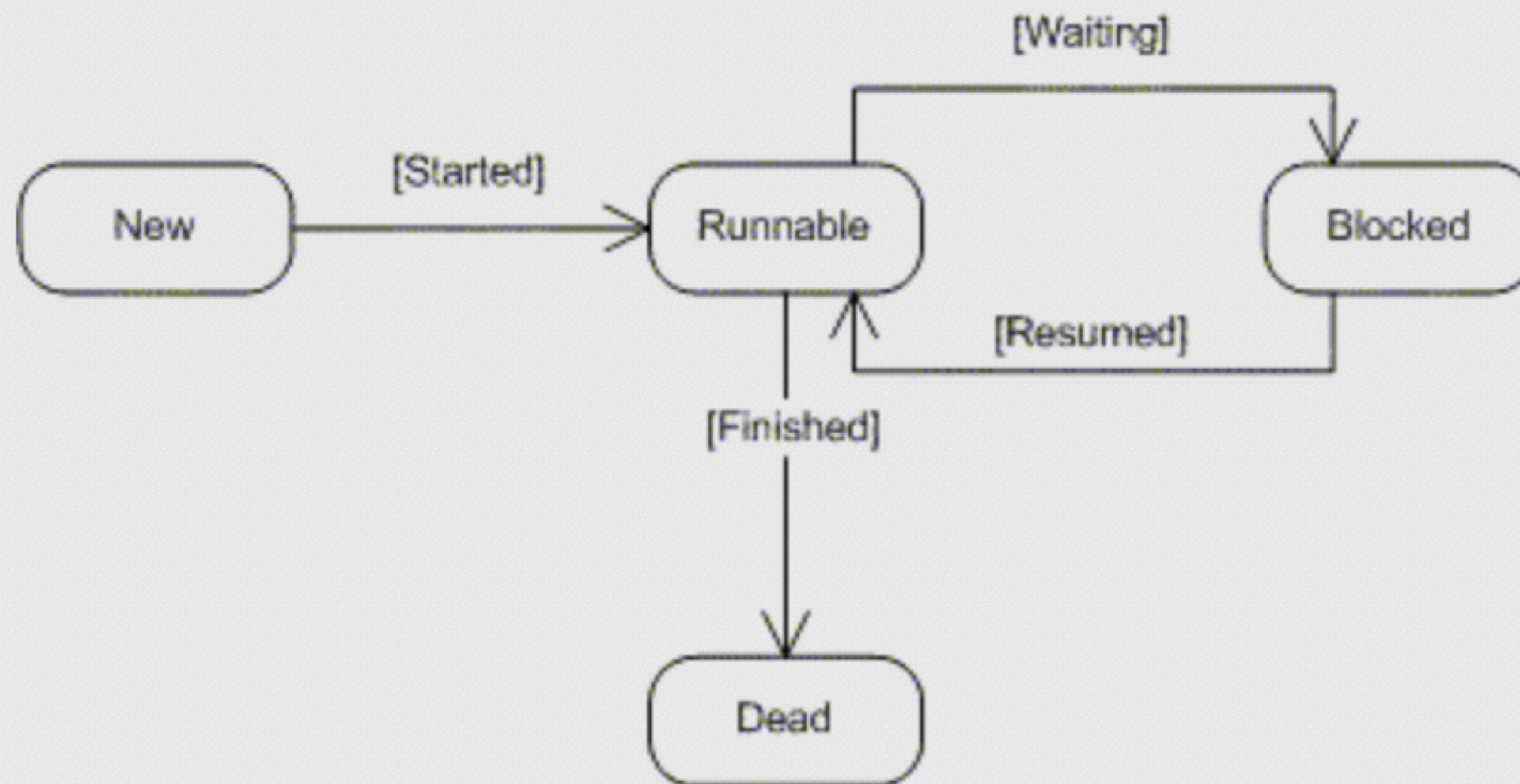
# Nobody is perfect, including threads

- Multi threading introduces a 2 big issues to your app that you'll need to account for:
  - Each thread needs to coordinate its actions with other threads to prevent memory corruption. If two threads are trying to modify the same data in your app at the same time, bad things happen.
  - All interface operations have to take place on the main thread.



# Thread states

Threads can be in 3 states:  
Running, Ready, Blocked



# Run Loops

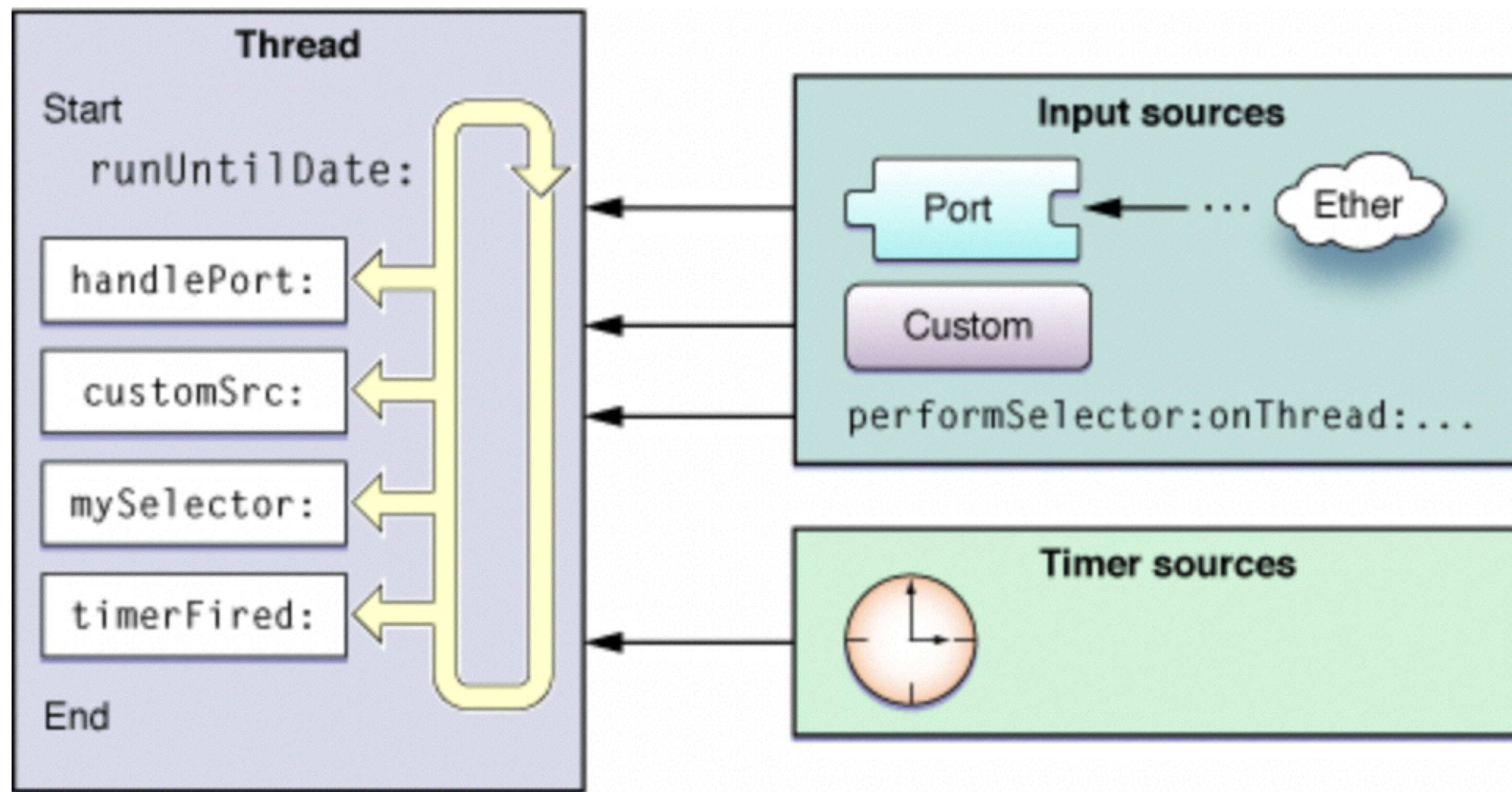
- “A run loop is a piece of infrastructure used to manage events arriving asynchronously on a thread”
- Run loops monitor for events. As events arrive the system wakes up the thread and dispatches events to the run loop, which then dispatches them to executed.



# Main event Loop

- “In the main event loop, an application routes incoming events to objects for handling, and as a result of handling, update its state and appearance”
- “The event loop is simply a run loop”
- “In all but the main thread, the run loop must be configured and run manually by your code”
- “In Cocoa applications, the run loop for the main thread - the main event loop - is run automatically by the application object”
- When your app is launched, the system sets up the infrastructure for the main event loop. It establishes all the connections it needs to deliver user events. Once this setup is done, the application is thereafter driven by external events.

# Run loops





# Laundry



# Operation Queues to the rescue

- “Cocoa operations are an object-oriented way to encapsulate work that you want to perform asynchronously.”
- Operations are added to an Operation Queue so they can be executed.
- Creating an operation queue is as simple as calling the init on NSOperationQueue
- You can add operations to a queue individually, in an array, or my favorite, by simply passing in a closure with the code you want to execute.
- NSOperationQueue has a type method for sending operations back to the main queue.



Demo