

iOS Dev Accelerator

Week1 Day2

- HW Review
- Closures
- Accounts & Social Framework
- Callbacks
- Concurrency
- HTTP Status Codes
- Swift Switch Statements

Homework Review



Closures

Closures

- Closures in Swift are similar to blocks in C and Objective-C and lamdas (or closures) 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, you're going to be in trouble — in a big way

So what is a closure?

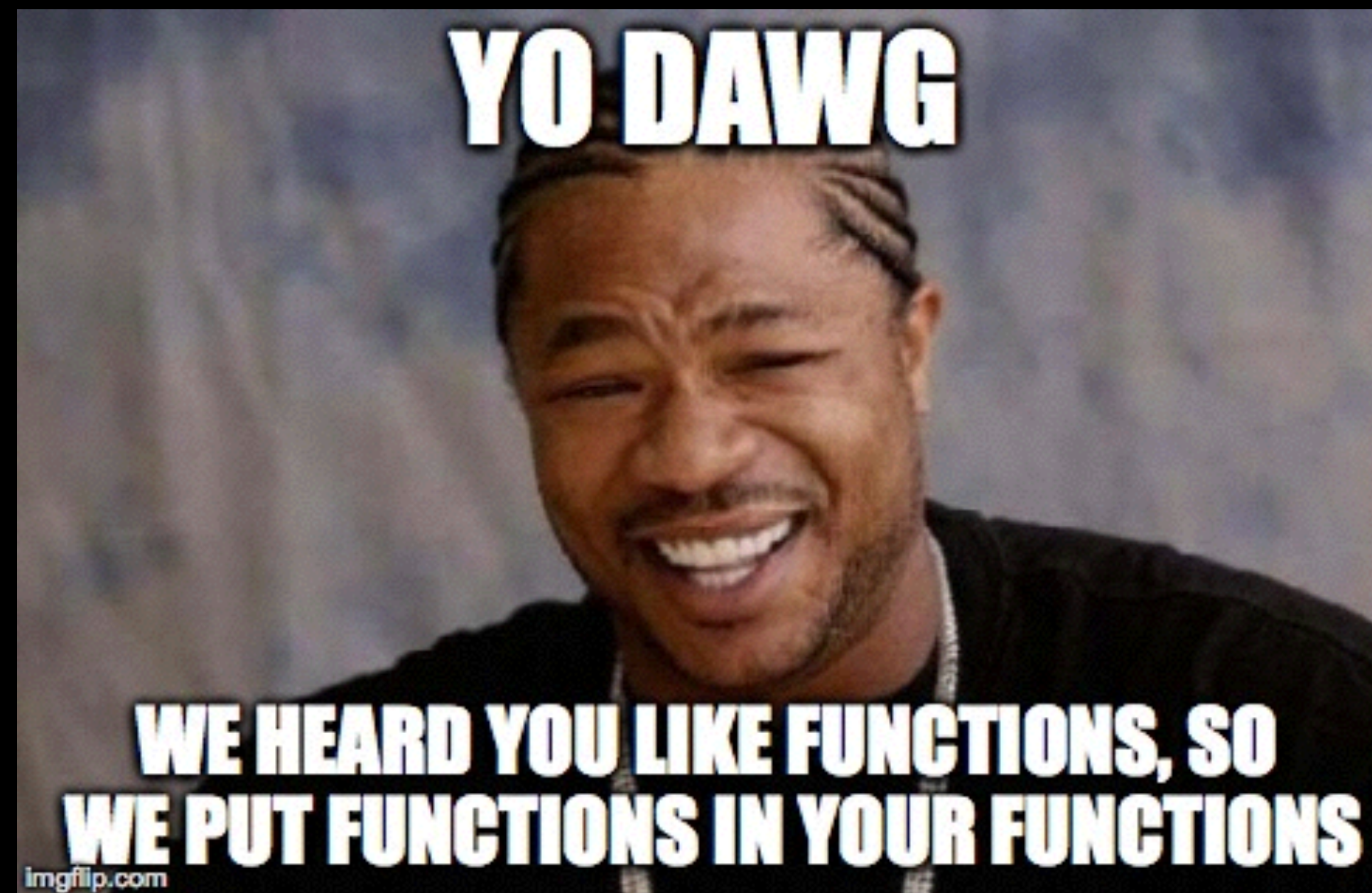
- “Closures are self-contained blocks of **functionality** that can be passed around and used in your code”
- Before we can really get into closures, we need to talk about what functions are.

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.
- Since a function has a type, that means we can store it as a variable if we want.
- **This allows us to 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, just like “Brad” would have a type of 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, and which is the main type of closure Apple sets their API up for. Whenever someone says closure while working with Swift, this is what they mean 99% of the time.

Syntax

- The general syntax of a closure expression is:

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


Capturing Variables

- So why 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

Closure Capturing

- “Closure expression: captures values from its surrounding context”
- So what does this actually mean?

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    self.tableView.dataSource = self  
    var myName = "Brad"  
    let backgroundQueue = NSOperationQueue()  
  
    backgroundQueue.addOperationWithBlock { () -> Void in  
        println(myName)  
    }  
}
```

- “The local variable myName, would ordinarily die at the end of this viewDidLoad() method. But since its being accessed in the closure, it is ‘captured’ by the closure, and kept alive until the closure is executed.

Demo

Closure Usage

- So Where are Closures commonly used in Apple's APIs?
- Many places!

Animation

- When you animate things, you simply change whatever you want to animate inside of a closure:

```
UIView.animateWithDuration(0.3, animations: { ()  
    -> Void in  
    self.view.alpha = 0  
})
```

This is one of my favorite Apple API's!

Network calls

- When using NSURLSession to make HTTP requests, you put code that you want to run after the response comes back inside of a closure:

```
NSURLSession.sharedSession().dataTaskWithRequest(request,  
completionHandler: { (data, response, error) -> Void in  
  
    //check for errors, then parse the response  
  
})
```


Swift Higher Order functions

- A higher order function is a function that does at least one of these:
 - takes a function as a parameter
 - returns a function
- Apple introduced three cool higher order functions for arrays:
- Map - Transforms an array using a function:

```
let numbers = [1,2,3,4,5,6,7,8,9,10]
```

```
let doubled = numbers.map({ (x : Int) -> Int in  
return x * 2  
})
```

Swift Higher Order functions

- Filter -Takes elements from an array that satisfy a condition:

```
let numbers = [1,2,3,4,5,6,7,8,9,10]

let evens = numbers.filter({(x : Int) -> Bool in
return x % 2 == 0
})
```

Swift Higher Order functions

- Reduce - Transforms the values in an array into a single value

```
let numbers = [1,2,3,4,5,6,7,8,9,10]
```

```
let sum = numbers.reduce(0, combine: { (x : Int, y : Int) -> Int  
    in  
    return x + y  
})
```


Closure Features

- There are many features of closures you need to know about:
- Shorthand parameter names - You can use shorthand parameter names in order to drop the first part of a closure declaration.
- So instead of naming the parameters of a closure, you can just use \$ and then numbers to refer to each parameter:

```
var numbers = [1,2,3,4,5,6,7,8,9,10]  
numbers.sort({ return $0 < $1 })
```

Trailing Closure

- If a closure is the last parameter of a function, you can write the closure after the function call:

```
UIView.animateWithDuration(0.3, animations: { () -> Void in  
    self.view.alpha = 0  
})
```

Regular

```
UIView.animateWithDuration(0.3) { () -> Void in  
    self.view.alpha = 0  
}
```

Trailing Closure

Implicit Return Values

- If a closure only has one line of code, the closure will try to return that result of that one statement (if it has a return value)

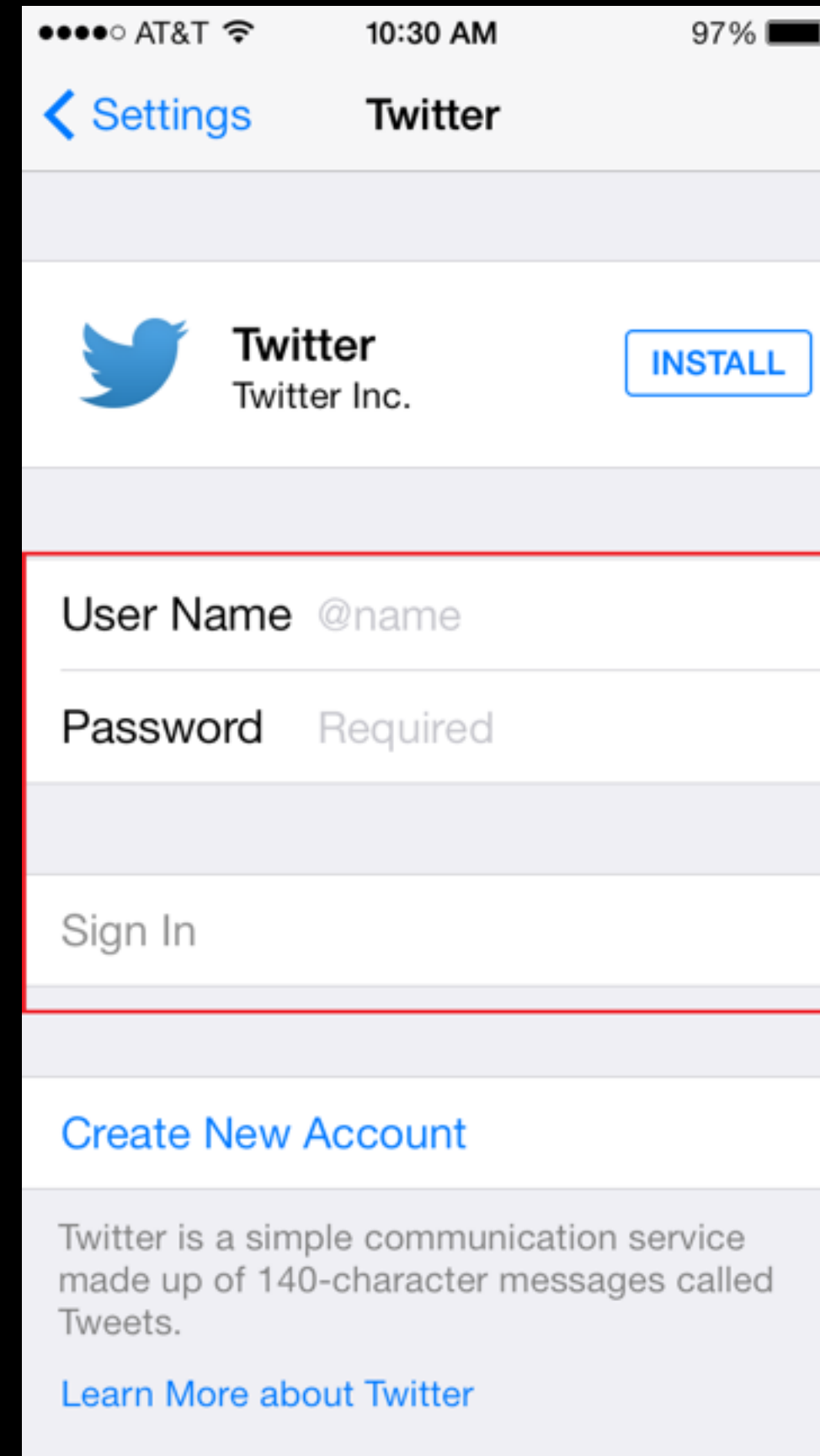
```
var numbers = [1,2,3,4,5,6,7,8,9,10]  
numbers.sort({ return $0 < $1 })
```



```
var numbers = [1,2,3,4,5,6,7,8,9,10]  
numbers.sort({ $0 < $1 })
```

Closure Memory Note

- Accessing 'self' inside of a closure can introduce something called a retain cycle in your app.
- This is only the case if the closure is being stored as a strong reference by the object representing self.
- When a closure captures a reference to a variable, it is a strong reference. So it captures a strong reference to self.
- And if the object representing self has a strong reference to the closure, then both objects have strong references to each other.
- This is a retain cycle.
- This will make much more sense once we study memory in depth later in the course.
- For now you can safely use self in your closures because we aren't capture strong references to the closures anywhere.



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, **which is a callback**, 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!

Callback

- Straight from Google: “In computer programming, a callback is a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time”
- A callback that happens at a later time is considered an asynchronous callback, which is what social framework uses!
- In addition to social, all of Apple’s networking API’s use asynchronous callbacks.

Callback

- So why do we need callbacks?
- Well, probably something like 80% of iOS apps out there make network calls. API calls, image downloads, social uploading, etc
- Network operations are considered 'blocking' operations.
- Blocking operations are operations that must wait for some event (a server to respond, the completion of a saving to disk operation)
- We should never run blocking operations on the main thread (we will look at concurrency in our next topic) because our app's interface will become unresponsive.
- The solution is to run that blocking code on a background queue (aka thread) so it doesn't slow our app's responsiveness down, and give it a callback to perform once it is done with that operation.

Synchronous Code

- Synchronous code, the style of code you are used to writing by now, is when you wait for code to finish before moving to the next task:

```
26      let myName = "Brad"  
27      let myFullName = "Brad Johnson"  
28      let x = 134 * 48  
29      self.view.removeFromSuperview()
```

line 26 is executed first

line 27 is executed after 26 has finished & returned

line 28 is executed after 27 has finished & returned

line 29 is executed after 28 has finished & returned

Asynchronous Code

- With Asynchronous code, you can move on to the next task without the previous task finishing:

```
26     let request = NSURLRequest()  
27  
28     NSURLSession.sharedSession().dataTaskWithRequest(request, completionHandler:  
29         { (data, response, error) -> Void in  
30             //callback  
31         }).resume()  
32  
33     self.name = "Downloading"
```

- Here line 26 is executed first. Once its done, line 28 is executed. Line 28 is an asynchronous method call, so the current thread returns immediately and line 32 is executed. The method call on line 28 is probably not done by the time the method on line 32 is executed. Weird!

Demo



Social Framework

Social Framework

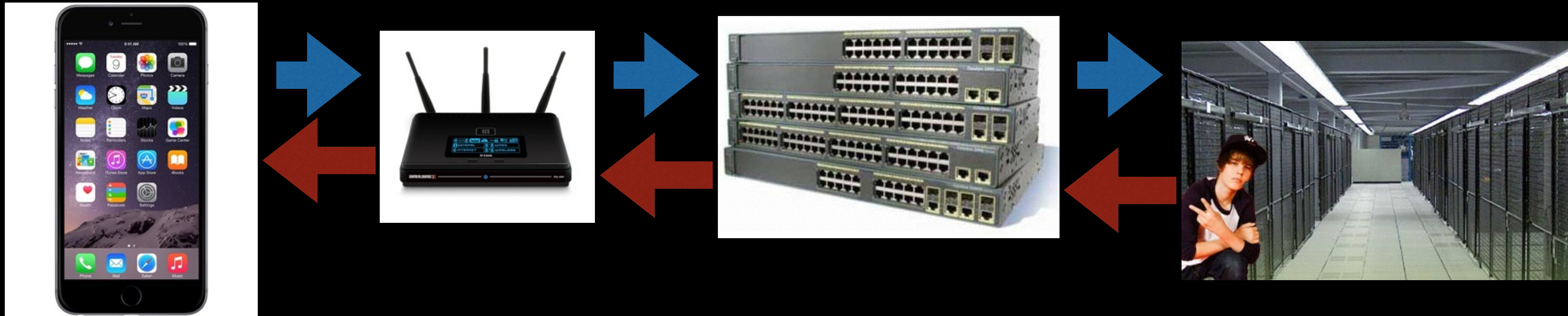
- “The Social framework provides a simple interface for accessing the user’s social media accounts”
- Prior to iOS6 there was only a Twitter framework. iOS6 introduced the Social framework which has support for Facebook, SinaWeibo, and TencentWeibo, in addition to Twitter.
- The social framework APIs 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!
It is considered a callback here, which is the next topic we will learn about.

SLRequest is cool

- Firing off your SLRequest with performRequestWithHandler is what actually packages up an HTTP Request and fires it off into crazy wilderness we call the internet:



The server gets our request, and sends a response back to the requesting client (our phone)

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. (NSOperationQueue, GCD, NSLock)

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 can 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 doling out execution time to each program based on its needs and the needs of other programs”
- Each program can have multiple threads of execution.

Threads

- 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.



In order to avoid problems with that first bullet point, you had to constantly lock and unlock your threads, which is a big pain. But now there is something better....

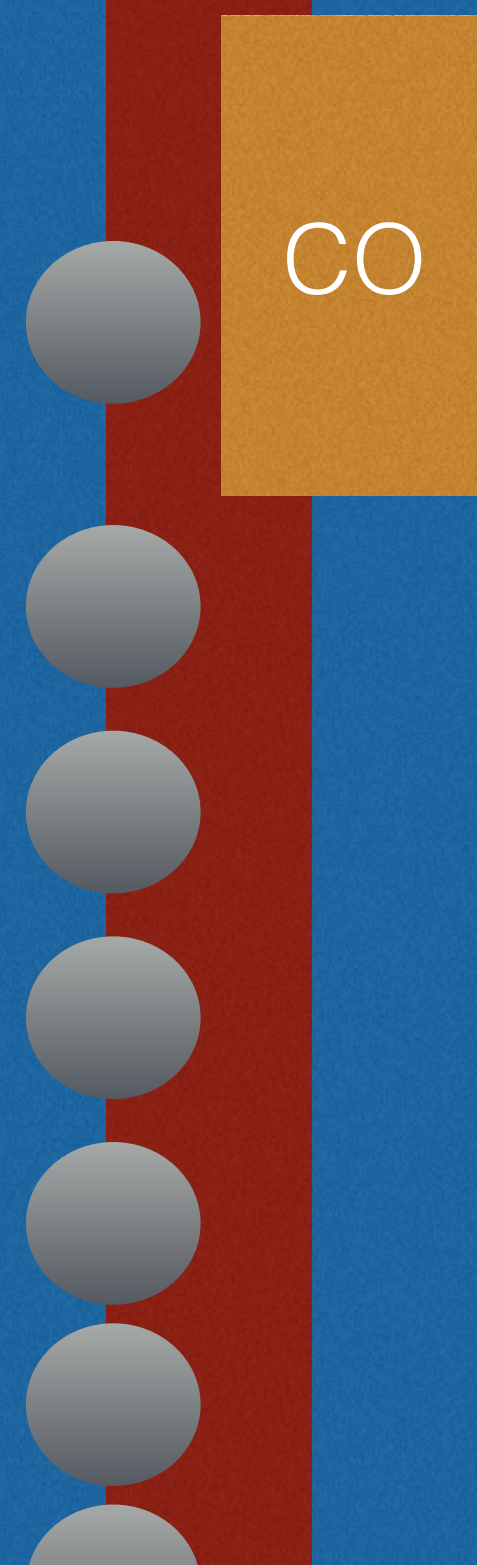
Operation Queues to the rescue

- `NSOperationQueue` is an API designed to abstract away adding operations to different threads.
- 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.

Queues vs Threads

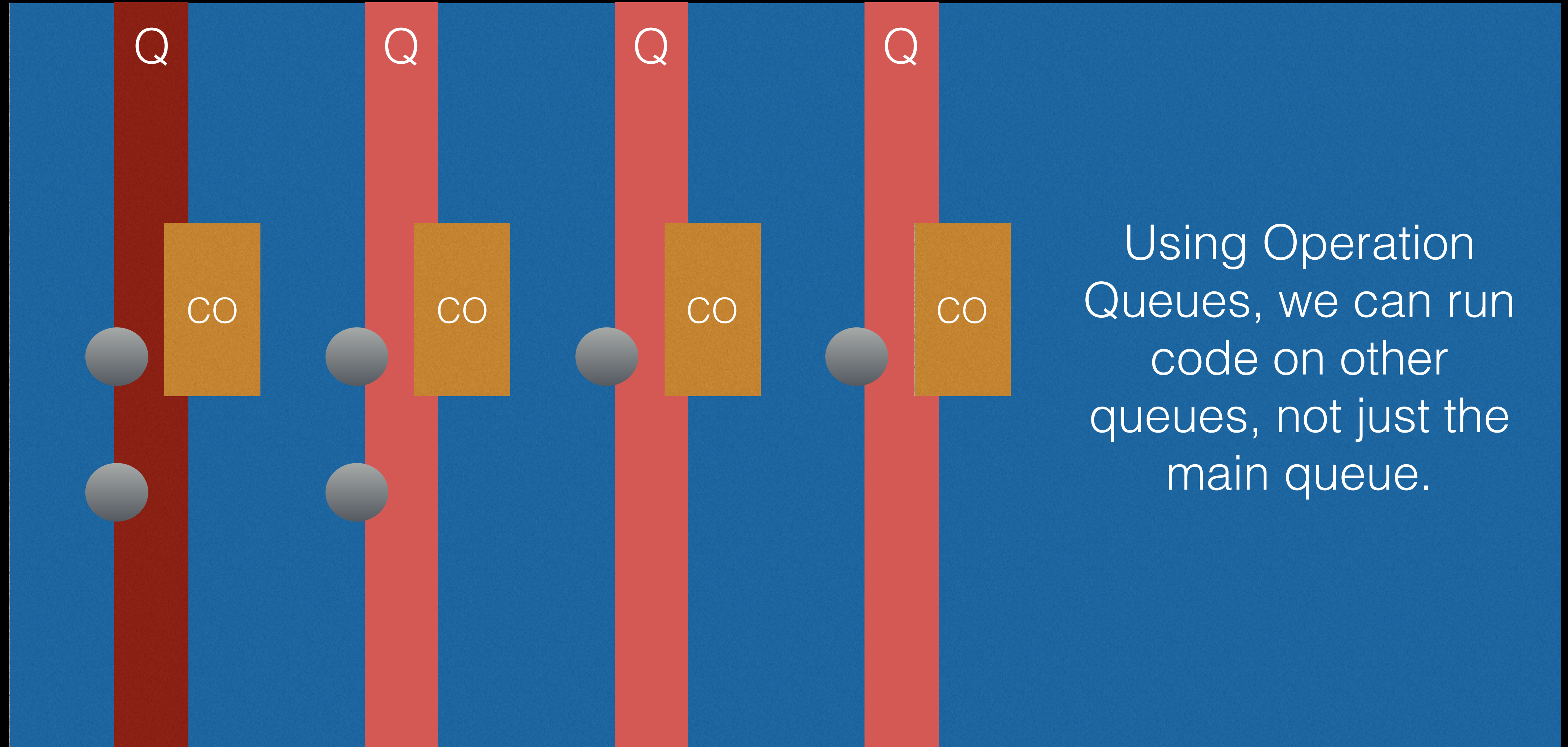
- Before Queue based API's were created by Apple, you had to directly create and add operations to NSThread objects.
- This worked, but as we discussed before, it was a very difficult API to use.
- Queues are an abstraction built on top of threads, meaning they use threads under the hood.
- A serial queue (like the main queue) uses only one thread under the hood (the main thread). Concurrent queues may use multiple threads.

Analogy



All code we have written so far is executed on the main queue (thus the main thread). By default code is executed there unless you explicitly tell it to execute somewhere else.

Analogy



Using Operation
Queues, we can run
code on other
queues, not just the
main queue.

Blocking the main thread

- You never want to run an expensive or blocking operation on the main thread/queue.
- Lets say we have a relatively expensive operation, like finding all the primes between 1 and 100000:

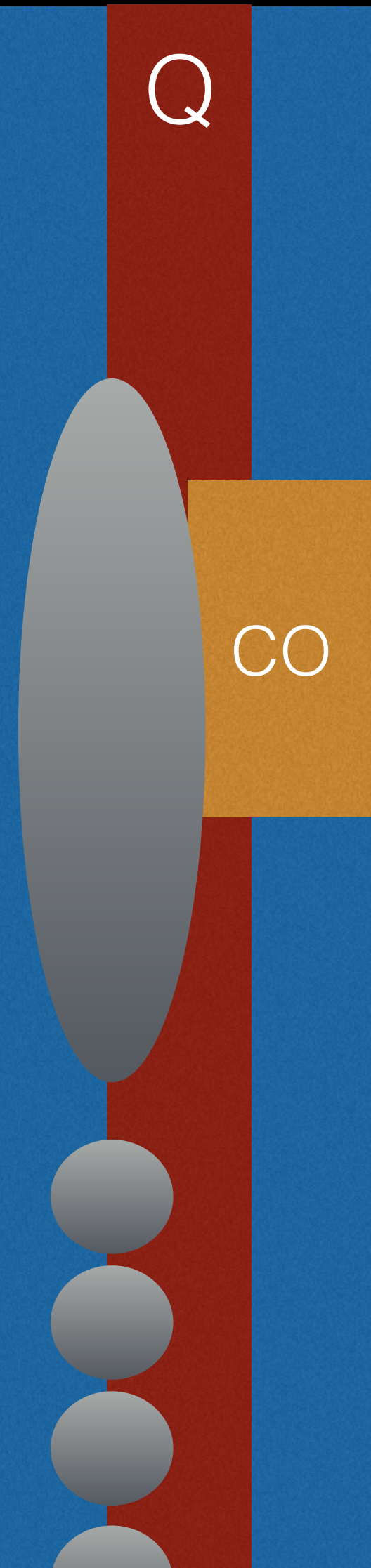
```
let numbers = 1...100000
var primes = [Int]()

for number in numbers {
    var prime = true

    for var i = 2; i <= number - 1; i++ {

        if number % i == 0 {
            prime = false
            break
        }
    }
    if prime == true {
        primes.append(number)
    }
}
```


Analogy



Running this big operation on the main queue slows the main queue down. The main queue's primary purpose is to refresh the interface. It needs to be clear of expensive operations so it can be constantly refreshing the screen and responding to user interactions (at least 60 times a second)

Sending it to a background queue

- Whenever you create an NSOperationQueue with its regular init, it will execute operations on a background thread, not the main thread:

```
var backgroundQueue = NSOperationQueue()

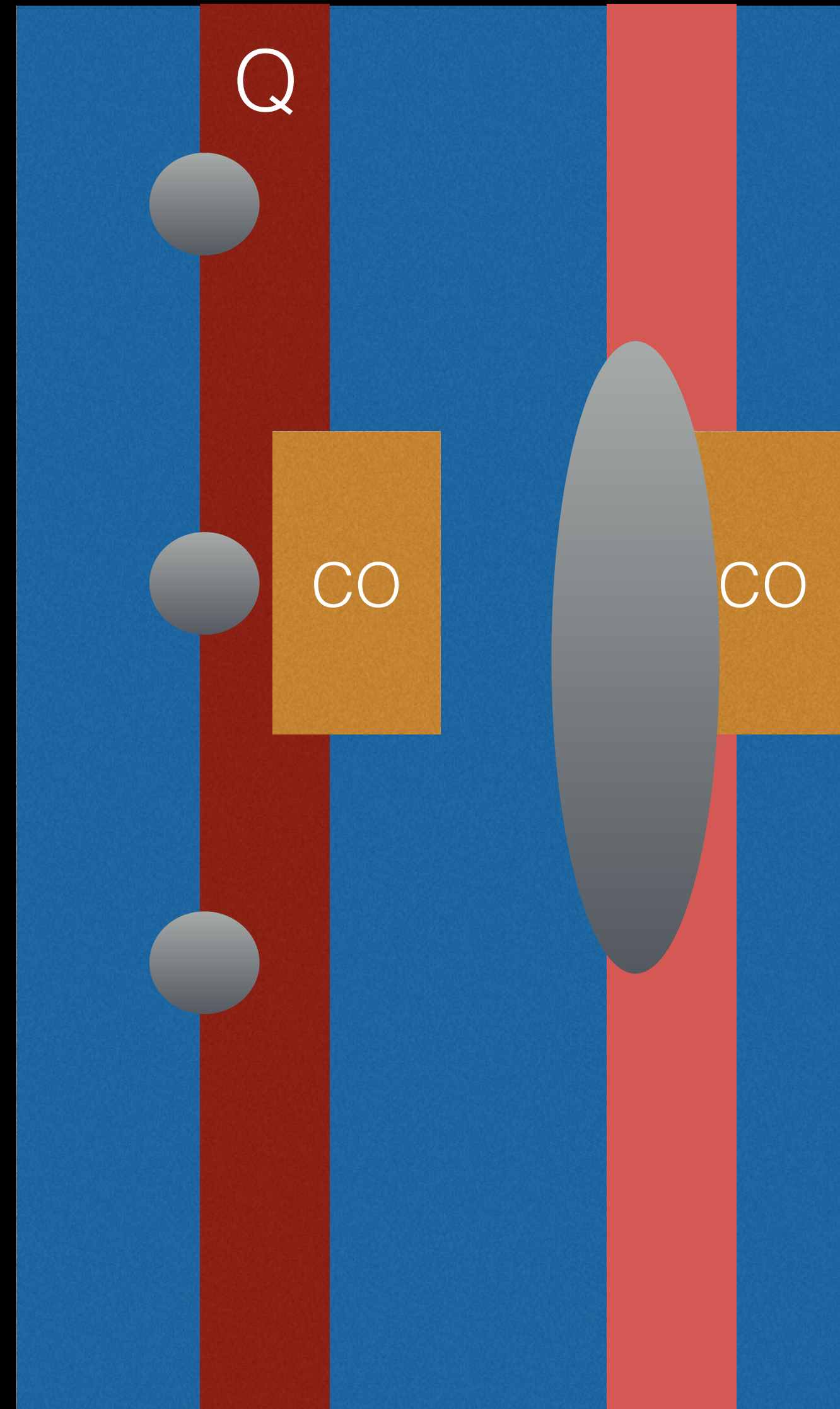
backgroundQueue.addOperationWithBlock { () -> Void in
    let numbers = 1...100000
    var primes = [Int]()

    for number in numbers {
        var prime = true

        for var i = 2; i <= number - 1; i++ {

            if number % i == 0 {
                prime = false
                break
            }
        }
        if prime == true {
            primes.append(number)
        }
    }
    println("done")
}
```


Analogy



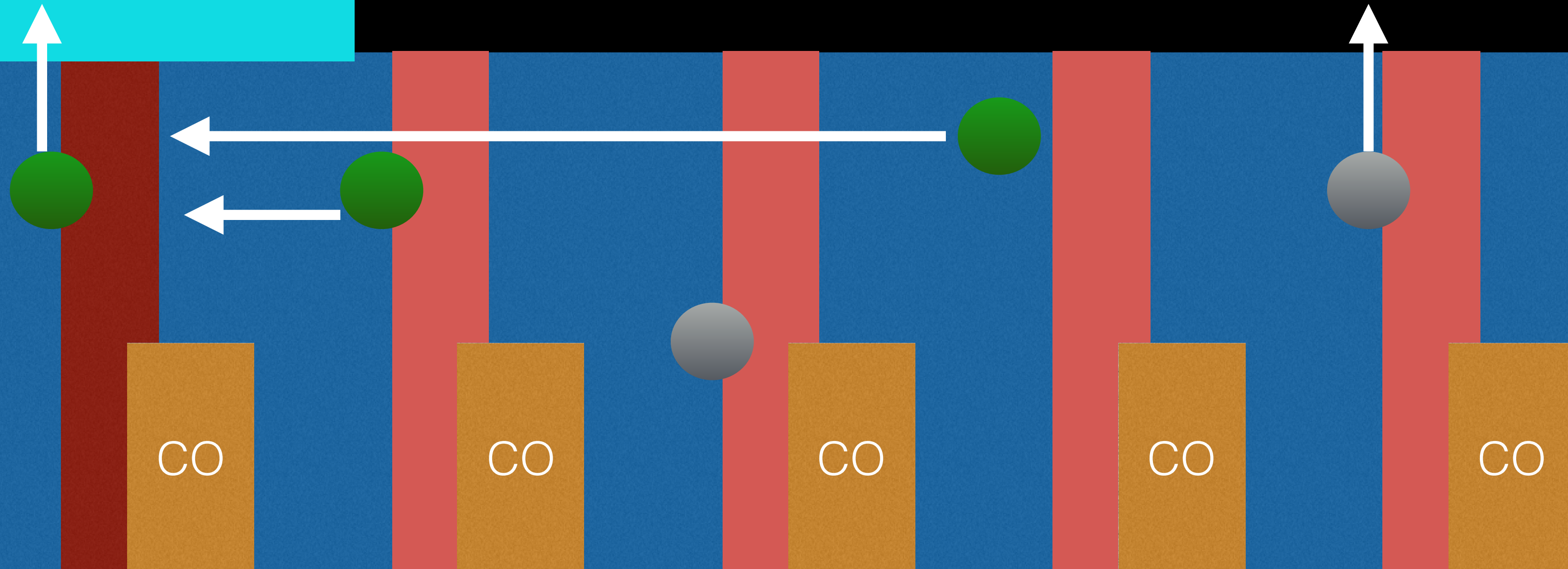
Now our main queue clear to run operations in response to the users actions, which will make our app appear much more responsive and smooth

Sending things back to the main queue

- ALL operations that change the interface MUST be executed on the main queue/thread.
- If you don't, the app will have undefined behavior, which is not acceptable to anyone (Apple will reject your app)
- This is a super important fact to know about iOS development.

Analogy

UI Garage



Sending it back main queue

- NSOperationQueue has a class method, mainQueue(), that returns the queue associated with the main thread. You can then add operations to it like any other operation queue:

```
backgroundQueue.addOperationWithBlock { () -> Void in
    let numbers = 1...100000
    var primes = [Int]()

    for number in numbers {
        var prime = true

        for var i = 2; i <= number - 1; i++ {

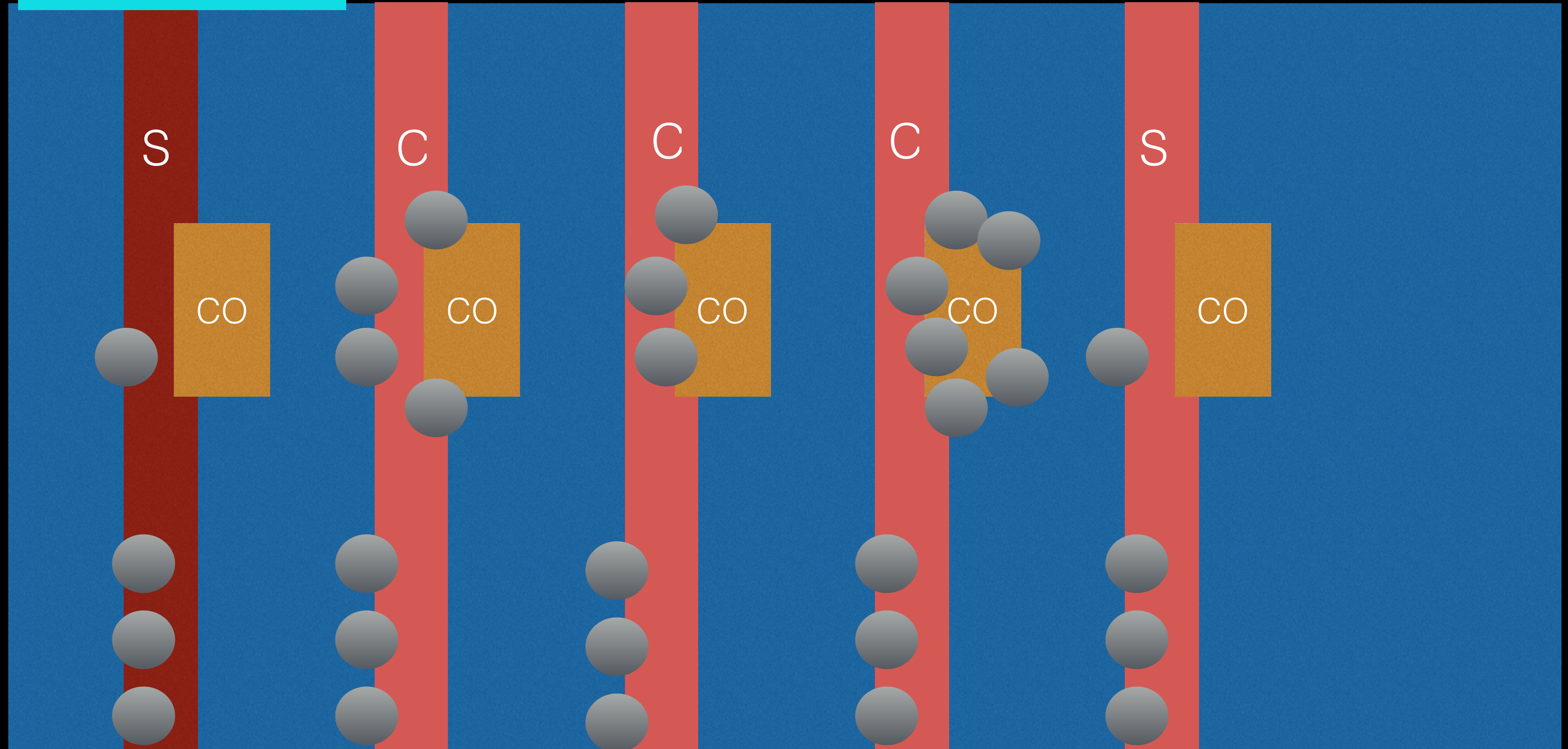
            if number % i == 0 {
                prime = false
                break
            }
        }
        if prime == true {
            primes.append(number)
        }
    }
    NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in
        self.myLabel.text = "Done!"
    })
}
```

Concurrent vs Serial

- Concurrent Queues can run more than operation at once (using multiple threads under the hood).
- Serial Queues can only run one operation at a time
- The main queue is strictly a serial queue.
- By default, an `NSOperationQueue` you create is set to be current.
- You can change that by changing its `maxConcurrentOperationCount` value. Setting this to 1 makes it a serial Queue!

UI Garage

Analogy



Demo

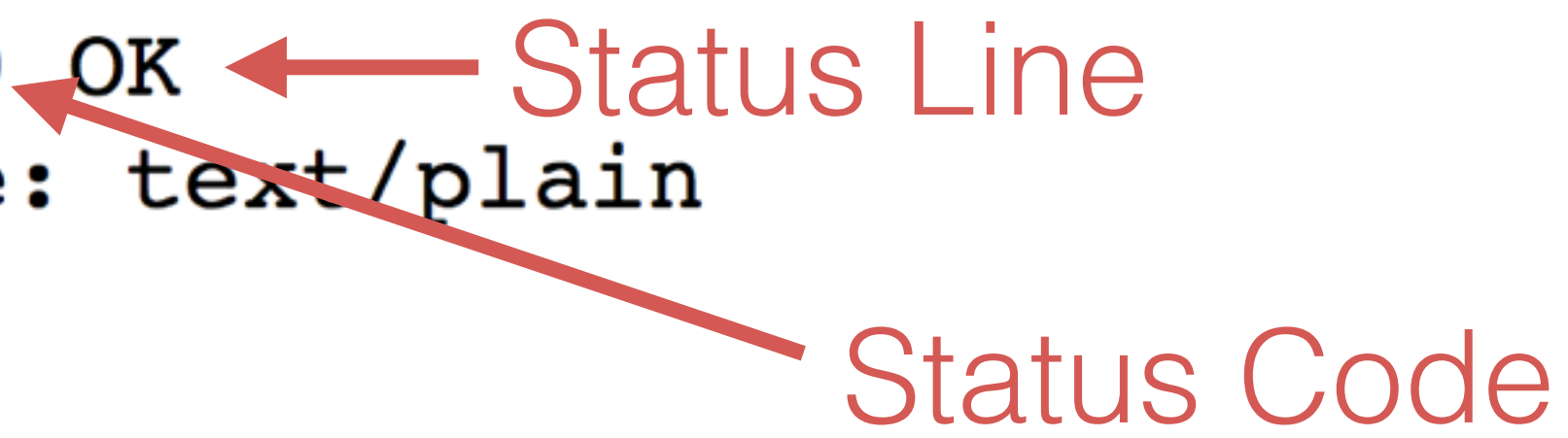


HTTP Status Codes

HTTP Status Codes

- When a web server responds to a request, one of the 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

- 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.
- 4XX - This is the clients fault (aka our app). This is usually due to wrong URL, bad authentication, or asking for a resource that isn't there
- 5xx Server Error - not your app's fault! Tell the user the servers are currently down
- **Keep in mind, its up to the server architects to decide which codes mean what, but they all generally all follow these best practices. Always check the API docs**

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