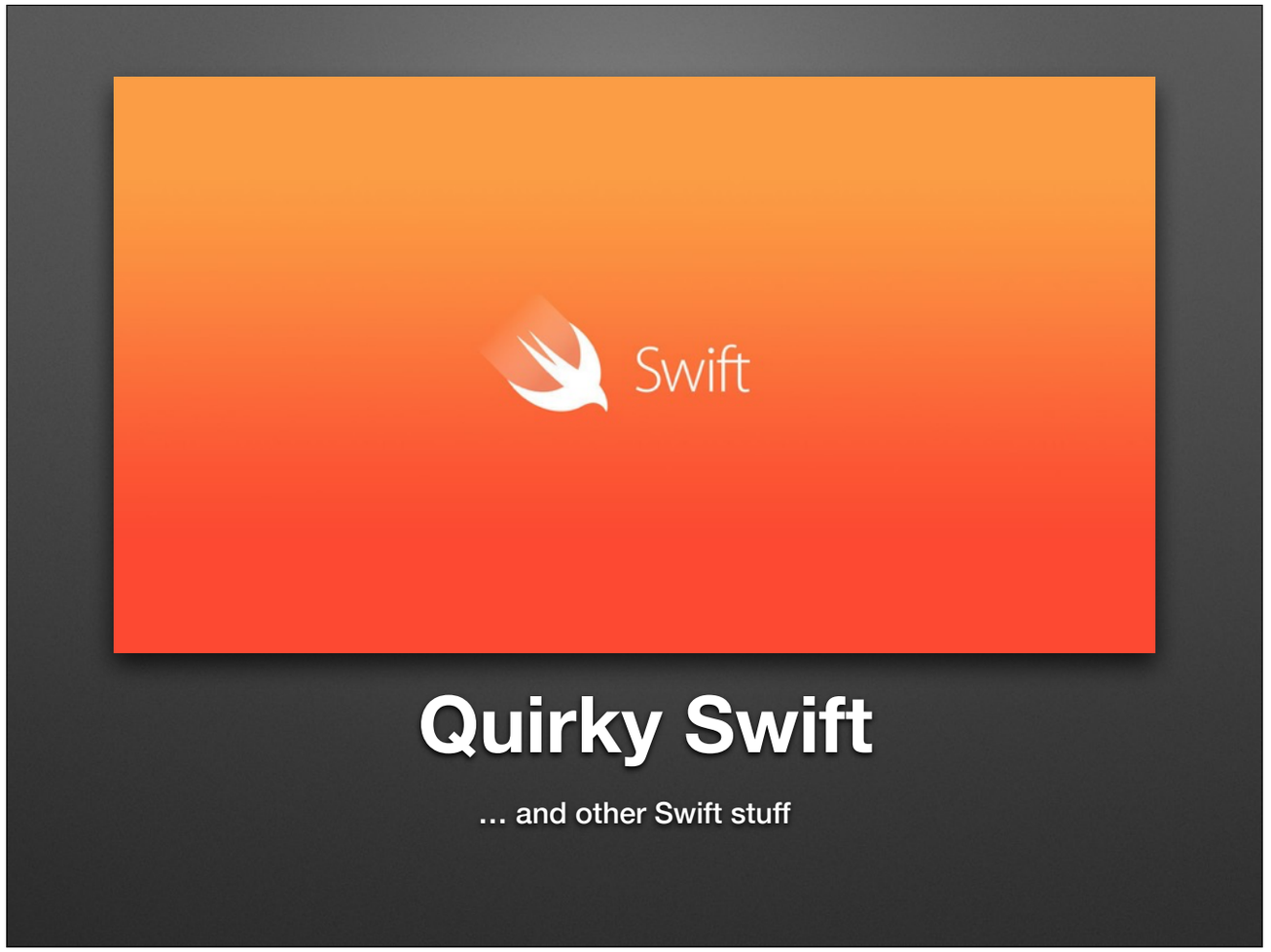

Swift
Quirky Swift
... and other Swift stuff

Overview

- Getting To Know Swift
- Confusing Stuff
- Advanced Stuff

Brief overview.
We are going to go over some of the obvious nice things that swift has to offer. Some that maybe you overlooked the first time.
Then we will go over some Confusing stuff. i.e. This Core Graphics call requires an `UnsafePointer<CGFloat>`, how do I make one of those.
Then the obligatory Monads JK no Monads in this talk, Currying, Type Inference conversation.

If you have been waiting to adopt swift. Now might be a pretty good time. The compiles got a lot faster in the last release. There is a solid baseline of simple questions and answers on stack overflow.
WWDC is in 2 weeks and everything could change or break again.

If your body is ready for Swift. These are my recommended Learning Resources.

Apple Swift Book

Seriously read most of it. But especially check out the crazy looking section at the back. This is where the magic is at. Maybe just read the back section. You can learn about cool things like …

Literal	Type	Value
`__FILE__`	String	The name of the file in which it appears.
`__LINE__`	Int	The line number on which it appears.
`__COLUMN__`	Int	The column number in which it begins.
`__FUNCTION__`	String	The name of the declaration in which it appears.

println()

I should have written my own custom one a long time ago.

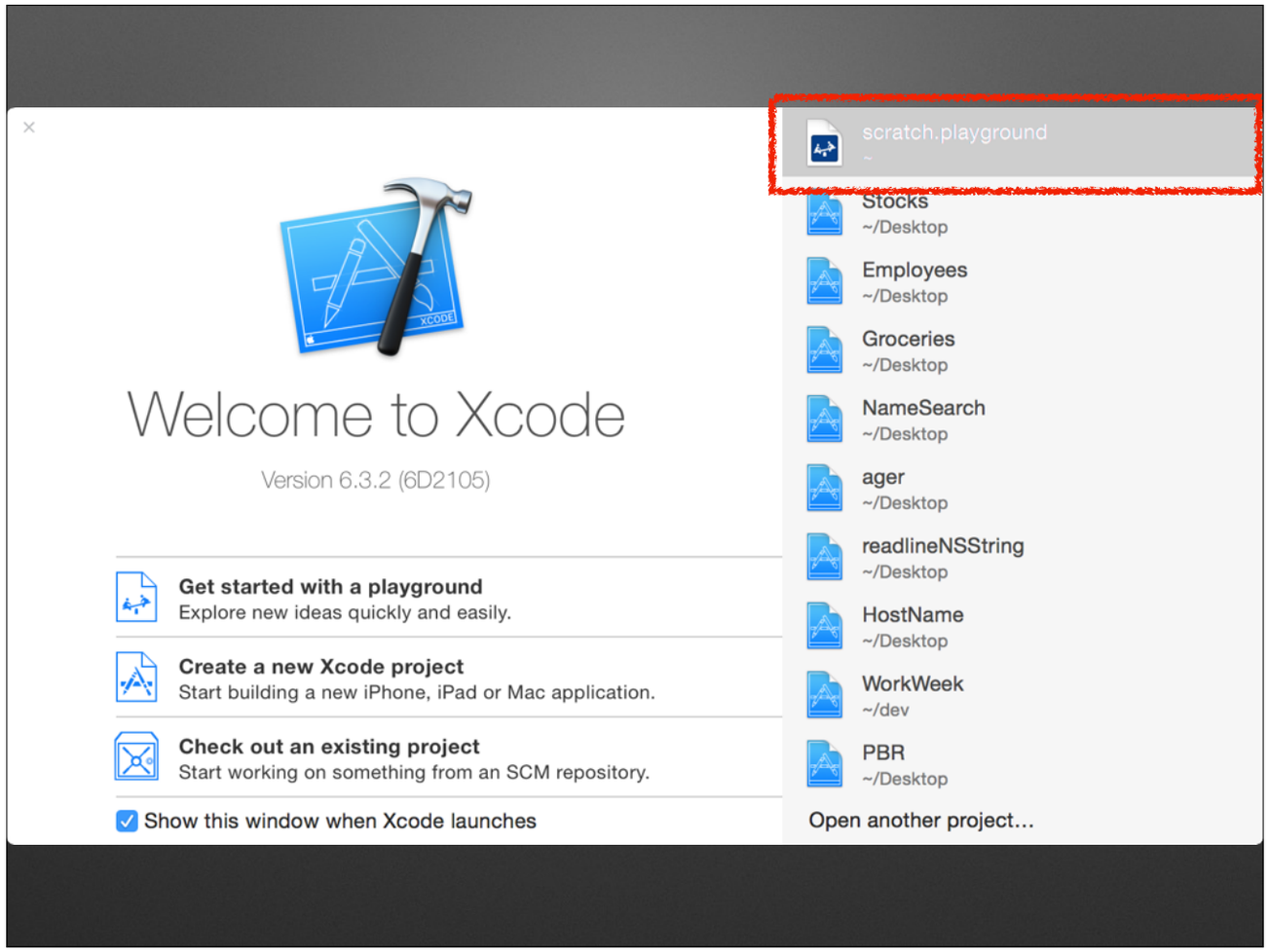
This is some cool stuff you can only learn by reading the back section of the apple swift book.

There was an amazing looking conference in London. a few months ago. The videos are coming out now. They are really great. Good variety of topics.
I will for sure be going to this next year.

Functional Swift Conference: Checkout some of the videos, Even if you really aren't sold on the whole functional thing. There are important Ideas Here. Avoiding State, Value Types over Reference Types, A really solid look at Generics. There are videos for this as well.

http://2014.funswiftconf.com

Her unicorn is named Swift-Wind so… I found this while preparing for this talk.



Finally Do some coding your self. See what stuff you can make.
Hot tip: Make a scratch.playground where you can just open it really quick with spotlight and try some stuff out.

```swift
import XCPlayground

func XCPCaptureValue<T>(identifier: String, value: T)

func XCPSetExecutionShouldContinueIndefinitely(continueIndefinitely: Bool = default)

func XCPExecutionShouldContinueIndefinitely() -> Bool

func XCPShowView(identifier: String, view: UIView)
```

Enums

Are awesome.

Have methods.

Conform to Protocols

Are not Just Ints.

```
enum CompassPoint {
    case North
    case South
    case East
    case West
}
```

```
enum Barcode {
    case UPCA(Int, Int, Int, Int)
    case QRCode(String)
}
```

```
enum ASCIIControlCharacter: Character {
    case Tab = "\t"
    case LineFeed = "\n"
    case CarriageReturn = "\r"
}
```

```
enum ConstructionStatus {
  case OnTime
  case Delayed(Int)
}

var deathStarConstructionStatus = ConstructionStatus.Delayed(7)

switch deathStarConstructionStatus {
case .OnTime:
  println("The Death Star is complete. The Emperor is pleased.")
case .Delayed(0...3):
  println("Construction is slightly behind schedule.")
case .Delayed(4...6):
  println("Construction is behind schedule.")
case .Delayed(let months):
  println("Construction is \(months) months behind. " +
    "The Emperor is most displeased.")
}
```

Swift enums are awesome, I love this example of where one value has an associated value and the other does not.

Code Organization

```swift
class MyTableViewController: UITableViewController {

    var data = Array<String>()
    override func viewDidAppear(animated: Bool) {
        tableView.reloadData()
    }
}
```

Wouldn’t it be great if your massive view controller could be smaller?

```swift
extension MyTableViewController: UITableViewDelegate {

    override func tableView(tableView: UITableView, heightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {
        return 80
    }
}
```

Put your delegates and datasources in different files. Once there are nailed down you can go back to your tableViewController and focus on view lifecycle and other important stuff.

```swift
extension MyTableViewController : UITableViewDataSource {

    //NO Stored Properties here
    var itemsCount: Int {
        return data.count
    }

    override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier("CELL", forIndexPath: indexPath) as! UITableViewCell
        return cell
    }

    override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
        return 1
    }

    override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return itemsCount
    }

}
```

You can not put Initializers, or stored properties in these. Not sure why. Maybe a planned future expansion. Current Limitation. I don’t know.

Default Parameters

Talk about passing the __FILE__, __FUNCTION__, and __LINE__ macros.
Dependency Injection. Decouple your code. Built in flexibility.

```swift
public func addArrival(_ date: NSDate = NSDate()){

    let newArrival = Event(inOrOut: .Arrival, date: date)
    eventsForTheWeek.addObject(newArrival)
}

addArrival()
let someCrazyInterval = 24*60*60* //MORE NUMBERS
addArrival(NSDate(timeIntervalSince1970: someCrazyInterval))
```

More flexible. Most of the time we just want to call `addArrival()` but if and when it becomes important to add a different time we can pass in our own date.

```swift
func postNotification(center: NSNotificationCenter = NSNotificationCenter.defaultCenter()){
    let note = NSNotification(name: "WorkWeekUpdated", object: nil)
    println("Posting Notification: \(note)")
    center.postNotification(note)
}
```

Dependency Injection Example.

Swift StdLib

SwiftDoc.org

Is filled with a bunch of Framework Agnosic stuff. Bounce out to SwiftDoc.org Show list of Types, Protocols, Functions.

Swift Protocols

Continuing on SwiftDoc.org

Swift Protocols are a big deal. Also a confusing deal. Lets look at an example.
If we want to use `reduce` we look it up on swiftdoc.org then see that it takes a SequenceType… what the hell is that, do i have one. Looking at the protocol Hierarchy we can see that both Array and Dictionary conform to this. Awesome. Lets reduce some stuff.

```
let stooges = ["Moe", "Larry", "Curly"]                  ["Moe", "Larry", "Curly"]

var all = reduce(stooges, "", {                          " Moe Larry Curly"
    $0 + " " + $1                                        (3 times)
})

all // " Moe Larry Curly"                                " Moe Larry Curly"

extension String {
    mutating func dropFirstChar(){
        self.removeAtIndex(self.startIndex)              " "
    }
}

all.dropFirstChar() // "Moe Larry Curly"                 "Moe Larry Curly"
```

A simple example of using reduce on an array, then fixing our mistake with an extension on string.

NOTE: There is a function in the STDLIB join which will do this for us.

Stuff You Might Run Into

Casting

is as? as!

You will for sure run into this.

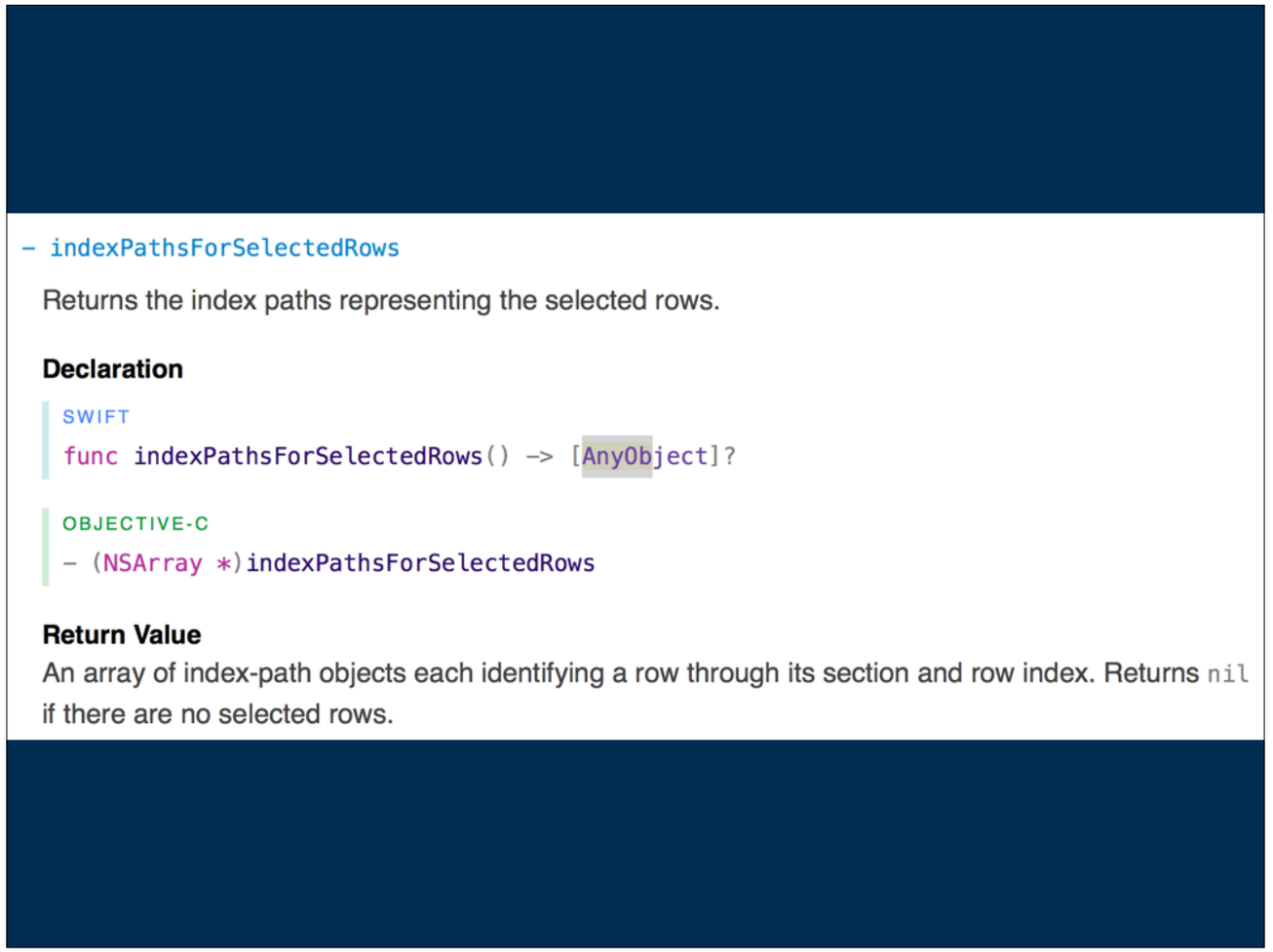
I still use as! even though it has the explode operator appended on the end.

```swift
let item: MyClass = MyClass()

if item is MyClass {
    //do MyClass specific stuff
}
```

Simple type checking. I am not sure what this does for super classes? or subclasses?

```swift
class B {
    func myFunc(){ println("B myFunc") }
}

class I: B{
    override func myFunc() { println("Inherited overridden myFunc") }
}

let b:B = I()
b.myFunc() //"Inherited overridden myFunc"

let i = b as? I
i?.myFunc() // "Inherited overridden myFunc"

let t = b as! I
t.myFunc() // "Inherited overridden myFunc"
```

Compile time type checking can be different than the run time execution.



So really it returns an optional Array of NSIndexPaths just force cast this guy right away. as! FTW

Properties

Stored Properties

```
struct Scene {
    var background: UIView
}
```

stored properties are pretty easy and work as expected.

```
struct MyStruct {
    var notify: ()-> Void = {
        let note = NSNotification(name: "NOTE", object: nil)
        NSNotificationCenter.defaultCenter().postNotification(note)
        println("Original Notification")
        }
}

var c = MyStruct()
c.notify()
c.notify = { println("Notify Called") }
c.notify()
```

Interesting stored property, Look closely this is a closure/ block saved as a property on the object.
It is also a var, meaning that it can be re defined later. For flexibility.

Computed Properties

```
extension NSDate {
    var isInThePast: Bool {
        return self.timeIntervalSinceNow < 0
    }
}

let d = NSDate(timeIntervalSince1970: 0)
d.isInThePast
```

```
true


"Dec 31, 1969, 6:00 PM"
true
```

Must give it a type. The inference engine can not figure out what your computed property is yet.
Not that we could have defined this as a computed property or a method or a closure, or …

WAT

```
4  import UIKit
3
2  struct Scene {
1      var color = UIColor.redColor()
5      var cg = color.CGColor
1  }
2
```

'Scene.Type' does not have a member named 'color'

Why does this not work!

The scene has not been initialized yet so you can’t refer to other properties of it.
cg must be declared as lazy var cg: CGColor = {self.color.CGColor}()

Let’s get Lazy

```
public class DayTimePicker: NSObject {
    lazy var dateFormatter: NSDateFormatter = {
        return NSDateFormatter()
    }()
    private let calendar: NSCalendar
    public init(calendar: NSCalendar = NSCalendar.currentCalendar()){
        self.calendar = calendar
    }
}
```

Look at those awesome Parens

Invoke the lazy! Use lazy for expensive things that may not be used.

```
class LM {
    lazy var manager: CLLocationManager = {
        let m = CLLocationManager()
        m.desiredAccuracy = kCLLocationAccuracyBest
        m.distanceFilter = 200
        m.pausesLocationUpdatesAutomatically = true
        return m
    }()
}
```

Great for objects which require some configuration before use.

Property Observers

```
class Person {
    var name = "Bill" {
        willSet { println("Setting name to \(newValue)")  }
        didSet  { println("Changed name from \(oldValue)")}
    }
}
```

newValue and oldValue are language given. You can use your own names if you want

willSet(myCustomName) { println(“Changing to \(myCustomName)”) }

```
class Person {
    let dateOfBirth: NSDate
    var age: Int {
        get {                                   WillSet variable may not also have a set specifier
            let years = 60 * 60 * 24 * 365.0
            return Int( -dateOfBirth.timeIntervalSinceNow / years)
        }
        set {
            //some math here, change dob to match age given
        }
        willSet {
            println("Changed to \(newValue)")
        }
    }
    init(dob: NSDate){
        dateOfBirth = dob
    }
}
```

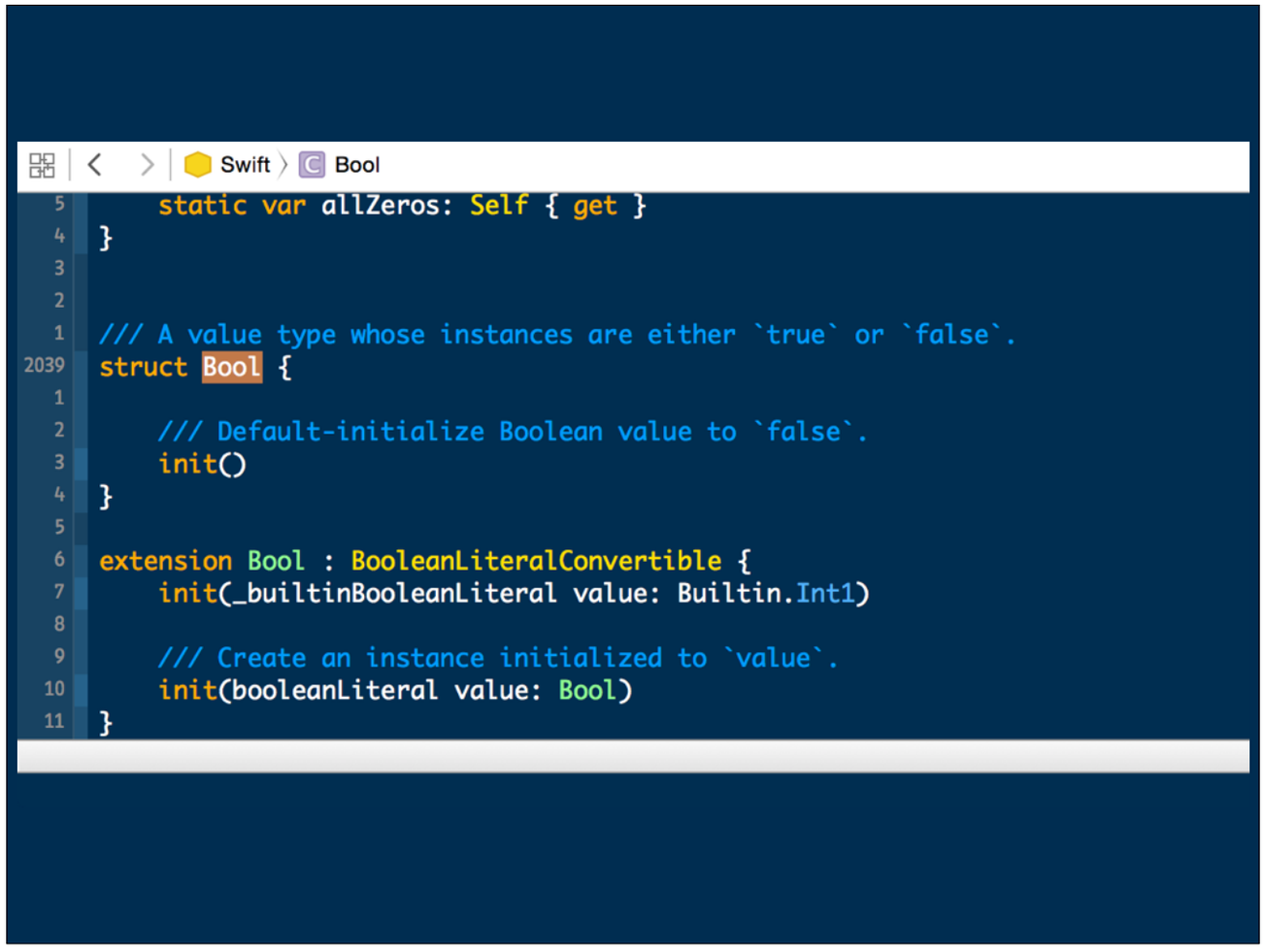
For computed Properties just put the willSet, didSet stuff right inside the get set blocks.

Switch on Bool

```
let imHere = true

switch imHere {
case true:
    println("Horray! Hope you are having fun")
case false:
    println("Bummer, Maybe It will be recorded")
//default:
//    println("HM.. Not sure")
}
```

Switch must be exhaustive, consider adding a default clause

Must add a default case for the Bool Why?



Looks like bool is built up from a Builtin.Int1 but the compiler is not smart enough to know that that can only have a true or false.
Note Bool is not an Enum? Wonder why.

Map

```
4
3  let stooges = ["Moe", "Larry", "Curly"]
2
1  let tooges = stooges.map{                 Cannot invoke 'map' with an argument list of type '((_) -> _)'
7      var name = $0
1      name.removeAtIndex(name.startIndex)
2      println(name)
3      return name
4  }
5  tooges
6
7  // Want ["oe", "arry", "urly"]
8
```

Looks easy. We take in an array of strings, and return a modified string.

```
let stooges = ["Moe", "Larry", "Curly"]

let tooges = stooges.map{ (stooge: String) -> String in
    var name = stooge
    name.removeAtIndex(name.startIndex)
    println(name)
    return name
}
tooges
```

```
["Moe", "Larry", "Curly"]

["oe", "arry", "urly"]
(3 times)
(3 times)
(3 times)
(3 times)

["oe", "arry", "urly"]
```

You could also just annotate the return value as
let tooges: [String] = stooges.map{ …

I think this is a slightly nicer way.

Pointers

```
CGColorCreate(CGColorSpaceCreateDeviceCMYK(),
    components: UnsafePointer<CGFloat>)
```

```
var darkGrey: [CGFloat] = [ 0.2, 0.2, 0.2, 1.0 ]

let color = CGColorCreate(CGColorSpaceCreateDeviceRGB(), &darkGrey)
UIColor(CGColor: color)
```


Just try to pass an address using the &.
In this case the function requires an UnsafePointer<> but it would not let me declare `let darkGrey: …` in order for the function to take it I needed to declare it as a var.

NSData

- `getBytes:length:`

Copies a number of bytes from the start of the receiver's data into a given buffer.

Declaration

SWIFT

```
func getBytes(_ buffer: UnsafeMutablePointer<Void>, length length: Int)
```

OBJECTIVE-C

```
- (void)getBytes:(void *)buffer length:(NSUInteger)length
```

Parameters

buffer	A buffer into which to copy data.
length	The number of bytes from the start of the receiver's data to copy to *buffer*.

Discussion

The number of bytes copied is the smaller of the *length* parameter and the `length` of the data encapsulated in the object.

Another API expecting a pointer. This one is pretty easy to use as well.

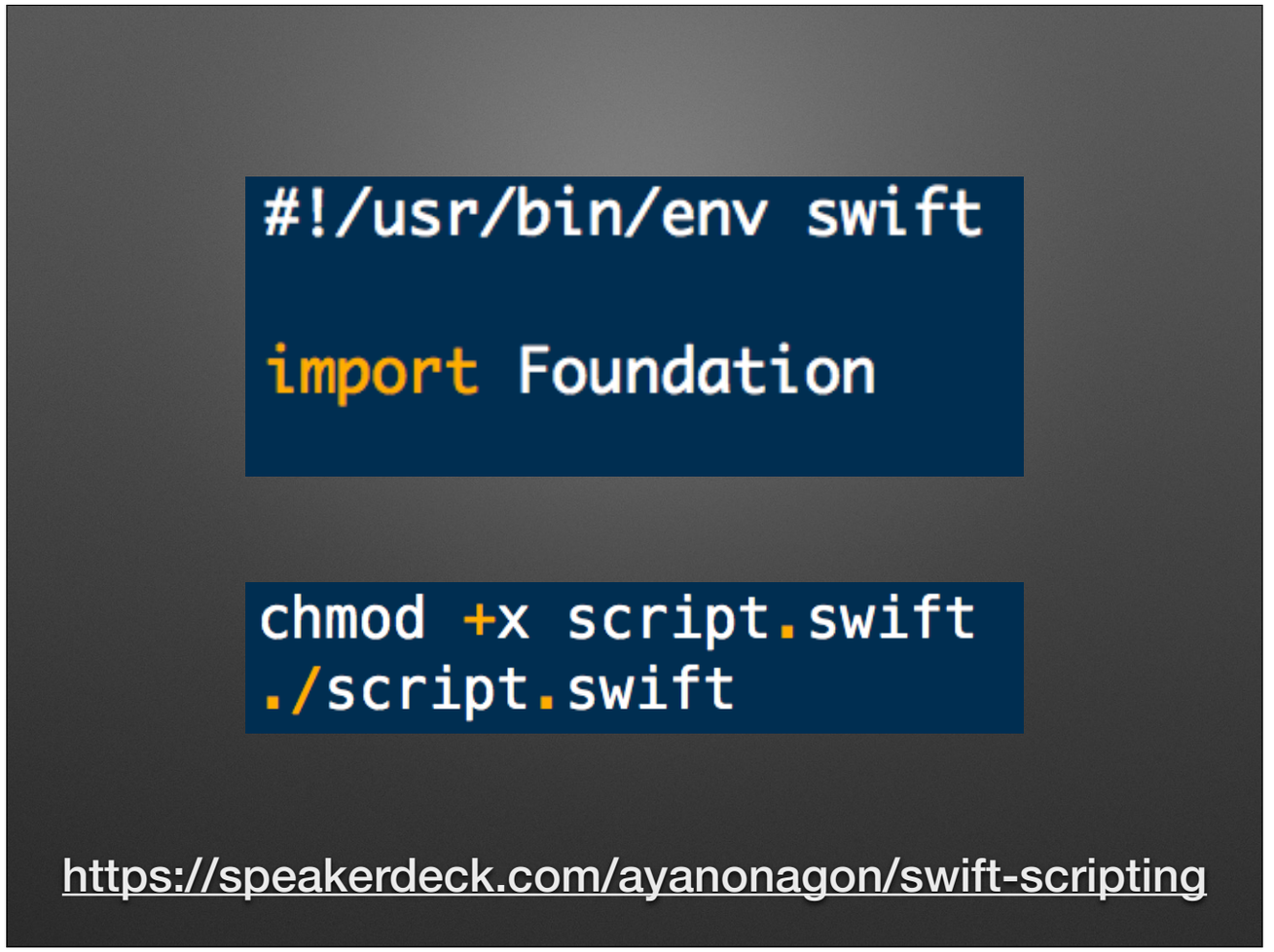
Just declare a var of the type you would like you bytes cast to then pass in your pointer using the & operator.

```swift
/// Grabs the next two bytes from data. Interprets these as a UInt16 Sequence number
///
/// Note: Caller should call readAndIgnoreBytes(data, 2) after this call.
///
/// :param: data NSData to read the first 2 bytes from
/// :returns: A sequence number
func getSequenceNumber(data: NSData) -> UInt16 {
    var seq: UInt16 = 0
    data.getBytes(&seq, length: 2)
    return seq.byteSwapped
}
```

Getting some UInt16 bytes. boom.

Scripting

Scripting just like bash, ruby or python.



Just put your #! line at the top.

You can also link to another framework path using

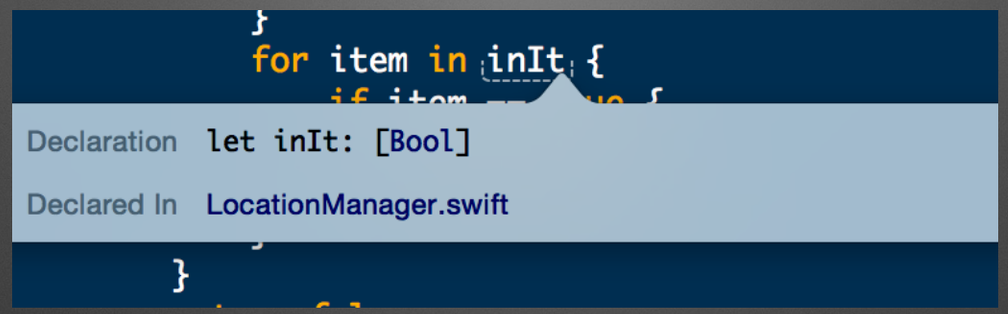
#!/usr/bin/env swift -F carthage/builds/iOS

With carthage build frameworks for example.

Advanced Stuff

Type Inference

ABC: Always Be option-Clicking



Use the Option-click pop up

Since Swift is Strongly Typed you must know the types

Option clicking is the BEST.

Where Do T an U come from

```
public func restoreCollectionFromPath<T>(path: String) -> Array<T>?

public func restoreCollectionFromPath<T>(path: String) -> Set<T>?

public func restoreCollectionFromPath<T,U>(path: String) -> Dictionary<T,U>?
```

Maybe Here?

```
public func restoreFromArchiveArray<T>( array: NSMutableArray) -> Array<T>

public func restoreFromArchiveArray<T>(array: NSMutableArray) -> Set<T>

public func restoreFromArchiveArray<T,U>(array: NSMutableArray) -> Dictionary<T,U>
```

These T get filled in from Return type type inference. It is a little crazy to write a library using this but it works great.

Curried Functions

```swift
func add(a: Int, b:Int) -> Int {
    return a + b
}
```

```swift
func add(a: Int, b:Int) -> Int {
    return a + b
}
```

```swift
func addTwo(a: Int) -> Int {
    return add(a, 2)
}
```

What about addThree(), or addOneHundred()

```swift
func add(a: Int) -> (Int -> Int) {
    return { b in a + b }
}



func addTwo(a: Int) -> Int {
    return add(a, 2)
}
```

Return a closure? this works, gives us the flexibility we want.

```swift
func add(a: Int) -> (Int -> Int) {
    return { b in a + b }
}
```

```swift
let addTwo = add(2)
```

Great now we just call it with one parameter and boom. addTwo is now a function that we can use.

```
func add(a: Int)(b: Int) -> Int {
    return a + b
}


let addTwo = add(2)
```

We can rewrite the add function to not use the hard to parse closure returning syntax.

“Instance Methods are really just curried class functions”

–Ole Begemann

```swift
class BankAccount {
    var balance: Double = 0.0

    func deposit(amount: Double) {
        balance += amount
    }
}
```

```
let account = BankAccount()
account.deposit(100) // balance is now 100
```

```swift
class BankAccount {
    var balance: Double = 0.0

    func deposit(amount: Double) {
        balance += amount
    }
}
```

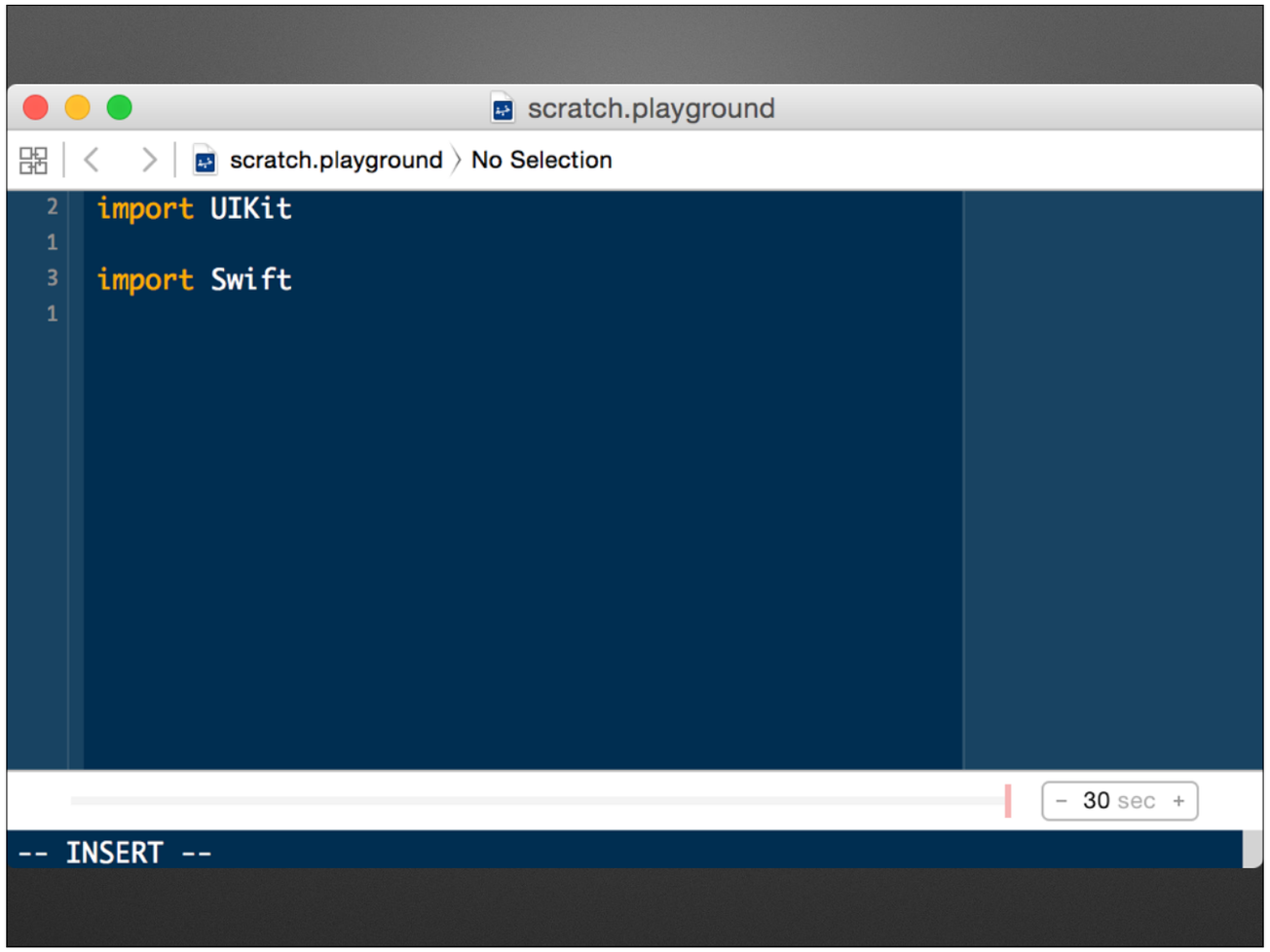
```swift
let depositor = BankAccount.deposit
```

```swift
depositor(account)(100) // balance is now 200
```

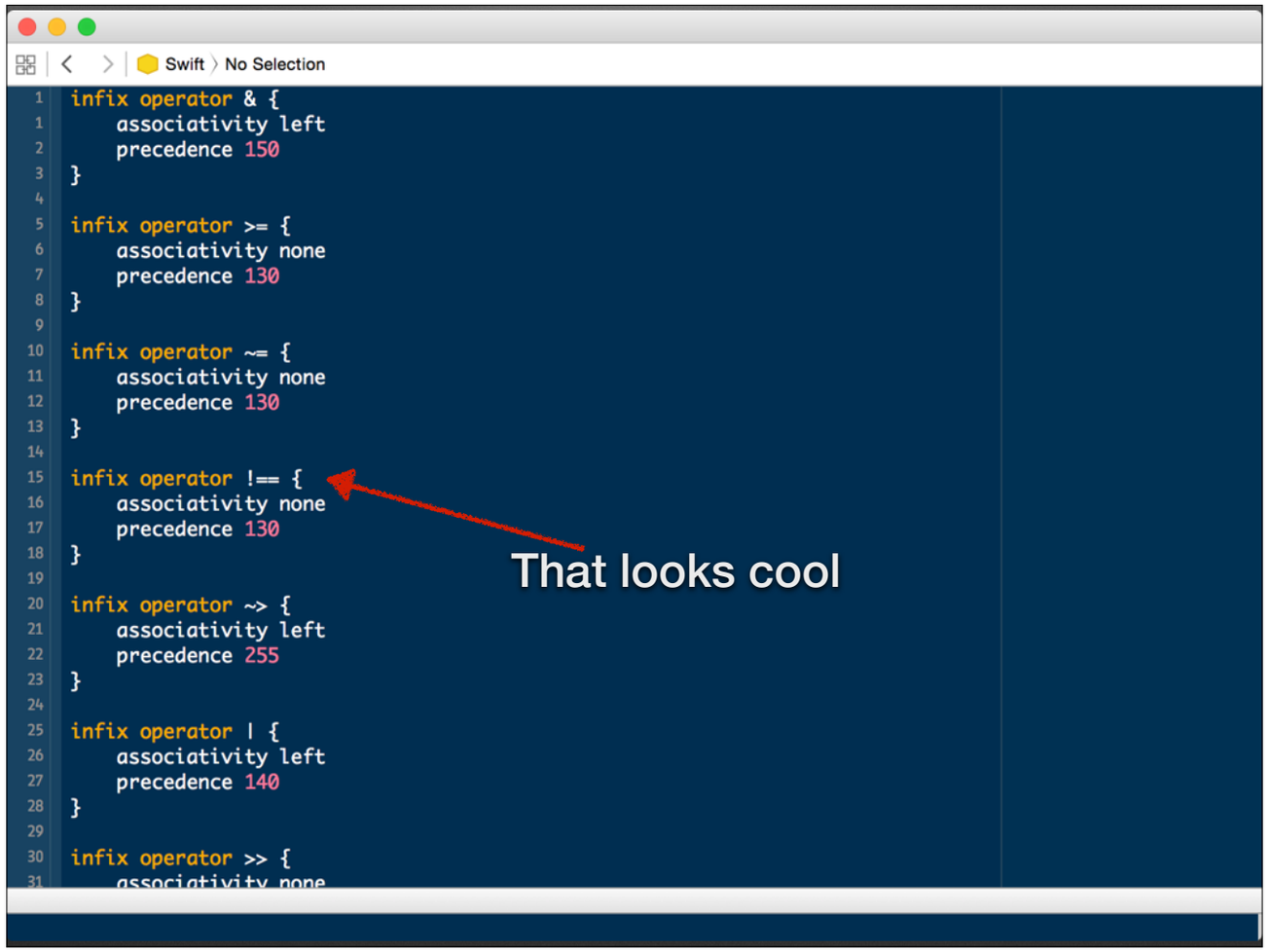
Often the Swift error messages might be given in this form. Or when you are looking through the assembly if debugging. You might see.

```
var s = "Hello"
String.removeAtIndex(&s)(s.startIndex)
```

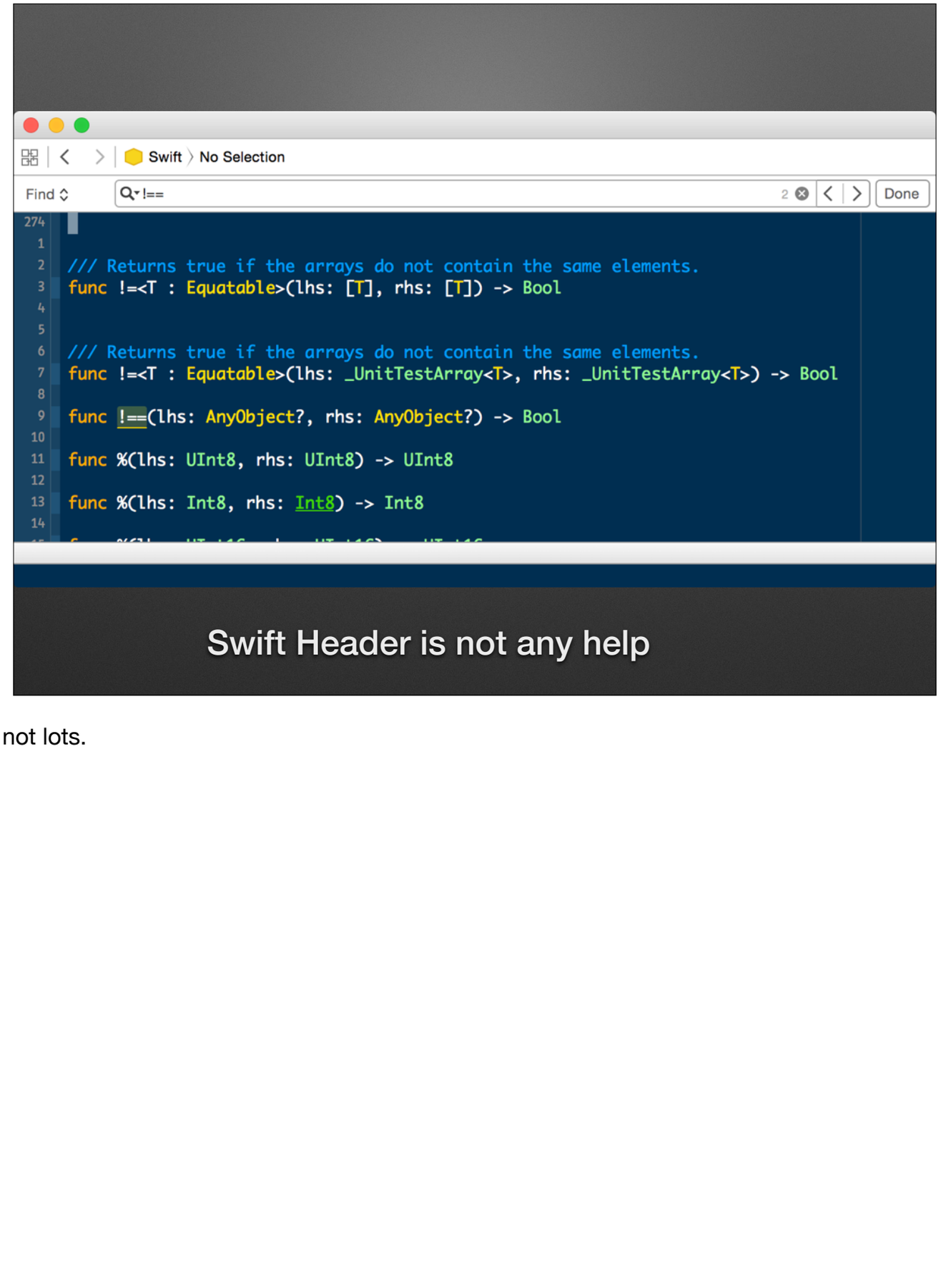
importing Swift



Command Click on Swift to see the declarations of the Std Library.

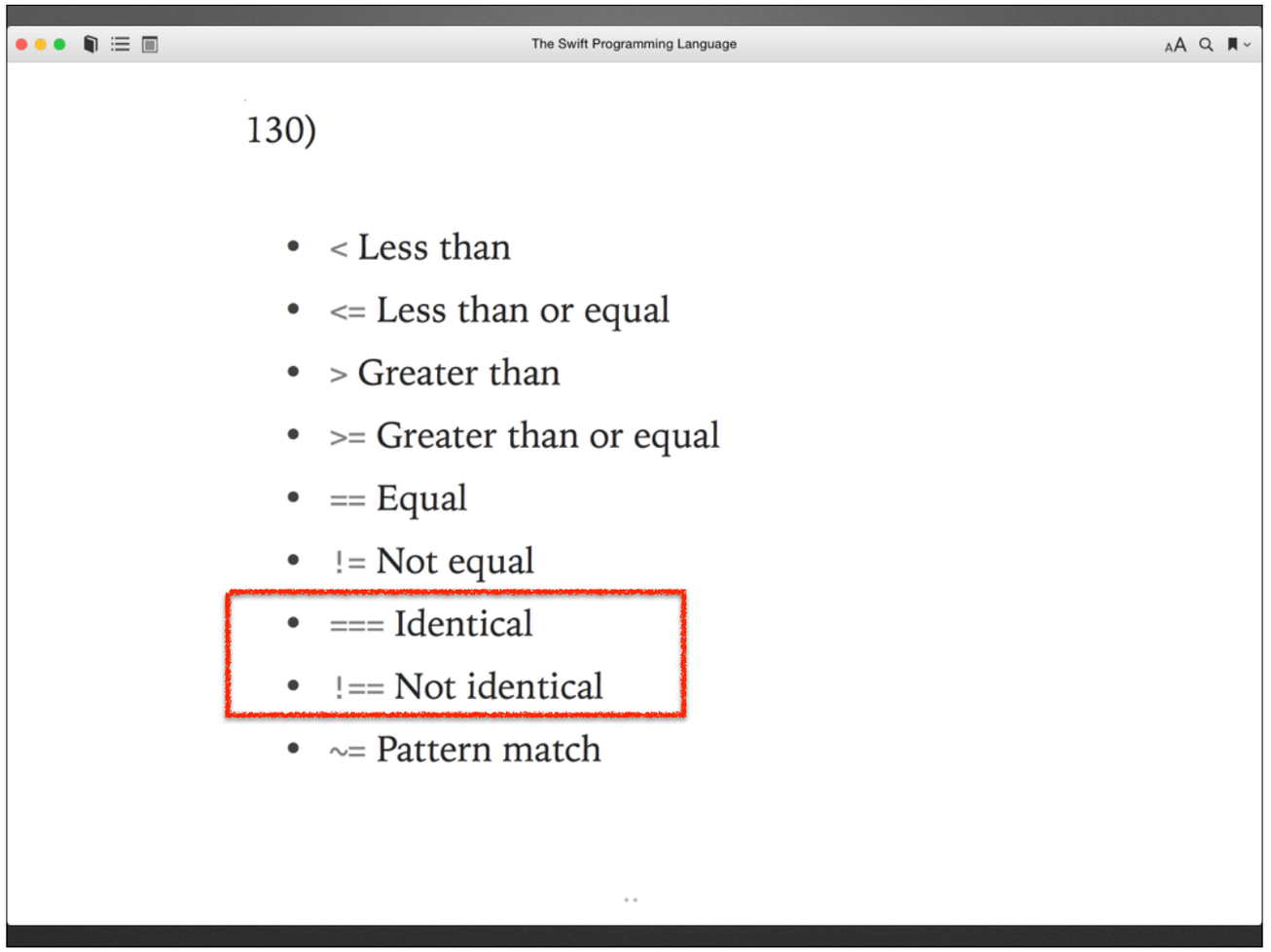


Lets see what that operator does.



Swift Header is not any help

the function signature is some help but not lots.



Couple this with our information about Equatable Protocol, And the fact that this function takes AnyObjects we can make a pretty good guess that this is for checking object equality.

Thanks

@john_regner

Follow me on twitter!