Get Swifty

Apple's Objective-C Replacement

Agenda

What we will cover

- How Apple arrived at Swift
- What’s new in Swift
- Swift’s syntax & basics

Apple’s road to Swift

Objective-C is created

Created in 1983 by Brad Cox & Tom Love

Built right on top of C

Added Object Oriented ideas (classes, objects, etc)

Apple’s road to Swift

Objective-C is engrained into Apple’s technology stack

2 years later, Jobs licenses it for NeXT (NeXTSTEP)

1996, Jobs is back at Apple and NeXT is acquired

Object Oriented libraries separated out from the OS as an API called OpenStep

1997, Apples includes NeXT OS and APIs into new OS - “Rhapsody”. This became Mac OS X.

This is where we get “NS” from - NeXT Step

Apple’s road to Swift

Objective-C peaks

Objective-C became the native language for iOS/OS X

Improved and iterated upon over the years

Works fine - but starts to show its age

Apple’s road to Swift

Objective-C’s weaknesses

- Objective-C is not easy to learn
- Syntax is unusual and unfamiliar
- C is mixed in and heavily used in Foundation
- C baggage (.h & .m files, for example)
- No generics (leads to tedious casts)
- Overloading not supported

Apple’s road to Swift

In comes Swift!

Keeps the best of Objective-C, such as named parameters

Brings in modern programming language advancements

Syntax is familiar - similar to C#, Python, and Rust

What’s new in Swift

Cool new features

Type inference

keyword name type initial value

```
let aString: String = "String variable"
```

What’s new in Swift

Type Inference

Omit the type - it’s inferred to be a String

```
let anotherString = "Another string variable"
```

Variables can be unicode characters

```
let 🐱 = "Catface"
```

What’s new in Swift

Types

String is lightweight like a C string

Powerful as an NSString

Concatenating is no longer a chore

```
NSURL *twitterURL = [[NSURL alloc] initWithString:
 [twitterURL stringByAppendingString:twitterHandle]];

let twitterURL = NSURL(string: twitterURL +
                               twitterHandle)
```

What's new in Swift

String Interpolation

Can even have expressions evaluated

```
let speakers = 20
let attendees = 300

let audience = "On average, there will be
\(attendees / speakers) people at each session."
```

No need for a mutable or immutable typed Strings

What's new in Swift

Constants

The `let` keyword defines a constant

```
let speakers = 20
let attendees = 300
```

```
let audience = "On average, there will be
\(attendees / speakers) people at each session."
```

- Opt for immutability
- Forces you to think about your declarations
- Safer for multithreaded environments
- Optimization gains

What’s new in Swift

Variable Initialization

- Variables always initialized before use

- But what about nil?

- Objective-C nil = Pointer to a non existent object

- Swift nil = Absence of a value of a certain type

- Optionals - more on that later

What’s new in Swift

Other notable additions

- Closures unified with function pointers
- Generics
- Tuples
- No more semicolons (though you can)
- and more

If Statements

Some basic and minor changes

No parentheses required

Braces always required

```
if 1 < 2 {
    print("True")
} else {
    print("False")
}
```

Collections

Array and Dictionary

They can work with any type - primitives included

Concise and powerful:

```
let nums = [1]
```

Specify the type if you want:

```
let names: [String] = ["Harlan", "Hayden"]
```

Collections

Literal declarations

Array

```
let names = ["Harlan", "Hayden", “Bryan”, “David”]
```

Dictionary

```
let namesAndAges = [“Harlan": 20, “Hayden": 26,
"Bryan": 30, “David”: 33]
```

No more “@“ in front of strings either

Collections

Easy to modify

Specify index

```
var modify = ["Harlan"]
modify[0] = "Hayden"
//["Hayden"]
```

Modify a collection with append<T>(element: T):

```
var modify = [“Harlan”]
modify.append(“Hayden”)
//["Harlan","Hayden"]
```

Use range operators

```
var modify = ["Harlan","Hayden"]
modify[0...1] = ["Bryan","David"]
//[“Bryan”,”David"]
```

Collections

Modify a dictionary

Just define a new key

```
var family = [“Harlan": 20]
family["Hayden"] = 26
```

Editing value works the same way

```
var family = [“Harlan": 19]
family["Harlan"] = 20
```

Collections

Varied types

If you want a collection with more than one type :

```
var multiTyped: [AnyObject] = ["foo", 01, true, 44.5]
```

That said, try to keep them strongly typed

For most intents and purposes, `AnyObject` is analogous to `id`

Loops

Ranges

Half open range

```
for i in 0..<2 {
    print(i)
}
//Output:
0
1
```

Closed range

```
for i in 0...2 {
    print(i)
}
//Output:
0
1
2
```

Loops

Flexible

Easily loop through characters in a string

```
let abc = "abc"

for char in abc.characters {
    print(char)
}
```

In Objective-C :

```
NSString *myStrings = @"abc";

for (NSInteger charIdx=0; charIdx < myStrings.length; charIdx++) {
    NSLog(@"%C", [myStrings characterAtIndex:charIdx]);
}
```

Loops

Loops cont.

Exclude value from the range (use _)

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
```

Iterating over collections

```
let family = [“Harlan": 20,”Hayden": 26,”Bryan": 30, ”Bryan": 30]

//KVPs from dictionary come back as tuples
for (name, age) in family {
    print("\(name) is \(age) years old.")
}
```

Loops

Traditional C-style iteration

Condition - increment looping

```
for var idx = 0; idx < MAX; idx++ {
    print("Index is \(idx)")
}
```

No parentheses

Initialization with `var` and not `let`

While loops are here too

Switch Statements

Fallthrough

No implicit fall through

You can still “break” out before execution is finished

If you want to fall through, you can use `fallthrough`

Switch Statements

Switches cont.

You can do Haskell-esque pattern-matching with them

```
let anInt = 40
switch anInt {
    case 0, 1, 2:
        print("Tiny")
    case 3...5:
        print("Medium")
    case 6..<39:
        print("Large")
    case _ where anInt % 2 == 1:
        print("It's odd")
    case _ where anInt % 2 == 0:
        print("Nope, it's not odd, it's even")
    default:
        break
}
```

Switch Statements

Compared to Objective-C

The old days

```
NSString *familyMember = @"Harlan";

if ([familyMember isEqualToString:@“Harlan"]) {
    NSLog(@"It's me!");
} else if([familyMember isEqualToString:@“Hayden"]) {
    NSLog(@“It's brother #1!”);
} else if([familyMember isEqualToString:@“Bryan"]) {
    NSLog(@"It's brother #2!”);
} else if([familyMember isEqualToString:@“David"]) {
    NSLog(@"It's brother #3!”);
} else {
    NSLog(@"We don't know who it is.");
}
```

The new days

```
let familyMember = "Harlan"

switch familyMember {
case "Harlan":
    print("It's me!")
case "Hayden":
    print(“It's brother #1!”)
case "Bryan":
    print("It's brother #2!”)
case "David":
    print("It's brother #3!”)
default:
    print("We don't know who it is.")
}
```

Objective-C doesn’t support Switch with NSString

Optionals

A core concept of Swift

We want the value - or to know it wasn’t found

Means we get a value back - or nothing at all

Optionals have ? by them

Optionals

Cont.

We could use magic numbers (i.e. -1)

NSNotFound if in Objective-C

Returns nil -or no value, or an int (Need to specify type)

```
let harlansAge: Int? = family["Harlan"]
```

Optionals

Unwrapping

So if it’s there, how do we get it?

```
let harlansAge: Int? = family["Harlan"]
if harlansAge == nil {
    print("Harlan is apparently timeless.")
} else {
    let foundAge = harlansAge!
    print("Harlan is \(foundAge) years old.")
}
```

Unwrap the value (i.e. the !)

No need to specify the type, the compiler knows

Optionals

Short syntax

This is a common pattern, shorthand is like so (no !):

```
if let foundAge = harlansAge {
    print("Harlan is \(foundAge) years old.")
}
```

If you know the value is there, you can unwrap it directly:

```
var name: String? = "Harlan"

let anotherHarlan = name!

print(anotherHarlan)
```

You’re crashing if you’re wrong

If forced unwrapped, you don’t need to set it to a var.

Optional Chaining

Query multiple optionals

What if you want a value that could be housed around other nil values?

```
class Residence {
    var street: String?
}

class Person {
    var home: Residence?
}


var aPerson = Person()
```

Optional Chaining

Cont.

Use ? operator to use optional chaining

```
class Residence {
    var street: String?
}

class Person {
    var home: Residence?
}


var aPerson = Person()

if let theStreet = aPerson.home?.street {
    print("The street is \(theStreet)")
} else {
    print("Person has no street")
}
```

Remember, any optional must be unwrapped via !

Functions

Overview

Defined with `func` keyword

```
func printName() {
    print("It's Harlan")
}
```

Named parameters, like Objective-C

```
func printName(name:String) {
    print("It's \(name)”)
}
```

Functions

Return types

Denote return type with ->

```
func printGreeting(name name: String) -> String {
    return "It's \(name), how ya doin' today?"
}
```

Define default values

```
func printGreeting(name name: String = "Hayden") -> String {
    return "It's \(name), how ya doin' today?"
}
```

Functions

Multiple return types

Return tuples

```
func nameAndAge() -> (String, Int) {
    return (“Harlan", 20)
}
```

Decompose them to access values

```
let (name, age) = nameAndAge()

print("\(name) is \(age) years old.")
```

Functions

Name multiple return values

Give meaningful names to return values

```
func nameAndAge() -> (name: String, age: Int) {
    return (“Harlan", 20)
}

let harlan = nameAndAge()

print("\(harlan.name) is \(harlan.age) years old.")

//Harlan is 20 years old.
```

Closures

Similar functionality

Much like blocks in Objective-C

Contain some code, you can pass them around

Lambdas or anonymous functions

```
let aClosure = {
    print("This is a closure")
}
```

Compiler sees it like this:

```
let aClosure: () -> () = {
    print("This is a closure")
}
```

Closures

Syntax

Notice that’s similar to a function’s signature

```
let aClosure: () -> () = {
    print("This is a closure")
}
```

Functions are just named closures

```
func aClosure: () -> () = {
    print("This is a closure")
}
```

Closures

Passed as a parameter

Define in the formal parameter list

```
func doTaskRepeated(count: Int, theTask: () -> ()) {
    for i in 0..<count {
        theTask()
    }
}
```

Calling it

```
doTaskRepeated(10, {
    print("A complex and awesome task.")
})
```

Closures

Trailing closure

Define closure as the last parameter in formal parameter list

Looks like a control flow statement

```
doTaskRepeated(10) {
    print("A complex and awesome task.")
}
```

Classes

Much like Java and .NET

No more import because Swift has no `.h/.m` files

No need to explicitly define a base class

```
class Person {

}
```

Classes

Properties

Swift provides the backing store for you

```
class Harlan {
    let name = "Harlan"
}
```

By default, all entities have `internal` access

```
class Harlan {
    let name = "Harlan"
    private let showMostWatchedPastMonth = "Seinfeld"
}
```

Classes

Exposing properties

Use internal, or nothing at all

```
class Harlan {
    let name = “Harlan”
    internal var age = 30
    private let showMostWatchedPastMonth = "SpongeBob"
}

let aHarlan = Harlan()

//Grew up quick during this talk
aHarlan.age = 35
```

Notice you don’t need “new” in front of the type

Classes

Computed properties

You can define custom getters and setters

```
class Harlan {
    let name = "Harlan"

    var location:(x: Float, y: Float) {
        get {
            return (43.08, -77.67)
        }
        set {
            self.location.x = newValue.x
            self.location.y = newValue.y
        }
    }
}
```

...can even use tuples

Create a read only computed property - just omit setter

Classes

Initialization

init() keyword — if you are inheriting, call super.init()

```
class Harlan {
    let name = "Harlan"
    var age = 20

    init() {
        //No need to return self
    }
}
```

Can also initialize constant values

```
class Harlan {
    let name = "Harlan"
    let hobby = ""

    init() {
        //No need to return self
    }

    init(hobby: String) {
        self.hobby = hobby
    }
}

var aHarlan = Harlan(hobby: "Programming")
```

Classes

Property Observers

Fires just before and right after value changes

```
class Kettle {
    private let fireDate = 9 //p.m.

    var date: Int {
        willSet {
            if date == fireDate {
                print(“DING DING DING")
            }
        }
        didSet {
            if date == fireDate {
                self.fireDate = 0
            }
        }
    }

    init() {
        self.date = fireDate
    }
}
```

Classes

Methods

Work the same way as functions

Only need to use self when property has the same name as a parameter in the function’s signature

```
class Hayden {
    var nickName = "Fabio"

    func changeNickName(nickName: String) {
        self.nickName = nickName
        print(“Hayden’s new nickname is \(self.nickName)")
    }
}
```

Classes

A note on initializers

You don’t even need to specify one

super.init needs to happen last in custom initializers

Sole purpose is to initialize values for the class

Structs

Not much has changed

Still works the same way

- Doesn’t support inheritance
- Value types

Think of them as you do in your OOP language of choice

Enumerations

Enums

Value types

They can have raw values (like in C)

```
enum NFLTeams: Int {
    case StLouisRams = 1, Patriots, Bucs, Chiefs
}

NFLTeams.StLouisRams.rawValue
// 1
```

Enumerations

Don’t always need underlying values

```
enum Directions {
    case North, South, East, West
}
//Compiler infers Directions as type
let directionToGo = Directions.North
```

Also, compiler will again infer the type

```
let lbl = UILabel()
lbl.textAlignment = .Right
```

Value type constants have all constant members
Reference type constants can have mutable members

Enumerations

Associated Values

Associate values within an enum

```
enum RamsVictory {
    case ByOnePoint
    case ByPoints(Int)
}

let ramsWinBig = RamsVictory.ByPoints(24)
```

Even custom properties

```
enum RamsVictory {
    case ByOnePoint, ByPoints(Int)
    var winSummary: String{
        switch self {
        case .ByOnePoint:
            return "Rams by one."
        case .ByPoints(let points):
            return "Rams win big by \(points)!"

        }
    }
}

var ramsWinBig = RamsVictory.ByOnePoint
print(ramsWinBig.winSummary) //Rams by one.
ramsWinBig = RamsVictory.ByPoints(14)
print(ramsWinBig.winSummary) //Rams win big by 14!
```

Access Modifiers

Control Access

Three modifiers

- Private - Available only from within source file
- Internal - Available to entire module that includes the definition (i.e. app or framework)
- Public - Intended for use with APIs, means entity can be accessed by any file that imports the module
- Final - Cannot be overridden by a subclass. Compiler can optimize dispatch for final methods.

There’s much more

Lots of new features

Interoperability with Objective-C

Extensions

Automatic Reference Counting

Pattern Matching

Functional Programming