Swift

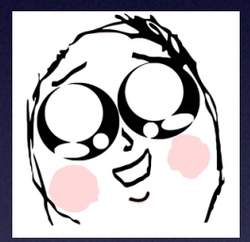
first impressions

Wojciech Jachowicz

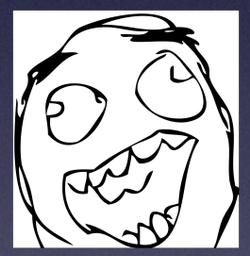
WWDC 2014 - Keynote

- HomeKit

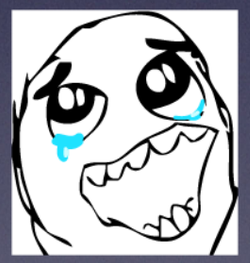
- TouchID



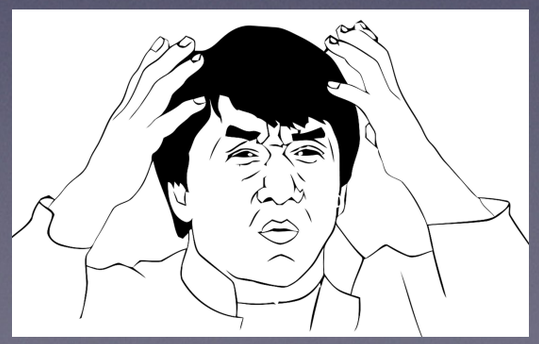
- CloudKit



- App Extensions



- Swift



Swift

Agenda

- variables, constants, optionals
- functions & closures
- tuples
- switch
- classes
- initializers
- structures
- memory management
- enums
- generics

Variables, constants

- Variable definition

```
var languageName: String
```

- No need to provide the type when assigning a value

```
var languageName = "Swift"
```

- Constant definition

```
let languageName: String = "Swift"
```

- Cannot be used when not initialized
- Cannot be nil

Optionals

- Represents possibly missing value

- By default initialized to nil

```
let animals = ["Cat": 4, "Snake": 0]
let possiblyLegCounts: Int? = animals["Dumbo Octopus"];
```

Unwrapping optionals

- We always have to check if value excists before assigning
- Forced unwrapping

```
let legCount: Int = possiblyLegCounts!
```

- Optional binding

```
if let legCount: Int = possiblyLegCounts {
}
```

- Optional chaining

```
if let homeNumber = czesiek.address?.buildingNumber?.toInt() {
}
```

Functions

- Functions are ARC objects
- Can be nested inside another function
- Can return another function

```
func uselessFunc(argument: String) -> String {
    return "Useless \(argument)";
}
```

Functions - example

```
func chooseFunction(isPositive: Bool) -> (Int) -> Int {
    func increase(input: Int) -> Int { return input + 1 }
    func decrease(input: Int) -> Int { return input - 1 }
    return isPositive ? decrease : increase
}
var currentValue = -4
let moveNearerToZero = chooseFunction(currentValue > 0)

while currentValue != 0 {
    println("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
```

Tuples

- Group of any values
- Can be used in functions to return multiple values

```
func networkStatus() -> (code: Int, description: String) {
    return (404, "Not found")
}

let status = networkStatus()
println("code: \(status.code), description: \
(status.description)”)

let (code, description) = networkStatus()
println("code: \(code), description: \(description)")
```

Closures

- Similar to blocks in Objective-C
- Functions are named closures or closures are anonymous functions
- Can be used as a function parameter
- Closures are ARC objects

```
var clients = ["Kowalski", "Nowak"]

clients.sort({(a: String, b: String) -> Bool in
    return a < b
})
```

Closures

- Type interfaces

```
clients.sort({ a, b in
    return a < b
})
```

- Implicit return

```
clients.sort({ a, b in a < b })
```

- Implicit arguments

```
clients.sort({ $0 < $1 })
```

- Trailing closures

```
clients.sort { $0 < $1 }
```

Closures

- Capture local state

```
func sum(numbers: Int[])  -> Int {
    var sum = 0

    numbers.map {
        sum += $0
    }

    return sum
}
```

Closures

And the biggest advantage of closures…

…you don’t need to use:

http://fuckingblocksyntax.com/

Switch

- **switch** statements do not fallthrough the bottom of each case into the next one. To reach another case you have to use **fallthrough** keyword.

- Every possible value of type being considered must be matched by one of the switch cases.

- Multiple values in single case.

- Can use objects as a case values.

Switch

- Range matching

```
var naturalCount: String
let count = 3_000_000
switch count {
case 0:
    naturalCount = "no"
case 1...3:
    naturalCount = "a few"
case 4..10:
    naturalCount = "several"
case 10...99:
    naturalCount = "tens of"
default:
    naturalCount = "hell lot of"
}
```

Switch

- pattern matching

```
let color = (1.0, 1.0, 1.0, 1.0)

switch color {
    case (0.0, 0.5...1.0, let blue, _):
        println("Green and \(blue * 100)% blue")

    case let (r, g, b, 1.0) where r == g && g == b:
        println("Opaque grey \(r * 100)%”)

    default:
        println("another color")
}
```

Classes

- No headers with declarations
- No universal base class (like NSObject)
- No disctinct between iVar and properties.
- All properties are public.

Classes

- Computed properties

```
class Vehicle {
    var numberOfWheels = 0

    var description : String  {
        get {
            return "\(numberOfWheels) wheels"
        }
    }
}
```

Initializers

- All properties must be set (except optionals)
- All values have to be set before calling method on self in initializer.
- **super.init** should be called after assigning all properties.
- initializers do not return self as in Objective-C
- lazy properties with **@lazy** keyword

Initializers - example

```
class Car {
    var topSpeed:Int
    var fuel: Int = 0
    @lazy var engine = Engine ()
    init(topSpeed: Int) {
        self.topSpeed = topSpeed
        fuelUp()
    }

    func fuelUp () -> Void {
        fuel = 100
    }
}
```

```
class SportCar: Car {
    var hasTurbo: Bool

    init (hasTurbo: Bool, topSpeed: Int)
{
        self.hasTurbo = hasTurbo
        super.init(topSpeed: topSpeed)
    }

    convenience init () {
        self.init(hasTurbo: true,
topSpeed: 200);
    }
}
```

Structures in Objective-C/C

- Used to package related data together.

```
struct Foo
{
  int x;
  int array[100];
};
```

Structures in Swift

```
struct Rect {
    var width, height : Float
    var area : Float {
        return width * height
    }
    func isBiggerThanRect(other: Rect) -> Bool {
        return self.area > other.area
    }
    mutating func multiplySize(factor: Float) -> Void {
        height *= factor
        width *= factor
    }
}

var frame = Rect(width: 200, height: 300)
var copyOfFrame = frame
println("area: \(frame.area)")
frame.multiplySize(3.0)
if frame.isBiggerThanRect(copyOfFrame) {
    println("Bigger")
}
```

Structures

- have default initializers
- may have computed properties
- can have its own methods
- if method wants to change values of struct the method must be marked as **mutated**

Class vs Struct

- Structures cannot inherit from another structures
- Classes are passed by reference while structures are passed by value (copy)
- When we have constant object, its values can be changed but we cannot assign new object. When we have constant struct, the whole struct is immutable.

Memory management

- ARC
- default reference is **strong**
- **weak** reference only for optionals
- **unowned** reference is similar to **unsafe_unretained**

```
class Person {
    var car: Car?
    var creditCard: CreditCard?
}

class Car {
    weak var owner: Person?
}

class CreditCard {
    unowned var owner: Person
}
```

Enums

- Values can be strings, characters, or any of the integer or floating-point number types
- Associated values
- Initializers
- Computed properties

Enums - example

```
enum MPKStatus {
    case OnTime
    case Delayed(Int)

    init() {
        self = Deleyed(10)
    }

    var description: String {
        switch self {
            case OnTime:
                return "on time"
            case Delayed(let minutes):
                return "deleyed by \(minutes) minutes"
        }
    }
}

var busStatus = MPKStatus()
busStatus = .Delayed(20)
println(busStatus.description)
```

Generics

- Let you write flexible, reusable functions and types that can work with any type

```
struct Stack<T> {
    var elements = T[]()
    mutating func push(element: T) {
        elements.append(element)
    }

    mutating func pop() -> T {
        return elements.removeLast()
    }
}

var intStack = Stack<Int>()

intStack.push(50)
```

Thank you