Good Morning

SWIFT

I'm Marc Prud'hommeaux

marc@glimpse.io

Swift

Public beta: June 2014

v1.0: September 2014

v1.2: February 2015

POLL: Swift?

Agenda

1. Swift Syntax
2. Interesting Stuff
3. Questions (GOTO Guide)

```swift
// Swift Bank & Account
class Bank {
    var accounts: Array<Account>

    init() {
        self.accounts = Array<Account>()
    }

    func addAccount(account: Account) {
        accounts.append(account)
    }

    func holdings() -> Int {
        var amount: Int = 0
        for (account: Account) in accounts {
            amount += account.pennies
        }
        return amount
    }
}

class Account {
    var owner: String?
    var pennies: Int

    init(owner: String, pennies: Int) {
        self.owner = owner
        self.pennies = pennies
    }

    func deposit(amount: Int) {
        self.pennies += amount
    }

    func withdraw(amount: Int) {
        self.pennies -= amount
    }
}
```

```objc
// Objective-C Bank & Account
@interface Bank : NSObject
@property NSMutableArray *accounts;
@end

@interface Account : NSObject
@property NSString *owner;
@property int pennies;
@end

@implementation Bank
- (id)init {
    if (self = [super init]) {
        self.accounts = [[NSMutableArray alloc] init];
    }
    return self;
}

- (void)addAccount:(Account *)account {
    [self.accounts addObject:account];
}

- (int)holdings {
    int amount = 0;
    for (Account *account in self.accounts) {
        amount += account.pennies;
    }
    return amount;
}
@end


@implementation Account
- (id)initWithOwner:(NSString *)owner pennies:(int)pennies {
    if (self = [super init]) {
        self.owner = owner;
        self.pennies = pennies;
    }
    return self;
}

- (void)deposit:(int)amount {
    self.pennies += amount;
}

- (void)withdraw:(int)amount {
    self.pennies -= amount;
}
@end
```

```java
// Java Bank & Account
class Bank {
    List<Account> accounts;

    Bank() {
        this.accounts = new ArrayList<Account>();
    }

    void addAccount(Account account) {
        accounts.add(account);
    }

    int holdings() {
        int amount = 0;
        for (Account account : accounts) {
            amount += account.pennies;
        }
        return amount;
    }
}

class Account {
    String owner;
    int pennies;

    Account(String owner, int pennies) {
        this.owner = owner;
        this.pennies = pennies;
    }

    void deposit(int amount) {
        this.pennies += amount;
    }

    void withdraw(int amount) {
        this.pennies -= amount;
    }
}
```

```swift
// Swift Bank & Account
class Bank {
    var accounts: Array<Account>

    init() {
        self.accounts = Array<Account>()
    }

    func addAccount(account: Account) {
        accounts.append(account)
    }

    func holdings() -> Int {
        var amount: Int = 0
        for (account: Account) in accounts {
            amount += account.pennies
        }
        return amount
    }
}

class Account {
    var owner: String?
    var pennies: Int

    init(owner: String, pennies: Int) {
        self.owner = owner
        self.pennies = pennies
    }

    func deposit(amount: Int) {
        self.pennies += amount
    }

    func withdraw(amount: Int) {
        self.pennies -= amount
    }
}
```

```java
// Java Bank & Account
class Bank {
    List<Account> accounts;

    Bank() {
        this.accounts = new ArrayList<Account>();
    }

    void addAccount(Account account) {
        accounts.add(account);
    }

    int holdings() {
        int amount = 0;
        for (Account account : accounts) {
            amount += account.pennies;
        }
        return amount;
    }
}

class Account {
    String owner;
    int pennies;

    Account(String owner, int pennies) {
        this.owner = owner;
        this.pennies = pennies;
    }

    void deposit(int amount) {
        this.pennies += amount;
    }

    void withdraw(int amount) {
        this.pennies -= amount;
    }
}
```

```swift
// Swift Bank & Account
class Bank {
    var accounts: Array<Account>

    init() {
        self.accounts = Array<Account>()
    }

    func addAccount(account: Account) {
        accounts.append(account)
    }

    func holdings() -> Int {
        var amount: Int = 0
        for (account: Account) in accounts {
            amount += account.pennies
        }
        return amount
    }
}

class Account {
    var owner: String?
    var pennies: Int

    init(owner: String, pennies: Int) {
        self.owner = owner
        self.pennies = pennies
    }

    func deposit(amount: Int) {
        self.pennies += amount
    }

    func withdraw(amount: Int) {
        self.pennies -= amount
    }
}
```

```java
// Java Bank & Account
class Bank {
    List<Account> accounts;

    Bank() {
        this.accounts = new ArrayList<Account>();
    }

    void addAccount(Account account) {
        accounts.add(account);
    }

    int holdings() {
        int amount = 0;
        for (Account account : accounts) {
            amount += account.pennies;
        }
        return amount;
    }
}

class Account {
    String owner;
    int pennies;

    Account(String owner, int pennies) {
        this.owner = owner;
        this.pennies = pennies;
    }

    void deposit(int amount) {
        this.pennies += amount;
    }

    void withdraw(int amount) {
        this.pennies -= amount;
    }
}
```

```swift
// Swift Bank & Account
class Bank {
    var accounts: Array<Account>

    init() {
        self.accounts = Array<Account>()
    }

    func addAccount(account: Account) {
        accounts.append(account)
    }

    func holdings() -> Int {
        var amount: Int = 0
        for (account: Account) in accounts {
            amount += account.pennies
        }
        return amount
    }
}

class Account {
    var owner: String?
    var pennies: Int

    init(owner: String, pennies: Int) {
        self.owner = owner
        self.pennies = pennies
    }

    func deposit(amount: Int) {
        self.pennies += amount
    }

    func withdraw(amount: Int) {
        self.pennies -= amount
    }
}
```

Generics

```
var strings: Array<String> = ["a", "b", "c"]
strings.append("d")
strings.append(1.2) // error!
```

Type Inference

```
var num : Int = 1
var strings : [String] = ["a", "b", "c"]
var dictionary : [String: Double] = ["a": 1.1, "b": 2.2]
```

Type Inference

```
var num = 1
var strings = ["a", "b", "c"]
var dictionary = ["a": 1.1, "b": 2.2]
```

optionals

```
var str: String? = "Hello"
str = "World"
str = nil // legal
```

optionals

```
var str: String = "Hello"
str = "World"
str = nil // illegal!
```

optional chaining

```
var pennies: Int? = account.owner?.deposit?.amount
if pennies != nil {
    // neither owner nor deposit was nil
}
```

tuples

```
var twoInts: (Int, Int) = (1, 2)
```

tuples

```
var twoStrings: (String, String) = ("dog", "cow")
```

tuples

```
var someStuff: (String, Double, Account) = ("Marc", 1.23, myAccount)
```

tuples

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

enums

enums

enums

enums

functions

functions

functions

functions

functions

functions

functions

functions

higher-order functions

function currying

functions

multi-paradigm

multi-paradigm

- Functional + Object-Oriented & Imperative

operator overloading

operators as functions

```
func isOrderedBefore(s1: String, s2: String)->Bool {
    return s1 < s2
}

var words = ["dog", "cow"]
words.sort(isOrderedBefore)
```

```
var words = ["dog", "cow"]
words.sort({ (s1: String, s2: String) in
    return s1 < s2
})
```

```
var words = ["dog", "cow"]

words.sort(<) // => "cow", "dog"
words.sort(>) // => "dog", "cow"
```

Immutable Structs

Other Features

- Access Control
- Bridging to Objective-C
- Automatic Reference Counting (no GC)
- Protocol (i.e., interfaces)
- Categories (i.e., type extensions)
- And Much, Much More!

```swift
var explicitDouble: Double = 70

var implicitInteger = 70
var implicitDouble = 70.0
var 國 = "美國"

let numberOfBananas: Int = 10

let numberOfApples = 3
let numberOfOranges = 5

let appleSummary = "I have \(numberOfApples) apples."
let fruitSummary = "I have \(numberOfApples + numberOfOranges) pieces of fruit."

var fruits = ["mango", "kiwi", "avocado"]

if fruits.isEmpty {
    println("No fruits in my array.")
} else {
    println("There are \(fruits.count) items in my array")
}
```

```swift
let people = ["Anna": 67, "Beto": 8, "Jack": 33, "Sam": 25]

for (name, age) in people {
    println("\(name) is \(age) years old.")
}

func sayHello(personName: String) -> String {
    let greeting = "Hello, " + personName + "!"
    return greeting
}

println(sayHello("Jane"))

func sayAge(#personName: String, personAge Age: Int) -> String {
    let result = "\(personName) is \(Age) years old."
    return result
}

// we can also specify the name of the parameter
println(sayAge(personName: "Jane", personAge: 42))
```

Immutable Values

Definitions

Definitions

- Mutability:
- Immutability:

Definitions

- Mutability: changeable
- Immutability: unchangable

All Languages have Some

```
final int i = 42;
i++; // illegal!
```

let

```
let i = 42
i++ // illegal!
```

Immutability

Value Types

Value Types

- Unshared

Value Types

- Value Types: Unshared
- Reference Types: Shared

Primitives are Values

```java
// java
int i = 42;
int j = i;
i++; // i = 43, j = 42
```

Immutability in Java

```
Animal a = new Animal("dog");
Animal b = a;

b.type; // "dog"
a.type = "cow";
b.type; // "cow"
```

Immutability in Java

```
final Animal a = new Animal("dog");
final Animal b = a;

b.type; // "dog"
a.type = "cow";
b.type; // "cow"
```

```
final Animal a = new Animal("dog");
final Animal b = a;

b.type; // "dog"
a.type = "cow";
b.type; // "cow"

b = new Animal("moose"); // illegal!
```

Reference vs. Value Types

```
class Animal {
    var type: String
}
```

```
let a = new Animal("dog")
let b = a

b.type // "dog"
a.type = "cow"
b.type // "cow"
```

Reference vs. Value Types

```
struct Animal {
    var type: String
}

var a = new Animal("dog")
var b = a

b.type // "dog"
a.type = "cow"
b.type // "dog" <--------
```

Reference vs. Value Types

Similarities

Reference vs. Value Types

Similarities

- contain properties

Reference vs. Value Types

Similarities

- contain properties
- contain methods

Reference vs. Value Types

Similarities

- contain properties
- contain methods
- contain initializers

Reference vs. Value Types
Similarities

- contain properties
- contain methods
- contain initializers
- conform to protocols

Reference vs. Value Types

Differences

- cannot inherit

Reference vs. Value Types

Differences

- cannot inherit
- no identity

Reference vs. Value Types

Differences

- cannot inherit
- no identity
- unshared
- immutable

```
struct Animal {
    var type: String
}

var a = new Animal("dog")
a.type = "cow"
```

```
struct Animal {
    var type: String
}

var a = new Animal("dog")
var b = a

b.type // "dog"
a.type = "cow"
b.type // "dog" <--------
```

```
struct Bank {
    var accounts: [Account] = []
    var holdings: Int { return accounts.map({ $0.pennies }).reduce(0, combine: +) }
}

struct Account {
    var owner: String
    var pennies: Int = 0
}

var bank = Bank()
var a1 = Account(owner: "Marc", pennies: 100)
bank.accounts += [a1]

var a2 = Account(owner: "Dave", pennies: 1_000)
bank.accounts += [a2]

bank.holdings // => 1,100

bank.accounts[0].pennies -= 50
bank.holdings // => 1,050
```

```
struct Bank {
    var accounts: [Account] = []
    var holdings: Int { return accounts.map({ $0.pennies }).reduce(0, combine: +) }
}

struct Account {
    var owner: String
    var pennies: Int = 0
}

var bank = Bank()
var a1 = Account(owner: "Marc", pennies: 100)
bank.accounts += [a1]

var a2 = Account(owner: "Dave", pennies: 1_000)
bank.accounts += [a2]

bank.holdings // => 1,100

bank.accounts[0].pennies -= 50
bank.holdings // => 1,050

a1.pennies -= 50
bank.holdings // => 1,050 // !?!
```

Deceptive!

But convenient

Strings in Java: pretend references

Strings in Swift: true references

Values all the way down

So What?

Benefits

Benefits

1. comprehensible

Benefits

1. comprehensible
2. testable

Benefits

1. comprehensible
2. testable
3. parallel

Benefits

1. comprehensible
2. testable
3. parallel
4. portable

Not 100%

Mac & iOS are built on reference types

Your App

- X% references
- Y% values

Model

Benefits

1. comprehensible
2. testable
3. parallel
4. portable

comprehensible

testable

parallel

portable

The Value of Values

1. comprehensible
2. testable
3. parallel
4. portable

Questions?

Thank You!

Please Rate using the GOTO Guide!

Thank You!

Marc Prud'hommeaux - marc@glimpse.io

Please rate this session with the GOTO Guide App