SWIFT!

```
class Session {
    let title: String // constant non-optional field: can never be null and can never be changed
    var instruktør: Person? // variable optional field: null is permitted
    var attendees: [Person] = [] // array that can only contain Person instances
    var requirements: [String: Float] = [:] // dictionary of String to Float

    init(title: String) { self.title = title } // required initializer w/ named parameter

    func ready() -> Bool { return attendees.count > 0 } // functon with return value
    func begin() { println("Let's teach some Swift to \(attendees.count) eager learners!") }
}
struct Person { let firstName, lastName, email: String } // immutable structure with automatic init

let sesh = Session(title: "Swift!") // class initializer
// struct initializer
sesh.instruktør = Person(firstName: "Marc", lastName: "Prud'hommeaux", email: "marc@impathic.com")
sesh.requirements["Mac OS"] = 10.10 // dictionary value assignment
sesh.requirements["Xcode"] = 6.10

if training.ready() { // braces required, parenthesis optional
    training.begin() // semicolons optional
}
```

Marc Prud'hommeaux

YOW! 2014 CONFERENCE

About Me

- Marc Prud'hommeaux, American
- Indie Software Developer
- Neither French nor an Apple employee

Swift

- Announced at WWDC June 2014
- It is: “a new language for the future of Apple software development”

Let's Build a Calculator App!

- https://gist.github.com/mprudhom

Thank You!

- Marc Prud'hommeaux <marc@impathic.com>
- Twitter: @mprudhom
- Apps: http://www.impathic.com

Agenda

- Swift vs. X Language Comparison
- Swift Language Features
- Swift Demo

Language Comparison

Objective-C

- Very different!

ObjC vs. Swift

```objc
@interface Concatinator : NSObject
@property NSString *separator;
@end
```

```objc
@implementation Concatinator
- (NSString *)concat:(NSString *)a withString:(NSString *)b {
    return [[a stringByAppendingString:self.separator]
        stringByAppendingString:b];
}
@end
```

ObjC vs. Swift

```
class Concatinator {
    var separator: String = ","

    func concat(a: String, b: String) -> String {
        return a + separator + b
    }
}
```

Java/C#

- Similar

Java/C# vs. Swift

```
class Concatinator {
    String separator = ",";

    String concat(String a, String b) {
        return a + separator + b;
    }
}
```

Java/C# vs. Swift

```
class Concatinator {
    var separator: String = ","

    func concat(a: String, b: String) -> String {
        return a + separator + b
    }
}
```

Scala

- Very similar!

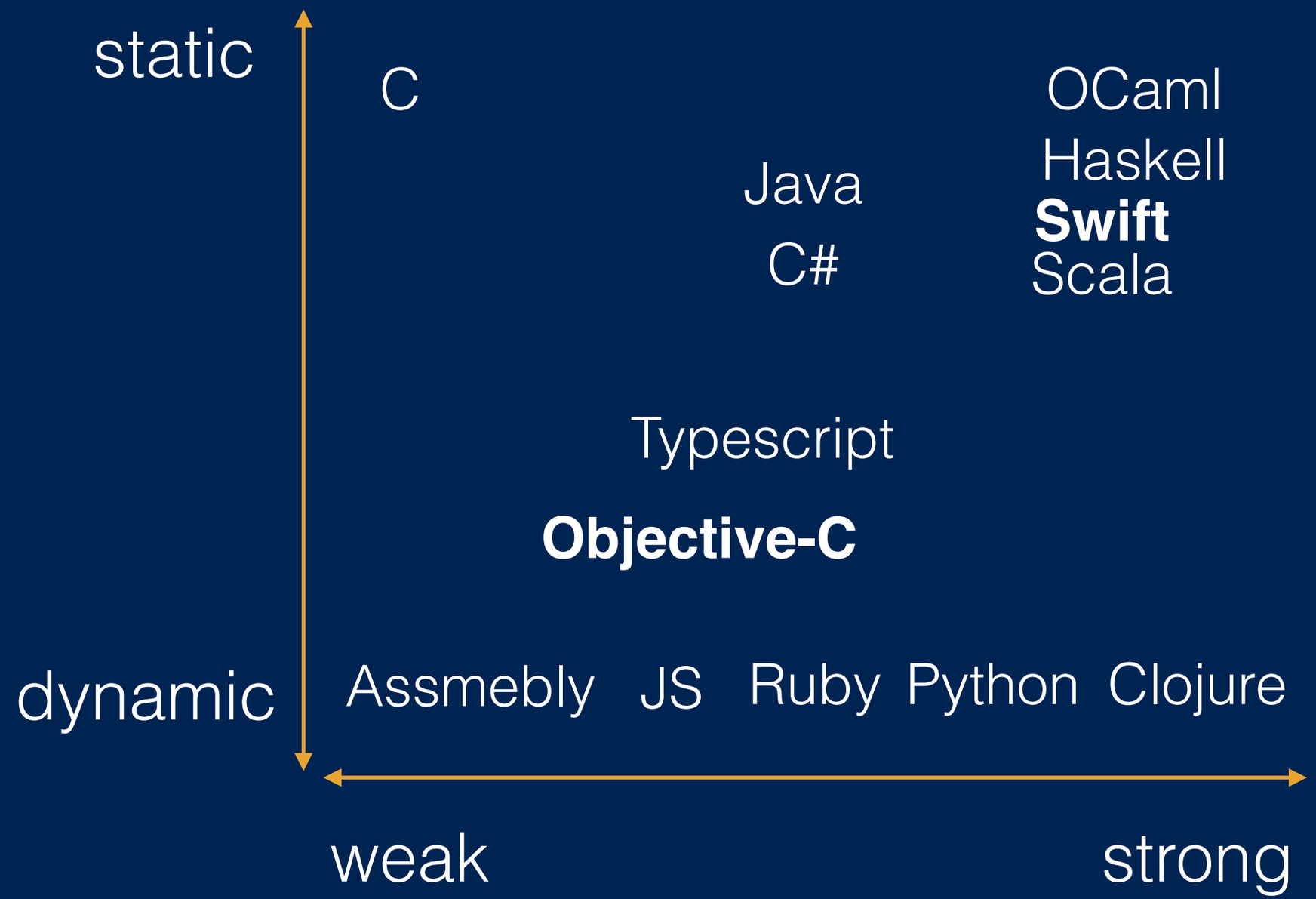
Scala vs. Swift

```
class Concatinator {
    var separator: String = ","

    def concat(a: String, b: String) : String = {
        return a + separator + b
    }
}
```

Scala vs. Swift

```
class Concatinator {
    var separator: String = ","

    func concat(a: String, b: String) -> String {
        return a + separator + b
    }
}
```

Type Systems Landscape



Slide Credit: Martin Odersky, "The Trouble with Types"

Language Features

(Im)mutability

- `let`: constant
- `var`: variable

let vs. var

```
let str: String = "Hello"
```

let vs. var

```
let str: String = "Hello"
str = str + " World" // illegal!
```

let vs. var

```
var str: String = "Hello"
str = str + " World" // legal
```

let vs. var

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

Optionals

- Optionals permit `nil`-ability
- Designated with `Type?` or `Optional<Type>`

Optionals

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

Optionals

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

Optionals

```
var str: Optional<String> = "Hello"
str = nil // legal
```

Generics

- Allow the parameterization of types

Generics

```
var strings: Array<String> = ["a", "b", "c"]
```

Generics

```
var strmap: [String : Int] = ["a": 1, "b": 2, "c": 3]
```

Type Inference

- Compiler figures out the type for you

Basic Type Inference

```
var str: String = "abc"
```

Basic Type Inference

```
var str = "abc"
```

Basic Type Inference

```
var strs: Array<String> = ["a", "b", "c"]
```

Basic Type Inference

```
var strs = ["a", "b", "c"]
```

Basic Type Inference

```
var strmap: [String : Int] = ["a": 1, "b": 2, "c": 3]
```

Basic Type Inference

```
var strmap = ["a": 1, "b": 2, "c": 3]
```

Simple Type Inference

```
[1, 2, 3].reverse()
    .map({ Double($0) })
    .filter({ $0 >= 2.0 })
    .map({ Float($0) })
```

Simple Type Inference

```
let floats : Array<Float> =
    [1, 2, 3].reverse()
        .map({ Double($0) })
    .filter({ $0 >= 2.0 })
    .map({ Float($0) })
```

Simple Type Inference

```
let floats =
    [1, 2, 3].reverse()
        .map({ Double($0) })
    .filter({ $0 >= 2.0 })
    .map({ Float($0) })
```

Complex Type Inference

```
lazy([1, 2, 3])
    .reverse()
    .map({ Double($0) })
    .filter({ $0 >= 0 })
    .map({ Float($0) })
```

Complex Type Inference

```
let mapped : LazySequence
    <MapSequenceView
        <FilterSequenceView
            <MapCollectionView
                <RandomAccessReverseView
                    <[Int]>, Double>>, Float>> =
lazy([1, 2, 3])
    .reverse()
    .map({ Double($0) })
    .filter({ $0 >= 0 })
    .map({ Float($0) })
```

Complex Type Inference

```
let mapped =
    lazy([1, 2, 3])
        .reverse()
        .map({ Double($0) })
        .filter({ $0 >= 0 })
        .map({ Float($0) })
```

Enumerations

- Fixed list of values

- With associated types!

Enums w/ Associated Types

```
enum Stuff<T> {
    case Nothing
    case Something(T)
}
```

```
let stuff: Stuff = Stuff.Something("Hello")

switch stuff {
case .Nothing:
    println("Nothing at all")
case .Something(let value):
    println("Something: \(value)")
}
```

Structs

- structs are value types: they are copied when passed around or re-assigned
- vs. classes, which are reference types
- Swift's built-in String, Array, and Dictionary are structs!
- Unlike Cocoa's built-in NSString, NSArray, NSDictionary, which are reference types

class vs. struct

```
class Concatinator {
    var separator: String = ","

    func concat(a: String, b: String) -> String {
        return a + separator + b
    }
}
```

class vs. struct

```
struct Concatinator {
    var separator: String = ","

    func concat(a: String, b: String) -> String {
        return a + separator + b
    }
}
```

Functions & Closures

- `func` keyword
- First-class types!
- Functions within functions
- Functions returning functions
- Functions accepting functions arguments

Functions & Closures

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

let sum = add(1, 2)
```

Functions & Closures

```
let add : (Int, Int) -> Int = { (a, b) in
    return a + b
}

let sum = add(1, 2)
```

Functions & Closures

```
let add : (Int, Int) -> Int = { (a, b) in
    return a + b
}

let sum = add(1, 2)
```

Other Features

- Tuples (anonymous structs)
- Operator Overloading (operators as functions)
- Named parameters & default parameter values
- Pattern Matching ("switch on steroids")
- Function Currying (Haskell might be proud)

Shold I Adopt Swift?

Pros	Cons
Familiar Syntax for Java/C# devs	Unfamilliar to ObjC devs
Faster Data Structures	Inflexible Runtime
Staticly Typed	Not Dynamically Typed
Playgrounds & REPL	Immature Tooling
Good ObjC & C interop	No C++ or Assembly interop
Functional Style	Functional Style
Mature Runtime	Immature Tooling
It's the Future	(probably)

Environment

Runtime

- Same as Objective-C Runtime
- Shared memory with Cocoa classes
- Automatic Reference Counting (ARC)

Cocoa Interoperability

- Swift can access all Cocoa Frameworks
- Objective-C can access some Swift code

Development Environment

- Xcode
- Swift REPL
- Swift Playgrounds

Playground Demo

```swift
import Darwin

struct Calc<T> {
    var lt: () -> T
    var op: (T, T) -> T
    var rt: () -> T

    init(lt: () -> T, op: (T, T) -> T, rt: () -> T) {
        self.lt = lt
        self.op = op
        self.rt = rt
    }

    init(const: T) {
        self.init(lt: { const }, op: { lt, rt in lt }, rt: { const })
    }

    var run: (() -> T) {
        return { self.op(self.lt(), self.rt()) }
    }

    mutating func push(op: (T, T)->T, value: T) {
        self = Calc(lt: self.run, op: op, rt: { value })
    }

}

infix operator ** { associativity left precedence 170 }

func ** (num: Double, power: Double) -> Double {
    return pow(num, power)
}

var clc = Calc(const: 1.0)
clc.push(+, value: 2)
clc.push(**, value: 3)

clc.run()
```

Swift Adoption Considerations

Immature Tooling

- Xcode
- SourceKit!

Uncertain Future

- Apple's history: Java, Python, Ruby, GC

Compiler Limitations

- enums with associated types: "unimplemented IR generation feature non-fixed multi-payload enum layout"
- class variables: "class variables not yet supported"
- no currying struct methods: "partial application of struct method is not allowed"
- perplexing error messages: "'() -> T' is not a subtype of 'UInt8'"

Future Compatibility

- Compiled Swift code will continue to run
- Language syntax will change

No Garbage Collection

- ARC requires that you manually break reference cycles

- ...or declare one side to be `weak` or `unowned`

Error Handling

- There is none!

Concurrency

- None!
- No locks, no synchronized, no nothing

Closedness

- LLVM & clang are open-source
- Swift standard library is not

Non-Portability

- Unlikely to be running on non-Apple devices anytime soon

Static vs. Dynamic

- Swift eschews Objective-C's dynamic dispatch
- No message passing
- No `objc_msgsend`

No C++/Assembly Interop

- Swift interoperates with Objective-C & C
- Does not interoperate with C++ or assembly

Objective-C to Swift interoperability

- Swift fully interoperates with Objective-C
- Objective-C partially interoperates with Swift

OS Support

- iOS 7+
- MacOS 10.10+

Thank You!

- Marc Prud'hommeaux <marc@impathic.com>
- Twitter: @mprudhom
- Apps: http://www.impathic.com