Swift
A Whirlwind Tour

Chris Lattner

Father of Swift (and of LLVM)

I started work on the Swift Programming Language in July of 2010. I implemented much of the basic language structure, with only a few people knowing of its existence. A few other (amazing) people started contributing in earnest late in 2011, and it became a major focus for the Apple Developer Tools group in July 2013.

The Swift language is the product of tireless effort from a team of language experts, documentation gurus, compiler optimization ninjas, and an incredibly important internal dogfooding group who provided feedback to help refine and battle-test ideas. Of course, it also greatly benefited from the experiences hard-won by many other languages in the field, drawing ideas from Objective-C, Rust,

Playgrounds

What we will cover

- Swift 2 (Xcode 7 Beta 3) (anything prior to Xcode 7 is Swift 1.2)
- Variables and Constants
    - Optionals
- Collections
- Control Flow
- Enums
- Structs
- Classes
    - Initialisation
- Methods / Functions
- Protocols
- Error Handling
- Some annotations

What we won’t cover

- Type System
- Functional aspects of Swift
- API availability checking
- Permutations of switch (see pattern matching)
- Generics
- Deep initialization
- Variadic functions
- guard and defer
- Swift Objective-C interoperability
- Memory Management

What is Swift about ?

Safe

Modern

Power

Protocols

Basic Types

Int = Int64 on 64 bit system
Int = Int32 on 32 bit system

Swift also has UInt,UInt8,UInt16,UInt32,UInt64

And Int,Int8,Int16,Int32,Int64

Float,Double and Character

And of course, String

These are all VALUE types

Variables and Constants

```
var someNumber: Int

var someNumber = 23 // Int is INFERRED

let someValue = 47 // Int is INFERRED
```

```
let someValue = Float(47)
```

Sometimes the compiler get's a bit fussy about types such as CGFloat and Float etc
Can convert with an initializer

```
var someFloat = 23.0
let newFloat:CGFloat = CGFloat(someFloat)
```

Optionals

New concept introduced by Swift, although C# has nullable types

We often want to represent the case where there is no valid value

```
let kilosInStorage = [“flour”:14,”sugar”:3,”pasta”:9]
let kilos:Int? = kilosInStorage[“cornflour”]
if kilos == nil
 {
 print(“we are out of cornflour”)
 }
```

We talk of wrapped and unwrapped optionals

We can explicitly (force) unwrap an optional with the ! (bang / exclamation mark)

```
print(kilos!)
```

Declaring Optionals

```
var someValue:Int? // Explicit unwrapping

var someValue:Int! // Implicit unwrapping
```

Implicitly unwrapped optionals are useful in certain circumstances where we know an optional will have a value by the time we use it

For example : an IBOutlet

```
class ViewController
  {
  @IBOutlet var firstNameField:UITextField!
  @IBOutlet var lastNameField:UITextField!

  func getFirstName() -> String
    {
    return(firstNameField.text)
    // without implicit unwrapping we would have to write
    // return(firstNameField!.text)
    }
```

Unwrapping Optionals

Several ways of unwrapping optionals

```
if optionalValue == nil
 {
 value = optionalValue!
 }
```

```
if let value = optionalValue
 {
 print(“\(value)”)
 }
```

Optional chaining

```
value = optional?.subValue?.subSubValue
```

```
value = optional!.subValue!.subSubValue
```

Strings

```
let someString = “This is a string” // inferred as String
```

All Strings double as NSString (which means NSString methods are available)

```
let components = “Home/vincent/Documents”.pathComponents
// [“Home”,”vincent”,”Documents”]
```

Strings in Swift are VALUE types, not reference types as in Objective-C

Hence they are immutable

Copy on write so the expense is kept to a minimum

Immutability is a recurring theme in Swift

Buildings Strings

```
let someString = ""

if someString.isEmpty
 {
 someString += "Something" // "Something"
 }
```

```
let character:Character = "!"

somestring.append(character) // "Something!"
```

String Interpolation

```
let someInt = 23
let someFloat = 23

let someText = “\(someInt) + \(someFloat) = \(someInt
+someFloat)”

// someText = 23 + 23 = 46
```

```
let string = String(format:”%d + %f =
%f”,someInt,someFloat,someInt+someFloat)
```

Array and Dictionary

```
let array = [“23”,”46”,”69”] // Compiler infers [String]
let array = [23,46,69] // Compiler infers [Int]


var array:[String] = [String]()
var array:[String] = [] // Compiler infers

let dict = [“one”:1,”two”:2,”three”:3] // Compiler infers
[String:Int]

let dict:[String:CGFloat] = [:] // Compiler infers
```

Working with Arrays

```
let array = [“23”,”46”,”69”] // Compiler infers [String]

array[0] // 23

array[0] = 24 // error because array is a constant

array.append(25) // error, because array is a constant
```

```
var array = [24,25,26] // Compiler infers [Int]

array[0] = 47

array.append(48)

array += [49] // Note the array literal
```

Working with Dictionaries

```
let dict = [“key1”:23,”key2”,46,”key3”:69]

value = dict[“key1”] // value is Int?

let actualValue = value!

dict[“newKey”] = 92

dict[“key1”] = 12
```

Control Flow

- repeat-while or while
- do
- for in
- if else
- switch
- continue
- return
- fallthrough
- break

Repeat - While

```
repeat
  {
  }
while condition

while condition
{
}
```

If - else

```
if let value = optionalValue // Note - no parenthesis
 {
 print(“value is \(value)”) // Braces are mandatory
 }
else
 {
 print(“value is nil”)
 }
```

Ranges

- Closed range operator … (1…5, start…finish)
- Half Range (1..<5)

for loop

- Two forms
- With range or collections
    - for index in 1…5
    - for element in array
    - for (key,value) in dictionary
- Traditional C form (note the lack of parenthesis)
    - for var index = 0; index < 5; index++

Switch

```
switch value
 {
 case value:
  value1 statements
 case value:
  value2 statements
 default:
  default statements
 }
```

- Note the lack of break statements
- Note that there is NO fall through

Functions

```
func name (
            externalName internalName:Type,
            externalName internalName:Type
            )
            ->
            Return Type
            {
            Function Body
            }

func add(first:Int,second:Int) -> Int
  {
  return(first+second)
  }

add(first:3,second:5)

func add(_ first:Int,second:Int) -> Int
  {
  return(first+second)
  }

add(3,second:5)
```

Closures

“Closures are self-contained blocks of functionality that can be passed around and used in your code. Closures in Swift are similar to blocks in C and Objective-C and to lambdas in other programming languages.”

Excerpt From: Apple Inc. “The Swift Programming Language (Swift 2 Prerelease).” iBooks. https://itun.es/us/k5SW7.l

```
{
  ( parameters ) -> return type in
    statements
}


{
  ( number:Int, string:String ) -> Void in
    return(“\(number) and \(string)”)
}
```

Enumerations

“An enumeration defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.”

Excerpt From: Apple Inc. “The Swift Programming Language (Swift 2 Prerelease).” iBooks. https://itun.es/us/k5SW7.l

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

An enumeration can have a raw type, but does not require one.
It can be any simple type, Int, Float, String, Character
If it has one it receives an raw type based initialiser
Enumerations can also have methods defined on them

Working with Enumerations

```
enum Currency : Int
 {
 case Rand = 1
 case Yen = 2
 case Pound = 4
 case Dollar = 10
init?(rawValue:Int) // Faillable initializer
}

// To assign for the first time, specify full name
var value = Currency.Rand
// thereafter, use the abbreviated form
value = .Yen
// Assign using rawValue
newValue = Currency(rawValue:10) // Inferred as Currency?
otherValue = Currency(rawValue:11) // nil
```

Working with Enumerations

An enumeration can also have an associated value
This is a natural extension to an enumeration when it would be logical to have some data associated with it

```
enum BarCode
  {
  UPCA(Int,Int,Int,Int)
  QRCode(String)
  }

var value = BarCode.UPCA(3,4567,3245,9)
var value = .QRCode(“ABCDEF”)

switch value
  {
  case .UPCA(let system,let manufacturer,let product,let checksum):
    print(“\(system) \(manufacturer) \(product) \(checksum)”)
  case .QRCode(let string):
    print(“\(string)”)
  }
```

Structures and Classes

“Classes and structures are general-purpose, flexible constructs that become the building blocks of your program’s code. You define properties and methods to add functionality to your classes and structures by using exactly the same syntax as for constants, variables, and functions.”

Excerpt From: Apple Inc. “The Swift Programming Language (Swift 2 Prerelease).” iBooks. https://itun.es/us/k5SW7.l

Unlike Objective-C Swift does not have header files, so classes and structures and their methods are defined in a single file.

All classes, structures and enumerations are available in the module that defines them

Structures and Classes

Classes and Structures in Swift have much in common

- They define properties to store values
- They define methods to provide functionality
- They define initializers to define initial state
- They conform to protocols

Classes have additional capabilities that structures don't

- They can inherit from other classes (single inheritance only)
- Type casting enables checking of an instance’s type at runtime
- They can have a deinitializer
- They are REFERENCE types not VALUE types like structures

Structures and Classes

```
struct Moon
  {
  var name:String = “”
  var radius:Float = 0.0
  var distanceFromPlanet:Float = 0.0
  let material = “rock”
  }

class Planet
  {
  var name:String = “”
  var moons:[Moon] = []
  var radius:Float = 0.0
  let material = “steel”
  }
```

Properties are accessed using the familiar “dot” notation

```
instance.name
instance.radius
```

Instances

Instances are created using initializer syntax

```
let moon = Moon() // Create a moon
moon.name = “Luna” // Name the moon
let planet = Planet() // Create a planet
planet.name = “Earth” // Name the planet
planet.moons.append(moon) // Attach the moon to the planet
```

It is possible to directly invoke the init method of a class, for example :-

```
let moon = Moon.init() // Create a moon
let planet = Planet.init() // Create a planet
```

REMEMBER : Structures are VALUE types, but classes are REFERENCE types

When to choose Structures

When the structure’s primary purpose is to encapsulate a few relatively simple data structures

When it it reasonable to expect that the encapsulated values will be copied rather than referenced when you assign them or pass them around

Properties stored by the structure are in fact structures themselves

There is no need for inheritance

Properties

We saw properties earlier on when we defined some Moons and Planets

Those were stored properties (var and let)

Classes and Structures can also have computed properties

```
struct Moon
  {
var gravity:Float // Read only
  {
  get
    {
    return(0.8)
    }
  }

var magneticField:Float // Read write
  {
  get
    {
    return(savedField)
    }
  set(newValue)
    {
    savedField = newValue * 10.0
    }
  }
}
```

Property Observers

Property observers observe and respond to changes in properties

They can be added to all properties, even inherited ones

```
struct Moon
 {
 var gravity:Float // Read only
  {
  willSet(newValue)
   {
   do something important
   }
  didSet
   {
   do something important
   }
}
```

Type Properties

Can store properties at the type level - these are called Type Properties

They are declared using the static keyword and MUST have a default value

```
struct Moon
 {
 static var MoonColor:UIColor = UIColor.whiteColor()
 }

let localColor = Moon.MoonColor
```

Methods

Methods are functions that are associated with a particular class or structure

They are declared in a similar way to properties

```
class Planet
  {
  var weight:Float!
  var color:UIColor!

  func setWeight(weight:Float,color:UIColor)
    {
    self.weight = weight
    self.color = color
    }

  class func planetGroup() -> PlanetGroup
    {
    return(PlanetGroup())
    }
  }
```

Modifying Value Types

Remember structures and unions are VALUE types

This means that by default the properties of value types can not be changed from methods on these classes, in order to do this they have to opt in to mutating behavior

```
struct Point
{
var x = 0
var y = 0

mutating func addOffset(offset:Int)
  {
  self = Point(x:x+offset,y:y+offset)
  }
}
```

Subclassing

Subclassing is done in the normal way

```
class Planet
  {
  var name:String?

  func setWeight(weight:Float)
    {
    …
    }
  }

class EarthTypePlanet:Planet
  {
  override setWeight(weight:Float)
    {
    super.setWeight(weight:weight)
    …
    }
  }
```

Both properties and methods must be marked with override if they are in fact over ridden

Methods and properties can be marked with final to prevent them being overridden

Initialization

Initialization is a huge topic and deserves several hours dedicated just to it.

Let us just say that all properties need to be initialised in an initializer and that somewhere along the line, a designated initializer must be called.

Unless a property is an optional, has a default value or is an implicitly unwrapped optional, it’s value must be set in an initialiser BEFORE the super designated initialiser is called

```
class EarthTypePlanet:Planet
  {
  var weight:Float?
  var color:UIColor!
  var name:String

  init()
    {
    name = “None”
    super.init()
    }
  }
```

Please read the Swift 2.0 book section on initializers.

Deinitialization

A class can have a deinitializer which will get called when an instance is released

```
class EarthTypePlanet:Planet
  {
  var weight:Float?
  var color:UIColor!
  var name:String

  deinit
    {
    // free up resources
    }
}
```

Type Casting

Typecasting is done using the “as” keyword

```
class Body
  {
  var name:String?
  }

class Planetesimal : Body
  {
  }

class Planet : Planetesimal
  {
  }

var body1 = Body()
var body2 = Planetesimal()
var body3 = Planet()

let body4 = body3 as? Body // Body?
ley body5 = body2 as! Planet
```

Type Testing

Type testing is done using the “is” keyword

```
class Body
  {
  var name:String?
  }

class Planetesimal : Body
  {
  }

class Planet : Planetesimal
  {
  }

var body1 = Body()
var body2 = Planetesimal()
var body3 = Planet()

body1 is Planet // true
body2 is Planet // false
```

Extensions

Any type can be extended in Swift (including system types).
Extensions are a standard mechanism for composing behaviour

```
extension Type
  {
  func newFunction() -> Int
    {
    return(0)
    }
  }
```

As of Swift 2.0 extensions can NOT add new properties (rumour has it that this might be forthcoming in a later release of Swift)

Protocols

A **protocol** defines a blueprint of methods, properties and other requirements that suit a particular task or piece of functionality. The protocol can then be **adopted** by a class, a structure or an enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to **conform** to that protocol.

```
protocol SomeProtocol
 {
 func someFunction() -> Int
 var someVar:Int { get set }
 }

class SomeClass,SomeSuperClass,Protocol1,Protocol2
 {

 }
```

Protocols

Protocols can require initializers as part of their requirements. In this case the initializer is written as part of the requirements, just like any other function or method, but it must be marked as required by any implementation to ensure that all subclasses implement it

```
protocol Planetlike
  {
  init(parameter:Int)
  }

class PlanetaryBody,Planetlike
  {
  required init(parameter:Int)
    {
    self.parameter = parameter
    }
  }
```

Protocol Extensions

Conformance to a protocol can be added by means of an extension

```
extension Planet,Planetlike
  {
  }
```

Not only can an extension add conformance to a protocol, an extension can add behaviour (properties and methods) to a protocol

```
extension Planetlike
  {
  func albedo() -> Float
    {
    return(0.7)
    }
  }
```

Protocol Extensions

By adding properties and methods to a protocol, ALL types that adhere to the protocol gain the properties and behaviour defined by the protocol

Error Handling

“Error handling is the process of responding to and recovering from error conditions in your program. Swift provides first-class support for throwing, catching, propagating, and manipulating recoverable errors at runtime.”

Excerpt From: Apple Inc. “The Swift Programming Language (Swift 2 Prerelease).” iBooks. https://itun.es/us/k5SW7.l

Error conformance is handled by the ErrorType protocol in swift

Enumerations are ideal for the handling of errors

Merely adding conformance to the ErrorType protocol allows an enumeration to be used to throw and catch errors

Error Handling

```
enum PlanetaryFaults,ErrorType
  {
  case EarthQuake
  case Hurricane
  case NuclearPlantCollapse
  }
```

A function is marked as throwing an error by adding the “throws” clause

```
func performAction() throws -> Int
  {
  if somethingWentWrong
    {
    throw PlanetaryFaults.EarthQuake
    }
  }
```

Error Handling

```
do
  {
  try performAction()
  }
catch PlanetaryFaults.EarthQuake
  {
  print(“No ! An earthquake”)
  }
catch PlanetaryFaults.Hurricane
  {
  print(“Wow - wet“)
  }
catch PlanetaryFaults.NuclearPlantCollapse
  {
  print(“Kiss your ass goodbye”)
  }
```

Error Handling of Objective-C

```
func fetchData(data:NSMutableData,inout error:NSError)

func fetchData(data:NSMutableData) -> NSError
```

Swift will convert this to

```
func fetchData(data:NSMutableData) throws
```

It is worth noting Swift error handling is NOT like the throws-catch exception handling in other languages. Catching an error does NOT involve unwinding the call stack the way other languages do it, and the overhead associated with catch is more along the lines of an if-else.

Memory Management

Automated Reference Counting

Exactly the same as Objective C

Use of weak keyword to avoid cycles

Just mention that Swift also has an unowned keyword can be used when you know that a reference will never be nil once it has been set during initialization

Privacy Modifiers

- public - internal and external to the module
- internal - internal to the module
- private - internal to the structure,enum or class

Annotations

- @objc (two forms)
- @IBAction
- @IBOutlet
- dynamic
- @IBInspectable
- @IBDesignable

Some Resources

- Swift 2 Book (PreRelease)
- http://developer.apple.com/swift
- https://developer.apple.com/videos/wwdc/2015/
- Plenty of resources on the web for Swift
- Dozens of sites and blogs