CONTENTS

Swift

BY ALEX BLEWITT

INTRODUCTION

Swift is the fastest growing programming language today. It is built on LLVM, which is a key part of both Clang and other popular native runtimes like Rust and Julia. It's easy to learn, compiles down to native code, and interoperates with existing languages like Objective-C. Developers are increasingly using this modern, type-safe, object-oriented/functional programming language for more than client-side mobile apps. This is because this popular mobile language moved into an open source project at Swift.org under an Apache license in December of 2015, just over a year after it was introduced at Apple's WWDC 2014 developer event. Developers from around the world are working to ensure Swift is available for use in other environments and for efforts spanning Web, Mobile, Cloud and IoT. As Swift, the programming language, matures, it can increasingly be the language of choice end-to-end, bringing its modern advantages to both client-side and server-side deployments.

GETTING STARTED

Although Swift is a compiled language, it comes with an interpreter `swift` which can be used to experiment with files and functions. For example, a factorial function can be defined in the interpreter as follows:

```
$ swift
  Welcome to Swift
   1>  func factorial(i:UInt) -> UInt {
   2.    if i == 0 {
   3.      return 1
   4.    } else {
   5.      return i * factorial(i-1)
   6.    }
   7.  }
   8> factorial(5)
   $R0: UInt = 120
```

This shows several features of Swift: the interpreter, which makes it really easy to experiment with code; the fact that the language is strongly typed (the function takes an unsigned integer argument, and returns an unsigned integer argument); and that--unlike C-derived languages--the type is on the right of the variable instead of the left (i.e. `i:UInt` and not `UInt i`).

The reason for the type appearing on the right is that Swift uses type inference where possible so that the types don't need to be declared:

```
 9> let fiveFactorial = factorial(5)
  fiveFactorial: UInt = 120
```

The `let` is a constant definition, which means it can't be changed. It has also been inferred to be a `UInt`, because that's what the return type of `factorial` was. Variables also exist, and are declared using the `var` keyword:

```
10> var myFactorial = factorial(6)
  myFactorial: UInt = 720
  11> myFactorial = factorial(7)
  myFactorial: UInt = 5040
```

Being a typed language, it is not possible to assign a value to a variable of the wrong type (or to initialize a constant with the wrong type):

```
12> myFactorial = -1
  error: negative integer '-1' overflows when stored into
  unsigned type 'UInt'
```

OPTIONALITY

Swift has a specific way of dealing with optionality; it encodes it into the type system. Instead of having values which may be reference types having a `nil` value by default, variables explicitly opt-in to being an optional type or not. This also applies to value types like integers, although the overhead is far less than a primitive wrapper like `Integer` in languages such as Java.

```
 1> var thinkOfANumber: Int?
  thinkOfANumber: Int? = nil
   2> thinkOfANumber = 4; // chosen by random dice roll
   thinkOfANumber: Int? = 4
```

Under the covers, there is a generic `Optional<Int>` type, but for brevity the `?` suffix can be used instead, which means the same thing.

Optional values can be unpacked using either `??` (which allows for a default value to be substituted in case of being `nil`) or forcibly with `!`. In most cases these are unnecessary since it's possible to call a function using `?.` on an optional type, which returns an optional result, or to use a `map` function to transform it into a different value.

IBM Brings Swift to the Cloud

IBM Cloud

Radically simplify end-to-end development of modern apps

Runtime Support

KITURA

IBM Swift Runtime

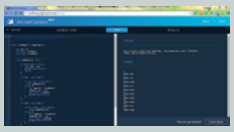
Package Sharing

IBM SWIFT
Package Catalog

Developer Recources

| Swift@ **IBM**

devCenter



IBM SWIFT Sandbox

- ✓ Experiment with Swift on the server
- ✓ Develop and deploy end-to-end applications
- ✓ Share Swift packages, libraries, and modules

Discover Swift@IBM

```
3> thinkOfANumber?.successor()
$R0: Int? = 5
4> thinkOfANumber ?? 0
$R1: Int? = 4
5> thinkOfANumber = nil
6> thinkOfANumber?.successor()
$R2: Int? = nil
7> thinkOfANumber ?? 0
$R3: Int = 0
```

CLASSES AND STRUCTS

Swift has reference types and value types in the language, which are defined in classes and structs. Classes are passed into functions by reference, and structs are passed by value (copied). They can both have methods associated with them, and the syntax for calling methods is identical in both cases. The only significant difference is that structs can't inherit from other structs (although they can embed them).

```
1> struct Time {
  2.   var hh: UInt8
  3.   var mm: UInt8
  4.   var ss: UInt8
  5.   var display: String {
  6.     return "\(hh):\(mm):\(ss)"
  7.   }
  8. }
  9> class Clock {
 10.   let time: Time
 11.   let h24: Bool
 12.   init(h24: Bool = true) {
 13.     self.h24 = h24
 14.     self.time = Time(hh: h24 ? 13 : 1, mm:15, ss:23)
 15.   }
 16. }
 17> Clock().time.display
 $R0: String = "13:15:23"
```

The example above demonstrates how a struct and a class are defined. A value type `Time` (storing the hours, minutes, and seconds as unsigned bytes – `UInt8`) allows mutable updates to the fields, and has a computed property `display` that returns a formatted `String`. Computed properties are represented with `var` (if they are mutable) or `let` (if they are immutable). A function could have been used instead, but then the invocation would have been `time()` in the example at line 3 in that case.

The class `Clock` stores a `Time` struct inside. Unlike languages such as Java or C#, the struct is in-lined into the class definition rather than as a separate pointer to a different area of memory. (This helps cache locality and reduces memory fragmentation.)

Constructors have the special name `init`, and can have named arguments. In this case, the property `h24` can be configured from initialization, although default parameter values can be used to supply the values on demand. Constructors for classes and structs are both called with the class name; `Clock()` and `Time()` will create instances of both. In the case of structures, the constructor is automatically created based on the names of the fields.

Finally, properties can be defined with either `let` or `var`. In this case, the `time` field is declared as a constant with `let`, so although the `Time` type allows mutability of its fields in general, in this specific case the `time` field cannot be mutated:

```
18> Clock().time.hh = 10
  error: cannot assign to property: 'time' is a 'let'
  constant
```

PROTOCOLS AND EXTENSIONS

It is possible to extend an existing class or struct by adding functions to the type. The new method can be used anywhere the extension is in scope.

```
19> extension Clock {
  20.   func tick() -> String {
  21.     return "tock"
  22.   }
  23. }
  24> Clock().tick()
  $R1: String = "tock"
```

Adding extensions to types doesn't require the code for the original to be changed; so it's possible to extend the built-in types with additional functionality:

```
25> 1.negate()
  error: value of type 'Int' has no member 'negate'
  26> extension Int {
  27.   func negate() -> Int {
  28.     return -self
  29.   }
  30. }
  31> 1.negate()
  $R2: Int = -1
```

This exposes the fact that a literal `1` is actually a struct type; we can add new functions to the structure that perform different operations. Since it's an immutable struct, we can't change the value of the constant; but we can add constant properties and functions to the mix.

This hasn't added negation to all integer types though, only the `Int` type -- which is a typealias for the platform's default word size (64 bits on a 64-bit system, 32 bits on a 32-bit system).

```
18> Int8(1).negate()
  error: value of type 'Int8' has no member 'negate'
```

This can be solved by adding an extension to a protocol instead of the actual type. A protocol is like an interface in other languages; it's an abstract type representation that has no implementation body, and types (both struct and class) can adopt protocols at definition time.

The `Int` type adopts several protocols; `IntegerType` (which is the parent of all integer types, including unsigned ones) and `SignedNumberType` (amongst others). We can add the extension to this protocol instead:

```
19> extension SignedNumberType {
  20.   func negate() -> Self {
  21.     return -self
  22.   }
  23. }
```

Note that the return type of the `negate` function is now `Self`--this is a special type for generic values that returns the actual type of the object, rather than a generic implementation of the `SignedNumberType` interface. This allows an `Int8` to be negated into an `Int8`, and an `Int16` to be negated to an `Int16`.

```
24> Int8(1).negate()
  $R3: Int8 = -1
  25> Int16(1).negate()
  $R4: Int16 = -1
```

Extensions can be used to retroactively adopt procotols for classes. Unlike Go, which uses structural interface adoption to automatically align interfaces, Swift uses an opt-in model where types explicitly have to say that they adopt the protocol in order to use it. Unlike Java or C#, these don't have to be done at the type definition time; they can be added retroactively through extensions.

```
26> extension Clock: CustomStringConvertible {
  27.   public var description: String {
  28.     return "The time is: \(time.display)"
  29.   }
  30. }
  31> print(Clock())
  The time is: 13:15:23
```

FUNCTIONAL PROGRAMMING

As well as object-oriented programming, Swift can be used for functional programming. Built-in data structures provide a way of taking existing functions and processing values:

```
1> func factorial(i:UInt) -> UInt {
  2.   if i == 0 { return 1 } else { return i *
 factorial(i-1) }
  3. }
  4> [1,2,3,4,5].map(factorial)
  $R0: [UInt] = 5 values { 1, 2, 6, 24, 120 }
```

Swift also allows anonymous functions using a {} block and an `in` to separate functions from the return result:

```
5> [1,2,3,4,5].map( { i in i * 2 } )
  $R1: [Int] = 3 values { 2, 4, 6, 8, 10  }
```

This can be syntactically rewritten by placing the anonymous function after the map call:

```
6> [1,2,3,4,5].map() {
  7.   i in i * 2
  8. }
  $R2: [Int] = 3 values { 2,4,5,8,10 }
```

It's also possible to iterate over loops using a `for ... in` with an optional `where` clause:

```
9> for i in [1,2,3,4,5] where i % 2 == 0 {
 10.   print(i)
 11. }
 2
 4
```

MODULES AND MORE

The Swift language is yet young, and the ABI stability or binary compatibility hasn't arrived – that's coming with Swift 3. However, Swift already has the concept of modules, both for importing and for consuming by other applications. On iOS, existing Objective-C frameworks can be imported trivially:

```
1> import UIKit
  2> class ExampleController : UIViewController {
  3.   ...
  4. }
```

There's work progressing for the Swift package manager, by placing a `Package.swift` file and a `swift build` command that can compile and build the application. This is part of the Swift 2.2 build but may experience changes before the Swift 3 release in 2016.

Swift has a standard documentation format, and can produce a swiftdoc file that can be interpreted by tools, as well as translation to a web-based format. The markup is based on CommonMark, with attributes that are used to record parameters and return types.

SWIFT QUICK REFERENCE

OPERATORS

OPERATOR	FUNCTION
+	Addition operator.
-	Subtraction operator/unary minus (negative value) operator.
*	Multiplication operator.
/	Division operator.
=	Assignment operator.
%	Remainder operator.
==	Equal to.
!=	Not equal to.
<	Less than.
<=	Less than or equal to.
>	Greater than.
>=	Greater than or equal to.
!	NOT logical operator.
&&	AND logical operator.
||	OR logical operator.
...	Range operator, inclusive.
..<	Range operator, exclusive of final value (up to but not including).
//	Single-line comment.
/*	Begin multiline comment.
*/	End multiline comment.

VARIABLES

KEYWORD	DEFINITION	EXAMPLE

`var`	Defines a mutable (changeable) variable. Variable may be explicitly typed, or Swift will infer variable type.	`var somevar: Int = 1`
		`var somevar = 1`
		`somevar = 2`

`let`	Defines an immutable (constant) variable. Variable may be explicitly typed, or Swift will infer variable type.	`let somevar: String = "something"` `let somevar = "different"`
`"\(x)"`	String interpolation; the value of the escaped variable x is inserted into the String.	`"text \(somevar)"`

TYPES

NAME	TYPE
Bool	Boolean values true and false.
Int	Signed integer value (whole number).
UInt	Unsigned integer value (non-negative whole number).
Float	32-bit floating-point number.
Double	64-bit floating-point number.
Character	Unicode scalar(s) producing a human-readable character.
String	A series of characters.
Tuple	A group of multiple values of any type.
Array	An indexed collection of data.
Set	An unordered collection of data of one hashable type.
Dictionary	A collection of key-value pairs.

CONTROL FLOW

LOOPS

KEYWORD	DESCRIPTION	EXAMPLE

while	Executes block as long as the while condition is true.	`while somevar > 1 {`
		`  print(somevar)`
		`}`

repeat-while	Executes a block, then repeats as long as the while condition is true.	`repeat {`
		`  print(somevar)`
		`} while somevar > 1`

for	Executes a block for a number of times determined by a boolean expression and iterator defined in the for loop statement.	`for var i = 0; i < 10;`
		`  ++i {`
		`  print(i)`
		`}`

for-in	Executes a block for each element in an array or range.	`for element in`
		`  someArray {`
		`  print(element)`
		`}`

CONDITIONALS

KEYWORD	DESCRIPTION	EXAMPLE

if	Evaluates a boolean expression and executes a block if the expression is true.	`if somevar < 1 {`
		`  print("Less than one.")`
		`}`

else	Executes a block once an if statement evaluates as false.	`if somevar < 1 {`
		`  print("Less than one.")`
		`} else {`
		`  print("Not less than one.")`
		`}`

else if	Can chain if statements, evaluating a boolean expression once the previous conditional evaluates as false, then executing a block if its expression is true.	`if somevar < 1 {`
		`  print("Less than one.")`
		`} else if somevar == 1 {`
		`  print("Exactly one.")`
		`} else {`
		`  print("Greater than one.")`
		`}`

switch	Evaluates conditions based on case and executes a block for the first matching case; if no matching case exists, the switch statement executes the block for the default case.	`switch someSwitch {`
		`  case 1:`
		`  print("One")`
		`  case 2:`
		`  print ("Two")`
		`  default:`
		`  print("Not one or two.")`
		`}`

where	Evaluates additional conditionals in switch cases.	`switch someSwitch {`
		`  case 0:`
		`  print("Zero")`
		`  case 1:`
		`  print ("One")`
		`  case let x where x.isPrime():`
		`  print("Prime")`
		`  default:`
		`  print("Not prime")`
		`}`

guard	Evaluates a boolean expression and, if the expression is not true, exits the code block containing the guard statement.	`for element in someArray {`
		`  guard let x = element where x > 0 else {`
		`  break`
		`  }`
		`}`

CONTROL TRANSFER

KEYWORD	DESCRIPTION	EXAMPLE

fallthrough	Falls into the next case in a switch statement once a case has already matched.	`switch someSwitch {`
		`  case 3:`
		`  print(“Three.”)`
		`  fallthrough`
		`  case 2:`
		`  print(“Two.”)`
		`  fallthrough`
		`  case 1:`
		`  print(“One.”)`
		`  fallthrough`
		`  default:`
		`  print(“Done.”)`
		`}`

continue	Ends the current iteration of a loop and begins the next iteration.	`for i in 1...10 {`
		`  if i == 3 {`
		`  continue`
		`  } else {`
		`  print("\(i) is not three.")`
		`  }`
		`}`

break	Ends execution of a running loop and continues running code immediately after the loop. Also used to pass over unwanted cases in switch statements (as switch cases must be exhaustive and must not be empty).	`for i in 1...10 {`
		`  if i == 3 {`
		`  break`
		`  } else {`
		`  print("\(i) is less than three.")`
		`  }`
		`}`

OTHER RESOURCES

TOOLS

- AppCode: IDE for Mac OS and iOS software developed by JetBrains.
  jetbrains.com/objc/special/appcode/appcode.jsp
- CodeRunner: An alternative IDE to Xcode.
  coderunnerapp.com
- Xcode: Apple's IDE for developing software for Mac OS, iOS, WatchOS and tvOS.
  developer.apple.com/xcode/downloads
- RunSwift: Browser-based "code bin" which lets you test your code.
  runswiftlang.com

LIBRARIES

- Swift Toolbox
  swifttoolbox.io
- CocoaPods: Dependency manager for Swift projects.
  cocoapods.org
- Carthage: A simple dependency manager for Cocoa projects.
  github.com/Carthage/Carthage
- Wolg/awesome-swift: A curated list of Swift frameworks, libraries and software.
  github.com/Wolg/awesome-swift

COMMUNITIES

- Apple Developer Forums - Swift
  devforums.apple.com/community/tools/languages/swift
- StackOverflow: Swift Questions
  stackoverflow.com/questions/tagged/swift
- Google Groups: Swift
  groups.google.com/forum/#!forum/swift-language
- Swift Language
  swiftlang.eu/community
- Swift Meetups
  meetup.com/find/?keywords=Swift&radius=Infinity

MISC

- Apple Swift Resources
  developer.apple.com/swift/resources
- Github's Trending Swift Repositories: Track trending Swift repositories to find the hottest projects.
  github.com/trending?l=swift&since=monthly

ABOUT THE AUTHOR

DR. ALEX BLEWITT has over 20 years of experience in Objective-C and has been using Apple frameworks since NeXTSTEP 3.0. He upgraded his NeXTstation for a TiBook when Apple released Mac OS X in 2001 and has been developing on it ever since. He is the author of the recently published Swift Essentials. Alex currently works for a financial company in London and writes for the online technology news site InfoQ, as well as other books for Packt Publishing. He also has a number of apps on the App Store through Bandlem Limited. When he's not working on technology and the weather is nice, he likes to go flying.

For more of Alex's writing, you can find him on his blog, alblue.bandlem.com, as well as Twitter, @alblue.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

BROUGHT TO YOU IN PARTNERSHIP WITH
IBM