Swift Essentials

WRITTEN BY ALEX BLEWITT
UPDATED BY JAMES SUGRUE CHIEF TECHNOLOGY OFFICER, OVER-C

CONTENTS

› INTRODUCTION
› GETTING STARTED
› OPTIONALITY
› CLASSES AND STRUCTS
› PROTOCOLS AND EXTENSIONS
› FUNCTIONAL PROGRAMMING
› SECURITY IN SWIFT
› TESTING IN SWIFT
› SWIFT QUICK REFERENCE
› OTHER RESOURCES

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 the 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 for end-to-end use cases, 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
func factorial(_ i:UInt) -> UInt{
    if i == 0 {
        return 1
    }else {
        return i * factorial(i-1)
    }
}
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); the fact that although argument labels are possible for each variable, they can be omitted using the underscore operator; 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:

```
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` is. Variables also exist, and are declared using the `var` keyword:

IBM Security

Stop hiding from your security officer.

#CoverYourApps with IBM Security's leading application security solution. Introduce a strong, security-first mindset into your DevOps culture, without disrupting agile practices that your organization relies on to produce quality, on-time software. Start your free trial today!

IBM Application Security on Cloud enhances web and mobile application security, improves application security program management and strengthens regulatory compliance. Testing web and mobile applications prior to deployment helps you identify security risks, generate reports and receive fix recommendations.

Access Complimentary Trial Now

IBM Application Security on Cloud

```
var myFactorial = factorial(6)
myFactorial: UInt = 720
myFactorial = factorial(7)
print(myFactorial)
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):

```
myFactorial = -1
error: repl.swift:15:15: error: negative integer '-1'
overflows when stored into unsigned type 'UInt'
myFactorial = -1
```

OPTIONALITY

Swift has a specific way of dealing with optionality; it encodes it into the type system. Instead of having values that may be reference types having a `nil` value by default, variables explicitly opt-in to be 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.

```
var thinkOfANumber: Int?
  thinkOfANumber: Int? = nil
thinkOfANumber = 4 //chosen by random dice roll
print(thinkOfANumber)
Optional(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.

```
thinkOfANumber?.advanced(by:1)
$R0: Int? = 5
thinkOfANumber ??  0
$R1: Int = 4
thinkOfANumber = nil
thinkOfANumber?.advanced(by:1)
$R2: Int? = nil
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).

```
struct Time{
    var hh: UInt8
    var mm: UInt8
    var ss: UInt8
    var display: String {
        return "\(hh):\(mm):\(ss)"
    }
}
class Clock {
    let time: Time
    let h24: Bool
    init(h24: Bool = true) {
        self.h24 = h24
        self.time = Time(hh: h24 ? 13 : 1, mm:15, ss:23)
    }
}
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:

```
Clock().time.hh = 10
error: repl.swift:18:17: error: cannot assign to
property: 'time' is a 'let' constant
Clock().time.hh = 10
```

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.

```
extension Clock {
     func tick() -> String {
         return "tock"
     }
}
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:

```
extension Int {
     func negate() -> Int {
         return -self
     }
}
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).

```
Int8(1).negate()
error: repl.swift:30:1: error: cannot use mutating
member on immutable value: function call returns
immutable value
Int8(1).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; `FixedWidthInteger` and `SignedInteger` amongst others. We can add the extension to the `SignedInteger` protocol instead:

```
 extension SignedInteger {
      func negate() -> Self {
          return -self
      }
 }
```

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 `SignedInteger` interface. This allows an `Int8` to be negated into an `Int8`, and an `Int16` to be negated to an `Int16`.

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

Extensions can be used to retroactively adopt protocols 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.

```
 extension Clock: CustomStringConvertible {
      public var description: String {
          return "The time is: \(time.display)"
      }
  }
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:

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

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

```
[1,2,3,4,5].map({i in i * 2})
$R1: [Int] = 5 values {
  [0] = 2
  [1] = 4
  [2] = 6
  [3] = 8
  [4] = 10
}
```

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

```
[1,2,3,4,5].map() {
     i in i * 2
}
$R2: [Int] = 5 values {
  [0] = 2
  [1] = 4
  [2] = 6
  [3] = 8
  [4] = 10
}
```

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

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

SECURITY IN SWIFT

When building applications in Swift, as in any other language, security needs to be properly considered. When building iOS apps , some key areas to review when it comes to securing your code include:

- Using SSL on all communication so that messages are not intercepted and modified
- Configure App Transport Security to provide secure default behavior
- Validate incoming and outgoing URL handler calls
- Do not store passwords, private keys or identities within your application. Instead use Keychain Services.
- Avoid SQL injection attacks by using prepared statements in cases where SQLite is being used

Swift also has a number of libraries available to help with security related concerns such as SSL/TLS , JWT (**Swift-JWT**), and cryptography (**BlueCryptor**).

TESTING IN SWIFT

Testing in Swift is facilitated through `XCTest`, which follows the similar test paradigms that you will notice in most major languages. As well as covering the basics of unit testing, the XCTest framework includes APIs for performance testing as well as user interface testing.

Tst classes are always subclasses of `XCTestCase` which each test inside that class must exist in a function prefixed with `test`. Any number of assertions can be added to your tests using the `XCTAssert` function and its variants.

Asynchronous operations can be tested using `XCTestExpectation` which allows you to specify an amount of time to wait for the completion of the operation before the test is considered to have failed.

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.
`~`	Bitwise NOT operator.
`&`	Bitwise AND operator.
`^`	Bitwise XOR operator.
`<<` `>>`	Bitwise Left and RIght SHift Operators.
`...`	Range operator, inclusive.
`<..`	Range operator.
`//`	Single-line comment.
`/*`	Begin multiline comment.
`*/`	End multiline comment.

VARIABLES

Operator	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

Operator	Definition	Example
`while`	Execute 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.	`let somevar: String = "something"` `let somevar = "different"`
`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

Operator	Definition	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 statemetns, evaluating 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

Operator	Definition	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 Question
  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
- Swift Package Manager
  swift.org/package-manager
- Github's Trending Swift Repositories: Track trending Swift repositories to find the hottest projects
  github.com/trending?l=swift&since=monthly
- Secure iOS Application Development
  github.com/felixgr/secure-ios-app-dev
- Ponemon Institute's Mobile and IoT Application Security Testing Study:
  securityintelligence.com/10-key-findings-from-the-ponemon-institutes-mobile-iot-application-security-testing-study

Written by James Sugrue, Chief Technology Officer, Over-C

is Chief Architect at Over-C, building mobile applications and services for managing compliance, using NFC and Bluetooth sensors for proof of presence. James is an expert in Java and Swift, building everything from desktop applications to high performance servers and mobile applications in Android and iOS.

James has been a Zone Leader at DZone since 2008, and has written many Refcardz, as well as a book on Backbone.js.