Swift for Open Source Developers

Dr. Alex Blewitt

Apple announced Swift at WWDC 2014 as a new programming language that combines experience with the Objective-C platform and advances in dynamic and statically typed languages over the last few decades. Before Swift, most code written for iOS and OS X applications was in Objective-C, a set of object-oriented extensions to the C programming language. Swift aims to build upon patterns and frameworks of Objective-C but with a more modern runtime and automatic memory management. In December 2015, Apple open sourced Swift at https://swift.org and made binaries available for Linux as well as OS X. The content in this article can be run on either Linux or OS X. Developing iOS applications requires Xcode and OS X.

In this article, we will present the following topics:

- How to use the Swift REPL to evaluate Swift code
- The different types of Swift literals
- How to use arrays and dictionaries
- Functions and the different types of function arguments
- Compiling and running Swift from the command line

(For more resources related to this topic, see here.)

Open source Swift

Apple released Swift as an open source project in December 2015, hosted at https://github.com/apple/swift/ and related repositories. Information about the open source version of Swift is available from the https://swift.org site. The open-source version of Swift is similar from a runtime perspective on both Linux and OS X; however, the set of libraries available differ between the two platforms.

For example, the Objective-C runtime was not present in the initial release of Swift for Linux; as a result, several methods that are delegated to Objective-C implementations are not available. *"hello".hasPrefix("he")* compiles and runs successfully on OS X and iOS but is a compile error in the first Swift release for Linux. In addition to missing functions, there is also a different set of modules (frameworks) between the two platforms. The base functionality on OS X and iOS is provided by the *Darwin* module, but on Linux, the base functionality is provided by the *Glibc* module. The *Foundation* module, which provides many of the data types that are outside of the base-collections library, is implemented in Objective-C on OS X and iOS, but on Linux, it is a

clean-room reimplementation in Swift. As Swift on Linux evolves, more of this functionality will be filled in, but it is worth testing on both OS X and Linux specifically if cross platform functionality is required.

Finally, although the Swift language and core libraries have been open sourced, this does not apply to the iOS libraries or other functionality in Xcode. As a result, it is not possible to compile iOS or OS X applications from Linux, and building iOS applications and editing user interfaces is something that must be done in Xcode on OS X.

Getting started with Swift

Swift provides a runtime interpreter that executes statements and expressions. Swift is open source, and precompiled binaries can be downloaded from https://swift.org/download/ for both OS X and Linux platforms. Ports are in progress to other platforms and operating systems but are not supported by the Swift development team.

The Swift interpreter is called *swift* and on OS X can be launched using the *xcrun* command in a *Terminal.app* shell:

```
$ xcrun swift

Welcome to Swift version 2.2!  Type :help for assistance.

>
```

The *xcrun* command allows a toolchain command to be executed; in this case, it finds */Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/swift* . The *swift* command sits alongside other compilation tools, such as *clang* and *ld*, and permits multiple versions of the commands and libraries on the same machine without conflicting.

On Linux, the *swift* binary can be executed provided that it and the dependent libraries are in a suitable location.

The Swift prompt displays > for new statements and . for a continuation. Statements and expressions that are typed into the interpreter are evaluated and displayed. Anonymous values are given references so that they can be used subsequently:

```
> "Hello World"

$R0: String = "Hello World"

> 3 + 4

$R1: Int = 7

> $R0

$R2: String = "Hello World"
```

```
> $R1

$R3: Int = 7
```

Numeric literals

Numeric types in Swift can represent both signed and unsigned integral values with sizes of 8, 16, 32, or 64 bits, as well as signed 32 or 64 bit floating point values. Numbers can include underscores to provide better readability; so, 68_040 is the same as 68040:

```
> 3.141

$R0: Double = 3.141

> 299_792_458

$R1: Int = 299792458

> -1

$R2: Int = -1

> 1_800_123456

$R3: Int = 1800123456
```

Numbers can also be written in **binary**, **octal**, or **hexadecimal** using prefixes *0b*, *0o* (zero and the letter "o") or *0x*. Please note that Swift does not inherit C's use of a leading zero (*0*) to represent an octal value, unlike Java and JavaScript which do. Examples include:

```
> 0b1010011

$R0: Int = 83

> 0o123

$R1: Int = 83

> 0123

$R2: Int = 123

> 0x7b

$R3: Int = 123
```

Floating point literals

There are three floating point types that are available in Swift which use the IEEE754 floating point standard. The *Double* type represents 64 bits worth of data, while *Float* stores 32 bits of data. In addition, *Float80* is a specialized type that stores 80 bits worth of data (*Float32* and *Float64* are available as aliases for *Float* and *Double*, respectively, although they are not commonly used in Swift programs).

Some CPUs internally use 80 bit precision to perform math operations, and the *Float80* type allows this accuracy to be used in Swift. Not all architectures support *Float80* natively, so this should be used sparingly.

By default, floating point values in Swift use the *Double* type. As floating point representation cannot represent some numbers exactly, some values will be displayed with a rounding error; for example:

```
> 3.141

$R0: Double = 3.141

> Float(3.141)

$R1: Float = 3.1400003
```

Floating point values can be specified in decimal or hexadecimal. Decimal floating point uses *e* as the exponent for base 10, whereas hexadecimal floating point uses *p* as the exponent for base 2. A value of *AeB* has the value *A*10^B* and a value of *0xApB* has the value *A*2^B*. For example:

```
> 299.792458e6

$R0: Double = 299792458

> 299.792_458_e6

$R1: Double = 299792458

> 0x1p8

$R2: Double = 256

> 0x1p10

$R3: Double = 1024

> 0x4p10

$R4: Double = 4096

> 1e-1

$R5: Double = 0.10000000000000001

> 1e-2
```

```
$R6: Double = 0.01> 0x1p-1

$R7: Double = 0.5

> 0x1p-2

$R8: Double = 0.25

> 0xAp-1

$R9: Double = 5
```

String literals

Strings can contain escaped characters, Unicode characters, and interpolated expressions. Escaped characters start with a slash (\) and can be one of the following:

- \\: This is a literal slash \
- *0*: This is the null character
- *'*: This is a literal single quote '
- \": This is a literal double quote "
- *t*: This is a tab
- *n*: This is a line feed
- *r*: This is a carriage return
- *u{NNN}*: This is a Unicode character, such as the Euro symbol *u{20AC}*, or a smiley *u{1F600}*

An *interpolated string* has an embedded expression, which is evaluated, converted into a *String*, and inserted into the result:

```
> "3+4 is \(3+4)"

$R0: String = "3+4 is 7"

> 3+4

$R1: Int = 7

> "7 x 2 is \($R1 * 2)"

$R2: String = "7 x 2 is 14"
```

Variables and constants

Swift distinguishes between variables (which can be modified) and constants (which cannot be changed after assignment). Identifiers start with an underscore or alphabetic character followed by an underscore or alphanumeric character. In addition, other Unicode character points (such as emoji) can be used although box lines and arrows are not allowed; consult the Swift language

guide for the full set of allowable Unicode characters. Generally, Unicode private use areas are not allowed, and identifiers cannot start with a combining character (such as an accent).

Variables are defined with the *var* keyword, and constants are defined with the *let* keyword. If the type is not specified, it is automatically inferred:

```
> let pi = 3.141

pi: Double = 3.141

> pi = 3


error: cannot assign to value: 'pi' is a 'let' constant

note: change 'let' to 'var' to make it mutable

> var i = 0

i: Int = 0

> ++i

$R0: Int = 1
```

Types can be explicitly specified. For example, to store a 32 bit floating point value, the variable can be explicitly defined as a *Float*:

```
> let e:Float = 2.718

e: Float = 2.71799994
```

Similarly, to store a value as an unsigned 8 bit integer, explicitly declare the type as *UInt8*:

```
> let ff:UInt8 = 255

ff: UInt8 = 255
```

A number can be converted to a different type using the type initializer or a literal that is assigned to a variable of a different type, provided that it does not underflow or overflow:

```
> let ooff = UInt16(ff)

ooff: UInt16 = 255

> Int8(255)

error: integer overflows when converted from 'Int' to 'Int8'

Int8(255)

^
```

```
> UInt8(Int8(-1))

error: negative integer cannot be converted to unsigned type 'UInt8'

UInt8(Int8(-1))

^
```

Collection types

Swift has three collection types: *Array*, *Dictionary*, and *Set*. They are strongly typed and generic, which ensures that the values of types that are assigned are compatible with the element type. Collections that are defined with *var* are mutable; collections defined with *let* are immutable.

The literal syntax for arrays uses *[]* to store a comma-separated list:

```
> var shopping = [ "Milk", "Eggs", "Coffee", ]

shopping: [String] = 3 values {

  [0] = "Milk"

  [1] = "Eggs"

  [2] = "Coffee"

}
```

Literal dictionaries are defined with a comma-separated *[key:value]* format for entries:

```
> var costs = [ "Milk":1, "Eggs":2, "Coffee":3, ]

costs: [String : Int] = {

  [0] = { key = "Coffee" value = 3 }

  [1] = { key = "Milk"   value = 1 }

  [2] = { key = "Eggs"   value = 2 }

}
```

For readability, array and dictionary literals can have a trailing comma. This allows initialization to be split over multiple lines, and if the last element ends with a trailing comma, adding new items does not result in an SCM diff to the previous line.

Arrays and dictionaries can be indexed using subscript operators that are reassigned and added to:

```
> shopping[0]
```

```
$R0: String = "Milk"

> costs["Milk"]

$R1: Int? = 1

> shopping.count

$R2: Int = 3

> shopping += ["Tea"]

> shopping.count

$R3: Int = 4

> costs.count

$R4: Int = 3

> costs["Tea"] = "String"

error: cannot assign a value of type 'String' to a value of type 'Int?'

> costs["Tea"] = 4

> costs.count

$R5: Int = 4
```

Sets are similar to dictionaries; the keys are unordered and can be looked up efficiently. However, unlike dictionaries, keys don't have an associated value. As a result, they don't have array subscripts, but they do have the *insert*, *remove*, and *contains* methods. They also have efficient set intersection methods, such as *union* and *intersect*. They can be created from an array literal if the type is defined or using the set initializer directly:

```
> var shoppingSet: Set = [ "Milk", "Eggs", "Coffee", ]

> // same as: shoppingSet = Set( [ "Milk", "Eggs", "Coffee", ] )

> shoppingSet.contains("Milk")

$R6: Bool = true

> shoppingSet.contains("Tea")

$R7: Bool = false

> shoppingSet.remove("Coffee")

$R8: String? = "Coffee"

> shoppingSet.remove("Tea")

$R9: String? = nil
```

```
> shoppingSet.insert("Tea")

> shoppingSet.contains("Tea")

$R10: Bool = true
```

When creating sets, use the explicit *Set* constructor as otherwise the type will be inferred to be an *Array*, which will have a different performance profile.

Optional types

In the previous example, the return type of *costs["Milk"]* is *Int?* and not *Int*. This is an *optional type*; there may be an *Int* value or it may be empty. For a dictionary containing elements of type *T*, subscripting the dictionary will have an *Optional<T>* type, which can be abbreviated as *T?* If the value doesn't exist in the dictionary, then the returned value will be *nil*. Other object-oriented languages, such as Objective-C, C++, Java, and C#, have optional types by default; any object value (or pointer) can be *null*. By representing optionality in the type system, Swift can determine whether a value really has to exist or might be *nil*:

```
> var cannotBeNil: Int = 1

cannotBeNil: Int = 1

> cannotBeNil = nil

error: cannot assign a value of type 'nil' to a value of type 'Int'

cannotBeNil = nil
              ^

> var canBeNil: Int? = 1

canBeNil: Int? = 1

> canBeNil = nil

$R0: Int? = nil
```

Optional types can be explicitly created using the *Optional* constructor. Given a value *x* of type *X*, an optional *X?* value can be created using *Optional(x)*. The value can be tested against *nil* to find out whether it contains a value and then unwrapped with *opt!*, for example:

```
> var opt = Optional(1)

opt: Int? = 1

> opt == nil

$R1: Bool = false
```

```
> opt!

$R2: Int = 1
```

If a *nil* value is unwrapped, an error occurs:

```
> opt = nil

> opt!

fatal error: unexpectedly found nil while unwrapping an Optional value

Execution interrupted. Enter Swift code to recover and continue.

Enter LLDB commands to investigate (type :help for assistance.)
```

Particularly when working with Objective-C based APIs, it is common for values to be declared as an optional although they are always expected to return a value. It is possible to declare such variables as *implicitly unwrapped optionals*; these variables behave as optional values (they may contain *nil*), but when the value is accessed, they are automatically unwrapped on demand:

```
> var implicitlyUnwrappedOptional:Int! = 1

implicitlyUnwrappedOptional: Int! = 1

> implicitlyUnwrappedOptional + 2

3

> implicitlyUnwrappedOptional = nil

> implicitlyUnwrappedOptional + 2

fatal error: unexpectedly found nil while unwrapping an Optional value
```

In general, implicitly unwrapped optionals should be avoided as they are likely to lead to errors. They are mainly useful for interaction with existing Objective-C APIs when the value is known to have an instance.

Nil coalescing operator

Swift has a *nil coalescing operator*, which is similar to Groovy's *?:* operator or C#'s *??* operator. This provides a means to specify a default value if an expression is *nil*:

```
> 1 ?? 2

$R0: Int = 1

> nil ?? 2

$R1: Int = 2
```

The *nil* coalescing operator can also be used to unwrap an optional value. If the optional value is present, it is unwrapped and returned; if it is missing, then the right-hand side of the expression is returned. Similar to the || shortcut, and the && operators, the right-hand side is not evaluated unless necessary:

```
> costs["Tea"] ?? 0

$R2: Int = 4

> costs["Sugar"] ?? 0

$R3: Int = 0
```

Conditional logic

There are three key types of conditional logic in Swift (known as branch statements in the grammar): the *if* statement, the *switch* statement, and the *guard* statement. Unlike other languages, the body of the *if* must be surrounded with braces *{}*; and if typed in at the interpreter, the *{* opening brace must be on the same line as the *if* statement. The *guard* statement is a specialized *if* statement for use with functions.

If statements

Conditionally unwrapping an optional value is so common that a specific Swift pattern *optional binding* has been created to avoid evaluating the expression twice:

```
> var shopping = [ "Milk", "Eggs", "Coffee", "Tea", ]

> var costs = [ "Milk":1, "Eggs":2, "Coffee":3, "Tea":4, ]

> var cost = 0

> if let cc = costs["Coffee"] {

.   cost += cc

. }

> cost

$R0: Int = 3
```

The *if* block only executes if the optional value exists. The definition of the *cc* constant only exists for the body of the *if* block, and it does not exist outside of that scope. Furthermore, *cc* is a non-optional type, so it is guaranteed not to be *nil*.

Swift 1 only allowed a single *let* assignment in an *if* block causing a pyramid of nested *if* statements. Swift 2 allows multiple comma-separated *let* assignments in a single *if* statement.

```
> if let cm = costs["Milk"], let ct = costs["Tea"] {

.   cost += cm + ct

. }

> cost

$R1: Int = 8
```

To execute an alternative block if the item cannot be found, an *else* block can be used:

```
> if let cb = costs["Bread"] {

.   cost += cb

. } else {

.   print("Cannot find any Bread")

. }

Cannot find any Bread
```

Other boolean expressions can include the *true* and *false* literals, and any expression that conforms to the *BooleanType* protocol, the == and *!*= equality operators, the === and *!*== identity operators, as well as the <, <=, >, and >= comparison operators. The *is type* operator provides a test to see whether an element is of a particular type.

The difference between the equality operator and the identity operator is relevant for classes or other reference types. The equality operator asks *Are these two values equivalent to each other?*, whereas the identity operator asks *Are these two references equal to each other?*

There is a boolean operator that is specific to Swift, which is the ~= *pattern match operator*. Despite the name, this isn't anything to do with regular expressions; rather, it's a way of asking whether a pattern matches a particular value. This is used in the implementation of the *switch* block.

As well as the *if* statement, there is a *ternary if expression* that is similar to other languages. After a condition, a question mark (?) is used followed by an expression to be used if the condition is true, then a colon (:) followed by the false expression:

```
> var i = 17

i: Int = 17

> i % 2 == 0 ? "Even" : "Odd"

$R0: String = "Odd"
```

Switch statements

Swift has a *switch* statement that is similar to C and Java's *switch*. However, it differs in two important ways. Firstly, *case* statements no longer have a default fall-through behavior (so there are no bugs introduced by missing a *break* statement), and secondly, the value of the *case* statements can be expressions instead of values, pattern matching on type and range. At the end of the corresponding *case* statement, the evaluation jumps to the end of the *switch* block unless the *fallthrough* keyword is used. If no *case* statements match, the *default* statements are executed.

A *default* statement is required when the list of cases is not exhaustive. If they are not, the compiler will give an error saying that the list is not exhaustive and that a *default* statement is required.

```
> var position = 21

position: Int = 21

> switch position {

.   case 1: print("First")

.   case 2: print("Second")

.   case 3: print("Third")

.   case 4...20: print("\(position)th")

.   case position where (position % 10) == 1:

.     print("\(position)st")

.   case let p where (p % 10) == 2:

.     print("\(p)nd")

.   case let p where (p % 10) == 3:

.     print("\(p)rd")

.   default: print("\(position)th")

. }

21st
```

In the preceding example, the expression prints out *First*, *Second*, or *Third* if the position is *1*, *2*, or *3*, respectively. For numbers between *4* and *20* (inclusive), it prints out the position with a *th* ordinal. Otherwise, for numbers that end with 1, it prints *st*; for numbers that end with 2, it prints *nd*, and for numbers that end with 3, it prints *rd*. For all other numbers it prints *th*.

The *4...20* range expression in a *case* statement represents a pattern. If the value of the expression matches that pattern, then the corresponding statements will be executed:

```
> 4...10 ~= 4

$R0: Bool = true

> 4...10 ~= 21

$R1: Bool = false
```

There are two range operators in Swift: an inclusive or *closed range*, and an exclusive or *half-open range*. The closed range is specified with three dots; so *1...12* will give a list of integers between one and twelve. The half-open range is specified with two dots and a less than operator; so *1..<10* will provide integers from 1 to 9 but excluding 10.

The *where* clause in the *switch* block allows an arbitrary expression to be evaluated provided that the pattern matches. These are evaluated in order, in the sequence they are in the source file. If a *where* clause evaluates to *true*, then the corresponding set of statements will be executed.

The *let* variable syntax can be used to define a constant that refers to the value in the *switch* block. This local constant can be used in the *where* clause or the corresponding statements for that specific case. Alternatively, variables can be used from the surrounding scope.

If multiple *case* statements need to match the same pattern, they can be separated with commas as an expression list. Alternatively, the *fallthrough* keyword can be used to allow the same implementation to be used for multiple *case* statements.

Iteration

Ranges can be used to iterate a fixed number of times, for example, *for i in 1...12*. To print out these numbers, a loop such as the following can be used:

```
> for i in 1...12 {

.   print("i is \(i)")

. }
```

If the number is not required, then an underscore (_) can be used as a hole to act as a throwaway value. An underscore can be assigned to but not read:

```
> for _ in 1...12 {

.   print("Looping...")

. }
```

However, it is more common to iterate over a collection's contents using a *for in* pattern. This steps through each of the items in the collection, and the body of the *for* loop is executed over each one:

```
> var shopping = [ "Milk", "Eggs", "Coffee", "Tea", ]
> var costs = [ "Milk":1, "Eggs":2, "Coffee":3, "Tea":4, ]
> var cost = 0
> for item in shopping {
.   if let itemCost = costs[item] {
.     cost += itemCost
.   }
. }
> cost
cost: Int = 10
```

To iterate over a dictionary, it is possible to extract the keys or the values and process them as an array:

```
> Array(costs.keys)
$R0: [String] = 4 values {
  [0] = "Coffee"
  [1] = "Milk"
  [2] = "Eggs"
  [3] = "Tea"
}
> Array(costs.values)
$R1: [Int] = 4 values {
  [0] = 3
  [1] = 1
  [2] = 2
  [3] = 4
}
```

The order of keys in a dictionary is not guaranteed; as the dictionary changes, the order may change.

Converting a dictionary's values to an array will result in a copy of the data being made, which can lead to poor performance. As the underlying *keys* and *values* are of a *LazyMapCollection* type, they can be iterated over directly:

```
> costs.keys

$R2: LazyMapCollection<[String : Int], String> = {

  _base = {

    _base = 4 key/value pairs {

      [0] = { key = "Coffee" value = 3 }

      [1] = { key = "Milk"   value = 1 }

      [2] = { key = "Eggs"   value = 2 }

      [3] = { key = "Tea"    value = 4 }

    }

  _transform =  }

}
```

To print out all the keys in a dictionary, the keys property can be used with a for in loop:

```
> for item in costs.keys {

.   print(item)

. }

Coffee

Milk

Eggs

Tea
```

Iterating over keys and values in a dictionary

Traversing a dictionary to obtain all of the keys and then subsequently looking up values will result in searching the data structure twice. Instead, both the key and the value can be iterated at the same time, using a *tuple*. A tuple is like a fixed-sized array, but one that allows assigning pairs (or more) of values at a time:

```
> var (a,b) = (1,2)

a: Int = 1
```

```
b: Int = 2
```

Tuples can be used to iterate pairwise over both the keys and values of a dictionary:

```
> for (item,cost) in costs {

.   print("The \(item) costs \(cost)")

. }

The Coffee costs 3

The Milk costs 1

The Eggs costs 2

The Tea costs 4
```

Both *Array* and *Dictionary* conform to the *SequenceType* protocol, which allows them to be iterated with a *for in* loop. Collections (as well as other objects, such as *Range*) that implement *SequenceType* have a *generate* method, which returns a *GeneratorType* that allows the data to be iterated over. It is possible for custom Swift objects to implement *SequenceType* to allow them to be used in a *for in* loop.

Iteration with for loops

Although the most common use of the *for* operator in Swift is in a *for in* loop, it is also possible (in Swift 1 and 2) to use a more traditional form of the *for* loop. This has an initialization, a condition that is tested at the start of each loop, and a step operation that is evaluated at the end of each loop. Although the parentheses around the *for* loop are optional, the braces for the block of code are mandatory.

It has been proposed that both the traditional *for* loop and the increment/decrement operators should be removed from Swift 3. It is recommended that these forms of loops be avoided where possible.

Calculating the sum of integers between 1 and 10 can be performed without using the range operator:

```
> var sum = 0

. for var i=0; i<=10; ++i {

.   sum += i

. }

sum: Int = 55
```

If multiple variables need to be updated in the *for* loop, Swift has an *expression list* that is a set of comma-separated expressions. To step through two sets of variables in a for loop, the following can be used:

```
> for var i = 0,j = 10; i<=10 && j >= 0; ++i,--j {
.   print("\(i), \(j)")
. }
0, 10
1, 9
…
9, 1
10, 0
```

Apple recommends the use of *++i* instead of *i++* (and conversely, *--i* instead of *i--*) because they will return the result of *i* after the operation, which may be the expected value. As noted earlier, these operators may be removed in a future version of Swift.

Break and continue

The *break* statement leaves the innermost loop early, and control jumps to the end of the loop. The *continue* statement takes execution to the top of the innermost loop and the next item.

To *break* or *continue* from nested loops, a *label* can be used. Labels in Swift can only be applied to a loop statement, such as *while* or *for*. A label is introduced by an identifier and a colon just before the loop statement:

```
> var deck = [1...13, 1...13, 1...13, 1...13]
> suits: for suit in deck {
.   for card in suit {
.     if card == 3 {
.       continue // go to next card in same suit
.     }
.     if card == 5 {
.       continue suits // go to next suit
.     }
.     if card == 7 {
```

```
.       break // leave card loop
.     }
.     if card == 13 {
.       break suits // leave suit loop
.     }
.   }
. }
```

Functions

Functions can be created using the *func* keyword, which takes a set of arguments and a body of statements. The *return* statement can be used to leave a function:

```
> var shopping = [ "Milk", "Eggs", "Coffee", "Tea", ]
> var costs = [ "Milk":1, "Eggs":2, "Coffee":3, "Tea":4, ]
> func costOf(items:[String], _ costs:[String:Int]) -> Int {
.   var cost = 0
.   for item in items {
.     if let ci = costs[item] {
.       cost += ci
.     }
.   }
.   return cost
. }
> costOf(shopping,costs)
$R0: Int = 10
```

The return type of the function is specified after the arguments with an arrow (->). If missing, the function cannot return a value; if present, the function must return a value of that type.

The underscore (_) on the front of the *costs* parameter is required to avoid it being a named argument. The second and subsequent arguments in Swift functions are implicitly named. To ensure that it is treated as a positional argument, the _ before the argument name is required.

Functions with *positional arguments* can be called with parentheses, such as the *costOf(shopping,costs)* call. If a function takes no arguments, then the parentheses are still required.

The *foo()* expression calls the *foo* function with no argument. The *foo* expression represents the function itself, so an expression, such as *let copyOfFoo = foo*, results in a copy of the function; as a result, *copyOfFoo()* and *foo()* have the same effect.

Named arguments

Swift also supports *named arguments*, which can either use the name of the variable or can be defined with an *external parameter name*. To modify the function to support calling with *basket* and *prices* as argument names, the following can be done:

```
> func costOf(basket items:[String], prices costs:[String:Int]) -> Int {

.   var cost = 0

.   for item in items {

.     if let ci = costs[item] {

.       cost += ci

.     }

.   }

.   return cost

. }

> costOf(basket:shopping, prices:costs)

$R1: Int = 10
```

This example defines external parameter names *basket* and *prices* for the function. The function signature is often referred to as *costOf(basket:prices:)* and is useful when it may not be clear what the arguments are for (particularly if they are of the same type).

Optional arguments and default values

Swift functions can have *optional arguments* by specifying *default values* in the function definition. When the function is called, if an optional argument is missing, the default value for that argument is used.

An optional argument is one that can be omitted in the function call rather than a required argument that takes an optional value. This naming is unfortunate. It may help to think of these as default arguments rather than optional arguments.

A default parameter value is specified after the type in the function signature, with an equals (=) and then the expression. This expression is re-evaluated each time the function is called without a corresponding argument.

In the *costOf* example, instead of passing the value of *costs* each time, it could be defined with a default parameter:

```
> func costOf(items items:[String], costs:[String:Int] = costs) -> Int {
.   var cost = 0
.   for item in items {
.     if let ci = costs[item] {
.       cost += ci
.     }
.   }
.   return cost
. }
> costOf(items:shopping)
$R2: Int = 10
> costOf(items:shopping, costs:costs)
$R3: Int = 10
```

Please note that the captured *costs* variable is bound when the function is defined.

To use a named argument as the first parameter in a function, the argument name has to be duplicated. Swift 1 used a hash (#) to represent an implicit parameter name, but this was removed from Swift 2.

Guards

It is a common code pattern for a function to require arguments that meet certain conditions before the function can run successfully. For example, an optional value must have a value or an integer argument must be in a certain range.

Typically, the pattern to implement this is either to have a number of *if* statements that break out of the function at the top, or to have an *if* block wrapping the entire method body:

```
if card < 1 || card > 13 {
  // report error
```

```
  return
}


// or alternatively:


if card >= 1 && card <= 13 {
  // do something with card
} else {
  // report error
}
```

Both of these approaches have drawbacks. In the first case, the condition has been negated; instead of looking for valid values, it's checking for invalid values. This can cause subtle bugs to creep in; for example, *card < 1 && card > 13* would never succeed, but it may inadvertently pass a code review. There's also the problem of what happens if the block doesn't *return* or *break*; it could be perfectly valid Swift code but still include errors.

In the second case, the main body of the function is indented at least one level in the body of the *if* statement. When multiple conditions are required, there may be many nested *if* statements, each with their own error handling or cleanup requirements. If new conditions are required, then the body of the code may be indented even further, leading to code churn in the repository even when only whitespace has changed.

Swift 2 adds a *guard* statement, which is conceptually identical to an *if* statement, except that it only has an *else* clause body. In addition, the compiler checks that the *else* block returns from the function, either by returning or by throwing an exception:

```
> func cardName(value:Int) -> String {
.   guard value >= 1 && value <= 13 else {
.     return "Unknown card"
.   }
.   let cardNames = [11:"Jack",12:"Queen",13:"King",1:"Ace",]
.   return cardNames[value] ?? "\(value)"
. }
```

The Swift compiler checks that the *guardelse* block leaves the function, and reports a compile error if it does not. Code that appears after the *guard* statement can guarantee that the value is in the *1...13* range without having to perform further tests.

The *guard* block can also be used to perform *optional binding*; if the *guard* condition is a *let* assignment that performs an optional test, then the code that is subsequent to the *guard* statement can use the value without further unwrapping:

```
> func firstElement(list:[Int]) -> String {

.   guard let first = list.first else {

.     return "List is empty"

.   }

.   return "Value is \(first)"

. }
```

As the *first* element of an array is an optional value, the *guard* test here acquires the value and unwraps it. When it is used later in the function, the unwrapped value is available for use without requiring further unwrapping.

Multiple return values and arguments

So far, the examples of functions have all returned a single type. What happens if there is more than one return result from a function? In an object-oriented language, the answer is to return a class; however, Swift has tuples, which can be used to return multiple values. The type of a tuple is the type of its constituent parts:

```
> var pair = (1,2)

pair: (Int, Int) ...
```

This can be used to return multiple values from the function; instead of just returning one value, it is possible to return a tuple of values.

Swift also has in-out arguments.

Separately, it is also possible to take a variable number of arguments. A function can easily take an array of values with *[]*, but Swift provides a mechanism to allow calling with multiple arguments, using a *variadic* parameter, which is denoted as an ellipses (…) after the type. The value can then be used as an array in the function.

Swift 1 only allowed the *variadic* argument as the last argument; Swift 2 relaxed that restriction to allow a single *variadic* argument to appear anywhere in the function's parameters.

Taken together, these two features allow the creation of a *minmax* function, which returns both the minimum and maximum from a list of integers:

```
> func minmax(numbers:Int…) -> (Int,Int) {
.   var min = Int.max
.   var max = Int.min
.   for number in numbers {
.     if number < min {
.       min = number
.     }
.     if number > max {
.       max = number
.     }
.   }
.   return (min,max)
. }
> minmax(1,2,3,4)
$R0: (Int, Int) = {
  0 = 1
  1 = 4
}
```

The *numbers:Int…* argument indicates that a variable number of arguments can be passed into the function. Inside the function, it is processed as an ordinary array; in this case, iterating through using a *for in* loop.

Int.max is a constant representing the largest *Int* value, and *Int.min* is a constant representing the smallest *Int* value. Similar constants exist for other integral types, such as *UInt8.max*, and *Int64.min*.

What if no arguments are passed in? If run on a 64 bit system, then the output will be:

```
> minmax()
$R1: (Int, Int) = {
  0 = 9223372036854775807
```

```
  1 = -9223372036854775808
}
```

This may not make sense for a *minmax* function. Instead of returning an error value or a default value, the type system can be used. By making the tuple optional, it is possible to return a *nil* value if it doesn't exist, or a tuple if it does:

```
> func minmax(numbers:Int...) -> (Int,Int)? {
.   var min = Int.max
.   var max = Int.min
.   if numbers.count == 0 {
.     return nil
.   } else {
.     for number in numbers {
.       if number < min {
.         min = number
.       }
.       if number > max {
.         max = number
.       }
.     }
.     return(min,max)
.   }
. }
> minmax()
$R2: (Int, Int)? = nil
> minmax(1,2,3,4)
$R3: (Int, Int)? = (0 = 1, 1 = 4)
> var (minimum,maximum) = minmax(1,2,3,4)!
minimum: Int = 1
maximum: Int = 4
```

Returning an optional value allows the caller to determine what should happen in cases where the maximum and minimum are not present.

If a function does not always have a valid return value, use an optional type to encode that possibility into the type system.

Returning structured values

A tuple is an ordered set of data. The entries in the tuple are ordered, but it can quickly become unclear as to what data is stored, particularly if they are of the same type. In the *minmax* tuple, it is not clear which value is the minimum and which value is the maximum, and this can lead to subtle programming errors later on.

A structure (*struct*) is like a tuple but with named values. This allows members to be accessed by name instead of by position, leading to fewer errors and greater transparency. Named values can be added to tuples as well; in essence, tuples with named values are anonymous structures.

Structs are passed in a copy-by-value manner like tuples. If two variables are assigned the same struct or tuple, then changes to one do not affect the values of another.

A *struct* is defined with the *struct* keyword and has variables or values in the body:

```
> struct MinMax {
.   var min:Int
.   var max:Int
. }
```

This defines a *MinMax* type, which can be used in place of any of the types that are seen so far. It can be used in the *minmax* function to return a *struct* instead of a tuple:

```
> func minmax(numbers:Int...) -> MinMax? {
.   var minmax = MinMax(min:Int.max, max:Int.min)
.   if numbers.count == 0 {
.     return nil
.   } else {
.     for number in numbers {
.       if number < minmax.min {
.         minmax.min = number
.       }
```

```
.        if number > minmax.max {
.          minmax.max = number
.        }
.      }
.      return minmax
.    }
. }
```

The *struct* is initialized with a type initializer; if *MinMax()* is used, then the default values for each of the structure types are given (based on the structure definition), but these can be overridden explicitly if desired with *MinMax(min:-10,max:11)*. For example, if the *MinMax* struct is defined as *struct MinMax { var min:Int = Int.max; var max:Int = Int.min }*, then *MinMax()* will return a structure with the appropriate minimum and maximum values filled in.

When a structure is initialized, all the non-optional fields must be assigned. They can be passed in as named arguments in the initializer or specified in the structure definition.

Error handling

In the original Swift release, error handling consisted of either returning a *Bool* or an optional value from function results. This tended to work inconsistently with Objective-C, which used an optional *NSError* pointer on various calls that was set if a condition had occurred.

Swift 2 adds an exception-like error model, which allows code to be written in a more compact way while ensuring that errors are handled accordingly. Although this isn't implemented in quite the same way as C++ exception handling, the semantics of the error handling are quite similar.

Errors can be created using a new *throw* keyword, and errors are stored as a subtype of *ErrorType*. Although swift *enum* values are often used as error types, *struct* values can be used as well.

Exception types can be created as subtypes of *ErrorType* by appending the supertype after the type name:

```
> struct Oops:ErrorType {
.   let message:String
. }
```

Exceptions are thrown using the *throw* keyword and creating an instance of the exception type:

```
> throw Oops(message:"Something went wrong")

$E0: Oops = {

  message = "Something went wrong"

}
```

The REPL displays exception results with the *$E* prefix; ordinary results are displayed with the *$R* prefix.

Throwing errors

Functions can declare that they return an error using the *throws* keyword before the return type, if any. The previous *cardName* function, which returned a dummy value if the argument was out of range, can be upgraded to throw an exception instead by adding the *throws* keyword before the return type and changing the *return* to a *throw*:

```
> func cardName(value:Int) throws -> String {

.   guard value >= 1 && value <= 13 else {

.     throw Oops(message:"Unknown card")

.   }

.   let cardNames = [11:"Jack",12:"Queen",13:"King",1:"Ace",]

.   return cardNames[value] ?? "\(value)"

. }
```

When the function is called with a real value, the result is returned; when it is passed an invalid value, an exception is thrown instead:

```
> cardName(1)

$R1: String = "Ace"

> cardName(15)

$E2: Oops = {

  message = "Unknown card"

}
```

When interfacing with Objective-C code, methods that take an *NSError*** argument are automatically represented in Swift as methods that throw. In general, any method whose arguments ends in *NSError*** is treated as throwing an exception in Swift.

Exception throwing in C++ and Objective-C is not as performant as exception handling in Swift because the latter does not perform stack unwinding. As a result, exception throwing in Swift is equivalent (from a performance perspective) to dealing with return values. Expect the Swift library to evolve in the future towards a throws-based means of error detection and away from Objective-C's use of ***NSError* pointers.

Catching errors

The other half of exception handling is the ability to catch errors when they occur. As with other languages, Swift now has a *try/catch* block that can be used to handle error conditions. Unlike other languages, the syntax is a little different; instead of a *try/catch* block, there is a *do/catch* block, and each expression that may throw an error is annotated with its own *try* statement:

```
> do {

.   let name = try cardName(15)

.   print("You chose \(name)")

. } catch {

.   print("You chose an invalid card")

. }
```

When the preceding code is executed, it will print out the generic error message. If a different choice is given, then it will run the successful path instead.

It's possible to capture the error object and use it in the catch block:

```
. } catch let e {

.   print("There was a problem \(e)")

. }
```

The default *catch* block will bind to a variable called *error* if not specified

Both of these two preceding examples will catch any errors thrown from the body of the code.

It's possible to catch explicitly based on type if the type is an *enum* that is using pattern matching, for example, *catch Oops(let message)*. However, as this does not work for struct values.

Sometimes code will always work, and there is no way it can fail. In these cases, it's cumbersome to have to wrap the code with a *do/try/catch* block when it is known that the problem can never occur. Swift provides a short-cut for this using the *try!* statement, which catches and filters the exception:

```
> let ace = try! cardName(1)

ace: String = "Ace"
```

If the expression really does fail, then it translates to a runtime error and halts the program:

```
> let unknown = try! cardName(15)



Fatal error: 'try!' expression unexpectedly raised an error: Oops(message:
"Unknown card")
```

Using *try!* is not generally recommended; if an error occurs then the program will crash. However, it is often used with user interface codes as Objective-C has a number of optional methods and values that are conventionally known not to be *nil*, such as the reference to the enclosing window.

A better approach is to use *try?*, which translates the expression into an optional value: if evaluation succeeds, then it returns an optional with a value; if evaluation fails, then it returns a *nil* value:

```
> let ace = try? cardName(1)

ace: String? = "Ace"

> let unknown = try? cardName(15)

unknown: String? = nil
```

This is handy for use in the *if let* or *guard let* constructs, to avoid having to wrap in a *do/catch* block:

```
> if let card = try? cardName(value) {

.   print("You chose: \(card)")

. }
```

Cleaning up after errors

It is common to have a function that needs to perform some cleanup before the function returns, regardless of whether the function has completed successfully or not. An example would be working with files; at the start of the function the file may be opened, and by the end of the function it should be closed again, whether or not an error occurs.

A traditional way of handling this is to use an optional value to hold the file reference, and at the end of the method if it is not *nil*, then the file is closed. However, if there is the possibility of an error occurring during the method's execution, there needs to be a *do/catch* block to ensure that

the cleanup is correctly called, or a set of nested *if* statements that are only executed if the file is successful.

The downside with this approach is that the actual body of the code tends to be indented several times each with different levels of error handling and recovery at the end of the method. The syntactic separation between where the resource is acquired and where the resource is cleaned up can lead to bugs.

Swift has a *defer* statement, which can be used to register a block of code to be run at the end of the function call. This block is run regardless of whether the function returns normally (with the *return* statement) or if an error occurs (with the *throw* statement). Deferred blocks are executed in reverse order of execution, for example:

```
> func deferExample() {
.   defer { print("C") }
.   print("A")
.   defer { print("B") }
. }
> deferExample()
A
B
C
```

Please note that if a *defer* statement is not executed, then the block is not executed at the end of the method. This allows a *guard* statement to leave the function early, while executing the *defer* statements that have been added so far:

```
> func deferEarly() {
.   defer { print("C") }
.   print("A")
.   return
.   defer { print("B") } // not executed
. }
> deferEarly()
A
C
```

Command-line Swift

As Swift can be interpreted, it is possible to use it in shell scripts. By setting the interpreter to *swift* with a *hashbang*, the script can be executed without requiring a separate compilation step. Alternatively, Swift scripts can be compiled to a native executable that can be run without the overhead of the interpreter.

Interpreted Swift scripts

Save the following as *hello.swift*:

```
#!/usr/bin/env xcrun swift

print("Hello World")
```

In Linux, the first line should point to the location of the *swift* executable, such as *#!/usr/bin/swift*.

After saving, make the file executable by running *chmod a+x hello.swift*. The program can then be run by typing *./hello.swift*, and the traditional greeting will be seen:

```
Hello World
```

Arguments can be passed from the command line and interrogated in the process using the *Process* class through the *arguments* constant. As with other Unix commands, the first element (0) is the name of the process executable; the arguments that are passed from the command line start from one (1).

The program can be terminated using the *exit* function; however, this is defined in the operating system libraries and so it needs to be imported in order to call this function. Modules in Swift correspond to Frameworks in Objective-C and give access to all functions that are defined as public API in the module. The syntax to import all elements from a module is *import module* although it's also possible to import a single function using *import func module.functionName*.

Not all foundation libraries are implemented for Linux, which results in some differences of behavior. In addition, the underlying module for the base functionality is *Darwin* on iOS and OS X, and is *Glibc* on Linux. These can also be accessed with *import Foundation*, which will include the appropriate operating system module.

A Swift program to print arguments in uppercase can be implemented as a script:

```
#!/usr/bin/env xcrun swift

import func Darwin.exit

# import func Glibc.exit # for Linux
```

```
let args = Process.arguments[1..<Process.arguments.count]

for arg in args {

  print("\(arg.uppercaseString)")

}

exit(0)
```

Running this with *hello world* results in the following:

```
$ ./upper.swift hello world

HELLO

WORLD
```

Conventionally, the entry point to Swift programs is via a script called *main.swift*. If starting a Swift-based command-line application project in Xcode, a *main.swift* file will be created automatically. Scripts do not need to have a *.swift* extension; for example, the previous example could be called *upper* and it would still work.

Compiled Swift scripts

While interpreted Swift scripts are useful for experimenting and writing, each time the script is started, it is interpreted using the Swift compiler and then executed. For simple scripts (such as converting arguments to upper case), this can be a large proportion of the script's execution time.

To compile a Swift script into a native executable, use the *swiftc* command with the *-o* output flag to specify a file to write to. This will then generate an executable that does exactly the same as the interpreted script, only much faster. The *time* command can be used to compare the running time of the interpreted and compiled versions:

```
$ time ./upper.swift hello world    # Interpreted

HELLO

WORLD

real  0m0.145s

$ xcrun swiftc -o upper upper.swift # Compile step

$ time ./upper hello world          # Compiled

HELLO

WORLD

real  0m0.012s
```

Of course, the numbers will vary, and the initial step only happens once, but startup is very lightweight in Swift. The numbers are not meant to be taken in magnitude but rather as relative to each other.

The compile step can also be used to link together many individual Swift files into one executable, which helps create a more organized project; Xcode will encourage having multiple Swift files as well.

Summary

In this article, we saw that the Swift interpreter is a great way of learning how to program in Swift. It allows expressions, statements, and functions to be created and tested along with a command-line history that provides editing support. The basic collection types of arrays and collections, the standard data types, such as strings and numbers, optional values, and structures, were presented. Control flow and functions with positional, named, and *variadic* arguments, along with default values were also presented. Finally, the ability to write Swift scripts and run them from the command line was also demonstrated.

To learn more about Swift, you can refer the following books published by Packt Publishing (https://www.packtpub.com/):

- **Learning Swift** (https://www.packtpub.com/application-development/learning-swift)
- **Swift Essentials** (https://www.packtpub.com/application-development/swift-essentials)
- **Swift 2 Design Patterns** (https://www.packtpub.com/application-development/swift-2-design-patterns)

You've been reading an excerpt of: Swift Essentials – Second Edition