APPENDIX: SWIFT BASICS

Swift is a new programming language for developing iOS, macOS, watchOS and tvOS apps. As compared to Objective-C, Swift is a neat language and will definitely make developing iOS apps easier. In this appendix, I will give you a brief introduction of Swift.

VARIABLES

Use `var` for variables that can change ("mutable") and `let` for constants that can't change ("non-mutable").

Integers are "whole" numbers, i.e. numbers without a fractional component.

```
var meaningOfLife: Int = 42
```

Floats are decimal-point numbers, i.e. numbers with a fractional component.

```
var phi: Float = 1.618
```

Doubles are floating point numbers with double precision, i.e. numbers with a fractional component. Doubles are preferred over floats.

```
let pi: Double = 3.14159265359
```

A *String* is a sequence of characters, like text.

```
var message: String = "Hello World!"
```

A *boolean* can be either `true` or `false`. You use booleans for logical operations.

```
var isLoggedIn: Bool = false
```

You can assign the result of an expression to a variable, like this:

```
var result: Int = 1 + 2
```

An *expression* is programmer lingo for putting stuff like variables, operators, constants, functions, etc. together. Like `a + b` in this example:

```
let a = 3

let b = 4
```

```
let c = a + b
```

Swift can determine the *type* (`Int`, `Double`, `String`, etc.) of a variable on its own. This is called *type inference*. In this example, the type of `name` is inferred to be `String`.

```
var name = "Arthur Dent"
```

Variables can be *optional*, which means it either contains a value or it's `nil`. Optionals make coding Swift safer and more productive. Here's an optional string:

```
var optionalMessage: String?
```

FUNCTIONS

Functions are containers of Swift code. They have input and output. You often use functions to create abstractions in your code, and to make your code reusable.

Here's an example of a function:

```
func greetUser(name: String, bySaying greeting:String = "Hello") -> String
{
    return "\(greeting), \(name)"
}
```

This function has two *parameters* called `name` and `greeting`, both of type `String`. The second parameter `greeting` has an *argument label* `bySaying`. The return type of the function is `String`, and its code is written between those squiggly brackets.

You call the function like the following. The function is called, with two arguments, and the return value of the function is assigned to `message`.

```
let message = greetUser(name: "Zaphod", bySaying: "Good Morning")
```

Courses, books, documentation, etc. uses a special notation for function signatures. It'll use the function name and argument labels, like `greetUser(name:bySaying:)`, to describe the function.

OPERATORS

The *assignment operator* `=` assigns what's right of the operator to what's left of the operator. Don't confuse it with `==`!

```
let age = 42
```

Swift has a few basic math operators:

- `a + b` for *addition* (works for strings too)
- `a - b` for subtraction
- `a * b` for multiplication
- a / b for *division*
- a % b for *remainder* (or use isMultiple(of:))
- `-a` for *minus* (invert sign)

Unlike other programming languages, Swift does not have `--` and `++` operators. Instead it has:

- `a += b` for *addition*, such as `n += 1` for `n++` or `n = n + 1`
- `a -= b` for subtraction, such as `n -= 1` for `n--` or `n = n - 1`

You can also use `+=` for arrays:

```
var rappers = ["Eminem", "Jay-Z", "Snoop Dogg"]

rappers += ["Tupac"]
```

Swift has 6 comparison operators:

- `a == b` for *equality*, i.e. "a is equal to b"
- `a != b` for *inequality*, i.e. "a is not equal to b"
- `a > b` for *greater than*, i.e. "a is greater than b"
- `a < b` for *less than*, i.e. "a is less than b"
- `a >= b` for greater than or equal
- `a <= b` for less than or equal

Swift also has the identity operators `===` and `!==`. You can use them to test if two variables reference the exact *same object*. Contrast this with `==` and `!=`, which merely test if two objects are equal to each other.

You can also compare strings, which is helpful for sorting. Like this:

```
"abc" > "xyz"        // false

"Starbucks" > "Costa" // true

"Alice" < "Bob"      // true
```

Swift has 3 logical operators:

- `a && b` for Logical AND, returns `true` if `a` and `b` are `true`, or `false` otherwise
- `a || b` for *Logical OR*, returns `true` if either `a` or `b` is `true`, or both are `true`, or `false` otherwise
- `!a` for *Logical NOT*, returns `true` if `a` is `false`, and `false` if `a` is `true` (i.e., the opposite of `a`)

Swift has a few range operators. You can use them to define ranges of numbers and strings.

- `a...b`, the *closed range operator*, defines a range from `a` to `b` including `b`
- `a..<b`, the half-open range operator, defines a range from `a` to `b` not including `b`

You can also use *one-sided ranges*. They're especially useful in arrays.

- `a...` in `array[a...]` defines a range from `a` to the end of the array
- `...a` in `array[...a]` defines a range from the beginning of the array to `a`
- `..<a` in `array[..<a` defines a range from the beginning of the array to `a`, not including `a` itself

CLASSES, OBJECTS, PROPERTIES

Classes are basic building blocks for apps. They can contain functions, sometimes called *methods*, and variables, called *properties*.

```
class Office: Building, Constructable
{
    var address: String = "1 Infinite Loop"
    var phone: String?

    @IBOutlet weak var submitButton:UIButton?
```

```
    lazy var articles:String = {
        return Database.getArticles()
    }()

    override init()
    {
        address = "1 Probability Drive"
    }

    func startWorking(_ time:String, withWorkers workers:Int)
    {
        print("Starting working at time \(time) with workers \(workers)")
    }
}
```

The class definition starts with `class`, then the class name `Office`, then the `Building` class it *inherits* from, and then the `Constructable` *protocol* it conforms to. Inheritance, protocols, all that stuff, is part of *Object-Oriented Programming*.

Properties are variables that belong to a class instance. This class has 4 of them: `address`, `phone`, the outlet `submitButton` and the lazy computed property `articles`. Properties can have a number of attributes, like `weak` and `@IBOutlet`.

The function `init()` is *overridden* from the superclass `Building`. The class `Office` is a *subclass* of `Building`, so it inherits all functions and properties that `Building` has.

You can create an *instance* of a class, and change its properties, like this:

```
let buildingA = Office()
buildingA.address = "Sector ZZ9 Plural Z Alpha”
```

Extensions let you add new functions to existing types. This is useful in scenarios where you can't change the code of a class. Like this:

```
extension Building
{
    func evacuate() {
        print("Please leave the building in an orderly fashion.")
    }
}
```

STRUCTS

Structs or *structures* in Swift allow you to encapsulate related properties and functionality in your code, that you can reuse. Structs are value types.

We declare them like this:

```
struct Jedi {
    var name: String
    var midichlorians: Int
}
```

You can create an instance of the `Jedi` struct like this:

```
var obi_wan = Jedi(name: "Obi-Wan Kenobi", midichlorians: 13400)
```

Here's how you can read a property from `obi_wan`:

```
print(obi_wan.midichlorians)
// Output: 13400
```

We can also include functions inside our structs, like this:

```
struct Jedi {
    var name: String
    var midichlorians: Int

    func mindTrick() {
        print("These aren't the Droids you're looking for...")
    }
}

// Instance of a struct
var obi_wan = Jedi(name: "Obi-Wan Kenobi", midichlorians: 13400)

// Reading a property
print(obi_wan.midichlorians)

// Calling mindTrick() function
obi_wan.mindTrick()
```

CONTROL FLOW: CONDITIONALS

This is an `if`-statement, or *conditional*. You use them to make decisions based on logic.

```
if isActive
{
    print("This user is ACTIVE!")
} else {
    print("Inactive user...")
}
```

You can combine multiple conditionals with the `if-elseif-else` syntax, like this:

```
var user:String = "Bob"

if user == "Alice" && isActive
{
    print("Alice is active!")
}
else if user == "Bob" && !isActive
{
    print("Bob is lazy!")
}
else
{
    print("When none of the above are true...")
}
```

The `&&` is the *Logical AND* operator. You use it to create logical expressions that can be evaluated to `true` or `false`. Like this:

```
if user == "Deep Thought" || meaningOfLife == 42
{
    print("The user is Deep Thought, or the meaning of life is 42...")
```

```
}
```

Conditionals can be challenging to comprehend. Like this:

```
if c < a && b + c == a
{
    print("Will this ever happen?")
}
```

Swift has a special operator, called the *ternary conditional operator*. It's coded as `a ? b : c`. If `a` is `true`, the expression returns `b`, and if `a` is `false`, the expression returns `c`. It's the equivalent of this:

```
if a {
    b
} else {
    c
}
```

LOOPS

Loops repeat stuff. It's that easy. Like this:

```
for i in 1...5 {
    print(i)
}
```

This prints `1` to `5`, including `5`! You can also use the half-open range operator `a..<b` to loop from `a` to `b` not including `b`. Like this:

```
for i in 1..<4 {
    print(i)
}
// Output: 1 2 3
```

When you don't know how many times a loop needs to run *exactly*, you can use a `while` loop. A `while` loop keeps iterating as long as its expression is `true`.

```
while b <= 60 && b > 0
{
    print(b)
    b -= 1
}
```

SWITCH

A `switch` statement takes a value and compares it against one of several *cases*. It's similar to the `if-else if-else` conditional, and it's an elegant way of dealing with multiple states.

An example:

```
let weather = ...

switch weather
{
    case .rain:
```

```
            print("Bring a raincoat!")
        case .clear, .sunny:
            print("Don't forget your sunglasses.")
        case .overcast:
            print("It's really depressing.")
        case .tsunami, .earthquake:
            print("OH NO! BIG WAVE!")
        default:
            print("Expect the best, prepare for the worst.")
    }
```

In Swift, `switch` statements don't have an implicit *fall-through*, but you can use `fallthrough` explicitly. Every case needs to have at least one line of code in it. You don't have to use a `break` explicitly to end a case.

The `switch` cases need to be *exhaustive*. For example, when working with an `enum`, you'll need to incorporate every value in the enumeration. You can also provide a `default` case, which is similar to `else` in a conditional.

You can also use Swift *ranges* to match interval for numbers, use tuples to match partial values, and use Swift's `where` keyword to check for additional conditions.

STRINGS

Strings are pretty cool. Here's an example:

```
var jobTitle: String = "iOS App Developer”
```

Inside a string, you can use *string interpolation* to string together multiple strings. Like this:

```
var hello = "Hello, \(jobTitle)"
// Output: Hello, iOS App Developer
```

You can also turn an `Int` into a `String`:

```
let number = 42
let numberAsString = “\(number)"
```

And vice-versa:

```
let number = "42"
let numberAsInt = Int(number)
```

You can loop over the characters of a string like this:

```
let text = "Forty-two!"

for char in text {
    print(char)
}
```

You can get individual characters and character ranges by using *indices*, like this:

```
let text = "Forty-two!"
```

```
text[text.startIndex] // F
text[text.index(before: text.endIndex)] // !
text[text.index(text.startIndex, offsetBy: 3)] // t
text[..<text.index(text.startIndex, offsetBy: 3)] // For
text[text.index(text.endIndex, offsetBy: -4)...] // two!
```

OPTIONALS

Optionals can either be `nil` or contain a value. You **must** always *unwrap* an optional before you can use it.

This is Bill. Bill is an optional.

```
var bill: String? = nil
```

You can unwrap `bill` in a number of ways. First, this is *optional binding*.

```
if let definiteBill = bill {
    print(definiteBill)
}
```

In this example, you bind the non-optional value from `bill` to `definiteBill` *but only when* `bill` *is not* `nil`. It's like asking: "Is it not `nil`?" OK, if not, then assign it to this constant and execute that stuff between the squiggly brackets.

You can also use *force-unwrapping* to unwrap an optional. Like this:

```
var droid: String? = "R2D2"

if droid != nil {
    print("This is not the droid you're looking for: \(droid!)")
}
```

See how that `droid` is force-unwrapped with the exclamation mark `!`? You should keep in mind that if `droid` is `nil` when you force-unwrap it, your app will crash.

You can also use *optional chaining* to work your way through a number of optionals. This saves you from coding too much optional binding blocks. Like this:

```
view?.button?.title = “BOOYAH!"
```

In this code, `view`, `button` and `title` are all optionals. When `view` is `nil`, the code "stops" before `button`, so the `button` property is never accessed.

One last thing... the *nil-coalescing operator*. You can use it to provide a default value when an expression results in `nil`. Like this:

```
var meaningOfLife = deepThought.think() ?? 42
```

See that `??`. When `deepThought.think()` returns `nil`, the variable `meaningOfLife` is `42`. When that function returns a value, it's assigned to `meaningOfLife`.

COLLECTIONS: ARRAYS

Arrays are a collection type. Think of it as a variable that can hold multiple values, like a closet that can contain multiple drawers. Arrays always have *numerical* index values. Arrays always contain elements of the same type.

```
var hitchhikers = ["Ford", "Arthur", "Zaphod", “Trillian"]
```

You can add items to the array:

```
hitchhikers += [“Marvin"]
```

You can get items from the array with *subscript syntax*:

```
let arthur = hitchhikers[1]
```

Remember that arrays are *zero-index*, so the index number of the first element is `0` (and not `1`).

You can iterate arrays, like this:

```
for name in hitchhikers {
    print(name)
}
```

A helpful function on arrays is `enumerated()`. It lets you iterate index-value pairs, like this:

```
for (index, name) in hitchhikers.enumerated() {
    print("\(index) = \(name)")
}
```

DICTIONARIES

Dictionaries are also collection types. The items in a dictionary consists of key-value pairs. Unlike arrays, you can set your own key type. Like this:

```
var score = [
    "Fry": 10,
    "Leela": 29,
    "Bender": 1,
    "Zoidberg": 0
]
```

What's the type of this dictionary? It's `[String: Int]`. Just like with arrays, you can use *subscript syntax* to get the value for a key:

```
print(score["Leela"])
// Output: 29
```

You can add a key-value pair to a dictionary like this:

```
score["Amy"] = 9001
```

Change it like this:

```
score["Bender"] = -1
```

And remove it like this:

```
score.removeValue(forKey: “Zoidberg")
```

You can also iterate a dictionary, like this:

```
for (name, points) in score
{
    print("\(name) has \(points) points");
}
```

SETS

Sets in Swift are similar to arrays and dictionaries. Just like arrays and dictionaries, the `Set` type is used to store multiple items of the same type in one collection.

Here's an example:

```
var fruit:Set = ["apple", "banana", "strawberry", “jackfruit"]
```

You can add and remove items like this:

```
fruit.insert("pineapple")
fruit.remove(“banana")
```

Sets are different from arrays and dictionaries, in these ways:

- Sets don't have an order – they're literally unsorted
- Every item in a set needs to be unique
- Sets don't have indices or keys
- Instead, a set's values need to be *hashable*
- Because set items are hashable, you can search sets in *O(1)* time

Here's how you can quickly search a set:

```
let movies:Set = ["Rocky", "The Matrix", "Lord of the Rings"]

if movies.contains("Rocky") {
```

```
    print("Rocky is one of your favorite movies!")
}
```

Sets are particularly useful for membership operations, to find out if sets have items in common for example. We can make a union of sets, subtract sets, intersect them, and find their differences.

Consider the following Italian coffees and their ingredients:

```
let cappuccino:Set = ["espresso", "milk", "milk foam"]
let americano:Set  = ["espresso", "water"]
let machiato:Set   = ["espresso", "milk foam"]
let latte:Set      = ["espresso", “milk"]
```

Can we find the **union** (add items) of two coffees?

```
machiato.union(latte)
// ["espresso", "milk foam", “milk"]
```

Can we **subtract** one coffee from another?

```
cappuccino.subtracting(americano)
// ["milk foam", “milk"]
```

Can we find the **intersection** (shared items) of two coffees?

```
latte.intersection(cappuccino)
["espresso", “milk"]
```

Can we find the **difference** between two coffees?

```
latte.symmetricDifference(americano)
["milk", "water"]
```

CLOSURES

With *closures* you can pass around blocks of code, like functions, as if they are variables. You use them, for example, by passing a callback to a lengthy task. When the task ends, the callback – a closure – is executed.

You define a closure like this:

```
let authenticate = { (name: String, userLevel: Int) -> Bool in
    return (name == "Bob" || name == "Alice") && userLevel > 3
}
```

You call the closure like this:

```
authenticate("Bob", 7)
```

If we had a user interface for authenticating a user, then we could pass the closure as a callback like this:

```
let loginVC = MyLoginViewController(withAuthCallback: authenticate)
```

Another use case for closures is multi-threading with Grand Central Dispatch. Like this:

```
DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + .seconds(60)) {
    // Dodge this!
}
```

In the above example, the last argument of `asyncAfter(deadline:execute:)` is a closure. It uses the *trailing closure* syntax. When a closure is the last argument of a function call, you can write it after the function call parentheses and omit the argument label.

GUARD

The `guard` statement helps you to return functions early. It's a conditional, and when it isn't met you need to exit the function with `return`.

Like this:

```
func loadTweets(forUserID userID: Int)
{
    guard userID > 0 else {
        return
    }
    // Load the tweets...
}
```

You can read that as: "*Guard that the User ID is greater than zero, or else, exit this function*". Guard is especially powerful when you have multiple conditions that should return the function.

Guard blocks always need to exit its enclosing scope, i.e. transfer control outside of the scope, by using `return`, `throw`, `break` or `continue`.

You can also combine `guard` and `if let` (optional binding) into `guard let`. This checks if the given expression is not `nil`, and assigns it to a constant. When the expression is `nil`, the `else` clause of `guard` is executed.

```
guard let user = object?.user else {
    return
}
```

You can now use the `user` constant in the rest of the scope, below the `guard let` block.

DEFER

With `defer` you can define a code block that's executed when your function returns. The `defer` statement is similar to `guard`, because it also helps with the flow of your code.

Like this:

```
func saveFile(withData data: Data) {

    let filePointer = openFile("../example.txt")

    defer {
        closeFile(filePointer)
    }

    if filePointer.size > 0 {
        return
    }

    if data.size > 512 {
        return
    }

    writeFile(filePointer, withData: data)
}
```

In the example code you're opening a file and writing some data to it. As a rule, you need to close the file pointer before exiting the function.

The file isn't written to when two conditions aren't met. You have to close the file at those points. Without the `defer` statement, you would have written `closeFile(_:)` twice.

Thanks to `defer`, the file is always closed when the function returns.

GENERICS

In Swift your variables are *strong typed*. When you set the type of animals your farm can contain to `Duck`, you can't change that later on. With *generics* however, you can!

Like this:

```
func insertAnimal<T>(_ animal: T, inFarm farm: Farm)
{
    // Insert `animal` in `farm`
}
```

This is a *generic function*. It uses a *placeholder type* called `T` instead of an actual type name, like `String`.

If you want to insert ducks, cows, birds and chickens in your farm, you can now do that with one function instead of 4.

TUPLES

With *tuples* you get two (or more) variables for one. They help you structure your code better. Like this:

```
let coffee = ("Cappuccino", 3.99)
```

You can now get the price of the coffee like this:

```
let (name, price) = coffee
print(price)
// Output: 3.99
```

When you need just the name, you can do this:

```
let (name, _) = coffee
print(name)
// Output: Cappuccino
```

You can also name the elements of a tuple, like this:

```
let flight = (code: "XJ601", heading: "North", passengers: 216)
print(flight.heading)
// Output: North
```

ENUMERATIONS

With enumerations, also known as enums, you can organize groups of values that are related. Here's an example:

```
enum Compass {
    case north
    case east
    case south
    case west
}
```

Here's how you use them:

```
let direction: Compass = .south
```

Enums and the `switch` statement are a powerful couple. Here's an example:

```
enum Emotion {
    case happy, sad, angry, scared, surprised
}

switch robot.mood {
case .angry:
    robot.destroyAllHumans()
case .sad:
    robot.cry()
case .happy:
    robot.play("happy.mp3")
case default:
```

```
    print("Error: emotion not supported.")
}
```

You can also assign raw values to enums, by using existing Swift types like `String`. Here's an example:

```
enum Flavor:String {
    case vanilla = "vanilla"
    case strawberry = "strawberry"
    case chocolate = "chocolate"
}
```

With this approach, you can get the string value for an enum like this:

```
let icecream = Flavor.vanilla
print(icecream.rawValue)
// Output: vanilla
```

You can now also use a string to create an enum, like this:

```
let icecream = Flavor(rawValue: "vanilla")
print(icecream)
// Output: Optional(Flavor.vanilla)
```

You can also associate values with individual cases of an enumeration, like this:

```
enum Item {
    case weapon(Int, Int)
    case food(Int)
    case armor(Int, Int, Double)
}
```

You can now use the enumeration's associated values, like this:

```
func use(item: Item)
{
    switch item {
    case .weapon(let hitPoints, _):
        player.attack(hitPoints)
    case .food(let health):
        player.health += health
    case .armor(let damageThreshold, let weight, let condition):
        player.damageThreshold = Double(damageThreshold) * condition
    }
}
```

In the above code, we're using `hitPoints` to "attack" in case `item` is the enum type `weapon(Int, Int)`. This way you can associate additional values with an enum case.

ERROR HANDLING

Errors in Swift can be thrown, and should be caught. You can define an error type like this:

```
enum CreditCardError: Error {
    case insufficientFunds
    case issuerDeclined
    case invalidCVC
```

```
}
```

When you code a function that can throw errors, you have to mark its function definition with `throws`. Like this:

```
func processPayment(creditcard: String) throws {
    …
```

Inside the function, you can then throw an error like this:

```
throw CreditCardError.insufficientFunds
```

When you *use* a function that can throw errors, you have to wrap it in a `do-try-catch` block. Like this:

```
do {
    try processPayment(creditcard: "1234.1234")
}
catch let error {
    print(error)
}
```

In the example above, the `processPayment(creditcard:)` function is marked with the `try` keyword. When an error occurs, the `catch` block is executed.