**Apple WWDC 2014**

(Video courtesy of [AppleKeynotes](https://www.youtube.com/watch?v=MO7Ta0DvEWA" \t "_blank))

**Objective-C**

**No, we don't need to learn Objective-C**. Many educational resources and Stack Overflow questions are for Objective-C, and many applications in the App Store are still written in Objective-C. However, **Apple introduced Swift to replace Objective-C,**and although most of the libraries (like Foundation) that we use in Swift are written in Objective-C, we don't ever have to look behind the scenes or change these libraries. Knowing how to read this language will help us better understand our errors, develop a deeper understanding of what developing for Apple used to be like, and also understand why Apple decided to introduce a new language, but **it is not a requirement**. This course will be focused on Swift because it is the**language of the future**.

**Why Swift?**

If you have written Objective-C before, you will completely understand why. Swift is similar to interpreted languages like Ruby and JavaScript. However, it is a compiled language. It's the best of both worlds. Since Swift is still relatively new, this is a great opportunity for new programmers like us to start mastering this language that was designed to power all future Apple programs. Everyone's in the same boat. Moreover, Swift is the language of the future for all applications developed in the Apple ecosystem. This offers a great opportunity to learn a new language with a lot of backing and interest.

[Check out what's new with Swift](https://developer.apple.com/videos/play/wwdc2021/10192/)

**Xcode**

**Even if you have installed it before make sure that you have the latest one**. Swift is constantly evolving with each new version of Xcode and as Swift developers, we have to be up to date with all of the versions. We need to know Xcode really well. This is what we will be using in writing our code. **Xcode is an IDE or Integrated Development Environment**which basically means that it does a lot to help the developer. Unlike our plain text files written using a simple text editor and then run through the terminal, Xcode provides us with the **integrated tools to write, build, test, and debug our code.**

**Documentation**

We have to become **self-sufficient iOS developers**. Every year, new features will be added with the introduction of new operating systems and new devices. To keep up, we are going to be **spending a lot of time in Apple's documentation**. We will be directing you to Apple's documentation because we believe that**learning how to digest Apple's documentation is the single most important skill for an iOS developer** to have. Let's get started by reading [About Swift](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/index.html#//apple_ref/doc/uid/TP40014097-CH3-ID0).

**NOTE:**

On the same site, there is a very well written and complete [Language Guide](https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html). Bookmark this page and look here first when learning swift syntax.

# Reading Documentation

Apple documentation is very comprehensive, but because of this, it can be difficult to find what you are looking for. Let's take a look at a specific page to get more comfortable. Open the [String documentation](https://developer.apple.com/documentation/swift/string) to follow along.

The most important thing to look at is the **Documentation Path**. It is similar to the directory structure of your computer. You can click on any level to see what that level contains.

1. At the root is **Documentation**. This is the highest level and gives you access to categories such as App Frameworks, Graphics and Games, App Services, Media, Web, etc.
2. Next, we see **Swift**. The Swift framework defines a base layer of functionality for writing Swift programs, including types and data structures, protocols, etc. Most of what we will be looking up will be here or in UIKit.
3. Next, we see **String**, a structure within the Swift framework.

This tree structure helps us see how different parts of the documentation relate to each other and make it easy to get a high-level understanding of the topic you are researching. You can bookmark the documentation page to search later, or you can just add "swift docs" to any google search and pick the option at **developer.apple.com**.

Spend a few minutes getting familiar with the Apple documentation and remember, if you don't know something, it's in the docs.

# Playground

Playgrounds were introduced starting from Xcode 6. Swift is a **compiled language**. What this means is that Xcode must first compile the entire program into 1s and 0s before it can execute the program. On the other hand, **Ruby and JavaScript are interpreted languages**. The difference is that compiled languages have to be translated completely before running while interpreted languages get translated on the fly as the program is getting read.

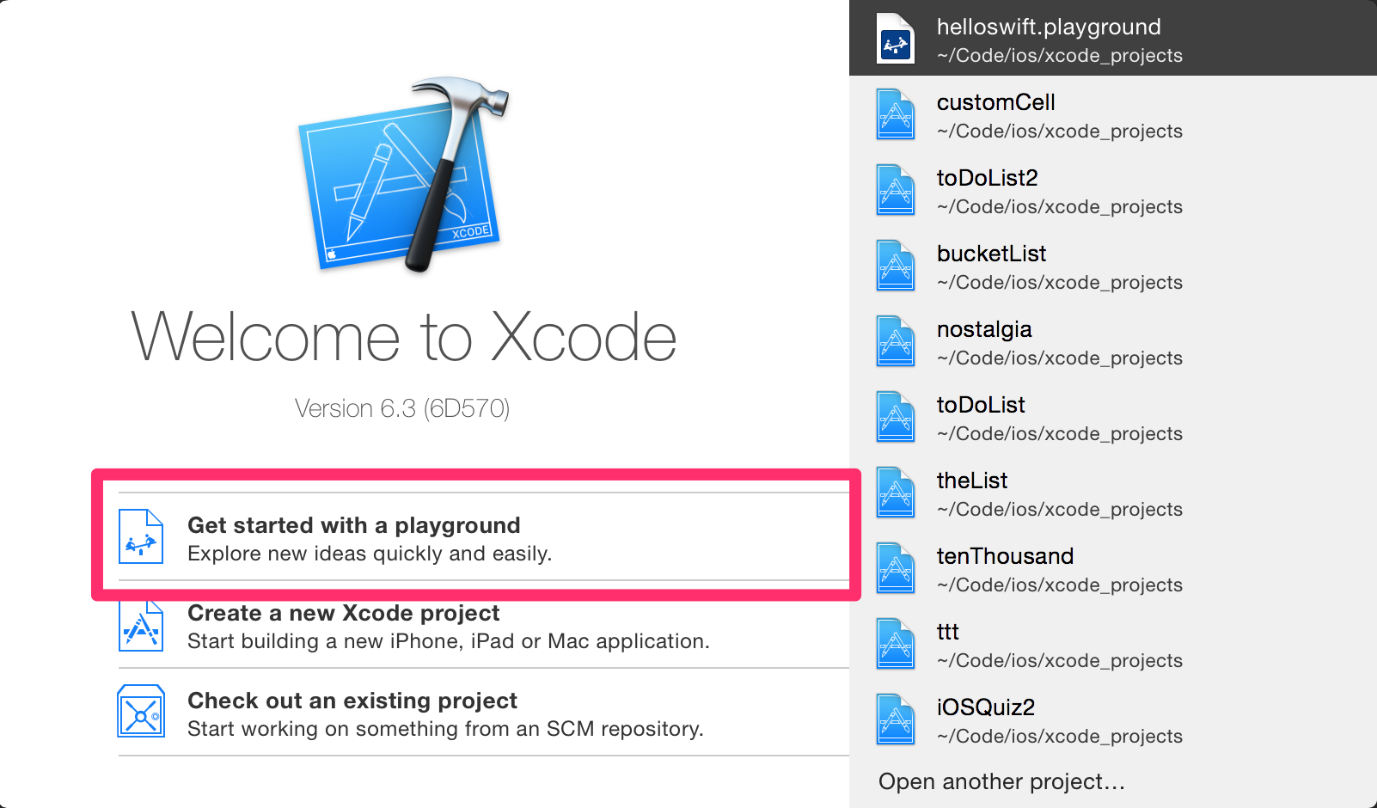
Interpreted languages are slower than compiled languages. However, they are great for certain tasks where speed doesn't matter very much. For example, Interpreted languages are great for web applications because the bottleneck is almost never the language and if it is, it is probably because of poorly written code. On the other hand, native applications and games are often built in compiled languages because once they are compiled, it takes less time to execute than a program written with an interpreted language.

Compiled languages are great but sometimes miss the immediate feedback you can get while playing around with Interpreted languages. This proposes a dilemma because playing around with code line by line is essential in learning a new language. However, this is hard to do with compiled languages. Xcode 6 fixed this issue with Swift with the introduction of Playgrounds. A playground does not require you to compile and run a complete project and evaluates Swift code on the fly like an interpreter for an interpreted language would.

Go ahead and let's create our first Playground project. Click on **Get started with a playground**and use a blank template to create a file called stringconcat.

**NOTE:**

If playgrounds give you trouble (they can be very slow depending on your version of Xcode and your computer hardware), there is an [online tool](http://online.swiftplayground.run/) you can use instead.



Note: You may also create a new playground with [option + command + shift + N](https://s3.us-east-1.amazonaws.com/General_V88/boomyeah2015/codingdojo/curriculum/content/chapter/1624248474__playground.png)

The playground is split into two sections. On the left, there is the Swift code editor where we can write our code and on the right we have a sidebar that evaluates Swift code.

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

//: Playground - noun: a place where people can play

On the first line, we have a comment. Comments will be ignored when programs are executed. They serve more as a reminder or notice for other developers. We can create a comment by prefixing our comment with '//' and the rest of the line will turn into a comment.

import UIKit

The import UIKit means that we are including all of the **Application Programming Interfaces (APIs)**made available to iOS and Mac developers by the **UIKit framework**. APIs are just **classes and functions written by someone** that we can use. We have to follow certain instructions depending on what API we are using to make our program run. **Utilizing Apple's APIs is a key component in becoming an effective developer**. We will be directed to Apple's documentation throughout the course so that we can practice learning new skills from reading the documentation. Spend no more than 10 minutes perusing through [UIKit Framework Reference](https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKit_Framework/" \t "_blank).

var str = "Hello, playground"

We declare a variable in swift with the keyword var. **A variable should only hold data that is going to change.**If the data is not going to change, we should not be using a variable. Inside the variable 'str' we are going to assign the value "Hello, playground". The quotations mean that we are creating an**instance of the String Type**. Strings are ordered **collections of characters**. We named our variable str but we could have given it any other name.

str += " you are so cool"

Here we are adding two instances of the String Type together to create a new instance of String. This is a shorthand way (compound assignment operator) of adding instances of String Type this way:

str = str + " you are so cool"

Here's a quick warning on creating instances of Strings. **Creating instances of String Type does not work with single quotes.**

var errorStr = 'single quotes do not work' // don't use single quotes to create instances of String

## print

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

We can print out our variables using a **print** function which is a function used to print a value to the console followed by a line break. To see the output we can click the small up arrow on the bottom left of the playground. Before we move on further let's read the Basic Operators overview, Terminology, Assignment Operator, Arithmetic Operators, Compound Assignment Operators in [Basic Operators](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/BasicOperators.html#//apple_ref/doc/uid/TP40014097-CH6-ID60).

# Variables

In order to store information in our code, we use variables. Swift is what we call a **statically typed language.** What this means is that the **types** (eg, int, boolean, string, etc) of our variables must be. First, let's look at how we declare our types:

// Variables are used to store data. We declare a variable using the "var" keyword.

// We also have to specify the type of that variable.

var name: String = "Anakin"

// Every variable declaration follows the same format:

// var variable\_name: variable\_type = variable\_value

var number: Int = -42 // "Int" type stores whole numbers both positive and negative

var dubs: Double = 4.2 // "Double" type stores numbers with decimals (allocates 64 bits)

var dec: Float = 4.2 // "Float" type also stores numbers with decimals (allocates 32 bits)

var isTrue: Bool = true

Although this may seem unnecessary, statically typed languages allow the computer to run more efficiently by allocating just enough memory to store the specific data type. Internally, computers store variables in blocks of memory that are just 1s and 0s. Specifying the type allows the computer to understand not only how much actual space to allocate but also how to interpret the 1s and 0s that are stored!

A statically typed language not only forces you to **declare the type**but, more importantly, disallows you from **changing the type.** This means that once a variable is declared as a "String" type it must always store a String. Let's see this in action:

var name: String = "Anakin"

name = "Darth Vader" // Changing the "name" variable to another string is allowed...

// ...but trying to change the var to a value of ANOTHER type is not allowed.

name = 42 // THIS WON'T WORK because we are trying to change a variable that is a "String"

// type to a value that is a "Int" type

**This means that once we declare a variable we can only hold one specific type of data inside of it!**

### Why is Swift Statically Typed?

**Performance**.

As a statically typed language, the type of the variables are checked before the code is run. Try running this example in your playground:

var number: Int = 5

if number > 2 {

number += 5

}

else {

number += "hello"

}

You should notice that we encounter an error before we even try running the code!

Swift, our statically typed language, notices that we cannot add an integer and a string, even though the problematic code never would have been run anyway! This is because the types are being checked before run time. Try the equivalent in Python, which also disallows adding strings and integers. Since Python is **dynamically** typed, the type check does not happen until runtime. Since the code inside the else condition never runs, Python does not throw an error. **Swift is statically typed** so it forces the developer to be more conscious about types, and it also allows the computer to run more efficiently by allocating just enough space for each variable.

# Type Inference

Constantly having to specify the type can get tedious very quickly. Fortunately, in addition to caring about performance, Swift also caters to the developer! Swift uses "type inference" to allow us to forego specifying the type when declaring a variable and assigning it to a value in the same line. Swift will look at the value and "infer" the type so that you don't have to specify it.

// Swift has "type inference"

var name = "Anakin" // String type

// Swift looked at val on the right and inferred type: we didn't have to specify!

#### Even though we didn't have to specify the type, the variable still has a type that cannot change after a declaration.

var name = "Anakin" // Type inferred as String based on the value "Anakin"

name = 42 // THIS IS STILL NOT ALLOWED since name is a "String" type

**Take a minute and try out the above code in a playground and see what the error message is.**

# Constants

**Variables hold data that varies**. In many programming languages, variables are used for everything, even for values that never change. **Constants hold data that does not vary**. We cannot change constants. Constants are a big deal in Swift. **Use constants in all parts of your code except for areas where you HAVE to use a variable**. Using constants to hold data that won't change allows our code to be more explicit while at the same time helps the compiler to run our programs more effectively.

We declare constants using the "let" keyword:

// Declare name as a constant because we anticipate that it will NOT change.

let name: String = "Anakin Skywalker"

// Note that just like variables, constants also have types

// Now we cannot change the name:

name = "Darth Vader" // THIS WON'T WORK because name is a constant so we CANNOT change it.

## Mutable vs. Immutable

**Mutable** means that we can change the value and**immutable** means that we cannot change the value. In Objective-C, there were explicit Types that were indicated as Immutable. For example, there was a class for mutable String Types and a separate class for immutable String Types. In Swift, we can still have immutable and mutable Strings without having another class because of the difference between the let and the var keywords.

We use the keyword let when we want to declare a constant. **Constants** are values that are not going to change during your program. We use the var keyword when we want to hold onto a value that might change. Using these keywords we can create mutable strings and immutable strings.

// We are declaring a mutable string because we store it in a var.

var myMutableString = "change me"

// We are declaring an immutable string because we store it in a const.

let myImmutableString = "can't change"

// We can change the value of a mutable string...

myMutableString += "!"

// ...but we cannot change the value of an immutable string.

myImmutableString += "ahhhh" // This will error.

This is the same for Arrays.

// We are declaring a mutable array because we are storing it in a variable.

var myMutableArray = ["one"]

// We are declaring an immutable array because we are storing it in a constant.

let myImmutableArray = ["uno"]

// We can append new String instances to the mutable array...

myMutableArray.append("two")

// ... in fact we can add new things to this mutable array as well...

myMutableArray + ["three", "four"]

// ...but we will get an error when we try to do the same to an immutable array.

myImmutableArray.append("dos")

And for dictionaries.

// We are declaring a mutable dictionary because we store it in a variable.

var myMutableDictionary = ["one": 1]

// We are declaring an immutable dictionary because we store it in a constant.

let myImmutableDictionary = ["uno": 1]

// We can add a new key-value pair to our mutable dictionary ...

myMutableDictionary["two"] = 2

// ... or update a value of an existing key...

myMutableDictionary["one"] = 0

// ... but we get an error when we try to do same to an immutable dictionary.

myImmutableDictionary["uno"] = 2

# Int

**There are unlimited numbers of numbers and a limited number of bits**. There is a minimum and maximum value that an instance of an Int Type can represent. Int Type is signed in Swift. What this means is that **it uses some of its possible representations to store negative numbers**. Let's see what the maximum value and minimum value of the Int Type is in Swift:

print("The maximum value \(Int.max)")

print("The minimum value \(Int.min)")

On OS X (the desktop operating system) allocates 64 bits to store an Int Type. On iOS, this is different because some devices store Int Types in 64 bits while older models such as the iPhone 5 utilize the 32-bit infrastructure. This seems like a big headache. Does that mean we have to write different code for different devices? **If we use the instances of the Int Type, the compiler will determine the appropriate size for an Int when it builds your program.**

## Int32

We can have an Int be stored in 32 bits by creating an instance of Int32 Type. It is important to note that Int and Int32 are different Types. Int32 is also signed. Let's see what the maximum and the minimum value of Int32 are.

print("The maximum value \(Int32.max)")

print("The minimum value \(Int32.min)")

There's also similar Types available such as Int8, Int16, and Int64. We only use these specific integer types when we are doing something like cryptography where the way integers are stored is important. We will also run across these types when we are interacting with libraries built when storing 32 bits for an integer was very common. **Most of the time you will simply use the Int type**

## UInt32

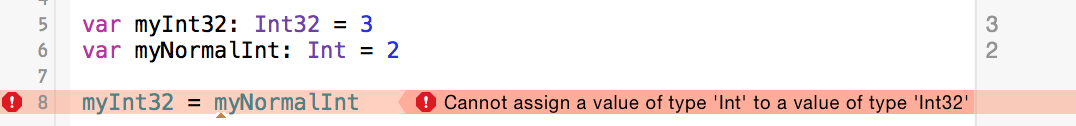
We also have the unsigned integer counterparts as well. Once again we should only be using these specific Integer type when the nature of our program forces us to. **We should use Int for all other cases and let the compiler decide how much memory to allocate when storing integers.**

print("The maximum value \(UInt32.max)")

print("The minimum value \(UInt32.min)")

## Swift is Strongly Typed

Int, Int32, UInt32 and all of the other variations of storing integers are all different types. What this means is that if we declare a constant or a variable to be of a specific Type, we can't assign a different type to it, even though they are just numbers.



**We want to use Int consistently throughout our code** so that we can leave the hard work to the compiler. We can convert other variations of the Int Type by using this syntax:

var myInt32: Int32 = 3

var myNormalInt: Int

// This will not error, because we first convert Int32 to instance of Int Type

myNormalInt = Int(myInt32)

# Floating Point Numbers

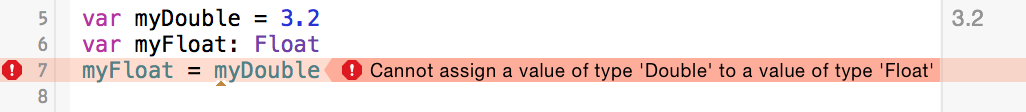
**There is a finite number of bits a computer has to represent an infinite amount of numbers**. So how would a computer represent something like 1/3 which is .3333333 with 3's repeating until forever? A computer will try to do its best. The most important concept in this tab is the **while Int's are precise, Floating Point Types are not**.

## Double vs. Float

Swift has two Types to represent numbers with decimals. The first is the **Double Type** which allocates 64 bits to represent a floating point number and the second is the **Float Type** which allocates 32 bits to represent a floating point number. Since a Double Type **has more representations available** to it, it will **store the floating point number more precisely**. Rather than the Int Type where the bits represented how many Int's it can represent, the extra bits for floating point numbers just represent its precision.

## Double Is the Inferred Type

If we store a floating point number to a variable or a constant **Swift will automatically assume that it is a Double**. Just like how we want to use Int's in our code as much as possible, we want to **avoid explicitly specifying "Float" whenever possible, so we can rely on Swift's type inference.**



## Operations

We can perform basic operations in Swift such as addition, subtraction, multiplication, and division.

print("Addition \(1 + 3)")

print("Subtraction \(1 - 3)")

print("Multiplication \(1 \* 3)")

print("Division \(1 / 3)")

When we divide an instance of the Int Type with another instance of the Int Type, we get another new instance of the Int Type. Then what will happen if we divide 1 with 3? We would expect the result to be something like .33333333.  **It results in zero because remember that Int's can only be whole numbers.**If there are decimal points then there is a different Data Type for that. If we divide two Int's together and its result should be represented in a decimal, Swift will just round the number down(even if it's 8.9999).

# Strings in Swift

A String is a data type containing a series of characters. We can make use of a string in many of the same ways as other languages and create them the same as any other data type.

//Normal Static String declaration

var name: String = "John Hancock"

//String can also be constants and make use of type inference

let PASSWORD = "SuP3rS3cRET!"

One interesting thing to note is that because Swift doesn't have semicolons to denote the end of a line, nor does it have the indentation rules of Python you may end up with a large block of text stored to a string all written to one line. Swift solves this by including the triple-quotes """ syntax to denote a block of text

//Though not the longest of text, it is already getting hard to look at

var Poem = "Alive without breath,As cold as death;Never thirsty, ever drinking,All in mail never clinking."

//This can be reformatted with triple-quotes to allow multiple lines!

Poem = """

Alive without breath,

As cold as death;

Never thirsty, ever drinking,

All in mail never clinking.

"""

**String Interpolation**

Swift's string interpolation lets us  **inject constants and variables into a new String**. This allows constructing instances of String Type a breeze. Anything inside of the parenthesis in \() gets evaluated and gets injected into the string. We can even put expressions inside of the parenthesis and Swift will evaluate it first and then inject it into the instance of the String Type.

var numberOfChampionships = 5

let name = "Kobe"

print("My favorite player is \(name) and he has \(numberOfChampionships) rings")

print("His jersey number is \(8 \* 3)")

**Conditionals**

 We use conditionals to **execute code based on a specific logical condition**. For example, let's say we are having a party for NBA Legends. We only want to allow players with at least 5 rings. We could write the logic something like this:

// Declare a variable called rings that is of the Int Type.

var rings = 5

if rings >= 5 {

print("You are welcome to join the party")

} else {

print("Go win some more rings")

}

If we have more than one condition we can add an *else if* statement:

var rings = 5

if rings >= 5 {

print("You are welcome to join the party")

} else if rings > 2 {

print("Decent...but \(rings) rings aren't enough")

} else {

print("Go win some more rings")

}

We used two comparison operators here. First, we checked if rings were greater than equal to 5. We also checked if rings were greater than 2. Here is a list of Swift's comparison operators:

**Comparison Operators**

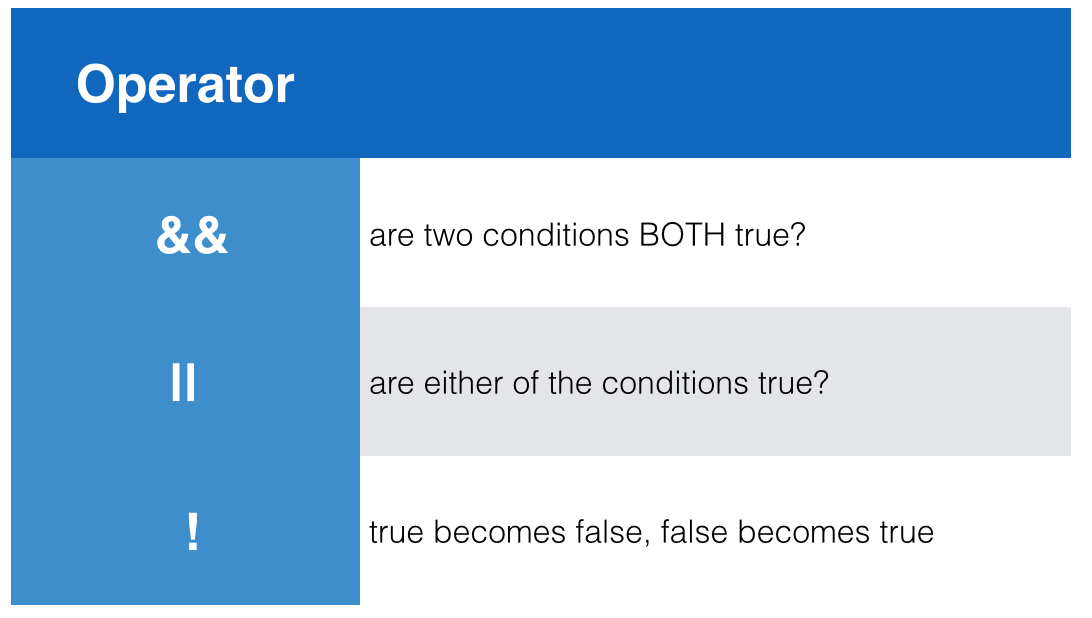
**صورة تحتوي على نص

تم إنشاء الوصف تلقائياً**

**Equality vs. Identity**

**There is a difference between equality and identity**. For example == checks whether the instances on the left and the right are equal. **Two instances can be equal, but they don't need to have the same location in memory**. The last two operators are called identity operators. We will be going over the differences more in depth once we start creating our own Data Types.

**Logical Operators**



We can use Logical Operators in our Conditionals as well. Let's say we want to change the criteria for entering our NBA Legends party. Let's say you have to have at least 5 rings AND have the name Kobe to enter the party:

var rings = 5

let name = "Kobe"

if rings >= 5 && name == "Kobe" {

print("Welcome to the party \(name), congratulations on your \(rings) rings")

}

We can change our criteria and say that you have to have at least 5 rings or have at least 3 All-Star appearances.

var rings = 5

var numOfAllStarAppearances = 17

if rings >= 5 || numOfAllStarAppearances > 3 {

print("Welcome you are truly a legend")

}

Or we can just let in everyone who is not crazy.

var crazy = true

if !crazy {

print("Let's party!")

}