**Functions**

A function is a **name that is related to a block of code we can execute**. To execute this code, we invoke it. A function is invoked when we**call its name, pass along the necessary input, and it will give us back some kind of output (most)**. Some functions take no input and some functions have no output. Even though a function doesn't return anything, it can do something inside the block of code that alters a program. For example:

func sayHello() {

print("Hello how are you doing today?")

}

Here we defined a function with a **func** keyword and declared its name to be *sayHello*, and specified that this function does not take in any arguments. When this function is invoked, it will print a greeting to the screen but it will not return anything.

sayHello()

We can invoke this function by **calling its name and passing in the necessary input**. In this case, the function requires no input so we won't pass anything in. This function doesn't return anything, but it will have a side effect of printing something to the **Assistant Editor**.

**Function Parameters**

We define what kind of input we want by **declaring parameters.** Let's declare a parameter named 'name' that will be of the String Type. Our function will still not return anything but its side effect will be a little bit more personalized. Note that we need to not only **declare a parameter name but also give the parameter a specific type.**

func sayHello(name: String) {

print("Hello, \(name), how are you doing today?")

}

We can invoke this function by calling its name and passing in our arguments for each of its parameters.

sayHello(name: "Andrew")

Wait, what's the difference between a parameter and an argument? These two words get mixed up a lot in programming. In this example, **'name' is a parameter**while **"Andrew" is an argument**. We **define parameters.** We **pass in arguments** into functions.

**Parameter Names**

Sometimes having an **external parameter name** helps make our functions more expressive. We can give external parameter names so that when we use the function there's a higher chance that we pass in the right arguments:

// The two parameters have external names of width and height

// and internal names of w and h

func printDescription(width w: Int, height h: Int) {

print("My width is \(w) and my height is \(h)")

}

// The func can be invoked by calling its name and passing in

// right arguments with corresponding external parameter names.

printDescription(width: 10, height: 20)

However, it is more Swift-like to leave out the external parameter name for the first argument and instead have the function name describe what the first parameter should be. **This convention is used a lot in Swift and is highly recommended in your own code.**

func printDescriptionWithWidth(w: Int, andHeight h: Int) {

print("My width is \(w) and my height is \(h)")

}

printDescriptionWithWidth(w: 10, andHeight: 20)

**Constants v. Variables**

When we pass in an argument to a function, our function **assigns the value to a constant** with the name of the **internal parameter name**. This becomes clear when we try to modify the constant.

var myInt = 1

func changeMyInt(someInt: Int) {

someInt = someInt + 1 // This throws an error because we are trying to modify a constant.

print(someInt)

}

If we want to modify the argument we have to **specify** that we want to store it in a **variable**.

var myInt = 1

func changeMyInt(someInt: Int) {

var someInt = someInt

someInt = someInt + 1

print(someInt)

}

However when we are passing in an argument and modifying it, its changes will not reflect outside of the function call for value types. We will be going over value types vs. reference types in the next chapter. Strings, Arrays, Dictionaries, Bool, and Ints are Value  Types. Classes and functions are reference types. Here's a quick peek at what's coming ahead. When we have a function that changes a value of an Int, its changes will not persist outside of the function:

var myInt = 1

func changeMyInt(someInt: Int) {

var someInt = someInt

someInt = someInt + 1

print(someInt)

}

changeMyInt(someInt: myInt) // => 2

print(myInt) // => 1

If we want to be able to make changes within a function to a specific instance of a Value type we need **in-out parameters**.

**In-out Parameters**

If we want to modify an argument and we want that change to appear on the outside of the function call, we need in-out parameters. There are two prerequisites to in-out parameters. First, we have to specify that our parameter is an in-out by following the parameter name with *in-out.*Next, when we invoke the function we have to pass in an argument with the '&' sign prefixed to it. Think of the '&' sign as **passing the memory address of the particular constant or variable** so that the function can go directly there and modify it so that the changes will persist outside of the function.

var myInt = 1

func changeMyInt(someInt: inout Int) { // must specify "inout"

someInt = someInt + 1

print(someInt)

}

changeMyInt(someInt: &myInt) // must pass in the variable with "&" symbol

print(myInt)

**Default Parameter Values**

It's good to have default parameter values sometimes. Let's revisit our sayHello function. If we don't know the name of the person that we are greeting we can have a default value to go with. For example, we can just say "Hey, buddy."

func sayHello(name: String = "buddy") {

print("Hey \(name)")

}

// We can call it without providing any arguments and the default value will be used...

sayHello()

// ...or we can call it with an argument and that argument's value will be used

sayHello(name: "Yoda")

**Return**

So far none of our functions had any tangible output. In many cases, we would want our function to return some sort of value that we can use later in our program. Let's modify our *sayHello* function and observe the differences:

func sayHello(name: String = "buddy") -> String {

return "Hey \(name)"

}

var greeting: String?

greeting = sayHello()

print(greeting)

This time, we are returning an instance of a String Type from our function. We have to specify in our function declaration that we are going to return an instance of a String Type. This is a **binding contract**, Swift will throw us an error when we construct our code in a way where an instance of a String Type might not return. Let's look at another example.

func calculateAreaOfRectangleWithWidth(w: Int, andHeight h: Int) -> Int {

return w \* h

}

var area = calculateAreaOfRectangleWithWidth(w: 10, andHeight: 3)

print(area)

Here, our function has two parameters that will both be an instance of Int Type and our function promises to return an instance of an Int Type. If we want to return a value in some cases, we just have to specify that our function will return an Optional Type. For example:

// We are declaring a function named lookForSomethingIn and declaring two parameters.

// The first parameter will be an instance of Dictionary, with keys being instances of String

// and values being instances of Int.

// The first parameter has no external name but it has an internal name of dictionary.

// The second parameter will be an instance of a String.

// It has an external name of forKey and an internal name of key.

// This function promises to return an Optional Type that can either be nil or hold an instance of Int.

func lookForSomethingIn(dictionary: [String: Int], forKey key: String) -> Int? {

if let value = dictionary[key] {

return value

} else {

return nil

}

}

var jerseyNumbers = ["Kobe": 24, "Curry": 30]

var jerseyNumber = lookForSomethingIn(dictionary: jerseyNumbers, forKey: "Kobe")

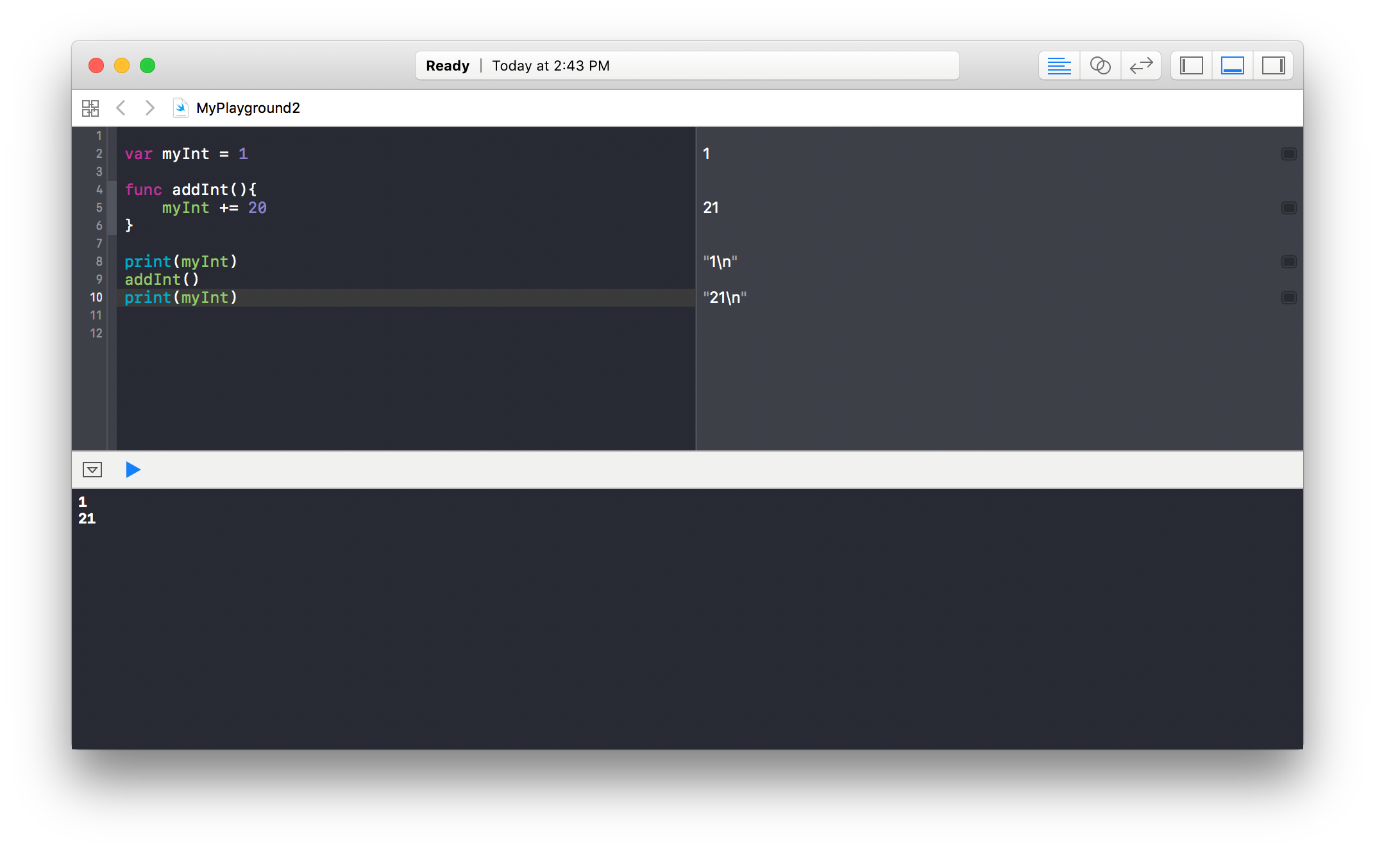
if let num = jerseyNumber {

print(num)

}

**Scope**

Scope describes the **visibility an instance or a function will have**. Anything within a function's scope will be visible to that function. Every time we type {} we can think of it as creating a new inner scope within the current scope. The inner scopes will have access to the outer scopes while outer scopes do not have access to inner scopes.



**Function Types**

So what's a function's type? Let's look at the following function:

func findMinOf(arr: [Int]) -> Int? {

if arr.count > 0 {

var min = arr[0]

for num in arr {

if num < min {

min = num

}

}

return min

} else {

return nil

}

}

This function takes in one argument of an instance of Array that holds a collection of Ints and returns an Optional Type that **may contain nil or contain a value of an instance of an Int**. This function is an instance of ([Int]) -> Int? type. The function's parameters are listed on the left side of the parentheses and the return type comes after the ->.

Let's look at another example. What type is this function?

func sayHello() {

print("Hello")

}

If we don't return anything we don't have to specify that we aren't returning anything in our function declaration. However, this function can be rewritten this way more explicitly.

func sayHello() -> Void {

print("Hello")

}

We can even have functions return functions! We can then indicate what that returned function's parameters and return type would be. For example, here we have a function where we purchase a bike, which returns the cycle function, which accepts an integer and returns a string.

func buyBicycle(cost: Int) -> (Int) -> String {

print("You just paid $\(cost) for your new bike!")

func cycle(miles: Int) -> String {

return "Have a great ride! See you in \(miles) miles!"

}

return cycle

}

// predict the output before running this code in your playground

print(buyBicycle(cost: 500)(30))

Sometimes, you will want your function to return multiple values. **Tuples,**which are key-value pairs, help with this! For example, let's say we would like to return the minimum, maximum, and average of an array of integers. The minimum and maximum would be integers, but the average should be a Double. Tuples are useful because they can hold different types! Access the value at a given position in a tuple with dot notation.

func maxMinAvg(arr: [Int]) -> (Int, Int, Double) {

    // your code here to determine max, min, and avg

    return (max, min, avg)

}

result = maxMinAvg(arr: [1,2,3,4])

print(result.1) // this will print the second value in the tuple, which in this case would be 1

You could even name your elements in a tuple to make it easier to reference them later.

func maxMinAvg(arr: [Int]) -> (max: Int, min: Int, avg: Double) {

    // your code here to determine max, min, and avg)

    return (max, min, avg)

}

let result = maxMinAvg(arr: [1,2,3,4])

print(result.max) // this will print 4

There's a lot to learn with functions. If you want to learn more go to the  [Functions](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html#//apple_ref/doc/uid/TP40014097-CH10-ID158) documentation.

**Function Overloading**

Since functions are identified by their types and not their names, we can actually define two functions with the same name. This is called function overloading. This lets us create different functionality for different input types, or group similar functions together under the same name.

func info(input: Int){

if input > 100 {

print("This is a large number")

} else {

print("This is a small number")

}

}

func info(input: [Int]){

if input.count > 100 {

print("This is a long array")

} else {

print("This is a short array")

}

}

info(input: 150) // This is a large number

info(input: [1,2,3]) // This is a short array

<https://docs.swift.org/swift-book/LanguageGuide/Functions.html#//apple_ref/doc/uid/TP40014097-CH10-ID158>

# Optionals

Optionals are an extremely important part of Swift programming. **Previously in Objective-C, any variable could hold a nil value.**This makes life easier for the developer in many cases but can also lead to many errors when the developer forgets to handle the nil case. **Swift forces the developer to be cautious of potential nil values through the use of Optionals.**An optional is a type that could either hold nil or a value of a specific type. Let's see an example:

var name: String = "Jay" // This is a String type

var pet: String? // This is an Optional String Type

// Right now the val of pet is nil (we've not given it a val, so it has none).

pet = "Skippy"

We know two things about an instance of an  Optional Type. **It either has a value or it has no value or nil**. For example, an empty string "" is different from nil. There still is the value from "". **When it is nil it means there's nothing there.** *Optionals* signal that an **instance may not have a value**. It warns developers that the value may **potentially be nil**. And **nil is very bad**. When we expect an instance to have a value and it doesn't, our application will crash. As iOS developers, we have to be very cautious of nil and **Optionals help us be better programmers and make Swift a safer language.**

var xFactor: String?

Here we declared the variable  xFactor to be of an **Optional Type that may contain an instance of a String Type**. We are declaring this as an optional because xFactor might potentially be nil and we should not assume xFactor to contain a value in our program. Let's store a value to this Optional Type and try to print it out.

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

We have to first unwrap the Optional before we can use it. It's like a present. We first make sure that it isn't empty by shaking it around. If we hear something then we safely unwrap the present.

Doing something like this is very common in Swift. If an Optional Type is not nil then do something. In fact, this is so common that there is a special syntax to make this process easier. This syntax is called **Optional Binding**.

var present: String? = "Apple Watch"

// If we can let "unwrappedPresent" equal "present" meaning there is something inside of present then go on to do something with "unwrappedPresent"

if let unwrappedPresent = present {

print("OMG THANK YOU FOR THE \(unwrappedPresent)")

}

## Force Unwrapping

If we are 100% confident that the optional has a value and does not hold nil we can use shorthand syntax to unwrap the optional like so:

var present: String? = "Apple Watch" // If we can let "unwrappedPresent" equal "present"

// (meaning there is something inside of present)

// then go on to do something with "unwrappedPresent"

print("OMG THANK YOU FOR THE \(present!)") // Force unwrap using the "!"

We can force-unwrap optionals using the "!" operator. Note that this should only be done when we are absolutely sure that the optional holds a value (usually right after the optional has a value assigned to it).

## Implicitly Unwrapped Optionals

Implicitly unwrapped optionals are a little weird. **They are still of the Optional Type**: it can either contain a value or it is nil. The difference is that **we don't need to unwrap them**. With great power comes great responsibility. If we try to access the value of an implicitly unwrapped optional, it will result in a runtime error if it does not have a value.

let possibleDevice: String? = "Apple Watch"

let watch: String = possibleDevice! // we have to have the exclamation point

let assumedDevice: String! = "iPhone"

let iphone:String = assumedDevice // no need to unwrap

print(iphone)

copy

Implicitly unwrapped optionals are reserved for special cases. We will go over them more in depth in a later chapter.

# Optionals Continued

#### Dog Day Care

Imagine we have a pet training (dog day care business) where we keep information about the dogs that are learning how to behave properly.

// a dog named Bailey training at our dog day care business

var name = "Bailey"

var age = 3

var quirks = "Loves chewing sticks and getting into mischief"

var photo = UIImage()

Storing information like this works great, until our dog, (Bailey in this case), graduates from puppy school and no longer visits the trainer. Our dog Bailey will never return to the kennel, and we **don't need this information anymore**.

In this case, it would be useful to have **absence of value.** The name Bailey with age and quirks and photo **should not exist** anymore.

Or, maybe our dog is private and doesn't share images to our dog day care business. In this case, the photo **shouldn't be set to an empty string**. It could be set to **nil**, or the absence of value.

* nil would be better than a blank image or white background

Of course, we could always set the name, age, and quirks to an empty string. But, **optionals** are a much better way to do this!

### An organized and powerful way to handle no value

In Swift, we use **optionals** to represent the possibility of no value at all, or **nil**.

If a variable contains no value, it is **nil**, regardless of the type of the variable.

If it's declared as an optional, it doesn’t matter whether it’s a image, a string, object, or a button – if it’s declared as an optional, it can be nil.

#### Waiting for a Web-service

Take for example that we may have properties of a struct that are nil **until values from a back-end web-service or database have been retrieved**

A view controller outlet property, which is nil before the view controller has been loaded. Until it’s loaded, you can’t use it.

#### Physical World Example

Think of an **optional** as a gift in a box that could be a "gag" gift. The box which holds the possible gift exists, but the contents inside may empty.

There's one catch that's worth noting:

**You need to unwrap optionals to safely use them**

## Declaring Optionals

let name: String? = nil

print(name)

On the first line, you’re creating a constant name of type String?

The question mark ?, written right after the type annotation, indicates that this constant is an **optional**.

You assign name to nil, so this constant does not have a value. It’s nil. We say that name “is an optional”.

## Unwrapping Optionals

You can unwrap optionals in 4 different ways:

1. With **force unwrapping**, using !
2. With **optional binding**, using if let
3. With **implicitly unwrapped optionals**, using !
4. With **optional chaining**, using a?.b?.c

## Force Unwrapping (1)

We use force unwrapping if we are absolutely confident that we have a value.

Remember our "physical world example" of the opening the gift? **Force unwrapping** is like quickly opening the gift in the box without shaking it to see if something is in there.

We only **force unwrap** if we know there's no chance of the value being nil.

## Optional Binding (2)

One alternative to force unwrapping is optional binding!

* a conditional and a constant is being used
* constant is only available for use **within** the conditional body

## If Let

combining multiple if let statements

You can use optional binding for any optional value, so also for a function that returns an optional:

if let favoriteNumber = Int("34") {

print(favoriteNumber)

}

Because the conversion of string to integer might fail, its return type is an optional.

## Guard

We can use guard to guard against errors crashing our app. We preface our code with guard and only run the code if the guard passes.

Imagine a function that uses an API to get current weather data. That weather fetch **might fail if the network is down**. We use an **optional**, which has a value if the fetch succeeds, and nil otherwise.

Sometimes you want to check a condition and **only continue executing a function** if the condition is true.

Guard is useful because the variable is not declared inside the scope, like with if let.

The block of code covered by the else will execute if the condition is false.

guard let password = passwordTextField.text else {

// this block of code must return

return

}

print("User signed up with password: \(password)")

The guard statement helps to make our code more readable and understandable.

## Implicitly Unwrapped Optionals (3)

Implicitly unwrapped optionals are **optionals**, so they can be nil, but you don’t have to unwrap them to access their value.

**When you access them, they’re implicitly and automatically unwrapped.**

We need to be certain that implicit optionals are not nil.

You can do this by declaring the outlet property as an implicitly unwrapped optional, like this:

@IBOutlet var passwordField:UITextField!

* Take a look at the exclamation mark ! after the property type.
* The ! signals that this is an implicitly unwrapped optional, just like a ? signals an optional.

An **implicitly unwrapped optional** is like giving us the power to unwrap an optional automatically.

The benefit for us is that we don't have to unwrap every time we reference a UI component in our code.

## Optional Chaining (4)

### Without Optional Chaining

Let's take the example of a **textfield** in our view controller.

Text Fields have a **text** property.

* Both the textfield and the text are **optionals**

**How do we unwrap these values?**

You could do that with optional binding, like this:

if let field = usernameTextField,

let text = field.text {

print("Logging in user with username: \(field.text)")

}

### With Optional Chaining

To avoid all of this mess, you can use optional chaining. Here’s an example:

usernameTextField?.text = "cool\_swift\_dev"

The variable usernameTextField is an optional. See how there’s a question mark ? right after it, but before the . period sign? This is optional chaining.

At this point, one of two things happens:

1. The variable usernameTextField **is not** nil, and the call to property text succeeds
2. The variable usernameTextField **is** nil, and the call to property text fails

When the call fails, execution of that line stops gracefully. In the above example, the text property of usernameTextField isn’t changed when usernameTextField is nil.

You can chain these calls together, like this:

car?.airCondition?.rightVent?.power = "90%"

## Nil Coalescing

There is another easy way to **unwrap an optional**.

What if we wanted to get a value out of an optional no matter what?

In the case of nil, we would use a **default value**. This is called **nil coalescing**:

* The nil coalescing operator ??

#### Without using the nil coalescing operator:

var optionalInt: Int? = 10

var mustHaveResult: Int

if let unwrapped = optionalInt {

mustHaveResult = unwrapped

} else {

mustHaveResult = 0

}

#### Using the nil coalescing operator:

var optionalInt: Int? = 10

var mustHaveResult = optionalInt ?? 0

* mustHaveResult is either set to the optionalInt if it has a value, otherwise, it's set to 0 if the value of optionalInt is nil.

# Closures

*Closures are powerful and used all over Cocoa Touch / UIKit.*

We will be using closures to perform **completion handlers** for many different situations, such as loading data from an external API. It is important to understand how to build our own closures alongside using them all over UIKit

* closures are commonly used in swift, but challenging for beginners to understand
* closures are functions that can be passed around (a lambda in other languages)

Simply, what are closures?

**Closures are variables that hold code**

* Remember that while an integer holds the number 300, closures hold lines of swift code

Closures also **capture the environment** where they are created! This means that they can take a copy of the values used inside of them.

UIKit asks you to write closures to match its needs - so we need to understand how they work.

## Closure Example #1 - UIKit

Let's take a look at an example inside of Xcode using UIKit:

let myView = UIView()

UIView.animate(withDuration: 0.5) {

myView.alpha = 0

}

// If the last parameter to a method takes a closure, you can eliminate that parameter and instead provide it as a block of code inside the braces, like we did in this example

UIView is an iOS datatype in UIKit that represents the most basic type of UI Container.

The UIView class has a static method called animate that let's you change the way the interface looks using animation

* we describe what's changing and for how many seconds (withDuration)

The animate method takes two parameters in that code

* number of seconds to animate over
* a **closure** containing the code to be executed as part of the animation

In this example, I have decided to make the opacity completely transparent or alpha = 0. UIKit takes a copy of the code in our closure and stores it away and it runs our code when it's ready.

### Capturing the Environment

Closures **capture** their environment. Here is an example to illustrate this:

let myView = UIView()

UIView.animate(withDuration: 0.5) {

myView.alpha = 0

}

In my code above, I declared the myView constant outside the closure, and used it inside the closure.

* Swift detects this, and **makes that data available inside the closure too.**

### Capturing the Environment

Automatically capturing a closure's environment is helpful. The only issue we sometimes run into is if some object (Object A), stores a closure as a property, and that property also references object A. This would lead to a **Strong reference cycle** or **retain cycle** - which leads to unhappy users due to bad performance.

## Closure Example #2 - Functions as Types

* Remember, a closure is a **function that can be passed around**

Let's create a function to demonstrate this with a closure.

This function isGreaterThanThree performs a basic check to see if the argument passed in is greater than three.

* if it is, return true
* If not, return false

func isGreaterThanThree(number: Int) -> Bool {

if number > 3 {

return true

}

return false

}

// this function takes in an Int, and returns a Bool

Let's create a variable called myFunction below

Let's take a look at the signature of our function: The function isGreaterThanThree takes in an Int and returns a Bool.

This can be represented as a **type** of taking in an Int and returning a Bool

var myFunction: ((Int) -> Bool)

#### How can a function be a type?

A function is just a **named set of code**, (that may also take in parameters and return something).

Why can't we define our isGreaterThanThree as something in a variable? **We can, and this is what a closure is.** Check it out below:

var myFunction: ((Int) -> Bool) = { number in

if number > 3 {

return true

}

return false

}

print(myFunction(34))

Now, myFunction variable is exactly equivalent to the function isGreaterThanThree

### What's the difference between the following statements?

print(myFunction(34))

print(isGreaterThanThree(number: 34))

**Answer** - While one is a function that is defined, the other is **a function assigned to a variable.**

The power of this is this is that we can pass variables around to other functions. Creating this **function chaining** becomes very powerful.

We call a function and pass in a parameter, **but that parameter is another function, but represented as a variable... aka a closure!**

#### Let's flesh out this closure code a bit more:

var myFunction: ((Int) -> Bool) = { number in

if number > 3 {

return true

}

return false

}

print(myFunction(34))

The **type** is a function and the function is ((Int) -> Bool). It takes one parameter of the Int type (unnamed) and it returns a Bool.

To assign it, we create the function body, and for each of the parameters, we add them there. In this case, that looks like number in. You can call the parameter whatever you want. If there are multiple, they should be comma separated such as number,anotherNumber in

For the parameters **in** this block, we can use them here:

if number > 3 {

return true

}

return false

While we can call myFunction like any other function, such as myFunction(34), we know that **the result is Bool**

Of course, write out the code yourself and print out the result...

let result = myFunction(34)

print(result)

## Closure Example #3 - Filtering Numbers From a List

Let's take what we know about closures and apply it to Arrays. In this example, we will:

* learn about passing a function as a parameter
* practice assigning a function to a variable

### Before using closures (more work, less power)

With the following code, we will go over how to **filter a set of numbers given a condition** without using closures.

We will give our function an array of numbers, and ask for **all the numbers greater than 3 (the condition)** to be returned in an array.

Here's the function that we plan to implement:

func filterGreaterThanValue(value: Int, numbers: [Int]) -> [Int] {

return []

}

let filteredList = filterGreaterThanValue(value: 3, numbers: [1,2,3,4,5,10])

Now in the following code, we filter the array of numbers, and anything greater than 3, we should get back in our list.

func filterGreaterThanValue(value: Int, numbers: [Int]) -> [Int] {

var filteredSetOfNumbers = [Int]()

for num in numbers {

if num > value {

filteredSetOfNumbers.append(num)

}

}

return filteredSetOfNumbers

}

let filteredList = filterGreaterThanValue(value: 3, numbers: [1,2,3,4,5,10])

This code works. But, what if we wanted to get numbers that are less than 5?

We would have to change our value and **we would have to go into our function and change the greater-than sign to less-than**. Then, we would change the function name to filterLessThanValue

### Using Closures

A better way to do this is with **closures**.

It would be better to pass in a predicate and then allow the predicate to do the checking for less than or greater than.

Let's write out a new function:

func filterWithPredicateClosure(closure: (Int) -> Bool, numbers: [Int]) -> [Int] {

return []

}

Let's put in the logic and then use the closure:

func filterWithPredicateClosure(closure: (Int) -> Bool, numbers: [Int]) -> [Int] {

var filteredNumbers = [Int]()

for num in numbers {

// perform a condition check

if closure(num) {

filteredNumbers.append(num)

}

}

return filteredNumbers

}

let filteredLst = filterWithPredicateClosure(closure: { num in

return num < 5

}, numbers: [1,2,3,4,5,10])

print(filteredLst)

Now, let's take it a step further by passing in a function as a parameter:

func filterWithPredicateClosure(closure: (Int) -> Bool, numbers: [Int]) -> [Int] {

var filteredNumbers = [Int]()

for num in numbers {

// perform a condition check

if closure(num) {

filteredNumbers.append(num)

}

}

return filteredNumbers

}

func greaterThanThree(value: Int) -> Bool {

return value > 3

}

// pass in greaterThanThree as a function

let filteredList = filterWithPredicateClosure(closure: greaterThanThree(value:), numbers: [10,5,1,2,0])

print(filteredList)

Check out another example below. Instead of filtering for greater than a number, we will check for numbers divisible by 5.

func filterWithPredicateClosure(closure: (Int) -> Bool, numbers: [Int]) -> [Int] {

var filteredNumbers = [Int]()

for num in numbers {

// perform a condition check

if closure(num) {

filteredNumbers.append(num)

}

}

return filteredNumbers

}

func divisibleByFive(value: Int) -> Bool {

return value % 5 == 0 // remainder of divsion of 5. if 0, it is divisble

}

let filteredList = filterWithPredicateClosure(closure: divisibleByFive(value:), numbers: [20,30,1,2,9,15])

print(filteredList)

Check out the results! You should be given back [20, 30, and 15]

## Closure Summary

Wahoo! If you step through this code carefully, you're on your way to using something very powerful – closures!

* With closures, our code is more robust and more manageable

# Weak and Strong References

You'll notice weak and strong references being used all over UIKit!

## Strong Vs Weak

**Strong** vs **Weak** is how memory is managed for objects. With iOS Development, we used to have to do this manually, but now we can mark things weak or (implicitly strong), and the system manages it for us.

#### Why do we need weak?

Imagine a parent object and a child object both pointing to each other. If we try to **deallocate** those two objects from memory, we cannot because both of these are pointing to another object.

This is where **weak** comes in!

Weak allows us to say that if the parent goes away, the child goes away. But, if the child goes away, the parent still exists!

Back in the day, we didn't have ARC - Automatic Reference Counting. But now that we do, it makes life much easier.

#### A quick illustration:

Imagine a person holding a balloon. In this analogy, the person is the strong object, and the balloon is the weak object. If the person lets go of the balloon, the balloon disappears.

Using this analogy as an example, understand that the **weak object** is at the mercy of the **strong object** that is pointing to it.

#### What if two objects are strongly pointing to each other? A memory leak is created...

In this hypothetical situation, the balloon needs the person to hold it, and the person needs the balloon. When the person lets go of the balloon, it **wouldn't** float away. This is a retain cycle or a memory leak!

## When to use it

In Swift, **strong** references are the default, so to make a reference **weak**, you can use the weak keyword.

ARC only affects reference types, like classes. It does not affect value types, like structs.

As we already discussed, value types are copied when they’re passed/referenced, so beyond the local scope there’s nothing to retain in memory.

Keep in mind that a weak property must be an optional, as well. Otherwise it cannot be nil.

Examples of places we see **weak**used in iOS Development:

1. **Delegates.** A delegate property is weak by design, because it would cause a strong reference cycle otherwise. A table view controller’s dataSource property is made weak. If it would be strong, the table view controller and data source would strongly hold onto each other and cause a memory leak.
2. **Closures.** A closure can capture variables, properties, etc. from its surrounding scope. When the closure outlives the scope it was declared in, it can strongly hold onto objects from that scope. This can cause a strong reference cycle.

*We learned that****closures****are self-contained blocks of code you can pass around.*

A closure can capture values from their surrounding scope. We can still use variables and properties outside of the closure, even though a closure may be used somewhere else in your code.

#### Let's take an example in UIKit to illustrate this:

let imageView:UIImageView =

let task = session.dataTask(with: , completionHandler: { data, response, error in

imageView.image = UIImage(data: data)

})

In the above code, we have the following:

(The **closure** is the stuff between the curly braces)

* An image view
* an asynchronous URLSession data task that downloads an image
* a closure that will load and display the image **upon completion** of the download task

Take notice of the **completion handler**. It gets called by the URLSession component when the image has finished downloading.

The closure is passed into the URLSession component, stored there, and called at a later time. That’s how closures work!

#### What does capturing mean?

What does **capturing** mean in the example above with the image view?

A reference to the UIImageView instance – assigned to imageView – is **captured** by the closure. It means that the UIImageView instance is available for use **after** the scope that surrounds the closure (i.e., a function) is gone.

When a closure captures a reference type from its surrounding scope, the closure creates a **strong reference** to the captured instance. You can create a strong reference cycle between a closure and an instance when you manage memory poorly. We can use weak to break this cycle

<https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html>

<https://developer.apple.com/documentation/swift/array/2296815-sorted>

Retain cycle

ARC Automatic Reference Counting. Like I said it keeps ‘counting’ how many “Strong” reference an object have, if 0 it deallocate it