//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CLOSURES\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// The following are Closures Examples that Broken down into steps and then practical examples

/\* Syntax for a closure:

{(parameters) -> return type in

statements

}

Step 1) Omit the func keyword and the function name.

Step 2) Take the curly brace from behind the return type and move it to the front

and then place the "in" keyword where the opening curly brace used to be.

// Here we would satisfy the requirements of a Closure, but we can further refine this down for

code elegance.

Step 3) Because of compiler inference we can remove the the argument types from both

\*/

// The example for the need of a closure - Imagine we were making a small calculator app and we wanted to program the default arithmetic functions in out calculator. By convention we would do the following:

func multiplyTwoNums(num1: Int, num2: Int) -> Int {

return num1 \* num2

}

func addTwoNums(num1: Int, num2: Int) -> Int {

return num1 + num2

}

/\* .... And so on with the functions as you know what they are we did not finish them all.

This method does work but we can do one better as Swift allow us to pass a function into a function and also return a function. So, lets see how we can refine this...

\*/

func calculateTwoNums(num1: Int, num2:Int, operation: (Int,Int)->Int) ->Int {

return operation(num1, num2)

}

// With the code above, we create a function that takes a function as one of its' parameters, it is explicit in the prototyping of that parameter that it must adhere to the protocol that the function that it takes in must take in two Ints as its parameter arguments and then return an Int. Now look below how we can use this function...

let finalNumber = calculateTwoNums(num1: 8, num2: 5, operation: addTwoNums)

// console logs out 13 as the value for "finalNumber" constant

// Important to notice that we call the "addTwoNums" function as part as the "operation:" argument. We just need to use the name we do not need the parentheses as part of the function call. This works fine and sums the two numbers and then outputs then and assigns then to a variable here, but we can refine multiple lines of codes down even further as follows...

// So Far, we have NOT used a Closure on passed a function into another functions via parameter arg. Lets use that Closure syntax to do so here we will Closurefy the "addTwoNums" function we used earlier.

/\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Original Function

func addTwoNums(num1: Int, num2: Int) -> Int {

return num1 + num2

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 1:

(num1: Int, num2: Int) -> Int {

return num1 + num2

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 2: Here we could actually pass this into the parameter argument, and it would work with no issues

{ (num1: Int, num2: Int) -> Int in

return num1 + num2

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 3: Remove the data types in the parameter arguments and then remove the return type from the end and have the compiler infer the data types

{ (num1, num2) in

return num1 + num2

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 4: get rid of return keyword in the body of the statement and the compiler will infer that we want some value returned.

{ (num1, num2) in

num1 + num2

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 5: We can use the anonymous operator "$" (where "$0" would be the first argument) in the closure statements which since they are passed and used in the body type the closure refines to this..

{$0 + $1}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

// ONE LAST THING YOU MUST USE TO SEE which is in SWIFT that if the last parameter in a function call is a closure then you can close off the other parameters and add the closure to the end of the function call. This is called a TRAILING CLOSURE.. which is what we normally see. See the following

let productFinalNum0 = calculateTwoNums(num1: 10, num2: 33, operation: {$0\*$1})

//assigns the product of num1 and num2 which is 330 to the "productFinalNum0" const

// Now lets see that same statement with a TRAILING CLOSURE:

let productFinalNum1 = calculateTwoNums(num1: 10, num2: 33) {$0\*$1}

//assigns the product of num1 and num2 which is 330 to the "productFinalNum1" const

/\*

SO, WE CUT OUT CODE DOWN FROM SOMETHING THAT LOOKED LIKE THIS:

calculateTwoNums(num1: 10, num2: 33, operation: { (num1: Int, num2: Int) -> Int in

return num1 + num2

})

TO THIS:

calculateTwoNums(num1: 10, num2: 33) {$0 + $1}

LESS CODE BUT A LOT GOING ON BEHIND THE SCENE THANKS TO SWIFT'S COMPILER (INFERENCE OF TYPES, TRAILING CLOSURE RULE, INFERENCE OF RETURN TYPE, ANONYMOUS OPERATORS)

\*/

//Example the map function

let oddArray = [1,3,5,7,9]

//let say that we want to add 1 to the oddArray and make all the numbers even. We can use the map function along with our Closure syntax all the way refined to do this in one line of code.

let evenArray = oddArray.map({$0 + 1})

// evenArray is now = [2,4,6,8,10]

// So simple and we didn't even have to write a add function or loop through the array