




3RD EDITION

Swift Programming

THE BIG NERD RANCH GUIDE

Mikey Ward



Swift Programming

THE BIG NERD RANCH GUIDE

Mikey Ward



Big Nerd Ranch

In-out parameters

Sometimes there is a reason to have a function modify the value of an argument. *In-out parameters* allow a function's impact on a variable to live beyond the function's body.

Say you have a function that will take an error message as an argument and will append some information based on certain conditions. Enter this code in your playground.

Listing 12.6 An in-out parameter

```
...
var error = "The request failed:"
func appendErrorCode(_ code: Int, toErrorString errorString: inout String) {
    if code == 400 {
        errorString += " bad request."
    }
}
appendErrorCode(400, toErrorString: &error)
print(error)
```

The function `appendErrorCode(_:toErrorString:)` has two parameters. The first is the error code that the function will compare against, which expects an instance of `Int`. Notice that you gave this parameter an external name of `_`, which has a special meaning in Swift. Using `_` in front of a parameter name will suppress the external name when calling the function. Because its name is already at the end of the function name, there is no reason for the parameter name to be used in the call.

The second is an `inout` parameter – denoted by the `inout` keyword – named `toErrorString`. This parameter expects an instance of `String` as its argument. `toErrorString` is an external parameter name used when calling the function, while `errorString` is an internal parameter name used within the function.

The `inout` keyword is added prior to `String` to express that the function expects to modify the original value. It does this by taking as its argument not a copy of the passed-in value, but a reference to the original. This way, any changes it makes to the string affect the original string, and those changes will remain after the function is done executing.

When you call the function, the variable you pass into the `inout` parameter must be preceded by an ampersand (&) to acknowledge that you are providing shared access to your variable instead of just a copy of it and that you understand that the variable's value may be directly modified by the function. Here, the function modifies `errorString` to read `The request failed: bad request.`, which you should see printed to the console.

Note that in-out parameters cannot have default values. Also, in-out parameters are not the same as a function returning a value. Lastly, because in-out parameters grant shared access to a variable, you cannot pass a constant or literal value into an in-out parameter. If you want your function to produce something, there is a more elegant way to accomplish that goal.

Returning from a Function

Functions can give you information after they finish executing the code inside their implementation. This information is called the *return* of the function. In fact, this is often the purpose of a function: to do some work and return some data. Make your

divisionDescriptionFor(numerator:denominator:withPunctuation:) function return an instance of the **String** type instead of simply printing a string to the console.

Listing 12.7 Returning a string

```
...
func divisionDescriptionFor(numerator: Double,
                           denominator: Double,
                           withPunctuation punctuation: String = ".") -> String {
    print("\(numerator) divided by \(denominator) is
        \(numerator / denominator)\(punctuation)")

    return "\(numerator) divided by \(denominator) is
           \(numerator / denominator)\(punctuation)"
}
divisionDescriptionFor(numerator: 9.0, denominator: 3.0)
divisionDescriptionFor(numerator: 9.0, denominator: 3.0, withPunctuation: "!")
...
```

The behavior of this new function is very similar to your earlier implementation, with an important twist: This new implementation returns a value to the code that called it. This *return value* is denoted by the `->` syntax at the end of the function signature, which indicates that the function will return an instance of the type that follows the arrow.

Your function returns an instance of the **String** type. The `return` keyword tells the program “Stop executing this function and resume the calling code where it left off.” If there is a value to the right of the `return` keyword, that value will be handed back to the calling code. The type of this value must be the same as the declared return type of the function.

When there is no value to return to the caller, a function will implicitly return at the end of its *scope* (at the closing curly brace that ends the function body – more on scope in just a moment). This is why your previous functions have not needed to explicitly return so far. You will learn more about implicit and explicit returns from functions in Chapter 13.

Because your **divisionDescriptionFor(numerator:denominator:withPunctuation:)** function no longer contains a call to **print()**, your calls to it no longer produce console output. But it returns a **String**, and **print()** accepts **String** arguments – so you can call your division function nested within a call to **print()** to log the string instance to the console.

Listing 12.8 Nesting function calls

```
...
print(divisionDescriptionFor(numerator: 9.0, denominator: 3.0))
print(divisionDescriptionFor(numerator: 9.0, denominator: 3.0, withPunctuation: "!"))
```

When one function call is nested within another like this, they are executed from the innermost function to the outermost. In this case, **divisionDescriptionFor(numerator:denominator:withPunctuation:)** will be executed by the program first, and then its **String** return value will be passed as the argument to **print()**.

Nested Function Definitions and Scope

Swift's function definitions can also be nested. Nested functions are declared and implemented within the definition of another function. The nested function is not available outside the enclosing function. This feature is useful when you need a function to do some work, but only within another function. Let's look at an example.

Listing 12.9 Nested functions

```
...
func areaOfTriangleWith(base: Double, height: Double) -> Double {
    let rectangle = base * height
    func divide() -> Double {
        return rectangle / 2
    }
    return divide()
}
print(areaOfTriangleWith(base: 3.0, height: 5.0))
```

The function **areaOfTriangleWith(base:height:)** takes two arguments of type **Double**: a base and a height. It also returns a **Double**. Inside this function's implementation, you declare and implement another function called **divide()**. This function takes no arguments and returns a **Double**. The **areaOfTriangleWith(base:height:)** function calls the **divide()** function and returns the result.

The **divide()** function above uses a constant called **rectangle** that is defined in **areaOfTriangleWith(base:height:)**. Why does this work?

Anything within a function's braces (**{}**) is said to be enclosed by that function's scope. In this case, both the **rectangle** constant and the **divide()** function are enclosed by the scope of **areaOfTriangleWith(base:height:)**.

A function's scope describes the visibility an instance or function will have. It is a sort of horizon. Anything defined within a function's scope will be visible to that function; anything that is not is past that function's field of vision. **rectangle** is visible to the **divide()** function because they share the same *enclosing scope*.

On the other hand, because the **divide()** function is defined within the **areaOfTriangleWith(base:height:)** function's scope, it is not visible outside it. The compiler will give you an error if you try to call **divide()** outside the enclosing function. Give it a try to see the error.

By the way, nearly any pair of braces in Swift defines a scope. For example, switches, loops, and conditionals define scopes of their own.

divide() is a very simple function. Indeed, **areaOfTriangleWith(base:height:)** could achieve the same result without it: `return (base * height) / 2`. The important point here is how scope works. You will see a more sophisticated example of nested functions in Chapter 13. Stay tuned!

Multiple Returns

Functions can only return one value – but they can *pretend* to return more than one value. To do this, a function can return an instance of the tuple data type to encapsulate multiple values into one.

Recall that a tuple is an ordered list of related values. To better understand how to use tuples, write a function that takes an array of integers and sorts it into arrays for even and odd integers.

Listing 12.10 Sorting evens and odds

```
...
func sortedEvenOddNumbers(_ numbers: [Int]) -> (evens: [Int], odds: [Int]) {
    var evens = [Int]()
    var odds = [Int]()
    for number in numbers {
        if number % 2 == 0 {
            evens.append(number)
        } else {
            odds.append(number)
        }
    }
    return (evens, odds)
}
```

Here, you first declare a function called `sortedEvenOddNumbers(_:)`. You specify this function to take an array of integers as its only argument. The function returns a *named tuple*, so called because its constituent parts are named: `evens` will be an array of integers, and `odds` will also be an array of integers.

Next, inside the implementation of the function, you initialize the `evens` and `odds` arrays to prepare them to store their respective integers. You then loop through the array of integers provided to the function's parameter, `numbers`. At each iteration through the loop, you use the `%` operator to see whether `number` is even. If the result is even, you append it to the `evens` array. If the result is not even, the integer is added to the `odds` array.

Now that your function is set up, call it and pass it an array of integers. (As usual, do not break the string passed to `print()` in your code.)

Listing 12.11 Calling `sortedEvenOddNumbers(_:)`

```
...
func sortedEvenOddNumbers(_ numbers: [Int]) -> (evens: [Int], odds: [Int]) {
    var evens = [Int]()
    var odds = [Int]()
    for number in numbers {
        if number % 2 == 0 {
            evens.append(number)
        } else {
            odds.append(number)
        }
    }
    return (evens, odds)
}

let aBunchOfNumbers = [10,1,4,3,57,43,84,27,156,111]
let theSortedNumbers = sortedEvenOddNumbers(aBunchOfNumbers)
print("The even numbers are: \(theSortedNumbers.evens);
      the odd numbers are: \(theSortedNumbers.odds)")
```

First, you create an instance of the **Array** type to house a number of integers. Second, you give that array to the **sortedEvenOddNumbers(_:)** function and assign the return value to a constant called **theSortedNumbers**. Because the return value was specified as **(evens: [Int], odds: [Int])**, this is the type the compiler infers for your newly created constant. Finally, you log the result to the console.

Notice that you use string interpolation in combination with a tuple. You can access a tuple's members by name if they are defined. So, **theSortedNumbers.evens** inserts the contents of the evens array into the string logged to the console. Your console output should be **The even numbers are: [10, 4, 84, 156]; the odd numbers are: [1, 3, 57, 43, 27, 111]**.

Optional Return Types

Sometimes you want a function to return an optional. When a function might need to return **nil** but will have a value to return at other times, Swift allows you to use an optional return.

Imagine, for example, that you need a function that looks at a person's full name and pulls out and returns that person's middle name. For the purposes of this exercise, assume that everyone has a first name and a last name (though this is not an assumption you would necessarily make in a production app). But not all people have a middle name, so your function will need a mechanism to return the person's middle name if there is one and return **nil** otherwise. Use an optional to do just that.

Listing 12.12 Using an optional return

```
...
func grabMiddleName(fromFullName name: (String, String?, String)) -> String? {
    return name.1
}

let middleName = grabMiddleName(fromFullName: ("Alice", nil, "Ward"))
if let theName = middleName {
    print(theName)
}
```

Here, you create a function called **grabMiddleName(fromFullName:)**. This function looks a little different than what you have seen before. It takes one argument: a tuple of type **(String, String?, String)**. The tuple's three **String** instances are for the first, middle, and last names, and the instance for the middle name is declared as an optional type.

The **grabMiddleName(fromFullName:)** function's one parameter is called **name**, which has an external parameter name called **fromFullName**. You access this parameter inside the implementation of the function using the index of the name that you want to return. Because the tuple is zero-indexed, you use **1** to access the middle name provided to the argument. And because the middle name might be **nil**, the return type of the function is optional.

You then call **grabMiddleName(fromFullName:)** and provide it a first, middle, and last name (feel free to change the names). Because you declared the middle name component of the tuple to be of type **String?**, you can pass **nil** to that portion of the tuple. You cannot do this for the first or last name portion of the tuple.

Nothing is logged to the console. Because the middle name provided is **nil**, the Boolean used in the optional binding does not evaluate to **true** and **print()** is not executed.

Try giving the middle name a valid **String** instance and note the result.

Exiting Early from a Function

You learned about Swift’s conditional statements in Chapter 3, but there is one more to introduce: guard statements. Just like if/else statements, guard statements execute code depending on a Boolean value resulting from some expression. But guard statements are different from what you have seen before. A guard statement is used to exit early from a function if some condition is *not* met. As their name suggests, you can think of guard statements as a way to protect your code from running under improper conditions.

Following the example above, consider an example in which you want to write a function that greets a person by their middle name if they have one. If they do not have a middle name, you will use something more generic.

Listing 12.13 Early exits with guard statements

```
'''
func greetByMiddleName(fromFullName name: (first: String,
                                           middle: String?,
                                           last: String)) {
    guard let middleName = name.middle else {
        print("Hey there!")
        return
    }
    print("Hey, \(middleName)")
}
greetByMiddleName(fromFullName: ("Alice", "Richards", "Ward"))
```

greetByMiddleName(fromFullName:) is similar to **grabMiddleName(fromFullName:)** in that it takes the same argument, but it differs in that it has no return value. Another difference is that the names of the elements in the tuple name match specific components of a person’s name. As you can see, these element names are available inside the function.

The code `guard let middleName = name.middle` binds the value in `middle` to a constant called `middleName`. If there is no value in the optional, then the code in the guard statement’s body is executed. This would result in a generic greeting being logged to the console that omits the middle name: `Hey there!`. After this, you must explicitly `return` from the function, which represents that the condition established by the guard statement was not met and the function needs to return early.

You can think of `guard` as protecting you from embarrassingly addressing somebody as “mumble-mumble” when you do not know their middle name. But if the tuple did get passed to the function with a middle name, then its value is bound to `middleName` and is available after the guard statement. This means that `middleName` is visible in the parent scope that encompasses the guard statement.

In your call to **greetByMiddleName(fromFullName:)**, however, you pass in a middle name to the tuple name. That means `Hey, Richards!` will be logged to the console. If `nil` were passed to the middle name element, then `Hey there!` would log to the console. (Go ahead and try it.)