Big Nerd Ranch

2ND EDITION

Swift Programming

THE BIG NERD RANCH GUIDE

Matthew Mathias and John Gallagher

Swift Programming: The Big Nerd Ranch Guide

by Matthew Mathias and John Gallagher

Big Nerd Ranch, LLC
200 Arizona Ave NE
Atlanta, GA 30307
(770) 817-6373
http://www.bignerdranch.com/
book-comments@bignerdranch.com

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
http://www.informit.com

ISBN-10 0134610687
ISBN-13 978-0134610689

Second edition, second printing, April 2017
Release D.2.2.1

Function Types

Each function you have been working with in this chapter has a specific type. In fact, all functions do. *Function types* are made up of the function's parameter and return types. Consider the **`sortedEvenOddNumbers(_:)`** function. This function takes an array of integers as an argument and returns a tuple with two arrays of integers. Thus, the function type for **`sortedEvenOddNumbers(_:)`** is expressed as `([Int]) -> ([Int], [Int])`.

The function's parameters are listed inside the left parentheses, and the return type comes after the `->`. You can read this function type as: "A function with one parameter that takes an array of integers and returns a tuple with two arrays containing integers." For comparison, a function with no arguments and no return has the type `() -> ()`.

Function types are useful because you can assign them to variables. This feature will become particularly handy in the next chapter when you see that you can use functions in the arguments and returns of other functions. For now, let's just take a look at how you can assign a function type to a constant:

```
let evenOddFunction: ([Int]) -> ([Int], [Int]) = sortedEvenOddNumbers
```

This code creates a constant named `evenOddFunction` whose value is the function type of the **`sortedEvenOddNumbers(_:)`** function. Pretty cool, right? Now you can pass this constant around just like any other. You can even use this constant to call the function; for example, `evenOddFunction([1,2,3])` will sort the numbers in the array supplied to the function's sole argument into a tuple of two arrays – one each for even and odd integers.

You accomplished a lot in this chapter. There was a lot of material here, and it may make sense to go through it all a second time. Be sure to type out all of the code in this chapter. In fact, try to extend the examples to different cases. Try to break the examples and then fix them.

If you are still a little fuzzy on functions, do not worry. They are also a major focus in the next chapter, so you will get lots more practice.

Bronze Challenge

Like `if/else` conditions, `guard` statements support the use of multiple clauses to perform additional checks. Using additional clauses with a `guard` statement gives you further control over the statement's condition. Refactor the **`greetByMiddleName(name:)`** function to have an additional clause in its `guard` statement. This clause should check to see if the middle name is less than four characters long. If it is, then greet that person by their middle name. If it is not, then use the generic greeting.

Silver Challenge

Write a function called **`siftBeans(fromGroceryList:)`** that takes a grocery list (as an array of strings) and "sifts out" the beans from the other groceries. The function should take one argument that has a parameter name called `list`, and it should return a named tuple of the type `(beans: [String], otherGroceries: [String])`.

Here is an example of how you should be able to call your function and what the result should be:

```
let result = siftBeans(fromGroceryList: ["green beans",
                                         "milk",
                                         "black beans",
                                         "pinto beans",
                                         "apples"])

result.beans == ["green beans", "black beans", "pinto beans"] // true
result.otherGroceries == ["milk", "apples"] // true
```

Hint: You may need to use a function on the **`String`** type called **`hasSuffix(_:)`**.

For the More Curious: Void

The first function you wrote in this chapter was **`printGreeting()`**. It took no arguments and returned nothing. Or, did it?

It turns out that functions that do not explicitly return something actually do still have a return. They return something called **`Void`**. This return is inserted into the code for you by the compiler.

So, while you wrote **`printGreeting()`** like this:

```
func printGreeting() {
    print("Hello, playground.")
}
```

The compiler actually added something like this to your code:

```
func printGreeting() -> Void {
    print("Hello, playground.")
}
```

In other words, it added a return value of **`Void`** for you. Just what is **`Void`**? Go ahead and make **`printGreeting`** return **`Void`**, as shown above. Command-click on the word "Void" and Xcode will show you what it looks like in the standard library.

```
public typealias Void = ()
```

`Void` is a `typealias` for `()`. You have not read about `typealias`es in this book yet, but you will in Chapter 21. Stay tuned. For now, think of `typealias`es as a way to tell the compiler that one thing is shorthand for another. In the excerpt above, the standard library is establishing that **`Void`** is another way of expressing `()`.

You have already seen the concept at play here in Chapter 5. The `()` refers to what is called an empty tuple. If a tuple is a list of ordered elements, then an empty tuple is simply a list with nothing in it.

Given what you know now, you can see that these three implementations of **`printGreeting()`** are equivalent.

```
func printGreeting() {
    print("Hello, playground.")
}

func printGreeting() -> Void {
    print("Hello, playground.")
}

func printGreeting() -> () {
    print("Hello, playground.")
}
```

The first version above is what you originally wrote. The second is what the compiler inserts for you. And the third uses the empty parentheses, which is what the standard library maps **`Void`** to.

Knowing that **`Void`** maps to `()` should help you to better understand what is going on in a given function type. For example, the function type for **`printGreeting()`** is `() -> Void`. This is simply the type for a function that takes no arguments and returns an empty tuple, which is the implicit return type for all functions that do not explicitly have a return value.

13
Closures

Closures are discrete bundles of functionality that can be used in your application to accomplish specific tasks. Functions, which you learned about in the last chapter, are a special case of closures. You can think of a function as a named closure.

In Chapter 12, you worked primarily with global and nested functions. Closures differ from functions in that they have a more compact and lightweight syntax. They allow you to write a "function-like" construct without having to give it a name and a full function declaration. This makes closures easy to pass around in function arguments and returns.

Let's get started. Create a new playground called `Closures`.

Closure Syntax

Imagine that you are a community organizer managing a number of organizations. You want to keep track of how many volunteers there are for each organization and have created an instance of the **`Array`** type for this task.

Listing 13.1 Starting with an array

```
import Cocoa

var str = "Hello, playground"

let volunteerCounts = [1,3,40,32,2,53,77,13]
```

You entered the number of volunteers for each organization as they were provided to you. This means that the array is completely disorganized. It would be better if your array of volunteers were sorted from lowest to highest number. Good news: Swift provides a *method* called **`sorted(by:)`** that allows you to specify how an instance of **`Array`** will be sorted. (We call a function a method when it is defined on a type, like the **`Array`** type. More on this topic in Chapter 15.)

`sorted(by:)` takes one argument: a closure that describes how the sorting should be done. The closure takes two arguments, whose types must match the type of the elements in the array, and returns a **`Bool`**. The two arguments are compared to produce this return value, which represents whether the instance in the first argument should be sorted before the instance in the second argument. Use < in the return if you would like argument one to be sorted before argument two. Doing so will sort the array in an *ascending* fashion – that is, from smallest to largest. Use > in the return if you would like argument two to come before argument one. This will sort the array in an *descending* fashion – that is, from largest to smallest.

Because your array of volunteer counts is filled with integers, the function type for **`sorted(by:)`** will look like `((Int, Int) -> Bool) -> [Int]` in your code. In words, "**`sorted(by:)`** is a method that takes a closure that takes two integers to compare and returns a Boolean value specifying which integer should come first. **`sorted(by:)`** returns a new array of integers that have been ordered according to how the closure organizes them."

Add the following code to sort your array.

Listing 13.2 Sorting the array

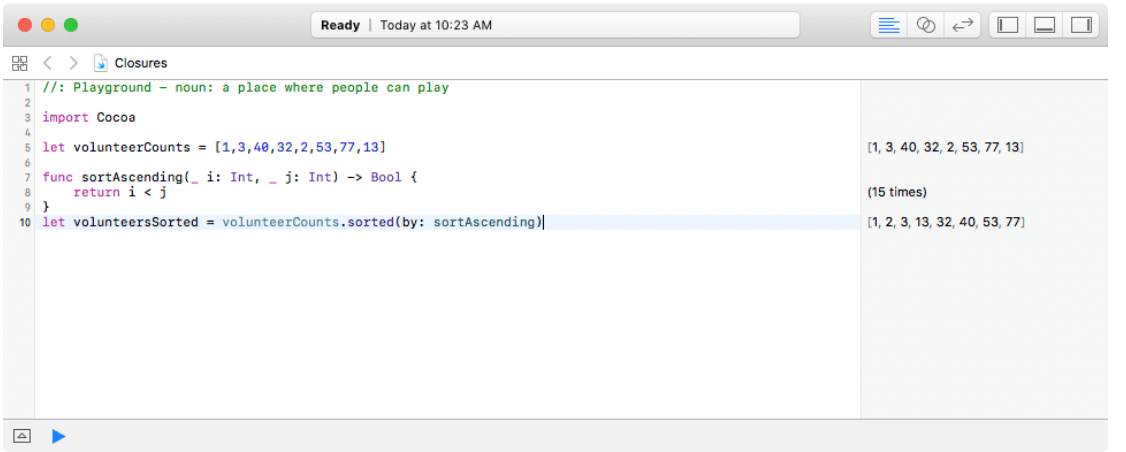
```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

func sortAscending(_ i: Int, _ j: Int) -> Bool {
    return i < j
}
let volunteersSorted = volunteerCounts.sorted(by: sortAscending)
```

First, you create a function called **`sortAscending(_:_:)`** that has the required type. It compares two integers and returns a Boolean that indicates whether the integer `i` should be placed before the integer `j`. Because the name **`sortAscending`** implies that we will be sorting two things, you use the _ to suppress the parameter names from being used in the call. The function will return `true` if `i` is less than – and should be placed before - `j`. As this global function is a named closure (remember, all functions are closures), you can provide this function as the value of the argument in **`sorted(by:)`**.

Next, you call **`sorted(by:)`**, passing in **`sortAscending(_:_:)`** for its argument. Because **`sorted(by:)`** returns a new array, you assign that result to a new constant array called `volunteersSorted`. This instance will serve as your new record for the organizations' volunteer counts, correctly sorted.

Look in the results sidebar of your playground. You should see that the values inside `volunteersSorted` are sorted from lowest to highest (Figure 13.1).

Figure 13.1 Sorting volunteer counts



Closure Expression Syntax

This works, but you can clean up your code. Closure syntax follows this general form:

```
{(parameters) -> return type in
     // Code
}
```

You write a closure expression inside of the braces ({}). The closure's parameters are listed inside of the parentheses immediately after the opening brace. Its return type comes after the parameters and uses the regular syntax. The keyword `in` is used to separate the closure's parameters and return type from the statements inside of its body.

Refactor your code to use a closure expression: Create a closure inline instead of defining a separate function outside of the **`sorted(by:)`** method.

Listing 13.3 Refactoring your sorting code

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

~~func sortAscending(_ i: Int, _ j: Int) -> Bool {~~
~~    return i < j~~
~~}~~
~~let volunteersSorted = volunteerCounts.sorted(by: sortAscending)~~

let volunteersSorted = volunteerCounts.sorted(by: {
    (i: Int, j: Int) -> Bool in
    return i < j
})
```

This code is a bit cleaner and more elegant than the first version. Instead of providing a function defined elsewhere in the playground, you implement a closure inline in the **`sorted(by:)`** method's argument. You define the parameters and their type (**`Int`**) inside of the closure's parentheses and also specify the return type. Next, you implement the closure's body by providing the logical test (is `i` less than `j`?) that will inform the closure's return.

The result is just as before: The sorted array is assigned to `volunteersSorted`.

This refactoring is a step in the right direction, but it is still a little verbose. Closures can take advantage of Swift's type inference system, so you can clean up your closure even more by trimming out the type information.

Listing 13.4 Taking advantage of type inference

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

~~let volunteersSorted = volunteerCounts.sorted(by: {~~
~~    (i: Int, j: Int) -> Bool in~~
~~    return i < j~~
~~})~~

let volunteersSorted = volunteerCounts.sorted(by: { i, j in i < j })
```