# Basic Exercises Part 4.5. Passing data back Closures

## A 🡨 B

* What are closures? Closures are blocks of code that you can pass around in your code, as if you assign a function to a variable. Mastering closures is a crucial aspect of learning iOS development.
* How a closure works in Swift? According to Apple documentation: *“closures are self-contained blocks of functionality that can be passed around and used in your code”.*
* In other words: a closure is a block of code that you can assign to a variable. You can then pass it around in your code, for instance to another function. That function then calls the closure and executes its code, as if the closure is an ordinary function. As you know, variables store information in your Swift code, and functions can execute tasks. With closures, you put a function’s code in a variable, pass it around, and execute its code somewhere else.
* Using closures to pass data between view controllers isn’t much different from using properties or delegation. The biggest benefit of using a closure is that it’s relatively easy to use, and you can define it locally – no need for a function or protocol.

### **1.1 Create a new project**

Create a basic Single View. We will create a new project on each approach, so we only going to write the steps once.

### **1.2 Write a closure**

Let’s write an example:

let sayHi = {

print(“Hello there!”)

}

sayHi()

// output: Hello there!

Here’s what happens in that code:

* On the first line, you’re defining a closure and assigning it to sayHi . The closure is the stuff between the squiggly brackets { and }. See how it’s similar to a function declaration? Note that the closure is assigned to sayHi with the assignment operator =.
* On the last line, the closure is called. It’s executed by calling sayHi(), the name of the constant sayHi with parentheses (). This is similar to calling a function.
* When you run the code, the closure is called with sayHi(), which will execute the closure, and print out *Hello there!* with print().
* At this point, the type of sayHi, and the type of the closure, is () -> ().

Next, let’s add a parameter to the closure. Parameters are input values for functions and closures. Just like functions, closures can have parameters.

let sayHi: (String) -> () = { name in

print("Hello there, \(name)!”)

}

sayHi("Julia”)

// output: Hello there, Julia!

Yes, it’s getting more complicated.

* Just like before, we’re declaring the closure on the first line, then assign it to the constant sayHi, and call the closure on the last line.
* The closure now has one parameter of type String. This parameter is declared as part of the closure *type* (String) -> ().
* You can then use the parameter name within the closure. When calling the closure, you provide a value for the parameter.

What do you make of that? In essence, there are three things that matter here:

* The *closure type* (String) -> ()
* The *closure expression* { name in ··· }
* The *closure call* sayHi(···)

The parameters of a closure aren’t *named*, unlike Swift functions. When you declare a closure you can specify the types of parameters it has, such as String in the above example, but you don’t specify a parameter name – only the type.

In the closure expression, the { name in … part, you assign a local variable name to the first parameter to the first parameter of the closure. This gives the parameter a name within the closure. You could have named it anything you wanted. In fact, you could have left out the variable name, and used the $0 shorthand! Like this:

let sayHi: (String) -> () = {

print("Hello there, \($0)!”)

}

sayHi("Carlos”)

// output: Hello there, Carlos!

The shorthand $0 is used to reference the value of that first parameter of the closure “sayHi”. How that works? We’re calling the closure and provide one paremter “Carlos” of type String. This parameter doesn’t have a name, just a value, and remember, the value gets a name within the closure.

### **1.3 Ok, now how to start?**

Start with creating a property on your secondary controller like this:

var completionHandler: ((String) -> (Int) )?

It’s a property completionHandler that has a *closure type*. The closure is optional, denoted by the ?, and the closure signature is (String) -> Int. This means the closure has one parameter of type String and returns one value of type Int.

Once more, in the secondary view controller, we call the closure when a button is tapped:

@IBAction func onButtonTap () {

let result = completionHandler?("Passing data”)

print("completion handler returns, \(result)!”)

}

In the example above, this happens:

* The closure completionHandler is called, with one string argument. Its result is assigned to result.
* The result is printed out with print()

Then, in the MainViewController you can define the closure like this:

vc.completionHandler = { text in

print("text = \(text)")

return text.characters.count

}

This is the closure itself. It’s declared locally, so you can use all local variables, properties and functions.

In the closure the text parameter is printed out, and then the string length is returned as the result of the function.

This is where it gets interesting. The closure lets you pass data between view controllers bi-directionally! You can define the closure, work with the data that’s coming in, and return data to the code that invokes the closure.

You may note here that a function call, with delegation or a direct property, also allows you to return a value to the caller of the function. That’s absolutely true!

Closures might come in handy in the following scenarios:

* You don’t need a complete delegation approach, with a protocol, you just want to create a quick function.
* You want to pass a closure through multiple classes. Without a closure you’d have to create a cascading set of function calls, but with the closure you can just pass the block of code along.
* You need to locally define a block of code with a closure, because the data you want to work with only exists locally.

One of the risks of using closures to pass data between view controllers is that your code can become very dense. It’s smartest to only use closures to pass data between view controllers if it makes sense to use *closures* over any other method – instead of just using closures because they’re so convenient!

So… what if you want to pass data between view controllers that don’t have, or can’t have, a connection between them? I’ll see in the next chapter.

### **1.4 Try**

Now try on your own. Pass data from your second view controller back to the main view controller

Instead of using the delegate pattern, that split the implementation in various part of the UIViewController class, you can even use closures to pass data back and forward. By assuming that you're using the UIStoryboardSegue, in the prepareForSegue method you can easily setup the new controller in one step

final class DestinationViewController: UIViewController {

var onCompletion: ((success: Bool) -> ())?

@IBAction func someButtonTapped(sender: AnyObject?) {

onCompletion?(success: true)

}

}

final class MyViewController: UIViewController {

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {

guard let destinationController = segue.destinationViewController as? DestinationViewController else { return }

destinationController.onCompletion = { success in

// this will be executed when `someButtonTapped(\_:)` will be called

print(success)

}

}

}

### **1.4 Want to know more?**

Full explanation of closures. Visit:

<https://learnappmaking.com/closures-swift-how-to/>