# Basic Exercises Part 4.4. Passing data back Delegation

## A 🡨 B

* Delegation, also known as the Delegate pattern. It is a must-have for any iOS developer’s tool.
* What is Delegation? According to Apple’s official definition: *“Delegation is a design pattern that enables a class to hand off (“or delegate”) some of its responsibilities to an instance of another class.* ***This design pattern is implemented by defining a protocol that encapsulates the delegated responsibilities, such that a conforming type (known as a delegate) is guaranteed to provide the functionality that has been delegated.*** *Delegation can be used to respond to a particular action, or to retrieve data from an external source without needing to know the underlying type of that source”*.
* Let’s break it down with a real code example:
  + You’ve all probably used delegation. It would be hard no to in iOS. Think of a UIViewController that you’ve subclassed (call it ViewController) to manage a UICollectionView. ViewController includes. A UICollectionView instance (call it collectionView). You set the collectionView instace’s delegate property to self and self is ViewController.
  + ViewController adopts the UICollectionViewDelegate protocol. The delegate is ViewController. To conform to the UICollectionViewDelegate protocol, ViewController implements methods like “didSelectItemAt indexPath”.
  + By implementing that method, ViewController gets informed of taps on UICollectionViewCell objects. When a cell is tapped, the ViewController implementation of didSelectViewCell is called. You get to define the body of the didSelectItemAt method so that, for example, when a cell is tapped, you can visually highlight the cell and perform some application-specific logic. collectionView has delegated the responsibility of handling taps on UICollectionViewCell objects to ViewController.
* That’s quite complex. Think about delegation in the real world. Imagine you and I are part of a team that delivers chocolate cookies to an event. You’re in charge of baking cookies, and you delegate making the cookie dough to me. Once I’m done I give the cookie dough to you, so you can use it to bake the cookies.
  + A few key points stand out:
    - You’re in charge of making cookies, and you delegate creating cookie dough to me
    - You could say that making cookies is your responsibility, and you’re handing off that responsibility to me
    - And it goes two ways: I give you the cookie dough once I’m done with my delegated task. It’s not much different in Swift programming! One class delegates a task to another class, handling off some of its responsibilities.
* Requirement: a protocol. In order to write a sample code implementing the delegation design pattern, a “protocol” is needed.
  + Remember, a protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to conform to that 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 Defining and creating a delegate**

Continue with the cookie example, let’s create a struct.

struct Cookie {

var size: Int = 5

var hasChocoChips: Bool = false

}

Then, define a class called Bakery:

class Bakery {

func makeCookie() {

cookie.size = 6

cookie.hasChocoChips = true

}

}

The Bakery class has a function called makeCookie() that creates a cookie with the Cookie struct, and sets some of its properties. At this point we want to sell the cookies in three different ways: 1) In the bakery shop; 2) on the bakery’s website; 3) wholesale, to cookie distributors.

Selling cookies isn’t the bakery’s responsibility, but delivering cookies is. So, we need a way to deliver cookies once they are baked without coding all that into the Bakery class. That’s where delegation comes in!.

First, we’re defining a protocol that will encapsulate the responsibilities that we’re handing off. Like this:

protocol BakeryDelegate {

func cookieWasBaked(\_ cookie: Cookie)

}

This BakeryDelegate protocol defines one function cookieWasBaked. This delegate function will get called whenever a cookie has been baked. Second, we’re incorporating delegation into the Bakery class:

class Bakery {

var delegate:BakeryDelegate?

func makeCookie() {

var cookie = Cookie()

cookie.size = 6

cookie.hasChocolateChips = true

delegate?.cookieWasBaked(cookie)

}

}

Two things changed in the Bakery class:

1. The delegate property, of type BakeryDelegate, has been added
2. The function cookieWasBaked(\_:) is called on the delegate in makeCookie()

And that’s not all. Check this out:

* The type of the delegate property is the protocol we defined earlier. You can assign any value to the delegate property, as long as it conforms to the BakeryDelegate protocol. **Note:** If you’re unfamiliar with protocols, read [this tutorial](https://learnappmaking.com/protocols-swift-how-to/) about them.
* The delegate property is an [optional](https://learnappmaking.com/swift-optionals-how-to/), and we use optional chaining when calling that cookieWasBaked(\_:) function. The code still runs OK when delegate is nil, and there is no delegate! Cookies are baked, just nothing is done with them.

So, summarizing: you’ve now defined a BakeryDelegate protocol that defines some of the responsibilities that the Bakery delegates, and you’ve implemented the hand-off in makeCookie().

Third, let’s create the actual delegate class! Like this:

class CookieShop: BakeryDelegate {

func cookieWasBaked(\_ cookie: Cookie) {

print("Yay! A new cookie was baked, with size \(cookie.size)")

}

}

The CookieShop adopts the BakeryDelegate protocol, and conforms to that protocol by implementing the cookieWasBaked(\_:) function.

And finally, here’s how to put the code together:

let shop = CookieShop()

let bakery = Bakery()

bakery.delegate = shop

bakery.makeCookie()

// Output: Yay! A new cookie was baked, with size 6

Here’s what happens:

* First, you create a CookieShop object and assign it to the shop constant.
* Then, you create a Bakery object and assign it to the bakery constant.
* Then, you assign shop to bakery.delegate. This makes the shop the delegate of the bakery.
* Finally, when the bakery makes a cookie, that cookie is handed off to the shop, that can sell it to a happy customer

And that’s delegation! The bakery delegates selling cookies to the shop, and hands-off a cookie whenever it makes one.

The power of delegation lies in the simple fact that the bakery doesn’t need to know where its cookies end up. It can provide them to any class that adopts the BakeryDelegate protocol!

The bakery doesn’t need to know about the implementation of that protocol, only that it can call the cookieWasBaked(\_:) function when needed.

Why doesn’t the shop call makeCookie() directly whenever it needs a cookie to sell? The answer lies in the nature of delegation, and which class is in control.

### **1.3 Still don’t get anything**

Ok, less talk, more action.

Delegates help two objects to communicate each other.

Create a new project with a simple label on one view. In the secondary view with the text from a textView, update the previous label.

A screenshot of a cell phone

Description automatically generated

* Next. Two ViewControllers (I’ll be referring to one as SenderVC and one as ReceiverVC)
* ReceiverVC is the initial view controller with a Navigation Bar at the top with one Navigation Item and a UILabel in the center of the view.
* The navigation item should be a segue to the SenderVC which only has a Textfield and a Button.

### **1.4 Creating the protocol**

In our SenderVC we want to create a protocol as below:

protocol UpdateLabelTextDelegate {

func updateLabelText(withText text: String()

}

class SenderViewController: UIViewController {

var delegate = UpdateLabelTextDelegate?

**@IBOutlet** **weak** **var** textField: UITextField!

}

In this protocol I am declaring a function that MUST be implemented by anyone that want’s to be my (in this case SenderVC’s) delegate and I am also creating a variable called delegate of type UpdateLabelTextDelegate? that will store the object that wants to be my delegate and adhere to my protocol.

*The reason delegate is being set as an optional is because if no one wants to be SenderVC’s delegate then the value will be nil.*

### **1.5 Becoming the delegate**

Now that we’ve created the protocol let’s go over to our ReceivingVC and setup our view to become the delegate of SenderVC.

**import** UIKit

**class** ReceiverViewController: UIViewController, UpdateLabelTextDelegate {

**@IBOutlet** **weak** **var** receiveLabel: UILabel!

**func** updateLabelText(withText text: String) {

receiveLabel.text = text

}

**override** **func** prepare(for segue: UIStoryboardSegue, sender: **Any**?) {

**if** segue.identifier == "goToSender" {

**let** dest = segue.destination **as**! SenderViewController

dest.delegate = **self**

}

}

}

First we added UpdateLabelTextDelegate next to UIViewController. All this is doing is saying that I (ReceiverVC) will adhere to UpdateLabelTextDelegate (the protocol). Now that ReceiverVC has done this it now MUST implement the function updateLabelText which you can see we do.

Going back to the beginning. The objective here is to update the label on ReceiveVC with the textField text from the SenderVC. To do this SenderVC wants to call the protocol function on whomever is the delegate and pass it’s textField text as the argument. Makes sense right? So if the argument coming into the function is going to be the SenderVC’s textField we want to set the label’s text on ReceiverVC to that argument. Which is what you see above in the updateLabelText function.

The last thing we do here is actually set ReceiverVC as the delegate when we segue to the SenderVC.

### **1.6 Sending the text**

Send the input text from the textField back to the label in the Receiver.

**@IBAction** **func** sendTextToReceiver(**\_** sender: UIButton) {

delegate?.updateLabelText(withText: textField.text ?? "")

dismiss(animated: **true**, completion: **nil**)

}

If you recall in the last step when we segued into SenderVC we set ReceiverVC as the delegate. So we know that ReceiverVC has implemented the updateLabelText function. Here we are simply calling the delegate’s (ReceiverVC’s) updateLabelText function and passing it the argument of the textField’s text.

The last line of code just dismiss the view so we can get back to ReceiverVC.

And that’s it! your label on ReceiverVC should now have the text you entered into the textField on SenderVC.