# Basic Exercises Part 4.6. Passing data between View Controller with NotificationCenter

## A 🡨🡪 B

* You can pass data between view controllers with Notificaciont Cente, via its NotificacionCenter class.
* The Notification Center handles notifications, and forwards incoming notifications to components that are listening for them. The Notification Center is the iOS SDK’s approach to the Observer-Observable software design pattern.
* These notifications are NOT push notifications.
* Working with Notification Center has three key components:
  + Observing the notification.
  + Sending the notification.
  + Responding the notification.
* That’s quite complex. Let’s start with observing the notification. Before you can respond to a notification, you need to tell the Notification Center that you want to observe it. The Notification Center then tells you about any notifications it comes across, because you’ve indicated you’re on the lookout for them.

Every notification has a name to identify them. In MainViewController you add the following static property to the top of the class:

static let notificationName = Notification.Name("myNotificationName")

This static property, also known as a class property, is available anywhere in the code by calling MainViewController.notificationName. This is how you identify the notification with one single constant. You wouldn’t want to mix up your notifications by mistyping it somewhere!

Here’s how you observe for that notification:

NotificationCenter.default.addObserver(self, selector: #selector(onNotification(notification:)), name: MainViewController.notificationName, object: nil)

You usually add this in viewDidLoad() or viewWillAppear(\_:), so that the observation is registered when the view controller is put on screen. Here’s what happens in the code sample above:

* You use NotificationCenter.default, which is the default Notification Center. You could create your own Notification Center, for instance for a certain kind of notifications, but chances are the default center is fine.
* You then call the function addObserver(\_:selector:name:object:) on the Notification Center.
  + The first argument is the instance that does the observation, and it’s almost always self.
  + The second argument is the selector you want to call when the notification is observed, and this is almost always a function of the current class.
  + The third parameter is the name of the notification, so you pass the static constant notificationName.
  + The fourth parameter is the object whose notifications you want to receive. You usually pass nil here, but you could use it to only observe notifications from one particular object.

At a later point you can stop observing the notification with this:

NotificationCenter.default.removeObserver(self, name: MainViewController.notificationName, object: nil)

You can also stop observing for all notifications with:

NotificationCenter.default.removeObserver(self)

Remember that notifications are explicit, so you always observe one type of notification that results in one function call on one object (usually self) when the notification occurs.

The function that will get called when the notification occurs is onNotification(notification:), so let’s add that to the class:

@objc func onNotification(notification:Notification) {

print(notification.userInfo)

}

The @objc keyword is required in Swift 4 and higher, because the NSNotificationCenter framework is Objective-C code. In the function, you’ll simply print out the notification payload with notification.userInfo.

Then, posting the notification is easy. Here’s how you do that:

NotificationCenter.default.post(name: MainViewController.notificationName, object: nil, userInfo:["data": 42, "isImportant": true])

Again, there’s a few moving parts:

* You call the function post(name:object:userInfo:) on the default Notification Center, exactly the same center as you used before.
* The first function argument is the name of the notification, that static constant that you defined before.
* The second argument is the object posting the notification. You can often leave this nil, but if you’ve used the object argument when observing the notification you can pass the same object here to exclusively observe and post for that object.
* The third argument is the notification payload called userInfo. You can pass a dictionary with any kind of data here. In this example, you’re passing some data and a boolean value.

That’s all there is to it!

The Notification Center comes in handy in a few scenarios:

* The view controllers or other classes you want to pass data between are not closely related. Think about a [table view controller](https://learnappmaking.com/table-view-controller-uitableviewcontroller-how-to/) that needs to respond when a REST API receives new data.
* The view controllers don’t necessarily have to exist yet. It could happen that the REST API receives data before the table view is put on screen. Observing for notifications is optional, which is an advantage if parts of your app are ephemeral.
* Many view controllers need to respond to one notification, or one view controller needs to respond to multiple notifications. Notification Center is many-to-many.

You can think of the Notification Center as a superhighway for information, where notifications are constantly sent over its lanes, in many directions and configurations.

If you just want some “local traffic” between view controllers, it doesn’t make sense to use Notification Center – you’d use a simple delegate, property or closure instead. But if you want to repeatedly and regularly send data from one part of your app to the other, Notification Center is a great solution.

A thing to remember **the difference between the delegation pattern, the delegates and protocols and the notifications and observers is delegates and protocols are a one-to-one communication pattern whereas notifications and observers** which we’re about to see when we dive into the code here **is a one-to-many**. For example, you can have 5 different observers waiting to hear one notification action and they all do 5 different actions.

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

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

### **1.2 Quick project overview**

We have two screens. When you hit one button in the first window (view controller), a selection screen will pop up modally, and then you can pick a background color in the second view. And then, the main screen will change based on your selections. In order to choose the colors, create 2 buttons in the 2nd screen, dismiss the view as soon the user tap one and set that background color in the 1st view.

### **1.3 Create the views**

Try this code on the first view.

let lightNotificationKey = "lightColor"

let darkNotificationKey = "darkColor"

class BaseScreen: UIViewController {

@IBOutlet weak var chooseButton: UIButton!

let light = Notification.Name(rawValue: lightNotificationKey)

let dark = Notification.Name(rawValue: darkNotificationKey)

deinit {

NotificationCenter.default.removeObserver(self)

}

override func viewDidLoad()\_ cookie: Cookie) {

super.viewDidLoad()

chooseButton.layer.cornerRadius = chooseButton.frame.size.height/2

createObservers()

}

func createObservers() {

//Background

NotificationCenter.default.addObserver(self, selector: #selector(BaseScreen.updateColorBackgroundLight(notification:)), name: light, object: nil)

NotificationCenter.default.addObserver(self, selector: #selector(BaseScreen.updateColorBackgroundDark(notification:)), name: light, object: nil)

}

@objc func updateColorBackgroundDark (notification: NSNotification) {

let isLight = notification.name == dark

let color = isLight ? UIColor.cyan : UIColor.blue

view.backgroundColor = color

}

@objc func updateColorBackgroundLight (notification: NSNotification) {

let isLight = notification.name == light

let color = isLight ? UIColor.cyan : UIColor.green

view.backgroundColor = color

}

@IBAction func chooseButtonTapped(\_ sender: UIButton) {

let selectionVC = storyboard?.instantiateViewController(withIdentifier: "SelectionScreen") as! SelectionScreen

present(selectionVC, animated: true, completion: nil)

}

}

Then, define the selection screen class

class SelectionScreen: UIViewController {

override func viewDidLoad() {

super.viewDidLoad()

}

@IBAction func lightButtonTapped(\_ sender: UIButton) {

let name = Notification.Name(rawValue: darkNotificationKey)

NotificationCenter.default.post(name: name, object: nil)

dismiss(animated: true, completion: nil)

}

@IBAction func darkButtonTapped(\_ sender: UIButton) {

let name = Notification.Name(rawValue: lightNotificationKey)

NotificationCenter.default.post(name: name, object: nil)

dismiss(animated: true, completion: nil)

}

}

### **1.4 Explanation**

When you tap the choose button we pop up the selection screen, and the selection screen we’re just dismissing it when you tap one of the buttons. Now let’s add the notifications and observers to change the base screen once we select aside.

The first thing we’re going to do is set up our notification key which is a string and these are unique to our notifications.

let lightNotificationKey = “lightColor”

let darkNotificationKey = “darkColor”

How we’re going to set this up is we’re going to have to post notifications one for each button is pressed in the second view. Remember, the observers are just waiting around to get this notification so we’re going to set up some observers that listen for when one button is pressed.

Note: This approach is correct, like third-party libraries or even Apple itself uses keys like this it’s best to uniquely identify it like this: “site.name.nameListener”, similar to the bundler ID.

Right now the keys are global. In a real project, where you can have a lot of keys for various other things it’s nice to create a separate constant file that holds all these but for now this is what we’re going to do.

Jump over to the selection screen. That screen is very simple, this is where we’re just posting the notifications. Type: NotificationCenter.default.post (here is the key point, this is where we’re posting the notification) and it’s going to autocomplete and you want to have a name, which names? Yes, the unique key names. Notice that one of this post methods, allows you to set any kind of object, so it is possible to pass information through the notification, but for now keep it simple. To get the name type Notification.Name and choose the rawValue accordingly to the key you want to use. So now, anytime we tapped one of those buttons, we’re posting a notification that has the name of the notifications keys. And that’s all what we have to do in the selection screen.

Now we need to add some observers that are looking for that one specifically. Back to the base screen.

Create a local variables with the notifications key names (just for convenience), because is common to using a lot when add observers. One best practices is dealing with observers is to remove them when they’re no longer necessary, for example remove them when the base screen is de-allocated from memory (or any other view in your project); use the deinit method. Inside, remove the observer, type NotificationCenter.default.removeObserver from self because self is the base screen for the observers. The reason you want to do this is because if we don’t remove them, you going to have all these observers listening for notifications and it could cause confusion if we have all these left over observers and things are trying to notify.

Now create the observers. Do that in a separate function just to keep things nice like createObservers, Each one of these observer is going to update a different item on the screen, or in our case the background color. Adding a observer is pretty similar to the post like we did in the selection screen, type NotificationCenter.default.addObserver and choose the 1st option; self because is that class itself. The selector is what method we want to execute when this observer gets the message. Finally in the method that receive the message change the background color.

Important. Make sure you call the method to create all the observers. The viewDidLoad is a good place.

### **1.5 Pros & Cons**

Now, can you name the pros and cons of this approach? When to use delegates and when to use notifications?

