Application Development with Swift

Apple offers a new creative, easy, and innovative programming language for application development, called Swift. Swift makes iOS application development a breeze by offering speed, security, and power to your application development process. Swift is easy to learn and has awesome features such as being open source, debugging, and so on. Swift has simplified its memory management with Automatic Reference Counting (ARC) and it is compatible with Objective-C.

This book provides you with the information and skills you need to use the new programming language, Swift. The book starts with an introduction to Swift and code structure. Following this, you will use Playgrounds to become familiar with the language in no time. Then the book takes you through the advanced features offered by Swift and how to use them with your old Objective-C code or projects. You will then learn to use Swift in real projects by covering APIs such as HealthKit, Metal, WatchKit, and Touch ID in each chapter. The book’s easy to follow structure ensures you get the best start to developing applications with Swift.

Who this book is written for
If you are an iOS developer with experience in Objective-C, and wish to develop applications with Swift, then this book is ideal for you. Familiarity with the fundamentals of Swift is an added advantage but not a necessity.

What you will learn from this book
- Use Playgrounds in Xcode to make the writing of Swift code productive and easy
- Get acquainted with the advanced features of Swift and make complete use of them in your code
- Add a new method for authentication to your app using Touch ID
- Develop health-related apps using HealthKit
- Take your apps to the next level of performance and capability using Metal
- Develop applications for wearables using WatchKit
- Use notification center to easily access all your notifications
- Make your users’ devices more stylish by using Apple’s built-in QuickType keyboard, instead of the native one

Hossam Ghareeb

Develop highly efficient and appealing iOS applications by using the Swift language
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Introduction to HealthKit'
- A synopsis of the book’s content
- More information on Application Development with Swift
About the Author

**Hossam Ghareeb** is a software engineer who graduated from the computer and system engineering department of Alexandria University in 2012. Currently, he is an iOS software engineer at Prototype Interactive in Dubai. He has a lot of experience in iOS development and software engineering; he always keeps himself up to date with new technologies. It is his passion to help people by sharing his experience and writing tutorials to get people engaged quickly without problems. His favorite hobbies are fishing and playing video games, especially FIFA and PES. If you need his help or wish to go fishing with him, then you can contact him at hossam.ghareb@gmail.com.
Preface

This book teaches you how to master the Swift programming language and use it in building apps with the new technologies introduced in iOS 8. Swift is a new programming language for iOS, OS X, and watchOS apps. In the beginning of this book, we will give you an advanced introduction to Swift, and then we will see how to apply this in real-app demos using the new technologies and APIs in iOS 8 such as TouchID, watchOS, Metal, and HealthKit.

What this book covers

Chapter 1, Hello Swift, gives a quick introduction and revision about Swift. This chapter is very important for Swift beginners and is highly recommended for Swift developers to revise their knowledge.

Chapter 2, Advanced Swift, boosts your Swift experience by allowing you to learn some advanced features in Swift. In this chapter, we will talk about protocols, extensions, and memory management.

Chapter 3, Touch ID, gives your app a new flavor of authentication using Touch ID. Users can now use their fingerprint to sign in to your app without needing to write their username and password.

Chapter 4, Introduction to HealthKit, teaches you how to work with HealthKit and how to share or read health data in the native Health app. If you have any apps or ideas about health or fitness, you should give this chapter a shot.

Chapter 5, Introduction to Metal, introduces a cutting edge technology for 3D GPU-accelerated graphics in iOS.

Chapter 6, Introduction to WatchKit, introduces a new device for you—the Apple watch. You will learn all the new technologies needed to build apps for the Apple watch and get your app prepared to exist on users wrists.
Chapter 7, *Swift App Extensions*, teaches you how to let your users use the app even if your app is not open, using app extensions. App extensions let you extend your app's functionality outside your app and users can access it using system or third-party apps.
Introduction to HealthKit

Starting with iOS 8, many frameworks and new APIs have been introduced to build awesome apps with new ideas. In this chapter, we are going to introduce one of these new frameworks, which is HealthKit. We will see how to use it to communicate with the native Health app in your iOS device.

What is HealthKit?
HealthKit is a new API provided by iOS 8. It enables apps to read the user's health information using the native Health app in their devices. You can consider that the Health app is a user interface for the HealthKit store. Other apps can write data in the Health app, such as workouts results, or statistics, so that the user can access them. This helps users to find all the information about their health in one location. The health information is stored in a secure location, and the user is responsible for deciding which data can be shared with apps. In the Health app, users can view, add, delete, and manage all their health data, and can also edit all the permissions to the third-party apps. HealthKit can also work with fitness or health devices, and it can save data from Bluetooth heart rate monitors. If your device has an M7 motion processor, HealthKit can import the step count data to it. So, all the developers who care about health and fitness apps will need to work with this new API.

HealthKit limitations
There are some limitations that everybody needs to be aware of before deciding to work with HealthKit:

- The HealthKit framework is introduced in iOS 8.
- HealthKit and the Health app are not available for iPad.
• The HealthKit framework can't be used in the app extensions, such as WatchKit.
• You need permission to access every type of data in the Health app. For example, a user can let your app read the blood type info, and prevent it from reading the heart rate or the step count info.
• The HealthKit store is encrypted, and your app can't read data when the phone is locked. So, when your app runs in background, it may not be able to access the store. However, your app can still write data even if the phone is locked, as the data will be cached, and then saved when the phone is unlocked.
• Any app that uses the HealthKit framework must provide a privacy policy.

HealthKit privacy
Health information is very sensitive, and HealthKit gives its users full control over their data to decide which data is to be shared with the third-party apps, and which apps can share or use this data. Here, I will list some points to consider while working with HealthKit:

• Don't use the information that is gained from HealthKit for advertising or other similar services.
• You must not disclose any information gained to a third party without getting permission from the user. Even with permission, you can share this data only with those third parties that provide fitness or health services.
• You can't sell information gained from HealthKit to any other service or platform.
• You can share this information with a third party, for medical research, only if the user consents.

Getting started with HealthKit
To get started with HealthKit, we will create a demo that will teach you how to request permission from the user to access his data from the Health app, read and format this information to make it readable for the user, and write info in the Health app. Now, go to Xcode and create a new project with a single view template, and make sure that Swift is selected as the project language.
Configuring the Xcode project

To configure our project using HealthKit, you have to first enable the HealthKit capabilities. To do this, select your target on Xcode, and click on the Capabilities tab. Then, enable HealthKit in the list by switching it ON; check the following screenshot:

After enabling the HealthKit capabilities, HealthKit will be added to the list of required device capabilities. This will prevent users from installing your app on devices that don’t support HealthKit, such as iPad. But what if HealthKit is not your main idea or the main operation in your app? In this case, you have to open the info.plist file of your app, and remove healthkit from the Required device capabilities array. Now, your project is ready to work with HealthKit.

Please make sure that your app identifier is set in a developer account, and you have enabled HealthKit on it. In the current example, I used Xcode 7, as Apple now allows developers to run apps on devices without the developer account, but you will have to login first with your iCloud account. So, if you use your developer account, please make sure that the HealthKit is enabled in the app identifier.
Getting your hands dirty with HealthKit

Now, we have configured our Xcode project. In the project demo, we will take permission for reading and writing information from HealthKit. We will ask for reading permissions to read age, weight, and height. Then, we will ask for writing permissions to write weight and height. Let's get started.

The HealthKit store

In your app, you need to instantiate a `HKHealthStore` object to be the interface between you and the HealthKit database. Once your app is launched, you need only one HealthKit store object per app; this is why we will declare it in `AppDelegate.swift`, and access it from all the view controllers or anywhere. Now, let's open `AppDelegate.swift` to import the HealthKit framework and define our store:

```swift
import UIKit
import HealthKit

let currentHealthStore = HKHealthStore()
```

As you see, after importing the HealthKit framework, we defined `currentHealthStore` to be our store, and we will access it anywhere in our project. Now, let's move to `ViewController.swift` to ask for the permission. Before this, we have to first check whether HealthKit is available in the current device or not. To do this, the `HKHealthStore` class has a class method called `isHealthDataAvailable` to tell you whether HealthKit is available in the current device or not. We do so because, as explained earlier, HealthKit is not available for iPad, and there are some specific editions of iPhone that also don't have HealthKit. Now, go to the `viewDidLoad` method in `ViewController.swift`, and check for the availability of HealthKit:

```swift
if HKHealthStore.isHealthDataAvailable(){
    // HealthKit is supported in this device
}
else{
    let alertController = UIAlertController(title: "Warning", message: "HealthKit is not available in your device!", preferredStyle: .Alert)
    alertController.addAction(UIAlertAction(title: "Ok", style: .Cancel, handler: nil))
    self.presentViewController(alertController, animated: true, completion: nil)
}
```
As you saw in the previous code, we first checked for availability, and if it is not available, we displayed an alert to the user that HealthKit is not available in his or her device.

**Asking for permissions**

We have checked whether HealthKit is available or not. We will now ask for permission to read and write data. We will start with the data that we want to read. As we said before, we will read age, height, and weight. Let's create a method that returns all the types that we want to read:

```swift
func dataToRead() -> NSSet{
    let heightType = HKObjectType.quantityTypeForIdentifier(HKQuantityTypeIdentifierHeight)
    let weightType = HKObjectType.quantityTypeForIdentifier(HKQuantityTypeIdentifierBodyMass)
    let birthdateType = HKObjectType.characteristicTypeForIdentifier(HKCharacteristicTypeIdentifierDateOfBirth)

    return NSSet(objects: heightType!, weightType!, birthdateType!)
}
```

In the previous code, we created a method that returns the NSSet of objects. These objects will be the types that we want to read. In HealthKit, all types are the subclasses of the `HKObjectType` class, which has all the common functionalities and properties of all the subclasses. In the data that we read, we used two types of data: `HKQuantityType` and `HKCharacteristicType`. The `HKQuantityType` class is a subclass of `HKObjectType`, and is used for quantity types that are stored as numeric values. The `HKCharacteristicType` class is also a subclass of `HKObjectType`, and is used to represent data that doesn't change over time, such as sex or date of birth. The characteristic types cannot be used to create new HealthKit objects. Instead, users must enter and edit their characteristic data using the Health app. Characteristic types are used only when asking for permission to read data from the HealthKit store.
Let's now create a method to return data that we want to write:

```swift
func dataToWrite() -> NSSet{
    let heightType =HKObjectType.quantityTypeForIdentifier(HKQuantityTypeIdentifierHeight)
    let weightType =HKObjectType.quantityTypeForIdentifier(HKQuantityTypeIdentifierBodyMass)

    return NSSet(objects: heightType!, weightType!)
}
```

The data that we write will be same as the data we read, only without the date of birth, because it's a characteristic type, and can be changed only from the Health app.

Now that the types are ready, let's ask for permission to read and write this data. We will use `currentHealthStore` in `AppDelegate` to ask for authorization. Update the code in `viewDidLoad` to look like this:

```swift
if HKHealthStore.isHealthDataAvailable() {
    // HealthKit is supported in this device
    let typesToRead = dataToRead()
    let typesToWrite = dataToWrite()

    currentHealthStore.requestAuthorizationToShareTypes(
typesToWrite as! Set<HKSampleType>, readTypes:
typesToRead as! Set<HKObjectType>, completion: { 
        (success, error) -> Void in
            if success {
                // We will update UI to preview data we read.
                dispatch_async(dispatch_get_main_queue(), { ()
                        -> Void in
                            self.updateUI()
                        })
            } else{
```
print("User didn't allow HealthKit to access these read/write data types",
appendNewLine:true)
}
})
})

We started our authorization by calling the requestAuthorizationToShareTypes method, which takes the read and write types as the parameters to display a view to the user asking for permission. After the user responds to the request, the completion block will be called with success and error. If everything goes fine, the success variable will be true, and you can update your UI this time. Now, run your app in the device, and you will see something like this:

![Image of Health Access view](image)

As you see, this view is displayed to ask the user to approve which data type we have access to. And as we said before, HealthKit gives users full control over which the data type can be accessed for read or write.
Updating the UI

Now, we can read and write the data types that the user has approved to access. We will now start to display the read data types. Let's first create a new view controller class with a subclass to UITableViewController, and it will be our HomeViewController class. Then move all the code we wrote in ViewController.swift, to the new one that is HomeViewController.swift, as shown in the following screenshot:

Now, switch to the Main.storyboard file to design the UI of HomeViewController. Now, from the right-hand side menu, drag UITableViewController and set its class to HomeViewController. Check out the following screenshot:
After adding `UITableViewController`, we will modify it to add the static cells that will display the information that we read from the HealthKit store. Now, select `UITableView` and change **Content** to **static cells**, and **Style** to **Grouped**. Then, add three cells to represent age, height, and weight. The design will be something like this:
Then, take outlets to these labels in your HomeViewController.swift. You will have something like this:

```swift
@IBOutlet weak var ageLabel: UILabel!

@IBOutlet weak var ageValueLabel: UILabel!

@IBOutlet weak var heightLabel: UILabel!
@IBOutlet weak var heightValueLabel: UILabel!

@IBOutlet weak var weightLabel: UILabel!
@IBOutlet weak var weightValueLabel: UILabel!
```

Now, the UI is ready to be filled with data that we read from the HealthKit store. Open the `updateUI` function, and we should read birth date, height, and weight. We will read each type in a separate method to keep our code more readable. Let's start with the birth date by creating a function called `updateUsersAge()`:

```swift
func updateUsersAge()
{
    do{
        var error : NSError!
        let birthdate = try
            currentHealthStore!.dateOfBirthWithError()

        let now = NSDate()

        let dateComponents =
            NSCalendar.currentCalendar().
            components(NSCalendarUnit.NSYearCalendarUnit,
            fromDate: birthdate, toDate: now, options:
            NSCalendarOptions.WrapComponents)
        let age = dateComponents.year

        self.ageValueLabel.text =
            NSNumberFormatter.localizedStringFromNumber
            (NSNumber(integer: age), numberStyle:
            NSNumberFormatterStyle.NoStyle)
    }
    catch{

        self.ageValueLabel.text = "Not Available"
    }
}
```
The code is straightforward—we used our store instance called `currentHealthStore` to call the `dateOfBirthWithError()` function, to get the user's data of birth from the Health app. The catch block will be called if there has been any error, or if the date of birth is not available in the store. Once you get the date of birth, you can use `NSDateComponents` and `NSCalendar` to get the difference of years to the current year. Then, update the UI by setting the text to the outlet label called `ageValueLabel`.

Now, let's see how to read height and weight. Before seeing how to read this data, I want to explain something. In HealthStore, when you read the quantity types, you run a query to fetch data, as this type has different values within the time. For example, in the previous year, if you have reported your weight in the Health app 10 times with different values, and then when you read the weight data from the store, you will have to specify which is the data that you need. In some apps, they may need to read the weight records of the previous year, only to display a chart about your progress in weight loss. In our app, we care only about the most recent value, which means your current weight. This is why, in reading the quantity types, you have to specify `NSSortDescriptors`, when you need to sort this data in any specific order. You can also specify `NSPredicate` if you want to filter the data; for example, read all the weights that are greater than 100 kg so that I will be able to know which times in the year I put on weight: in summer, or winter, or during school days. The last thing is that you can specify the limit that you want to read, for example, if you only care about the last five records you will set the limit value as 5. It makes the execution of the query faster if you don't need all the records, and care only about a specific number.

So, because we only care about the most recent value, let's first use the Swift feature of extensions to create an extension for `HKHealthStore`, to read the most recent value of a specific data type. Check out the following code:

```swift
typealias MostRecentCompletionHandler = (quantity: HKQuantity!, error: NSError!) -> Void

extension HKHealthStore{

    func getMostRecentValueOfType(quantityType: HKQuantityType,
        predicate: NSPredicate, handler: MostRecentCompletionHandler) {

        let sortDescriptor = NSSortDescriptor(key:
            HKSampleSortIdentifierEndDate, ascending: false)
        let query = HKSampleQuery(sampleType: quantityType,
            predicate: predicate, limit: 1, sortDescriptors: [sortDescriptor]) {
            (query, samples, error) -> Void in
```
As we saw, in the previous extension, we created NSSortDescriptor to sort the quantities by the end date to get the most recent record. Then, we created HKSampleQuery object using four parameters; the quantity type that we want to read and its passed as a parameter in the getMostRecentValueOfType method, the sort descriptor which we have created earlier, the NSPredicate as a nil value because we don't want to filter records, the results limit with 1 value as we want to read only the most recent one, and finally the completion block that gives you with query, samples and error. After executing the query, the completion block will be called; inside the block we will call our handler that was passed also as a parameter in the getMostRecentValueOfType method with the first quantity and error if occurred.

Now, let's go back to creating a function to read the height:

```swift
func updateUsersHeight() {

    self.heightLabel.text = "Height (m)"
    let heightType = HKQuantityType.quantityTypeForIdentifier(HKQuantityTypeIdentifierHeight)
    currentHealthStore.getMostRecentValueOfType(heightType!, predicate: nil, handler: { (quantity, error) -> Void in

        if quantity != nil {

            let mUnit = HKUnit.meterUnit()
            let height = quantity.doubleValueForUnit(mUnit)

            self.heightLabel.text = \"Height (m)\" = \\
" + String(height) + "
        }
    })
}
```
As we saw in the previous code, we called the `getMostRecentValueOfType` method that we added using the Swift extension. We passed the height type and the completion handler to get the quantity and error if occurred. The quantity will be `nil` if the user has not set his height in the Health app, and this is why we display the `Not Available` message. If the quantity is not `nil`, we format the value to be in a meter using the `HKUnit` class. `HKUnit` has many other units to use, such as inch or mile.

In the same way, we will now read the weight from the HealthKit store:

```swift
func updateUsersWeight()
{
    self.weightLabel.text = "Weight (Kg)"
    let weightType =
        HKQuantityType.quantityTypeForIdentifier
        (HKQuantityTypeIdentifierBodyMass)
    currentHealthStore.getMostRecentValueOfType(weightType!,
        predicate: nil, handler: { (quantity, error) -> Void in

            if quantity != nil{

                let gUnit = HKUnit.gramUnit()
                let weight = quantity.doubleValueForUnit(gUnit) /
                    1000.0
                dispatch_async(dispatch_get_main_queue(), { () ->
                    Void in

                self.weightValueLabel.text = "\((weight)"

            })
        }else{

```
Now, update your `updateUI` method to be like this:

```swift
func updateUI()
{
    updateUsersAge()
    updateUsersHeight()
    updateUsersWeight()
}
```

Now, if you run your app, you will see something like this:
Sharing data in the Health app

We have learned how to ask for the permission, and read information from the Health store. Now we see an example of how to share data in Health store. In this example, when the user clicks on the weight row, a pop-up view will be displayed, where the user will have to enter his new weight. Then, the user can save this value in the HealthKit store. Let's start with the UI to detect click on table cells and display the pop-up view. To detect, click on UITableViewCell, we will add the didSelectRowAtIndexPath function, as follows:

```swift
override func tableView(tableView: UITableView,didSelectRowAtIndexPath indexPath: NSIndexPath) {

    if indexPath.row == 2{
        //Weight cell
        let alertController = UIAlertController(title: "Your current weight", message: nil, preferredStyle: UIAlertControllerStyle.Alert)
        alertController.addTextFieldWithConfigurationHandler({
            (textField:UITextField) -> Void in
                textField.keyboardType = UIKeyboardType.DecimalPad
        })
        alertController.addAction(UIAlertAction(title: "Ok", style: UIAlertActionStyle.Default, handler: {
            (alertAction:UIAlertAction) -> Void in
                if let textField = alertController.textFields?.first {
                    let value = (textField.text! as NSString).doubleValue
                    self.saveWeightInStore(value)
                }
        }))
            (alertAction) -> Void in
        })))

        tableView.deselectRowAtIndexPath(indexPath, animated: true)
    })
```
We checked first that the selected row is the weight row and then displayed `UIAlertController` with `UITextField` to enter the new height. When the user clicks on **OK**, we will take the new value and save it in HealthKit store using the function `saveWeightInStore(value)` that we will now implement.

Now that the new weight value is ready, let's implement the `saveWeightInStore(newValue)` function:

```swift
func saveWeightInStore(weight: Double) {
    let gramUnit = HKUnit.gramUnit()
    let quantity = HKQuantity(unit: gramUnit, doubleValue: weight * 1000.0)
    let weightType = HKQuantityType.quantityTypeForIdentifier(HKQuantityTypeIdentifierBodyMass)
    let now = NSDate()
    let sample = HKQuantitySample(type: weightType!, quantity: quantity, startDate: now, endDate: now)
    currentHealthStore.saveObject(sample) { (success, error) in
        if success{
            self.updateUsersWeight()
        } else{
            print("Error in saving \(error)")
        }
    }
}
```
To save any data in the Health store, you have to prepare the HKQuantitySample instance that takes the quantityType parameter, which here is weightType. Then we also have the quantity type parameter which we created using the gram unit and the value. We multiplied the value by 1000.0 to convert it into grams. Then, we called the saveObject method in currentHealthStore, to save the data in store, and, if the saving operation succeeded, we refreshed the UI with the new value by calling the updateUserWeight method.

**Summary**

In this chapter, we covered in detail how to get started with HealthKit. We explained the idea of the HealthKit framework, and how to deal with sensitive data. We mentioned the HealthKit limitations, and its availability in the iOS devices. We saw how the users' data is protected in the Health app, and any app that needs to access or share any data type will need permission from user. The user can enable or disable this permission anytime from the Health app. We also learned how to read and share the data to the Health store, and how to deal with units. In the next chapter, we will give a good introduction, and the basic information that is needed to get started with Metal. Metal is a new framework for the GPU-accelerated 3D graphics in iOS.
Where to buy this book

You can buy Application Development with Swift from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.