

Swift — in — 30 Days

Build iOS Apps by Learning Swift, Xcode, and SwiftUI in Just Four Weeks



GAURANG RATNAPARKHI



Swift in 30 Days

*Build iOS Apps by Learning Swift,
Xcode, and SwiftUI in Just Four Weeks*

Gaurang Ratnaparkhi



www.bpbonline.com

FIRST EDITION 2022

Copyright © BPB Publications, India

ISBN: 978-93-91030-100

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:



www.bpbonline.com

Dedicated to

My family and all my friends

About the Author

Gaurang Ratnaparkhi has always been an iOS developer. With almost 11 years of experience, he has finally being able to contribute to the community. He started working in 2010 in Pune, India. He worked for almost 6 years in India, before moving to France. During this time, he worked on various apps belonging to different sectors like real estate, restaurants, and banking. He used various frameworks like CoreData, CoreLocation as well as developed cross-platform apps in Xamarin C#. He later moved to France where he did his Masters in Computer Engineering from the University of Grenoble Alpes, which is one of the top-ranking universities in France for Computer Science. After a year long internship in Windows 10 app development, he has been working in Toulouse as an iOS developer since 2018. Recently, he finished the Scrum Master certification and is looking forward to transitioning.

About the Reviewer

Rudra S Misra is an Apple Certified Trainer (ACT) in various technologies, including the App Development with Swift. He holds degrees in Business Administration and Computer Applications. He started his career as a teaching staff, followed by a Software Engineer position, and performs various roles related to the middle-to-senior-level management. He is also actively involved in content creation, R&D, technical blogging, and technology awareness programs.

Acknowledgement

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank Akshay Aher, a good friend, without whose encouragement I would have never started iOS development.

I am grateful to all the different places where I worked throughout my career. I learned a lot while working on different and challenging projects. I would also like to thank all my colleagues at Benchmark IT solutions, POS Nirvana, and Cognizant. I am grateful to my current company Ausy, France for allowing me to write this book and accommodating it in my working contract. Thank you for all hidden support provided. I am also thankful to all the bad experiences and critics I met during my professional life. Without you, I would not have worked harder and stayed more focused.

I would like to gratefully acknowledge Mr. Rudra Misra for his kind technical scrutiny of this book. My gratitude also goes to the team at BPB Publications for being supportive enough to let me take the creative liberty while writing the book and providing valuable feedback on the writing aspects. Last but not the least I would like to thank ME. Yes!! I am glad I was able to complete the book in time of crisis between working and travelling.

Preface

This book tries to make iOS development with Swift easy and fun. The book starts with an app to give the reader some confidence in creating an iOS app. Coding must be practiced rather than read. Every chapter has exercises and apps to get the user accustomed to the development environment.

This book focuses more on Swift, which is used in the real world, rather than trying to cover everything proposed by Apple. It aims at making you an iOS app developer with the necessary know-how of the Swift language. At the same time, it guides you to follow the standard naming conventions and good coding practices.

The book is divided into 10 chapters. They cover the basics of Swift and with each chapter, we dive deeper into the concepts. The 10th chapter introduces you to the SwiftUI framework. Often looked at as the future of iOS app development, this chapter is important. By the end of the book, you should be successfully developing your own idea into an app. With a little more practice, you would be able to find a job as a junior iOS app developer.

The book aims at teaching you Swift in a month. You can learn at your own pace, but here is a roadmap. The first 4 chapters, which are quite easy and cover the basics, can be done in the first week. You will finish them in no time, and this will give you some days to practice Swift. In the second week, you can cover

[chapters 5](#) and You can start creating your own simple apps. In the third week, you can move to [chapters 7](#) and These are important but by now, you will be able to easily understand complex concepts. In the last week, you can cover [chapters 9](#) and where you focus on the real-life debugging and error handling part of coding. At the same time, you will learn about the latest SwiftUI framework.

The details of the chapters are as follows:

[Chapter 1, Building your first](#) helps take a tour of Xcode and get acquainted with it. It teaches how to first create a small app. It gives you a brief introduction of the development and at the same time, motivates you to move ahead and learn more!

[Chapter 2, Swift programming](#) covers the basics of Swift programming. It teaches playgrounds and talks about how to use them to practice coding in Swift. It also covers the naming conventions.

[Chapter 3, Auto](#) teaches you how to use autolayout. It helps to create precise user interfaces and find the right placements for your buttons and labels. It covers the lifecycle of the view controller and different types of view controllers while building an application.

[Chapter 4, Types and control](#) dives into the concepts of Swift such as types and control flow. It covers the different ways of enumerations such as for loops and while loops. It also covers arrays and dictionaries. Though this is a bit theoretical chapter, you can still have fun with playgrounds.

[Chapter 5, Optionals and](#) covers very important and Swift-specific concepts such as optionals, unwrapping, and type aliases. It also covers access control. It teaches how to make a simple calculator app.

[Chapter 6, Code](#) teaches you how to structure your code. It also teaches you write methods and functions in Swift. On the app front, it covers navigation.

[Chapter 7, OOP in](#) revisits the concepts of object-oriented programming and see how they are implemented in Swift. It covers advanced app concepts such as extensions and design patterns.

[Chapter 8, Protocols and the Delegate Design](#) covers an important design pattern, the delegate design pattern. It teaches protocol-oriented programming, a paradigm Apple is trying to push with Swift. It also covers how to develop a complex app which has image capturing capabilities.

[Chapter 9, Error Handling and](#) helps complete your app developer learning curve. It covers the different ways to avoid errors, log them, and debug them. This is really helpful in real-world apps when you start working on existing apps.

[Chapter 10,](#) covers the latest SwiftUI framework. It is looked at as the future of iOS app development. Learning this will put you ahead of others in the game.

***Downloading the code
bundle and coloured images:***

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

<https://rebrand.ly/cdc8c4>

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

BPB IS SEARCHING FOR AUTHORS LIKE YOU

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

IF YOU ARE INTERESTED IN BECOMING AN AUTHOR

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Table of Contents

1. Building Your First Application

Structure

Objective

Setting up the environment

Creating your first project

Understanding Xcode

Running the application

Working of an application

Building the interface

Adding views

Finding help

Conclusion

Key learnings

Test your knowledge

2. Swift Programming Basics

Structure

Objectives

What is Swift

Playgrounds

Variables and constants

Type Inference

Commenting

Operators

Naming convention

App structure

Conclusion

Key learnings

Test your knowledge

3. Auto Layout

Structure

Objectives

Need for auto layout

Sources of changes

Understanding the layout

Size classes explained

Constraints

Adding multiple constraints

Editing a constraint

Unsatisfiable layouts

Ambiguous layouts

IBOutlets

View controller lifecycle

Types of views

Conclusion

Key learnings

Test your knowledge

4. Types and Control Flow

Structure

Objective

Basic types

String manipulations

Creating

Concatenation

Interpolation

[Iterating over strings](#)

[Counting characters](#)

[String indices](#)

[Inserting and removing strings](#)

[Substrings](#)

[String comparison](#)

[Type conversion](#)

[Collection type](#)

[Arrays](#)

[Creating an array](#)

[Modifying and accessing an array](#)

[Sets](#)

[Hash values for Sets](#)

[Creating a set](#)

[Modifying and accessing a Set](#)

[Dictionaries](#)

[Creating a dictionary](#)

[Modifying and accessing a dictionary](#)

[Tuples](#)

[Control Flow](#)

[If else](#)

[Switch](#)

[For in](#)

[While](#)

[Conclusion](#)

[Key learnings](#)

[Test your knowledge](#)

[5. Optionals and more](#)

[Structure](#)

[Objective](#)

[Simple calculator](#)

[Optionals](#)

[Unwrapping](#)

[If let and guard let](#)

[Defer](#)

[Control transfer statements](#)

[Type aliases](#)

[Access control](#)

[Understanding the delegate](#)

[Sophisticated calculator \(Exercise\).](#)

[*Adding an icon to the app*](#)

[*Preparing the layout*](#)

[*Making the calculator work*](#)

[Conclusion](#)

[Key learnings](#)

[Test your knowledge](#)

[6. Code Structuring](#)

[Structure](#)

[Objectives](#)

[App setup](#)

[Functions](#)

[*Methods*](#)

[Closures](#)

[*Closure expression*](#)

[Structures](#)

[Properties](#)

[Subscripts](#)

[Initializers](#)

[*Default stored values*](#)

[Customized initialization](#)

[Deinitializer](#)

[Navigation](#)

[Conclusion](#)

[Key learnings](#)

[Test your knowledge](#)

[7. OOP in Swift](#)

[Structure](#)

[Objective](#)

[Enumerations](#)

[Raw values](#)

[Implicitly assigned raw values](#)

[Initializing using a raw value](#)

[Associated values](#)

[Switch case over an enum](#)

[Iterating over an enum](#)

[Classes](#)

[Classes vs structures](#)

[Initialization of a class](#)

[Structures and enumerations are value types](#)

[Classes are reference types](#)

[Identity operators](#)

[Inheritance](#)

[Base class](#)

[Subclass](#)

[Overriding](#)

[Type casting](#)

[Type checking](#)

[Down casting operator](#)

Type casting for Any

Generics

Extensions

Computed properties

Extending initializers

Custom operators and operator overloading

Design patterns

Advantages of design patterns

Drawbacks of design patterns

Model view controller

MVC in iOS

Drawbacks of MVC

Conclusion

Key learnings

Test your knowledge

8. Protocols and the Delegate Design Pattern

Structure

Objective

Protocols

Protocol syntax

Defining a protocol

Protocol-oriented programming

Protocols as types

Protocol extensions

Protocol inheritance

Delegate design pattern

Word Bank App

Configuring the table view

Creating the second view

[Adding new words](#)

[Adding images](#)

[Updating the app](#)

[Making the app better](#)

[Conclusion](#)

[Key learnings](#)

[Test your knowledge](#)

[9. Error Handling and Debugging](#)

[Structure](#)

[Objective](#)

[Memory management](#)

[Automatic Reference Counting](#)

[Optional chaining](#)

[Error handling](#)

[Throwing functions](#)

[Using try-catch](#)

[Debugging](#)

[Logging](#)

[Conclusion](#)

[Key learnings](#)

[Test your knowledge](#)

[10. SwiftUI](#)

[Structure](#)

[Objective](#)

[Introduction to the SwiftUI framework](#)

[Design and layout in SwiftUI](#)

[Creating a project](#)

[Playing around with views](#)

[*Adding custom views*](#)

[User actions](#)

[List views](#)

[Navigation](#)

[SwiftUI and UIKit](#)

[Conclusion](#)

[Key learnings](#)

[Test your knowledge](#)

[**Index**](#)

CHAPTER 1

Building Your First Application

Welcome to ‘ *Learning swift in 30 days* ’. During these 30 days, we will learn Swift and alongside learn to build basic iOS applications. You will familiarize yourself with the iOS application development environment. As we do not have enough time, let us jump right away into building our first application!

Structure

In this chapter, we will cover the following topics:

Setting up the environment

Creating your first project

Understanding Xcode

Running the application

Working of an application

Building the interface

Finding help

Objective

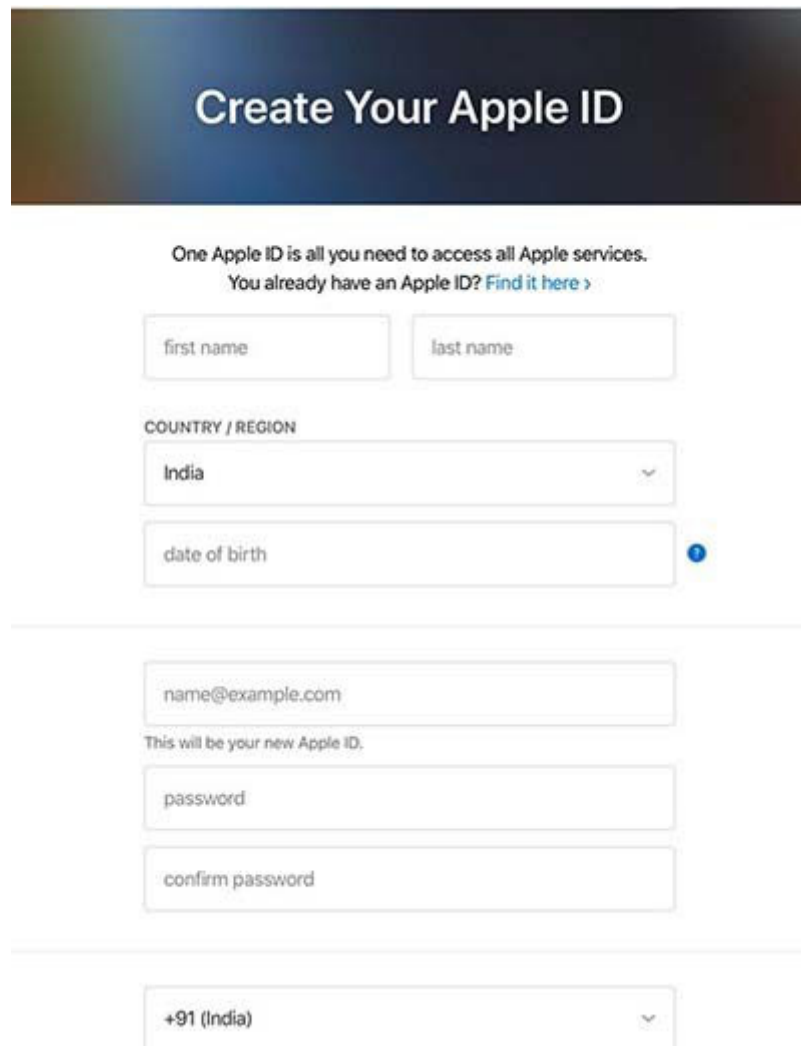
The objective of this chapter is to give you some initial confidence. We will do this by introducing you to the Xcode environment and then building a simple application. By the end of this chapter, you should be able to work easily with Xcode and create an iOS project.

Setting up the environment

One of the characteristic features of Apple is its bubble. You can enter this bubble and have all the Apple services. This holds true even for application development. If you want to develop an iOS or a macOS application, you will need a mac OS. Following are the steps to set up the environment for iOS application development:

You need a mac system with a minimum OS version 10.15.

Then, you need to create an Apple account. Go to <https://appleid.apple.com/account> as shown in [Figure](#)

The image shows the 'Create Your Apple ID' web page. At the top, a dark blue banner contains the title 'Create Your Apple ID' in white. Below the banner, a message states: 'One Apple ID is all you need to access all Apple services. You already have an Apple ID? [Find it here >](#)'. The form consists of several input fields: 'first name' and 'last name' (side-by-side), 'COUNTRY / REGION' (a dropdown menu with 'India' selected), 'date of birth' (with a blue information icon to its right), 'name@example.com' (with a note below it saying 'This will be your new Apple ID.'), 'password', and 'confirm password'. At the bottom, there is a dropdown menu for the phone number prefix, currently showing '+91 (India)'.

Create Your Apple ID

One Apple ID is all you need to access all Apple services.
You already have an Apple ID? [Find it here >](#)

first name last name

COUNTRY / REGION
India

date of birth ⓘ

name@example.com
This will be your new Apple ID.

password

confirm password

+91 (India)

Figure 1.1: Apple account Sign up page

Complete the preceding steps to create your app store account. They may ask for credit card details. Don't worry you won't be charged.

Once your app store account is set up, go to the Apple App Store by clicking on the icon in Applications as shown in [Figure](#)

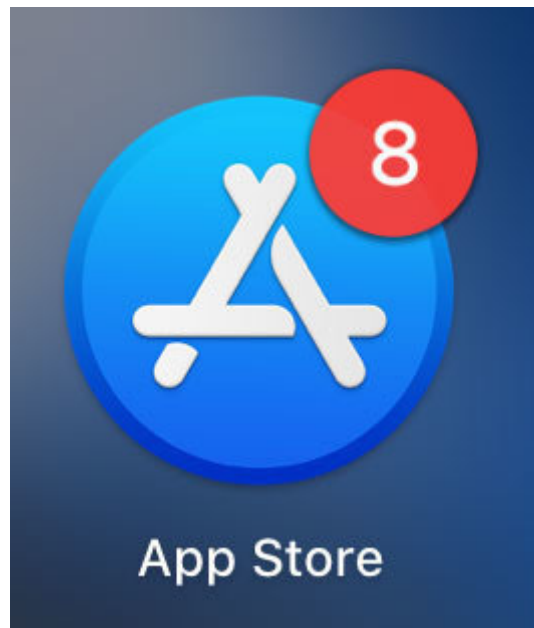


Figure 1.2: *App Store icon*

Then, type “Xcode” in the search box. Here, you will need the Apple account created above to download the latest version of Xcode. Make sure you have at least 20 GB of free space on your Mac.

Once installed, you should be able to see the Xcode icon in your applications.

We will build simple applications, so you won’t need an iPhone. Xcode comes with a simulator.



Figure 1.3: *Xcode icon*

[Creating your first project](#)

We will create our first project. This will help you understand the different elements that go into making of a project. The end product is a simple application with an image. No complicated logic or even coding! You will even understand the Xcode development environment and after reading this chapter, you will be easily able to navigate through Xcode.

Let us create our brand new Xcode project:

Click on the Xcode icon. You should be able to see a window similar to [Figure](#). The version (red box) might be different from yours but a minimum of 11.x is recommended. Here, we choose a starting template for our iOS. We are new developers, so let us stick to the simplest form, *Single View*

×



Welcome to Xcode

Version 12.4 (12D4e)



Create a new Xcode project

Create an app for iPhone, iPad, Mac, Apple Watch, or Apple TV.



Clone an existing project

Start working on something from a Git repository.



Open a project or file

Open an existing project or file on your Mac.



Show this window when Xcode launches

Figure 1.4: Xcode launch window

Next, click on the second option (green box) to start creating our first iOS project. You should then be able to see a window like the one in [Figure](#)

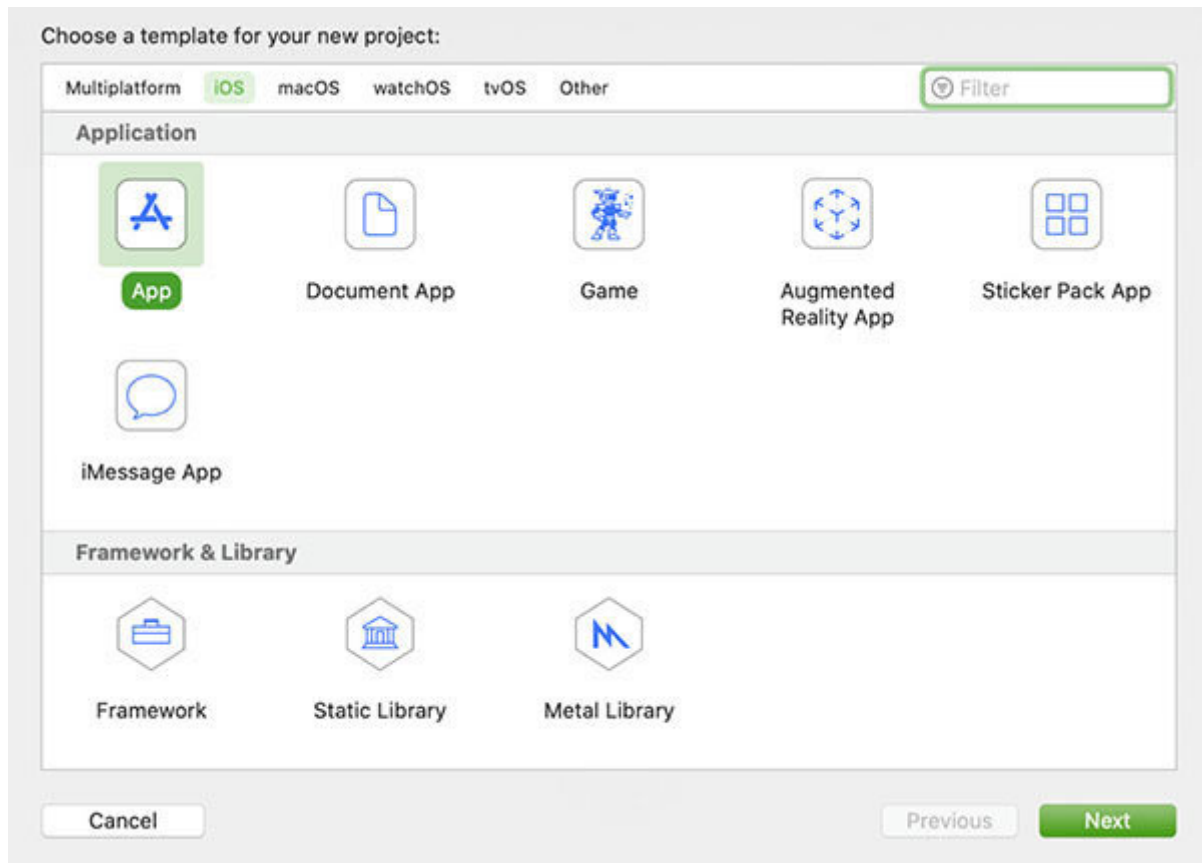


Figure 1.5: *Template selection for new project*

The other templates are up to you to explore. But the single view application is the most basic and customizable template available.

Once you click on you will see a window as shown in [Figure](#)

Choose options for your new project:

Product Name:

Team:

Organization Identifier:

Bundle Identifier:

Interface:

Life Cycle:

Language:

☐ Use Core Data

☐ Host in CloudKit

☐ Include Tests

Cancel Previous Next

Figure 1.6: Choose options for your new project

Before you create a new project, we have to specify some details. So, let us understand what details we need to fill in:

Product Name: This is the name of our application. This is a demo application so it is not so important right now. But in future when you develop applications for the App Store, this name is very important. Let us name our application

Team/Organisation Name: This can be For real-world applications, this option is often the Team account of your company and the organisation name is your company name. Well, right now you are

your own boss so go ahead and put your name. If you have an Apple developer account, you can link it and then you can see your own name under teams.

Organisation/Bundle Identifier: This is a unique identifier for the application. It is the web address equivalent for an application. You can see the required format in tiny grey letters below the box. You just have to put the first two parts:
com.yourCompanyName.

Interface: THIS IS IMPORTANT. Set this to Storyboard. We will make SwiftUI applications later. But for now, we will work with Storyboards.

Language: Swift! If you want to choose Objective C, this is the wrong book for you.

Once done, your window will look something like the one in [Figure](#) Now, click on and choose where you want to save the project. That's it! We have created our first iOS Application project. Congratulations!!

Choose options for your new project:

Product Name:

Team:

Organization Identifier:

Bundle Identifier:

Interface:

Life Cycle:

Language:

☐ Use Core Data

☐ Host in CloudKit

☐ Include Tests

Figure 1.7: Filled options for new project

Understanding Xcode

So what is Xcode? Xcode is an **Integrated Development Environment** by Apple. It has the complete developer toolset to develop software for macOS, Apple watch, Apple TV, iPhone, and iPad. It is the **ONLY** way to develop applications for the Apple environment. This makes it our only option and we must know it well.

Our new project is ready and by now, you should see a window like the one in [Figure](#). What you are looking at is the project settings. These are quite self-explanatory, but diving deep into the project settings is not our aim. Next, click on the file which is in the pane of the left-hand side of the screen.

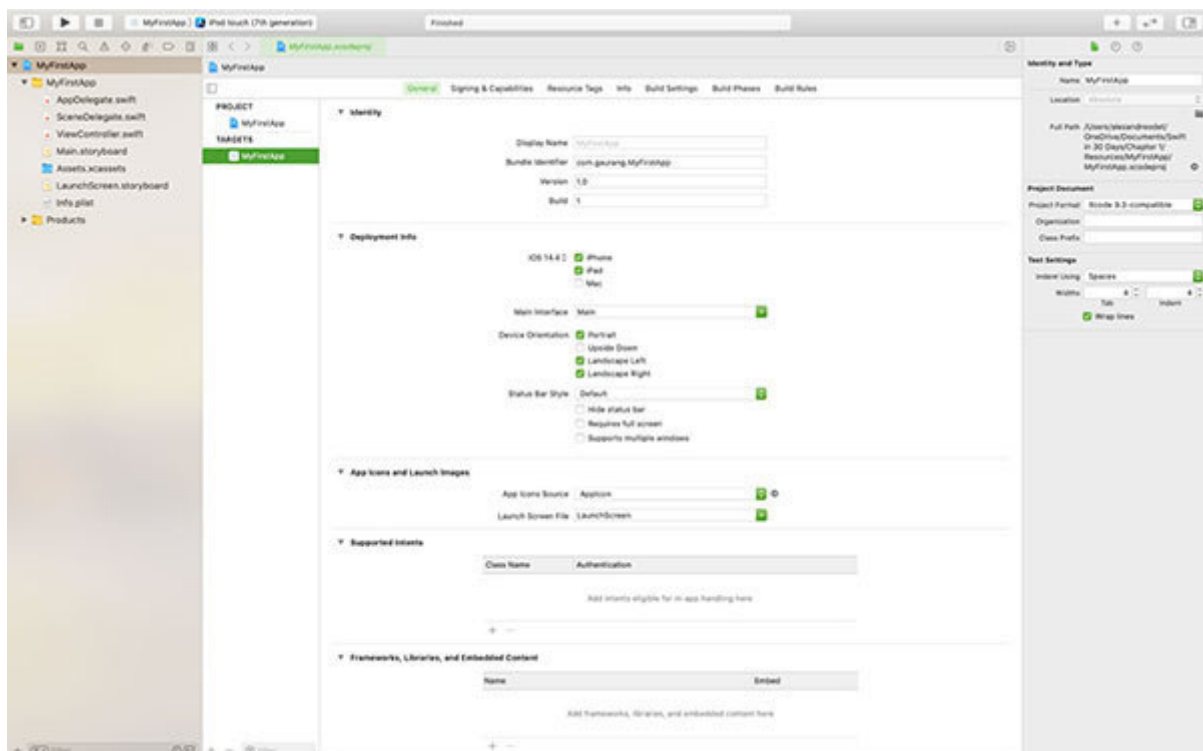


Figure 1.8: *New project landing*

Now, you will see Xcode the way we will be using it. So, let us take a small tour of this environment. Look at [Figure](#) This shows the Xcode screen for version 11.x:



Figure 1.8.1: *Different panes in Xcode 11*

[Figure 1.8.2](#) shows the Xcode screen for version 12.x. There are small differences which we will discuss:



Figure 1.8.2: Different panes in Xcode 12

You can see the different areas in the Xcode. You can hide or show these areas by clicking on the three buttons in the right-hand corner as shown in [Figure](#). Or in Xcode 12, you will see these buttons at the two corners of the status bar. The middle button, to hide and show the debug pane, is however missing in the Xcode 12, but this can be done by clicking on *View > Debug Area > Activate*



Figure 1.9: Status bar buttons

As you can see the editor pane remains constant. Following are the zones:

Status This shows the status of your application. Like *Build Ready* and also the errors. It mainly comprises the buttons to run the application and open and hide the different panes as shown in [Figure](#)

Navigation From here, you can navigate through your files and do much more. You might wonder what the small icons in [Figure 1.10](#) do?



Figure 1.10

Let us find out. The top part of the image is from Xcode 11 and the bottom from Xcode 12. Though the icons have changed, the functionality remains the same. Here are the different buttons and their functions:

Project This helps you navigate through your workspace.

Source control If you are using version control like Git, here you can see all the history of changes to your files.

Symbol This helps view the class hierarchy of your project.

Find This helps you find keywords. You can make them case sensitive and even change the scope of where you want to search. The find option can be changed to replace, if you want to find and replace.

Warnings and The place where you don't wish to see anything. Here, you can find your build time and run time issues and warnings.

Test Here, you can see all the unit tests you have written. You can create, run, and disable unit tests from here.

Debug In this space, you can analyse the running threads and stack information at specific times when the application is running.

Breakpoint This helps navigate through all your project breakpoints. You can enable or disable them and even put conditions.

Report This helps you see the history of all your builds.

Debug The debug pane consists of two views: The console view and the variable view. More about this in the [chapter 9, Error Handling and](#)

Editor This is the main area where you can see your code.

Inspector This is really useful when you are working with the interface builder, which we will discover in the next section. Refer to [Figure](#)



Figure 1.11: *Inspector pane buttons (Xcode 11 vs Xcode 12)*

Following are the functionalities of the inspector pane buttons in the above images:

File This helps to manage and view the tags, source, file path, target, and so on for the file.

Version This helps to see the history of changes made when using versioning.

Help This helps to see the Apple documentation for a selected item.

Attribute This helps to change the attributes of the view (when using interface builder).

Running the application

We are making an iOS application. The only way to test this application is to run it. The final of the application is of course the iPhone or iPad. But as a developer, you can even run this on your own machine using a simulator. Let us understand both.

IOS SIMULATOR

We will use the iOS simulator for this book, as it does not require any initial setup.

The Xcode comes with a lot of pre-loaded simulators as shown in [Figure](#) You can use the device of your choice:

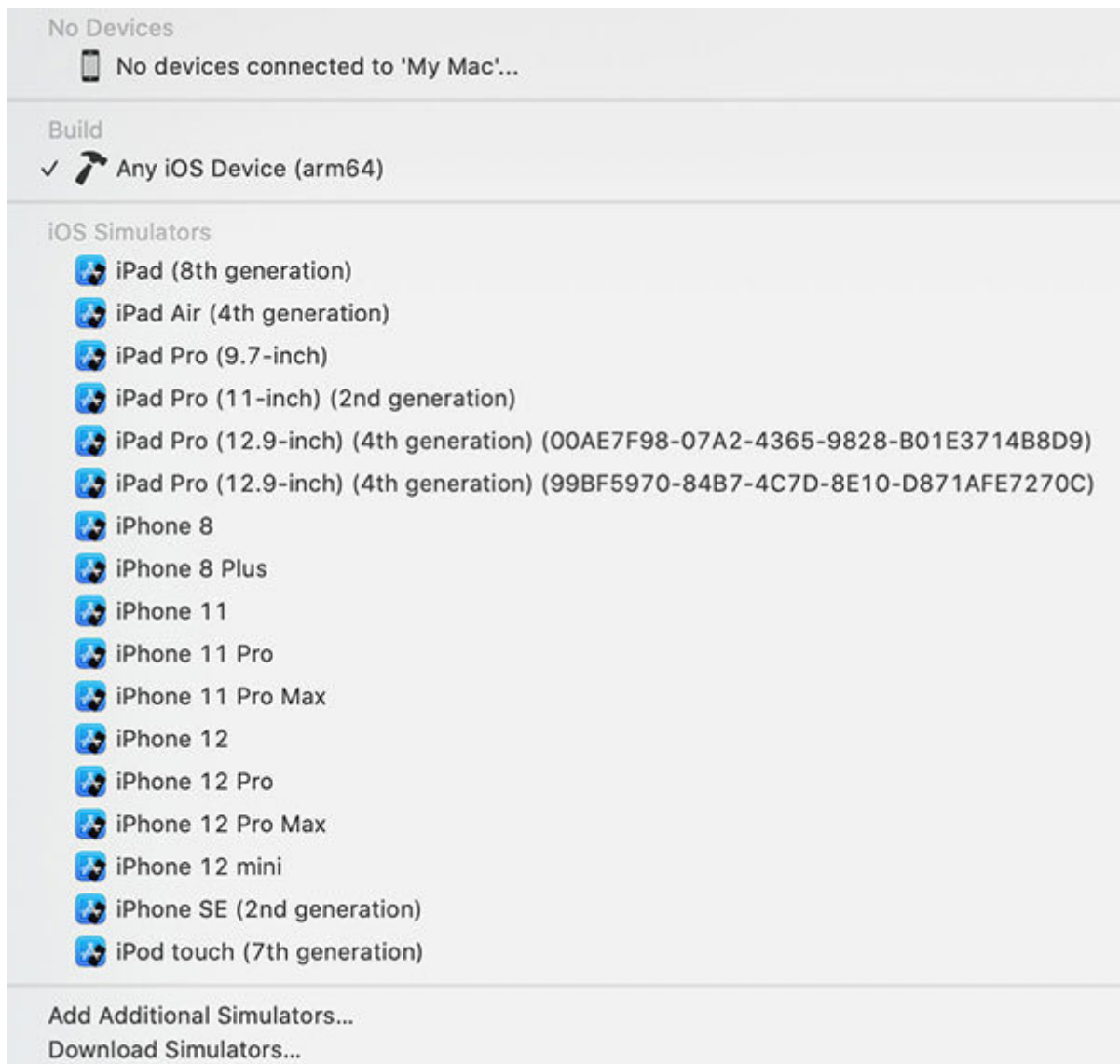


Figure 1.12: *Choosing the right device or simulator*

You can drag and move the simulator around. Or, also resize it using your mouse. If you click on you will even see options to rotate or take a screenshot, as shown in [Figure](#)



Figure 1.13: Options to manipulate simulator

If you do not find the device in the list, you can even add or download simulators by clicking on the options at the end of the list. Remember that the simulators take space.

You can press the Play button or click on *Cmd+R* to run your simulator. The first time takes a while, so do not worry. There is a big difference between a simulator and a device. The simulator emulates the eventual device environment. But it uses your machine's Intel chip instead of the ARM chip of a device. This is also the reason why with the new M1 chip macs you can run your iOS applications on the mac machine.

The device also has many other hardware features, like the camera, which the simulator does not have.

RUNNING ON A DEVICE

If you already have an iPhone or iPad, you can use it to run the applications. As soon as you connect the device, it will show up at the top of the list as shown in [Figure](#)



Figure 1.14: *Choosing a device*

But because your phone has sensitive information, we cannot run our application just by clicking on play. Following are the steps to run the application on a device

Check that the iOS version and Xcode version This is the first step. Your iOS and Xcode versions should match. You can go to **Settings | General |** to check the software version of your iPhone. Your iOS version on phone cannot be higher than that of the Xcode. For example, Xcode 12 supports iOS 14 and lower.

Add an Apple developer For this, you can use your existing Apple store ID which we created to download the Xcode. To enter your application ID, go to the Xcode preferences. Then, click on accounts as shown in [Figure](#)

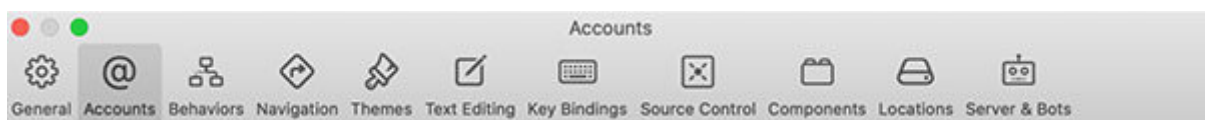


Figure 1.15: *Xcode preferences*

Then, click on the add (+) button as shown in [Figure](#)



Figure 1.15.1: Adding an apple account

You will then see a window as shown in [Figure](#). Click on the Apple ID option and enter your credentials:

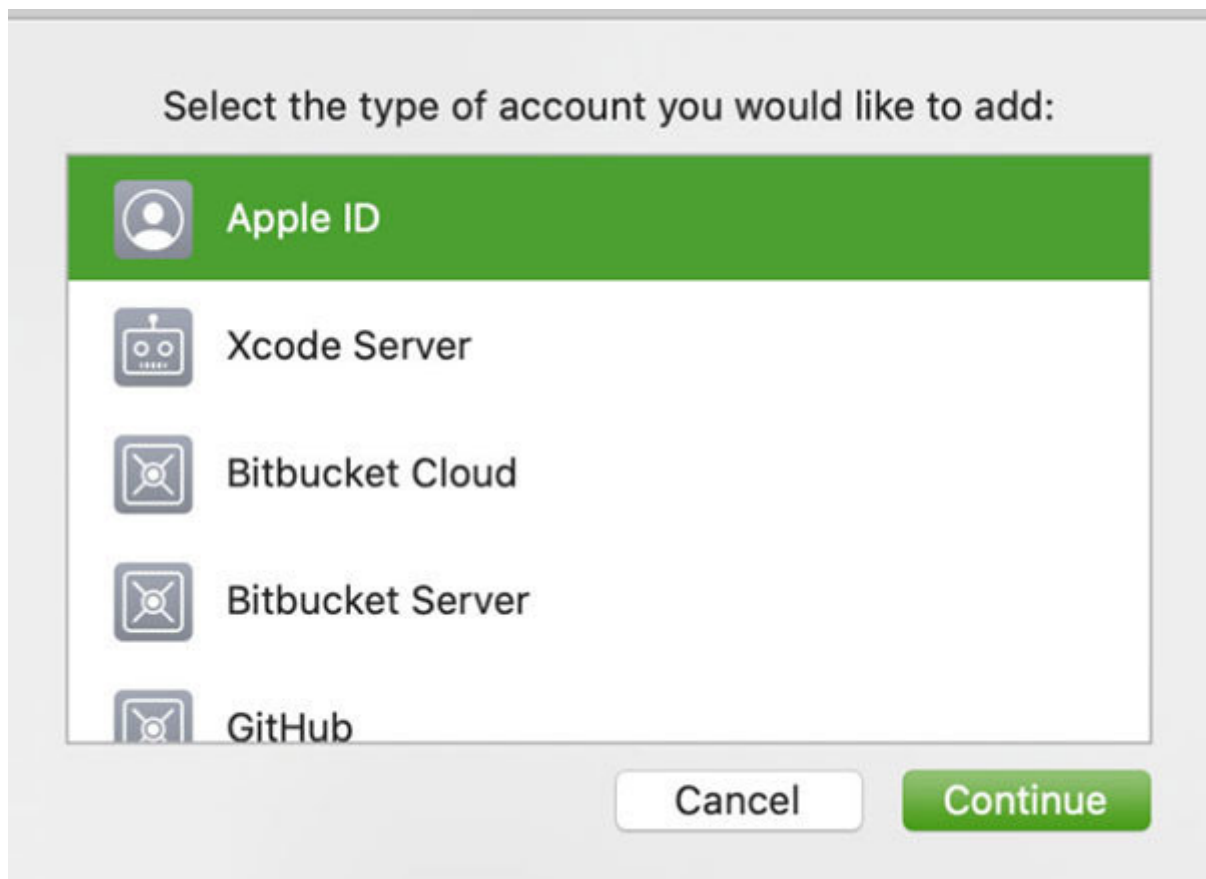


Figure 1.15.2: *Selecting a type of account*

Sign the To do this, click on the **Project | Target | Signing &**
Refer to [Figure](#)

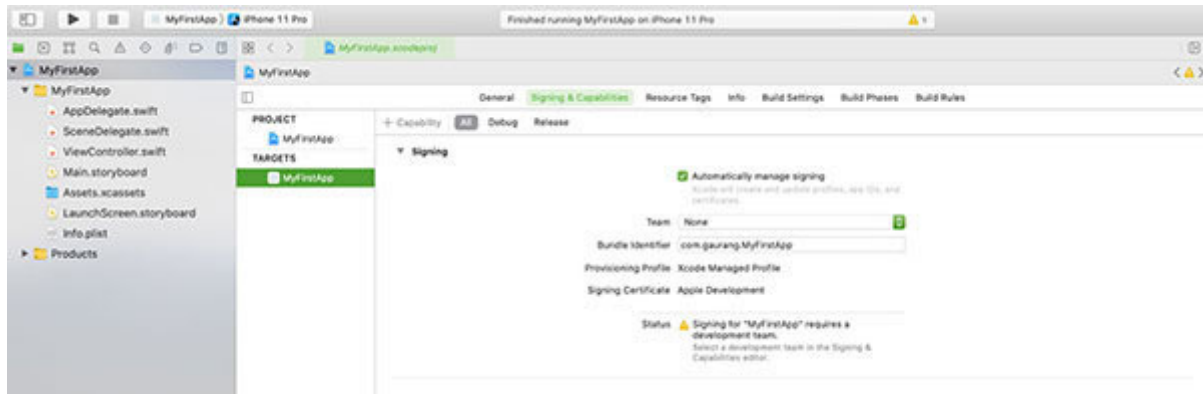


Figure 1.16: *Automatically manage singing*

Make sure that the automatically manage signing is checked. In the team drop down, you should be able to see your own personal team. If not, make sure you have logged in properly. Select your team and Xcode does the rest.

Connect the physical Once you are through with the preceding steps, connect your device. If it is the first time you are connecting your iPhone to the computer, you might get an alert on your iPhone asking if you the device. Click on yes. If there is a mismatch in the version of the Xcode and iPhone, Xcode will let you know.

Run your Now all you need to do is build and run your application.

Working of an application

Before we move ahead, let us first understand what is an application? App is an abbreviation of the word application. In simple words, it is a software which runs on your phone. This software can come preinstalled with the hardware or can be installed from other sources. This program helps you exploit the hardware. Let us look at an example:

You open an app and see a button

When you click on this button, the sensor detects and registers many factors, like the pressure, for how long it was clicked, and so on.

This information is then sent to the OS, in our case iOS.

The OS in turn uses this information and asks the code of the app, what to do next?

The code then responds with a change in state, like updating the colors, or showing another screen, and so on.

Let us look at another example:

You pick up your phone and open the “Phone” app.

You then search a number or dial it.

The OS then looks up for the number in the contacts and makes the call.

Here, we are dealing with data. The phone number here comes from a set of data already present on the phone.

Thus, it is important to remember that the app can be divided into three parts. Screen, Code, and Data as shown in [Figure](#)



Figure 1.17: App division

The data can have different forms. It can either come preinstalled in the app. It can be already present in the phone in other apps. Or, it can be downloaded from a backend server.

In the latter case, the server is the **backend** and your app serves as the front end.

So how can we build screens for our apps? Apple makes this very easy for us by providing us with UIKit. It is a framework where you can build windows with views having interactive and attractive

interfaces. What is a window? Imagine you are standing next to a real-world window. What happens? The colour of the sky changes and people are moving. Does the window move? No. This is exactly what the window in the app does. It shows you different views from its window. All the apps have only one primary window. Recently, Apple introduced ways to create auxiliary or supporting windows to manage different instances of the same app.

What is a view? Everything you see on the phone is a view. If it is a button, it is nothing but a view that accepts touch. If it is a text field, it is a view that accepts text as input. Thus, UIKit provides us the window-view architecture to handle our touches and render our interfaces. As we move forward, we will also learn the code and the data part.

Remember, the iPhone is a small powerful computer. But it has a limited battery life as compared to the computer. So, your apps should be efficient and not use the resources too much. The iOS takes care of this. If your app is consuming more than it should, the OS terminates your app. So, remember the OS is like a manager of your app.

Let us build our first interface!!

Building the interface

Now, coming back to our project, we will try to build the screen part of our app. Here are the steps:

Go ahead and click on the file called You will see something as shown in [Figure](#). This is called the Interface Builder. It does exactly what its name suggests; it builds the interface. The left-hand side is called the document outline. It is here that you can see all the views you have added on your view controller. So why is it called a Storyboard? Because here you can add multiple viewcontrollers and connect them. Thus, you are telling a story of how your app will navigate from one view to another. You can have multiple storyboards in a single project:

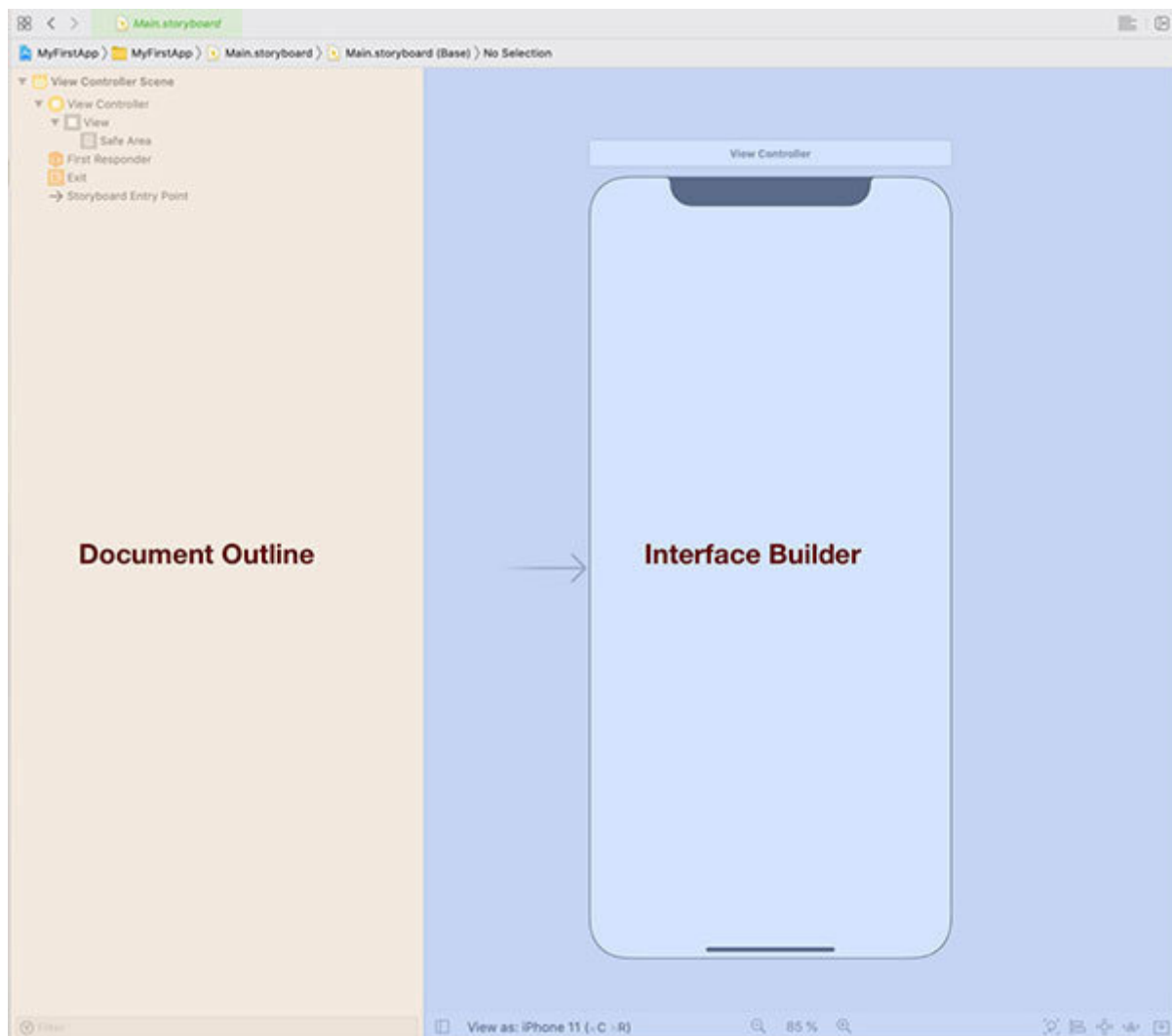


Figure 1.18: Storyboard space

In the document outline, you also see a view highlighted in green. That is the view of your app, and on this view, you will add other images, buttons, and so on.

Now, go ahead and click on the button in the left-hand side corner of the status bar. Just see to it that a default simulator is selected. It takes some time when you are running the simulator for the first time. The subsequent runs are faster. Try to keep the same simulator throughout this book. Every simulator takes its

own space. So what did you see? An empty white screen as shown in [Figure](#) Let us change this:

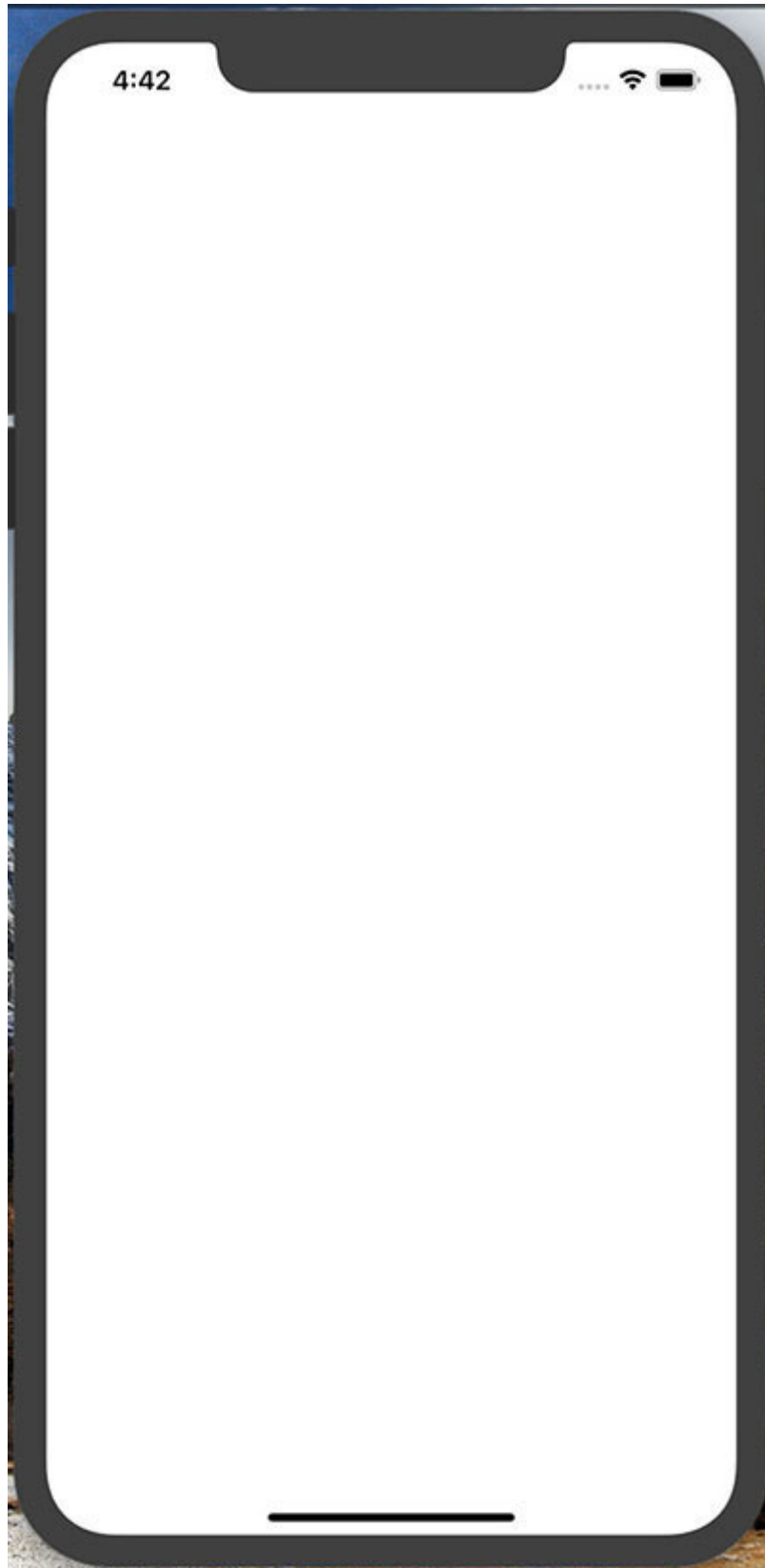


Figure 1.19: *Empty white screen on first launch*

Go back to the main storyboard and click on the view.

Now, go to the inspector pane and click on the attribute inspector. Go to the Background option and select the colour you like.

Stop the build, the button next to play, and re run the app. You will see the app with your selected colour as shown in [Figure](#) (You will see some new options next to the attribute inspector button. We will cover them in the upcoming chapters.)

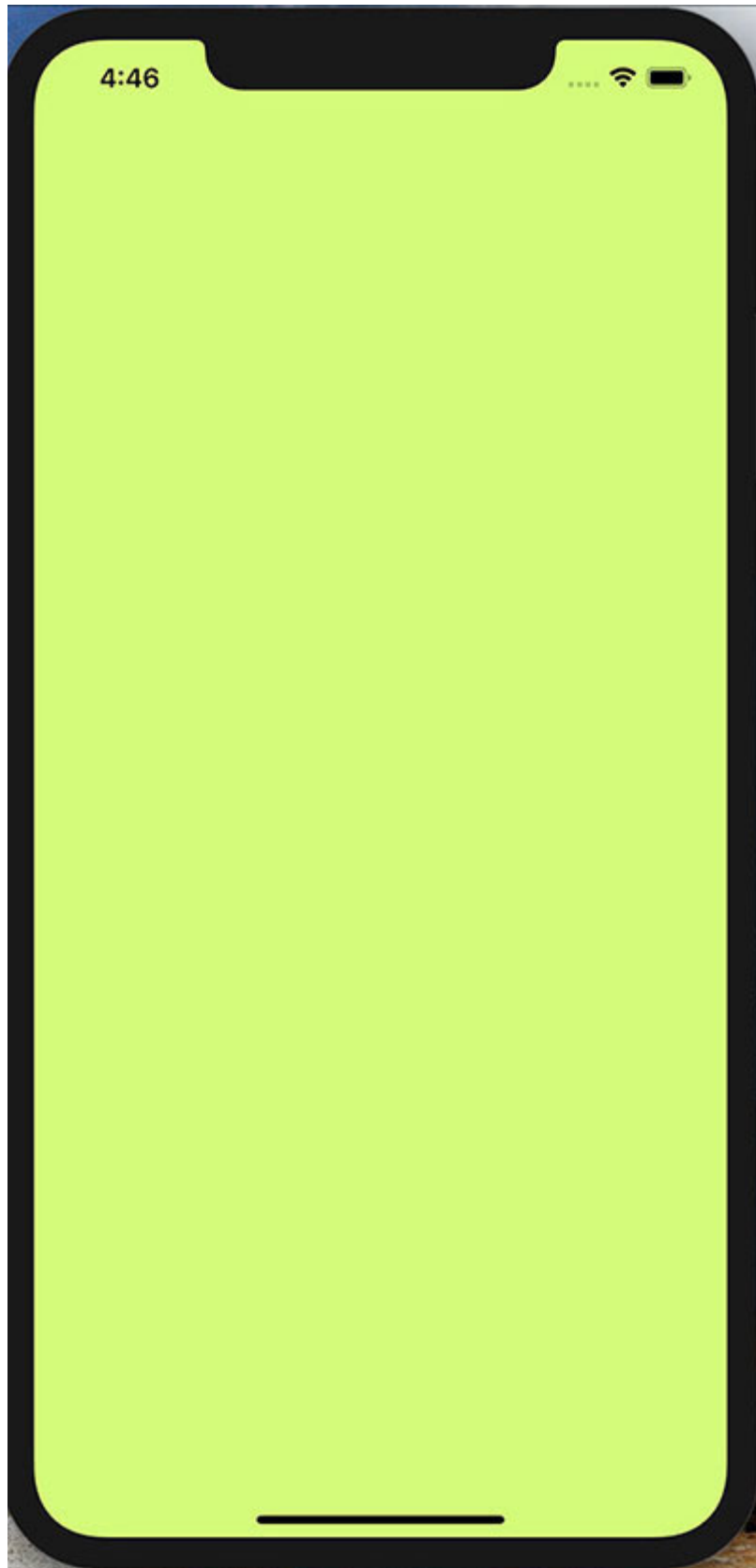


Figure 1.20: *Screen after changing background color*

Use shortcuts to Build Stop and Run builds. More shortcuts can be found by clicking on the options next to Xcode in the top bar. Go ahead and try.

You will see a white screen before your colored view in the simulator. Try changing it. `Launch.storyboard`

Adding views

Only if the iOS app development was about changing background colors!! The next step is to add views. Where do you find them?

Click on the “+” button in the status bar. A window should open up, like the one in [Figure](#). This is the library. It has all the UI components from the UIKit you can add. Let us start by adding a simple image.

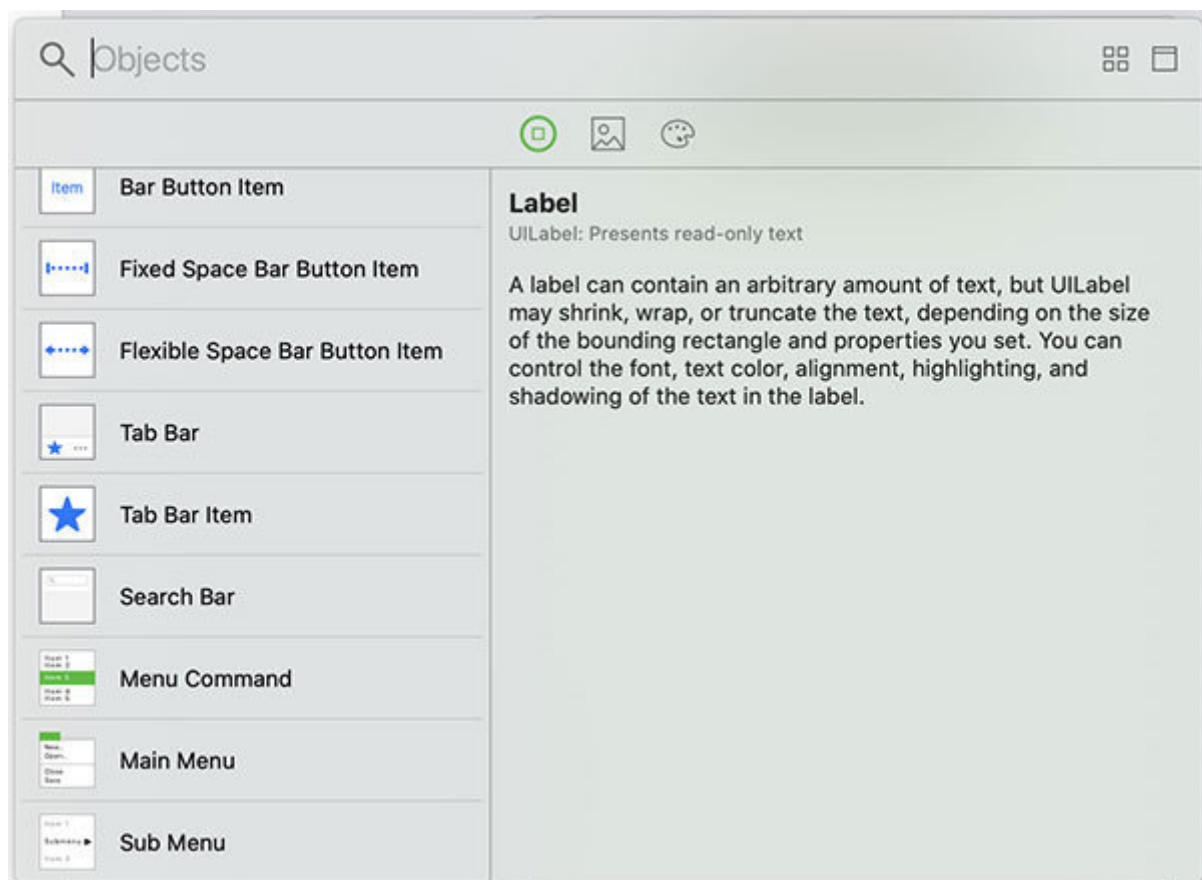


Figure 1.21: The UIKit library

Type in the library and you will see the option Image View. Drag and drop it in the center of your app as shown in [Figure](#)

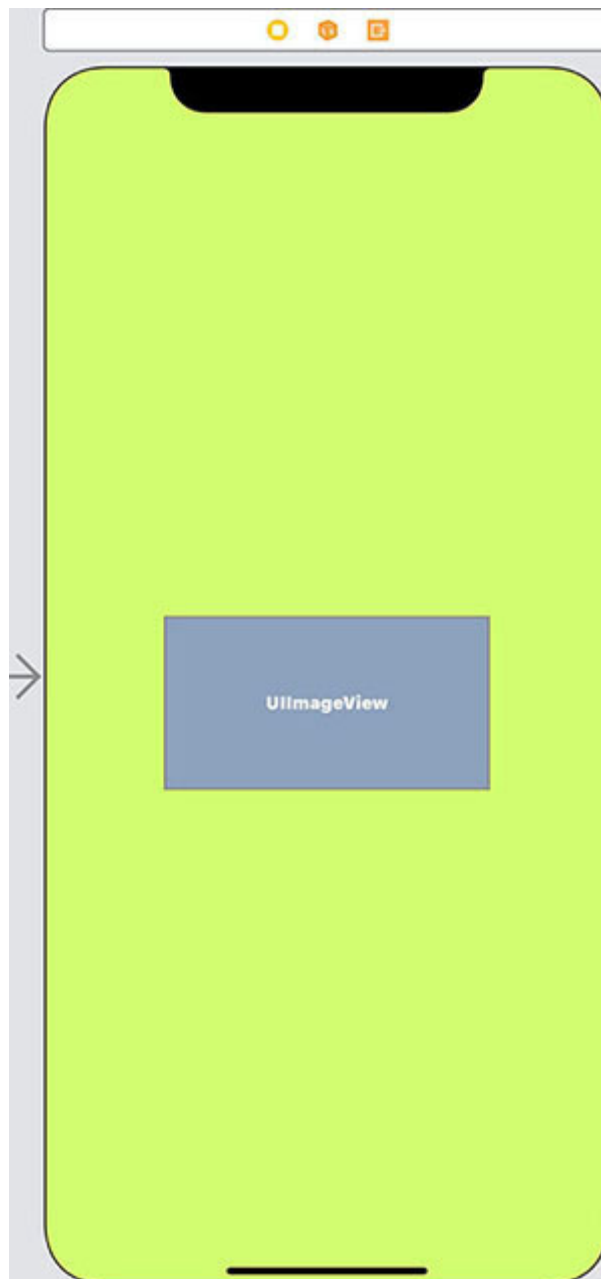


Figure 1.22: Adding an image view

Now, when you run the app, you won't see anything. So, add a background color to the image view to confirm and rerun the app. You will see something as shown in [Figure](#) But what is the use of an image view if there is no image. Let us fix that.

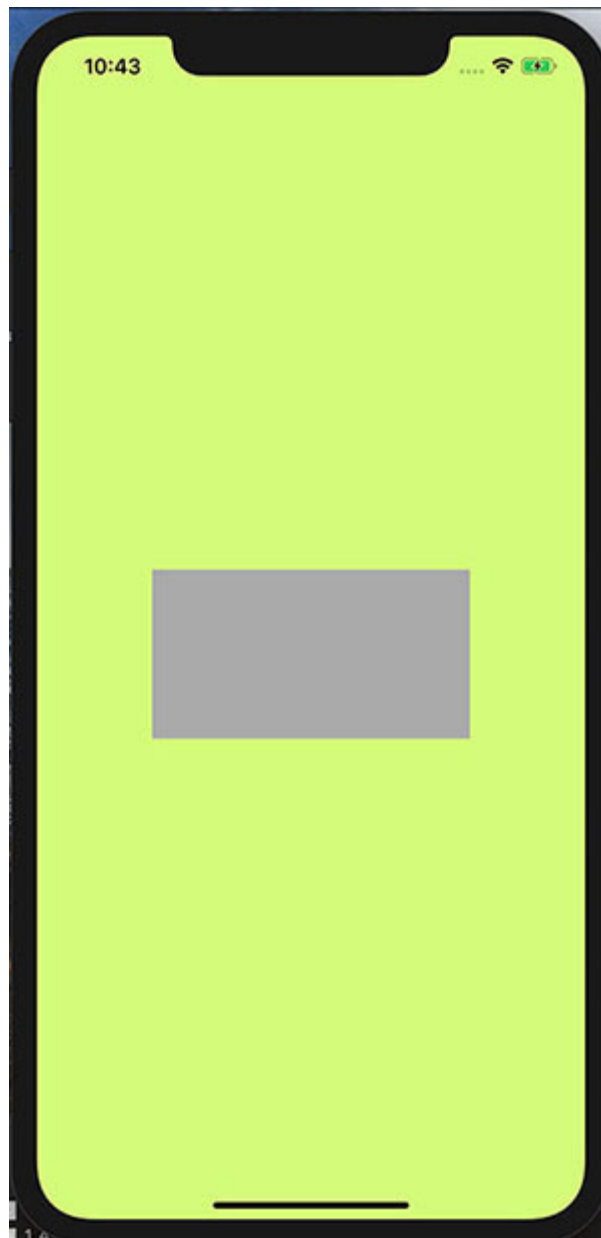


Figure 1.23: Testing if image view has been added

ASSETS CATALOG

The assets catalog is where you can put all your images and even color sets.

Click on the “+” sign at the bottom of the document pane and you will see all the options available. Select **Image** Now, name this image set as You will see three empty boxes next to it with 1x, 2x, 3x as shown in [Figure](#) These are nothing but the different image resolution images we need for different iphones. In words of Apple documentation, *you have a standard resolution @1x image that's 100px by 100px, for example. The @2x version of this image would be 200px by 200px. The @3x version would be 300px by*



Figure 1.24: Image set

Just select any random image from your desktop and drag and drop it three times in the three boxes as shown in [Figure](#)

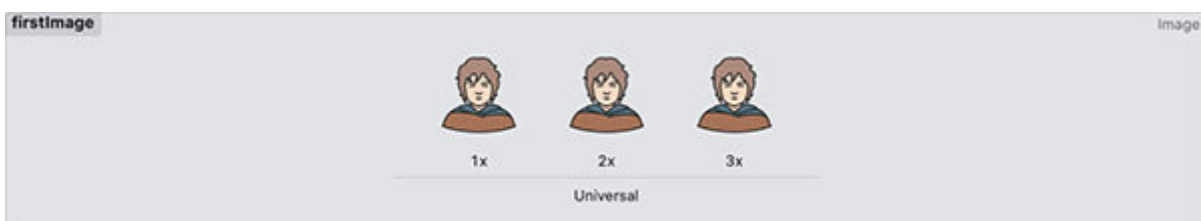


Figure 1.25: Image set with images

Now, go back to the **Main.storyboard | Image View | Attributes manager | Image** | You can even change the background color to clear. That's it!! Now, run the app. You will see something as shown in [Figure 1.26](#) but with your image:

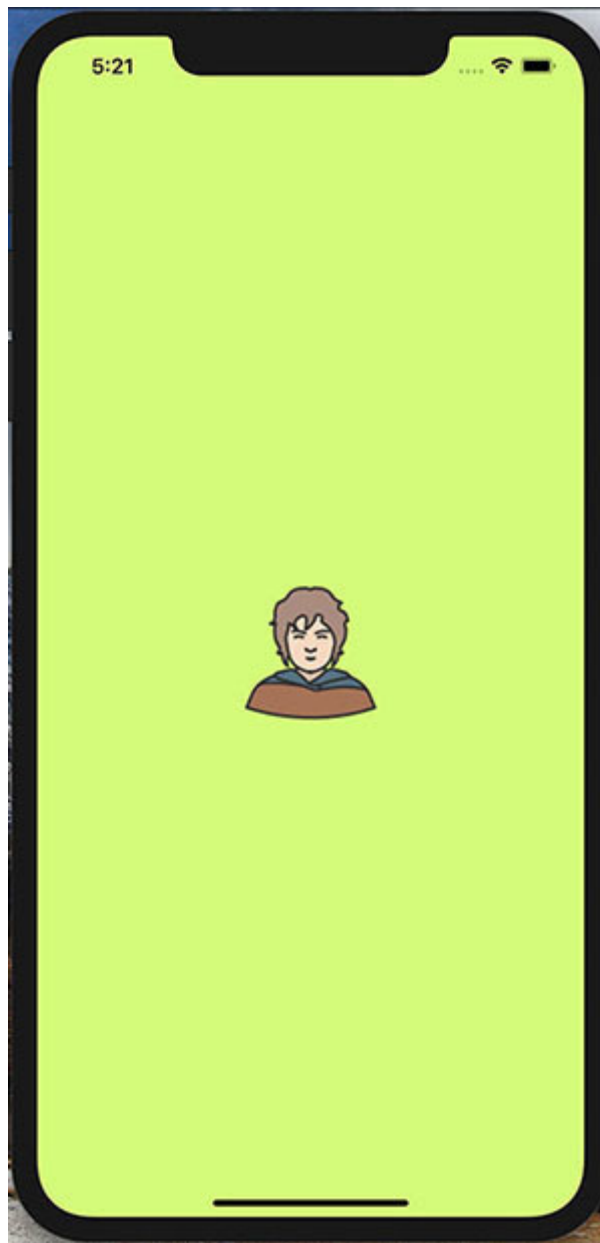


Figure 1.26: Screen with added image

Create an app with a launch image. When you open the app, it should show your image with your name under it. Use label.

Finding help

The biggest problem about working with new development environment is the errors. If you are learning from a book, it gets more difficult to seek help. But don't worry. Here are some tips to solve errors in case you are learning on your own:

The first step is to *read* the error. Most Apple errors tell you exactly what is wrong.

If you do not understand it, copy and paste it in Google and it will show you the top results.

The next best option to seek help is stack overflow. It's a forum where all developers find help. So, be sure to put the tag or An example would be *Xcode not showing navigation* Or you can very well bookmark the following link.

The other place where you can find help is the Apple documentation. You can click on any word and a small documentation window will open in the Inspector Pane -> Help Inspector as shown in [Figure](#)

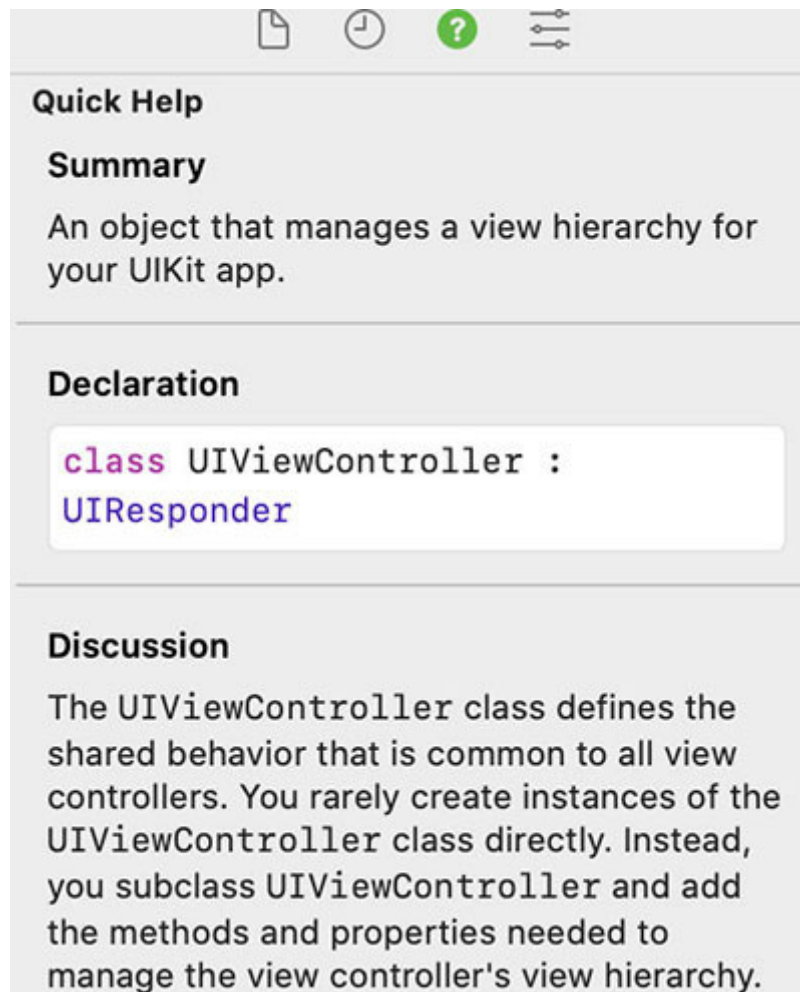


Figure 1.27: Quick Help window

If you don't find what you need, as shown in [Figure](#) go ahead and click on the button. Develop a habit of reading Apple documentation.

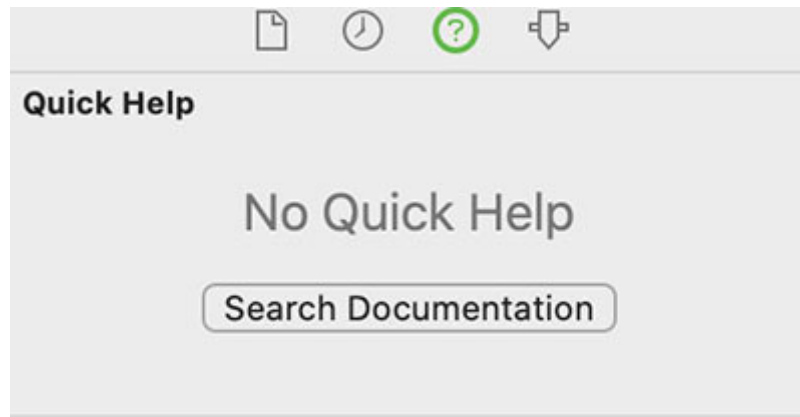


Figure 1.28: *Search documentation button*

Apple developers hardly get stuck. Almost all the errors have been encountered by someone. For beginners, it is best to just copy and paste your error in Google and it will do the rest. Of course, it is not going to come and fix your problem, but rather give you results of forums where you can find the answers. Apple has its own developer forum which you can

As you become a good developer, try answering questions on these forums. It is the best way to enhance your knowledge about the system.

Conclusion

So here ends our first chapter. Hope you are excited to learn more! I am sure you have a lot of questions in your mind and a million-dollar app idea to implement. But Rome was not built in a day!! By now, you should be confident to create a new app and navigate through Xcode. So, the next time when I write *a simple view* you should not feel confused or lost. Next, you learned what the different zones of Xcode are and you are familiar with the terms such as Attribute Inspector and Help Inspector. Lastly, you learned where to seek help in case you get blocked while learning Swift and iOS app development. In the next chapter, we will learn some Swift basics to familiarize ourselves slowly with the syntax.

Key learnings

We started the chapter with creating a project. We learned the different steps and information we needed to create our first project.

We then went on a tour of Xcode and got familiar with this indispensable tool.

We also saw how to deploy your app on iOS Simulator and iOS Device.

We then understood how an application works.

We also created some simple UI with Interface Builder and learned to add images to the project.

We ended the chapter by understanding how to find help, in case you get stuck while working through this book.

Test your knowledge

What is a Bundle Identifier?

This is a unique identifier for the app. It is the web address equivalent for an app. The required format is :
com.yourCompanyName.

What are the different panes in Xcode?

Navigation Pane

Debug Pane

Inspector Pane

Editor Pane

Which is the framework provided by Apple to build the interfaces?

UIKit

What is the entity which holds the images called?

Assets Catalog

What are the two spaces of Interface Builder called?

Document Outline

Interface builder

Name any three types of navigators.

Project Navigator

Find Navigator

Symbol Navigator

What is the screen which appears before your main screen called?

Launch Screen

Which 3 parts can the app be divided into?

Screen

Code

Data

How do we check information about any keyword in the code?

Select the word and seek help from the Inspector Pane.

What is the forum to find help called?

Stackoverflow

Swift Programming Basics

In the first chapter, we warmed up with a simple iOS project. But our main aim is to learn Swift. So, let us learn some Swift basics in this chapter. Slowly, we will dive deeper and deeper into the Swift world. We will first learn about Swift and then move to *Playgrounds* , an important tool for learning Swift. Then, we will learn about some basic concepts which will come really handy moving forward. We will also learn about commenting and naming conventions to make our code more readable and understandable. We will end the chapter by understanding the iOS App structure.

Structure

In this chapter, we will cover the following topics:

What is Swift?

Playgrounds

Variables and Constants

Type Inference

Commenting

Operators

Naming convention

App structure

Objectives

The objective of this chapter is to initiate you in Swift. We will learn to work with playgrounds, where you can practice Swift programming. We will learn about variables and constants. We will also learn about operators in Swift and see how collections work. At the same time, there would be some exercises for practice. The other objective is also to help you understand the structure of the iOS app. So, next time you would know what classes do, and know terms like targets, libraries, etc.

What is Swift

Swift is a programming language that was first introduced by Apple in 2014. It worked perfectly with the Objective C environment of Apple. It was then open sourced in 2015 and since then it has evolved a lot over time. The biggest evolution being from Swift 3.0 to 4.0. It now currently stands at 5.x. Unlike Objective C, which had not changed a lot since 1980, Swift has a lot of modern features like *closures*, *pattern matching switch* and so on. Hence, within 4 years of its launch, Swift surpassed Objective C in popularity. Swift supports Cocoa and Cocoa touch frameworks, which means we can develop macOS and iOS apps using only Swift. It also has a *safety* feature, which means we can avoid the common programming errors like null pointer exceptions in Swift. XCode supports the previous Swift versions. In [Figure](#) you will see the Objective C code on your left-hand side and the Swift code on your right-hand side. You will see how the Swift code is cleaner and easy to read. So, let us practice writing some Swift code!!

```
code!! code!! code!! code!! code!! code!! code!! code!! code!!  
code!! code!! code!! code!! code!! code!! code!! code!! code!!  
code!! code!! code!! code!!
```

Figure 2.1: Objective C vs Swift syntax

Playgrounds

Apple provides a very interesting tool called It is included in the Xcode and you can practice Swift programming without having the need to create a project and run it. You can even see the results live. Let us start by creating a new playground:

Open Xcode and select the first option as shown in *Figure*



Figure 2.2: *Welcome to Xcode*

In the new Xcode 12.x, you might not see the option to create a Playground on the first window of Xcode. In this case, go to **File** | **New** | Refer to [Figure](#)

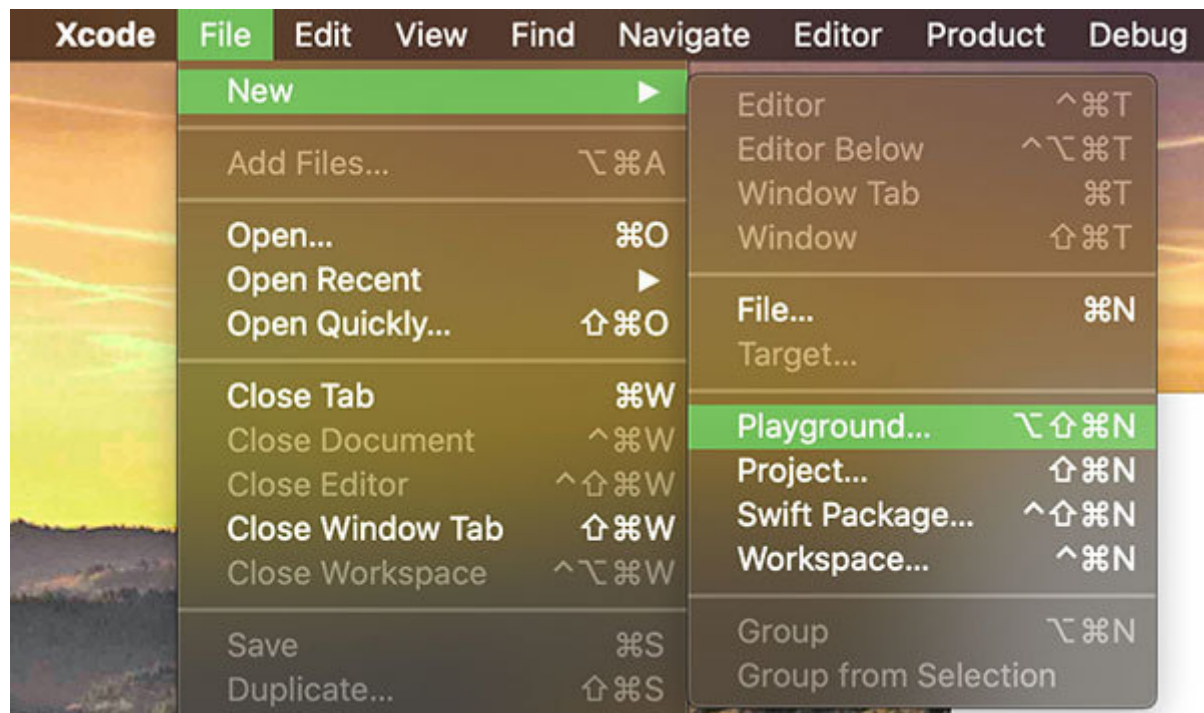


Figure 2.2.1: *Creating a playground*

Next, you will see a window as shown in [Figure](#). Select the first option and call it MyPlayground:

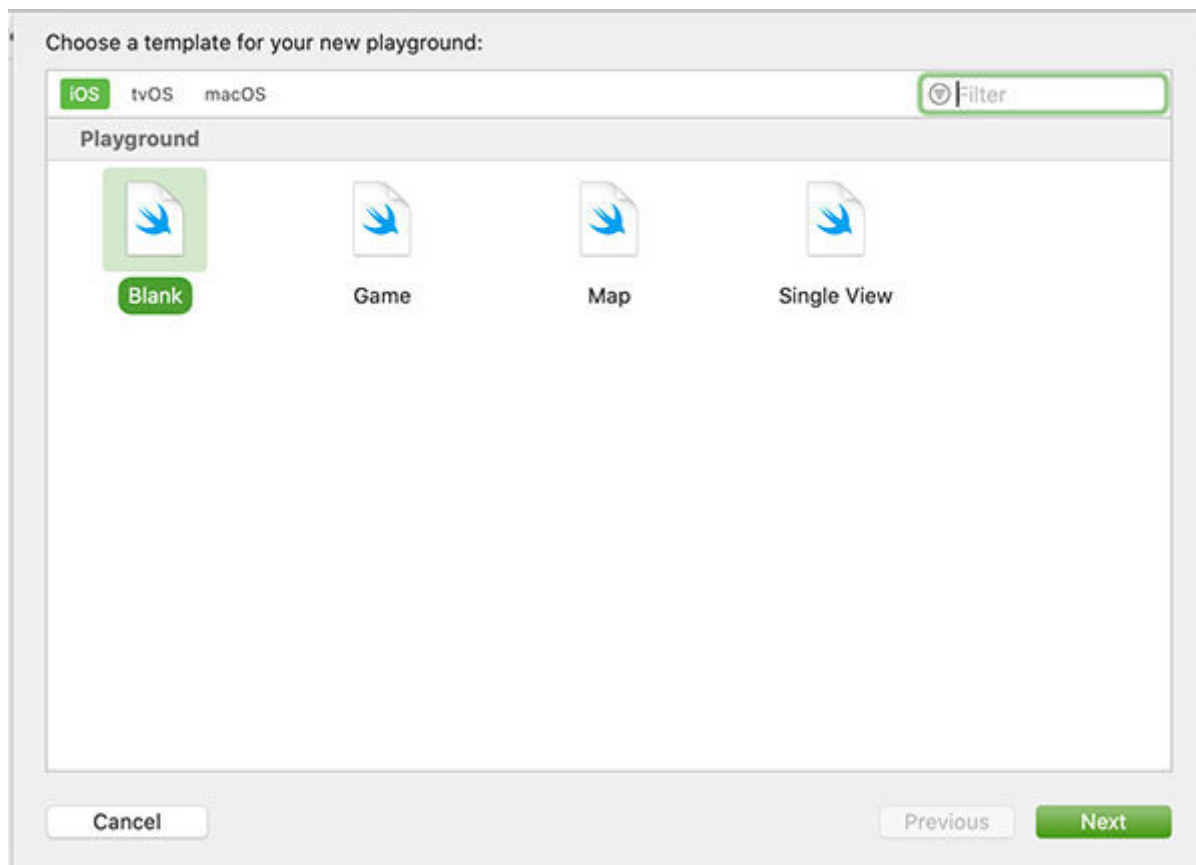


Figure 2.3: Playground template selection

Now, you should land on [Figure](#). There are three panes in this window. The left pane is the editor pane, the right pane is the interpreter pane, and the one below is the debug pane.

As shown in [Figure](#), click and hold the small play button on the *debug pane* and change it to *Once*. This is done, every time you type a code, you will see the results in real time in the *interpreter*.

Next, replace `import UIKit` with `import Foundation`. The import indicates which framework or library we would be using. We replaced the `UIKit` because we won't be needing any UI elements for now. That's it!! We are ready to get *Swift*.

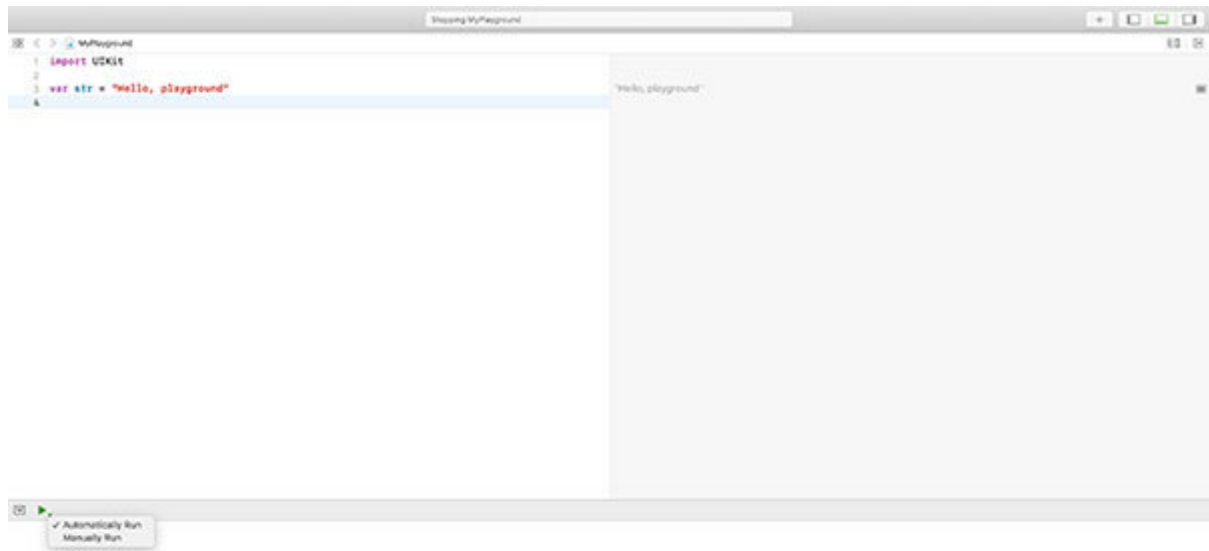


Figure 2.4: Layout of a playground

Apple has recently released an iPad app for Swift Playground. If you have an iPad, go ahead and have some programming fun!! There are even online playgrounds for Swift You can now practice Swift even if you are not on a mac machine.

Variables and constants

You will see the following line in your playground editor:

```
var str = "Hello, playground"
```

This is what we call a **variable** in Swift. As the name suggests, a variable is an element which can change. In Swift, a variable is declared with the word `var`. Now, let us try to decode the preceding line. We have declared, i.e., told the code that we want a variable, whose name is `str` and whose value is `"Hello, playground"`. Try to change it. On the next line, change it to some other value:

```
str = "I have changed the value"
```

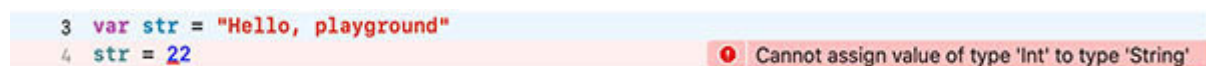
You will see in the preceding line that we have not used the keyword `var`. Why? Because we have declared it already. If we try to use the word **var** before you will get a re-declaration error. Go ahead, try it!

The opposite of variables is constants. Constants are elements whose values don't change. In Swift, constants are declared using the word `let`. Now, in the preceding code, go to the first line and replace the **var** with a `let`. The code should immediately throw an error. Why? Because we are trying to change the value of a constant.

The use of constants is also called Swift developers prefer immutability a lot. The code is much easier to read, if you use a lot of constants. Secondly, you are sure that the values of the constants do not change. Lastly, for Swift, it is easier to store data more efficiently, if it knows that your data is a constant. So, try to use constants whenever possible.

Type Inference

Change our **str** to a variable. Now, instead of changing its value to a new string in the next line, try changing it to a number as shown in [Figure](#)

A screenshot of a code editor showing two lines of Swift code. Line 3: `var str = "Hello, playground"`. Line 4: `str = 22`. A red error bar at the bottom right of the code area contains a red circle with an exclamation mark and the text "Cannot assign value of type 'Int' to type 'String'".

```
3 var str = "Hello, playground"
4 str = 22
```

Cannot assign value of type 'Int' to type 'String'

Figure 2.5: Type inference error

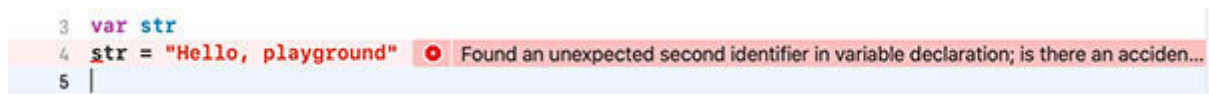
You will see an error: *Cannot assign value of type Int to So*, Swift knows, or it **Inferred** the type of the variable based on its declaration. This is called type Swift does not allow you to change the type of the variable; this is called **type** The real declaration should look something like this:

```
var str: String = "Hello, playground"
```

But because Swift can infer or guess types, we remove the String keyword. The type is used when we do not want to assign a value while declaring the variable, as follows:

```
var str :String
str = "Hello, playground"
```

If you do not declare a type and try to assign a value, the code will throw an error like the one in [Figure](#)

A screenshot of a code editor showing Swift code. Line 3 has 'var str' in blue. Line 4 has 'str = "Hello, playground"' in red. Line 5 has a vertical bar. A red error bar at the top right contains a red circle icon and the text 'Found an unexpected second identifier in variable declaration; is there an acciden...'.

```
3 var str
4 str = "Hello, playground"
5 |
```

Found an unexpected second identifier in variable declaration; is there an acciden...

Figure 2.6: *Declaration without a type*

This is because Swift is a statically, strongly typed language, i.e., it must know the type of all its variables and constants. If not mentioned, Swift tries to guess the value of the variables and constants using **type**. The type inference we just saw also applies to constants.

Commenting

The most important part of coding is its A code must be easily understandable to the one who codes and any other person who reads it. One of the ways to do this is by using comments. Following are the ways of commenting a code:

```
// a single line comment
```

```
/*  
a multi  
line  
comment  
*/
```

```
/*  
/*  
a nested comment  
*/  
*/
```

Figure 2.7 shows how comments can be used to make code more readable:

```

3 var str : String // example of variable declaration
4 str = "Hello, playground"
5 |
6
7 /*
8 A constant is declared using the keyword let and
9 Swift can infer its type based on the value assigned at the time of declaration
10 */
11 let otherString = "A constant String"
12 print(type(of: otherString)) //Swift guessed the type of the constant otherString

```

Figure 2.7: Code commenting

EXERCISE 2.1:

```

var firstString = "Hello"
var secondString = "World"
/*

```

without using direct assignment (ex. firstString = "World"), try to write a code where the end result would be

firstString: World

secondString: Hello

```
*/
```

```
print("firstString: \(firstString)")
```

```
print("secondString: \(secondString)")
```

Operators

Operators are **symbols** used to combine, verify, or change values. Though this sounds very fancy, we have been using operators for long. A simple example is the addition operator (+). Swift offers all the basic operators and along with that some other operators which we shall see. So, if we do $a + b$, the '+' sign is an **operator** and a , b are There are three types of operators:

Unary This operator operates only on a single operand. Depending on their position, they are called *prefix* ($!a$) or *postfix* ($a!$).

Binary This operator operates on two operands. Because they appear in between the two targets, their position is *infix*. $a + b$.

Ternary This has three operands or targets. Swift has only one ternary operator, the conditional one. We will see an example later in this chapter.

Following are the different operators used while coding in Swift:

Assignment The assignment operator (=) assigns the value of the right side to the left side. It can be used to initialize a value or update it. Following are some examples:

```
let a = 10
```



```
var b = 22
b = a
```

Arithmetic Addition (+), subtraction (-), multiplication (*), and division (/) are the basic arithmetic operators. These are performed on the same types, i.e., an integer cannot be divided by a string. The addition (+) operator, can be used to concatenate strings. Following are the examples:

```
let a = 10
let b = 22
a + b//addition
a - b//subtraction
a * b//multiplication
a / b//division
let name = "John" + "Adams"//string concatenate
```

Compound assignment This operator is similar to that in C. We can combine the assignment operator (=) with another arithmetic operator. This is useful to keep the code clean. Here is an example:

```
var a = 10
a += 2//this is equivalent to a = a + 2
a -= 2//this is equivalent to a = a - 2
a *= 2//a = a * 2
```

Comparison Comparison operators compare the value of the two elements. They are *Equal* (==), *Not Equal* (!=), *Greater than* (>),

Less than (<), Greater than or equal to (>=), Less than or equal to
The result is returned true if the statement holds or false if otherwise. Let us continue the above example and use the var a which we have created:

```
a == b//false
```

```
a != b//true
```

```
a < b//false
```

```
a > b//true
```

```
a >= b//false
```

```
a <= b//true
```

Logical Logical operators are of three type: They are used to combine the true and false values. The following table shows the symbols on the left and how they can be used in the actual code on the right. Following are their rules and symbols:

It is represented by it toggles the value from true to false and vice versa.

It is represented by Both values need to be true for the AND statement to be true.

It is represented by Either of the two values needs to be true for the statement to be true.

true. true. true. true. true. true. true. true. true. true. true. true. true. true. true. true. true. true.

Ternary conditional The ternary conditional operator, like the compound assignment operator, is a shorthand operator. It has the following format:

condition ?value 1 : value 2

If the condition evaluates to *true*, the first value is chosen, else the second value is chosen. It is a sophisticated way of writing an if else statement. The following table shows the if else and its equivalent ternary conditional operator:

operator: operator: operator: operator: operator: operator: operator: operator: operator:

Naming convention

You have seen some Swift code by now and you would have noticed a couple of things. The usual semicolon symbol ';' at the end of each statement is missing. And everything is in small letters. Look at the following example:

```
var firstString = "Hello"
```

The variable name **firstString** has two words: first and string. But when we combine these two, the second word starts with a capital letter. Thus, This is the called the **camel case** naming convention. This is adopted by You will see in the upcoming chapters that everything follows the same pattern. Except for file names, which always start with a capital letter.

EXERCISE 2.2:

```
let age = 22
/*
write a code such that if age is less than 18 the code should
print "Access denied"
and if the age is greater than 18 the code should print
"Congratulations! Access granted because you are xx years old",
where xx is the age
*/
```

App structure

iOS apps follow the *event-driven* paradigm, i.e., an event triggers the functioning of your app. For example, a button click saves the data or a pull down on the list refreshes the view. All such touch events are handled by the Cocoa Touch library. So, let us dive into a couple of components of this library.

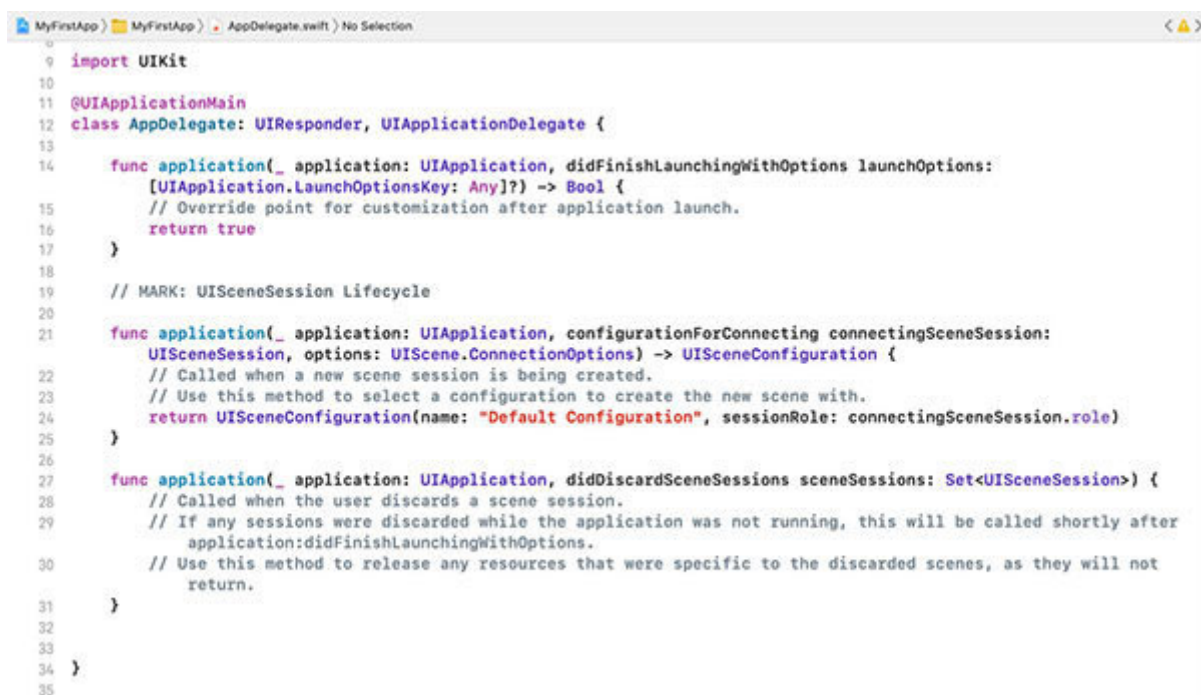
Event driven The iOS and macOS, at their core, have an infinite loop running. This loop is awaiting an event such as a touch or a mouse click. This event can also be a timer trigger or a drag or another event. Hence, when you debug your application, you can pause the run-loop. Although most of these events are handled by the in-built libraries, our job is to write the code to personalize these events. For example, we can write the code to save the data when a button is clicked.

App Open the project which we created in the first chapter. You will see that there is a **AppDelegate** file already present. What does this file do? The center of our application is the instance of the **UIApplication** class. However, we do not code directly, or we do not need to interact with this instance. We instead, work with its representative or delegate, **UIApplicationDelegate** which is our **AppDelegate**. (We will learn more about classes, instances, and delegates in the upcoming chapters). If you observe this class, ref to [Figure](#) you will see three main events:

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions:
func application(_ application: UIApplication,
configurationForConnecting
```

```
func application(_ application: UIApplication,
didDiscardSceneSessions
```

The first event serves as the entry point of the app, while the next two are used to manage the scenes, their creation, and their deletion:



```
1  import UIKit
2
3  @UIApplicationMain
4  class AppDelegate: UIResponder, UIApplicationDelegate {
5
6      func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
7          [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
8          // Override point for customization after application launch.
9          return true
10     }
11
12     // MARK: UISceneSession Lifecycle
13
14     func application(_ application: UIApplication, configurationForConnecting connectingSceneSession:
15         UISceneSession, options: UIScene.ConnectionOptions) -> UISceneConfiguration {
16         // Called when a new scene session is being created.
17         // Use this method to select a configuration to create the new scene with.
18         return UISceneConfiguration(name: "Default Configuration", sessionRole: connectingSceneSession.role)
19     }
20
21     func application(_ application: UIApplication, didDiscardSceneSessions sceneSessions: Set<UISceneSession>) {
22         // Called when the user discards a scene session.
23         // If any sessions were discarded while the application was not running, this will be called shortly after
24         // application:didFinishLaunchingWithOptions.
25         // Use this method to release any resources that were specific to the discarded scenes, as they will not
26         // return.
27     }
28 }
29
30
31
32
33
34
35
```

Figure 2.8: AppDelegate

Scene There is another file which you can see right after the creation of your project, the This class did not exist before iOS 13 (Xcode 11). With iOS 13, we can open multiple windows, or *instances*, of the same app. Hence, *Apple* introduced Scenes to

manage multiple UIs at the same time. To manage these multiple scenes, we need the **SceneDelegate** class.

Before, the **AppDelegate** was responsible for the foreground, background tasks as well as presenting the window. These two jobs have now been divided between the **AppDelegate** and In simple words; if Scene delegate manages the UI lifecycle, **AppDelegate** serves as the application entry point and manages the relation between different scenes.



```
1 import UIKit
2
3 class SceneDelegate: UIResponder, UIWindowSceneDelegate {
4
5     var window: UIWindow?
6
7     func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options connectionOptions: UIScene.ConnectionOptions) {
8         // Use this method to optionally configure and attach the UIWindow 'window' to the provided UIWindowScene 'scene'.
9         // If using a storyboard, the 'window' property will automatically be initialized and attached to the scene.
10        // This delegate does not imply the connecting scene or session are new (see 'application:configurationForConnectingSceneSession'
11        // instead).
12        guard let _ = (scene as? UIWindowScene) else { return }
13    }
14
15    func sceneDidDisconnect(_ scene: UIScene) {
16        // Called as the scene is being released by the system.
17        // This occurs shortly after the scene enters the background, or when its session is discarded.
18        // Release any resources associated with this scene that can be re-created the next time the scene connects.
19        // The scene may re-connect later, as its session was not necessarily discarded (see 'application:didDiscardSceneSessions' instead).
20    }
21
22    func sceneDidBecomeActive(_ scene: UIScene) {
23        // Called when the scene has moved from an inactive state to an active state.
24        // Use this method to restart any tasks that were paused (or not yet started) when the scene was inactive.
25    }
26
27    func sceneWillResignActive(_ scene: UIScene) {
28        // Called when the scene will move from an active state to an inactive state.
29        // This may occur due to temporary interruptions (ex. an incoming phone call).
30    }
31
32    func sceneWillEnterForeground(_ scene: UIScene) {
33        // Called as the scene transitions from the background to the foreground.
34        // Use this method to undo the changes made on entering the background.
35    }
36
37    func sceneDidEnterBackground(_ scene: UIScene) {
38        // Called as the scene transitions from the foreground to the background.
39        // Use this method to save data, release shared resources, and store enough scene-specific state information
40        // to restore the scene back to its current state.
41    }
42
43 }
```

Figure 2.9: *SceneDelegate*

So, if you check the delegate methods in the **SceneDelegate**, you will see the following:

```
func scene(_ scene: UIScene, willConnectTo
func sceneDidDisconnect(_ scene: UIScene)
func sceneDidBecomeActive(_ scene: UIScene)
func sceneWillResignActive(_ scene: UIScene)
func sceneWillEnterForeground(_ scene: UIScene)
func sceneDidEnterBackground(_ scene: UIScene)
```

Within these, you will also see their respective descriptions. In brief, all these methods serve to manage the different states of the UI, along with the background and foreground states of the UI.

Views & We have already briefly seen views and windows in the first chapter. Traditionally, iOS apps were single window apps. With iOS 13, Apple introduced multiple windows for iPads. Usually, you don't have to worry about managing windows. You will mostly be customizing views. The class which represents a view is `UIView`. A view has a frame, with *origins* (x, y) and *size* (*width*, *height*). The *origin* (0,0) of an iOS window is the top left-hand corner. You would be using subclasses of the `UIView` class. In the next chapter, we will also learn about constraints, which decide the position of the view.

Like the word says, they control the view. These handle the events of the views. Every **ViewController** has a view, which is its main view. You can then add multiple subviews to this main view. View controllers can be of different types, depending on the type of content they show. For example, a **UITableViewController** is a view controller which shows a list. So, it will have all the events and delegates to populate and manage the list. But the base class is always the `UIViewController`. You will learn more about this.

Storyboards and These are files to manage the interface in a simple way. If you check the source code of these files, it is in fact a very complicated XML. The storyboards are used to tell a story. Your app flow is nothing but a story represented by screens and connections. You can have multiple storyboards in the app. Nib are interface files for an individual view or window. Their extension is In simple words, **nib** is a single interface file while a storyboard is a collection of interface files.

Targets and Refer [Figure](#) Under targets you will see the **MyFirsApp** file. Targets are the products your project will produce, i.e., the app bundles your project will build. You can find these products under the **Products** folder in the navigation pane on the left. *Every project has at least 1* You can then add other targets for different configurations like watch OS or macOS, or the same app with a different name which can have different functionalities at run-time. For example, the same code base can have a customer app and an admin app. You need to select the right target and build the app to have the right bundle. You can see that every target has its own settings. You can even have different assets, launch images and libraries depending on the target.

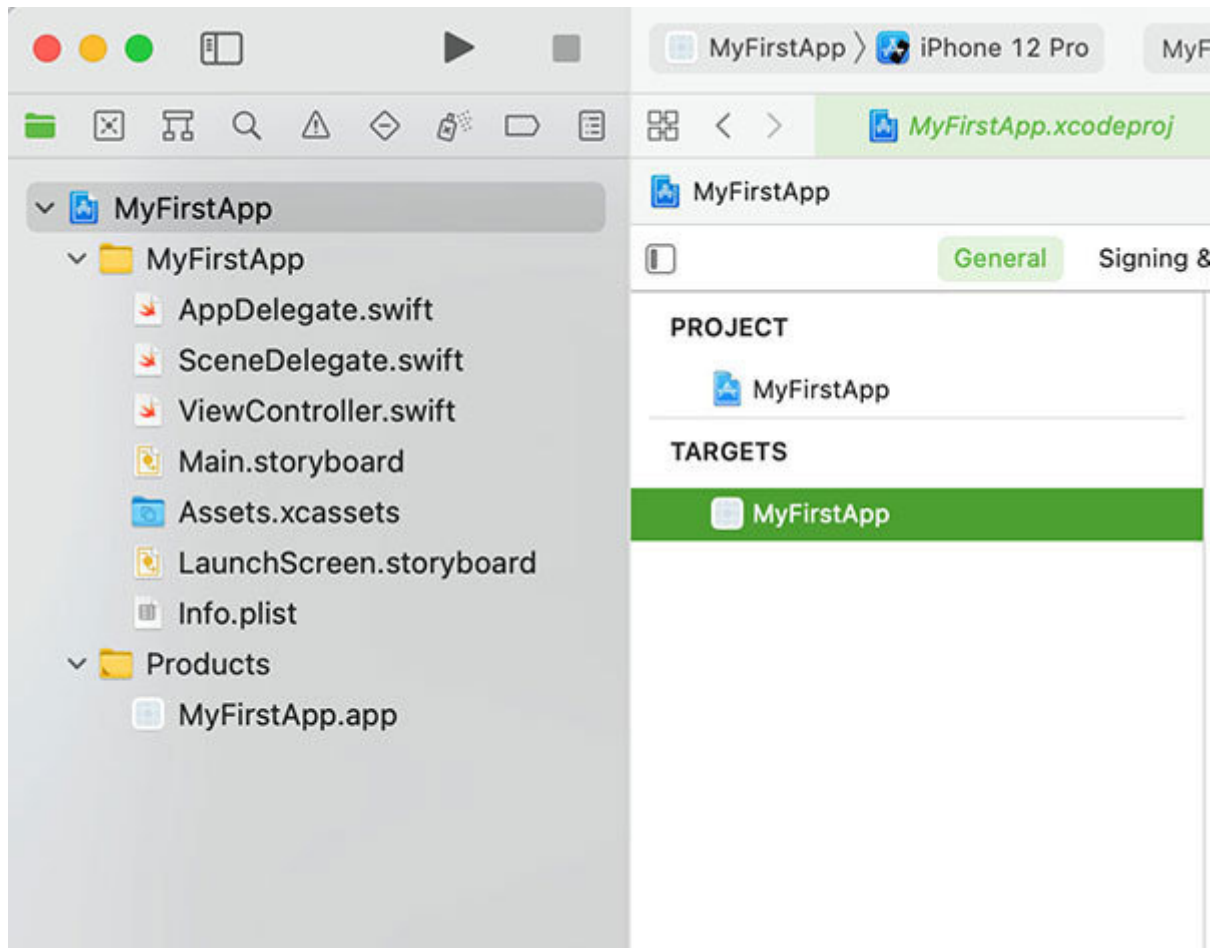


Figure 2.10: Targets and products

Conclusion

We saw that *Playgrounds* is a powerful, live tool to learn Swift programming. We learned about variables and constants in Swift. The variables are declared with the keyword **var** and **constants** are declared with the keyword **let**. We also found out that Swift is a strongly typed language and it uses type inference to find out the type, if it is not declared. We then learned how commenting can be helpful for a clean code and the different ways in which it can be done. At the end, we saw some operators in Swift. We ended the chapter by learning more about the app structure and some important words you will come across often as an iOS developer like windows, view controllers, views, targets, and products. In the next chapter, we will return to some iOS app development. We will learn about auto layout and interface building.

Key learnings

We started the chapter by learning about the brief history of Swift.

We then created our own first Playground. This is a very handy important tool to practice Swift.

First, we saw the basic elements of Swift, the variables and

Then, we learned about type inference and how Swift is a strongly typed language.

We also saw how to comment in Swift and use of multi-line commenting.

Later, we moved on to operators and saw the basic operators in Swift +, -, *, / and the AND, OR, NOT.

We also learned about the standard naming conventions followed by coding in Swift.

We then moved back to iOS App development and learned about the structure of the App.

We saw a brief theory of **SceneDelegate** and Views and Windows.

We even learned about storyboards and targets.

Test your knowledge

When was Swift introduced?

2014

Is Swift open source? Since when?

Yes, since 2015

How to declare a constant in Swift?

By using the keyword *let*

Can I add integer to a string using the "+" operator?

No, because Swift is a strongly typed language.

What are the different types of operators?

Unary

Binary

Ternary

What is the position of the binary operator called?

Infix

What is the Apple naming convention called?

Camel case

Which type of programming do iOS apps follow?

Event-driven programming

What is the AppDelegate?

It is the representative or the delegate of the UIApplication class.

How many targets can a project have?

A project must have at least one target, or it can have multiple targets.

CHAPTER 3

Auto Layout

By now, you have seen some Swift code. You can also tell the difference between a **var** and **let** . In this chapter, we will build a new iOS app. The user interface is the most important part of a mobile app. Apple provides us with the interface builder to create beautiful interfaces. At the same time, Apple has a range of devices for which we need to build different screens. This time we will try to understand how sizing works in iOS and how it is useful to build universal apps. We will start with understanding auto layout and then size classes. Then, we will learn to add constraints and also learn about the view controller lifecycle. All this while building an iOS app.

Structure

In this chapter, we will cover the following topics:

Need for auto layout

Size classes explained

Constraints

Adding multiple constraints

IBOutlets

View controller lifecycle

Types of view controllers

Objectives

The objective of this chapter is to help you get a better understanding of the interface builder. You will also get to know the different types of views, view controllers, and their lifecycle. By the end of this chapter you should be comfortable constructing a basic UI for an iOS app and play around with the interface builder.

Need for auto layout

Let us understand the need for auto layout while making an application:

Open the *MyFirstApp* project we created in [*Chapter 1, Building Your First*](#)

You will see the result as shown in [*Figure*](#)

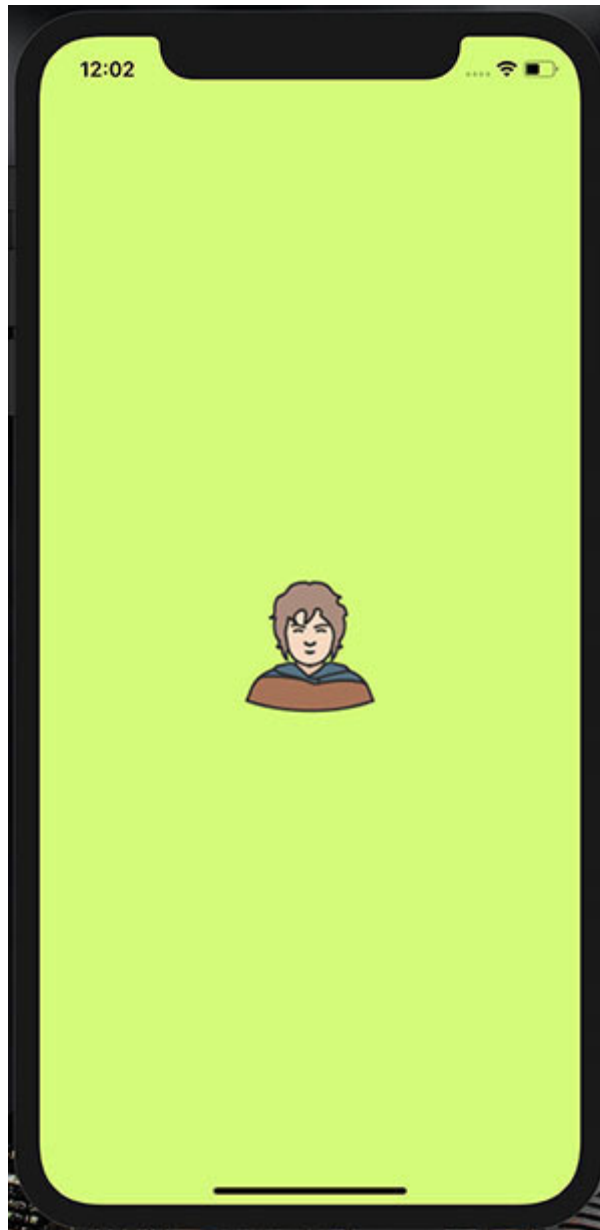


Figure 3.1: *Expected result of first app*

Now, change the device in the simulator in the status bar. I changed it from iPhone 11 to iPhone 11 pro. Click on the simulator and you will see a drop down as shown in [Figure](#)



Figure 3.2: Simulator options

Run the app again. You will see something as shown in [Figure](#)

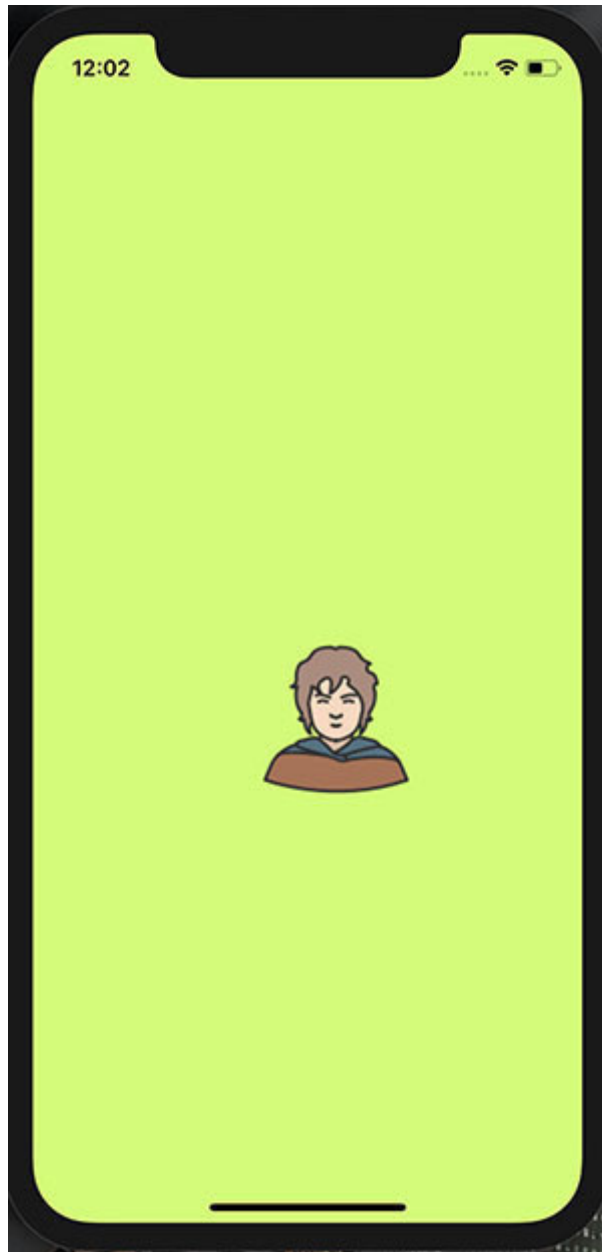


Figure 3.3: Image not centered

Now, change the device to iPad. Run it again.

You will see something as shown in [Figure](#)

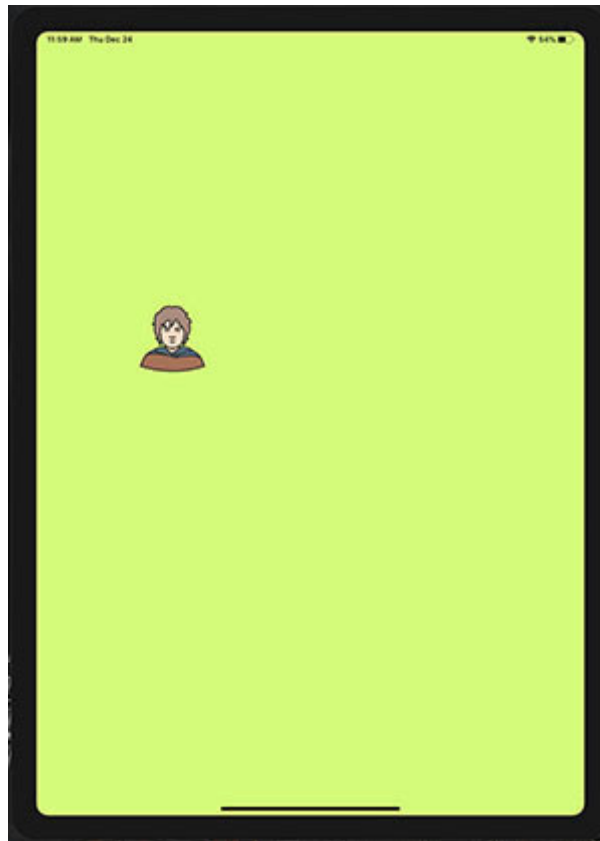


Figure 3.4: *Image changes position on iPad*

Do you see the difference? The image is not centered anymore. But in the interface builder, it is. So, what is happening here?

Now, click on the size inspector, in the right inspector pane. You will see the following screen as shown in [Figure](#)

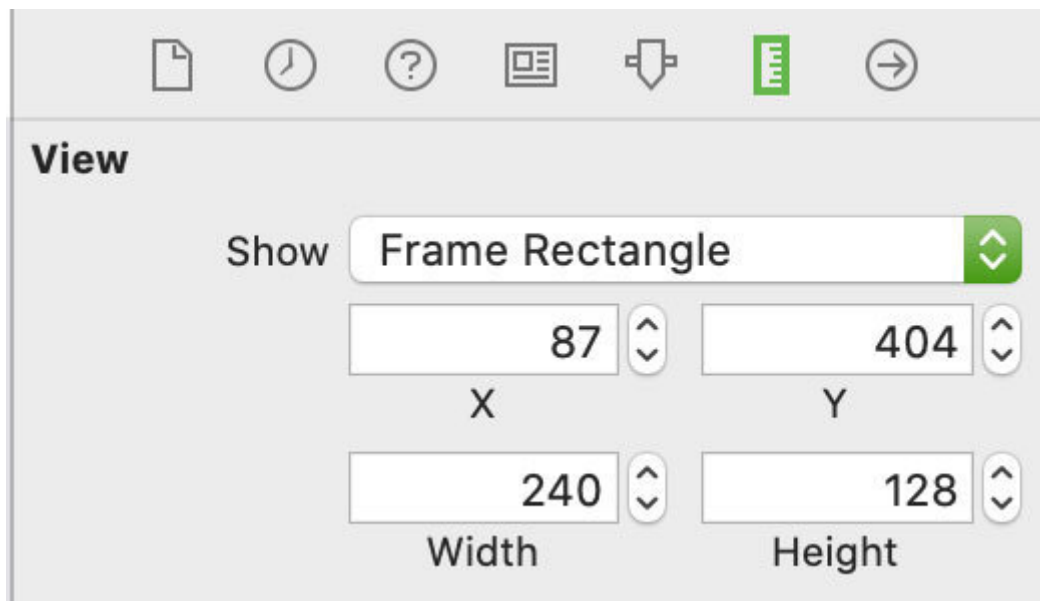


Figure 3.5: Frame adjustment in Size Inspector

What do these co-ordinates indicate? These tell us that the image is at a position where x is 87 and y is 404. But where is the 0,0 point of this co-ordinate system? It is on the top left-hand corner of the screen. *This point never* It is the same in every device. So, if (87,240) is the center of the screen for iPhone 11, it is not the same for an iPad. Hence, we will see the image moving around for different devices. *This is exactly the reason why we need auto-layout; to build the adaptive UI which will work well on many*

But this does not mean, you will have the same UI on every device. Looking at the different screen sizes, it would not be possible. Even then we would need the auto layout. We can activate and deactivate the distance and sizes of the views to change the layout completely depending on screens. Let us understand the different elements which work together to make the auto-layout possible.

Sources of changes

We saw that changing the devices disturbed our UI. But this is not the only source which can trigger a UI disturbance. The changes can be divided into two types:

External These changes are triggered when the superview changes. This change demands rearrangement of the subview(s).

Following are the changes which can trigger a rearrangement of subviews:

The app enters the split mode in iPad.

The device rotates.

There are calls or other popups superimposed on your app.

When you change the device type, from iPhone to iPad or iPhone SE to iPhone 12.

Internal These are triggered when the subviews or data in your app changes.

Following are the internal changes which can trigger a rearrangement of subviews:

When the content of the app changes, like hiding some view or animating a neighbor.

The app supports the dynamic type. This is when some people choose a larger font for their phone. This can increase the size of your labels which can trigger a view rearrangement.

The app supports multiple languages. This happens when a sentence in English might not have the same length in another language. This can also trigger re-arrangement of views.

Understanding the layout

Before we move ahead, you must understand how the layout works. In the preceding example, you saw that every view on your screen is in the co-ordinate system. Let us go into more details. There are two important factors to determine the placement of a view. When we use the word view here, it means you will see all the different types of objects on the screen. They are all different types of views, which are as follows:

Position is usually the x and y co-ordinate of the view to its superview. So, if you add a label on a view, which is on the window; the label's x and y positions are relative to the view and not the window.

This is the height and width of the view. This is important to determine the amount of space the view will take on your screen.

The layout can be further divided into the following two types:

Frame We can call it the absolute position. The absolute position is the position as seen in the co-ordinate system. Like (0,10) indicates its exact position. This was the traditional way of indicating the positions of views. But it turned out to be very static as the different size devices were introduced. This was often done programmatically.

Auto This we can simplify to call the relative position its position, relative to other views. Like 10px away from the neighbor and 20px away from the edge. This can be really helpful when you design for multiple screens. The relative position is used when we assign the constraints of the auto layout. Auto Layout is easier with the interface builder. However, it can also be done programmatically.

The relative position makes layouts simple. You can just tell the storyboard that your view will be 10px away from its neighbor and 20 px from the bottom. You can even specify that if the neighbor is absent, you want the view to be 10 px away from the right edge. [Figure 3.6](#) explains the different terms used to describe the positions/constraints of a view:

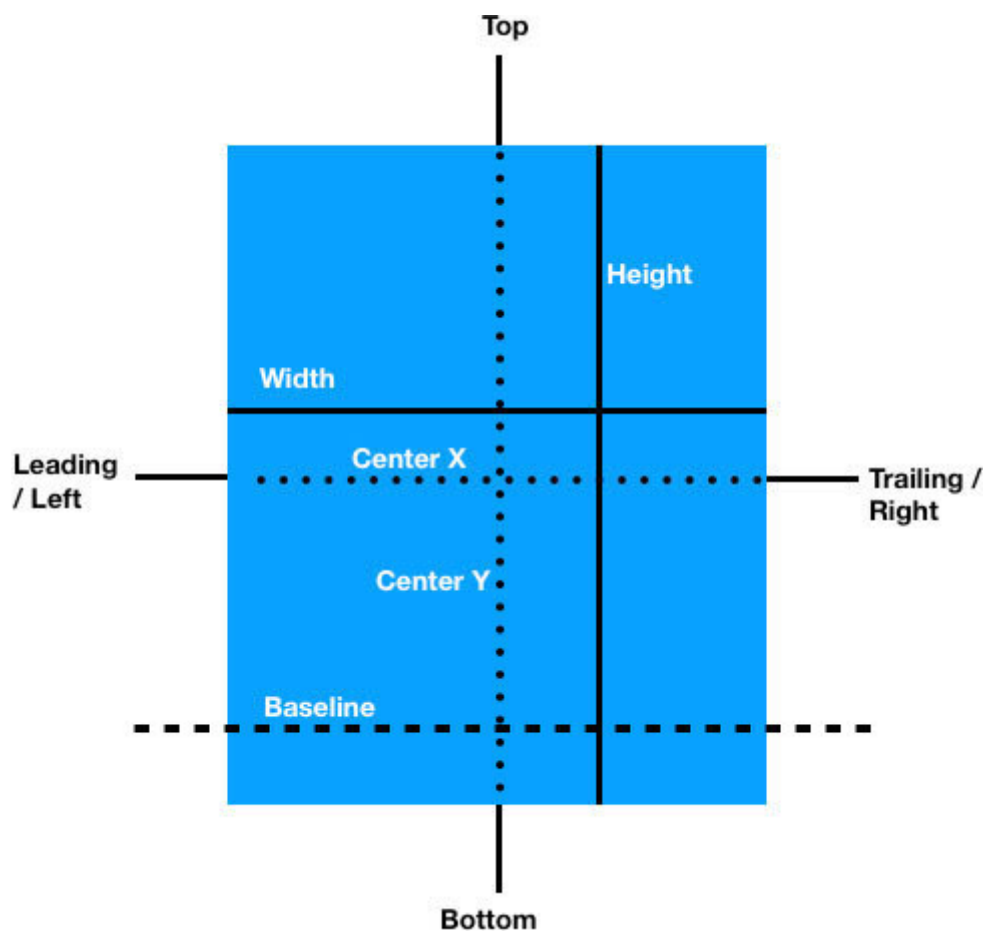


Figure 3.6: *Terms used in Auto Layout*

Size classes explained

The Apple environment currently has many devices. Just this year, they released iPhone 12, 12 mini, 12 pro, 12 pro max, and by the time, you are reading this book, there are probably new devices already in the market. We still have many old iPhone users. Then, there are the iPads. Designing apps for so many sizes can be very impractical.

Size classes are groups of screen sizes that are applied while designing for different Currently, there are two types of size classes: *Compact* and *For each of these two types, there are height and the width* variations:

Compact Width, **wC**

Compact Height, **hC**

Regular Width, **wR**

Regular Height, **hR**

Figure 3.7 displays your storyboard. There are many elements. Let us try to understand them one by one.

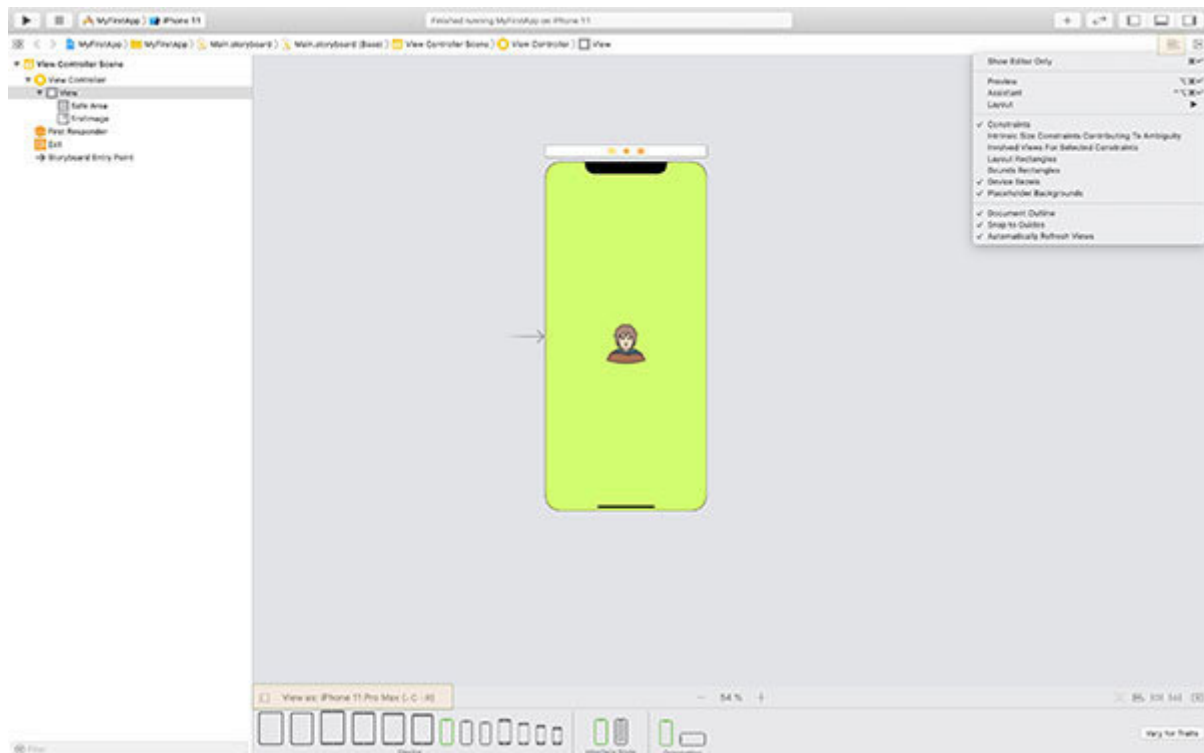


Figure 3.7: Storyboard landing screen

In [Figure](#) in the highlighted part at the bottom left, you will see something like **View as: iPhone 11 (wC** If you click on this tab, the size class pane opens up. It looks like the one in [Figure](#)



Figure 3.7.1: Size Class pane

Here, you can change and choose the device you want. As you change, you will see what the different variations mean; so **wC hR** means the width is compact and the height is regular, that is, the width is smaller than the height, which means it is an iPhone. If

it is **wR** it means height and width are both equal, that is, it is an iPad.

Adjust Editor This is the window which opens up on the top right-hand side corner of the editor when you click on the button, the one shown in [Figure](#)

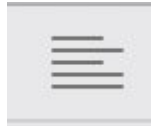


Figure 3.7.2: *Adjust Editor Options button*

When you click on this button, the Adjust Editor Options tab pops up. Refer to [Figure](#)

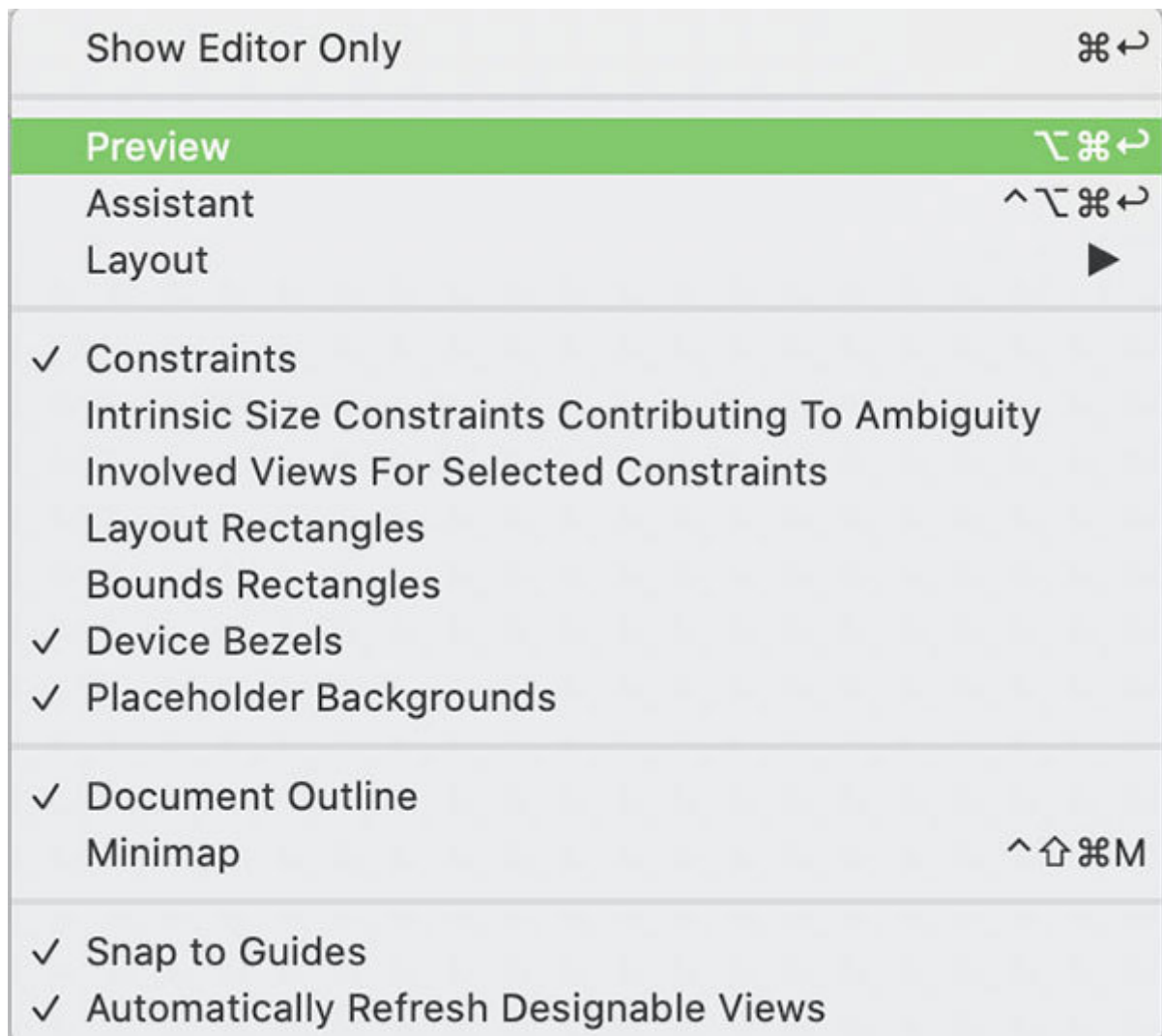


Figure 3.7.3: *Adjust Editor Options menu*

We will concentrate on two options: **Preview** and The other options are for you to discover. Click on the As soon as you do this, you will see another pane opening up on the side. This is the how your view will look like. This is practical if you want to avoid running the simulator all the time, and instead check your results in real time. Refer to [Figure](#)

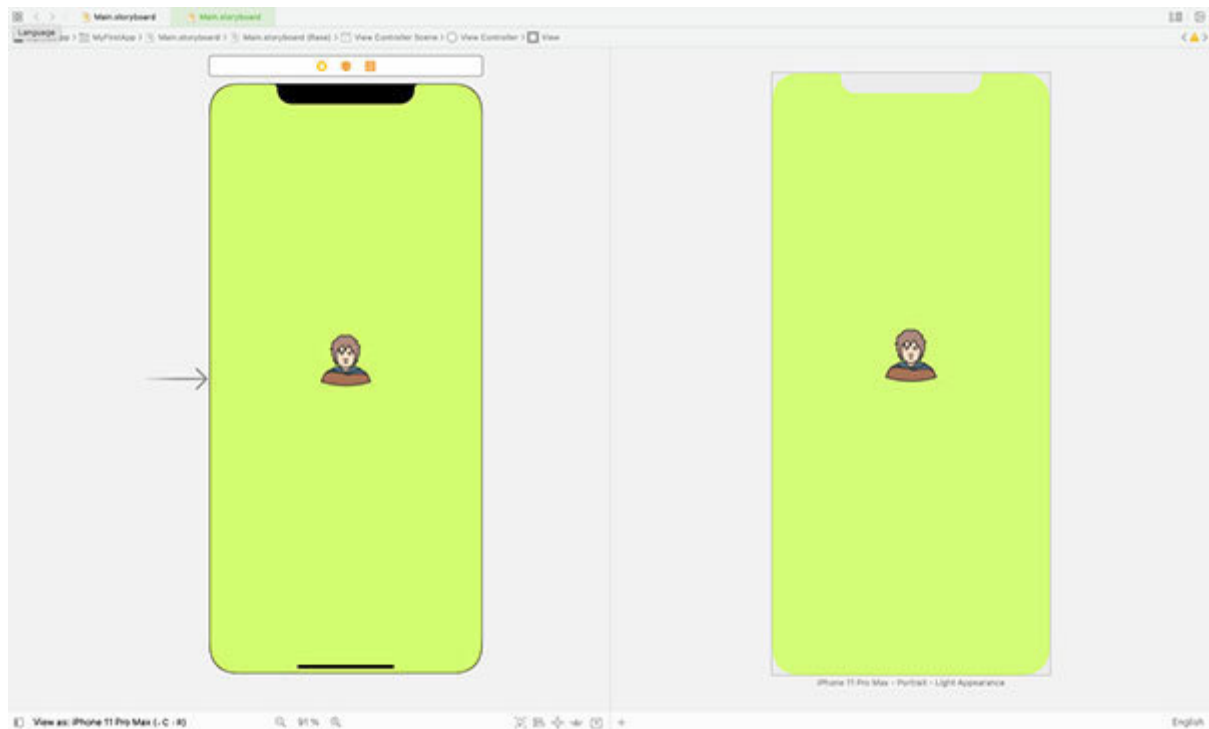


Figure 3.8: *Preview assistant*

You can even click on the '+' button on the left right corner of the preview pane. A pop-up with all the possible devices will open up. You can now add multiple devices at the same time. Refer to [Figure](#)



Figure 3.8.1: *Preview assistant device options*

Constraints

How do we tell the interface builder to change the position for every device? This is where the *constraints* come into picture. We constraint the UI object to stay within certain limits. Constraints help us assign relative positions to the UI objects. How do we assign these constraints? In the bottom right-hand corner, you will see a line of icons, as shown in [Figure](#)

Let us find out what they are. Go ahead and click on each one as you read through:



Figure 3.9: *Constraint Manipulation options*

Update After you add constraints, you can click on this to update your UI. This lays out all your UI objects according to the constraints you have added.

When you select multiple objects and click on this, you can add multiple constraints at a time. For example, I can select three buttons and then add constraints to align their horizontal centers. Refer to [Figure](#) You can see all the options in the figure. But

there are many options which are disabled because you have only one image view that is selected. When you want to align a single view, it is aligned with respect to its superview. Hence, the other options are disabled and the only alignment which we can assign is that of centering the view relative to its superview.

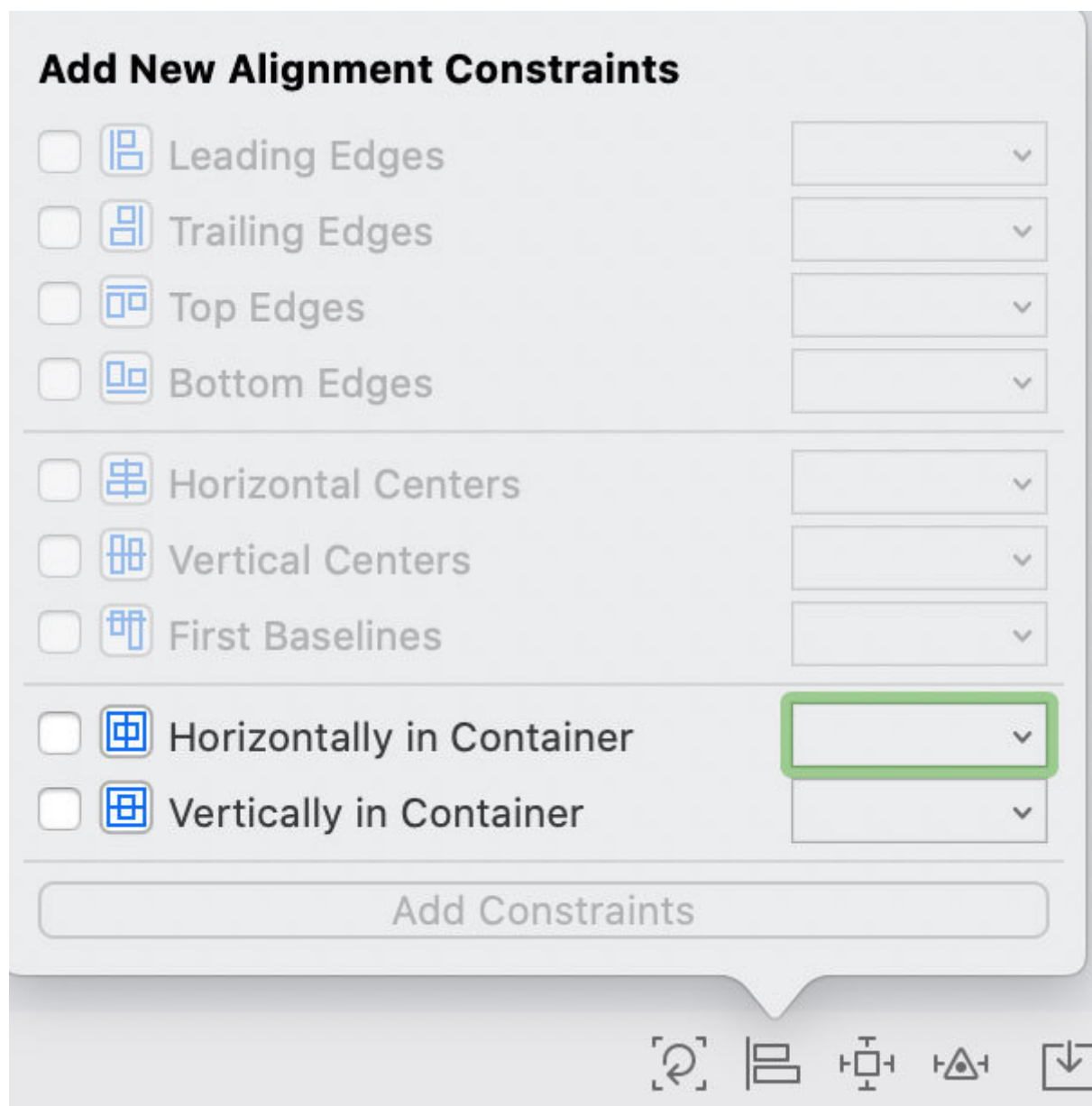


Figure 3.9.1: Align

Add new constraints/Pin This is used to add constraints relative to the object's neighbors. Refer to [Figure](#) You will see many

options. Notice that we can set the view's distance in all the 4 directions. We can also set its height and width. As discussed earlier, these are the two important things needed to determine the placement of the view on the screen.

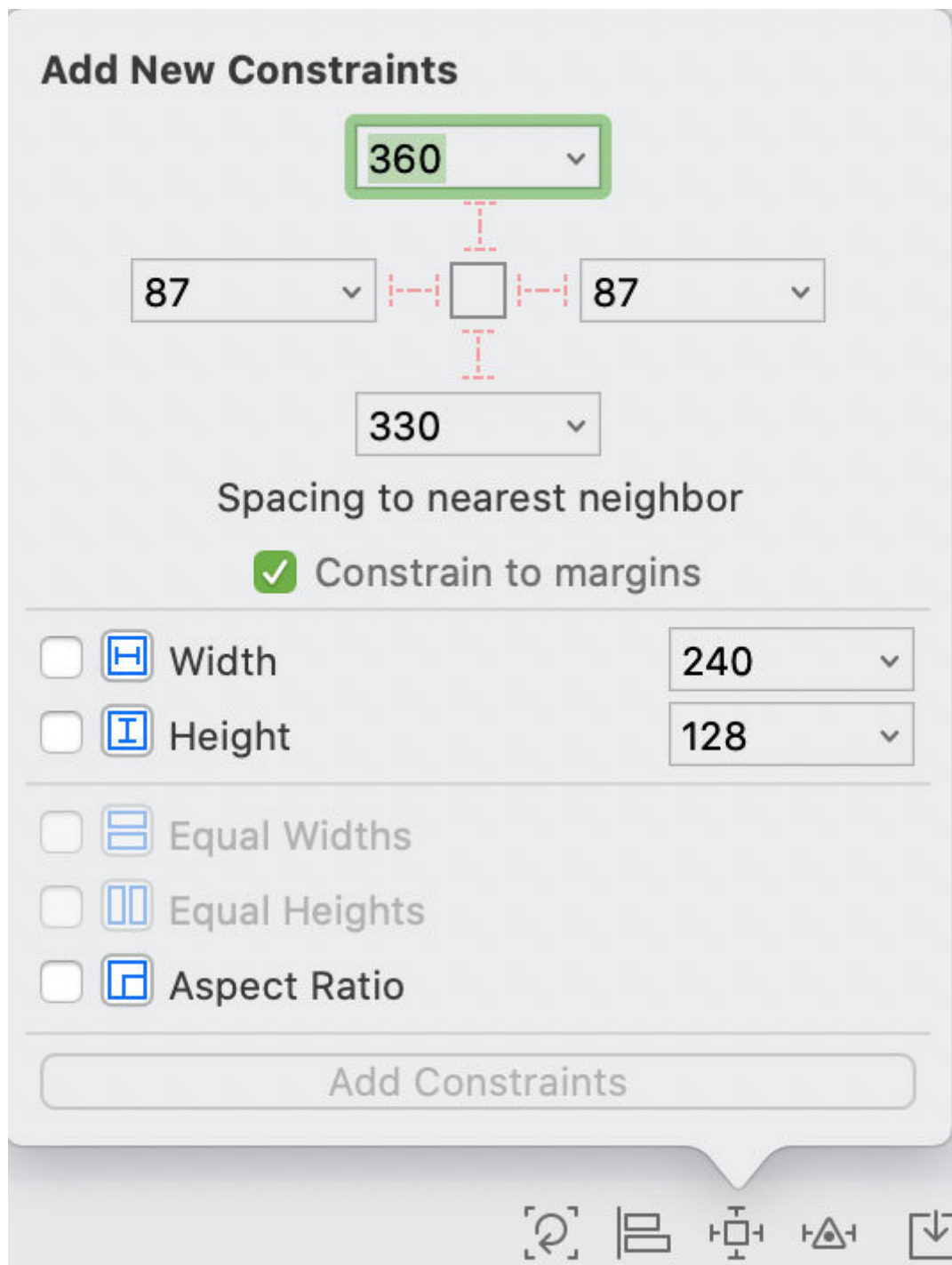


Figure 3.9.2: Pin Tool menu

Resolve auto layout This is used to add missing constraints. Though this may seem the easy way out, most of the times the constraints added are not the right ones. Real-world interfaces are much more complex and have a lot of conditions. Xcode cannot pre-determine all these conditions and assign the right constraints. It's always wise to assign your own constraints to have a better understanding. You can even update or remove constraints of a selected view or all the views. This is helpful when you get the constraints wrong and would rather start from zero than change each constraint one by one.

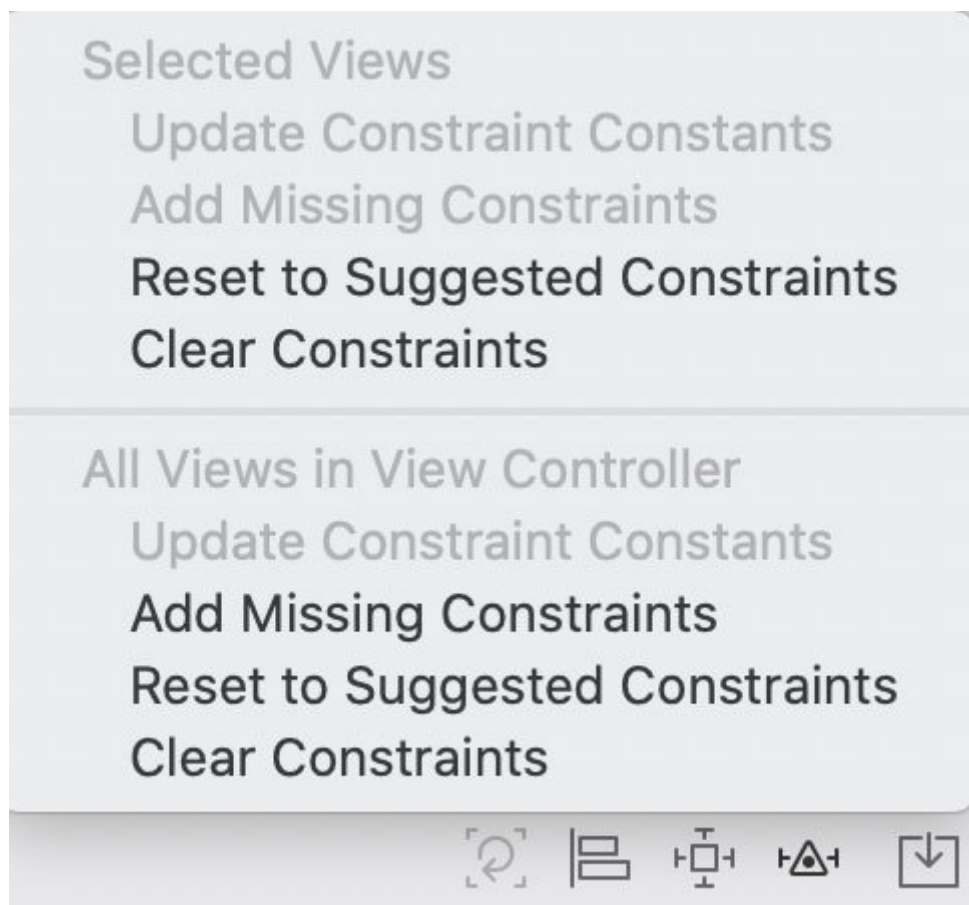


Figure 3.9.3: *Resolve auto layout issues menu*

Embed This is useful when you want to embed or group multiple views together for better UI management. Refer to [Figure](#)

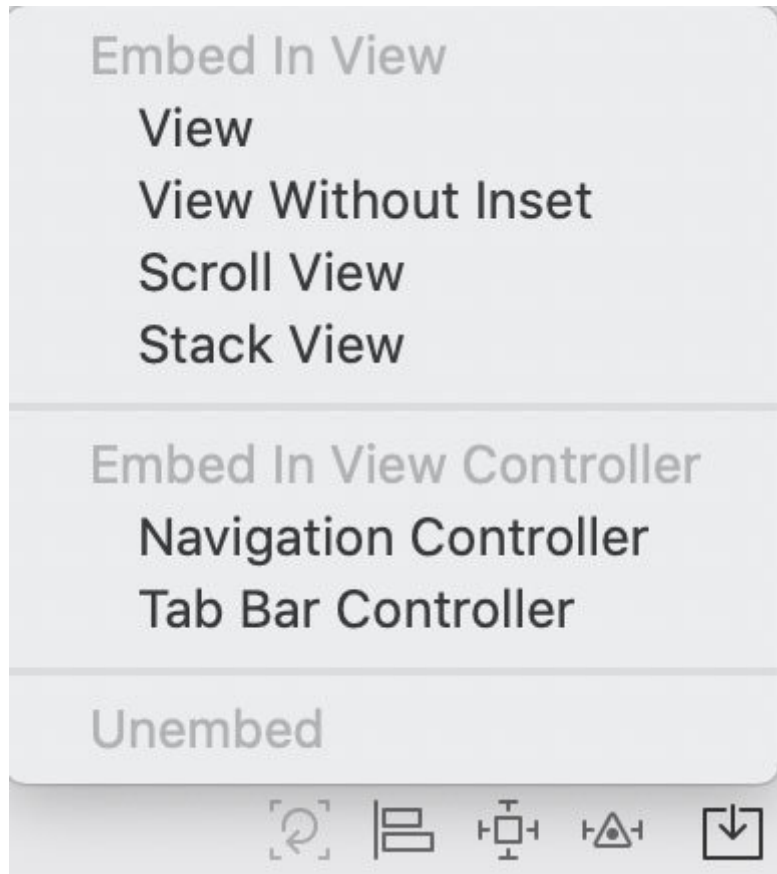


Figure 3.9.4: *Embed In menu*

Now, let us try to add some constraints to our image:

We want the image to be at the top center. This tells us two things; that the image would be closer to the top and horizontally it would always be in the center.

Select the image. Then, click on **Add new** constraints, and the pop up will open.

Activate the top constraint, as shown in [Figure](#) and change the number to 10. Then, click **Add 1** constraint.

Next, click on **Align** and choose **Horizontally** in the container.

If your image does not update its frame, click on **Update**

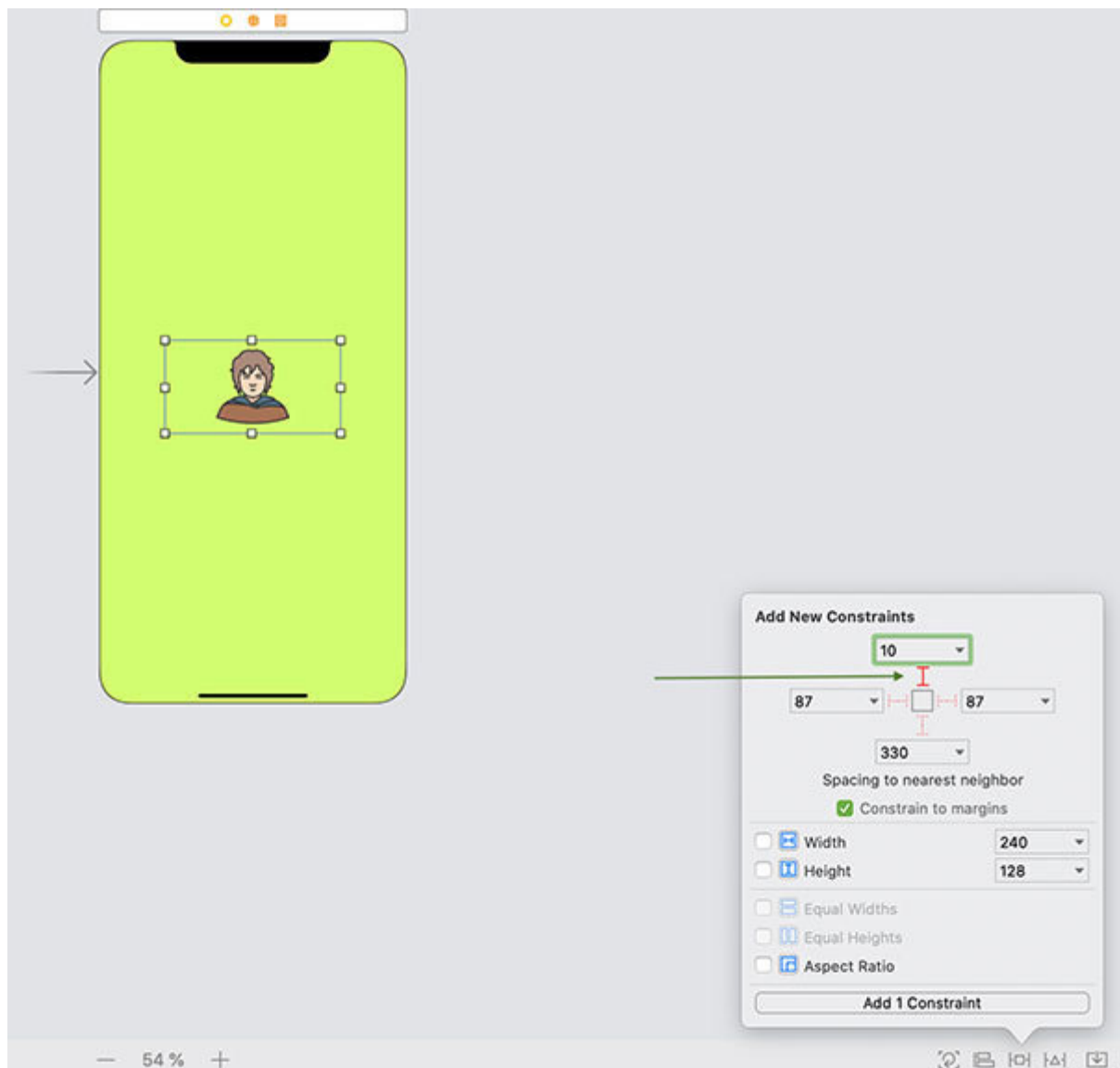


Figure 3.10: *Activating top constraint*

Tip: When adding the top constraint, you might see a small drop down where you need to enter the value. There you will see *Safe*. Safe area is the portion of the view which is not covered by bars or notches, i.e., the visible area of the phone.

If the *Constrains to margins* checkbox is checked, it means that the constraints to the superview uses its margins. If not, the constraints to the superview use it's edges.

Adding multiple constraints

So far, we have only been playing around with one image view. Let us now try to add more views.

We will try to build a simple form. Follow the next steps:

Open the **Library** from the status bar. Type in the text, and you will see two options: **Text Field** and **Text**. Text field is for single line entries and text view for multiple.

We need single line entries for a form; hence, we will select a text field. Drag and drop it on the view, or double click on it.

Explore the different options in the Attribute inspector. Text, background color, and so on.

Tip: In the *Attribute inspector* of the text field, you will see an option called **Placeholder**. Placeholder is the grayed-out text which appears in the text fields, when they are empty. Put in *First name* as the placeholder for the first text field.

Place the text field at the desired position below the image view.

Now, here is the easy part: We want the text field to be below the image. So, select the text field. Now, click on Ctrl and drag it

towards the image as shown in [Figure](#)

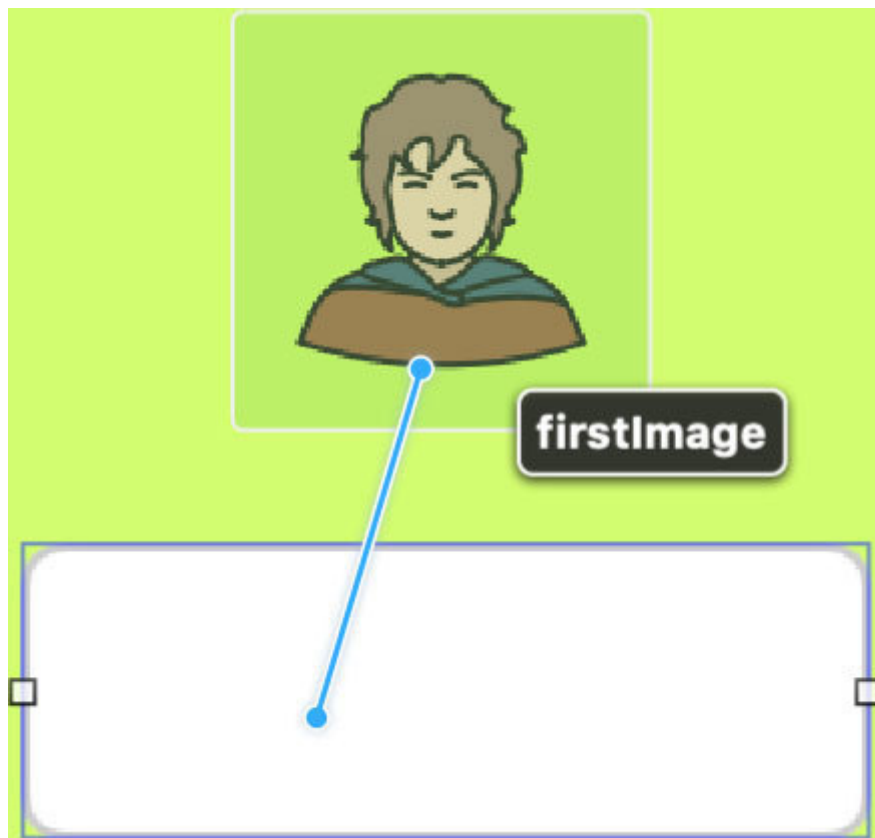


Figure 3.11: *Ctrl+drag method of adding constraint*

You will see a pop up with various options. Like the one in [Figure](#)
We want to space the text field vertically below the image. So,
select the option **Vertical**

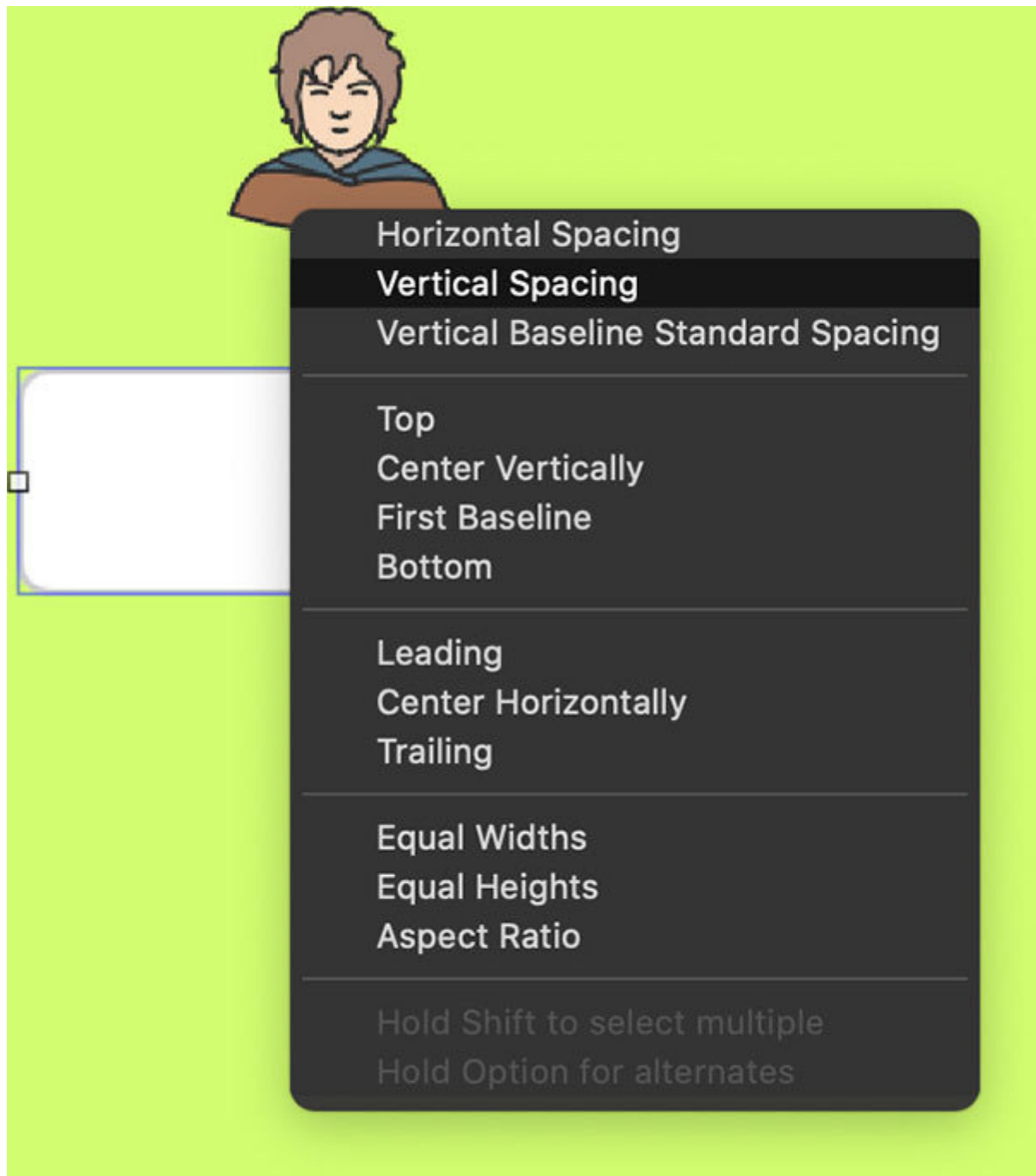


Figure 3.12: *Constraint type menu*

Now, go to the size inspector. You will see a constraint added to the text field as shown in [Figure](#)

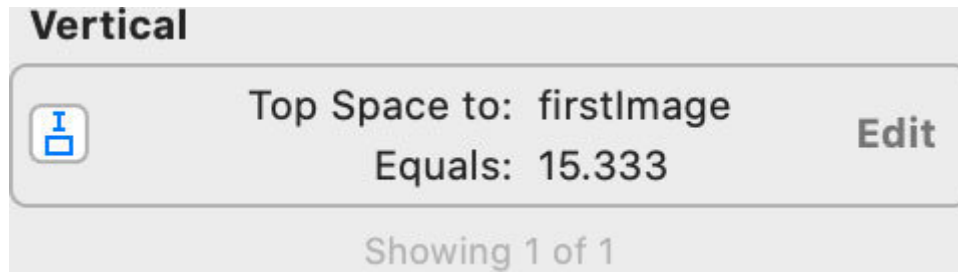


Figure 3.13: *Added constraint*

But to place an object, we need its horizontal AND vertical position. We have just assigned the vertical. We want the text field and image view to have the same center.

Follow the same steps but this time select **Center horizontally** as shown in [Figure](#). What we are doing here is aligning the text field with respect to the image view. This is because our image view already has an anchor or position. Now, if we move our image view 20px from the top, our text field will move automatically down. Or if we move our image view to the right, so will our text field.

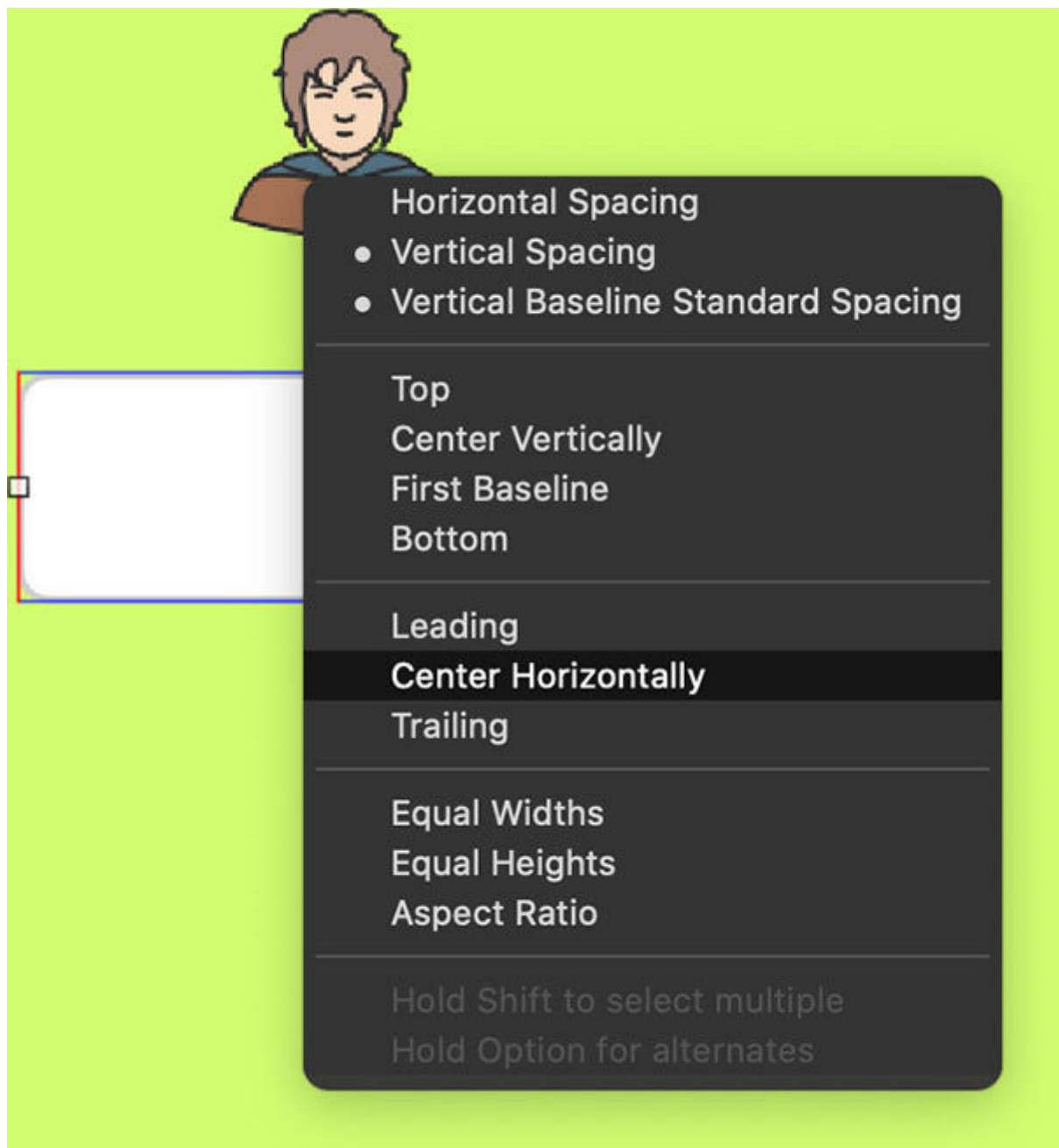


Figure 3.14: Center the image horizontally

You can then change the width and height of the text field using the **Pin**

Tip: The method we saw, was using *Ctrl + Dragging* You can try to add all the constraints using the pin tool as explained earlier.

The preceding way was just to give you an idea of both the types.

Editing a constraint

Double click on the vertical spacing constraint which we added (Refer to [Figure](#) This will allow you to edit the constraint. You can see all the editing options of the constraint. Let us try to understand all of them as shown in [Figure](#)

The screenshot shows the 'Vertical Space Constraint' dialog box. At the top, there is a toolbar with icons for file, undo, help, view, settings, a green triangle, and a refresh button. The dialog title is 'Vertical Space Constraint'. It contains the following fields and controls:

- First Item:** A text field containing 'First name.Top' with a right arrow icon and a green dropdown arrow.
- Relation:** A text field containing 'Equal' with a green double-headed arrow icon.
- Second Item:** A text field containing 'firstImage.Bottom' with a right arrow icon and a green dropdown arrow.
- Constant:** A text field containing '15' with a dropdown arrow and a green up/down arrow icon.
- Priority:** A text field containing '1000' with a dropdown arrow and a green up/down arrow icon.
- Multiplier:** A text field containing '1' with a dropdown arrow and a green up/down arrow icon.
- Identifier:** A text field containing 'Identifier'.
- Placeholder:** A checkbox labeled 'Remove at build time'.
- Installed:** A green checkmark icon followed by the text 'Installed'.

Figure 3.15: Editing a constraint

On the top, we can see the type of constraint: **Vertical Space** Let us understand each item one by one:

First As this constraint is between two views, the first item represents the first view. If you click on the drop down, you will see many options, as shown in [Figure](#). You can then change the first item altogether:

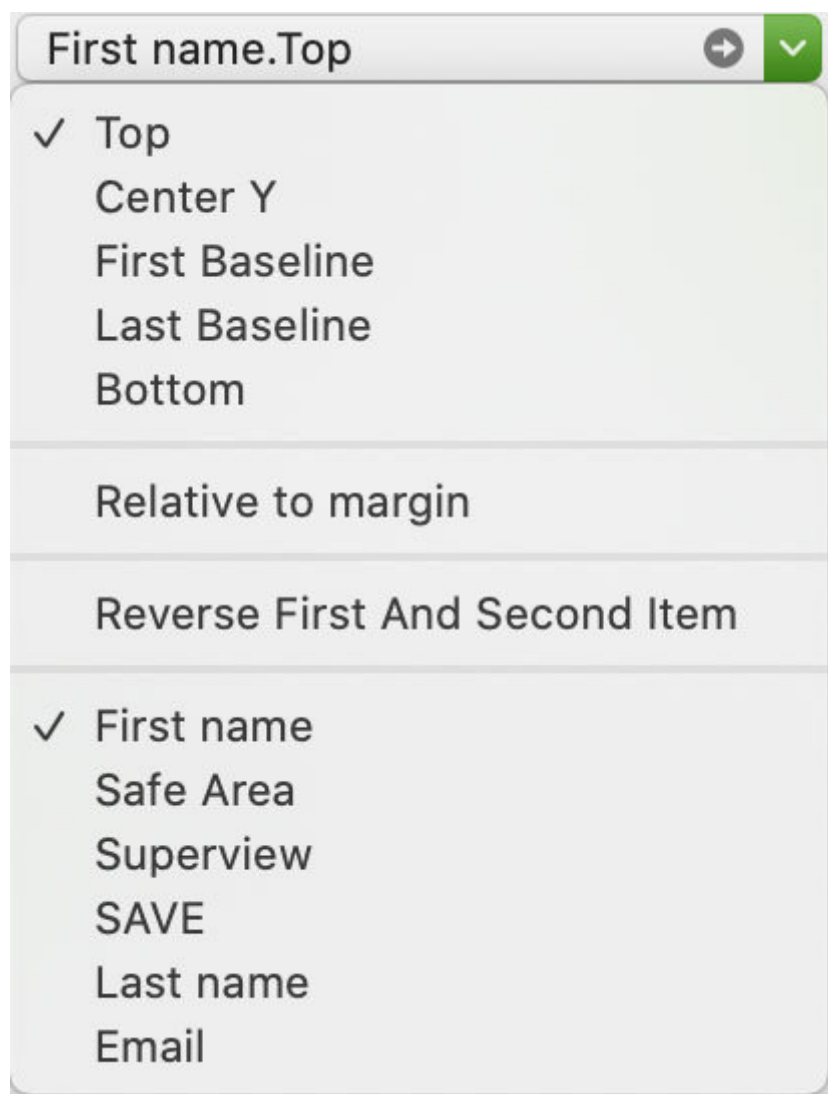


Figure 3.15.1: Item menu

This defines the type of relation between the two items; it can be equal, greater than or equal and less than or equal. For example, if the distance is greater than equal to 15, i.e., ≥ 15 , then if the other constraints set in a way, the text field can extend to more than 15px from the image.

Second This is the second view in the relation.

This is the value which would define the relation.

The priority of a constraint can be set from 1 to 1000. 1000 being the highest, i.e., required. This means you can have optional constraints and all other constraints below 1000 are optional. When the view is layout, the Auto Layout will try to satisfy the constraints based on their priority. If it is unable to satisfy a constraint which is optional, it just skips it and moves ahead. The optional constraint and relation work together. You can have two constraints with different types of relation and then change their priorities. For example, you might prioritize a view being closer to an image.

This is the multiplier of the constant. The default value is 1.

The installed check This indicates if the constraint is active or not. If you click on the small '+' on the side, you will see that you can add a variation based on the screen size. In other words, you can install certain constraints only for iPad or iPhone as shown in [*Figure*](#)

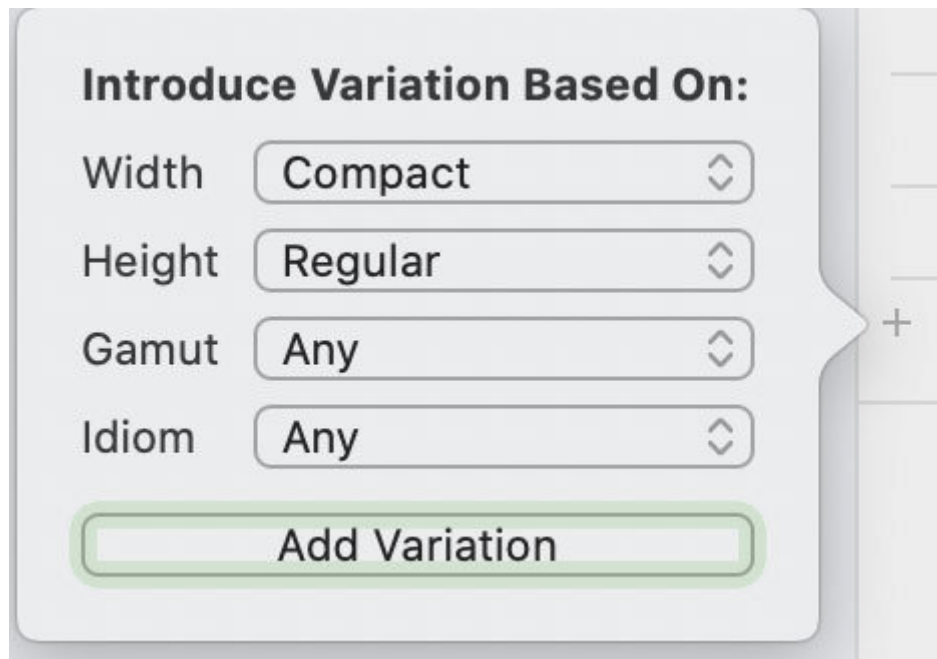


Figure 3.15.2: Size variation menu

So, if you *read* the constraint, it reads as follows:

$$\text{First name.top} = 1 \times \text{firstImage.bottom} + 15$$

Unsatisfiable layouts

Whenever you assign constraints, Auto Layout tries to figure out a solution based on them. The unsatisfiable error occurs when the system cannot find a valid solution for the constraints assigned by you. It is either because there are conflicting constraints or because you did not assign enough.

Usually, Xcode tells you about the conflicting errors. It also warns you about possible warnings. Refer to [Figure](#)

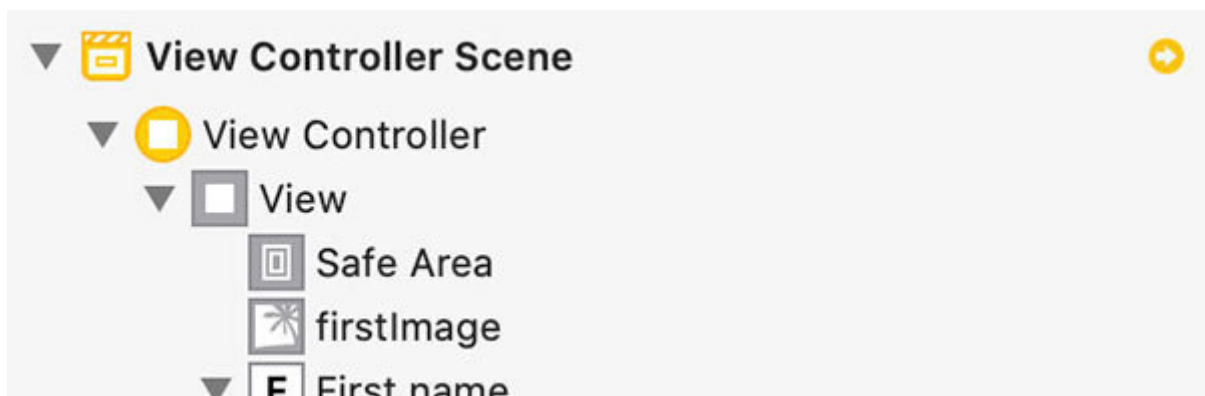


Figure 3.16.1: Auto layout warning arrow

If you click on the yellow arrow, you can even see the warnings as shown in [Figure](#). Auto layout also proposes possible solutions:

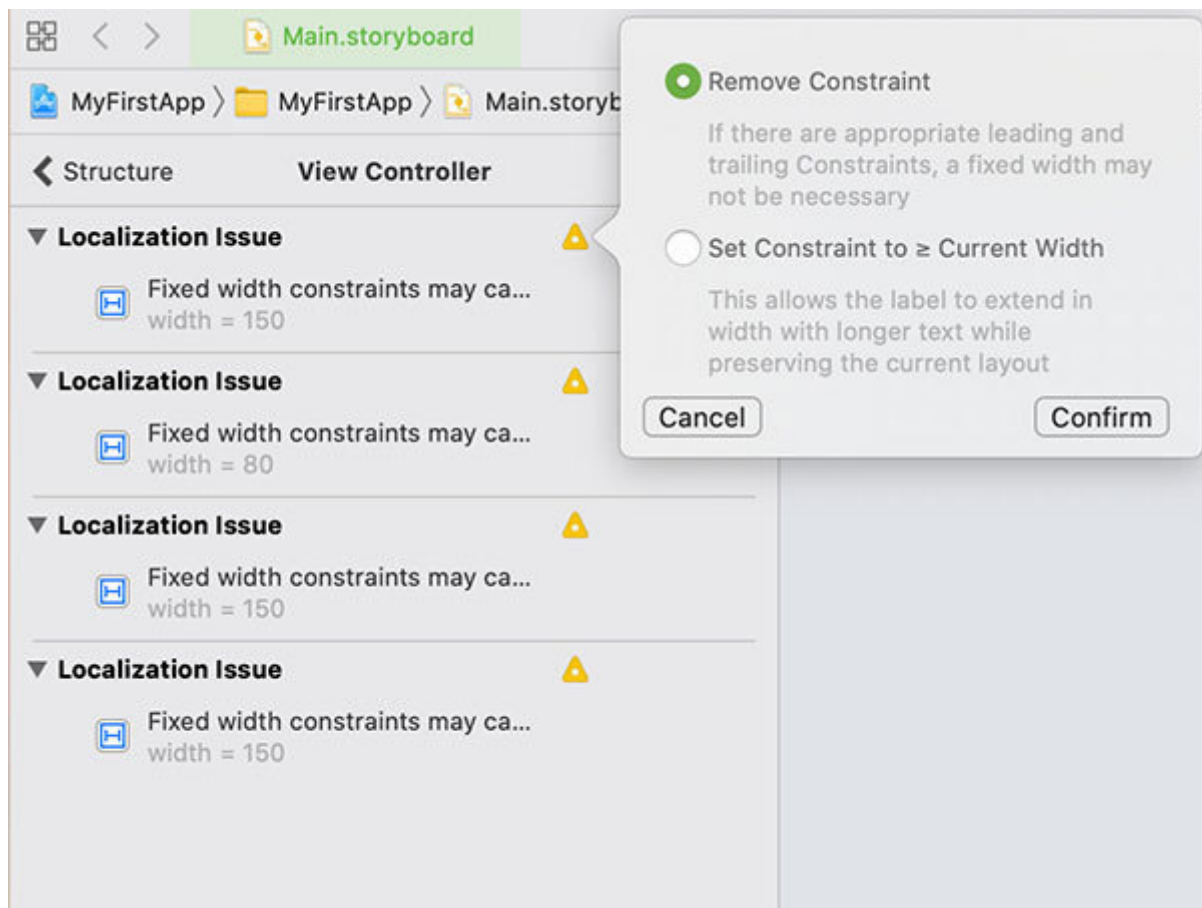


Figure 3.16.2: *Resolving auto layout warnings*

If you assign the wrong constraints, you will see a red arrow as shown in [Figure](#)



Figure 3.16.3: *Auto layout error arrow*

If you click on the red arrow, Xcode proposes you to delete the conflicting constraints, as shown in [Figure](#)

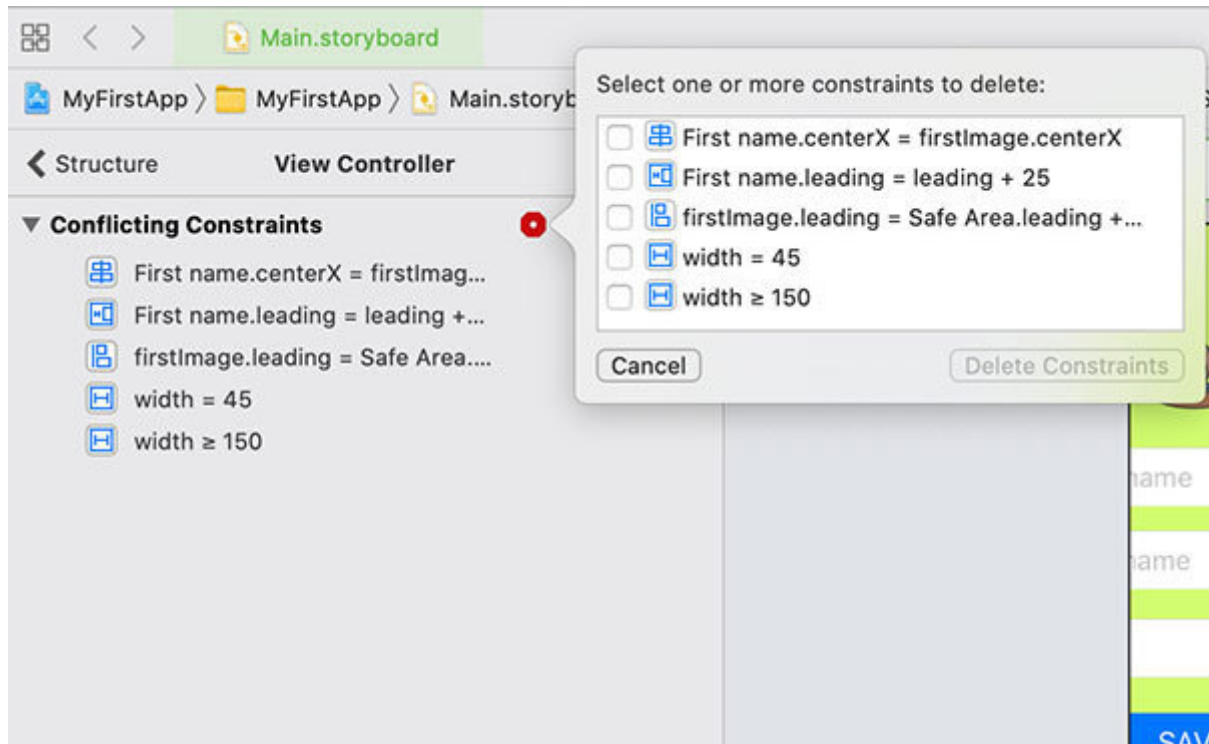


Figure 3.16.4: Resolving conflicting constraints

Ambiguous layouts

Sometimes, we assign constraints, as they can be satisfying more than one solution. Auto-Layout does not detect them at compile time. If all these constraints have high priority, when the app runs, auto-layout does not know which constraint to break to make a solution. This is called as **ambiguous**. In such cases, you can see the logs in the console listing the conflicting constraints.

Exercise 3.1: Add 2 more text fields to the app: Last name and email. Then, add a button with the text The button should be blue with white text. All the text fields must be aligned vertically, and the button must be small but centered horizontally with the rest (hint: use the Align tool). Check out the GitHub project for the completed version. Try to assign the right constraints, without having any warnings.

IBOutlets

We have now created a small form. But you will notice an issue; the keyboard of the simulator does not go down. We will fix this in an upcoming chapter. So do not worry about it. Then, how do we manipulate the UI? How to save when someone clicks on the button? Let us fix that:

Go to the Interface Builder and click on the **View Controller** under the **View Controller Scene** as shown in [Figure](#)

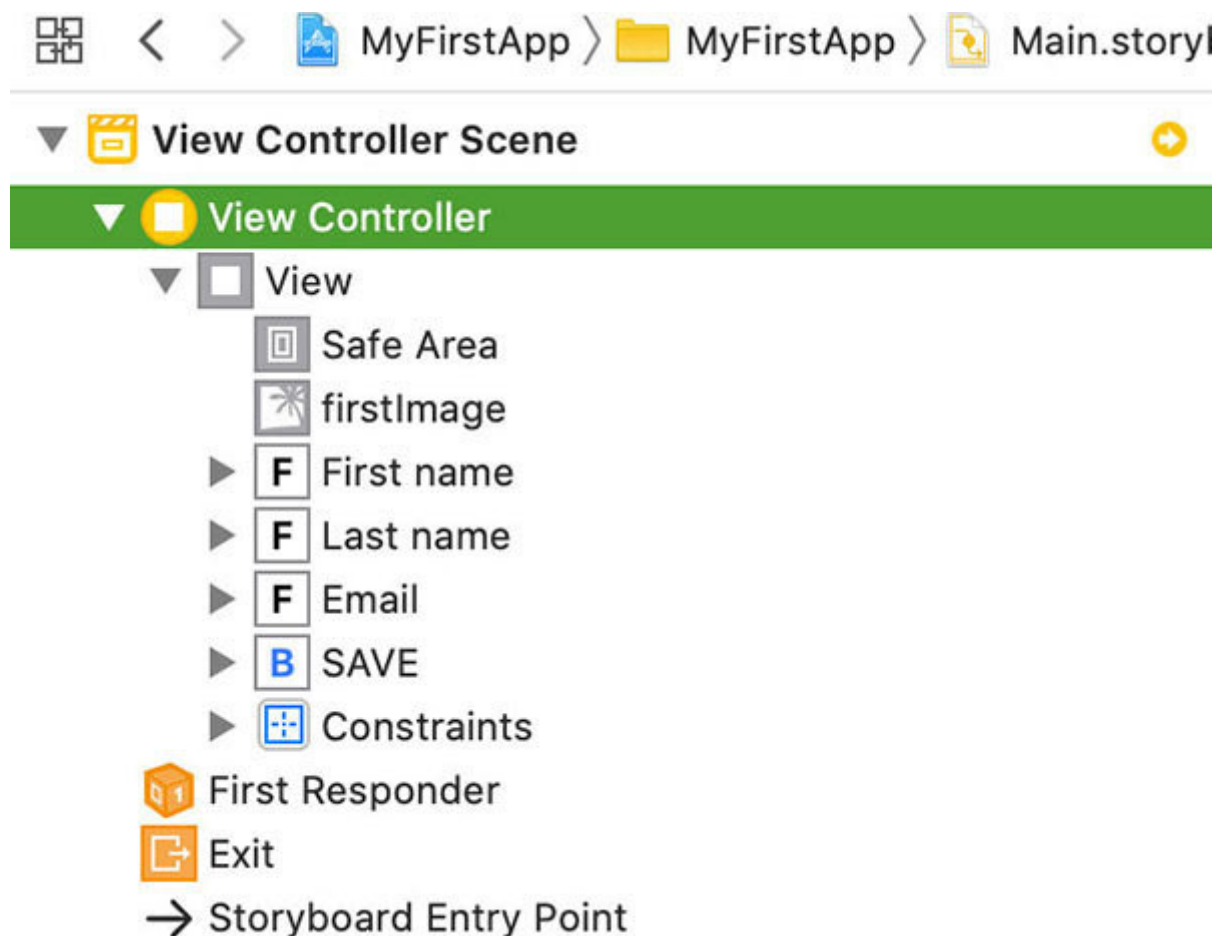


Figure 3.17: ViewController hierarchy

Now, in the Identity Inspector, check what is written in the **Class** field; it is **ViewController** as seen in [Figure](#)

Custom Class

Class

ViewController

Module

MyFirstApp

Inherit Module From Target

Identity

Storyboard ID

Restoration ID

Use Storyboard ID

User Defined Runtime Attributes

Key Path	Type	Value

+

—

Document

Label

Xcode Specific Label

×

Object ID

BYZ-38-t0r

Lock

Inherited - (Nothing)

Localizer Hint

Comment For Localizer



Figure 3.18: *Identity Inspector*

Tip: Click on the text field in the simulator. If you do not see the keyboard, it is probably hidden.

This is the same class **ViewController.swift** which you will see in the navigation pane. It only means that the interface builder knows where to go searching for the code when the view is in action.

However, the code does not know that the view has 3 text fields and a button. This is when we make IB stands for **Interface**

Like we opened Preview using the Adjust Editor Options, open the Assistant. You will see a new editor window next to the Interface Builder as shown in [Figure](#)

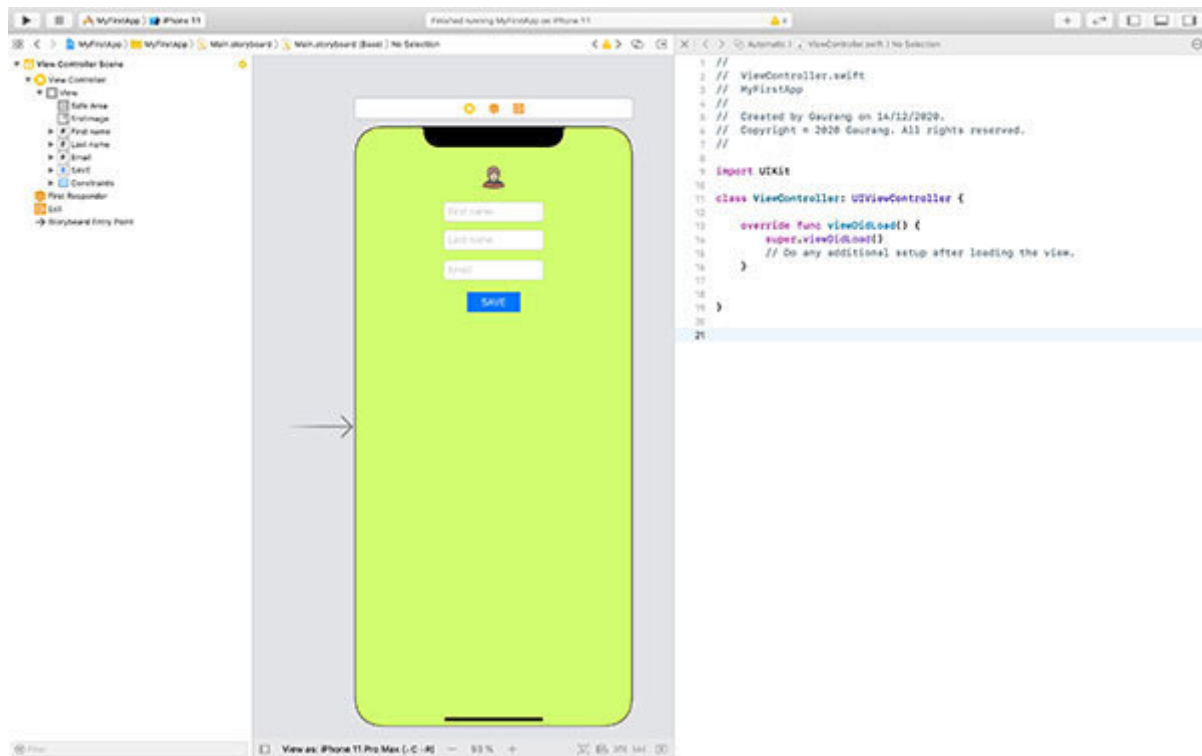


Figure 3.19: Assistant

Now, right click on a text field, and you will see a pop up like the one in [Figure](#). Here, you can choose the option, New Referencing Outlet. Now, drag and drop the arrow to the top part of the code:

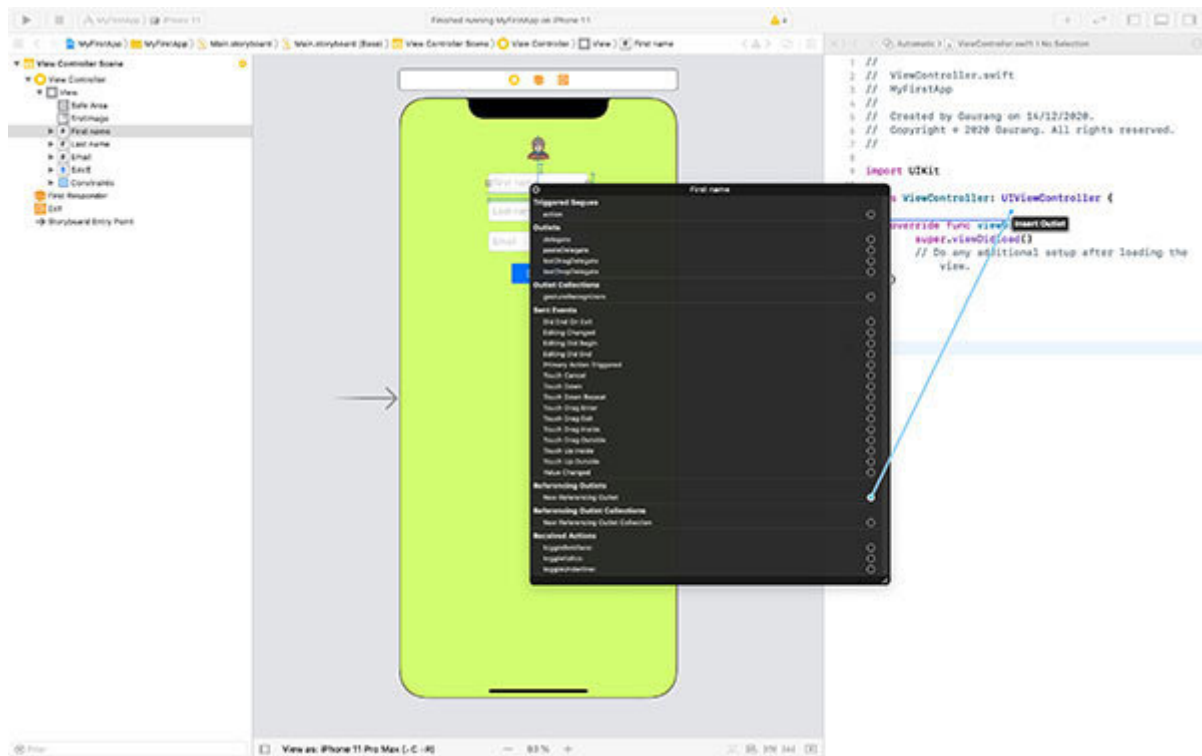


Figure 3.20: Referencing new outlet

[Figure 3.20.1](#) shows the details of the pop-up. You will see all the ways in which we can the elements to the code. You will see that they are divided into various categories: **Triggered Outlet Sent Referencing Referencing Outlet Received**

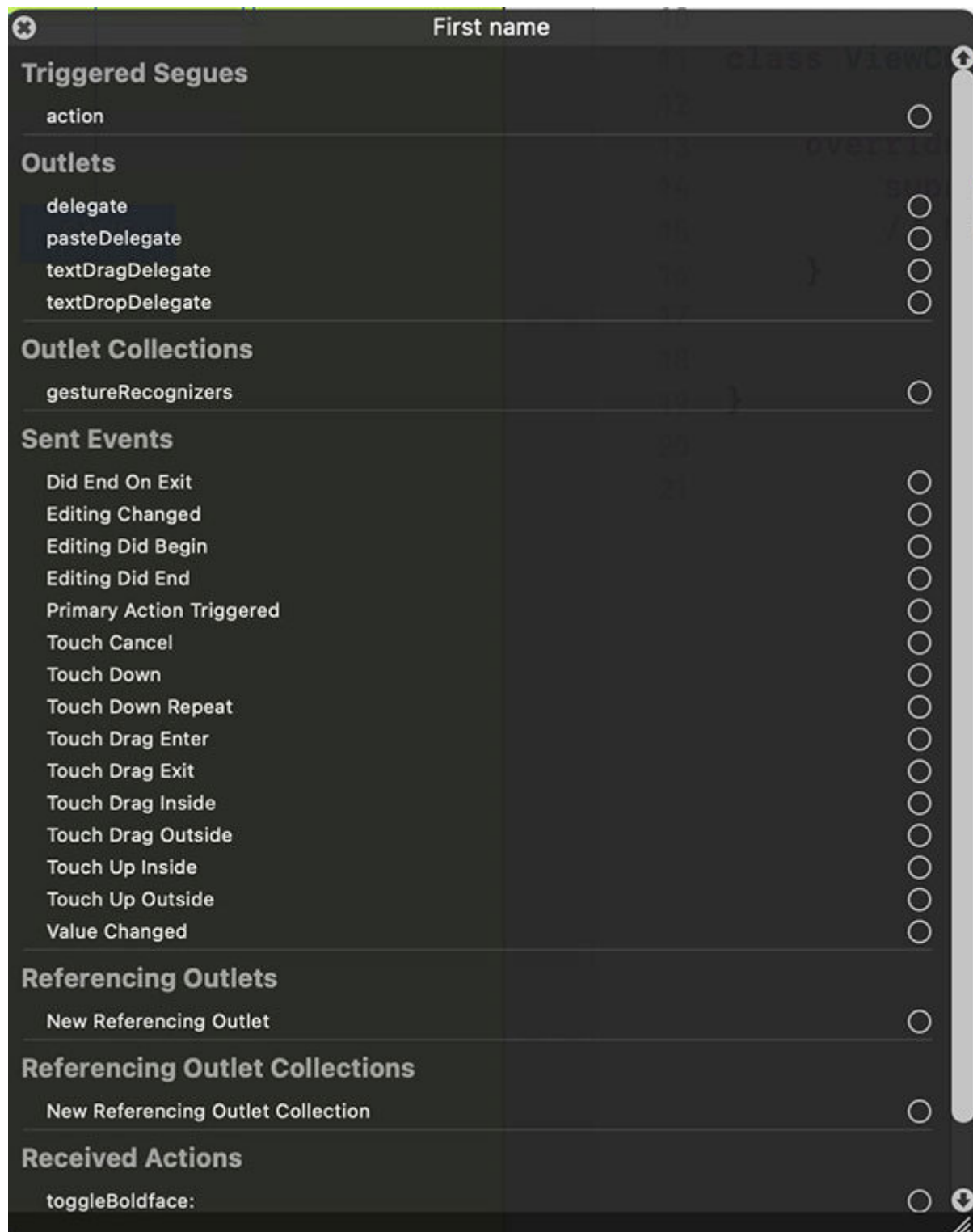


Figure 3.20.1: *Outlet popup*

As soon as you drop the arrow, a small pop up like the one in [Figure 3.21](#) is displayed. Here, you can type the name you want to

assign to the text field. By standard naming conventions (remember camel case?), name it

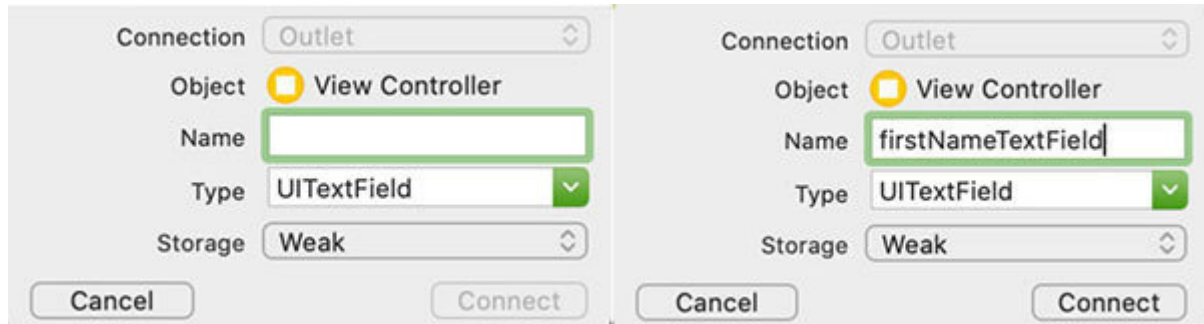


Figure 3.21: *Outlet naming popup*

Do the same for the next two text fields; **lastNameTextField** and

For the button, we are more concerned with its click event. Right click on the button and this time check out the **Send Events** list. We want the code to know when the user clicks inside the button; hence, we choose **TouchUp**

Drag and drop in the code, but this time below the **viewDidLoad** method. But within the scope of the ViewController *scope of a class is between the two curly braces {}, so is the scope of a method or function. We will learn about methods and functions in the upcoming*

Now, name your method **saveClicked** or *Your names should be easily readable and understandable by a third person reviewing your*

At the end, your code should look something like [Figure](#)


```

8
9  import UIKit
10
11  class ViewController: UIViewController {
12
13      @IBOutlet weak var firstNameTextField: UITextField!
14      @IBOutlet weak var lastNameTextField: UITextField!
15      @IBOutlet weak var emailTextField: UITextField!
16
17      override func viewDidLoad() {
18          super.viewDidLoad()
19          // Do any additional setup after loading the view.
20      }
21
22      // MARK: Button Clicks
23      @IBAction func saveClicked(_ sender: Any) {
24      }
25
26  }

```

Figure 3.22: IBOutlets in ViewController

Let us try to understand everything that is happening in the code:

Line 9 tells us that we are using the UIKit library for this class.

In line 11, we will declare our class **ViewController** and open its scope. From the previous chapter, we can understand that the type of this class is

Line 13-14-15 are our declarations of text fields. We already know the keywords **IBOutlet** and The new keyword is weak. Weak represents the type of reference the **ViewController** has of the text field. The other type is strong. If you look in the document

outline in the interface builder, you will see that the text field is added in the view and not directly in the view controller as shown in [Figure](#). So, the real owner of the text field is its direct view and not the ViewController; hence, it will hold a weak reference.

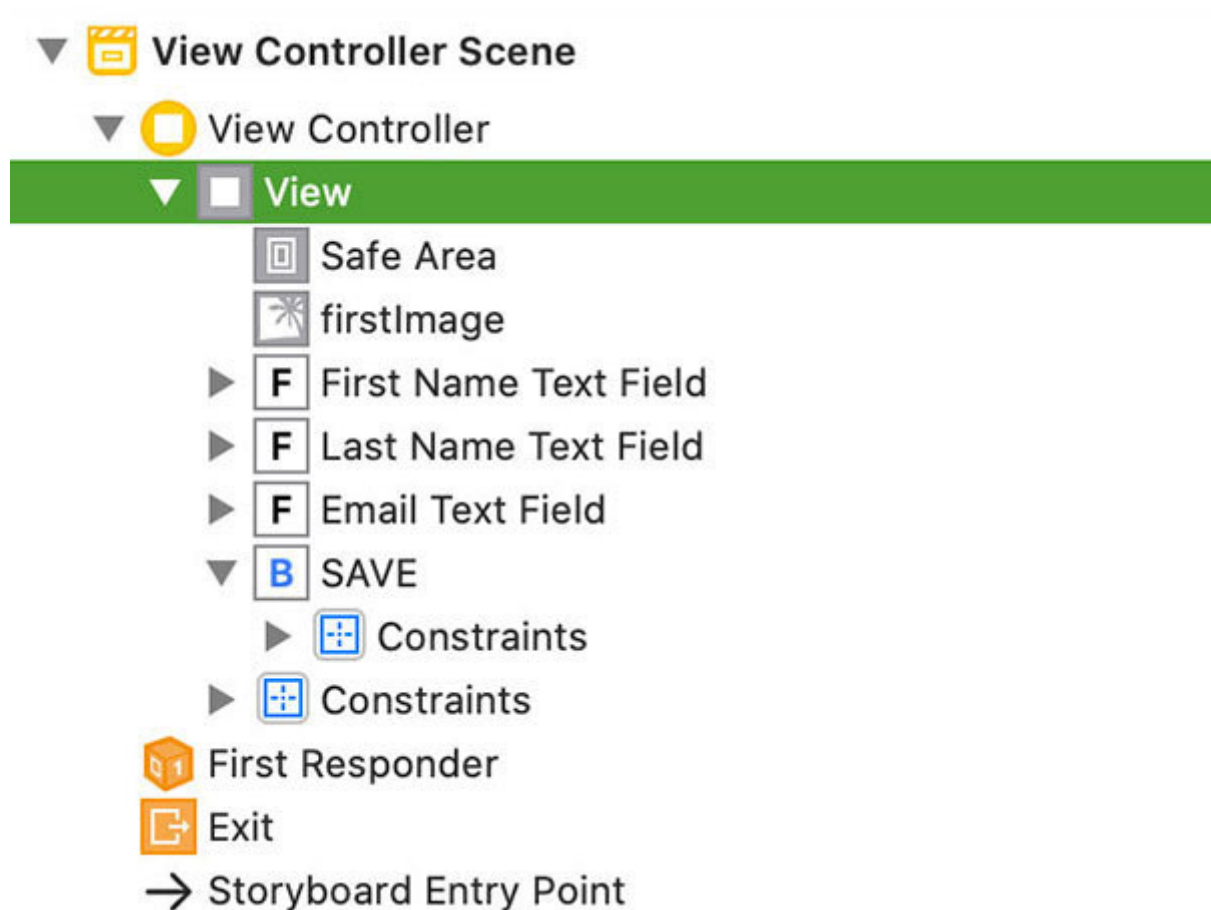


Figure 3.23: *Textfield placement in view hierarchy*

Line 17-20 is the **viewDidLoad** method. It is a view life cycle function. We learn more about it in the next topic of this chapter.

In line 22, you will see a new keyword **//**. You can use it to divide your code into zones. This is really helpful once the view controller becomes huge.

In line 23, we will see a new keyword `@IBAction`. This function is called when the button is clicked. So, within the scope of this method, we have to write the code which we want to execute once the button is clicked.

Tip: Instead of right clicking on each UI element, you can do **Ctrl+ drag and drop**. According to coding standards, all the declarations and connections are written right at the beginning of the ViewController's scope. They are then followed by the view life cycle methods, Then, there are all the public or private methods written by the programmer.

View controller lifecycle

In the last section, we created connections for our text fields. We have still not modified them. Where is this done? For this, we first need to understand the different life cycles of the View Controller. As we build complex interfaces, we have to show multiple views; transition from one view to another; or load and unload views on different clicks. The OS calls different methods in the View Controller when these transitions happen. The [Figure 3.24](#) explains it:

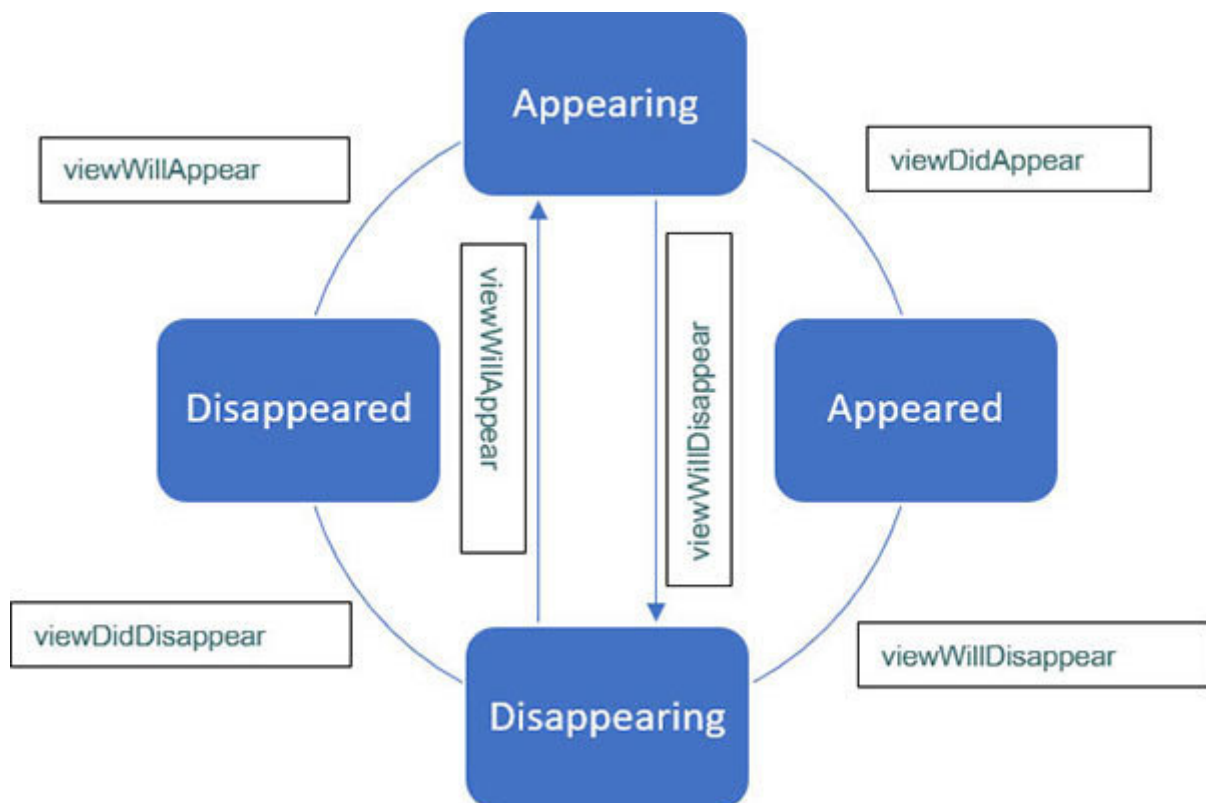


Figure 3.24: View Controller lifecycle

In the figure, you will see the flow of methods. To understand this, you need to just know basic English grammar. Will Appear gets called **before** Did Appear, and similarly Will Disappear gets called **before** Did Disappear. The OS calls these methods every time the view is added to the view hierarchy, the view is presented on the window. So, if you want to do something before the view is presented, *viewWillAppear* is the right choice and so on. However, these are called only after the view is loaded, *i. e.*, **This method is called only once, when the view is first created and loaded from the storyboard.** *viewWillAppear* is however called every time your view makes an appearance on the screen. The easy way to remember is did load get called when you load the view. Once the OS has loaded the view, only appear gets called.

Once the **viewDidLoad** method is called, you can be sure that all your **IBOutlet**s have been loaded, thus you can start with your modifications.

But why do we see the keyword **override** before Because our **ViewController** is a *subclass* of the **UIViewController** class. So, when the OS calls the **viewDidLoad** method, because we override it, it will be called in our **ViewController** class. We will learn about subclassing in the upcoming chapters.

With these things clear, let us now try to modify some of our code. Let us fill our text fields with default values. We must modify the code in It will look as follows:

```
override func viewDidLoad() {  
    super.viewDidLoad()
```

```
// Do any additional setup after loading the view.  
firstNameTextField.text = "John"  
lastNameTextField.text = "Mayer"  
emailTextField.text = "johnM@apple.com"  
}
```

Exercise 3.2: Change the title of the button to *Change* instead of *Save* by code. Then, change the name to Michelle Boyer and the email to michelleB@test.com once the Change button is clicked. Try to explore all the properties of button and text field and try to change through coding as much as possible. (Hint: Explore the Apple documentation.)

Types of views

In the first chapter, we learned that all the UI elements are in fact views. What does that mean? We will learn about *subclassing* later, but remember that all the different UI elements are subclasses of UIView. Apple provides us a library of UI components ready to use. Here, are the ones often used:

The father of all the UI classes. It is the basic view type and is also the container in every view controller.

UIScrollView: A type of container view like the UIView, but with a scrollable content. It is very useful when you have UI elements moving out of the screen area.

UIStackView: A recent addition to the UIKit; this is very useful to stack or place views in one container stack. The stack is a horizontal or a vertical orientation, often called as **HStack** and The spacing of the elements in the stack can be defined. This is highly useful when we want to manage device orientation and split screen. When we use stacks we do not have to manage the constraints of the individual elements present in the stack. Remember that stack view is only a container. It helps with alignment of its subviews. You cannot change its background color or add border to it.

UILabel: A simple text. Can be multiple lines. With the attributed text, you can even have multiple fonts and colors in one line.

UIImageView: A simple view to show different types of images.

UITextField: Single line entry field.

UITextView: Multiple line text entry field.

UITableView: Scrollable repetitive content. Apple manages this very well in terms of memory. A list of items can be shown in a scrollable content.

UICollectionView: This is a grid view with rows and columns. Also a recent addition to the UIKit library, this is very useful to show items in a Grid.

These are some basic views you will find over and over again in iOS apps. These are often changed and tweaked to the app's needs. Over the course of this book, we will try to touch most of them in our apps, but mastering them and playing around with them is your responsibility.

Exercise 3.3: Remove the constraints of the text fields using the remove constraints button. Then, embed the text fields into a stack view. Use the Embed In button of auto-layout, which we saw earlier. Then, add constraints to the stack view. Next, add another image to the project and on the button click, change the image view to the new image. Try to play around with the app by

adding more and more apps. Check out on GitHub for the final app.

Conclusion

We started this chapter by trying to understand *why we need* Then, we learned about size classes, and how we can use them to build the interface for different devices. Next, *we learned about constraints* and how they are useful for the adaptive layout. With the help of an app, we learned *to add different views and add multiple* We then moved on to learn IBOutlet and how to connect the interface builder to our class. We also learned to modify the values of the UI elements. We learned about the *view life cycle* and when is the right time to modify the code. We ended this chapter with a basic understanding of the *types of views* used in iOS apps.

With this chapter, we end the first section of our book. We now have a basic understanding of the iOS environment and some basic Swift. From here on, we will dive deeper. *Make sure you are familiar with all the terms used so* It would be easy to create apps and follow the steps if you remember the names right. All the best for the next section. In the next chapter, we learn more about Swift. We will learn about the syntax and the different structures used in Swift.

Key learnings

We learned how to design the interface for iOS apps using auto-layout.

We first understood the need of auto layout by using an example. We then read about the different reasons that might trigger a change in the UI.

We learned about the different size classes and then the constraints. We also looked at the different positions of the view.

We then saw the different controls used in the auto-layout. We looked at each of the pop-up and saw how we can use them to add or remove constraints.

We then added constraints to our image view by using the Ctrl+Dragging method.

We then tried to edit a constraint. We saw each property of a constraint and finally how to read it.

We then learned about the different errors you can face because of the conflicting constraints.

We then learned about IBOutlets and IBActions and how we can use them to make a connection between our interface and our code.

We ended the chapter by understanding the lifecycle of the view controller. We also completed our simple app by adding more text fields.

Test your knowledge

What are the left and right constraints of a view called?

Leading

Trailing

What are the different variations depending on the height and width?

Compact Width, **wC**

Compact Height, **hC**

Regular Width, **wR**

Regular Height, **hR**

Which are the different size classes based on the above variations?

wC hR

wR hR

What is the tool used to assign constraints called?

Pin tool

Which button helps you to arrange the views once you have assigned the constraints?

Update frames

What are the three types of relations for constraints?

Equal

Less than or equal

Greater than or equal

What is the highest priority of a constraint?

1000

What are the two ways to connect your views to the code?

IBOutlets

IBActions

Which is the base class of all the views?

UIView

Which is the default background color of the stack view? Can we change it?

Clear

No, we cannot change it as stack view is a container view only used to align the subviews.

Types and Control Flow

We will be diving a bit deeper into Swift in this chapter. We will learn about the different *types* or elements in Swift. We will start with the basic types and then spend some time with Strings. We will then learn about the collection type in Swift, i.e., arrays, dictionaries, and tuples. Then, we will learn about the different ways to loop over these collections. So, let us get our playgrounds ready for some Swift!

Structure

In this chapter, we will cover the following topics:

Basic types

String manipulations

Type conversion

Array

Set

Dictionaries

Tuples

If-else

Switch

For Loops

While

Objective

The objective of this chapter is to give you a basic understanding of the different types used in Swift. So, the next time you see a keyword, you won't be confused. Next, we will see the different conditional statements we can use in Swift. This will help you have better control over your code. We will end the chapter by learning about enumeration, i.e., how to loop over collections.

Basic types

We have already used variables. When we declare variables, we reserve a memory location for it. Different variables occupy different memory spaces, depending on the *type* of data they hold. Swift comes with all the basic *types* we already know of. Open a new playground so you can code in parallel.

Following are the basic types in Swift:

This is used to store 1 bit true or false value.

```
let bool: Bool = true // true or false
```

Integers are used to store whole numbers, signed, or unsigned. Swift has signed and unsigned integers in 8, 16, 32, and 64 bit forms. They have the same naming conventions as that of C. An unsigned integer is of type UInt8 for example. Likewise, a signed 64-bit integer is called

Each type of integer type has minimum and maximum properties. These are called **bounds** and you can find them easily. This can help you decide which type to use. You can find some examples in the code below:

```
let minimumValue = UInt64.min //the lower bound of unsigned 64
bit integer
```

```
let maximumValue = UInt64.max //the upper bound of unsigned
64 bit integer
```

```
let minValue = Int64.min //the lower bound of unsigned 64 bit
signed integer
```

```
let maxValue = Int64.max //the upper bound of unsigned 64 bit
signed integer
```

You can use `Int` instead of using specific **Int16** or **Int32** or **Int64**. The size of this integer depends on the size of the platform. **Int** is the same as **Int32** when used on 32-bit platform. **Int** is the same as **Int64** when used on a 64-bit platform. When you want to use unsigned form, you can use `UInt`. Following code shows the different types of `Int` in Swift:

```
//different ints
```

```
let number: Int = 2
```

```
let number1: Int8 = 4 // 1 byte, range -127 to 127
```

```
let number2: Int16 = 14
```

```
let number3: Int32 = 40 // 4 bytes, range -2147483648 to
2147483647
```

```
let number4: Int64 = 400 // 8 bytes, range -9223372036854775808
to 9223372036854775807
```

```
let number5: UInt = 2
```

```
let number6: UInt8 = 4 // 1 byte, range 0 to 255
```

```
let number7: UInt16 = 14
```

```
let number8: UInt32 = 40 // 4 bytes, range 0 to 4294967295
```

let number9: UInt64 = 400 // 8 bytes, range 0 to
18446744073709551615

Floating point These are numbers which have a fractional component like 52.996, -55.8958. Because of this, they have a wider range as compared to integers. They can be of two types:

This is used to store 32 bits decimal numbers. Usually, they are smaller in size.

let float: Float = 4.332 // 4 bytes, range 1.2E-38 to 3.4E+38

This is used to store 64 bits decimal numbers. They are larger decimal numbers.

let double: Double = -555.543 // 8 bytes, range 2.3E-308 to 1.7E+308

String manipulations

String is a collection or a sequence of unicode characters. These characters are encoding independent. This makes the Swift String a fast and a modern type implementation. It is also the reason why there are many ways in which we can access them. Thus, it makes String an important type to look at. Unlike many other languages, String in Swift is not an array. Although, the contents of a String can be accessed considering it as a collection of character values. There are many ways in which we can manipulate a String, so let us look at them one by one.

Creating

In the previous chapter, we have already seen how to create a string. Here are a few other ways to create it:

```
let place = "A beautiful place" //Direct initialisation with string literals
```

The preceding way is called creating using *string*. A string literal is a sequence of characters encompassed in double quotes. As discussed in [Chapter 2, Swift Programming](#), Swift *infers* the type of the preceding constant using **type**

There are also ways of creating by mentioning the type as given in the following example:

```
let name: String = "Some string" //By mentioning the type
```

In case you want to initialize the string without any value, i.e., empty string, you can do it by creating an instance as follows:

```
var emptyString = String() //Empty string declaration  
var emptyString = "" //This also initializes an empty string and is equivalent to the above.
```

You can use the **isEmpty** property to see if the string is empty:


```
if emptyString.isEmpty {  
  print("the string is empty")  
}
```

Special characters can be introduced using a backslash:

```
let specialCharacter = "\u{25}" //special characters can be  
introduced in a string with a backslash
```

```
let doubleQuotation = "\"Writing this between double quotes\""  
//double quotes
```

```
let singleQuotation = "\"Writing this between single quotes\""  
//single quotes
```

A multiple line string can be declared using the three quotation marks, delimiter (each on a new line), where you can use a `\` to add new line:

```
let multiLineString = ""  
Some strings are multiline  
and they should be defined \  
like this  
"" //These delimiters should always be on a new line
```

In case you want to use the special characters, without invoking their effects, you can place the string within **extended** For this,

you need to place your normal string within (#). Let us understand this with an example. In the following example, placing the string within the # sign prints the instead of introducing a new line character:

```
let noExtendedDelimiters = "First line \nSecond line"
print(noExtendedDelimiters) /*First line
Second line*/
let withExtendedDelimiters = #"First line \nSecond line"#
print(withExtendedDelimiters) //First line \nSecond line
```

Concatenation

String concatenation in Swift is very easy. Strings can be added by using a simple '+' sign. And like integers, we can concatenate literals using "+=":

```
let sampleString = "Sample" + " string" //concatenate strings  
using a simple addition sign
```

Do not forget what we learned in the second chapter, *Swift Programming*. If you want to mutate a string, you will have to assign it to a **var** instead of

```
var anotherSample = "This is"  
anotherSample += " a new sample" //mutate existing string using  
+=  
anotherSample.append("!") //we can use the append function
```

Interpolation

String interpolation helps us to introduce strings in an already existing string. This is done by using a backslash followed by the value in parentheses

```
let age = 144
let ageString = "I am \$(age) years old." // we can introduce any
kind of value
```

Tip: A string is a *value* The value of the string is copied when it is assigned to a variable or constant, or when it is passed to a function or method.

```
var originalString = "Original" //"Original"
var copiedString = originalString // "Original"
originalString = "I am changed"
print(copiedString) //"Original"
print(originalString) //"I am changed"
```

Iterating over strings

String is a collection of Unicode characters. Hence, we can iterate over them using the for loop:

```
let name: "String"
for character in ageString {
  print(character)
}
// s
// t
// r
// i
// n
// g
```

Inversely, you can initialize a string with an array of characters and get a continuous string:

```
let characters: [Character] = ["I"," ","a","m"," ","I", "r", "o"]
let characString = String(characters) //I am Iro
```

Counting characters

As we can iterate over Strings, we can also count the number of characters in Strings in Swift are treated as a collection of characters, so we must check the count, unlike the *length* which we did with NSString in Objective C:

```
ageString.count // 19
```

String indices

As explained earlier, String in Swift is a collection of Unicode characters, but it is not a collection type, i.e., it is not an array. Also, these Unicode characters can take different memory spaces when a String is stored. Each of these characters have an index type linked to them, These indexes can be used to access the characters at different positions. We can use them in the following way:

```
let occupationString = "iOS Developer"
occupationString[occupationString.startIndex] // i
occupationString[occupationString.index(before:
occupationString.endIndex)] // r
occupationString[occupationString.index(after:
occupationString.startIndex)] // O
let index = occupationString.index(occupationString.startIndex,
offsetBy: 7)
occupationString[index] // e
```

However, we cannot access a character as

Inserting and removing strings

We can use these indices to *insert* and *remove* strings by using the **insert(newElement: Character, at: String.Index)** and **remove(at: String.Index)** functions. Here are a few examples:

```
var occupationString = "iOS Developer"
occupationString[occupationString.startIndex] // i
occupationString[occupationString.index(before:
occupationString.endIndex)] // r
occupationString[occupationString.index(after:
occupationString.startIndex)] // O
let index = occupationString.index(occupationString.startIndex,
offsetBy: 7)
occupationString[index] // e
occupationString.insert("!", at: index)
print(occupationString)
occupationString.remove(at: index)
print(occupationString)
```

We can even use a range to remove many characters at the same time:

```
let range = occupationString.index(occupationString.startIndex,
offsetBy:4)..
//the range syntax is "starting point, offset and then end point).
In the above example we offset the startIndex by 4, so we start
```


with D until we reach the endIndex, thus removing the word
"Developer"

```
occupationString.removeSubrange(range)
```

```
print(occupationString)
```

Substrings

We can find a substring using indices and ranges. Substrings in Swift, for optimization, reuse the memory of the main String. So, if you plan to use the substring for a long term, do not forget to convert it into a new string:

```
let twoWords = "First, Second"
let wordIndex = twoWords.firstIndex(of: ",") ?? twoWords.endIndex
//here the starting point is the ',' till the endIndex
let firstWord = twoWords[..//the first word is First
let longTermSubstring = String(firstWord) //Convert the substring
so that you can use it for long term
```

String comparison

String comparison in Swift is very easy. You can compare two strings by the '=' sign. The '!=' sign is used to check the inequality:

```
let first = "First"
let second = "First"
let third = "Third"
if first == second {
    print("First and second are equal")
}
```

```
if second != third {
    print("second and third are not equal")
}
```

Type conversion

We saw different basic types such as Integers, Floats and Strings. But can we convert them from one form to another? Yes, we can transform values in the following ways:

Int to The **int** to **float** conversion is easy. But you will get a decimal point after the conversion. If nothing is there after the decimal, you will simply get “.o” as shown in the following example:

```
let originalInt = 40// 40
let convertedInt = Float(originalInt) //40.o
```

Float to The **float** to **int** conversion should be done carefully because you lose the digits after the decimal point, as shown in the following example:

```
let originalFloat = 58.47 //58.47
let convertedFloat = Int(originalFloat)//58. You lose the decimal values
```

Int to Int to string is simple. You just have to pass the **int** in the **string** initializer as shown in the following example:

```
let intToString = String(originalInt) //"40"
```

String to This is a bit tricky. If you convert a string, where you know the contents is an integer, you will get a valid result. But if you convert a normal string to you will get nil, as seen in the following examples. Even if you convert a string containing a float value, you will get nil after converting to nil:

```
let backToInt = Int(intToString)//40  
let someString = "string"  
let someInt = Int(someString) //nil
```

Float to This follows the same rule as that of Int to String which we saw in the previous example:

```
let floatToString = String(originalFloat) //"58.47"
```

String to This follows the same rules as that of String to Float:

```
let backToFloat = Float(floatToString) //58.47
```

Collection type

The most common collection types you will be using in Swift are arrays, dictionaries, sets, and You probably must have not heard about tuples. Swift is unique in a way that even collections in Swift are a type. So, in Swift, an array of numbers is a different *type* than an array of strings. Open your playgrounds for some collection manipulations.

Tip: All the collections declared with `let` are immutable and those declared with `var` are mutable. In Objective C, you had `NSArray` and `NSMutableArray` as two different classes. But in Swift, if you assign a collection type to `let`, it's an immutable collection type and if you assign it to a `var` it's a mutable collection type. This means you can add, remove, or change the items in the collection.

```
let array: [Int] = [1, 2] //immutable array
var array: [String] = ["A", "B"] //mutable array
```

Arrays

An array is an ordered collection of objects. The objects in arrays are not unique and can be repetitive. The Swift array is written as **Array** or its shorthand [Element], where Element is the *type* the array stores. [Table 4.1](#) shows the diagram representation of an array:

array:
array:
array:
array:
array:
array:

Table 4.1: Array

Creating an array.

Following are some ways in which you can create new arrays:

```
let emptyArray = [Int]() //general syntax to define an array
let immutableArray : [Int] = [0, 0, 0] // creates an empty Int array
with 3 initial values of [0, 0, 0]
var mutableArray = Array(repeating: 0, count: 3) // this does the
same job as the above array. But because we have declared it as
a variable it is mutable
```

The following example shows how to create an array with **array** Array literals are a list of values enclosed in squared brackets [value 1, value 2, value 3]. In this example, we are creating an array of type String with default values:

```
var typeInferredArray = ["Hello", "Bonjour", "Namaste"] //Declaring
an array with literals, Swift can infer the type
```

In the preceding example, you will observe that Swift can guess the type of the array depending on the values present in the array. As mentioned earlier, this is called **type** This saves some coding and we do not have to specify the type of the array every time.

Another way of creating an array is by adding two arrays. But ensure that these arrays are of same type. If not, Swift will throw

a compile time error. In the following example, we are creating a new array by adding two arrays of type integers created above.

```
var addedArray = unmutableArray + mutableArray //adding two  
arrays of the same type is very simple. Just use the '+' operator
```

Modifying and accessing an array.

Accessing and manipulating an array is very easy. Arrays are ordered and indexed; hence, you can access the objects with the help of indices. The same method can be used to modify or remove objects. However, you cannot insert a different type of object in a different type of array, i.e., you cannot insert a **String** in an **Int** type array. Here are some self-explanatory ways of accessing and modifying an array:

```
mutableArray.append(22) //adding new values at the end of the array
```

```
mutableArray += [44] //adding new values at the end of the array  
mutableArray += [55, 66, 77] //adding multiple values at the same time
```

If you come from the Objective C background, you are probably used to accessing objects by using the **objectAtIndex** method. But in Swift, we do this by using shorthand, this is called The following example shows how we can use a subscript to access the value at an index using the subscript. In the upcoming chapter, we will learn how to write our own subscripts:

```
mutableArray[3] //accessing value at a specific index using subscript syntax
```

```
mutableArray.insert(22, at: 1) //inserting at a specific index
```

```
mutableArray[o] = 11 //changing the value at a specific index using subscript
```

Earlier, while looking at Strings, we checked whether a string is empty by using the **isEmpty** property. We use the same property for our array. Here is the example:

```
emptyArray.isEmpty //checking if an array is empty
```

```
addedArray.count //to check the number of elements
```

Similarly, the contains method can be used to check whether an array has a certain element. Again, the type is important. You cannot check whether a string is present in an array of type

```
mutableArray.contains(22) //checks if an array contains a certain element. Returns true if found
```

Swift also comes with some methods which make removing values easy. Here are the examples:

```
mutableArray.remove(at: 2) //remove an object at the index you want
```

```
mutableArray.removeFirst() //removes the first object
```

```
mutableArray.removeLast() //removes the last object
```

```
mutableArray.reverse() //reverses the array
```

Tip: A multi-dimensional array is an array within an array within. If you want to define one, the syntax is as simple as follows:

```
var array2D: [[Int]] = [[1, 2], [3, 4], [5, 6], [7, 8]]
```

```
var array3D: [[[String]]] = [[[ "A", "B"], ["C", "D"]], [{"E", "F"}, ["G",  
"H"]]]
```

Sets

An unordered collection of **unique** objects. The objects in a set are distinct. You can use Sets when the order of the objects is not important to you, or you want to make sure that the object is the only one in the set. We can use the array literals “[]” to define a set. However, the type of the Set should be mentioned only if you are not initializing it with any value. [Figure 4.1](#) shows the representation of a set:

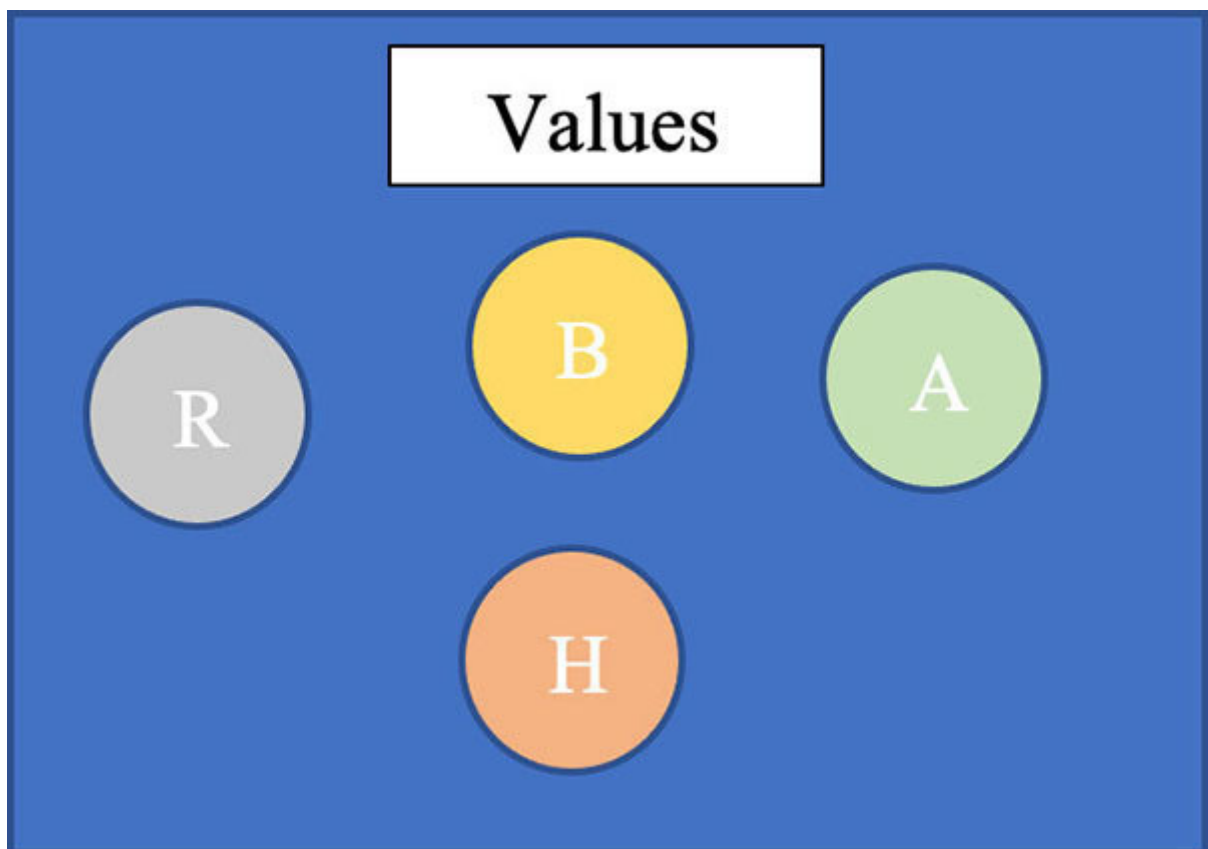


Figure 4.1: Set

Hash values for Sets

For a type to be stored in a set, it must be a type which is hashable provides a way to compute a **hash value** for itself. A hash value is an integer. It is the same for all the equally comparable objects such that if $x == y$ the hash value of x and y is the same. But do not worry, all the basic types which we saw at the beginning of the chapter are hashable by default.

Creating a set

Here are the ways in which you can create a set:

```
let emptySet = Set() //this is how you can declare a set
let immutableSet : Set //the type of a set must be mentioned
```

Like how we created an array using array literals, we can create a set with array literals too. But we must specify the type as Set. You can specify the type of Set using but thanks to type inference, Swift can do that for you:

```
var mutableSet : Set = ["Apples", "Oranges", "Apples"] //the result
of this would be a set with {Apples, Oranges}. We have used the
array literal [] to define a set.
var typeInferredSet : Set = [1, 3, 2] //When initialized with values,
Swift can infer the value
```

Modifying and accessing a Set

When you access a set, remember it is unordered. So, there are no indices or sequences. Imagine a room full of people standing randomly. No two people are the same. How do you add or remove them from the room? By calling out their names. This is exactly how sets work:

```
mutableSet.isEmpty //checking if a set is empty
mutableSet.count //to check the number of elements in the set
mutableSet.insert("String")//insert a new object in a set
mutableSet.contains("Apples") //check if the set has a specific
value
mutableSet.remove("String") //remove a member by its name
mutableSet.removeFirst() //removes the first object
```

But we are dealing with Sets. So, if you remember your Mathematics lessons, we can use operations like and so on. Here are a few examples:

```
let animalsSet : Set = ["tiger", "cat", "lion", "dog", "cat", "cow",
"hamster"]
let petSet : Set = ["dog", "hamster", "cat", "cow"]
let birdSet : Set = ["crow", "sparrow", "cuckoo"]
let commonSet = animalsSet.intersection(petSet) //finds the
common members of the two sets
//Result {"cat", "hamster", "dog", "cow"} Refer figure 4.2
```


let uncommonSet = animalsSet.symmetricDifference(petSet) //the opposite of intersection

//Result {"tiger", "lion"} Refer [figure 4.2](#)

let mixedSet = petSet.union(birdSet) //adds two sets

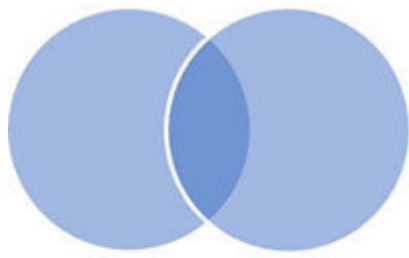
//Result {"dog", "sparrow", "cat", "crow", "cow", "cuckoo", "hamster"} Refer [figure 4.2](#)

petSet.isSubset(of: animalsSet) //returns true if second set has all the values of the first set. here true

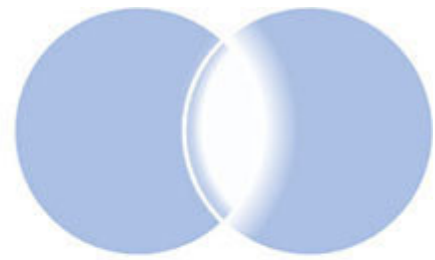
petSet.isDisjoint(with: birdSet) //to determine if two sets are completely different

animalsSet.isSuperset(of: petSet) //returns true if the first set has all the values of the second set. here true

The [figure 4.2](#) will help you understand the different operations like Intersection, Symmetric Difference and Union.



Intersection



Symmetric Diff



Union

Figure 4.2: *Operations on a Set*

Dictionaries

This is a key value type collection. The members of a dictionary are stored in pairs with a This key is unique, and like sets, the members are also unordered and are retrieved with the help of these keys. Now that we have looked at Arrays and Sets, you should easily understand manipulations of a dictionary. Dictionaries are used when we want to store values in key, value pairs. The keys make it easy to access and modify the values. [Figure 4.3](#) shows the representation of a dictionary:

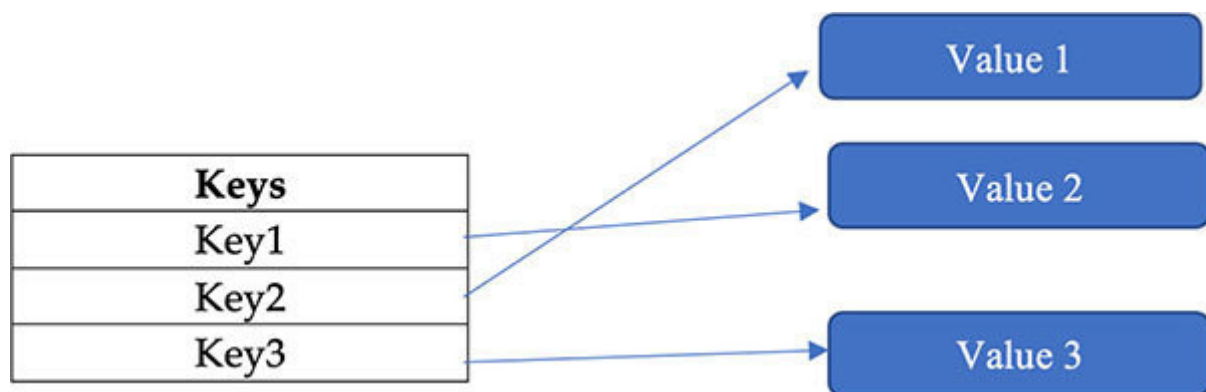


Figure 4.3: Dictionary

Creating a dictionary

Like we saw for Array, the dictionary type in Swift is written as **DictionaryValue**> and the shorthand is As discussed in the Sets section, the basic types in Swift are This also makes it possible to use them as keys in the dictionaries. Here are a few ways to create a dictionary:

```
//Dictionary
```

```
let dictionary : DictionaryString> //A standard declaration of a dictionary
```

```
let emptyDictionary = [Int : String]() //Creates an empty dictionary with keys of type Int and values of type String
```

We saw how we can create an Array or a Set using array literals. Similarly, we can create a dictionary with default values, using a dictionary literal, **[key1: value1, key2: value2, key3: value3]**:

```
var mutableDictionary = ["key1" : "value 1", "key2" : "value 2", "key3" : "value 3"] //creating a dictionary with default values using the dictionary literal
```

Like Arrays and Sets, the way to check whether a Dictionary is empty or to count the number of elements remains the same. The following is the syntax:

```
mutableDictionary.isEmpty //to check if the dictionary is empty  
mutableDictionary.count //number of members
```

Modifying and accessing a dictionary.

In Arrays, we access the data using indices. Similarly, in Dictionaries, we access the data using the keys. Arrays are ordered, while Dictionaries are not.

```
mutableDictionary.values //returns a collection of all the values of  
the dictionary  
mutableDictionary["key4"] = "value 4" //add a new member  
mutableDictionary.updateValue("key1", forKey: "watermelon")  
//update an existing member  
mutableDictionary["key3"] = "apple" //update an existing member  
using subscripts  
mutableDictionary.removeValue(forKey: "key1") //remove value for a  
key  
mutableDictionary["key4"] = nil //removes the value for a key. same  
as above
```

Tuples

We saw Arrays, Sets, and Dictionaries. They are ordered, indexed but they are still strongly typed. You cannot store different types together. An array of type **Int** will only store Ints. This is where we can use Tuples are small collections, where you can store different types of members together. *They are used only when we must store simple data* or if we have to return multiple values in a function. As we shall see in the upcoming chapters, in normal functions, we can only return a single value. But with tuples, you can return multiple values at the same time. *Remember not to use tuples to store large complex data.*

You can have tuples with any types **Double**, or **Int**,

In the following example, we will create a tuple of type **String, Int**, As seen for other types, thanks to type inference, we do not have to mention the type of the tuple:

```
var infoTuple = ("Adam", "Scott", 43, 09892356) //Creating a tuple
```

By default, the elements in the tuple have incremental index. So, will have index 0, index 1, and so on. This makes accessing the values of the tuple easy. We can access them using their indices as seen in the following example:

```
infoTuple.0 //accessing a tuple by its index Result: "Tom"
```

Like arrays, we can use the index to change the value. But instead of the subscript syntax, we will use the dot convention:

```
infoTuple.o = "Tom" //modifying a tuple with the help of index
```

But if you have multiple values in a tuple, accessing using the index can be a bit tedious. This is where labelling the values comes in handy. You can give names to each of your values. Then, you can access the values using their labels. Refer to the following examples:

```
let labelledTuple = (name: "Adam", lastName:"Scott", age:43,  
tel:09892356) //A tuple with labels  
labelledTuple.lastName //accessing a tuple by its label name
```


Control Flow

When you write a code, it is not a simple set of instructions. There are many conditions and you want a certain code to be executed only when a certain event happens. For example, you want the keyboard to hide, when the return key is pressed, i.e., you want to *control the flow* of the code. Swift provides us with many ways to do this. Here are a few.

If else

We have already seen some examples of *if-else* in the previous chapters. This is the simplest form of a conditional statement: if condition 1, then execute statement 1, else execute statement 2. Here is the syntax:

```
//a simple if statement
if mutableDictionary.count > 2 {
    print("Some random error message")
}

//an if statement with an alternate statement to execute
if mutableDictionary.isEmpty {
    print("The dictionary is empty")
} else {
    print("The dictionary has some value")
}

//multiple if statements
if mutableDictionary["key3"] == "apple" {
    print("The dictionary has an apple")
} else if mutableDictionary["key3"] == "orange" {
    print("The dictionary has an orange")
}
```

Switch

In the previous example of the if statement, you will see that there are multiple condition checks. Switch provides a cleaner way to match patterns and execute conditional statements. A switch statement is ideal when we need to match the value against one or more values of the same type.

Here is the syntax of a switch statement:

```
switch value {  
case pattern 1:  
statement 1  
case pattern 2:  
statement 2  
case pattern 3:  
statement 3  
case pattern 4:  
statement 4  
case pattern 5:  
statement 5  
default:  
statement 6  
}
```

A switch statement must be exhaustive, i.e., it should have a statement for every pattern. If it is not possible to do it, we must

provide a **default** statement. You will notice that unlike the switch statements, you have seen in other languages, there is no In Swift, if a pattern matches, the code exits the switch statement, without checking the other patterns. But if we want the code to continue, we have to use the keyword

```
let animal = "Tiger"
```

```
switch animal {  
case "Cat", "cat": //compound case  
    print("Meow Meow")  
case "Tiger", "Leopard": //when you want multiple patterns to have  
    same result  
    print("Big Meow")  
case "Lion":  
    print("Very big Meow")  
case "Dog":  
    print("Bow Bow")  
default:  
    print("I am a human")  
}
```

With Swift, you can even use tuples and other complex patterns for matching. The switch statement in Swift is very powerful as compared to the other languages.

For in

We saw different types of collections like arrays, sets, and even sequence of characters like Strings. But how do we iterate over these collections? This is where we use the **for** loop. Apple dropped the traditional **for** loop i.e. **i=0; i** in Swift 3.x. The one which we see now is the **for in** loop. Let us see a standard example:

```
let fruits = ["Apples", "Oranges", "Banana"]
```

```
for fruit in fruits {  
    print(fruit)  
}
```

Let us try to understand what is happening in the preceding code. **fruit** is the name we are assigning for the members of the array. As discussed earlier, Swift can infer the type of the members of the array. So, we do not need to specify it. We can read the preceding code as, *every fruit in fruits array print*

You can see that the **for in** loop has been simplified over the traditional for loop. Here is how we can iterate over a dictionary. *Swift does not support iterating over a tuple:*

```
for (key, value) in mutableDictionary {  
    print("key: \(key) - value: \(value)")  
}
```

```
}
```

We can even use the `values` property of the dictionary and iterate only over its values:

```
for value in mutableDictionary.values {  
    print(value)  
}
```

While learning about Strings, we saw how they are a sequence of characters. Thus, we can use the **for in** loop to iterate over it:

```
let iteratingString = "I am a Swift developer"  
for character in iteratingString {  
    print(character)  
}
```

But what if we want to iterate by using a count or indexes. This can also be done by using the close range operator (`...`) and half-open range operator (`..<`). Swift also allows us to write for loops with defined intervals. By looking at the examples, you will also understand the difference between a closed range operator and a half-open range operator:

```
//closed range operator. starts at 0 and ends at 3  
for index in 0...3 {  
    print("index \(index) - \(mutableArray[index])")  
}
```

```
//half open range operator. starts at 0 and ends at 2
for index in 0..<3 {
  print("index \((index) - \((mutableArray[index])")
}
```

In the preceding example, if you do not want to use the you can replace with underscore. Notice that in the following example, the looping is important and not the index:

```
for _ in 0..3 {
  print("loop executed")
}
//increment upto a count with an increment of 2 at each step
let interval = 2
for num in stride(from: 0, to: mutableArray.count, by: interval) {
  print(num)
}
```

Exercise Create a dictionary with the format ***name:*** and some default values. Add more values to it. Now, iterate over it using the for loop and print the following message: ***movie has x*** Where movie is the movie name and x is its rating.

While

With the **for** loop, we iterated over collections. We already knew the number of items in the collection. In real-world coding applications too, we have an idea. What if we do not know the number of iterations we might have to perform? What if we want to iterate until a certain condition is true or false. This is where the **while** loop comes in picture. This is the general form of the **while** loop:

```
while condition {  
  code  
}
```

The while loop starts by evaluating a condition and continues executing the code inside until the condition is satisfied. In the following example, we start at a given value of a second. If the seconds have not reached 60 or a minute, we increase the value by 5 seconds. As explained, the **while** loop will first check whether the seconds have reached a minute, only then it will go ahead and increase the value by 5 seconds:

```
var seconds = 5  
let minute = 60
```

```
while seconds <= minute {  
  //the condition is evaluated and only  
  then the loop is executed
```



```
print("The minute is not over")
seconds += 5
}
```

```
print("The minute is over")
```

Exercise Create an array with the first 8 months of the year. Add the other months using `append` and `+=` . Now, iterate over this array using a `while` loop and print the following message: ***n is*** Where *n* is the number of the month. For example, Month 1 is January.

The **repeat while** loop, on the other hand, evaluates the condition at the end and repeats until the condition is satisfied. Here is the syntax with one example:

```
repeat {
  code
} while condition
```

If we change the solution we just saw, to `repeat-while`, it should look something like the following code:

```
var seconds = 5
let minute = 60

repeat {
  print("The minute is not over")
  seconds += 5
```

```
//the loop is executed at least once before evaluating the  
condition.  
} while seconds <= minute
```

```
print("The minute is over")
```

So, what is the difference between these two? The **repeat while** loop executes at least once before checking the condition. On the other hand, the **while** loop does not execute if the initial condition is satisfied. Hence, in the above example, you will see that the log prints “The minute is not over” first and then move ahead to increment the seconds until it is incremented to 60.

Exercise Iterate over the numbers from 1 to 100. For every number divisible by 3, your code should print and for every number divisible by 5 your code must print and for every number divisible by 3 and 5, your code must print Otherwise, the code must print the number.

We will solve the preceding exercise right here and try to use all what we have learned so far. Let us try to first write the algorithm.

Algorithm

Loop over the numbers from 1 to 100.

If number is divisible by 3, print Fizz.

If number is divisible by 5, print Buzz.

If number is divisible by 3 and 5 or 15, print FizzBuzz.

If none of the above, print the number.

Open a playground. Let us write the code for the first point. Then, write your algorithm in comments, so it is easy for you to follow.

We will use the for loop. Here, we must use the index, so we will use a closed range operator with index:

```
//1. Loop over the numbers from 1 to 100
for index in 1...100 {

//2. If number is divisible by 3 print Fizz
//3. If number is divisible by 5 print Buzz
//4. If number is divisible by 3 and 5 or 15 print FizzBuzz
//5. If none of the above, print the number
}
```

Now, let us write the conditions for the points 2 to 5. We will use simple if else condition. We will use the mod operator to check whether the numbers are divisible. Our code should look something like the following:

```
//1. Loop over the numbers from 1 to 100
```

```

for index in 1...100 {

if index % 3 == 0 { //2.If number is divisible by 3 print Fizz
print("Fizz")
} else if index % 5 == 0 { //3.If number is divisible by 5 print
Buzz
print("Buzz")
} else if index % 15 == 0 { //4.If number is divisible by 3 and 5
or 15 print FizzBuzz
print("FizzBuzz")
} else {
print(index) //5.If none of the above, print the number
}
}

```

As soon as you run the preceding code, you will see the problem. It does not print **FizzBuzz** for the number 15 and its multiples. Let us correct that:

```

for index in 1...100 {

if index % 15 == 0 {
print("FizzBuzz")
} else if index % 3 == 0 {
print("Fizz")
} else if index % 5 == 0 {
print("Buzz")
} else {
print(index)
}
}

```

```
}
```

The preceding code looks perfect. But can we improve it further? Let us try to use the pattern matching of Swift. We are going to check for both the division conditions and then use the pattern which we will get as an output:

```
for index in 1...100 {  
  
    switch (index % 3 == 0, index % 5 == 0) {  
    case (true, true):  
        print("FizzBuzz")  
    case (true, false):  
        print("Fizz")  
    case (false, true):  
        print("Buzz")  
    default:  
        print(index)  
    }  
}
```

The two conditions used in the switch case will generate a pair of values, (true, true), (true, false), (false, true), and (false, false). We can either write all the conditions exhaustively or leave the last one in default.

This is a very simple problem commonly asked in interviews. If you are able to solve this with you can show that you have a better understanding of the language.

Conclusion

We did a lot of theory in this chapter. But I am sure you must be very fluent with Playgrounds and Swift in general. We learned about the basic types in Swift and dived into the world of Strings. Then, we saw the different types of collections such as *Dictionaries*, and We learned how to create, access, and manipulate them. Then, we moved towards the control flow of Swift. We got to know the conditional statements; *if-else* and Then, we learned about *for* loops and *while* loops and saw how they are different from the ones found in traditional languages. In the next chapter, we will make a small app and dive deeper into Swift. We will learn about a very important type called *Optionals* and introduce

Key learnings

We started the chapter by learning about Integers. We saw signed and unsigned integers. Swift supports 8, 16, 32, and 64 bit forms of integers.

We then saw floats and doubles which are used by Swift to support fractional number types.

We then learned about Strings and different ways to access and modify it.

While learning Strings, we saw how we can use delimiters to have multiple line strings. We also learned about extended delimiters to print special characters without invoking their values.

Strings are in fact a collection of characters and properties like 'count' can be used to check the length of the String.

We even saw different collection types in Swift.

Arrays are indexed collections of objects. We even saw the use of subscripts to access and modify an array.

We then learned about Sets, which as opposed to arrays are an unordered collection of distinct objects. This changes the way we

access objects in an array.

We then saw Dictionaries, which are key value collection types.

We even saw Tuples, which are a unique type of Swift. They are used to return multiple values at the same time. They can be even used with labels and serve as small collections which have different types of data at the same time.

Then, we learned about the different ways of iterating over these collection types: For loop, while loop, repeat while.

We even saw the conditional operator if-else and switch.

Test your knowledge

What is the size of the type Int?

Int32 on a 32-bit platform

Int64 on a 64-bit platform

What are floating numbers?

They represent the fractional values.

What are 64 bit floating numbers called?

Double

Are Swift Strings value types or reference types?

Value types

Which is the property used to check whether a string is empty?

String.isEmpty

Which property is used to check the length?

String.count

What are extended delimiters?

Extended delimiters are used when you want to print special characters without invoking their values. `#"Line 1\nLine 2"#`

What are arrays?

Indexed collection of objects which may or may not be unique.

What are sets?

Unordered collection of unique objects.

What is a dictionary

A key-value type collection.

Optionals and more

In this chapter, we will learn about a very interesting type in Swift, the *optional* . This makes the code crash proof and very clean. We will see how we can use this in our day-to-day coding. Then, we will learn about some *convenience features like Guard let and defer*. Then, we will learn about type **aliases** and **access control** . We will start this chapter by building a simple *calculator* app. Then, we shall end the chapter with an exercise, where we will together implement a more elaborate calculator.

Structure

In this chapter, we will cover the following topics:

Optionals

Unwrapping

Guard let vs if let

Defer

Control transfer statements

Type aliases

Access control

Objective

The objective of this chapter is to help you understand Along with convenience features like **guard let** and **defer**, you will learn to write fail safe code. Swift is a modern language, and if you are coming from a traditional language background, it will take time to exploit these Swift features. The objective of the app building exercise is to revise all that we have learned so far. This app will also be a good practice exercise for Auto Layout.

Simple calculator

We will start with a simple calculator. We will try to use whatever we have learned so far. Then, we will turn this calculator into a full functioning calculator. Let us start building our Simple Calculator:

Open Xcode and create a new single view app called

Open the Story board and add three text fields in the view. Use the view library for this. Refer to [Figure](#)

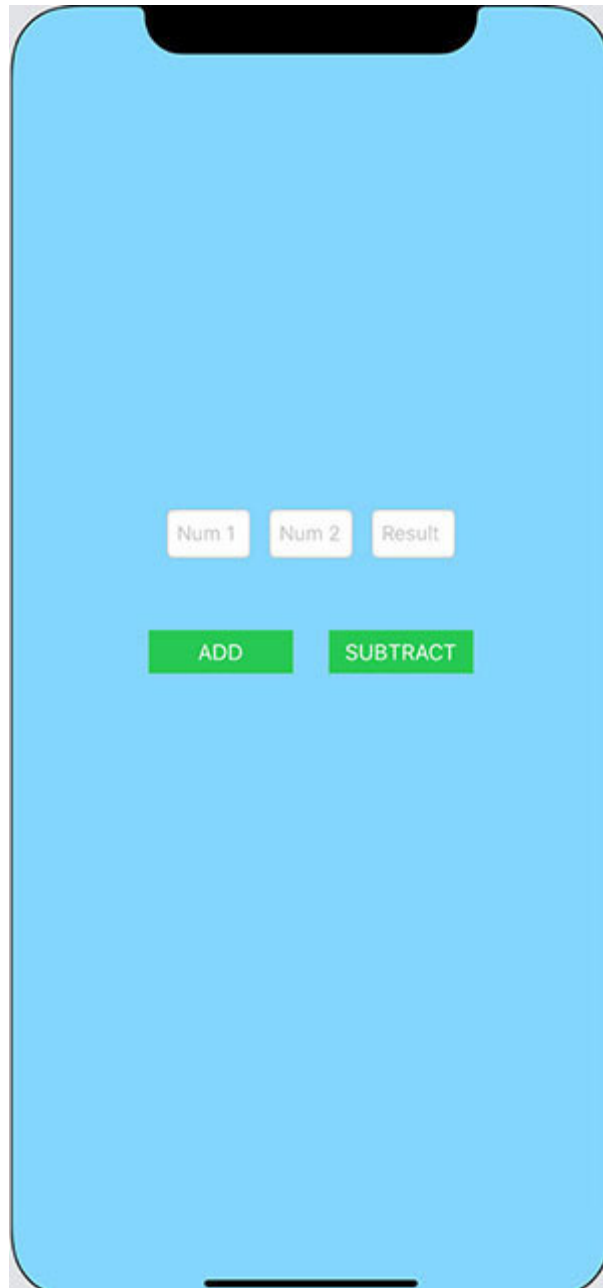


Figure 5.1: *Simple Calculator final UI*

Then, add two buttons **Add** and

Next, we will make connections to our ViewController. Open the assistant pane and then drag and drop the connection to the ViewController as we did in [Chapter 3, Auto](#) Refer to [Figure](#)

Next, we will create **IBActions** for our buttons. You should have two empty functions as follows:

```
@IBAction func subtractButtonClicked(_ sender: Any) {  
  
}
```

```
@IBAction func addButtonClicked(_ sender: Any) {  
}
```

Now, it's time to add some constraints. You can see that we have 3 identical text fields. So, we can place them in a Stack View. Click on the three of them and choose the Embed In Stack from the Embed In tool. Refer to [Figure](#)

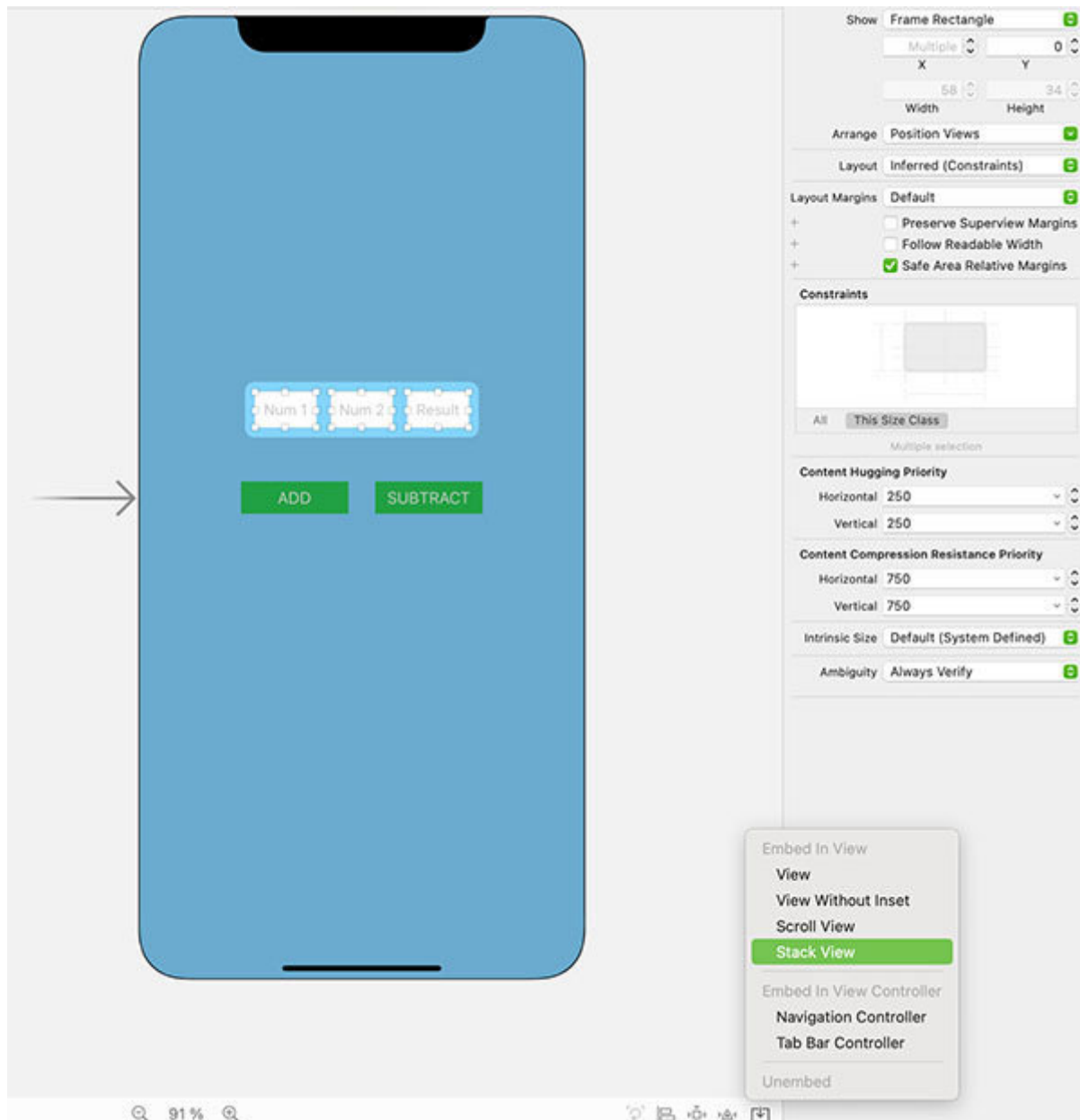


Figure 5.4: Embedding the text fields in a stack view

You can repeat the same for the buttons. Then, you can center the two stack views and add some widths and heights. Otherwise, you can choose your own constraints. You can even try to do it without the Stack View.

We have made our basic setup. Now, we will take a break and learn some more about Swift.

Optionals

Look at the following two declarations:

```
var firstNumber: Float?  
var secondNumber: Float?
```

Do you see anything different? Did you notice the question mark at the end of the This is what an **optional** type looks like. With this declaration, we mean that the **firstNumber** can have a **nil** value, *i.e.*, it is an **optional** Why is this important? Because *Swift is a strongly typed* If a variable is going to have a **nil** value, we need to explicitly tell Swift about this. Swift makes a clear distinction between a variable which can never have a **nil** value and a variable which can have a **nil** value. Remember that **optional is also type**. So, if **String** is a type, is an optional type.

Now, open the ViewController of our Simple Calculator app. If you have created the necessary connections for your text fields, you should see something like this:

```
@IBOutlet weak var firstNumberTextField: UITextField!  
@IBOutlet weak var secondNumberTextField: UITextField!  
@IBOutlet weak var resultTextField: UITextField!
```

Do you notice the **!** at the end? What does that mean? It means that the text field will always be non-nil, *i.e.*, we are telling the

code that there will always be a text field in our view. The opposite of the ?

Unwrapping

When a variable is of the optional type, it can either have a value or not have a value. So, how do we determine this? When we extract a value out of an optional type, it is called There are many ways to do this:

Unwrapping with The simplest way is to check whether our optional value is not nil and then retrieve the value it has:

```
if firstNumber != nil {  
  let thirdNumber = firstNumber!  
}
```

Forced This is when we assume that the optional type will have a value. We put an exclamation mark! in front of the optional type to extract the value. Be careful while doing this because if your optional type is nil, this can lead to a crash. Here is an example of forced unwrapping:

```
let thirdNumber = firstNumber!
```

But the **firstNumber** is **nil** the app will crash because we told the code that **firstNumber** will surely have a value and instead it was nil.

Or, we can use the optional to have a default value in case the **firstNumber** is This is called the **nil-coalescing** This can be done in the following way:

```
let thirdNumber = firstNumber ?? 22.0
```

The preceding code checks whether the **firstNumber** is if not it will use 22.

If let and guard let

Another way of checking the non-nil condition for the optional type is by using the **if-let** and **guard**. Here are the examples. The example on the left is of if-let and the one on the right is that of guard-let. Refer to [Table](#)

Table 5.1: *If let vs guard let*

Let us try to understand the difference between these two ways of

If we *read* the if let, it will be as follows; *if the name optional string has a value, let it be assigned to the*. Remember that only if the name has a value, the if condition will be executed.

If we *read* the guard-let, it will be as follows; *guard the value of the name optional string in firstName, else print a message and move out of the*. Otherwise, proceed with the. Remember that if the name is the execution stops and moves out of the function.

In the **if-let** way, the first name exists in the scope of the if condition, i.e., in the '{ }' while in the **guard-let** way, the first name exists outside the scope of the guard condition.

In the the **else** condition is i.e., if the name is **nil**, you can continue writing the code in the scope of

While in the **else** condition is *compulsory* and is used to exit the scope of So, if the name is then the **else** condition must return and exit the scope of its function, you cannot continue.

In both the cases, if you want to check the unwrapping, just try to use the variable You will see in the autocomplete that the type of **firstName** is String and not String? Refer to [Figure](#)



Figure 5.5: Type changes from String? to String

So, which one to use? Remember the golden rule: If in doubt, use the guard let way.

Coming back to our code. Let us try to use what we just learned.

We will make step by step changes to our code:

Use **guard let** in one function and **if let** in another. Here is how your code will look like:

```
@IBAction func subtractButtonClicked(_ sender: Any) {
```

```

if let firstValue = firstNumberTextField.text, let secondValue =
secondNumberTextField.text {

}
}

```

```

@IBAction func addButtonClicked(_ sender: Any) {
guard let firstValue = firstNumberTextField.text else {
return
}
guard let secondValue = secondNumberTextField.text else {
return
}
}
}

```

In the **subtractButtonClicked** function, we have used the **if** As you can see, we can chain them together in one sentence. Next, we will try to use the type conversion we learned in the previous chapter:

```

@IBAction func subtractButtonClicked(_ sender: Any) {
if let firstValue = firstNumberTextField.text, let secondValue =
secondNumberTextField.text {
if let firstInt = Int(firstValue), let secondInt = Int(secondValue) {
resultTextField.text = String(firstInt+secondInt)
}
}
}
}

```

You will see we used **if let** for conversion too. Why? Because the conversion returns an optional value.

We will change the and use the force-unwrapping this time:

```
@IBAction func addButtonClicked(_ sender: Any) {  
    guard let firstValue = firstNumberTextField.text else {  
        return  
    }  
    guard let secondValue = secondNumberTextField.text else {  
        return  
    }  
    let result = Float(firstValue)! + Float(secondValue)!  
    resultTextField.text = String(result)  
}
```

Go ahead and test the app and check for different test cases.

Defer

Sometimes, we want a piece of code to execute, no matter what. The best example is that of performing operations on a file. First, we open a file and then we perform some operations. But no matter whether the operations are successful or not, we want that the file should be closed. This is when the **defer** keyword is used. This convenience feature was introduced in Swift 2.0. It allows us to execute some code at a later time. Now, let us look at [Table](#)

Table 5.2: *Defer use case*

The example on the left helps you understand the sequence in which a **defer** statement is called. As seen, because we printed Step 2 in the scope of the defer statement, it gets printed at the end. The Step 1 is executed, then Step 3, and before the execution leaves the function, we will execute the defer statement.

The example on your right is a dummy real-world example. Here, we are performing operations on a file. But we want to make sure that the file is closed before we leave the function. Hence, we defer the closing.

Though the defer is called later, it should not be confused with an **asynchronous**

Control transfer statements

We saw in the preceding examples how the order of the execution changes when we use certain keywords. Like in guard-let, we allow the control out of the scope of the function, if certain conditions are not met. We use the return keyword. Similarly, there are other keywords which we can use to do this. Let us understand them.

This keyword is used when you want the loop to stop what it is doing and start again from the beginning. The control does not quit the scope of the function. The following example will help you understand it better:

```
let foods = ["bread", "ice cream", "chocolate", "soft drinks", "salad",  
"dal rice"]  
let unhealthyFoods = ["ice cream", "chocolate", "soft drinks"]  
var healthyFoods = [String]()  
  
for food in foods {  
  if unhealthyFoods.contains(food) {  
    continue //control moves to the next iteration without executing  
    the append statement  
  }  
  healthyFoods.append(food)  
}  
print(healthyFoods) //Result: ["bread", "salad", "dal rice"]
```

This is used when we want to stop the execution of the loop and exit the scope as soon as some condition is matched. Let us modify the function which we just wrote and replace the **continue** keyword with You will see the difference in the results:

```
let foods = ["bread", "ice cream", "chocolate", "soft drinks", "salad",  
"dal rice"]  
let unhealthyFoods = ["ice cream", "chocolate", "soft drinks"]  
var healthyFoods = [String]()  
for food in foods {  
    if unhealthyFoods.contains(food) {  
        break //the loop execution is broken and the control leaves the  
        loop.  
    }  
    healthyFoods.append(food)  
}  
  
print(healthyFoods) //Result: ["bread"]
```

As soon as we encounter the first unhealthy food, the execution will move out of the loop; hence, we see that the **healthyFoods** array has only one element.

The **break** statement is also used in switch. In the **switch** statement, as soon as the **break** keyword is encountered, the control stops the execution and moves to the line out of the curly braces of the **switch** statement. This is useful when you want to break the execution right after the first case is matched.

You must have already seen the difference between a switch statement in Swift and a traditional switch statement of C. Swift does not have a break statement. This is because, in Swift, as soon as the first case is encountered, the execution moves out of the switch statement. But what if we want to match multiple cases? This is when we can use the word Fallthrough makes the control move to the next case, instead of moving out of the switch statement. Study the following example to understand this better:

```
var statement = ""
switch unhealthyFood {
case "chocolate":
print("chocolate is bad")
break
case "ice cream":
fallthrough
default:
statement = unhealthyFood + " is bad for health"
}

print(statement) //ice cream is bad for health
```

This is a simple keyword to understand. We have already seen the use of this with guard-let. It moves the control out of the function and returns to the place where the function was being called from.

This is used during error handling. We will use this to raise exceptions and move the control out of the function. We will see

its use case in [Chapter 9, Error handling and](#) while studying error handling.

Type aliases

Swift allows you to rename the existing data types. This is a very cool feature, which you can use to make your code cleaner and more readable. The general syntax of the type alias is as follows:
typealias type name = type expression

```
typealias calculatorNumber = Float?  
var firstNumber: calculatorNumber = nil  
var secondNumber: calculatorNumber = nil
```

In the preceding example, you will see that we have renamed the optional float type as So, if we want to use we will instead use Also if we use the type alias, we have to set an initial value. Here, the initial value is nil.

Access control

We will take a small detour to understand **access**. As the name suggests, it is the ability to control the access to certain parts of your codes. But first, we need to understand some of the following terms:

A module is a shippable chunk of code like a library or a framework. You can find an example in your own code. The line **import UIKit** is in fact importing the UIKit module in your class. Simply put, modules are chunks of code you can easily import in your app.

Source A single file within a module is called a **source**. For example, the files you see in your navigation pane:

SceneDelegate.swift are in fact source files.

Along with access to your source files, you can grant access to your entities or variables, types, and functions. Swift defines five levels of access control. They are as follows, starting from the least restrictive to the most restrictive:

public and Properties, classes, and methods accessible by any part of the module and by the module which imports it have a **public** or **open** access. All the classes in the UIKit have a public access.

This level enables intra-module access. Entities used in any source file of the same defining modules are accessible to each other. Example, all the entities defined in your files.

When you do not want to grant access to your entity outside of your source file, you can use the **fileprivate** access level.

This access level restricts the use of an entity to its enclosing declaration. For example, if you define a variable and mark it as private, it won't be accessible outside the scope, {}, in which it is declared.

To simplify the writing of code, all the entities in your code have a default **internal** access level, unless specified otherwise. This makes writing code faster.

Understanding the delegate

If you remember, we had made a connection from our text fields to our view controller. We will learn more about delegates in the [Chapter 8, Protocols and](#) In brief, the text field delegate gives us all the functions, which we can use to manage the text field better. The following function for example will hide the keyboard when we press the *return* key:

```
func textFieldShouldReturn(_ textField: UITextField) -> Bool {  
    textField.resignFirstResponder() //hides the keyboard  
    return true  
}
```

This is a delegate method. We can even the delegate to the view controller programmatically. Adding the following lines to the **viewDidLoad** tells the text field that the view controller is going to be its delegate, or view controller is a place where it will find functions to manage itself:

```
firstNumberTextField.delegate = self  
secondNumberTextField.delegate = self  
resultTextField.delegate = self
```

Sophisticated calculator (Exercise)

We will try and build a better calculator. This is also an exercise for you. We will start by creating a new project. Then, we will first work on the layout and slowly add functionalities. The final UI of our app should look something like [Figure](#)

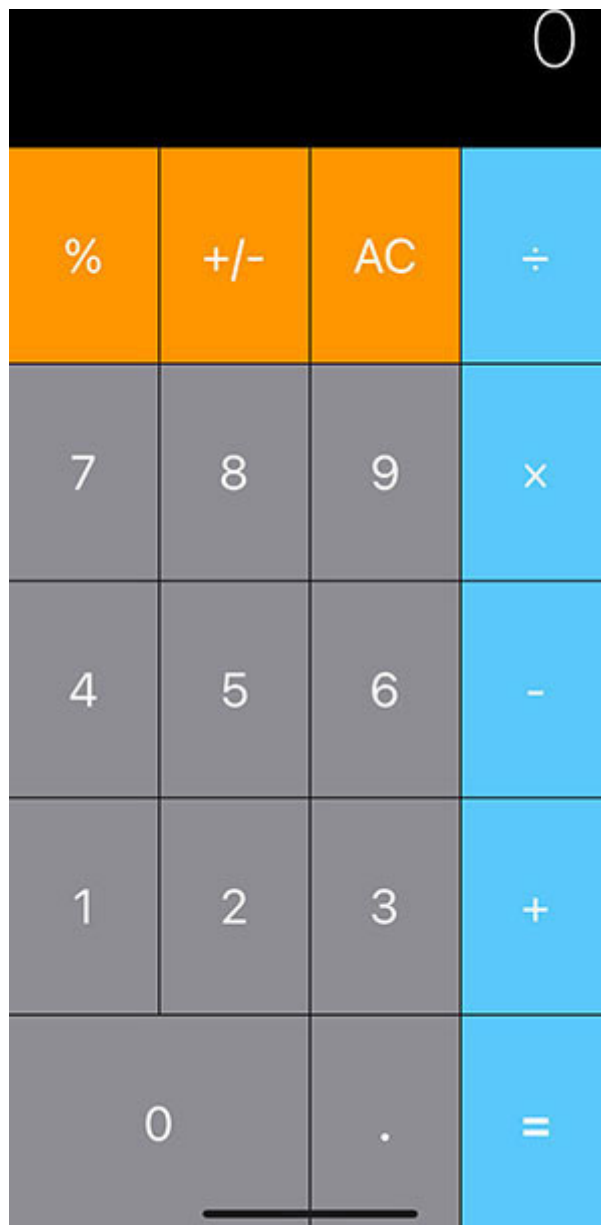


Figure 5.6: *Sophisticated Calculator final UI*

Let's start!!

Open Xcode and create a new app called *SophisticatedCalculator*.

Next, we are going to add icon to our app. This icon will appear on the dashboard of your phone.

[Adding an icon to the app](#)

In this section, we will add an icon image to our app. Icons are very important and your app won't be accepted on the app store unless all the image requirements are satisfied.

Usually, the icons are provided by the designer in your workplace. But if you are making your own app, there is an easy solution. Click on the Assets in the navigation pane. Then, click on the **AppIcon** as shown in [Figure](#)

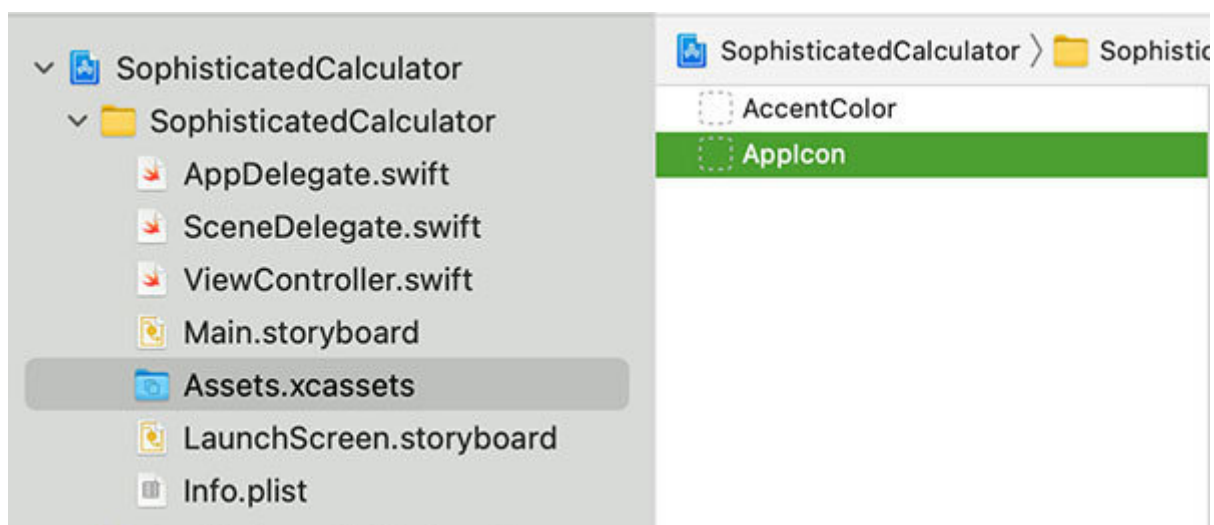


Figure 5.7.1: *Situating the AppIcon*

You should land on a screen like [Figure](#). These are all the empty image boxes with their respective resolutions. In the next few steps, we will fill all of these:

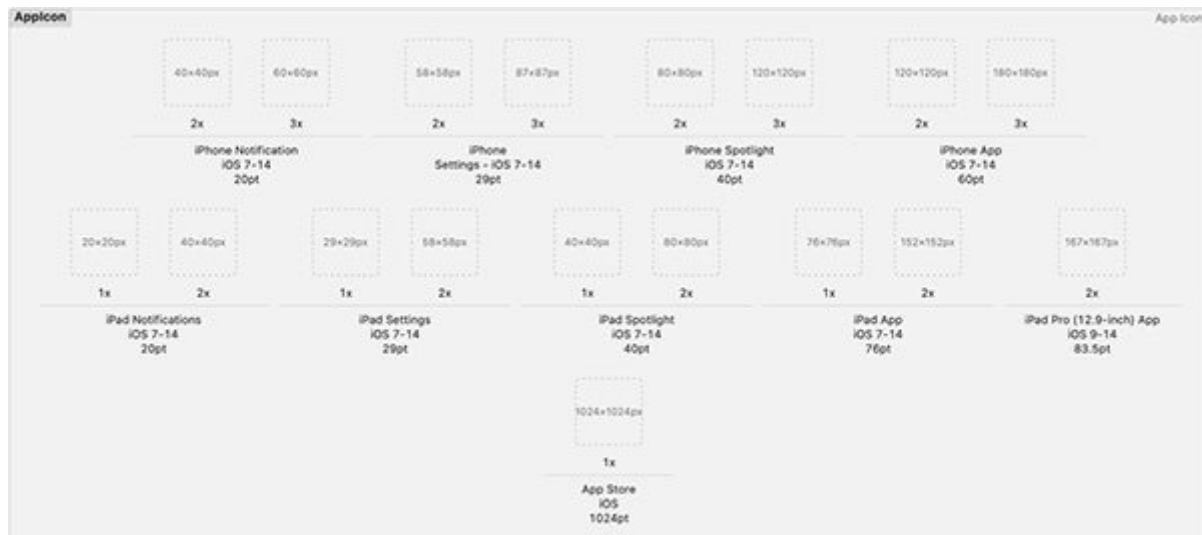


Figure 5.7.2: *AppIcon library*

We must create so many images!! But do not worry, there is an easy solution. All you need is a square icon of 1024x1024 px. We will try a free image from the Internet for demo purposes. But you should try to create your own icon to make it stand out and unique. First, go to Refer to [Figure](#)

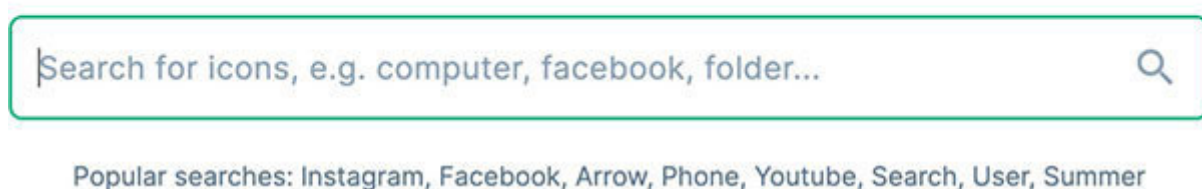


Figure 5.8: *Iconfinder website*

Then, download the free icon which you like. I chose the one which was available in 1024px size. If you do not find the size,

you will first have to resize your icon to 1024. We are not submitting the app to the store, so do not worry a lot about the quality. The main purpose is to understand icons. Refer to [Figure](#)

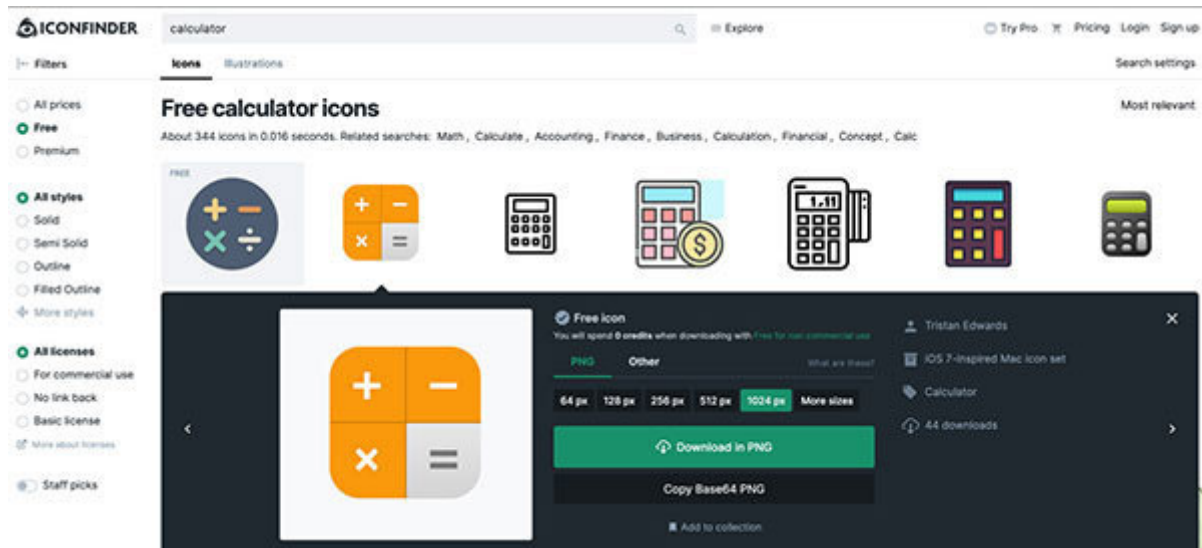


Figure 5.9: Finding the right icon

Next, you go to <https://appicon.co/#app-icon> and drag and drop your image there. Click on **Generate** and it will give you a zip file. Refer to [Figure](#)

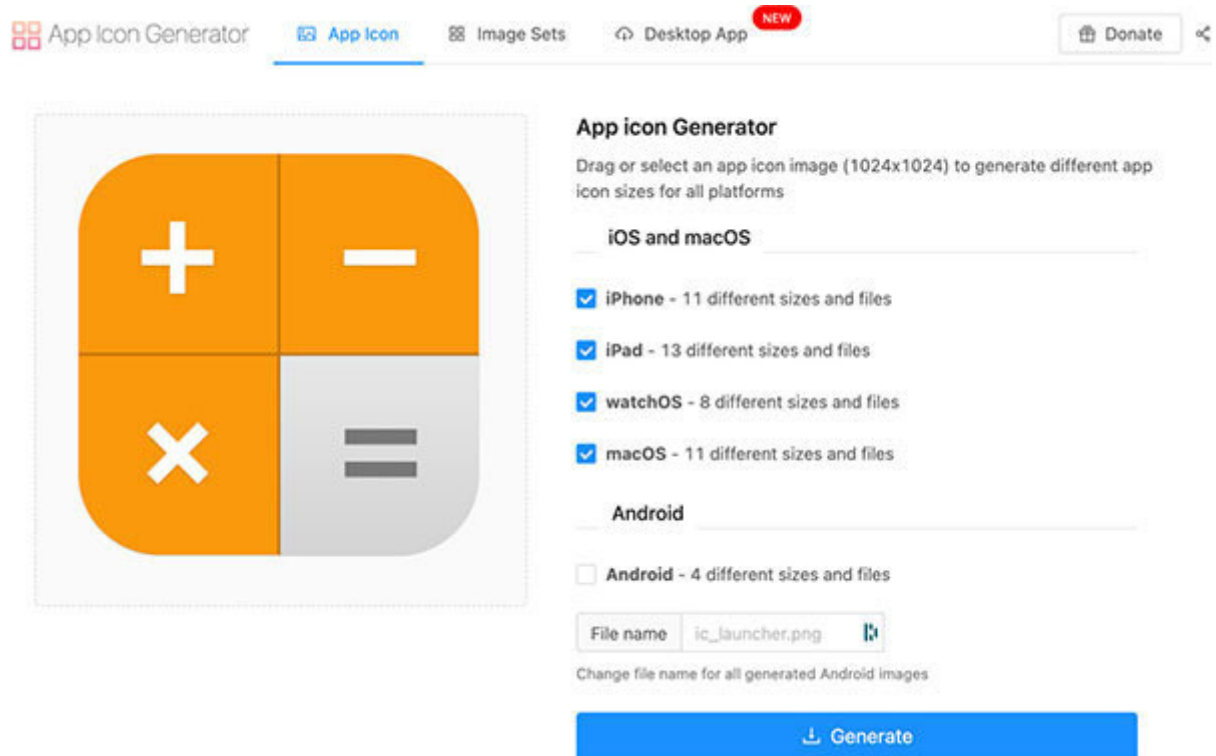


Figure 5.10: *Generating icon set*

Next, delete the existing AppIcon image set in your Assets library and drag and drop the folder from the zip folder. Refer to [Figure](#)

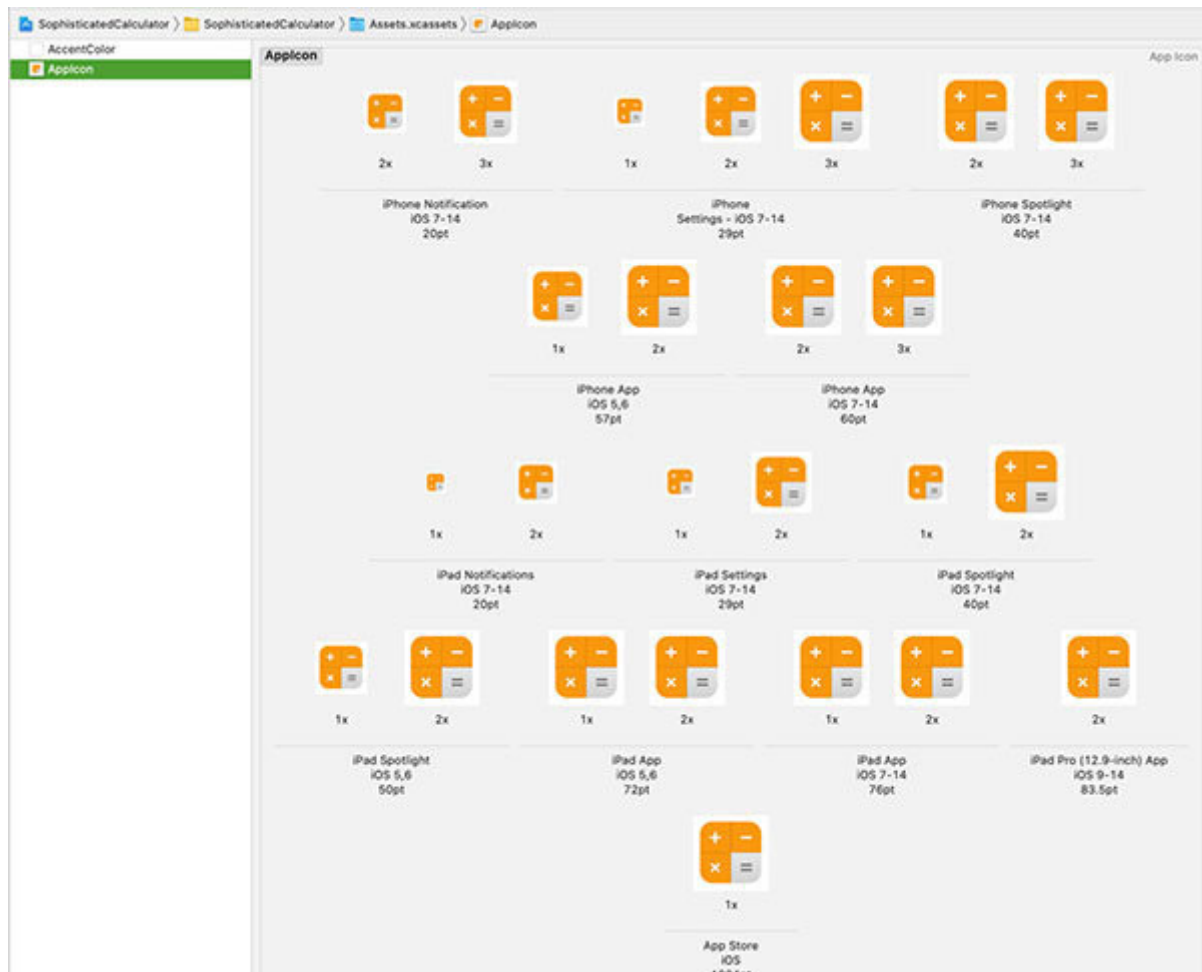


Figure 5.11: *AppIcon library after using all the icons*

You will see a white border around, but don't worry, the icon is only for demo purposes. Run the app and you should see the new icon on your iPhone dashboard. Voilà!! Refer to [Figure](#)

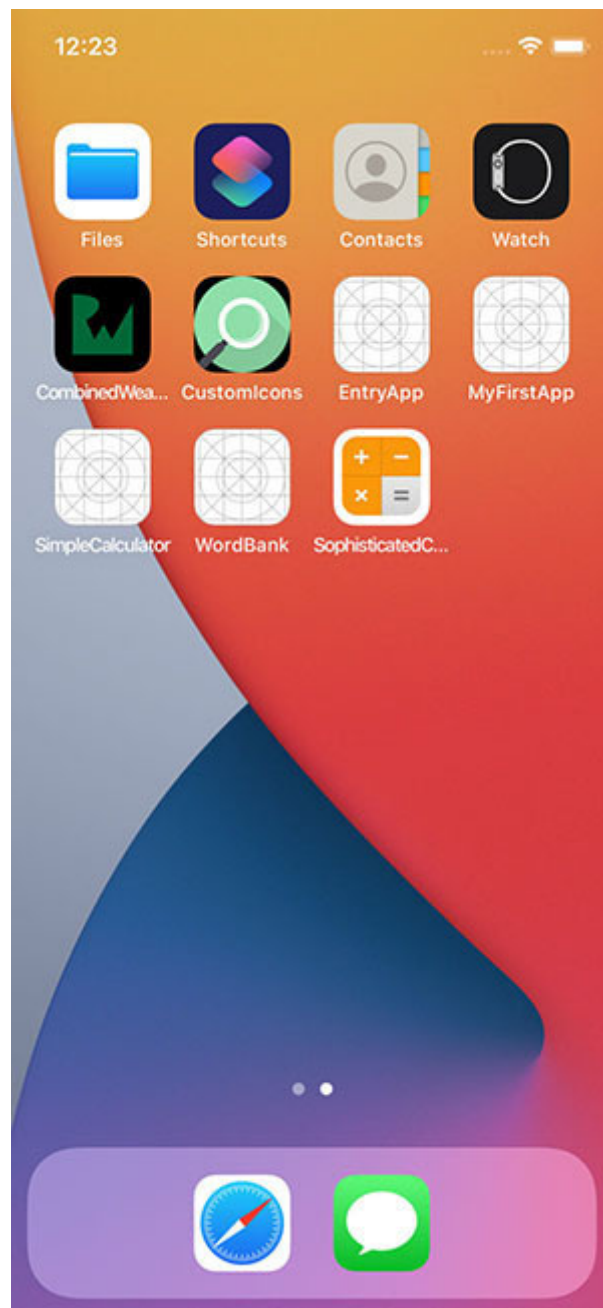


Figure 5.12: Icon as seen on the dashboard

Preparing the layout

We have done the basic setup and you can now change icons for an app. The next step is to design the layout. Let us try to create:

Now, we will try to mimic the layout. Open the storyboard and try to create a layout. I used a mixture of stacks and views. [Figure 5.13](#) shows the hierarchy of my views. This will help to create a layout for yourself:

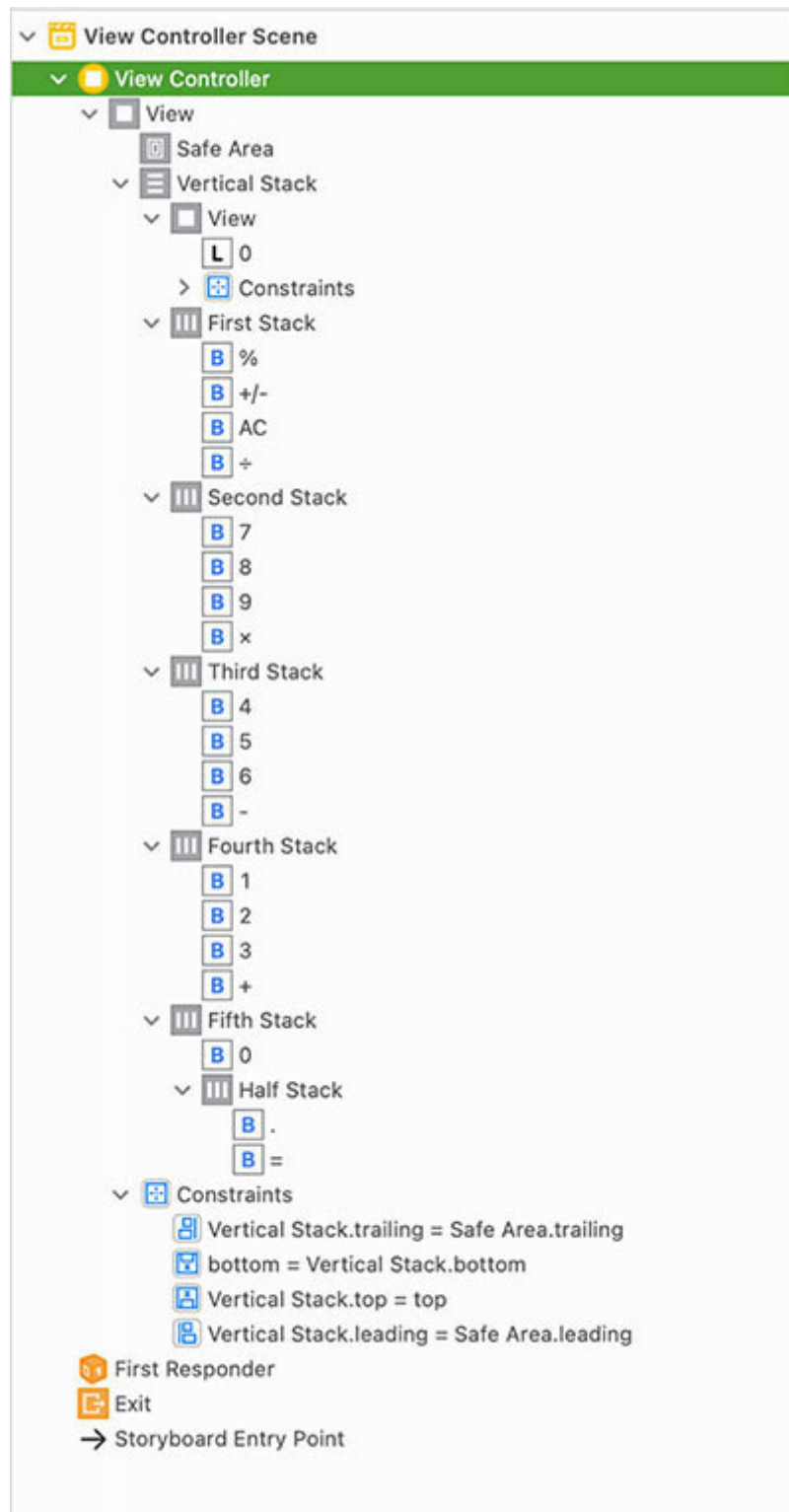


Figure 5.13: View hierarchy of Sophisticated calculator

Use StackView to place buttons one above the other. You can choose a combination of horizontal and vertical stacks if you want.

Use label instead of text field to show the result. This will avoid the keyboard overlap issue.

Assign the necessary constraints. Check whether your app runs fine on different iPhones. A plus would be to make it work on an iPad.

[Making the calculator work](#)

We have covered the UI part. Our app now has a design which resembles a calculator.

But the problem is that our app does not calculate anything!! Let us fix that:

Make a connection to the label. Let us call it result label. Set its text to empty in the

We have so many buttons. Do we create connections for every button? No. We are not going to hide the buttons or change their state. All we are interested in is their click event. But this would also mean creating more than 10 Instead, we can use the switch case and use only one **IBAction** for all the numbers. Create a first **IBAction** and then connect all the numbers to the same

All we have to do on every number click is to append the number to the result label. Your final click action should look as follows. You will see how simplified the code is if you have your logic in place!!

```
@IBAction func numberpadClicked(_ sender: Any) {  
    let button = sender as! UIButton
```

```
if let title = button.titleLabel?.text {  
    resultLabel.text?.append(title)  
}  
}
```

We will write the code for operation. Create a variable to hold the value of the last operator clicked:

```
var operatorClicked = ""
```

Next, you can create an event for all the operations, except the equals. Here, you will only hold the last operator value clicked. Run and check your code from time to time:

```
@IBAction func operationButtonClicked(_ sender: Any) {  
    let button = sender as! UIButton  
    if let title = button.titleLabel?.text {  
        resultLabel.text?.append(title)//show the operator in the result  
        operatorClicked = title  
    }  
}
```

Now, create an IBAction to register the equals click. Our job is to look at the last operator clicked and perform the action on the two targets. The code should look like as follows. You should be able to understand the code:

```
@IBAction func equalsClicked(_ sender: Any) {
```

```

if let title = resultLabel.text { //check if there are any values in the
label
//find the two targets surrounding the operator
let targets = title.components(separatedBy: operatorClicked)

//convert them and then perform the operations
if let value1 = Float(targets[0]), let value2 = Float(targets[1]) {
switch operatorClicked {
case "÷":
resultLabel.text = String(value1/value2)
case "×":
resultLabel.text = String(value1*value2)
case "-":
resultLabel.text = String(value1-value2)
case "+":
resultLabel.text = String(value1+value2)
default:
break
}
}
operatorClicked = "" //clear the operator before leaving the function
}
}

```

The last step is to make sure that there are only two targets at a given time. This means every time you click on an operator check whether there was already an operator clicked and perform the operation first before adding the new operator. You need to modify your **operatorClicked**

```
@IBAction func operationButtonClicked(_ sender: Any) {  
    let button = sender as! UIButton  
  
    if !operatorClicked.isEmpty {  
        //if there is already an operator clicked, find the result first  
        self.equalsClicked(button) //we are not using the sender in the  
        function, so we can send Any button for convinience.  
    }  
    if let title = button.titleLabel?.text {  
        resultLabel.text?.append(title)//show the operator in the result  
  
        operatorClicked = title  
    }  
}
```

This is a very positive use case of the app. You will find many ways to break the code. Find those and fix them!! This would be an exercise for you.

You can even add a layout screen.

You can handle better the decimal points and the type conversions. But this app should give you some confidence in building simple apps and thinking like a developer.

This might not be THE only solution. You will probably find better ways to do this. Don't hold back and make the app the way you want!! The aim of the app is to practice.

Conclusion

In this chapter, we learned about the **optional**. We saw how useful it is and also how we can unwrap values of an optional type. We then learned about **if let** and **guard**. We used them to check the non-nil condition of an optional type. The last topic we touched was that of type aliases. It is a very interesting feature, which can be used to make the code more readable. We did this while creating a simple calculator app. We ended the chapter with an exercise of creating a sophisticated calculator. In the next chapter, we will learn about code structuring. Swift is about to get more interesting. Are you ready?

Key learnings

We started the chapter by building a simple calculator. After the initial setup, we jumped into learning about Optionals.

We then learned about **guard-let** and We looked at the difference between these two and implemented them in our simple calculator.

We then looked at the convenience feature, defer. We saw how it can be used to perform certain operations at a later time, before leaving the scope of the function.

Then, we looked at the different keywords used to transfer the control. We looked at return, break, throw, fallthrough, and continue.

We then looked at Type Aliases, and how we can use them to create our fake types to make the code more readable.

At the end, we looked at the different access levels of the app, namely, open, private, fileprivate, and internal

Last but not the least, we turned our simple calculator into a more sophisticated one with a nice UI.

We practiced a lot of things which we have learned so far while building this app.

Test your knowledge

What is forced unwrapping?

When you retrieve a value from an optional assuming it is not nil. The ! is used for this.

What are the two ways we can use to unwrap optionals?

guard-let

if-let

If I want to stop the execution of the loop and exit it, I should use fallthrough.

No, to exit a loop, one must use break.

How is the switch case in Swift different from the traditional switch case?

You do not need break statement.

You can match patterns.

Open and public are two different access levels.

No, both are the same. All the classes and functions accessible to everyone are public.

Which access level should I use if I do not want to grant access to your entity outside of my source file?

fileprivate

Where do we change the icons for the app?

Assets

Which keyword is used when you want the loop to stop what it is doing and start again from the beginning?

continue

CHAPTER 6

Code Structuring

We will build another app in this chapter. While doing so, we will learn how to keep our code modular and clean. We will start with functions. We will see the different types of functions and all other terms associated with it, followed by *Closures* . Then, we will learn about a very important concept called *Structures* . This will be followed by *Properties* . We will end the chapter by learning about *Subscripts* and *Initializers* while building a navigation-based app.

Structure

In this chapter, we will cover the following topics:

App setup

Functions

Closures

Structures

Properties

Subscripts

Initializers

Navigation

Objectives

The objective of this chapter is to help you understand structuring of code. How you can keep your code clean and reuse chunks of codes with the help of functions. The second objective of the app is to help you understand how navigation works in iOS. We will build a navigation-based app, which will also help you understand how we can pass data between two apps.

App setup

Let us first create an application. We will build a simple form like we did in the first chapter, Building your first application. But this time, we will show the entries on the next screen.

Let us call it *EntryApp*. Follow the steps below to create the app:

Open Xcode and create a new app. Call it EntryApp.

Now, open the storyboard.

Open the library and drag and drop a **Navigation Controller** on the storyboard.

You should see something similar to [Figure](#)



Figure 6.1: *Navigation Controller in storyboard*

Now, delete the **Root View**

Change the position of the arrow which is next to ViewController. Drag and drop it on the left-hand side of the Navigation Controller as shown in [Figure](#)

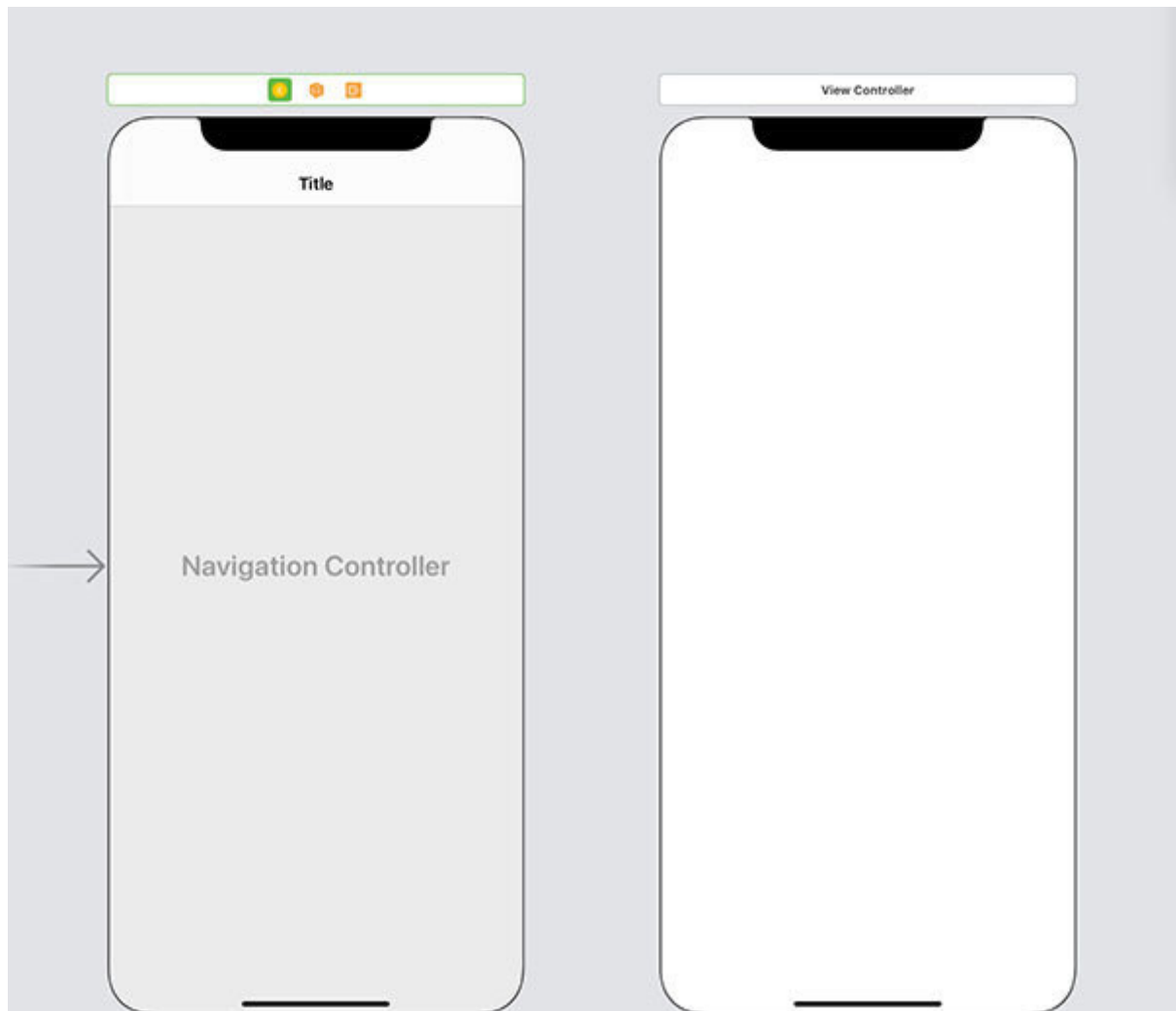


Figure 6.2: *Initial View Controller arrow is now on the UINavigationController*

The other way to do it is to click on the **Navigation Controller** and then in the **Attributes Inspector**, select the option **Initial View**



Simulated Metrics

- Size
- Top Bar
- Bottom Bar
- Appearance

Navigation Controller

- Bar Visibility ☒ Shows Navigation Bar
- ☐ Shows Toolbar
- Hide Bars ☐ On Swipe
- ☐ On Tap
- ☐ When Keyboard Appears
- ☐ When Vertically Compact

View Controller

- Title
- ☒ Is Initial View Controller
-
- Layout ☒ Adjust Scroll View Insets
- ☐ Hide Bottom Bar on Push
- ☒ Resize View From NIB
- ☐ Use Full Screen (Deprecated)
- Extend Edges ☒ Under Top Bars
- ☒ Under Bottom Bars
- ☐ Under Opaque Bars

Figure 6.2.1: Attributes inspector of UINavigationController

Now, make your ViewController as the new root view controller for the navigation controller as shown in [Figure](#)

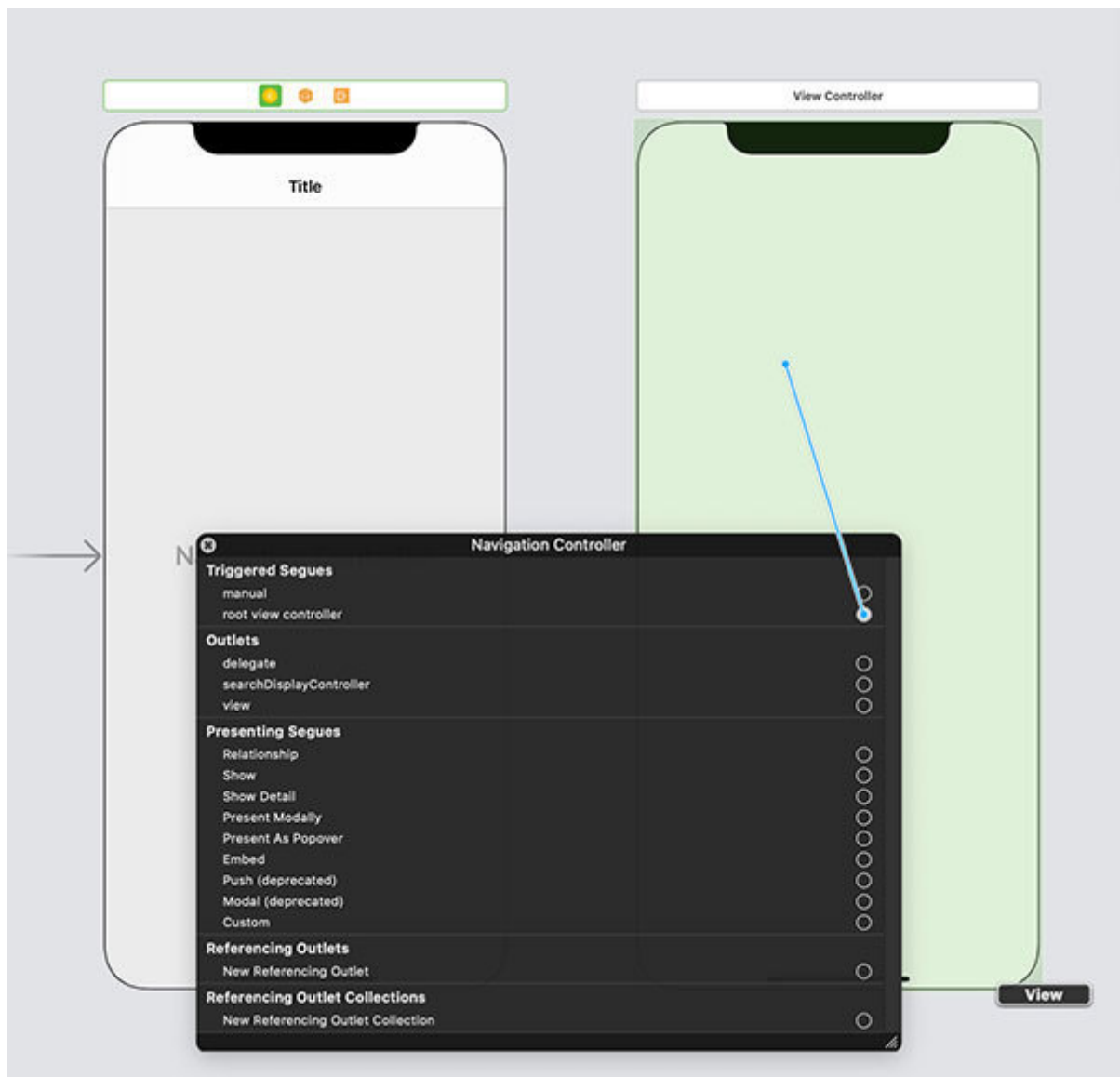


Figure 6.3.1: Assigning ViewController as the root view controller

There are many ways to do the preceding step. You can do this by right clicking on the **Navigation Controller** in the Document Outline and then choosing the **root view controller** option and connecting it with the View Controller. Or you can *Ctrl* + click on **Navigation Controller** and drag it to the **View**. As soon as you release the click, a window pops up. You can then choose the **view** option. See [Figure](#)

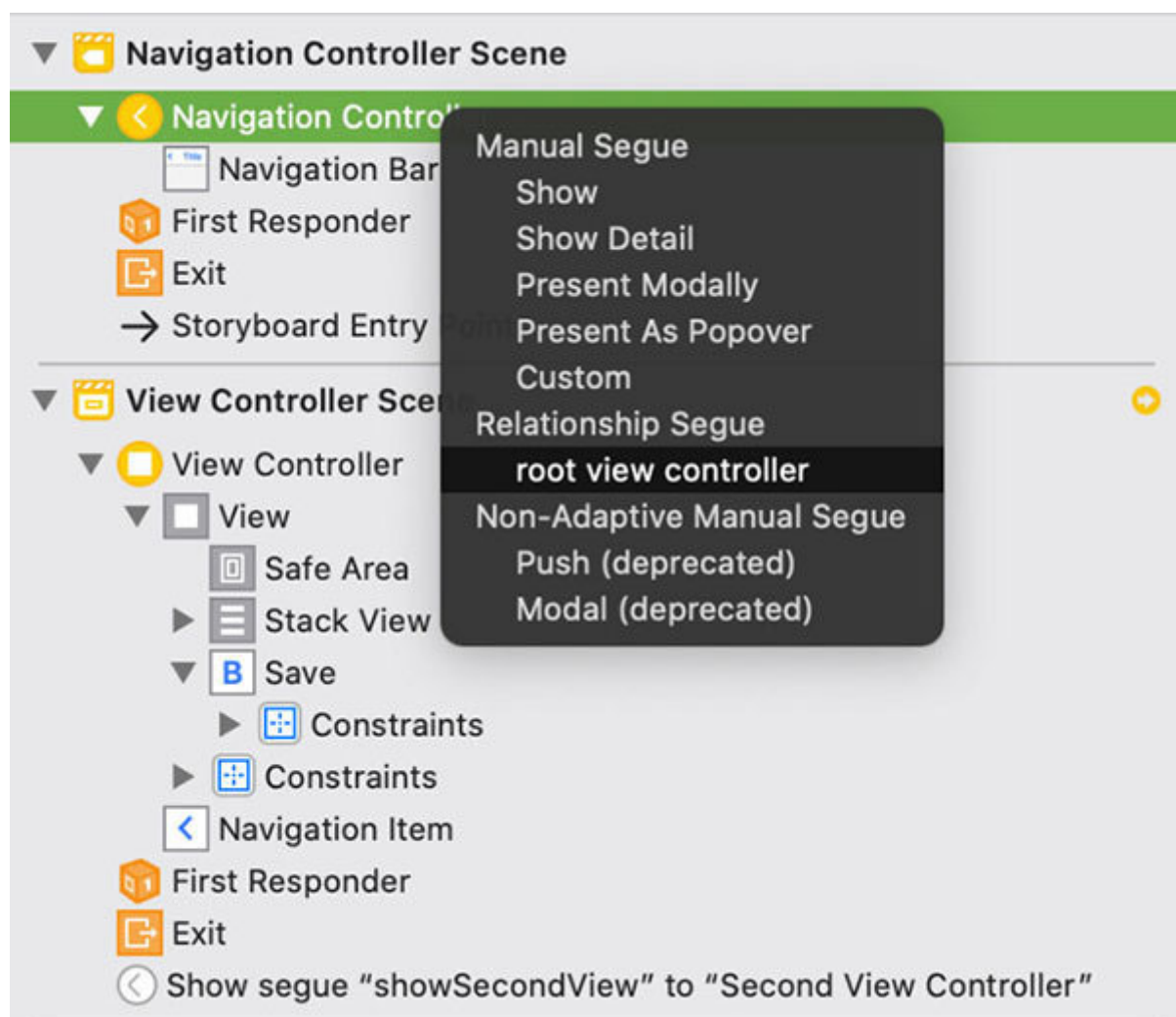


Figure 6.3.2: Choosing root view controller option

Notice the bar on the top. That's a navigation bar. If you add `self.title = "Entry App"` in your `AppDelegate` you should now be able to see the

title on your navigation bar. Or else, you can change the title and the back button title in the storyboard as shown in [Figure](#) Now, run the app and you should see a screen as shown in [Figure](#)

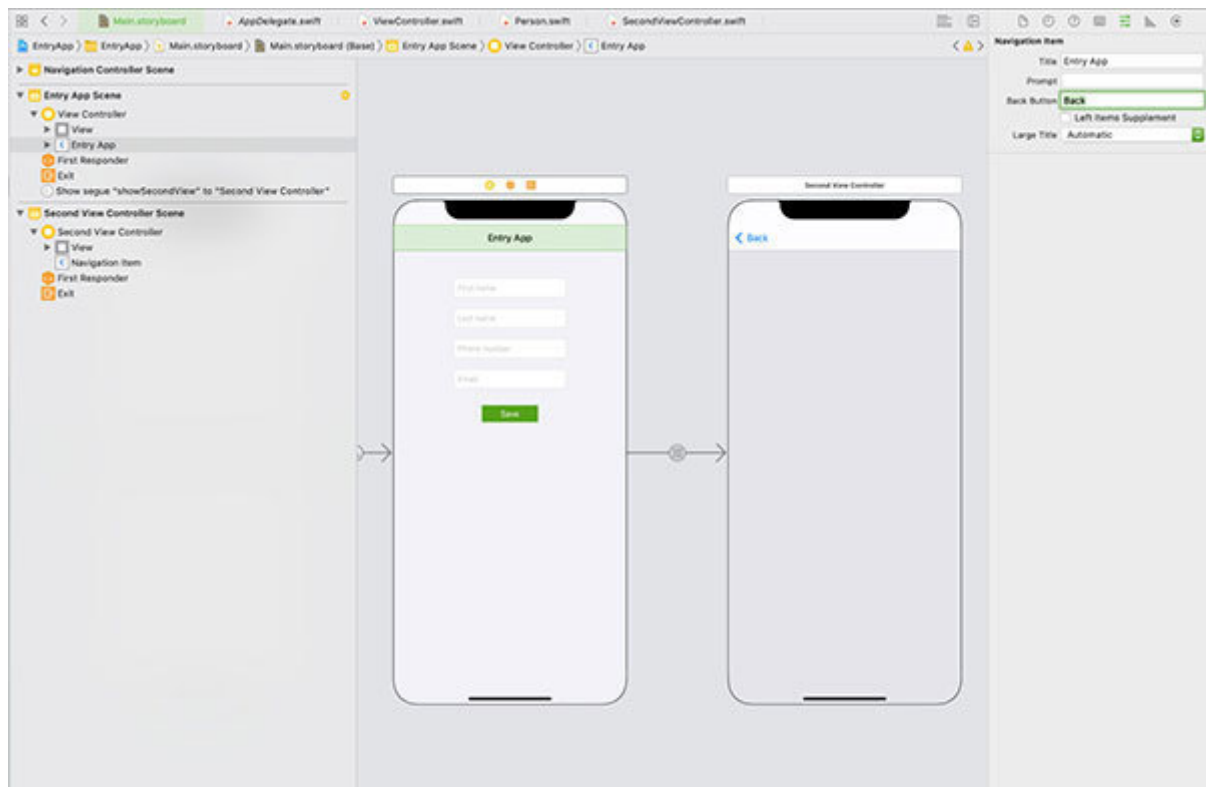


Figure 6.4.1: Changing back button title

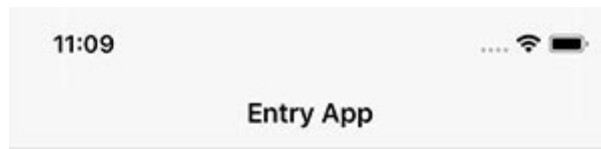
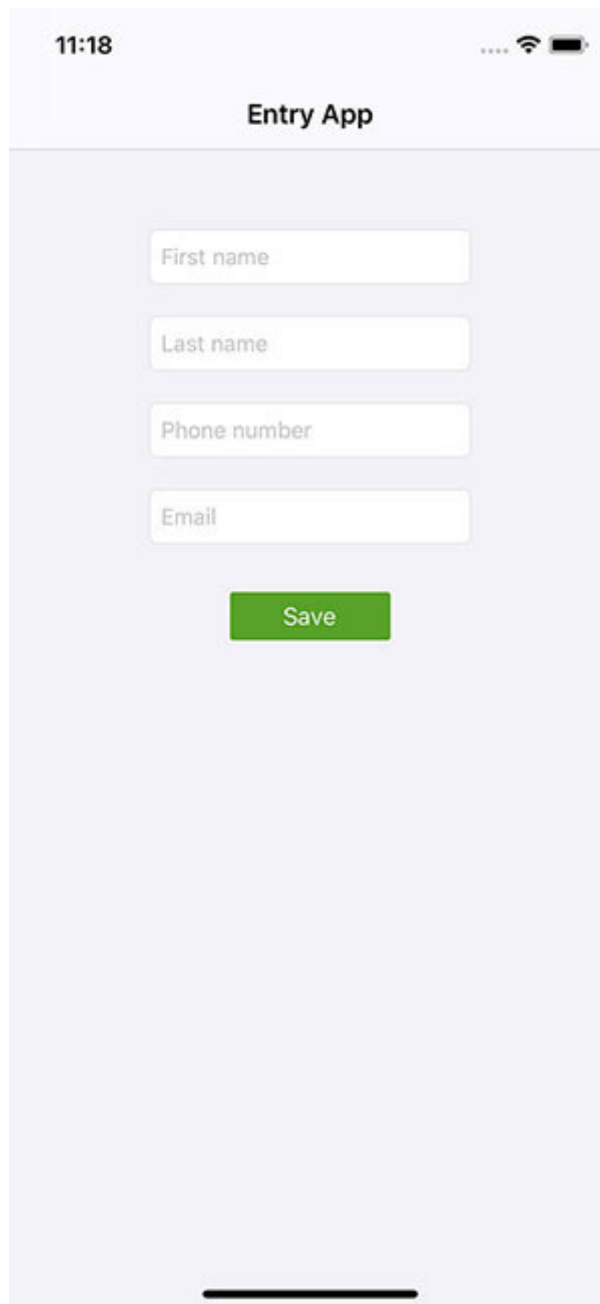


Figure 6.4.2: *First screen with title on the top*

Your ViewController is now a root view controller of a navigation controller, i.e., it is now the starting point of the navigation which will follow. The navigation controller is nothing but a stack of view controllers which follows the principle of **Last In First Out** So

when you want to add another view controller, you need to push and when you have to move back, you have to pop or dismiss.

The last step for you is to add some text fields to the view and a button. Your final result should resemble [Figure](#) I will allow you to add the constraints and create the event for the button.



The image shows a mobile application interface on an iPhone. At the top, the status bar displays the time 11:18, signal strength, Wi-Fi, and battery icons. Below the status bar is a header with the title "Entry App". The main content area contains four text input fields stacked vertically, labeled "First name", "Last name", "Phone number", and "Email". Below these fields is a green button with the text "Save". The interface is clean and minimalist, with a light gray background and white text fields.

Figure 6.5: *Modified screen with text fields and button*

Functions

Let us now take a detour to learn about functions. You have so far seen pieces of code, which begin with { and end with another }, like the ones in [Table](#)

Table 6.1: *Example of functions*

These are nothing but The one on the left-hand side, like the names suggests, runs when the view has finished loading. The one on the right-hand side is when the save button is clicked. So, let us define what a function is; it is a piece or chunk of code which performs a specific task and is contained within its limits, i.e., a self-contained *task-specific* block of code.

Defining a A function is defined by the keyword The name of the function is always prefixed or preceded by this keyword. The inputs of a function are called A function which passes back a value at the end of its execution is said to have a *return* The function name must be a self-explanatory word explaining the task the function will be performing. Following is the general syntax of a Swift function:

```
func name(parameters) -> return type {  
function body
```



```
}
```

The preceding syntax is that of a function with a returning type. When we want the function to return a value, we use the preceding syntax; if not, we just remove the arrow:

```
func name(parameters) {  
    function body  
}
```

Calling a When we want to use the function, we call it. The inputs we give to the function are called Let us define a new function called This is how it will look like:

```
func setDefaultValues () {  
    self.firstNameTextField.text = "John"  
    self.lastNameTextField.text = "Cena"  
    self.phoneNumberTextField.text = "+1 23 456 78"  
    self.emailTextField.text = "jc@gmail.com"  
}
```

After looking at the preceding function, you can now tell that it is a function which accepts no arguments and has no return type, also called a **void type**. Remember, a parameter is when you define the function and an argument is when you call the function. So, the preceding function has no parameters and hence when we call it, we will pass no arguments.

To increase the readability, you can add an argument label to the parameter name. Argument labels are visible when we call the function. Here is an example:

```
func drawRectangle (withHeight height: Int, andWidth width:Int) {  
  print(height + " " + width)  
}
```

And this is how it looks when we call the function:

```
drawRectangle(withHeight: 30, andWidth: 50)
```

We can add labels to the parameters, which make the arguments more readable. You will notice that the preceding function is much more readable when we use the label `withHeight` along with the parameter name `height`. If you want to omit the argument label, use an underscore (`_`) instead:

```
func drawRectangle (_ height: Int, _ width:Int) {  
  print(height + " " + width)  
}
```

Function parameters and return Now that you know how to write a function, try writing a function for the following statement; write a function which adds to the input name and returns the new value:

```
func addHello (name: String) -> String {  
  return"Hello "+ name
```

```
}
```

Now, modify the function to accept the first name and last name. Multiple parameters are separated by a comma:

```
func addHello (firstName: String = "James", lastName: String) ->
String {
return"Hello "+firstName+" "+lastName
}
```

You must have noticed the in the parameters. This is the **default** If the **firstName** is not passed, James would be the default value for the When you do this and try to call the function, you will see that autocomplete gives you two options as seen in [Figure](#)

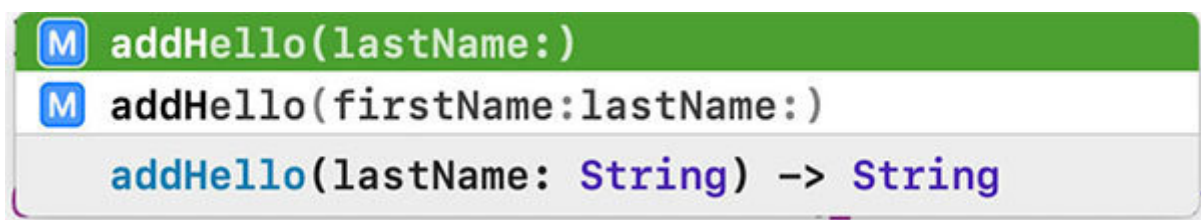


Figure 6.6: Functions with default parameters

Similarly, we can have comma-separated multiple return values:

```
func greet (firstName: String = "James", secondName: String) ->
(firstGreeting: String, secondGreeting: String) {
return ("Hello "+firstName, "Hello "+secondName)
}
```

This is how you will call the preceding function:

```
let greeting = greet(firstName: "Johnny", secondName: "Jinny")
print(greeting.firstGreeting)
print(greeting.secondGreeting)
```

Does this remind you of something? Exactly, we are returning a tuple. You can even have a return value without the labels, **(String)**, but with labels, it is much easier to access the values in the tuple. You can even make the return type optional like **(String?)**, but then you will have to use the optional rules for unwrapping the value, like we have seen in the previous chapter.

Function as You can define a variable as a function and pass it as an argument in another function. You can even have a function as a return value. Let us define two functions:

```
func greet(_ firstName:String, _ lastName:String) ->String {
return "Hello "+firstName+" "+lastName
}
```

```
func sayBye(_ firstName:String, _ lastName:String) ->String {
return "Bye "+firstName+" "+lastName
}
```

The parameters and the return values together form the function type. Here, **(String, String) ->String** is a new type. We can use this type to create variables as follows:

```
let greetFunc: (String, String) ->String = greet(_:_:)
```

The preceding code defines a **greetFunc** with the type **(String, String)** It even initializes it with a default value Let us write a third function which prints any type of messages it receives:

```
func printMessage(_ a: String, _ b: String,_ messageFunction:
(String, String) ->String) {
print("Result: \"(messageFunction(a, b))\"")
}
```

The preceding function takes in three parameters such as a function **(String, String) ->String** and two Strings. We can then use our function variables to print the messages as follows:

```
var greetFunc: (String, String) ->String = greet(_:_:)
printMessage("Tom", "Foyer", greetFunc) //Result: Hello Tom Foyer
greetFunc = sayBye(_:_:)
printMessage("Lilly", "Bon", greetFunc) //Result: Bye Lilly Bon
```

The function variable was changed to now store the **sayBye** function. As you can see, this can be very useful. You can create functions which have similar parameters and return types and then pass them as arguments to other functions.

Nested So far, all the functions we have seen were in a global scope, i.e., in the class where we were writing it. Hence, for example, a function was visible or could be called from view did load. Nested functions are functions written within the scope of a

function. Such functions are not visible outside the scope of the function. Following is an example of a nested function:

```
func greet(_ firstName:String, _ lastName:String) ->String {  
  
    func addHello() ->String {  
        return "Hello "+firstName  
    }  
    return addHello() +" "+lastName  
}
```

In the preceding function, we have declared a new function **addHello** within the scope of the function **greet**. Thus, it will not be visible and accessible to the code outside. But we can call it from within the **greet** function.

Methods

When a function is associated with a particular type, it is called a method. A method has exactly the same syntax as that of a function. A method is always associated with a class, structure, or an enumeration. A method associated with an instance of a class, structure, or an enumeration is called an **instance method**. (We will learn about the meaning of the word instance, when we learn about structures later in this chapter.)

On the other hand, the method which we can call on the type itself is called a **type method**. We need to add the word static before a type method. We can turn the function into a type method by adding the word static before it. This is how we call it:

```
ViewController.greet("Lilly", "Bon")
```

Closures

Closures are blocks of codes, which can be passed around. They are like *blocks* if you are familiar with Objective C or lambda expression in C#. They have three forms: Global function, Nested function, and Closure expression.

Global All the functions we saw were in fact a form of closure. A simple global function does not *capture* the values from its environment. It only uses the values passed to it as arguments.

Nested A nested function is nothing but a closure with names and parameters. If you look at our example of a nested function, you will understand Simply put, the **addHello** function can use all the values and references from the outer greet function. Thus, **addHello** *captures* the values from the greet function.

Closure expression

Having a nested function can be very useful, but you will see that we have to a function with parameters and names. Closure expressions simplify this. These are simple in-line expressions which are task focused like the lambda expressions in other languages.

Here is the general syntax of a closure:

```
{(parameters) ->return type in  
statements  
}
```

The **in** keyword is very important. It tells Swift where the declaration part of the closure ends. The most common example of a closure is the sort function on an array:

```
var names: [String]  
names.sorted {(String, String) -> Bool in  
//sorting code goes here  
}
```

You should be able to read the preceding closure easily. It says that the sort function will take a closure with two strings as the parameters and return a bool. You can even rename the

parameters like you do in your functions to make the code more readable:

```
names = names?.sorted(by: {(s1: String, s2: String) ->Bool in  
s1 > s2 //In Swift you can avoid writing the return keyword  
}))
```

Let us learn how to write a closure expression. Let us look at the function we created which accepts a function as a parameter:

```
func printMessage(_ a: String, _ b: String, _ messageFunction:  
(String, String) ->String) {  
    print("Result: \(messageFunction(a, b))")  
}
```

We passed our greet function to the preceding function as a parameter:

```
func greet(_ firstName:String, _ lastName:String) ->String {  
    return "Hello " + firstName + " " + lastName  
}  
printMessage("Tom", "Foyer", greet) //Result: Hello Tom Foyer
```

Now in the preceding code, we can change the argument of greet to that of a closure. So, how do we write a closure?

First, you will remove the **func** keyword. Then, you will remove the name of the function:

```
(_ firstName: String,_ lastName: String) -> String {  
return "Hello " + firstName + " " + lastName  
}
```

Finally, you will remove the open curly brace and place it before the parentheses. Replace the curly brace with the **in** keyword. You should get the following:

```
{(_ firstName:String,_ lastName:String) -> String in  
return "Hello " + firstName + " " + lastName  
}
```

This is how a closure looks. Now, you can pass this closure as an argument to the function

```
printMessage("Tom", "Foyer", {(_ firstName:String,_  
lastName:String) -> String in  
return "Hello " + firstName + " " + lastName  
}))
```

But thanks to the type inference of Swift, we can further simplify and optimize the preceding closure. The compiler sees that the two input types are strings, so we can remove the type in the closure. The compiler will infer the type based on the first two parameters. The same goes for the return type:

```
printMessage("Tom", "Foyer", {(firstName,lastName) in  
return "Hello " + firstName + " " + lastName
```

```
})
```

We can further optimize it by removing the **return** keyword:

```
printMessage("Tom", "Foyer", {(firstName,lastName) in "Hello " +  
firstName + " " + lastName})
```

If you think this is the simplest form, you are wrong. We can simplify this further. We can use the anonymous parameters. \$0 and \$1 are the first and the second parameters. So, we can instead use that. Here is how the simplified code looks like:

```
printMessage("Tom", "Foyer", {"Hello " + $0 + " " + $1})
```

In Swift, when the last parameter is a closure, you can simply close the parentheses before it and allow the closure to trail at the end. This is called as a trailing closure. Here is how the final simplified version will look like:

```
printMessage("Tom", "Foyer") {"Hello " + $0 + " " + $1}
```

You will notice that the closures simplify the code a lot. But on the downside, it suffers readability. For someone who is new to Swift, the preceding line will not make sense. But if you are pro in Swift, you can quickly understand what is happening here. Following is another real-world example of use of closures. Imagine the following function:

```
func add(n: Int) -> Int {  
    return n + 2
```

}

Imagine we have an array **[3, 5, 7, 9]**, and we need to multiply every element of this array by 2. Instead of going the traditional way, you can transfer the preceding function into a closure and then pass it to the advanced function of an array called `map` I will allow you to work on this one. Here is how the final solution will look like:

```
let arr = [3, 5, 7, 9, 11]
print(arr.map{$0 + 2})
```

You can see that closures simplify the code a lot.

Structures

Let us get back to our code. We will make some modifications to our code. Then we will see how we need Structures. Follow the steps below:

We need to add another view controller to display our saved data.

In the navigation pane, right click on the **EntryApp** folder and then create a new swift file. Call it

An empty file will be created. Add the following code to it:

```
class SecondViewController: UIViewController {  
}
```

Replace the import from Foundation to UIKit.

You just defined this class and called it

The next step is to create its associated view in the storyboard.

Go to the storyboard and add a view controller from the library.

Now, click on this new view controller and in the Identity Inspector, change its class to Refer to [Figure](#)

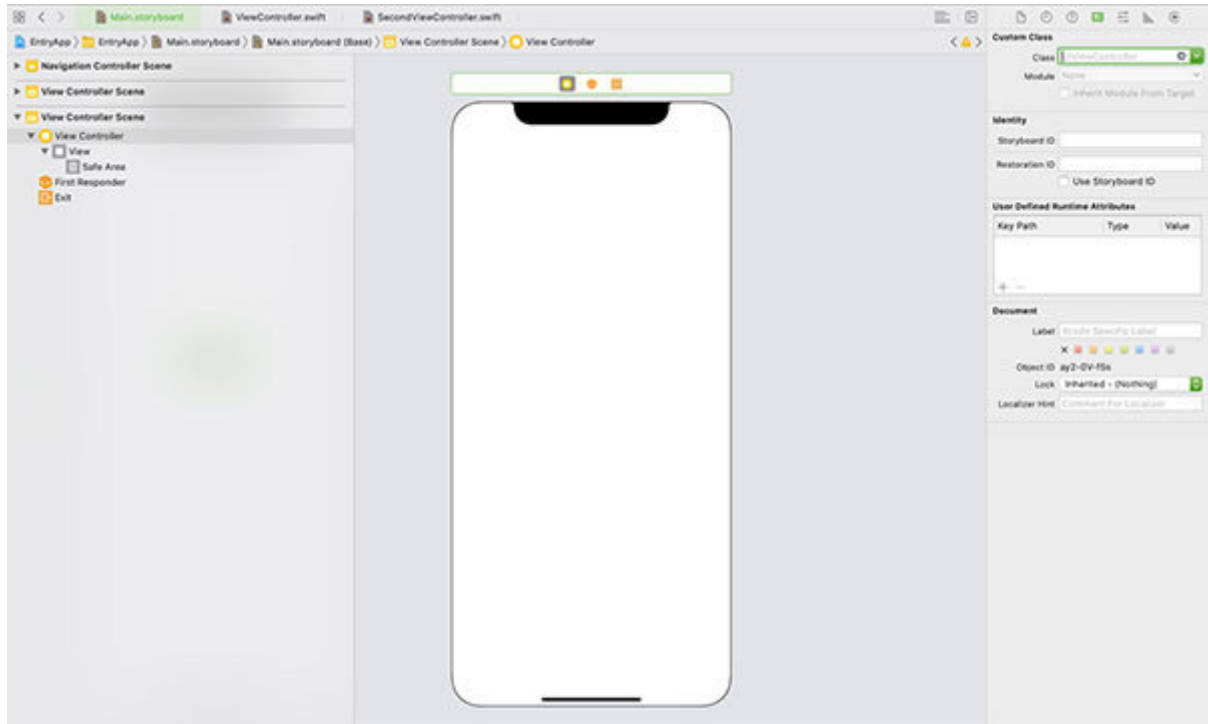


Figure 6.7: Changing the class in Identity inspector

In the same identity inspector, add **SecondViewController** in the **Storyboard ID** field:

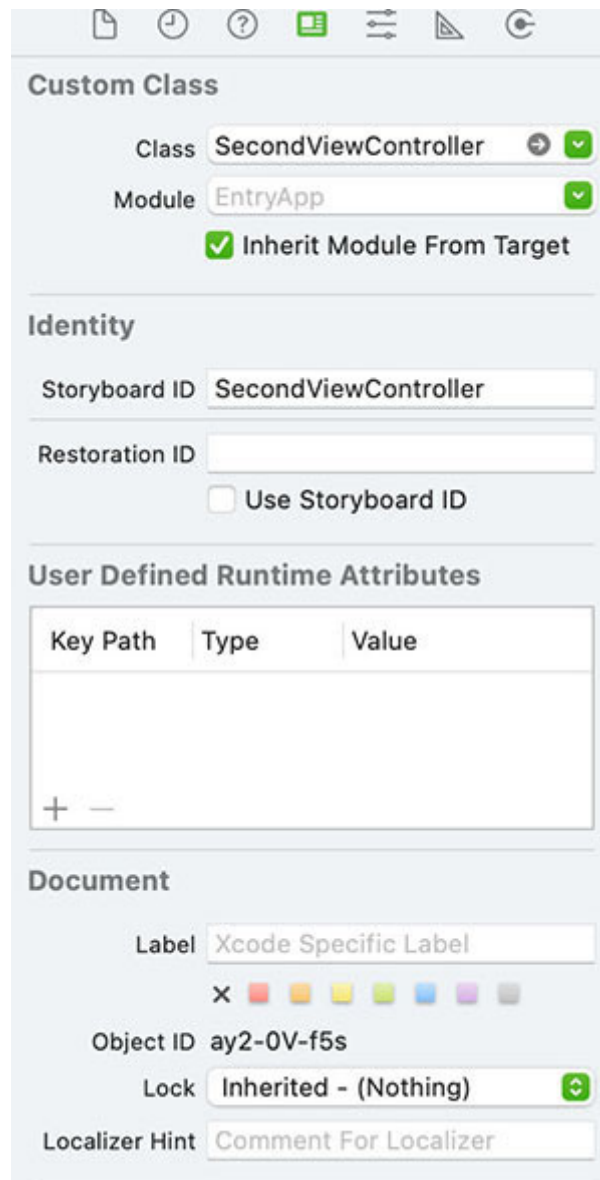


Figure 6.8: Identity inspector post changes

Now, there are many ways to navigate from **ViewController** to the But we will start with the traditional way. Add the following two lines in your **saveButtonClicked** IBAction:

```
@IBAction func saveButtonClicked(_ sender: UIButton) {
let secondViewController =
self.storyboard?.instantiateViewController(withIdentifier:
```



```
"SecondViewController") as! SecondViewController
self.navigationController?.pushViewController(secondViewController,
animated: true)
}
```

The two lines are self-explanatory. We will first create a variable with the second view controller. We will create its instance or an object with the help of the StoryboardID we previously assigned.

We will choose the navigation controller in which our ViewController is added with We will then push the **secondViewController** on this navigation controller stack.

Run the app and click on the **Save** button. You should see the magic happening.

You can try to change the back button title and also assign a title to the second view controller.

Last but not least, create 4 labels in the second view controller to show the data entered in the first view.

The next step for us is to send the data from the first view to the second view. But we have four text fields. Are we going to send them one by one? What if there was more than one field? Wouldn't it be easy if we could send all the data in just one variable to the second view controller? This is when we use

Structures are blocks of code which can be used to build your program. They are defined like any other constant or variable. Like other types, structures are value types. Following is their general syntax:

```
struct name {  
  fields  
}
```

Let us create a new struct for our app called Create a new swift file in the **EntryApp** and define a **struct** called **Person** in it. Then, add the fields we want. Your struct should look something like this:

```
struct Person {  
  var firstName: String?  
  var lastName: String?  
  var phoneNumber: String?  
  var email: String?  
}
```

In the before we navigate, you can create an instance of a Person and assign the values to it. You will see that Swift will automatically create an initializer for your struct:

```
let person = Person(firstName: firstNameTextField.text, lastName:  
lastNameTextField.text, phoneNumber: phoneNumberTextField.text,  
email: emailTextField.text)
```

Our next job is to send this person to the second view and show their details.

Properties

Before we send our Person to the second view, let us understand the code we have written for the struct:

```
struct Person {  
    var firstName: String?  
    var lastName: String?  
    var phoneNumber: String?  
    var email: String?  
}
```

Look at the preceding code again. Now, you can easily read and say that these are variable declarations. But when we have values associated with a class or a structure, they are called **properties**. There are two types of properties: **stored properties** and **computed**. We can associate properties with structures, classes, and enumerations. But we can associate stored properties with only structures and classes. We will learn more about classes and enumerations in the next chapter.

Stored Like the name suggests, they are variables or constants associated with an instance of a type, which can store values. In the preceding code, when we create an instance of Person, its properties would be the stored properties. As seen for variables, you can define a property with a default value. The rules remain the same.

Tip: Though in our `Person` struct, the properties are variable-stored properties if we declare an instance of `Person` with `let`, i.e., a constant, then all its properties become constants.

Computed Unlike stored properties, computed properties provide values indirectly with the help of getters and setters. You must be already familiar with these terms:

```
var fullName: String {  
  get {  
    return (firstName ?? "") + (lastName ?? "")  
  }  
  set {  
    let names = newValue.split(separator: " ")  
    firstName = String(names[0])  
    lastName = String(names[1])  
  }  
}
```

In the preceding code, we added a computed property **fullName** to our structure. When we try to access the value in the full name, the **getter** is called and when we try to change the value of the full name, the **setter** is called. What if we remove the setter? Such a property is called **read-only computed**. If it is a read-only property, should the **var** be changed to `No`, because the getter value keeps changing. Hence, it will still be a variable computed property.

Swift allows us to add observers to our properties that are stored and computed. Observers help us monitor the different stages, just before the new value is set to **willSet** and right after the new value is set to **didSet**. This can be very useful to trigger some other code or perform validation over the new value.

Tip: You can only add observers to the computed properties you inherit, not the ones you define. For example, we cannot add the observers in the `fullName` computed property, as we defined it. But if the property was inherited from a parent class, we would be able to do it.

Subscripts

Remember how we accessed members of arrays and dictionaries?

```
mutableDictionary["key3"] = "apple" //update an existing member  
mutableArray[3] //accessing value at a specific index
```

These are called Subscripts are shortcuts to access members of your type. You can use **subscripting** to define shortcuts to set and retrieve values by the index. Subscripting helps us to replace methods. In the preceding example, instead of retrieving the element at the third index with the help of a method, we used square brackets and the index. This simplifies the code and improves the readability. Structures, classes, and enumerations can define subscripts. The definition of a subscript is preceded by its keyword subscript:

```
struct Name {  
  var name: String = ""  
  subscript(row: Int) ->String {  
    get {  
      if row == 0 {  
        return name  
      }  
      else {  
        return "Does not exist"  
      }  
    }  
  }  
}
```

```
}  
set(newValue) {  
  if row == 0 {  
    name = newValue
```

```
  
}  
}  
}  
}
```

```
var name = Name()  
name[0] = "Johnny"  
print(name[0]) //Johnny  
print(name[1]) //Does not exist
```

Even though you can find your own use case, subscripts are usually used with collections, sequences, or lists. You can even use multiple parameter subscripts. If you remove the set part, it becomes a read-only subscript.

Initializers

If you read the declaration of the person instance, and then go back to our Structure, you will notice that we have not written any method which accepts these parameters. So, what is this method? This method is called Initializers are methods used to create an instance of a particular type. Following is the general syntax of an initializer:

```
init(parameters) {  
  statements  
}
```

If you do not have any parameter, the initializer looks something like this:

```
init() {  
}
```

This is the **default initializer** and every **struct** and **class** has it by default. In our case, we will create a new instance of Person as **let person = Person()**.

Default stored values

Initializers are used to assign default values to your stored properties. But nonetheless, it is recommended that you assign the default value right at the declaration to make the code clean and type inference easy.

Customized initialization

Mostly initializers are customized to create different instances of a type, with different values. For example, if we want to create multiple instances of `Person`, with a different first name and last name, we can write a custom initializer as follows:

```
init(firstName: String, lastName: String) {  
    self.firstName = firstName  
    self.lastName = lastName  
}
```

If we do not write any **init** function, we still have a default **init** with parameters as follows:

```
let person = Person(firstName: firstNameTextField.text, lastName:  
lastNameTextField.text, phoneNumber: phoneNumberTextField.text,  
email: emailTextField.text)
```

Why? This is because of something called **memberwise**. When we write a struct and declare its properties, it gets a default initializer as written above. If you write your own initializer, you can no longer use the default memberwise initializer.

Failable We have already seen optionals. Just like optionals, we can have initializers which can return a nil value. These are called

failable For this, you just need to add? at the end of the init.
Here is an example of a failable initializer:

```
init? (firstName: String) {  
  if firstName.isEmpty {  
    return nil  
  }  
  self.firstName = firstName  
  
}
```

Initializer is a Remember after all, initializer is a function without the keyword **func** before it. So, all the rules we saw for argument labels, parameter names, and so on apply to an initializer.

Deinitializer

Like how there is an initializer for a class, we also have a You can have deinitializers only in classes. Each class can have only one You cannot call a **deinit** function. Swift automatically does that. Deinit is called by Swift before the instance of your class is deallocated. We will see more about memory management later in the book. If you want to clean up your own resources before the instance of your class is deallocated, you can do that in Following is its general syntax:

```
deinit{  
statements  
}
```

As you can see, the **deinit** function does not take any parameters and is written without the usual ().

Navigation

Navigation is when you move from one view to the other. As seen in the preceding code, we will see how to navigate using the following code:

```
@IBAction func saveButtonClicked(_ sender: UIButton) {  
  
    let secondViewController =  
        self.storyboard?.instantiateViewController(withIdentifier:  
            "SecondViewController") as! SecondViewController  
    self.navigationController?.pushViewController(secondViewController,  
        animated: true)  
}
```

But navigation can be achieved using multiple techniques. The first one is using storyboards. In storyboards, we will use connections called *segues* to define the flow of the app. A segue can be initiated between two ViewControllers. The starting point of a segue is a button, a table row, or a touch.

Navigation using storyboards:

We will now try to achieve the navigation without writing any code. Follow the steps below:

Open the storyboard of the EntryApp.

Right click on the **Save** button. You will see a window like the one in [Figure](#)

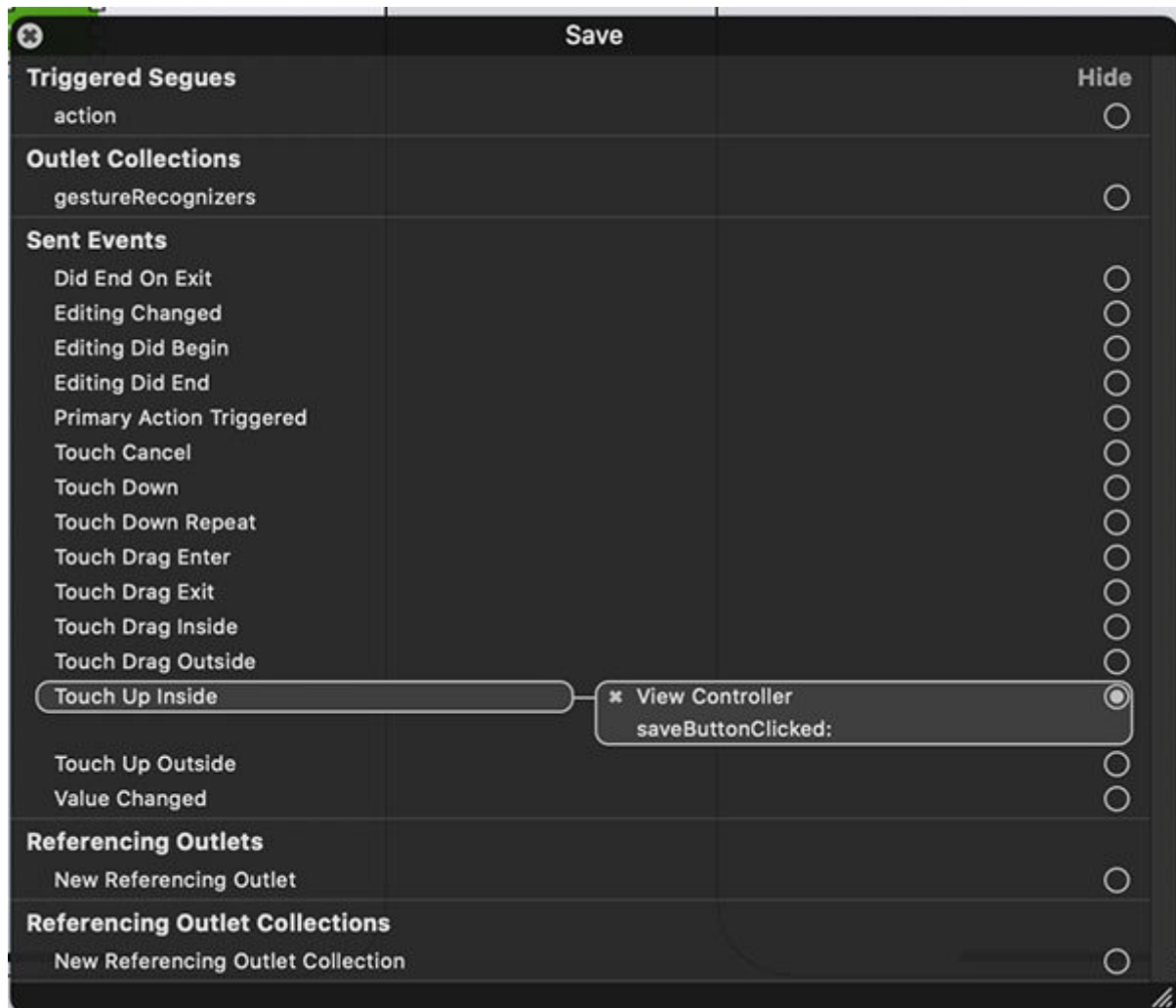


Figure 6.9: Right click on Save button

Now, remove the **saveButtonClicked** connection. If you click on the **Save** button, nothing will happen. You can run the app and verify this.

Next, drag the under Triggered Segues and drop it in our Second View Controller. You will now see a pop up like the one in [Figure](#)

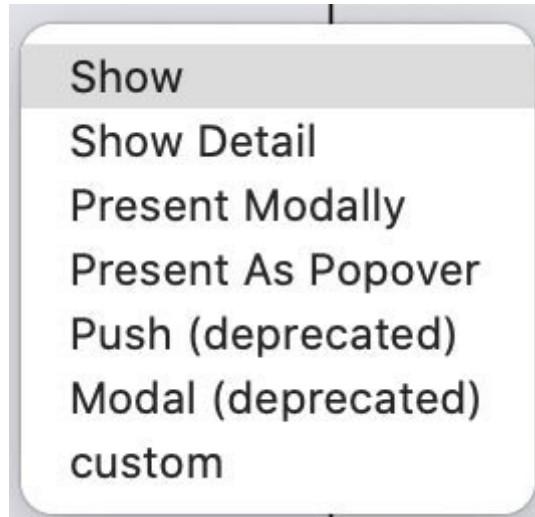


Figure 6.10: *Different ways of navigation*

These are all the ways in which you can present the pop up. We will select the first one As soon as you do this, you will see a **Back** button appear on the top of your Second View Controller. This indicates that the view has been added in the navigation stack. Refer to [Figure](#)

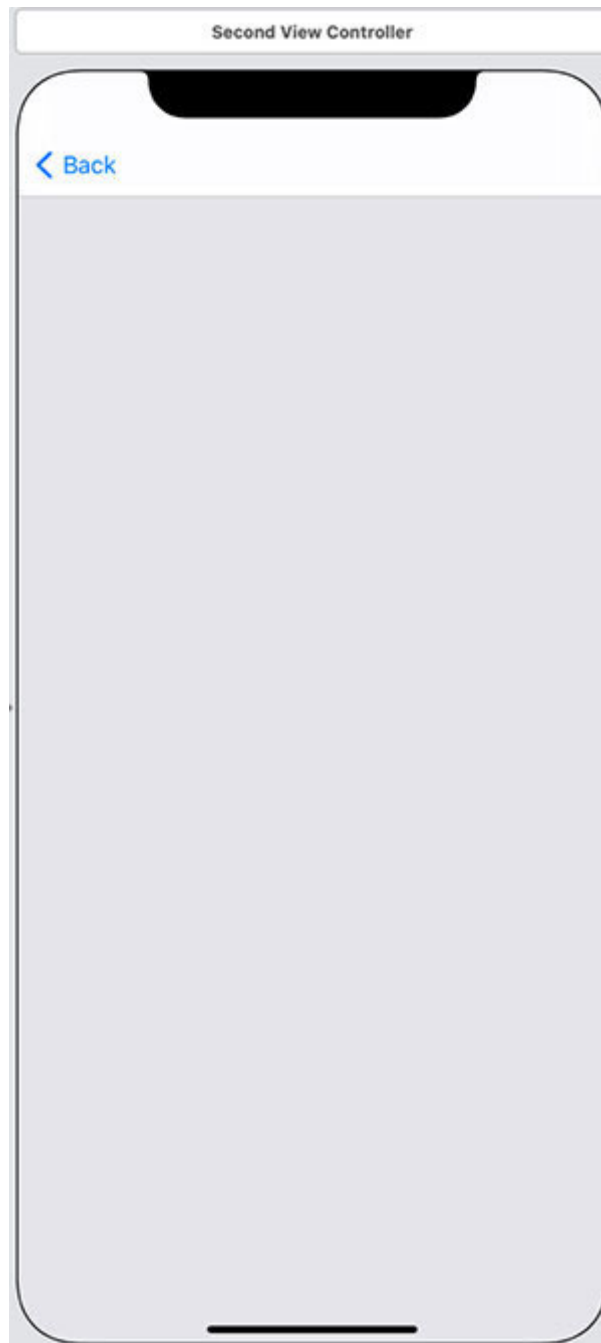


Figure 6.11: *Back button added automatically*

Run the application to check whether your navigation works properly.

We have thus established a segue between our two View Controllers.

Now sometimes, we need to navigate to a page from different points and not just one button click. At such times, it would be easier to have one connection, which can then be called to make the navigation happen.

Delete the action segue between the **Save** button and the Second View Controller. Reestablish the **saveButtonClick** connection in the storyboard.

Now, right click on the ViewController in the Navigation Pane. Again, you will see a pop up, similar to [Figure](#)

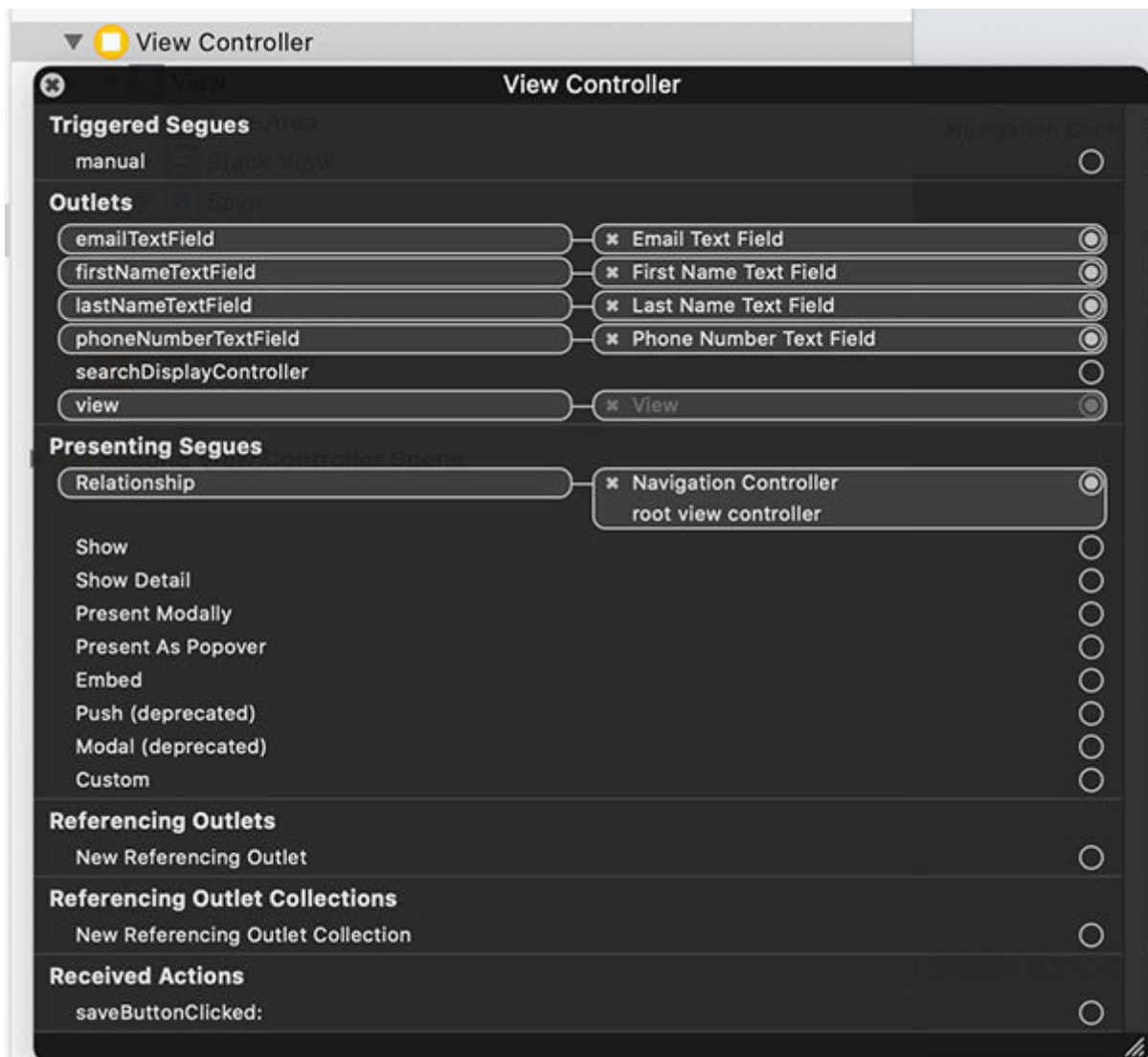


Figure 6.12: Right click on ViewController

This time select the **manual** segue and drag and drop it on the Second View Controller. You will see the back button.

But this segue is manual, i.e., it needs to be called from our code. For this, we will have to name it. Click on the link and open the attributes inspector. Here, you can name the segue as Refer to [Figure](#)

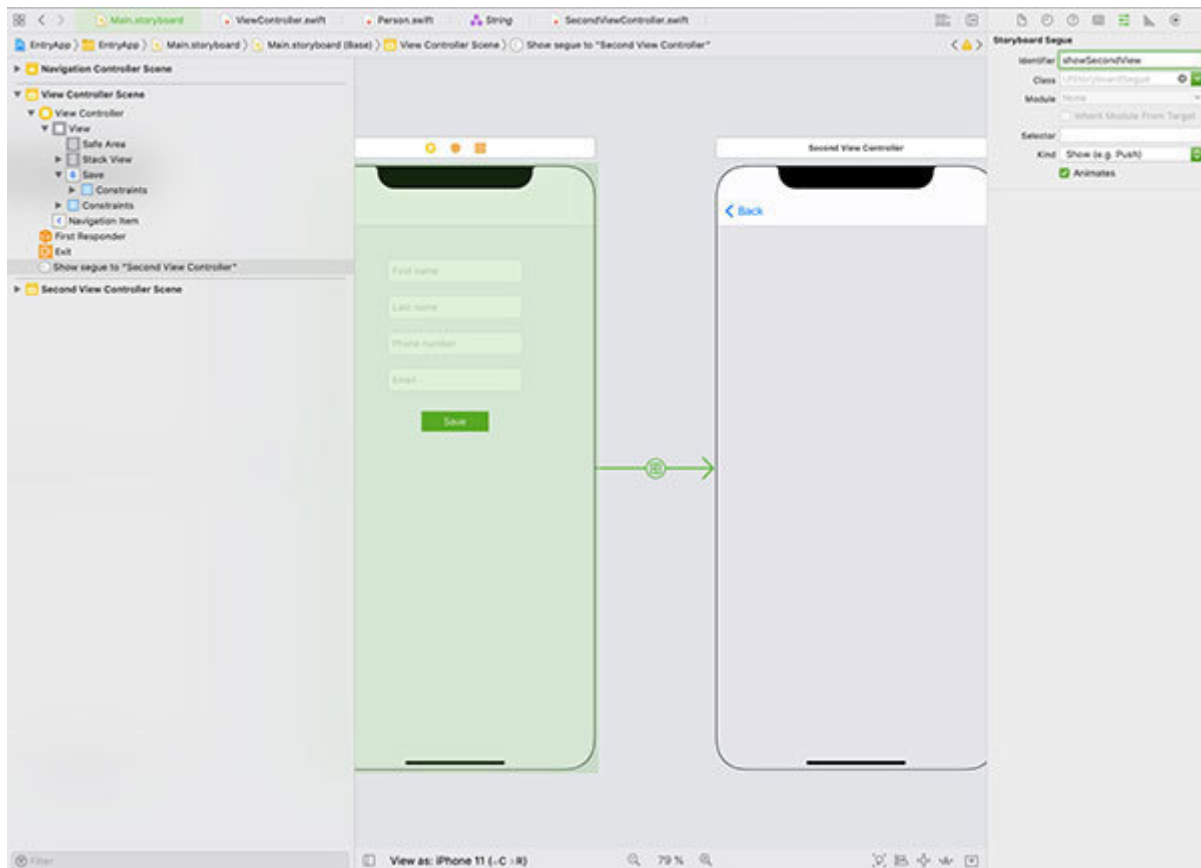


Figure 6.13: Naming the segue

Now, the last step is to call this segue from our Return to our code and comment the existing code by placing it between `/*`. Your method should now look like this:

```
@IBAction func saveButtonClicked(_ sender: UIButton) {
    performSegue(withIdentifier: "showSecondView", sender: self)
}
```

In the preceding code, we are asking the code to perform the segue with a specific identifier. The sender is nothing but the object initiating the segue; hence, `self`.

Segue behavior at runtime.

Now, we need to send our data to the SecondViewController. But if we use the segues in storyboard, at what point can we add this data to our segue request? For this, we need to first understand the flow of a segue. Refer to [*Figure*](#)

Segue Triggered



shouldPerformSegueWithIdentifier:



Create the segue and the
destination View Controller



prepareForSegue:sender:



Display destination view controller

Figure 6.14: *Flow of a segue*

Here, we will use the (We will learn more about what delegates are in [Chapter 8, Protocols and the Delegate Design](#)) The preceding figure explains the different methods getting called after a segue is triggered. Following is the sequence of events:

As soon as you call perform segue, the segue is triggered in the current view controller.

The **shouldPerformSegueWithIdentifier:sender** delegate method allows you to control the segue from happening. If this method returns a NO, the trigger is stopped. This can be useful in case you want to perform some validations before another view is presented. If it is not written in your View Controller, this method returns YES/ true by default.

If the preceding step returns YES, the Storyboard creates a segue and initializes the Destination View Controller.

This is the method which gets called right before our destination view controller is presented. Here, you can do all the preparation. You can configure the stored properties of the destination controller, i.e., you can send the data you want.

The last step is the execution of the segue and the displaying of the Destination View Controller.

Now, let us get back to our code and use what we just learned about segues. We are now going to try to pass a value from ViewController to the SecondViewController, while remembering the flow of a segue. Follow the steps below:

In your **ViewController** class, as soon as you type prepare, you should get an option to choose the prepare **for segue** method. After you select it, it will look something like this:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?)
{
    code
}
```

Now, we need to send our **Person** instance to the For this, we will create a property in the **SecondViewController** called person.

Our next step is to assign person from ViewController to the person in the For this, we need the instance of the **SecondViewController** in the prepare segue method. Here is how the final code will look like:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?)
{
    if segue.identifier == "showSecondView" {
        let person = Person(firstName: firstNameTextField.text, lastName:
lastNameTextField.text, phoneNumber: phoneNumberTextField.text,
email: emailTextField.text)
```



```
let secondViewController = segue.destination as!  
SecondViewController  
secondViewController.person = person  
}  
}
```

The preceding code is easy to understand. In the **prepare for segue** method, we will first check the identifier of the segue. This helps when you have more than one segues in your app.

Next, we will create a person which we want to send to the

In the end, we will get the instance of the **SecondViewController** by accessing the destination of our segue. Then, assign the created person to the person property of the destination view controller.

These are all the ways in which you can send data while navigating from one view to another.

We have not completed the app yet. Now, it is upto you to complete it. I will give you pointers and hints. Create a second view where you will show all the information from the first view on labels. We have already created a property, person, of type Person. Now, assign the person we created in the saveButtonClicked to this Use this in the second view to assign values to your labels. Here is the code to give you a start:

```
@IBAction func saveButtonClicked(_ sender: UIButton) {
```

```
let person = Person(firstName: firstNameTextField.text, lastName:
lastNameTextField.text, phoneNumber: phoneNumberTextField.text,
email: emailTextField.text)
let secondViewController
= self.storyboard?.instantiateViewController(withIdentifier:
"SecondViewController") as! SecondViewController
secondViewController.displayPerson = person //your hint
self.navigationController?.pushViewController(secondViewController,
animated: true)

}
```

Conclusion

In this chapter, we covered a lot of important topics in Swift as well as in iOS development. We started the chapter by building a simple app to learn navigation. Along the way, we learned about `functions` which are very useful to create chunks of task-specific code. These functions can be called from multiple locations and thus avoid code repetition. We then learned about closures, which are like the *lambda functions* or *blocs* found in other languages. Then, we learned about structures, a basic building block of a program. We then moved towards learning about properties and initializers in Swift. We ended the chapter by diving into navigation by using `segues`. In the next chapter, we will talk about classes and how they are different from structures. We will also talk about object-oriented development in Swift.

Key learnings

In this chapter, we learned about Functions and Closures.

We then studied the basic blocks of Swift and Structures.

We also saw properties and their different types mainly, computed properties and stored properties.

We also learned about subscripts and how they can simplify accessing information without using traditional methods.

We then saw initializers and deinitializers. We learned how initializers are nothing but methods and can be used to create instances with different default properties.

At the same time, we created a simple app to learn about the navigation controller.

We learned about the different ways in which we can navigate. Also, we learned about storyboard segues.

Test your knowledge

What are functions?

Function is a self-contained, task-specific block of code.

What are the different types of functions?

Global function

Nested function

What is the general syntax of a function?

```
func name(parameters) -> return type {  
function body  
}
```

What is a method?

When a function is associated with a particular type, it is called a method. A method has exactly the same syntax as that of a function. A method is always associated with a class, structure, or an enumeration.

What is a closure?

Closures are blocks of codes, which can be passed around.

Are structures value types or reference types?

Structures are value types.

What are the types of properties?

Stored properties

Computed properties

What is a failable initializer?

It is like an optional. A failable initializer can return nil in case some conditions are not met.

In which delegate method, do we control the triggering of a segue?

`shouldPerformSegueWithIdentifier:sender`

In which method, do we configure the stored properties of the destination view controller?

prepareForSegue:sender:

CHAPTER 7.

OOP in Swift

In this chapter, we will take a break from building an app. We will dive into the world of object-oriented programming. First, we will look into enumerations and classes. These are basic blocks like structures. We will then move towards concepts like *inheritance* and *polymorphism* . Then, we will learn about type casting. We will end the chapter with extensions, where we can add functionalities to an existing class, structure, enumeration, or protocol type.

Structure

In this chapter, we will cover the following topics:

Enumerations

Classes

Inheritance

Type casting

Generics

Extensions

Custom operators

Design patterns

MVC design pattern

Objective

The objective of this chapter is to help you understand **object-oriented programming** in Swift. Because when it comes to real-world coding, it is all about OOP. Classes are really important when it comes to OOP and so it is important to understand the difference between a structure and a class. Inheritance also works a bit differently in Swift. So, without further delay, let us jump right into it.

Enumerations

An enumeration is a group of related values. This makes their functioning type safe. If you already know any other traditional languages, you must have already seen them. But unlike enums in other languages, where the *name : integer* combination is used, Swift enumerations are very flexible. Here is the general syntax:

```
enum name {  
  case name  
}
```

The enums are declared by a keyword. Then, the {} contains the definition of this enum. Here is an example of an enum:

```
enum Shape {  
  case circle  
  case square  
  case rectangle  
  case triangle  
}
```

As you can see, you don't have to provide a value for each case in Swift. The values, circle, square, rectangle, etc are called **enumeration**. If we want to access the value of the shape, we do it as **let newShape = Shape.circle.**

Tip: The name of an enum always starts with a capital letter. The name should be singular. Because although it represents a group of values, when we access them, the code becomes self-explanatory as Also in the preceding example, though we have not assigned integer values, it does not mean that circle has a default value 0 and square 1.

Another way of writing an enum is by cases separated by commas as shown in the following example:

```
enum Shape {  
    case circle, square, triangle, rectangle  
}
```

Raw values

Alternatively, you can provide any type of value to be stored with the case as shown in the following examples:

```
enum Shape: String {  
  case circle = "A round shape"  
  case square = "A four equal sided shape"  
  case rectangle = "A square with larger length"  
  case triangle = "A three-sided shape"  
}
```

In the preceding example, we have changed the type of our enum Shape to a **String** type. We can now assign string values to each case. These are called **raw**. They must be unique and can be of type strings, characters, integers, and floating-point numbers.

This allows us to assign different raw values to the cases. We can then access them as shown in the following code:

```
let someShape = Shape.circle  
print(someShape.rawValue)//Result: A round shape
```

Implicitly assigned raw values

Look at the following example:

```
enum Sequence: Int {  
  case first = 1, second, third, fourth  
}
```

```
let seq = Sequence.fourth  
print(seq.rawValue) //Result: 4
```

We have *explicitly* assigned a value, only to the first case. Swift then deduces the values for the other cases. The *implicit* values of the next cases would be 2, 3, 4, respectively. If you then go and change the value of the first case to 8, then the next cases would change to 9, 10, 11. Now, coming back to our Shape example:

```
enum Shape {  
  case circle  
  case square  
  case rectangle  
  case triangle  
}
```

In this case, the *implicit* raw value of circle is circle, square is square, and so on. You can access the raw value of a case as follows:

```
print(Shape.circle.rawValue) //Result: circle
```

Initializing using a raw value

When we define an enumeration with a raw value, it automatically receives an initializer, as seen for structures:

```
enum Sequence: Int {  
    case first = 1, second, third, fourth  
}
```

Coming back to our **Sequence** example, as soon as you will try to create a new variable, you will see default initializers. Refer to [Figure](#)

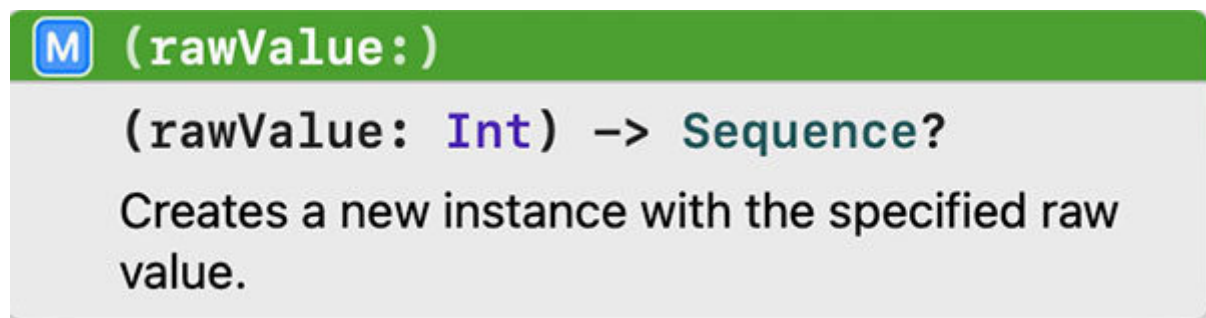


Figure 7.1: Default initializers

You will see it returns an optional because not every value will return a case.

```
let seq = Sequence(rawValue: 2)
```


In the preceding example, if we try to initialize a variable with a **rawValue** 7, it will return nil. Hence, the initializer returns an optional.

Associated values

In the preceding examples, enum is very strongly typed. But what if we want to have different types stored together in the same enum? This is when we can use *associated*. Let us understand this with the help of an example:

```
enum Shape {  
  case circle(Float)  
  case square(Float)  
  case rectangle(Float,Float)  
  case triangle(Float,Float,Float)  
}
```

Though we have not assigned a type to the enum itself, we have what is called *associated values* for each case. We will provide these values when we want to assign the enum to a variable:

```
let someShape = Shape.circle(24.0)  
someShape = .rectangle(10.0, 20.0) //Replaces the previous value
```

As you can see, we were able to assign a different type of value to But the variable **someShape** can only have one value at a time. So, if you change the circle to rectangle, the rectangle replaces the circle.

Let us look at another example:

```
enum Entry {  
  case firstName(String)  
  case lastName(String)  
  case height(Int)  
  case postalCode(Int)  
}
```

In the preceding example, we created an **enum** for the different types of entries. Now, each entry has a different type of input. Hence, we can use the associated values. We can have a mix of Strings and Ints. We will define the type of the associated values and not the type of the **enum** itself.

Switch case over an enum

In Swift, the switch is quite powerful. We saw in the previous chapter that we can even match patterns and strings. We just saw that how enums in Swift can store more than just So, we can use switch over the enum too. Here is an example:

```
enum Sequence: Int {  
    case first = 1, second, third, fourth  
}
```

```
let someSequence = Sequence.first
```

```
switch someSequence {  
    case .first, .third:  
        print("Odd Numbers")  
    case .second, .fourth:  
        print("Even Numbers")  
}
```

In the preceding example, our switch case is exhaustive. Hence, we do not have to provide a default.

Iterating over an enum

As seen in control flow, we can enumerate over some enums. We need to provide the keyword `CaseIterable`. This means our enum must conform to the `CaseIterable` protocol. Let us look at the following example of

```
enum Shape: CaseIterable {  
    case circle  
    case square  
    case rectangle  
    case triangle  
}  
let shapeCount = Shape.allCases.count  
for value in Shape.allCases {  
    print(value)  
}
```

The preceding code reads as *Shape conforms to CaseIterable protocol* (we will learn more about protocols in the next chapter). In the preceding example, you can iterate over all the cases of the `Shape` enum. After all enums are nothing but a collection of different cases.

Enumerations in Swift can provide custom initializers, have computed properties, extend, conform to protocols, and can

support many such features, traditionally supported only by This makes them very important and useful like structures and classes.

Classes

When we talk about OOP, it is clear that this programming paradigm revolves around In Swift, the concepts of **structures** and **classes** are very close. Hence, we often use the generic term instance rather than objects. In simple words, classes and structures are blueprints of our instances. So, when you look at the definition of a class or a structure, you know what it is capable of doing. Instead of explaining more, we will compare classes with structures, and this will help you understand them better.

Classes vs structures

Following are the common points between structures and classes:

Have properties

Have methods to perform specific tasks

Have custom initializers when creating instances

Can be extended using extensions

Can conform to protocols to provide some functionalities

Classes go further than this and provide some additional functionalities as follows:

This enables one class to be the child class of another and inherit its characteristics.

Classes are reference types as compared to structures which are value types.

Type This is where you can determine the type of an instance of class at runtime.

Like the name suggests, they are called to free up any resources assigned by an instance.

Reference counting enables an instance to have more than one class references.

Of course, the additional functionalities make classes make a bit more complex as compared to structures. Hence, it is recommended that you use structures whenever possible. Following are some rules when we must choose between a class and a structure:

By default, go for a structure.

Use classes if you want to have interoperability with the old Objective C code.

Use classes when the protocol is not enough and inheritance is the only way.

Use classes when you want your instance to have an identity. This means, if you change the value of the instance, it reflects everywhere. This is only possible with classes because they are reference types.

Initialization of a class

The syntax of a class is same as that of a structure. The only difference is the keyword. Even the initialization is the same as structures. Here are the examples of a structure and then of a class:

```
struct Shape {  
var length: Int!  
var width: Int!  
}
```

```
let firstShape = Shape(length: 20, width: 10)
```

// CLASS Definition

```
class Shape {  
var length: Int!  
var width: Int!  
}
```

```
let firstShape = Shape() // no default memberwise initializer
```

You will see that there is almost no difference. Except, unlike structures, the class does not have a default member wise initializer that we saw in the last chapter. Before we move ahead,

let us first understand the basic difference between a value type and a reference type.

Structures and enumerations are value types

What do we mean when we say a type is a *value* type? Let us understand this with an example. Open your playgrounds and create a struct called Shape with length and width as properties:

```
struct Shape {  
  var length: Int!  
  var width: Int!  
}
```

Create an instance called **firstShape** and then assign it to another instance. Now, if you change the width of the second shape, you will see that the width of the first shape remains unchanged:

```
let firstShape = Shape(length: 20, width: 10)
```

```
var secondShape = firstShape  
secondShape.width = 30  
print(firstShape.width!) // result = 10  
print(secondShape.width!) // result = 30
```

This means that the value of the first shape is copied to the second shape. Thus, there are two instances with values of their own. If you change one, the other does not get affected. This is what we mean when we say *structures are a value*. The same rule applies to the enumerations:

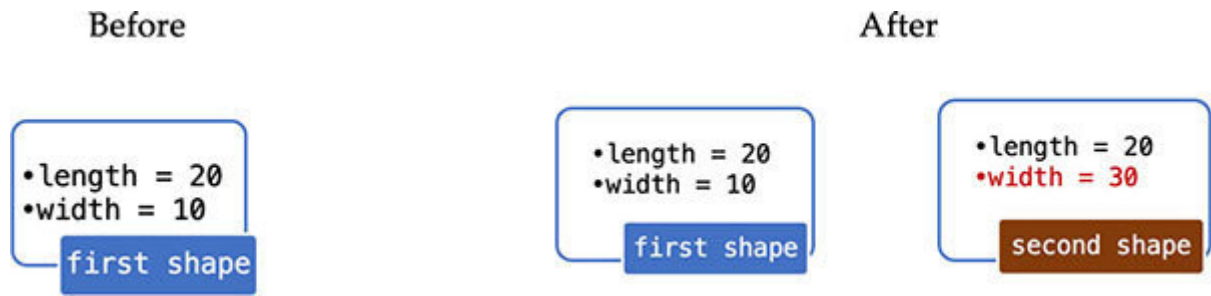


Figure 7.2: Structures are value types

[Figure 7.2](#) explains what is happening in the diagram. As you can see, on the left-hand side, we create a shape instance and assign it to the first shape. But when we assign the first shape to the second shape, two different instances of shapes are created. The values from the first shape are copied into the second shape. Then, when we make changes, only the width of the second shape is changed.

Classes are reference types

Now, change your struct to a class and do the same:

```
class Shape {  
    var length: Int!  
    var width: Int!  
}
```

```
let firstShape = Shape()  
firstShape.length = 10  
firstShape.width = 20
```

```
var secondShape = firstShape  
secondShape.width = 30  
print(firstShape.width!) // result = 30  
print(secondShape.width!) // result = 30
```

You will see that as soon as we change the width of the first shape, the width of the second shape also changes. This is because as class is a reference type; when we assign the first shape to the second shape, they both point towards the same reference. Hence, when we change one, the other changes too. This is what it means *classes are reference*



Figure 7.3: *Classes are reference types*

In [Figure](#) you will see that when we create an instance of a class, the first shape points towards this instance. When we assign the first shape to the second shape, both now point towards the same instance. Hence, if you change one, you can change the other.

Identity operators

After looking at the preceding examples, it is clear that multiple variables can point towards the same instance of a class.

However, this is not true for structures and enumerations. They are always copied. We can use the identity operators to find out if two variables or constants are pointing towards the same instance. The identity operators are as follows:

`===` (Identical to)

`!==` (Not identical to)

You can use the preceding operators to see if your constants or variables are pointing to the same instance:

```
if firstShape === secondShape {  
    print("they are pointing to the same instance")  
}
```

```
if firstShape !== secondShape {  
    print("they are not pointing to the same instance")  
}
```

Remember, they are three = signs. The double equals to is used to check the equality. The equality operator is different from the

identity operator.

Inheritance

Like in the real world, we can create **children classes** of a class. The original class is then called the **parent class** or **super class** and the **child class** is called the This process of creating a subclass is known as Inheritance allows the subclass to inherit the properties, methods, and other characteristics from its parent class. As discussed earlier, *inheritance is a behavior specific to only classes in*

Inheritance allows the subclass to add observers to the inherited properties. *Property observers can be only added to inherited* computed, or stored. Inheritance allows the subclass to modify the behavior of the methods and properties of its parent class.

Base class

By default, Swift classes do not have a parent class. If you create a new class in Swift and never create its subclass, it turns into a **base** A class without any parent or super class is known as a base class. With the following example, you will understand subclassing better:

```
class Animal {  
    var color: String!  
    var description: String!  
  
    func makeSound() -> String {  
        return "I am an animal"  
    }  
}
```

We have created a fresh class called Animal. As you can see, it inherits from no other class. Hence, it will be treated as a base class.

Subclass

Now, we will subclass the **Animal** class:

```
class Herbivore: Animal {  
  var eatsMeat: Bool = false  
}
```

Here, we have created a subclass herbivore, which represents all the non-meat-eating animals. Let us get more specific. We will further subclass this **Herbivore** class:

```
class Cow: Herbivore {  
  override func makeSound() -> String {  
    return "Moo"  
  }  
}
```

In the preceding examples, the base class **Animal** has now become a super class. Then, we subclassed it to create a class. This class will have access to all the properties and methods of the class. Herbivore has its own property, This property will be unique only to Herbivore and its subclasses. The **Animal** class does not have access to this property. The next class, is a subclass of Herbivore. It will have access to all the properties of **Animal** as well as the properties of Herbivore. In the class we

have overridden the function This allows the subclass **Cow** to modify the method. and add its own functionality.

Overriding

You must have noticed that in the **Cow** class, we have redefined the **makeSound** method. But at the same time, you must have noticed a word before it. What does this mean? As discussed earlier, all the subclasses of **Animal** will inherit the **makeSound** function. But, if we want to change the behavior of the function in a specific subclass, we can override it. This can also be done for properties. Coming back to our example:

```
class Cow: Herbivore {  
  override func makeSound() -> String {  
    return "Moo"  
  }  
}
```

```
let someCow = Cow ()  
someCow.makeSound() // Result: Moo
```

But now, if I want the **makeSound** in the **Cow** to show the results of **Animal** as well as its own, we must use the word

```
class Cow: Herbivore {  
  override func makeSound() -> String {  
    super.makeSound() // we called the same function of the super  
    class  
    return "Moo"  
  }  
}
```

```
}  
}
```

```
let someCow = Cow ()  
someCow.makeSound() // Result: I am an animal \n Moo
```

You will notice that we first called the **super** class function and then returned our own result. This way, we had the results for both.

Following are the important points to remember about inheritance:

You can override a method of a super class by using the keyword **override**.

If you want to call the super class method from the subclass, you can use the keyword **super**. For example, **super.makeSound()**.

You can even override the getters and setters of the properties of the super class.

As with methods, the superclass version of properties can also be accessed using the word **super**.

If you do not want a method or property to be overridden, you can add the word **final** before the definition, as **final final var**.

Method overriding, which we saw in the example of is also called **run-time polymorphism**.

Tip: Like other languages, Swift does not support multiple inheritance. A Swift class can only inherit from a single class. To add multiple functionalities, we use Protocols. We will look at Protocols in the next chapter.

Type casting

In real-world coding scenarios, we end up using a lot of pods and libraries. There can be a lot of subclassing and use of many protocols. In such cases, it can be useful to know what type of instance we are dealing with. **Type casting** helps us to check what type of instance we are dealing with. Inversely, it can be useful to treat an instance as a superclass in its own hierarchy. Let us learn this with examples.

Type checking

The keyword used to check a type is also called as the *type check*. The operator returns true if a constant or variable belongs to a certain subclass type. It returns false if it's not. Here is an example:

```
for item in animals {  
    //we are using the is operator  
    if item is Cat {  
        print("The animal is a cat")  
    }  
}
```

In the preceding example, we use the **type check operator** to verify if the animal in the list of animals is a Cat.

Down casting operator

The keyword for this is also called the **down-casting**. We can cast a value to a different type or check the type of a value simply by using these two operators. There might be situations where we must use a certain variable of a certain class as that of a different subclass.

Here is an example:

```
for item in animals {  
  //we use the as operator to use the item as an instance of Cow  
  subclass  
  if let animal = item as? Cow {  
    print("The animal says \ (animal.makeSound())")  
  }  
}
```

We used the optional because we are not sure if the animal is a Cow. We can use the to force unwrap, if you are sure that the item is going to be a Cow.

Type casting for Any.

Sometimes, we are just not sure of the type which might exist in the collection. Or, we might not be sure what might be the type of an instance. Swift provides the following two keywords to overcome this.

Any : This keyword represents an instance of any type, including function types.

This represents an instance of the class type.

If used wisely, you can exploit this to the fullest. Here is an example:

```
var someList = [Any]()  
  
someList.append(4.0)  
someList.append(10)  
someList.append("This is a string")
```

In the preceding example, we could add different types to a list. This was possible because of the Any keywords. We can then use the type check operator on the elements while iterating to find their types.

Generics

Another important feature of Swift is This helps us write reusable functions. We have already seen that in Swift, an array of a String is a type and an array of **Int** is a different type. But do you think Apple has written the code to address each type? No. Arrays and Dictionaries are both generic type collections in Swift. What does generic mean? Let us understand this with an example.

Here are three functions which do the same job:

```
func multiplyTwoInts(a: Int, b: Int) -> Int {  
    return a * b  
}
```

```
func multiplyTwoFloats(a: Float, b: Float) -> Float {  
    return a * b  
}
```

```
func multiplyTwoDoubles(a: Double, b: Double) -> Double {  
    return a * b  
}
```

If we want to combine them, write it as a single function. Generics can be helpful. Here is the end function:

```
func multiplyTwoNumbersNumeric<(a: T, b: T) -> T {  
    return a * b  
}
```

Numeric< in this represents a generic type. You can replace it with any word. But make sure that you use the same everywhere in the function. **Numeric**< tells the function that the function will accept the generic type T which conforms to the Numeric protocol. If we try to use the function with strings, it won't work. Here is an example of a Generic class:

```
class Storage {  
    var value: Element  
    private var storageSpace: [Storage] = []  
    init(value: Element) {  
        self.value = value  
    }  
    func addToStorage (item: Element) {  
        self.storageSpace.append(Storage(value: item))  
    }  
}
```

Once this class is created, we can use it for non-generic, type-specific solutions as follows:

```
let integerStorage = Storage(value: 22)  
integerStorage.addToStorage(item: 23)  
  
let stringStorage = Storage(value: "Hello")
```

```
stringStorage.addToStorage(item: "Bye")
```

We can make the generic type conform to specific protocols to make them case-specific. Like we did for the function, which only multiplied numbers.

Extensions

In some scenarios, we might want to add functionalities to existing types, without rewriting it or subclassing it. It can be a class, structure, enumeration, or protocol type. It can be one of our own written classes or a type to which you do not have access to, i.e., one of Apple's. This is when we use them. They help us extend or create an extension of an existing class. Imagine using an electric extension with multiple sockets.

Extensions in Swift can allow us to do the following things:

Define new methods; instance and type methods

Define new properties; computed instance properties and computed type properties

Enable an existing type to conform to a new protocol.

Define new custom initializers.

Define subscripts.

The syntax of an extension is quite simple as follows:

```
extension RandomType {
```



```
//code to add new functionality goes here  
}
```

The preceding example shows the basic syntax of writing an extension for an existing type. Let us look at a real-world example:

```
extension String {  
  var isValidEmail: Bool {  
    let emailRegex = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,64}"  
    let emailTest = NSPredicate(format: "SELF MATCHES %@",  
    emailRegex)  
    return emailTest.evaluate(with:self)  
  
  }  
}
```

In the preceding example, we created an extension on String to check whether a string is a valid email address. We can use it as follows:

```
let anEmail = abc@bcd.com
```

```
if anEmail.isValidEmail {  
  //do something  
}
```

Thus, we added an extra functionality directly on the String instead of creating a function and passing every string through it. This makes the code cleaner.

Computed properties

The example we saw for the string was that of writing a method for the extension. We can even write properties in extension. Here is an example:

```
extension Int {  
  var square: Int {return self * self}  
  var cube: Int {return self * self * self}  
}
```

```
let someInt = 3.square  
print("the square of the integer is:", someInt) // Result the square  
of the integer is: 6
```

In the preceding example, we wrote some computed properties for `Int`. We can then use them directly as shown in the example. But you will notice that the result is 6 instead of 9. Why is that? Because while overloading the operators, we had changed the multiplication method.

Extending initializers

We have seen that structures come with default initializers. With extensions, we can write different initializers. Let us take the example of our Shape **struct** which we previously used:

```
struct Shape {  
  var length: Int!  
  var width: Int!  
}
```

```
let firstShape = Shape(length: 20, width: 10)
```

With an extension on Shape, we can write another initializer to accept more parameters as follows:

```
extension Shape3 {  
  init(length: Int, width: Int, multiplicationFactor: Int) {  
    self.length = length * multiplicationFactor  
    self.width = width * multiplicationFactor  
  }  
}  
  
let secondShape = Shape3(length: 10, width: 10,  
  multiplicationFactor: 2)
```

Custom operators and operator overloading

In the very beginning of this book, when we started to learn the Swift basics, we learned about the operators such as multiplication, division, addition, etc. With extensions, you can *overload* the existing operators. Swift also allows you to define new operators. Let us first look at an example of overloading:

```
extension Int {  
  
    static func * (left: Int, right: Int) ->Int {  
        return left + right  
    }  
}  
  
let result = 4*2//result = 6
```

In the preceding example, we overloaded the multiplication method for **Int** and changed the result to return addition instead. This is called **method overloading** or **compile time**

While learning about we saw, there are three types of operators: infix (in between), prefix, and postfix. We can create our own operators using these keywords. Here is an example:

```
infix operator °
```

We can then use the extension like we did for overloading and write a function for this operator as follows:

```
extension Int {  
  static func ° (left: Int, right: Int) -> Int {  
    return (left * right) + 2  
  }  
}  
let result = 4 ° 2 //result = 10
```

Design patterns

In this section, we will take a look at the design patterns. In real-world coding, design patterns are really important to organise your code better. Huge apps have multiple architectures working together. Design patterns provide a common, reusable solution to recurring problems while designing a software. It can be described as a template or a blueprint, which one can use to structure their code. Design patterns are not like copy paste libraries. It provides structure to your code.

In the object-oriented programming context, a design pattern helps to understand the relationship between different objects and classes. As design patterns provide reusability, they help to increase the development speed when working on huge codes. There are four types of design patterns:

Creational design This pattern describes the instantiation and creation of objects: For example, *Builder pattern*, *Singleton*, *Abstract pattern*.

Behavioural design This pattern describes the behaviour between different instances and the way they communicate amongst themselves. For example, *Observer pattern*, and so on.

Structural design This organizational pattern describes the way classes and structures can be organised to form larger ones

towards implementing a certain functionality. For example, *Façade*, *Model-View-Controller*, and *Model-View-ViewModel*.

Concurrency design This pattern is used in the multithreaded software context. In iOS app development, this pattern is not used. For example, *Reactor pattern* and *Active object pattern*.

Advantages of design patterns

Following are the advantages of using design patterns:

They provide a *common* As mentioned earlier, design patterns are like templates or blueprints. So, once someone says they have implemented the MVC pattern in their code, another developer can get a first-hand understanding without actually looking at the code.

When a new member joins a development team, it is easier for them to understand the structure if a known design pattern is used.

As design patterns are built around solving a common problem, inversely they can help to spot problems easily.

They are part of the best practices, i.e., you become a better developer if you know the design patterns.

Drawbacks of design patterns

Following are the disadvantages of using design patterns:

Usually in a code, there are multiple design patterns. As they are built around solving problems, a code can become really complex if design patterns are overused.

As simple it is to jump on a project with a design pattern, it can be really difficult to adapt an existing project to a new design pattern. This can be time consuming and can give rise to new issues.

An inexperienced team can suffer due to lack of knowledge of a design pattern. This can lead to new bugs and time delays.

Though we won't be learning about all of them, two patterns are really important: **Model-View-Controller** and We will learn more about the delegate pattern in [Chapter 8, Protocols and the Delegate Design](#) Let us now take a look at the Model-View-Controller.

Model view controller

Model-View-Controller or MVC is the most widely used design pattern while designing iOS apps. You have been using it all along without even realising. Let us first look at [Figure 7.4](#) to have a better understanding:

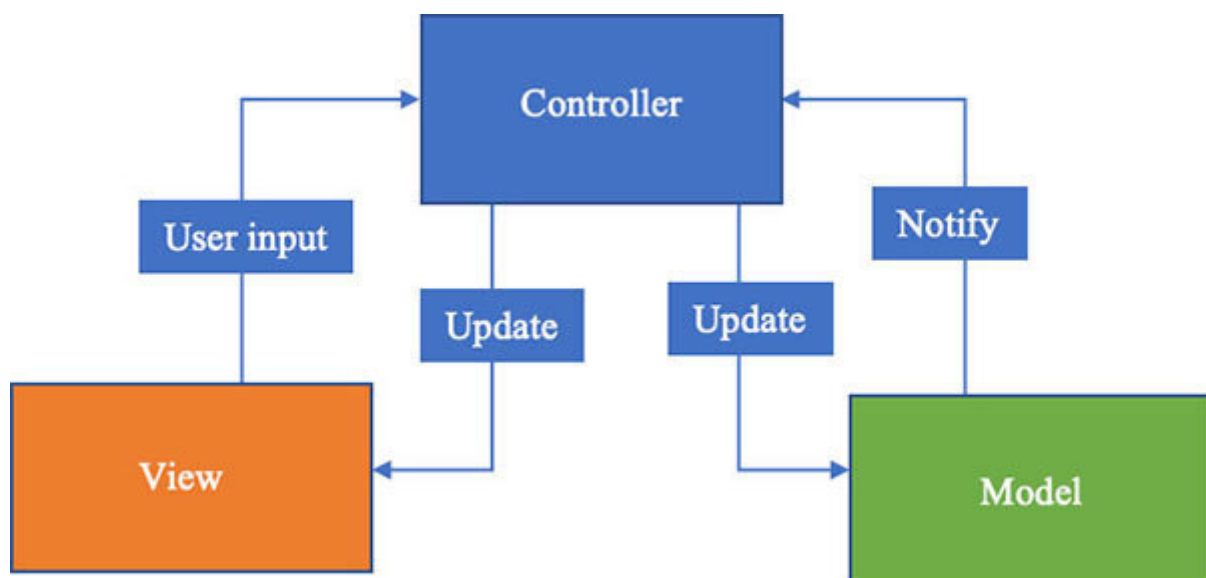


Figure 7.4: Model View Controller

The preceding figure explains the model-view-controller design pattern. In simple words, we want to see data in our app and interact with it; the model is our data, the view is our point of interaction and the controller is the manager. Let us understand each component and then the way they communicate:

Model is the data we want to show in our app. Remember the app we created in [Chapter 6](#) in which **Person** was our model. Multiple models can share relationships, thus creating a wider graph. This is also known as the **model**. They usually hold the logic to manipulate and process them. They can be reused as needed by the controller. This data represents the state of the application, i.e., when the app is launched, the data is loaded in these models. This data can also be persisted in the form of a database or files if needed.

Model As you can see, the model *notifies* the controller of its change. This change is then processed by the controller to *update* the appropriate view. There are various ways in which the controller can be notified. The most used is the `NSNotificationCenter`. With it we can broadcast a notification and all the classes listening to the notification can then get the update.

This is the interface that users can see. User inputs are captured via the Views. As seen in the preceding figure, Views and Models do not communicate to each other. They do so through the Controller. In other words, *views* and *models* are decoupled. This decoupling also makes the view components reusable. They can be reused at multiple places in the same project as they are not directly linked to any data. Their job is to only show what the controller tells them to. All the `UIButton`s, `UILabel`s, and `UITextField`s we have created so far are examples of views. You can thus see that *Views* can have multiple subviews and can be complex.

View communication: The trigger is the *user*. This input is then processed by the controller to change the data associated with it. The model then notifies its change to the controller, which

in turn again *updates* the view so that the user can see the changes.

By now, the job of the controller must be pretty clear for you. The controller is the intermediate class between the View and the Model. It has the logic to communicate data changes to views and vice versa.

Remember our ViewController used in the app built in [Chapter Code Structuring](#). The controller has all the methods to process the click events of the buttons. It also handles the navigation between two views. When we enter data in the text field, it is the view controller which uses this data to create a **Person** instance. When we create and send this instance to the again it's the controllers' job to assign the data to the labels. We even saw the use of controller to manipulate data before a segue is executed. Thus, you can clearly see how important a controller is in the app life cycle.

Controller The way a controller gets user actions is via IBActions, touches, and other gestures. It then notifies the model by changing its properties or calling its functions.

MVC in iOS

We have already built a couple of applications so far. The **ViewController** class which we use is in fact a controller. The nib files are our Views and Model files are for us to create. Most Cocoa applications use MVC at its core, as a design pattern. This can be a very simple starting point for iOS application development. As the application grows bigger, you will find yourself using multiple patterns along with MVC.

Drawbacks of MVC

Following are the drawbacks of MVC:

One of the biggest drawbacks of the MVC is that over time the ViewController gets bigger and more complex. This can be hard to manage and debug.

You can find yourself with classes which cannot be categorised as model, views, or controllers.

As the pattern relies on the event-driven nature of the application, the debugging can be complicated.

Unit testing becomes pain staking with this pattern.

It can be complex to implement for bigger applications.

Conclusion

We have completed an important chapter. This chapter helped you understand how object-oriented programming works in Swift. We started with enumerations and then compared structures and classes. This helped us understand classes better. Then, we saw the behavior of inheritance, unique only to classes. With type-casting, we learned how to use the type check operator and the down-casting operator. We ended the Swift part with extensions, which helps us add functionalities to existing types. We then moved towards design patterns. We looked into the MVC design pattern, which is widely used for simple iOS applications. In the next chapter, we will learn about protocols. We will also see the protocol-oriented programming in Swift.

Key learnings

We started with studying enumerations. We saw how they are different in Swift. We learned about raw values, assigned values, and ways to iterate over enums.

Then, we dived into classes. We learned more about them by comparing them with structures. We saw how they are not so different in syntax, but do not have the default initializers.

We also learned the difference between reference types and value types using examples.

We then discussed how inheritance is implemented in Swift. We saw the different ways of overriding and also run time polymorphism.

We then saw how we can use the identical operator to verify if two variables or constants are pointing towards the same instance of class.

We then saw the two operators used for type casting: type-check operator, down-cast operator.

We then looked at Generics and how they can be used to create and write reusable code.

On the same lines, we learned about extensions. How they can be used to override functions.

Finally, we had a brief introduction to design patterns. We learned their advantages as well as their disadvantages. We also saw how the Model-View-Controller design pattern works.

Test your knowledge

What is the difference between the initialization of classes and structures?

Structures come with default initializers.

Enumerations are strongly typed. True or false?

True. But we can use assigned values to get around this.

Which protocol does an enum needs to conform to, so we can iterate over its cases?

CasIterable

List two major differences between classes and structures.

Classes are reference types and Structures are value types.

Classes support inheritance.

What are the different type casting operators?

Type checking operator : is

Downcasting operator: as

What is custom overloading?

Custom overloading is when we extend a type and override an existing operator.

What does AnyObject represent?

It represents any instance of a class type.

Which is the most widely used design pattern in iOS?

The MVC or model view controller pattern.

Which is biggest drawback of MVC?

Over time, the view controller gets bigger and more complex.

Protocols and the Delegate Design Pattern

In this chapter, we will learn about **protocols** . In the previous chapter, we saw how design patterns help us address common problems in a code. We already learned about the **Model-View-Controller** design pattern in the previous chapter. Now, we will learn about the delegate design pattern. To understand the delegate pattern, you need to also understand the protocols. To get a better understanding, we will build a simple application alongside.

Structure

In this chapter, we will cover the following topics:

Protocols

Protocol-oriented programming

Delegate design pattern

Word Bank App

Objective

The objective of this chapter is to help you with protocols. We have already seen in the previous chapter that we cannot inherit structures. To overcome this problem, Swift proposes *protocol-oriented*. We will build a table view-based app. This will help you understand the UITableViewController as well as the delegate design pattern.

Protocols

We saw in [Chapter 7, OOP in](#) that in Swift we cannot have multiple inheritance. If you come from a traditional object-oriented programming background, you might be used to multiple inheritance. Inheritance helps your class to have multiple functionalities from different classes. Nonetheless, these are concrete classes. We saw how we can override a super class method in a subclass. Now, imagine we have inherited a class C from two classes A and B. Both A and B have a method. If C calls this method, without overriding it which class, A or B, will the method be called upon? This is one of the problems of multiple inheritance.

This is where the *protocols* in Swift help. *Protocol provides a definition of a specific* For example, we often have real-life protocols when there is a fire emergency, or the most famous is the one where the President of the country is in danger. Protocol is a set of steps to be taken to ensure the safety of the President. Here, the task is *safeguard the* Similarly, in Swift a protocol is like a blueprint which has all the methods, properties, and other such requirements to fulfill a task. This protocol can be then implemented by a class or structure to extend its own set of functionalities. A type can implement multiple protocols at the same time. We can say that a type conforms to a protocol.

Protocol syntax

The syntax of a protocol is simple. It is as follows:

```
protocol SampleProtocol {  
  //protocol definition  
}
```

The way a type can conform to a certain protocol is by placing it in front of its name after a colon. Here's how:

```
struct SampleStruct: SampleProtocol {  
}
```

And here is how, you can implement multiple protocols by just placing them one after the other separated by a comma:

```
struct SampleStruct: SampleProtocol, AnotherProtocol {  
}
```

However, if you have a class inheriting from another class, the name of the superclass goes first as follows:

```
class Giraffe: Animal, SampleProtocol, AnotherProtocol {  
}
```


Here, **Giraffe** inherits from **So**, we place the **Animal** at the beginning and then the names of the protocols the **Giraffe** class will implement.

Defining a protocol

Let us try to understand what can be written inside the protocol definition or the *requirements of a*

PROPERTY REQUIREMENTS

Properties in a protocol are always declared as Their getter and setter are defined by declaring **{get}**, Here is an example:

```
protocol SampleProtocol {  
  var firstProperty: Int {get} // only gettable  
  var secondProperty: Int {get set} //gettable and settable  
}
```

As soon as you declare these properties, you will see a compiler error in the **Giraffe** class. Note that these are only declarations. A protocol does not tell if the property is a stored property or a computed property. You can see the message reads, *Giraffe does not conform to* This means that when we declare that a class will implement a protocol, we tell the compiler that our class will fulfill all the requirements to implement this protocol. So now, the first thing we need to do this is to fix the error:

```
class Giraffe: Animal, SampleProtocol, AnotherProtocol {  
  var firstProperty: Int = 0  
  var secondProperty: Int = 0
```

```
}
```

As soon as you implement the two properties in your **Giraffe** class, your errors will disappear. This is because you have fulfilled all the requirements of the protocol.

METHOD REQUIREMENTS

Like properties, protocols also require specific methods to be implemented by types conforming to them. These methods are written along with the properties when we define a protocol. They have the same rules as normal methods; however, *method requirements cannot have default values in a protocol* Let us rename our protocol and define a method in it:

```
protocol DanceProtocol {  
  var canDoJazz: Bool {get} // only gettable  
  var canDoHipHop: Bool {get set} //gettable and settable  
  func danceHipHop()  
  func danceJazz()  
}
```

So, we renamed our protocol to **DanceProtocol** and now we have changed some properties. Any class conforming to this protocol will look like this:

```
class Giraffe: Animal, DanceProtocol {  
  var canDoJazz: Bool = true  
  var canDoHipHop: Bool = false
```

```
func danceHipHop() {  
}  
func danceJazz() {  
}  
}
```

Now, if Giraffe wants to conform to the it must dance hip hop and as well as dance jazz.

INITIALIZER REQUIREMENTS

Like methods, protocols may require the conforming types to have specific initializers. Let us modify our **DanceProtocol** to have an initializer as follows:

```
protocol DanceProtocol {  
  var canDoJazz: Bool {get} // only gettable  
  var canDoHipHop: Bool {get set} //gettable and settable  
  func danceHipHop()  
  func danceJazz()  
  init(numberOfSteps: Int) //new init method  
}
```

As soon as you do this, your **Giraffe** class implementation will throw an error. Fix it to implement the **init** stub. Your class should now look like this. You will notice a keyword, This modifier ensures that all the subclasses of the class conforming to the protocol have this initializer implementation:

```
class Giraffe: Animal, DanceProtocol {  
  required init(numberOfSteps: Int) {  
    //code to initialize goes here  
  }  
  var canDoJazz: Bool = true  
  var canDoHipHop: Bool = false  
  func danceJazz() {  
  }  
  func danceHipHop() {  
  }  
}
```

In case, you are overriding an initializer from the super class you can put the override modifier after the required modifier as follows:

```
Required override init(numberOfSteps: Int) {  
  //code to initialize goes here  
}
```

Initializers of classes marked with the final modifier do not need to be marked with the required modifier because final classes cannot be subclassed.

Protocol-oriented programming

With Swift, Apple is trying to redefine the way we think about programming. So, it introduced protocol-oriented programming or POP.

Though one might think that this way of programming is all about using protocols; instead, it is to overcome the problems of object-oriented programming. Let us first try to understand the major problems of object-oriented programming:

Tracking the OOP is all about objects or as we say in Swift, Hence, we have to keep track of all the instances you create. If you are using classes, you have to make sure they are not unexpectedly changed. Or ensure that your instance of class is not wrongly referred somewhere else because classes are reference types.

Huge We saw that Apple does not support multiple inheritance. As you are allowed only one superclass, you have to choose the right one at the start. It is not so flexible to change it. Also, if your superclass has a lot of stored properties, which might not be useful to one of the subclasses, you cannot do anything about it. The subclass has to bear the cost of initialization of properties it does not need. Over the time, your super class can become huge due to all the different functionalities it has to support.

Lack of blueprint If a superclass defines a method, it **MUST** have a body. Even though the superclass does not use this method, it needs to implement it. Inversely, if a method does not need to be overridden, you have to explicitly specify it using the keyword `final`.

We need thus a way of programming that can have the following features:

Supports both structs (value types) and classes (reference types).

Non-monolithic approach, i.e., not one superclass.

Should be able to conform to another type's requirements.

Has no initialization cost on the subclass.

The distinction between implementation and override is clear.

The answer to all the preceding problems is the POP. Following are the reasons why we should use POP in Swift:

Apple has made it clear that *at heart, Swift is protocol*. Even in a simple array definition, which implements around 11 protocols to the standard Swift library, we can see that Swift emphasizes heavily on **protocol-oriented programming**

Swift pushes for the use of value types (like structs) than reference types. These value types can conform to protocols but

cannot use inheritance.

As we will see later in this chapter, protocols form the base of the delegation pattern.

With POP, types can conform to multiple protocols, thus eviting the problem of multiple inheritance.

Unlike other languages, in Swift, we can use protocols as types. We can even extend Swift protocols.

Protocols as types

As seen already, protocols are definitions. Nonetheless, they can be used as **types** like structs and classes. When you use a protocol as a type, it is called **existential**. This is because the type is in fact a type which *conforms* to the protocol. Following are the places where you can use a protocol instead of a type:

The type is a constant, var, or property.

In an array, it is a dictionary or other collections.

In a function, method or initializer it is a parameter type or a return type.

Here is an example of a protocol being used as a type:

```
init(danceProtocol: DanceProtocol) {  
    super.init()  
}
```

We have passed the protocol as a parameter in the initializer. So, when the type is initialized, we can pass a type, *which conforms to* as a parameter to the initializer. Thus, you can send an instance of any type which conforms to the DanceProtocol. This is very useful as compared to passing a superclass. Passing a superclass

can be limiting and if we pass `AnyObject`, that can be too wide. While learning about Generics, we saw how we can use the protocol as a type while creating a generic function.

In the following example, the Generic `T` is of type `Numeric`. But in fact, the generic type conforms to the `Numeric` protocol.

```
func multiplyTwoNumbersNumeric<T>(a: T, b: T) -> T {  
    return a * b  
}
```

Protocol extensions

The other feature of POP mentioned was that protocols can be extended. This feature is helpful when you want optional implementations. Here is an extension of the DanceProtocol:

```
extension DanceProtocol {  
  func danceHipHop() {  
  }  
}
```

Thus, you have provided a default empty implementation for the **danceHipHop** method. Now, if your type conforms to DanceProtocol, you do not need to implement the **danceHipHop** method. The compiler won't throw an error. However, if you do implement the method, it will override the default implementation as follows:

```
class Giraffe: Animal, DanceProtocol {  
  var canDoJazz: Bool = true  
  var canDoHipHop: Bool = false  
  func danceJazz() {  
  }  
  func danceHipHop() {  
    //this will override the default implementation of the extension.  
  }  
}
```

Protocol inheritance

Now, we don't want our Giraffe to only dance; what if we want it to study Maths? Let us define a new protocol as follows:

```
protocol CanDoMaths {  
    func canAdd()  
    func canMultiply()  
    func canSubtract()  
    func canDivide()  
}
```

If we want our **Giraffe** class to do Maths, we must make it conform to the preceding protocol. As discussed earlier, this can be achieved with multiple protocols working together. Your final class will look something like this:

```
class Giraffe: Animal, DanceProtocol, CanDoMaths{  
    required init(numberOfSteps: Int) {  
        //code to initialize goes here  
    }  
    //MARK: CanDoMaths protocol  
    func canAdd() {  
    }  
    func canMultiply() {  
    }  
    func canSubtract() {
```

```

}
func canDivide() {
}
//MARK: Dance Protocol
var canDoJazz: Bool = true
var canDoHipHop: Bool = false

func danceJazz() {
}
func danceHipHop() {
//this will override the default implementation
}
}

```

Unlike classes, protocols can inherit from one or more protocols. This enables an existing protocol to add more functionalities to an existing protocol. This approach is way better than multiple inheritance, as you can avoid unnecessary initialization costs. Here is an example of multiple inheritance syntax:

```

protocol SalsaProtocol: DanceProtocol, AnotherProtocol {
}

```

In the preceding example, the **SalsaProtocol** conforms to So, if our **Giraffe** class conforms to it can do hip hop as well as jazz!!

Delegate design pattern

It is simple to understand delegation. Imagine the conferences at the United Nations. Does the whole country go there? No!! Instead, a *delegation* or a bunch of people representing the country attend the conference. Here is the definition provided by Apple documentation:

“Delegation is a design pattern that enables a class to hand off (or “delegate”) some of its responsibilities to an instance of another class.”

– Apple documentation

It is a very clear definition. So, where does the protocol get involved in this? Let us look at the following figure to get a better understanding:

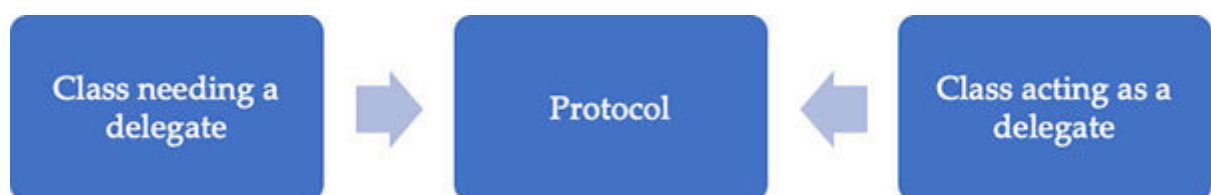


Figure 8.1: Protocol when used as delegate

As seen in [Figure](#) a protocol is in fact the definition of the delegate. The class acting as a delegate conforms to the protocol. This enables the delegate to implement the protocol and execute

the functionalities on behalf of the class. The protocol is more important than the class implementing it. Thus, any class implementing the protocol can become the delegate for the class. This adds enormous flexibility while coding. We will see a delegation more while making our next app.

We have already implemented delegates in the previous chapters. Do you remember UITextFieldDelegate? So basically, the view controller acts as the delegate to the UITextFieldDelegate. Now, we will make an app to get a better understanding.

Word Bank App

We will make a simple word bank app for people learning a new language. The idea is simple. You can make a list of new words you learn with their translation in your language and their meanings. You can also add a picture to remember the word. While doing so, we will use the MVC (Model View Controller) along with the delegate design pattern. The final app will look something like this. Refer to [Figure 8.2](#) for the landing screen and the detail screen. When we click on the row, it will take us to the second screen. For this, we will use the navigation controller. We will try to use the same screen to add new words and update existing words as shown in [Figure](#)

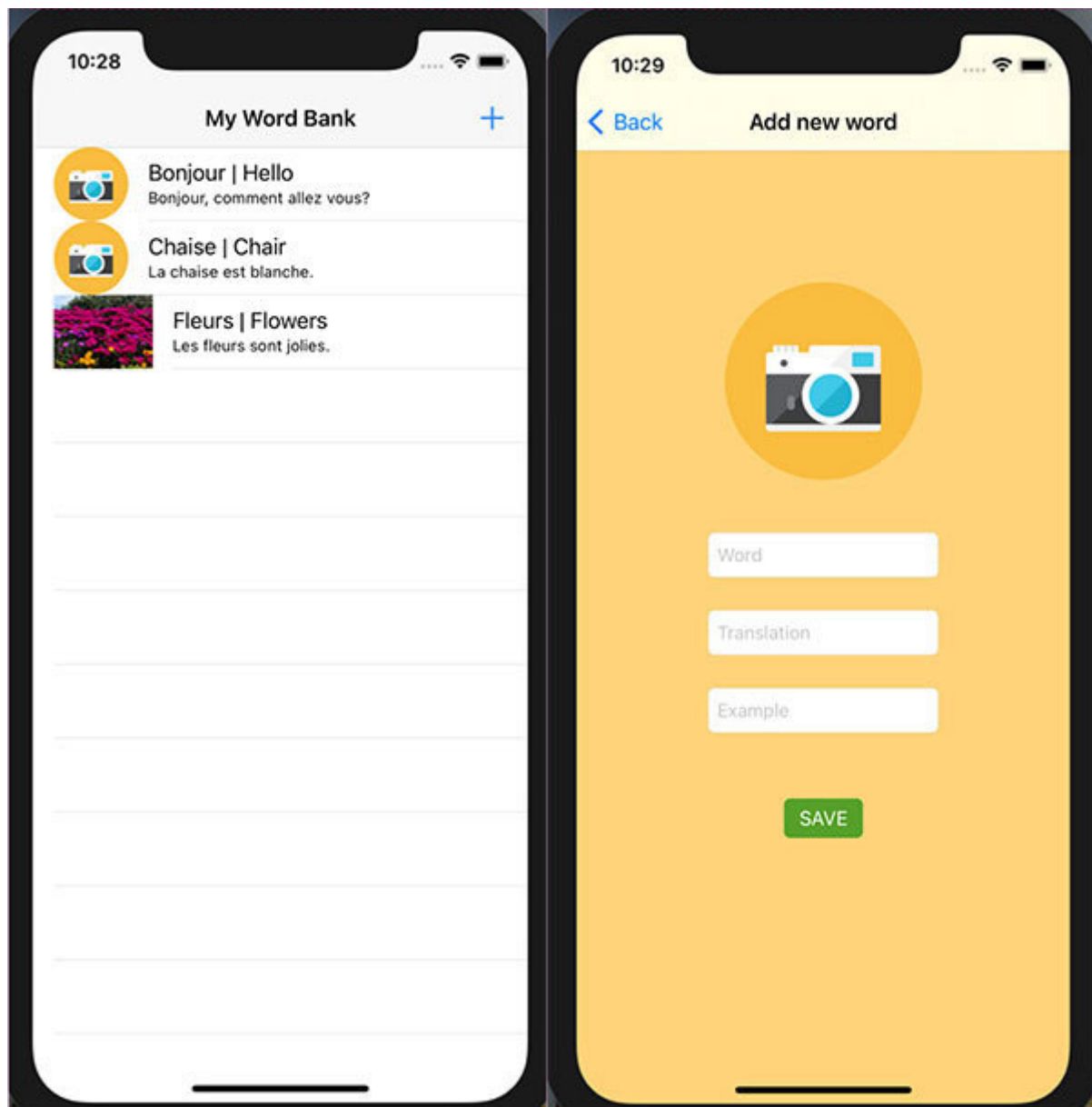


Figure 8.2: Landing and detail screen

Also, we will use UITableViewController, which is important to learn while learning the iOS app development. So, open your Xcode and let us begin!!

We will start by creating a new project. Follow the steps below:

Open Xcode. Create a new project and name it Then, in the storyboard, add a navigation controller. Make your view controller the new rootviewController of the navigation controller. We have already done this in the previous chapter. Refer to [Figure](#)

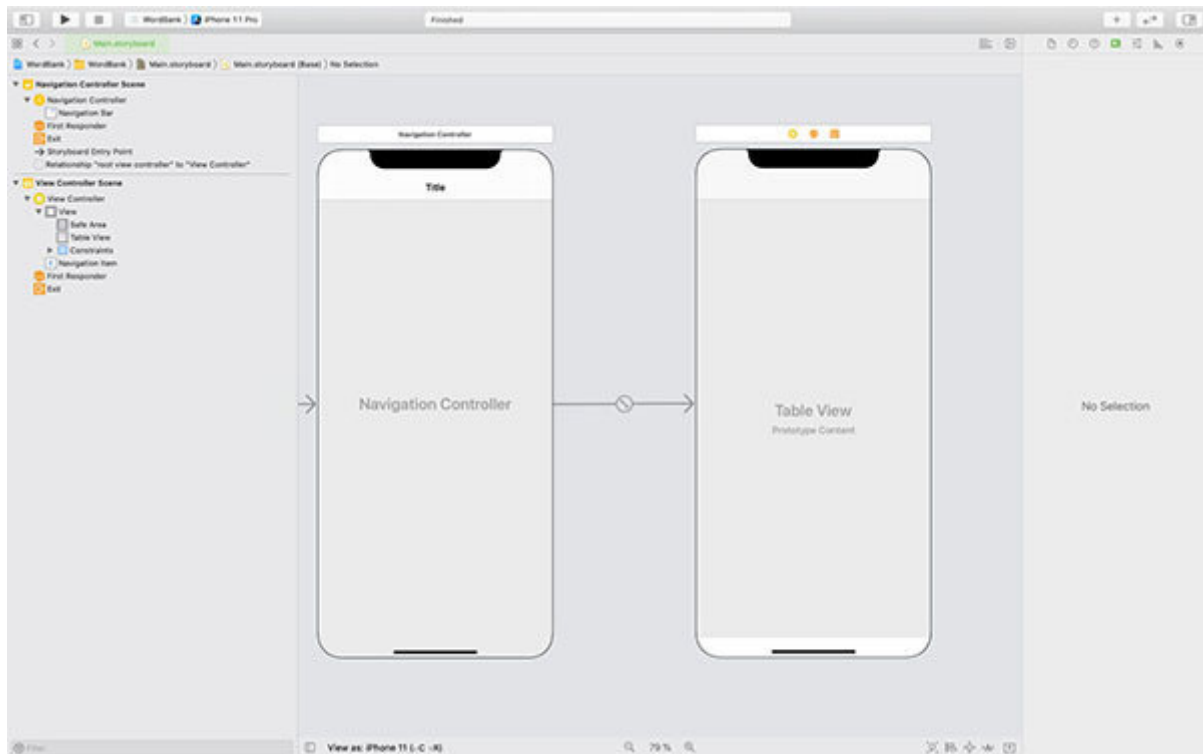


Figure 8.3: Changing the rootview controller

Next, add a table view to the view controller. Add the necessary constraints to stretch it from end to end. Then, create its outlet in the view controller as follows:

```
@IBOutlet weak var wordsTableView: UITableView!
```

Now, we need to fill the table view. Usually, this job is done by the **UITableViewDelegate** and We can create a delegate class for

this. But our ViewController is a small one so we can just conform it to the **UITableViewDelegate** and

As soon as you do this, you will see the compiler throw an error. Go ahead and add the protocol stubs. These are nothing but empty methods of the Refer to [Figure](#)



Figure 8.4: Protocol non-conformation error

Tip: Apple usually divides its protocols into two types such as DataSource and Delegates. DataSource is the one which sources or gives data and Delegates is the one which implements some functionality.

Configuring the table view

Let us now build our table view. We want to show items, one below the other. Follow the steps below to configure your table view:

You will not see any error for **UITableViewDelegate** because this delegate does not impose any required methods. But you can jump to its definition to go and check what they are.

First, let us understand what a table view is. A table view in iOS is a list of items. It has sections and every section has rows. The default section count is 1. Every table view section has a header and a footer. Let us look at the methods of the **UITableViewDelegate**. The first method sources the number of rows. This for us is going to be the count of our array. So, go ahead and create an array. We return the array count in this method.

```
func tableView(_ tableView: UITableView, numberOfRowsInSectionSection
section: Int) -> Int {
    return self.wordsArray.count
}
```

The next method is the **cellForRowAt**. In a **tableview**, every row has an index path. This has its row number and its section number. Every row must return a cell to display data. A table view cell is

nothing but a view. Thus, we repeat the same view multiple times but source it with data from our array:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
}
```

Open your storyboard to create a table view cell. Open the view library and drag and drop a table view cell on your table view. Click on the newly added cell and in the attributes inspector put its identifier as Refer to [Figure](#)

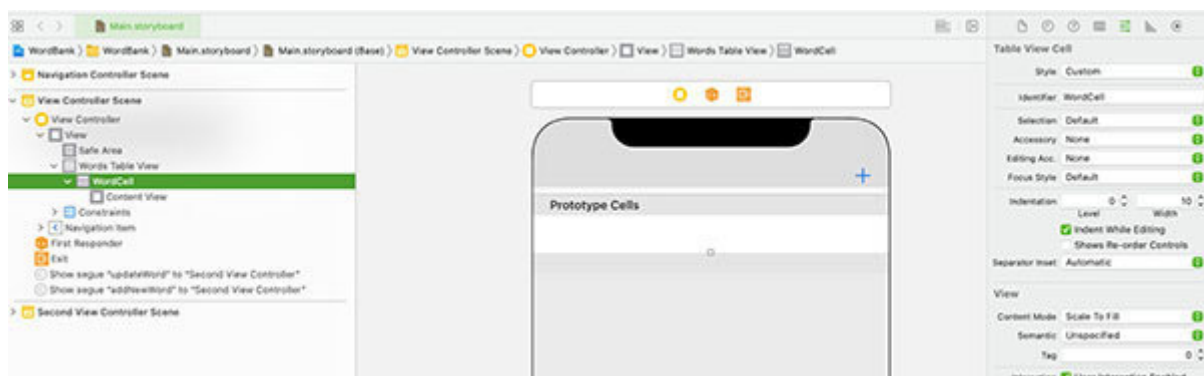


Figure 8.5.1: Changing attributes of a table view cell

Change the **Style** in the attributes inspector to Subtitle, as shown in [Figure](#). This will let us add a description label to our cell:

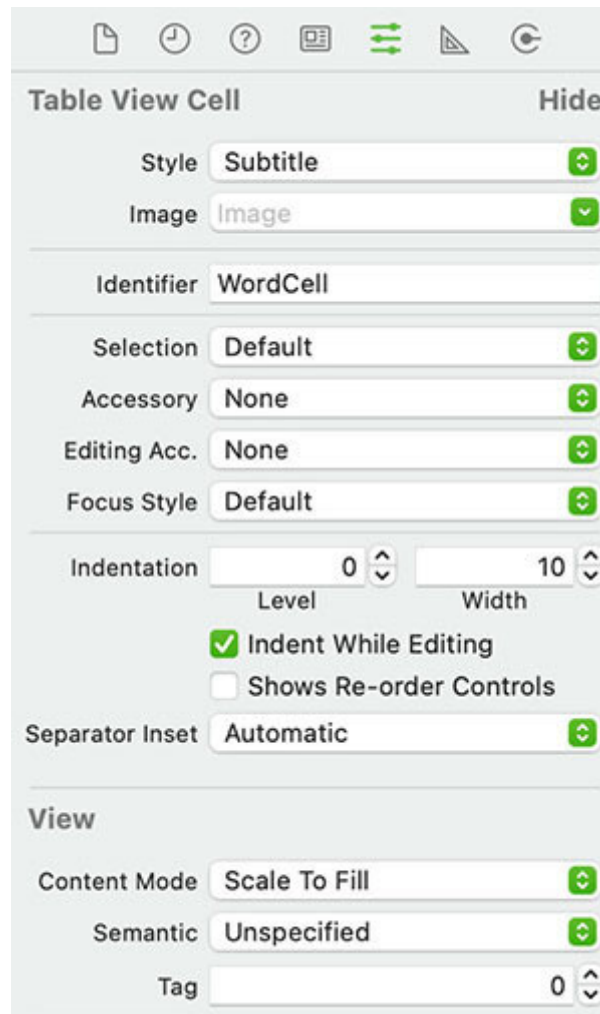


Figure 8.5.2: *Different attributes of a table view cell*

Next, click on the table view and check the attributes inspector. Refer to [Figure](#) The **Content** is nothing but the type of cell we are going to display. Here, we have a single cell which is going to repeat itself; thus, it is dynamic. Static prototypes are used when all your cells have to be defined in the storyboard and there is no code to source them.

Next is the This can be grouped and plain. Grouped is when you have multiple sections. Then, there are other attributes like

Separator, etc. Try to play around with them later. You can even set the *estimated row height* for the table view cell, in the size inspector or set it to Automatic:

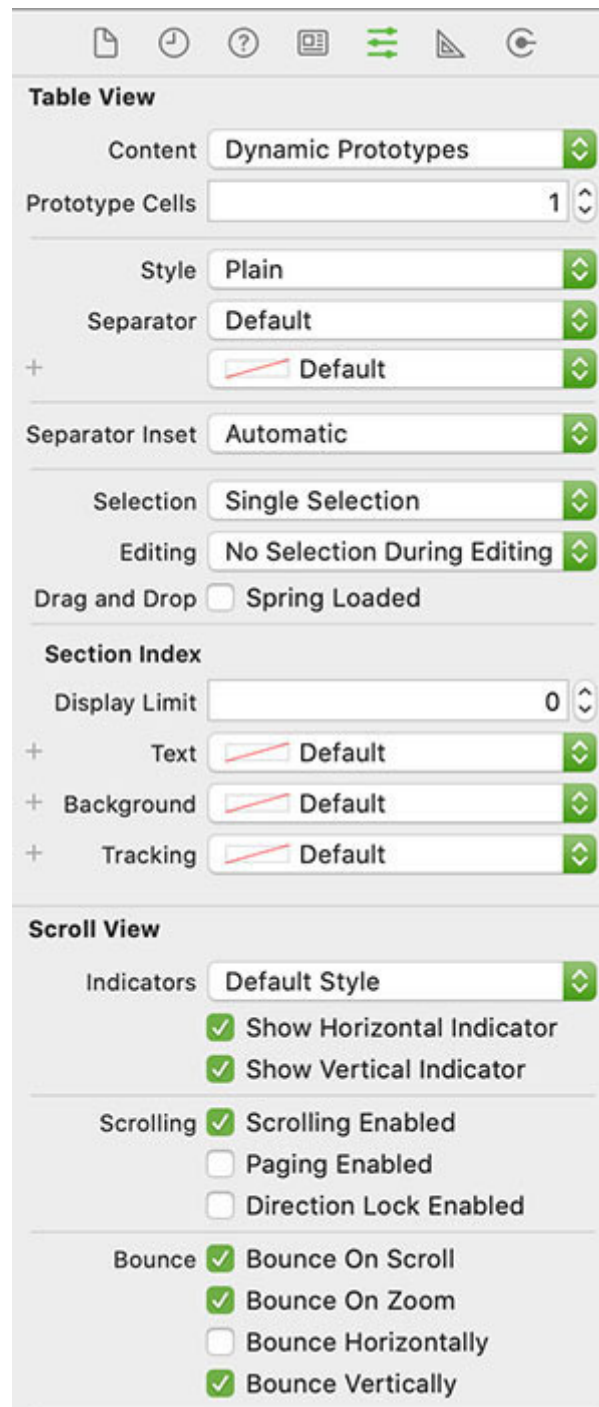


Figure 8.6: Different attributes of a table view

We then go back to our code to modify our **cellForRow** method. This is how the code should look after the modifications:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
let cell = tableView.dequeueReusableCell(withIdentifier: "WordCell",
for: indexPath)
return cell
}
```

For every row, we will create a new cell which is a For better memory management, cells which are not on the screen are dequeued and reused to display new cells. This is done very efficiently, so you do not see any difference.

Let us test if our code works. Return a random count in the **numberOfRowsInSection** method:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
return 2
//return self.wordsArray.count
}
```

Next, add a test text to the cell as follows:

```
cell.textLabel?.text = "test"
```


Then, run the code. You will see nothing. Why? Remember in the delegate design pattern, we learned that there is a **delegate** class which implements the protocol. Here, the **delegate** class is not a separate class, but the view controller itself. So, we need to assign that relationship in the Add the following two lines in the

```
wordsTableView.delegate = self  
wordsTableView.dataSource = self
```

Once you run the app after adding the preceding lines, you will see a table view similar to [Figure](#)



Figure 8.7: Testing list view

Creating the second view

We have created our first view. Now, we will try to show the details of each row in the second view.

We will reuse the same view to add new words. Follow the steps below to reuse the same view:

Our table view is ready and waiting for data. So, let us add some words in our table view. We will start by creating a model for Here, is a sample **Word** struct:

```
struct Word {  
    var word: String!  
    var translation: String?  
    var example: String?  
    var image: UIImage?  
}
```

Next, we will create another view controller which will help us enter a new word. You can call it **SecondViewController** for simplicity. You know how to design a view. You can make it as fancy as you want. I have chosen a simple format as shown in [Figure](#). On the top, there is a button super-imposed on an image. Don't forget to make the necessary **IBOutlet** and **IBAction** connections in the Swift file:

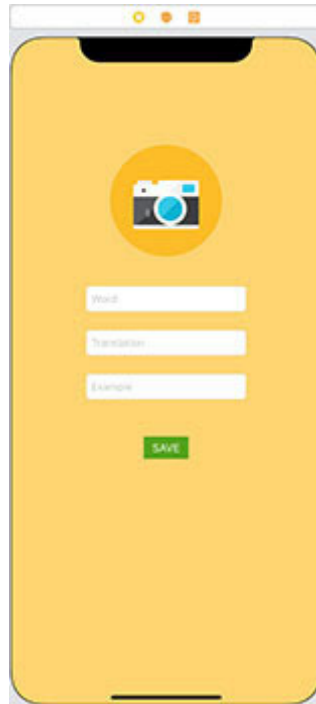


Figure 8.8: *Detail screen*

Next, add a bar button item on the navigation of your ViewController. Change it to system **Add** as shown in [Figure](#)

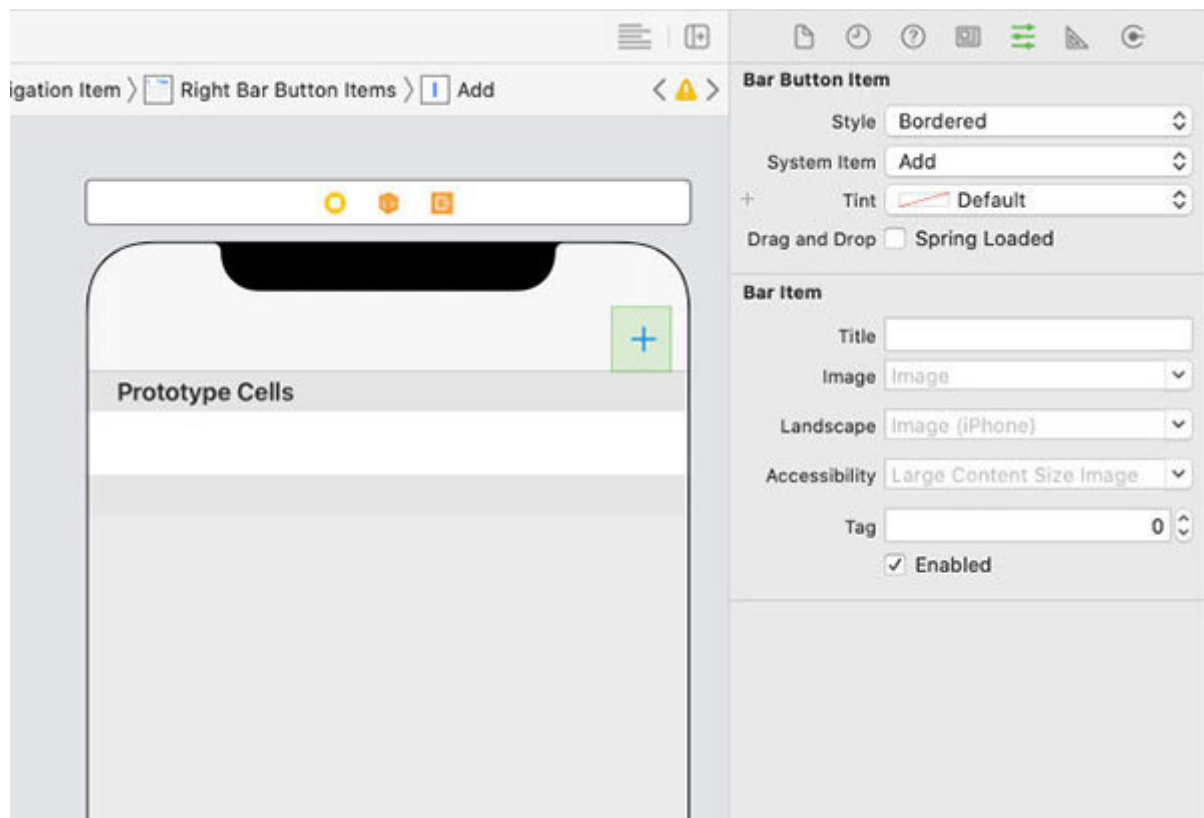


Figure 8.9: Adding a bar button

Then, make a segue from this bar button item to our Give the segue an identifier, Run the app and check the navigation.

You can even add titles to each view controller. Our next step is to create new words and show them in our table view.

Adding new words

We have created a view to add new words, but it lacks functionality. Let us fix this.

Follow the steps below to add new words functionality:

We will start by adding new words to our word bank. For this, we will use the delegate design pattern. When we add a new word in the view, we need to send this word to the controller. Here, we will add it to our word bank. So, we will first create a protocol. You can create a class and define your protocol:

```
protocol WordDelegate {  
    func didSaveWord(word savedWord: Word)  
    func didUpdateWord(word updatedWord: Word)  
}
```

The next step is to make our ViewController conform to this new protocol. This is how your code should look after the changes. You need to add the **WordDelegate** to the list of protocols the **ViewController** already conforms to. As soon as you do this, the compiler will throw an error and you will have to implement the protocol stubs:

```
class ViewController: UIViewController, UITableViewDelegate,  
UITableViewDataSource, WordDelegate
```

In the delegate design pattern, you will see that we have the protocol and the class acting as the delegate. We now need the class needing a delegate. Create an instance of the delegate in the **SecondViewController** as follows:

```
var delegate: WordDelegate?
```

The last link is to tell the **SecondViewController** that the class acting as the delegate is the ViewController. We can do this while we prepare our segue

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?)
{
    if segue.identifier == "addNewWord" {
        let secondView = segue.destination as! SecondViewController
        secondView.delegate = self
    }
}
```

Let us now test our code. First, change the **numberOfRowsInSection** to return the **wordsArray** count:

```
func tableView(_ tableView: UITableView, numberOfRowsInSectionSection: Int) -> Int {
    return self.wordsArray.count
}
```

If not done already, change the **wordsArray** to an array of words:

```
var wordsArray: [Word] = []
```

Then, we need to send our new word to the delegate from the
For this, you will need to modify the **saveButtonClicked** method:

```
@IBAction func saveButtonClicked(_ sender: Any) {  
    if let delegate = self.delegate {  
  
        let savedWord = Word(word: self.wordTextField.text, translation:  
            self.translationTextField.text, example: self.exampleTextField.text,  
            image: nil)  
        delegate.didSaveWord(word: savedWord)  
    }  
    self.navigationController?.popViewController(animated: true)  
}
```

Our next step is to process this new word in the class which acts
as the delegate. Here, it is the ViewController. We already have
the protocol stubs for We need to just modify the **didSaveWord**
method as follows. Here, we will add the new word to the list
and then reload our table view to show the new word:

```
func didSaveWord(word savedWord: Word) {  
    self.wordsArray.append(savedWord)  
    self.wordsTableView.reloadData()  
}
```

To test our code, we need to modify our method to show some
more data:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
let cell = tableView.dequeueReusableCell(withIdentifier: "WordCell",
for: indexPath)
cell.textLabel?.text = wordsArray[indexPath.row].word + " | " +
(wordsArray[indexPath.row].translation ?? "")
cell.detailTextLabel?.text = wordsArray[indexPath.row].translation

return cell
}
```

Run the app and test the code.

[Adding images](#)

One of the key features of the app is that you can associate an image with the word. Capturing image is not straightforward in iOS.

Apple provides us with a controller to see our gallery and then choose from it. Let us implement this:

We will work on images now. We must capture an image which suits our word. For this, we will need the Add the following code to your **imageButtonClicked** method:

```
@IBAction func imageButtonClicked(_ sender: Any) {
    let pickerController = UIImagePickerController()
    pickerController.delegate = self
    pickerController.allowsEditing = true
    pickerController.mediaTypes = ["public.image", "public.movie"]
    pickerController.sourceType =
        UIImagePickerController.isSourceTypeAvailable(.camera) == true ?
        .camera : .photoLibrary
    self.present(pickerController, animated: true, completion: nil)
}
```

This is the code to present an image picker. If you have a device, you can open a camera. As soon as you write this code, you will

see that the compiler throws an error. Change the code so that the **SecondViewController** conforms to

Then, you will need to implement the protocol stubs for

```
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey
: Any]) {
}
```

```
func imagePickerControllerDidCancel(_ picker:
UIImagePickerController) {
}
```

Let us do a test run. Open the **SecondViewController** and then click on the camera button. You will see something as shown in [Figure](#) You can click on an image and then select But nothing will happen. This is because we haven't written the code to use the captured image. For this, we need to use the delegate methods we have implemented.

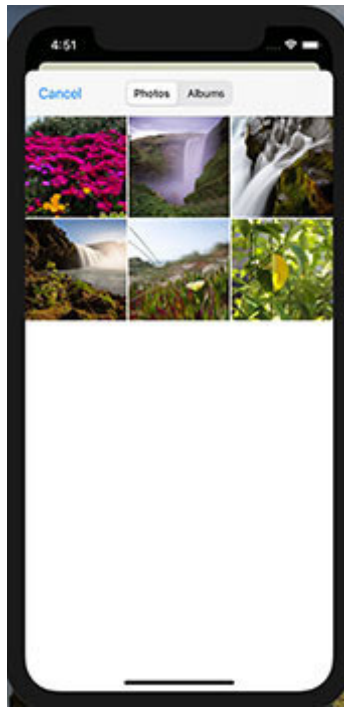


Figure in action

Here is how your code will look like once you change your delegate methods. It's very easy to understand. We use the info dictionary to extract the image from it using the appropriate key. We then assign the image to our image view.

```
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [UIImagePickerController.InfoKey
: Any]) {

let capturedImage =
info[UIImagePickerController.InfoKey.editedImage]
as? UIImage
if let image = capturedImage {
wordImageView.image = image
}
dismiss(animated: true, completion: nil)
```

```

}
func imagePickerControllerDidCancel(_ picker:
UIImagePickerController) {
dismiss(animated: true, completion: nil)
}

```

Our next job is to send the image to the ViewController. Change your **saveButtonClicked** method to the following:

```

let savedWord = Word(word: self.wordTextField.text, translation:
self.translationTextField.text, example: self.exampleTextField.text,
image: wordImageView.image)
delegate.didSaveWord(word: savedWord)

```

The last bit is to change our cell code to show the captured image. Here is how that can be done:

```

func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
let cell = tableView.dequeueReusableCell(withIdentifier: "WordCell",
for: indexPath)
cell.textLabel?.text = wordsArray[indexPath.row].word + " | " +
(wordsArray[indexPath.row].translation ?? "")

cell.detailTextLabel?.text = wordsArray[indexPath.row].example
if let image = wordsArray[indexPath.row].image {
cell.imageView?.image = image
} else {
cell.imageView?.image = UIImage(imageLiteralResourceName:
"Camera")
}
}

```

```
return cell  
}
```

Run the code and play around with your app!!

Updating the app

As mentioned earlier, we will reuse our existing Add Word view to Update the word. Here is the step-by-step implementation:

The last part of our app would be to update an existing entry. For this, we will use the This delegate is called when we click on a row. Our task is to send the selected row to the

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
}
```

First, create a new segue and name it Next, call it from the

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
self.performSegue(withIdentifier: "updateWord", sender:
wordsArray[indexPath.row])
}
```

Create a variable word in the

Next, update your prepare segue method. Pass the word to update as follows:

```

override func prepare(for segue: UIStoryboardSegue, sender: Any?)
{
    if segue.identifier == "addNewWord" {
        let secondView = segue.destination as! SecondViewController
        secondView.delegate = self
    } else if segue.identifier == "updateWord" {

        let secondView = segue.destination as! SecondViewController
        secondView.delegate = self
        secondView.word = sender as? Word
    }
}

```

In the update your **saveButtonClicked** method to accommodate the changes for update as follows:

```

@IBAction func saveButtonClicked(_ sender: Any) {

    if let delegate = self.delegate {
        if word != nil {
            word?.word = wordTextField.text
            word?.translation = translationTextField.text
            word?.example = exampleTextField.text
            word?.image = wordImageView.image
            delegate.didUpdateWord(word: word!)
        } else {
            let savedWord = Word(word: self.wordTextField.text, translation:
            self.translationTextField.text, example: self.exampleTextField.text,
            image: wordImageView.image)
            delegate.didSaveWord(word: savedWord)
        }
    }
}

```

```
}  
self.navigationController?.popViewController(animated: true)  
}
```

Lastly, update your **didUpdateWord** method implementation in the ViewController. Remember structs are value types, so you must update your array.

Now, launch the app and give it a test run.

Making the app better

There are many ways in which the app can be made better. You can even add more features to it. The purpose of this app was to give you a brief introduction of the table view and revision of navigation. Here are some ways you can work on making the app better:

Change the UI. You can start by making the UI more attractive. You can create your own custom cell and modify its UI the way you want.

A missing element in the preceding app is updating of the word. Try to solve the problem of updating the word. Passing the **indexPath.row** or filtering the array by adding an index.

You can also add validations on the button. If some fields are not filled, you can show an alert and not save blank entries.

You can write the code to change the name of the save button depending on whether we are Adding or Updating a word.

You can optimize the prepare segue method to combine the two if conditions.

Lastly, you can try to persist the data. But this is out of the scope of this book.

Conclusion

We learned about protocol-oriented programming in this chapter. This is a new programming paradigm pushed heavily by Swift. We also learned about the delegate design pattern and a very important UIKit component, the Table View Controller. We also built a simple but interesting app to help you understand the delegate design pattern. This was the last app we built, using imperative coding. In the next chapter, we will have a brief introduction of memory management and error handling.

Key learnings

We started the chapter by learning about protocols.

We then saw how to define a protocol. We saw how to satisfy the different property and method requirements for a protocol. We even saw how to write initializers in protocols.

Then, we started to study Protocol-oriented Programming. We saw how it can be an alternative to the multiple inheritance.

We learned that protocols can be used as types.

We also saw how to extend protocols and use protocols conforming to other protocols.

After learning so much about protocols, it was easy to understand the delegate design pattern.

We saw how a delegate is nothing but a protocol.

We completed the chapter by building a Word Bank app, which helped you get a better understanding of the delegate design pattern.

Test your knowledge

Why is it difficult to keep track of multiple instances in object-oriented programming?

Because classes are reference types and they can be wrongly referred somewhere else.

What is one of the biggest problems of single inheritance?

The superclass can get larger and complex, and all the subclasses must bear the cost of initialization.

How can protocols be used as types?

They can be passed as parameters. This makes it possible to pass any type which conforms to the said protocol.

Which keyword makes sure that all the subclasses of the class conforming to a protocol use the same initializer?

required

What are static prototypes?

Static prototypes are used when all your cells must be defined in the storyboard and there is no code to source them.

What are the two protocols necessary for table view implementation?

UITableViewDelegate

UITableViewDataSource

Which delegate is used to pick images from UIImagePickerController?

UIImagePickerControllerDelegate

Which method is used to refresh the contents of the table view?

reloadData

Which navigation controller method is used to move back to the root view controller?

popViewController

Error Handling and Debugging

In this chapter, we will take a break from building an app. This might seem a bit boring, but it is a really important chapter. If you pay attention, you will make fewer mistakes in your code. When you start working in a company, you generally start working on an existing project. In such cases, it is really important to jump on a running train and understand the code as soon as possible. This is when you will need the most important tool for a developer, the debugger. We will understand the Xcode debugger and see how we can customize it to our own benefit.

Structure

In this chapter, we will cover the following topics:

Memory management

Optional chaining

Error handling

Debugging

Logging

Objective

The objective of this chapter is to get you used to with the **debugging** tool. This chapter will be very useful in real-world situations. We will learn about **memory management** and **error**. Memory management is very important when working with data heavy applications.

Memory management

Swift *manages* the memory of your code. So, you do not have to worry about tracking and managing your instances. Swift uses **Automatic Reference Counting** to manage the deallocation of the instances created. As seen earlier, classes are references types, so ARC concerns mainly the classes. Structures and enumerations are value types and hence are never passed by references or stored. So, let us see how ARC works.

Automatic Reference Counting

In simple words, when an instance of a class is being used, ARC stores its properties and values in memory. Once it sees that it's no more being used, it deallocates the instance, thus freeing up memory every time a chunk of memory is allocated and deallocated. In most cases, ARC works fine to track and manage your memory, hence the name. But there are times when it needs to understand the relationships of your instances in order to manage memory better.

So, how does ARC keep track of this? It does this by **reference**. Every time we assign a variable to an instance, the reference counter is incremented by 1 or the instance's retain count is up by 1. When a variable is not referencing to an instance, this retain count reduces by 1. When this counter is 0, it means no other variable is referencing to this instance and the instance is then removed from the memory. All this tracking happens at the compiler level, and you just need to make sure that your instances are not referenced when not in use.

For this tracking to happen, whenever you use a class instance, i.e., assign it to a property or variable or a constant; the property, variable, or instance makes a reference to this instance. This reference is called a **strong**. It is a strong reference because it does not allow the ARC to deallocate as long as the reference exists. Hence as a developer, it is really important that you specify clearly the relationship between instances of classes. Otherwise,

your instances might always have a strong reference and never get deallocated. Let us see ARC with examples.

The regular functioning of Open a new playground and create a class, **Student**:

```
class Student {  
    let name: String  
    init(name: String) {  
        self.name = name  
        print("\(name) is being initialized")  
    }  
    deinit {  
        print("\(name) is being deinitialized")  
    }  
}
```

You should be able to easily understand the preceding code by now. It is a simple class with an initializer which will print a message whenever we create an instance of this class. You can create a new instance of the **Student** class:

```
var student1: Student?  
student1 = Student(name: "Tom")//Tom is being initialized
```

Now, you can forcefully set this instance to nil and see the deinit message being printed. As discussed earlier, the deinit cannot be called. It is automatically called by Swift. However, you can

override it and do some additional saving or set some references to nil before the class is deallocated:

```
student1 = nil//Tom is being deinitialized
```

So far so good. Then, where is the problem? Let us now understand the strong retain cycles problem.

Strong retain This is a problem of ARC, where two instances reference each other but are not being used. Thus, ARC thinks that they should be retained and never frees up their memory. We will get a better understanding of this problem by looking at some examples.

Remove the line where you set the **student1** instance to nil. Then, create another variable of **student2**, and then assign the **student1** to You should see the respective lines of code:

```
var student2: Student?  
student2 = student1
```

When we assign the new **Student** instance to the **student1** variable, a strong reference is created. Now, if we assign the **student1** variable to the another strong reference is created. ARC makes sure that the instance of Student is kept in memory because it has at least one strong reference. If you now force the first instance to nil, you will see that the log of deinit is not printed:

```
student1 = nil
```

While deallocating, ARC will not free up the memory until all the strong references are broken. You will see the deinit log only when we force both **student1** and **student2** to nil, thus ending the strong references:

```
student1 = nil //Tom is being deinitialized  
student2 = nil
```

We will create another class **Book** to get a better understanding of the strong retain cycles. Do not forget to add a book variable in your **student** class, like we have a student in Book. Your **book** class should look as follows:

```
class Book {  
    let name: String  
    var student: Student?  
    init(name: String) {  
        self.name = name  
  
        print("\(name) is being initialized")  
    }  
    deinit {  
        print("\(name) is being deinitialized")  
    }  
}
```

And your **Student** class should look as follows after adding the new variable for book:

```

class Student {
let name: String
var book: Book?
init(name: String) {
self.name = name
print("\(name) is being initialized")
}
deinit {
print("\(name) is being deinitialized")
}
}

```

Now, if you create two variables and initialize and deinitialize them, you will see the usual behavior: (Remember, we are able to assign nil only because the variables are optional.)

```

var student: Student?
var book: Book?

```

```

student = Student(name: "Mark") //Mark is being initialized
book = Book(name: "Swift in 30 days") //Swift in 30 days is being
initialized

```

```

student = nil//Mark is being deinitialized
book = nil//Tom is being deinitialized

```

Now, we will relate the **student** and the We will tell Mark that Swift in 30 days belongs to him and tell the book that Mark is

its owner:

```
book?.student = student  
student?.book = book
```

But now, if you try to set these instances to nil, you will not see the deinit logs being printed:

```
student = nil//Nothing happens  
book = nil//Nothing happens
```

This is because both **Student** and **Book** have strong references to each other. ARC will see these references and hence not free them from memory storage. In simple words, ARC will think that Mark still has a book so Mark cannot be nil, and at the same time, the book has an owner so it cannot be nil. This is what we call a **strong retain**

Weak How can we solve this? In the real world, you would tell ARC, *the book can exist even if it does not have an* This is exactly what we are going to do.

Now, change the **student** variable in the **Book** to a weak one, and then try the process again. Your **Book** class will look as follows after the modification:

```
class Book {  
    let name: String  
    weak var student: Student? //notice the weak word being added.  
    init(name: String) {
```



```
self.name = name
print("\n(name) is being initialized")
}
```

```
deinit {
print("\n(name) is being deinitialized")
}
}
```

By writing the **weak** word, we define the type of relation the book is going to have with a Student. Here, it only means that a book will be weakly attached to its student. So, if a student is set to nil, there won't be a strong reference cycle. Let us again try to set the instances to nil:

```
student = nil//Mark is being deinitialized
book = nil//Swift in 30 days is being deinitialized
```

Student now has a strong reference to but not the other way round. Hence, when you break the student reference, you also break the strong reference it has with As there are no more strong references left, you can easily deinitialize the book instance. You will now see the logs being printed:

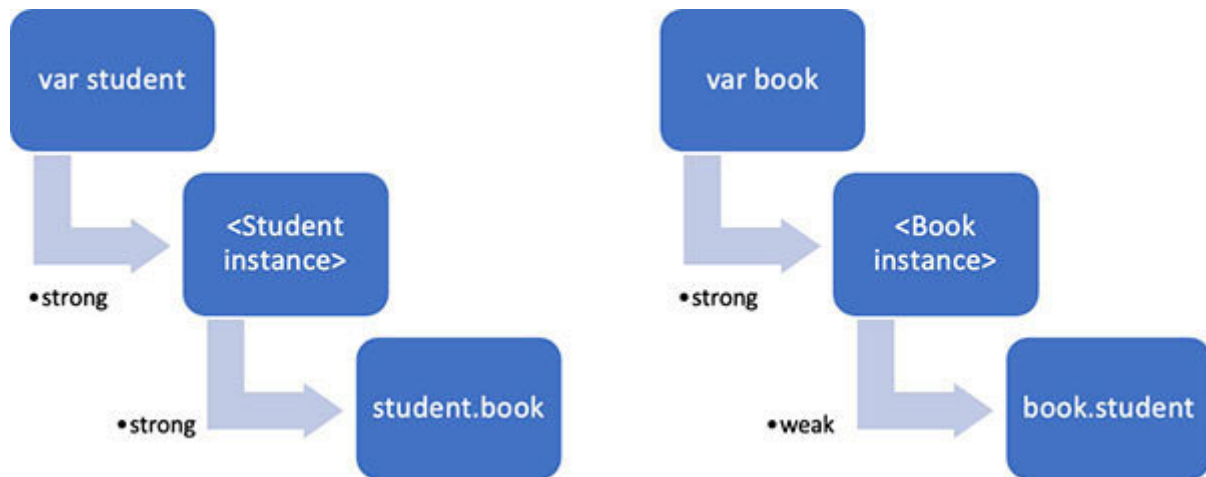


Figure 9.1: Strong vs weak reference

[Figure 9.1](#) will help you understand the strong references in a better way. Whenever you create a new instance and assign it to a variable, a strong reference is created. Hence, you will see a strong relationship between the student and the Instance>. It is the same with the book variable and instance>. But you will see that the student to book relation is a strong one, while the book to student is a weak one.

Unowned An optional reference to a class can also be marked as unowned. For ARC an unowned optional reference and a weak reference are similar. The only difference is that for an unowned reference, you are responsible to make sure it always points to a valid object. You are also responsible to set it to nil.

In simple words, if the other instance has a lifetime same as the first one, i.e., if the other instance cannot exist irrespective of the first one. Let us look at an example. Create two classes **Customer** and

```

class Customer {
  let customerId: String
  var order: PizzaOrder?
  init(customerId: String) {
    self.customerId = customerId
  }
  deinit {
    print("\(customerId) is being deinitialized")
  }
}

class PizzaOrder {
  let orderId: String
  var customer: Customer?
  init(orderId: String) {
    self.orderId = orderId
  }
  deinit {

    print("\(orderId) is being deinitialized")
  }
}

```

In the preceding example, a PizzaOrder can exist only if the customer orders one. So, as the customer will have a strong reference to a pizza order, this is also true inversely.

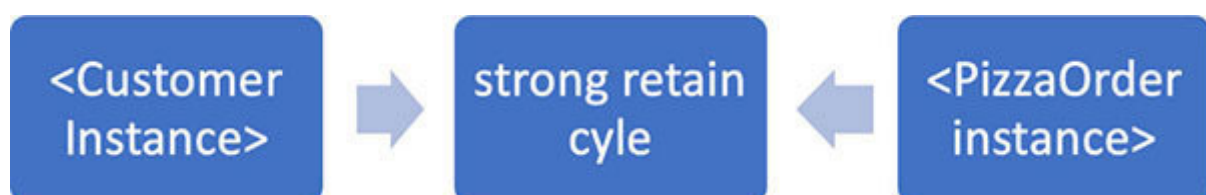


Figure 9.2: Strong retain cycle

As seen in [Figure](#) if both have strong references, we will have a strong retain cycle. This is when you can use the keyword `unowned` instead of `weak`. You can mark `customer` as `unowned` instead of `weak`:

```
class PizzaOrder {  
    let orderId: String  
    unowned var customer: Customer?  
    init(orderId: String) {  
        self.orderId = orderId  
    }  
}
```

An `unowned` reference is always expected to have a value. Hence, when we deallocate a the **PizzaOrder** associated to it is also deallocated. Thus, breaking the strong retain cycle. [Figure 9.3](#) shows the graphical representation of the relationships:

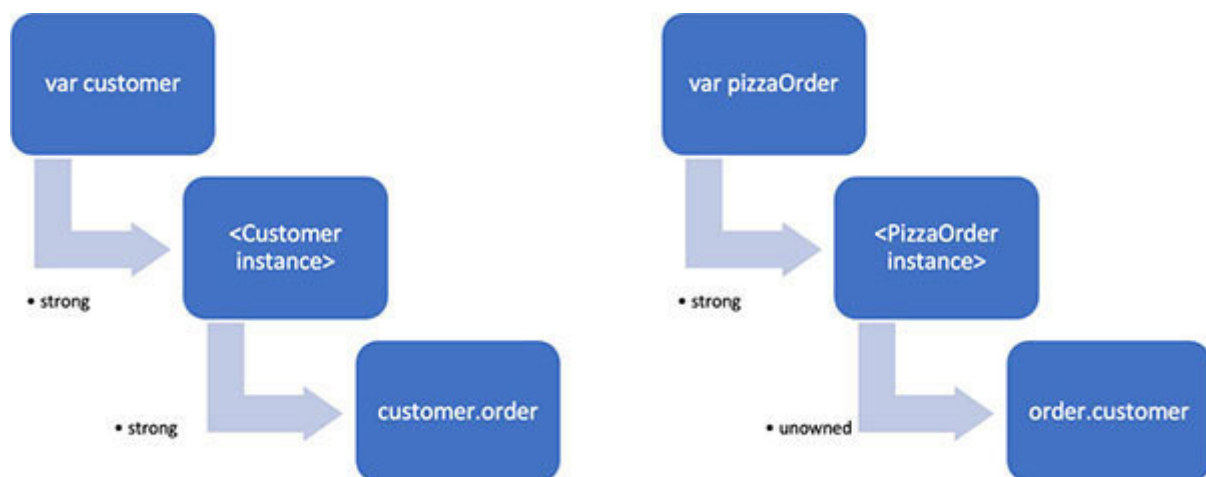


Figure 9.3: *Unowned reference*

Optional chaining

We have already seen the optional type and how we can *unwrap* an optional. We even created optional properties for our structures and classes. **Optional chaining** allows you to check whether a property, method, or subscript is optional. This is really helpful as we can avoid errors and check for an optional that might be `Optional`. Optional chaining is a very good alternative to forced unwrapping. Remember our **Person** struct. Let us add another variable to it, a pet! We have added the **Animal** class as a relation to our **Person** struct. Here is how the animal class will look like:

```
class Animal {  
  var color: String!  
  var description: String!  
  func makeSound() -> String {  
    return "I am an animal"  
  }  
}
```

Here is how the **Person** struct will look like with the new relationship:

```
struct Person {  
  var firstName: String?  
  var lastName: String?  
  var phoneNumber: String?
```

```
var pet: Animal? //new relation
}
```

The following example will help you understand optional chaining:

```
Let person = Person()
```

```
if let petColor= person.pet?.color {
print(petColor)
}
```

In the preceding code, the **petColor** will never be printed because we have not assigned a pet to this new instance of `Person`. But instead of unwrapping the we first called to check whether **pet** was assigned. When the code finds that the **pet** is nil, the color will also be nil. **color** is non-nil, but because **color** is queried through an optional chain, the call will return `String?` instead of `String`!

This chaining can be on multiple levels and works even for methods and subscripts:

```
if let sound = person.pet?.makeSound() {
print(sound)
}
if let sound = person.pet?.someOtherRelation?.makeSound()
{//multi-level relation
print(sound)
}
```

In error handling, optional chaining is really useful. The code becomes quite readable. When there are multiple relations, optional chaining is handy for quick querying.

Tip: If the type you are querying through optional chaining is not optional, it will become optional. But if the type you are querying is optional already, it will not become more optional. It will keep its optional behavior.

Error handling

To err is Humans make mistakes, so a human-written code is no exception to that. In real-world applications, even though we try to cover all the scenarios, there are times when our code will encounter problems. **Error handling** is the way in which our code will respond to errors and try to recover from it, without crashing the application. App crashes are very unpleasant, so they can lead to loss of data and even bad reviews for your app.

We already saw how optionals can be useful to handle the absence of values. Because Swift adopts the *value-based coding* even errors are represented by values of types. These types conform to the Error protocol. This is an empty protocol and serves as an indicator such that the type conforming to it will be handling errors. Here is an example:

```
enum SaveToFileError: Error {  
    case invalidFileFormat  
    case locationNotFound  
    case fileTooBigError  
}
```

In the preceding example, our **enum SaveToFileError** has a list of errors that might occur.

Whenever our code will encounter one of these errors, we need to handle it. The error can either be handled right in the function where it occurs, or we can propagate it through our code to another function, where it will be taken care of. Swift proposes four ways for error handling. We will look at them one by one.

Throwing functions

A very important keyword in error handling is `throws`. When we need to propagate errors, i.e., handle it in some other place, our function must be an error throwing function:

```
func saveToFile () -> Bool //cannot throw any errors
func saveToFile () throws -> Bool
```

In the preceding code, you can see two functions. One with a keyword **throws** and other without it. They are quite self-explanatory. If you need to handle the errors right in the function where it occurs, we can write a simple function. But if you need to send the error to another function where it will be handled, we add the **throws** keyword to our function. This tells Swift that an error is expected from our function.

This is usually done when we need to show the user an alert indicating that an error has occurred. We have already seen the MVC pattern and how the roles are divided. So, if your error occurs in the network part, the UI should be made aware. This is when the throwing functions are useful.

Using try-catch

Now, if we are *throwing* an error, then there must be another function which should *catch* it. This is when we use the traditional try catch, which we all know of. Here is the syntax:

```
do {  
  try throwing expression  
} catch pattern {  
  statements  
}
```

You can handle different errors by catching them. The caught error is then matched with the **catch** pattern and treated:

```
do {  
  try saveToFile()  
} catch SaveToFileError.invalidFileFormat {  
  print("Error : File format invalid")  
} catch SaveToFileError.fileTooBigError {  
  print("Error : The file you are writing is too big")  
}
```

In the preceding code, we try to call the **saveToFile** function. If it throws an error, we catch it and then print the respective error messages. If we need to handle this throw in another function, we can have multiple throws.

Note: When you throw an error, the control quits the function like a return statement. So, make sure you handle all the resources of the function before throwing the error. Here, for example, you will need to close the file before throwing the error.

Debugging

As no one is perfect, we tend to leave some bugs in our code, though unintentionally. If not bugs, we tend to forget to address some use cases. In such instances, we have data which our code cannot handle. This anomaly can occur deep in the code.

Debugging, like the word suggests, means removing the bugs. If you have followed the naming conventions and put logs at proper places, finding the *bug* can be really easily. As the bugs are manmade, the issues must also be removed by us. Apple has taken good care of this and Xcode provides us with a nice set of debugging tools to find our issues.

UI We will build a complex layout using many constraints. We even have to hide and show views depending on specific conditions. But sometimes this does not happen. Or sometimes, we just don't know why a button is suddenly bigger. This is when we can use the UI debugger.

Let us use one of the apps we worked and debug its UI. Follow the steps:

Open the *SimpleCalculator* app or any other app with a UI.

Run the app.

Then, click on this button , which is on top of the debugger pane. Please refer to [Figure](#) This button will allow you to debug your view hierarchy:

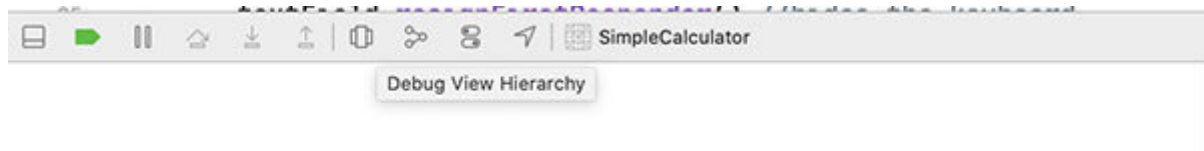


Figure 9.4

As soon as you click on it, you will see a replica of your screen in your editor pane. You will see a screen similar to [Figure](#)

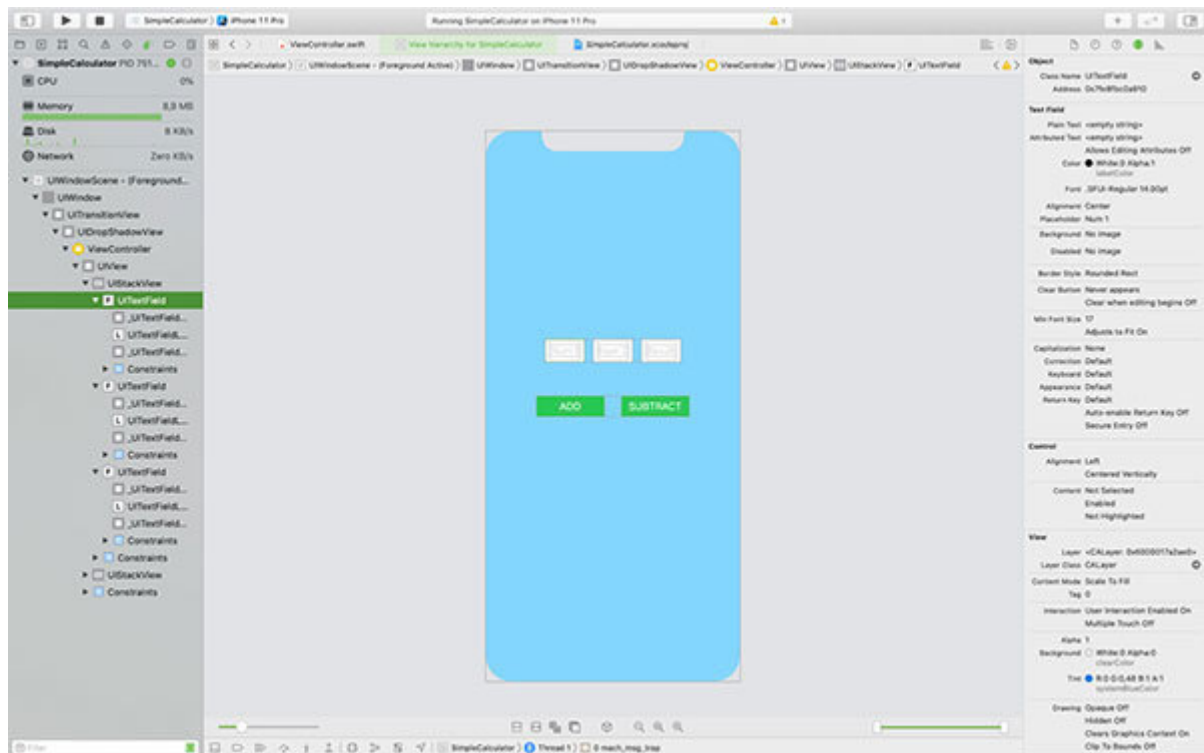


Figure 9.5: UI debug editor

In the navigation pane, you will see the view hierarchy. Please refer to [Figure](#) Observe the view hierarchy. It is the perfect way to understand how the different layers of views are added on top of each other. As you open and close the arrows, you can easily understand which are the subviews and which are the super views. At the same time, the sequence is important. Views higher up in the list are farther behind. For example, UIWindow is on the top, so it is behind everyone, i.e., it is the superview holding your first view controller.

This will give you a better understanding of how views work in iOS.



Figure 9.6: View hierarchy

In the middle or editor pane, you will see your UI, and you can even drag and move it at angles to see all layers. Refer to [Figure](#)

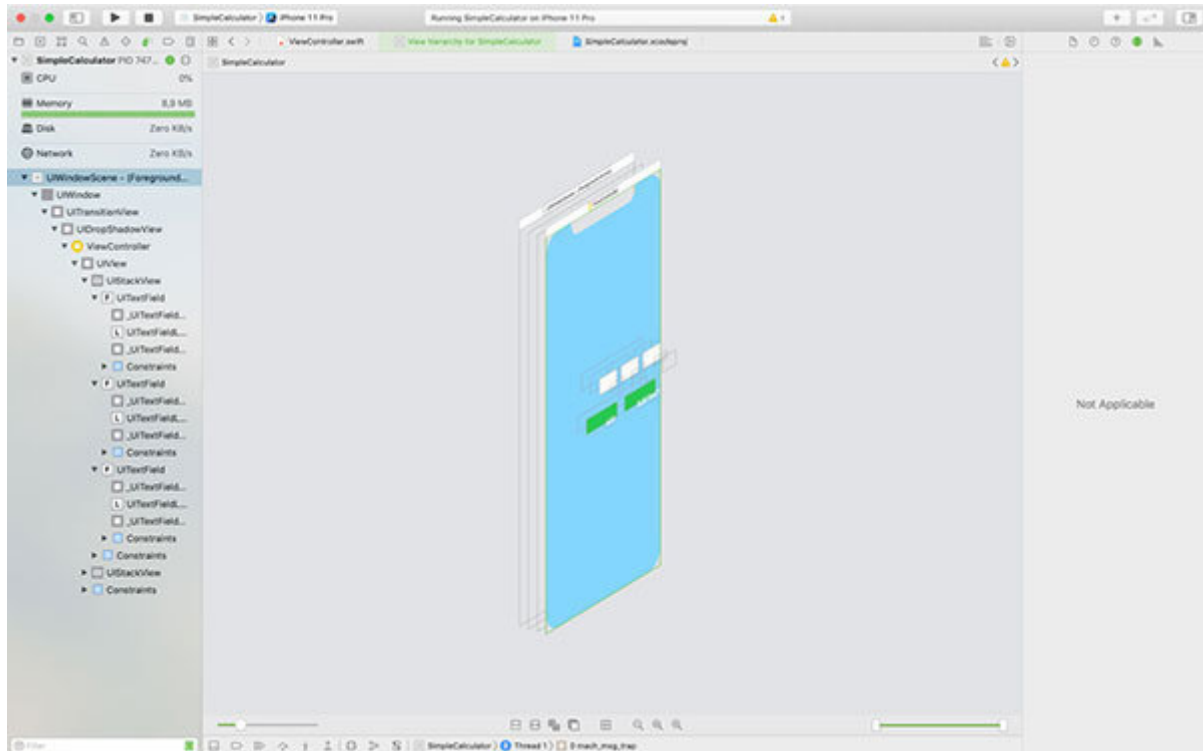


Figure 9.7: *Turning the views to see all layers*

When you click on any view element in the navigation pane, you should see its details in the inspector pane. But this time, these are the attributes affected at run time. You can use this information to find out why a view is not showing up, or why a button is big. Refer to [Figure](#)

As seen in [Figure](#) the inspector gives us a lot of run time information. We are looking at the information of a You will see all its properties. In the size inspector, you will even see the active constraints. As you will hover over these active constraints, you will see that the respective constraints get highlighted in the

view. This is highly useful when you activate and deactivate constraints for multiple conditions.

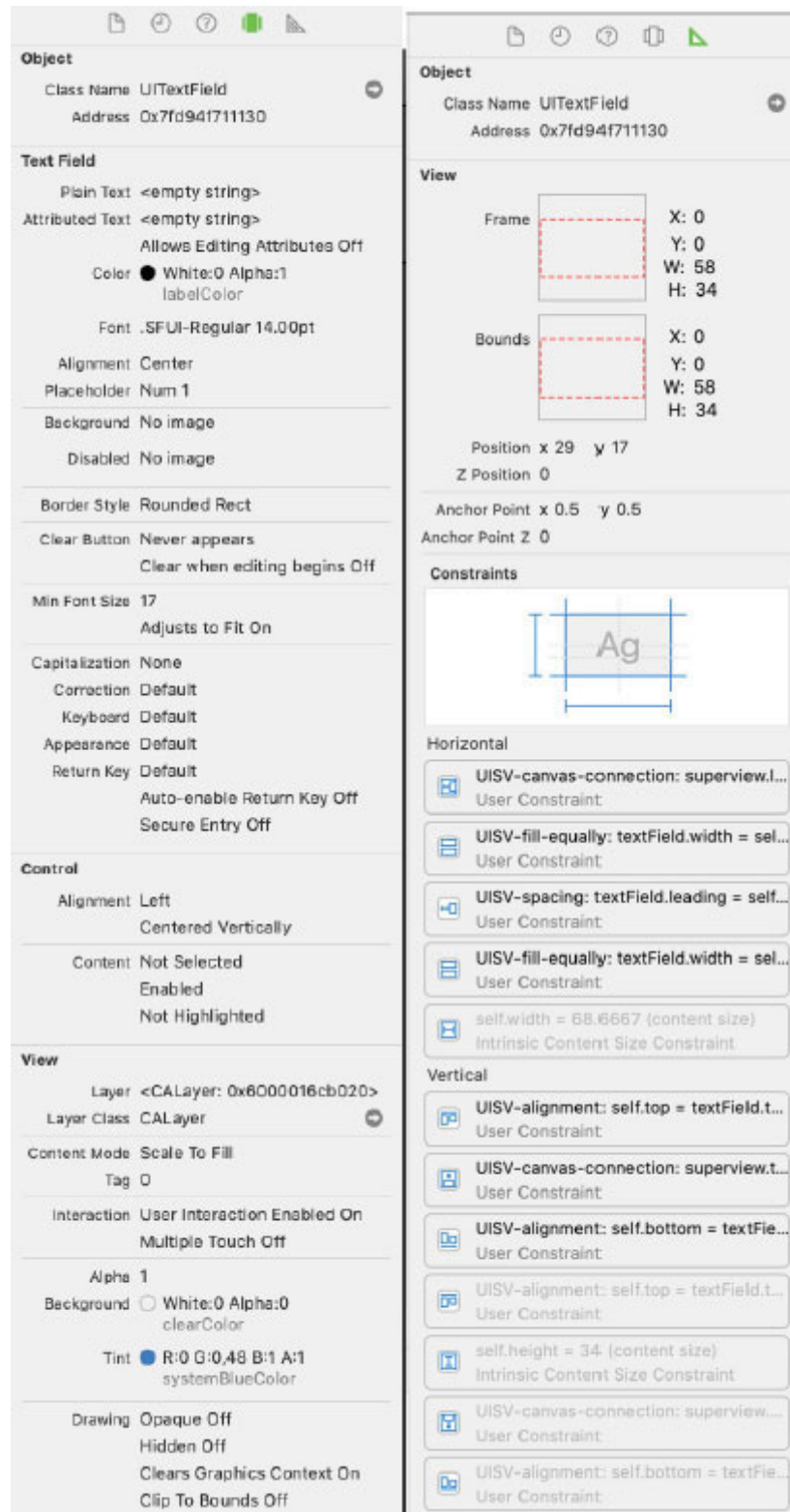


Figure 9.8: Run time information of view elements

Go ahead and check out the buttons which are on the top of the debugger pane. You will see each button helps you understand the UI in a better way as shown in [Figure](#)



Figure 9.9: UI debugging buttons

This exercise should also help you understand how UI works in the iOS apps. You can use this to find the problems in your UI.

Code We saw how to debug the UI, but there is a layer of data and business logic which can also have bugs. This is when we use the traditional code debugging. This is used when there are code anomalies. If you want to check a crash or a return value of a function, but how do we do this? Imagine this simple day-to-day scenario. You forget your keys while walking home. You go back again to the coffee shop you were in, and then slowly start tracing back your steps. You then go past the bus stop where you think would have gone and, on the way, you end up finding your keys. This is exactly how you debug. Like coffee shop and the bus stop, you *mark* points in your code through which your execution must have gone through. Then, your execution is *paused* at these points and you can then check the values and problems. Let us watch it in execution.

We will continue using our *Simple Calculator* app from [Chapter 5, *Optionals and*](#) Open the ViewController. We will put debug pointers at different points in our code and then make the code pause at these points. We can then execute the code step by step using the stepper. Let us understand how this works:

First, add a function in your code which returns an `Int`. Call it from the **addButtonClicked** function:

```
func debugTesting()->Int {  
    for i in 1...6 {  
        if i==4 {  
            return i  
        }  
    }  
}  
  
return 0  
}
```

Now, put a debug point, also called a breakpoint, on the line where you call the function. This can be done by just clicking on the left of the line number also called the **line**. Refer to [Figure](#)

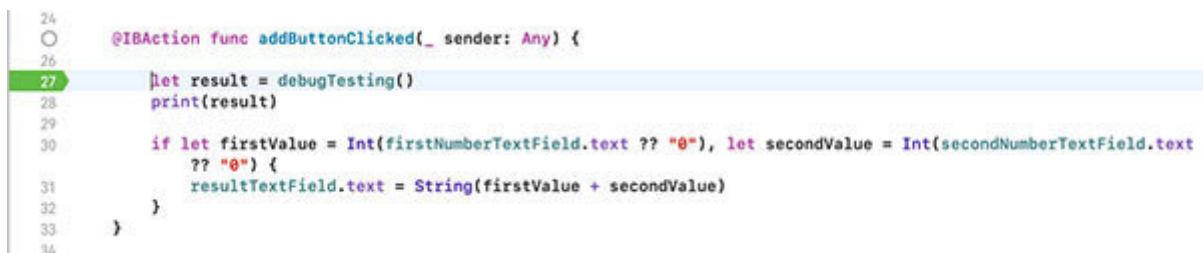


Figure 9.10: Debug point in the line gutter

You will see a green label. If you click on it again, it will be deactivated. An active debug point only means that your code execution will stop at that particular breakpoint.



Figure 9.11: *Active vs inactive breakpoint*

Tip: You can go to Xcode preferences -> Text Editing -> Line numbers. Check it to see the line numbers in your code. Line numbers are really helpful while debugging. It is easier to follow a stack trace when you have the line numbers.

Now, run the code and click on the **Add** button. Your code execution should now stop at the line where you have put your debugger point as shown in [Figure](#)

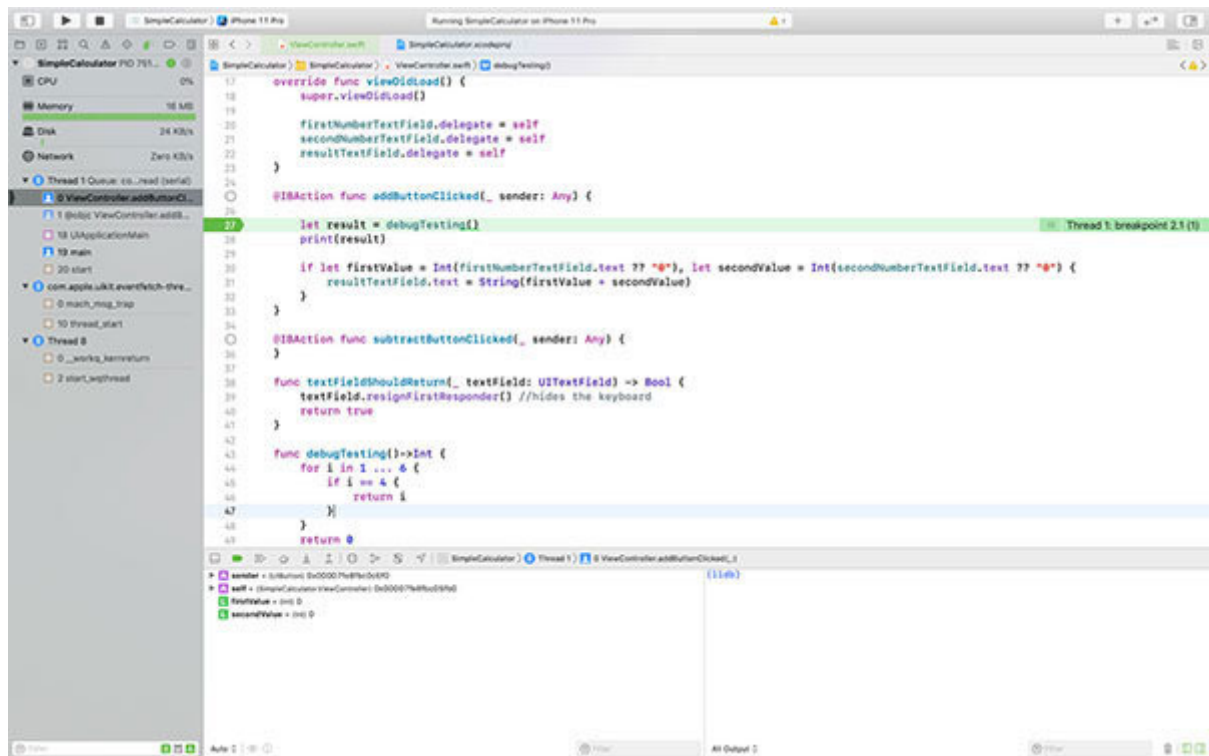


Figure 9.12: Execution stops at the breakpoint

Let us try to understand what is happening. In the navigation pane, you can see the memory consumption and all the threads running. You can also see the current stack trace in Thread 1. In the debug pane, you can watch the variables. In the editing pane, you see a green line at the debug point indicating that the execution has paused at this point.

Next, you can observe the debug control buttons on the top of the debug pane. As soon as your debugger point stops, the pause control changes to play. As soon as you click on either, you can pause or continue the execution of your code as shown in [Figure](#)



Figure 9.13: *Debugger pane buttons*

Let us understand the buttons one by one:

: The *debug* control, which is green and activated, helps to enable and disable the debugging of the entire code.

: The **pause/ play button** allows you to stop the main thread at any point, irrespective of whether you have put a debugger point or not. In case your debugger stops at some point and you want to move on to the next, you can click on the play button.

: The **step over** causes the debugger to execute the current line and move on to the next one.

: The **step in** button makes the debugger follow a function or a method at which it has stopped.

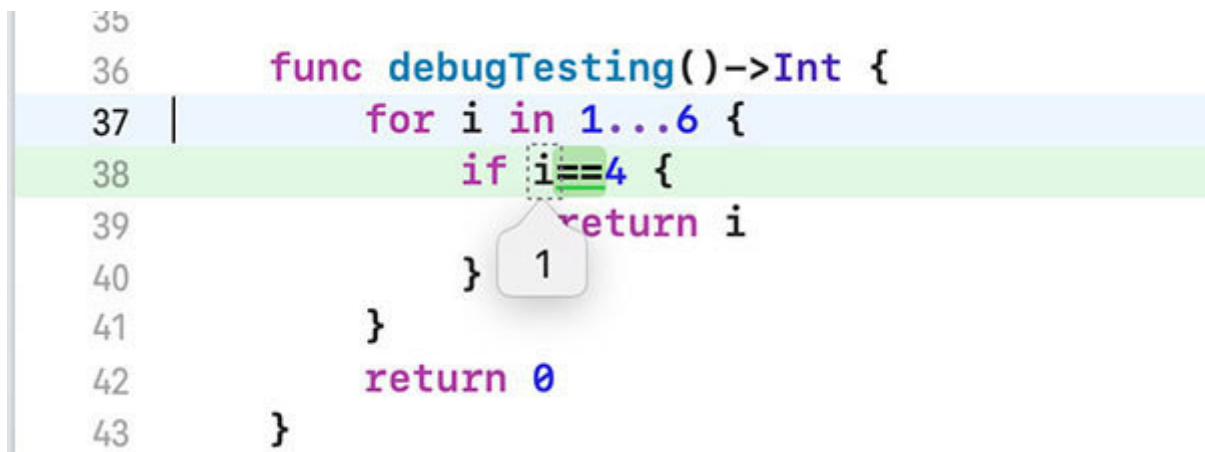
: The **step out** button does exactly the opposite of step in. It will step out of the function and the debugger will return to the line from where the method was called.

Using the steppers

Let us get back to our debugging. Now, click on the step in. At this point, you will see that the control moves to the

debugTesting function. This is exactly what the step into control does. It moves the control inside the function at the debug point.

Now, click on the step over button. This will take the control to the next line. You can hover over the variables to check their values at runtime. So, you can check the step-by-step execution of the for loop as shown in [Figure](#)

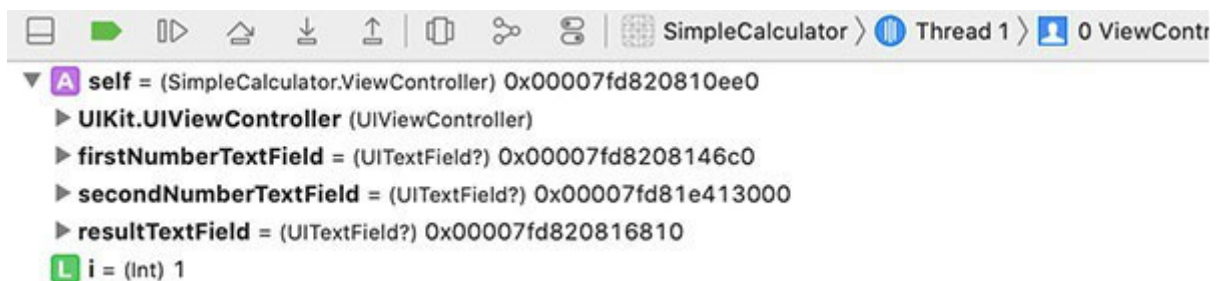


```
35
36     func debugTesting()->Int {
37         for i in 1...6 {
38             if i==4 {
39                 return i
40             }
41         }
42         return 0
43     }
```

A screenshot of Swift code in a debugger. The code defines a function `debugTesting()` that returns an `Int`. It contains a `for` loop from 1 to 6. Inside the loop, there is an `if` statement `i==4`. A tooltip is shown over the variable `i` in the `if` statement, displaying the value `1`. The code is highlighted with alternating light blue and light green background colors for each line.

Figure 9.14: Step by step execution debugging

The other way to check values is also in the debug pane on the left-hand side called the variable view as shown in [Figure](#)



A screenshot of the variable view in the debug pane. The top bar shows the current context: `SimpleCalculator > Thread 1 > 0 ViewContr`. The variable view lists the following variables:

- `self` = (SimpleCalculator.ViewController) 0x00007fd820810ee0
 - `UIKit.UIViewController` (UIViewController)
 - `firstNumberTextField` = (UITextField?) 0x00007fd8208146c0
 - `secondNumberTextField` = (UITextField?) 0x00007fd81e413000
 - `resultTextField` = (UITextField?) 0x00007fd820816810
- `i` = (Int) 1

Figure 9.15: Variable view in debug pane

At this point, if you click on the step out button, the control will move out of the **debugTesting** function. It will then return to the **addButtonClicked** function. This is how debugging works in brief.

Editing the debug point

If you right click on the debug point, you will see many options as shown in [Figure](#)

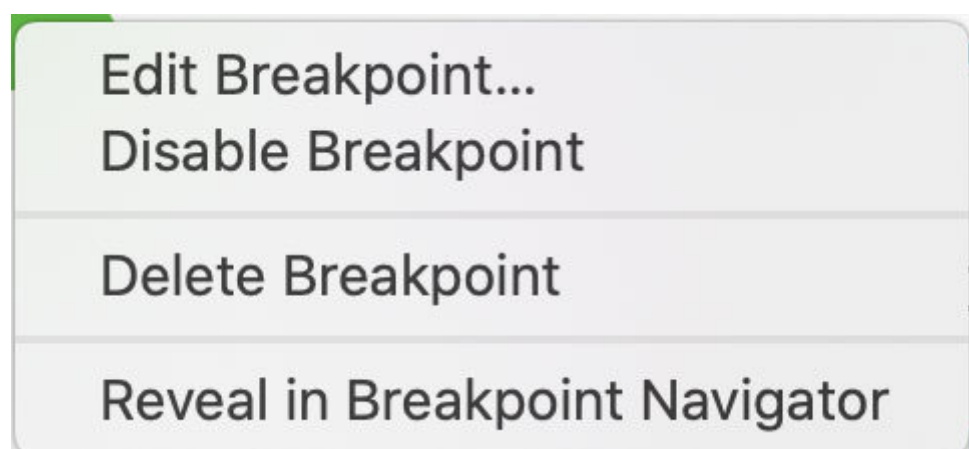
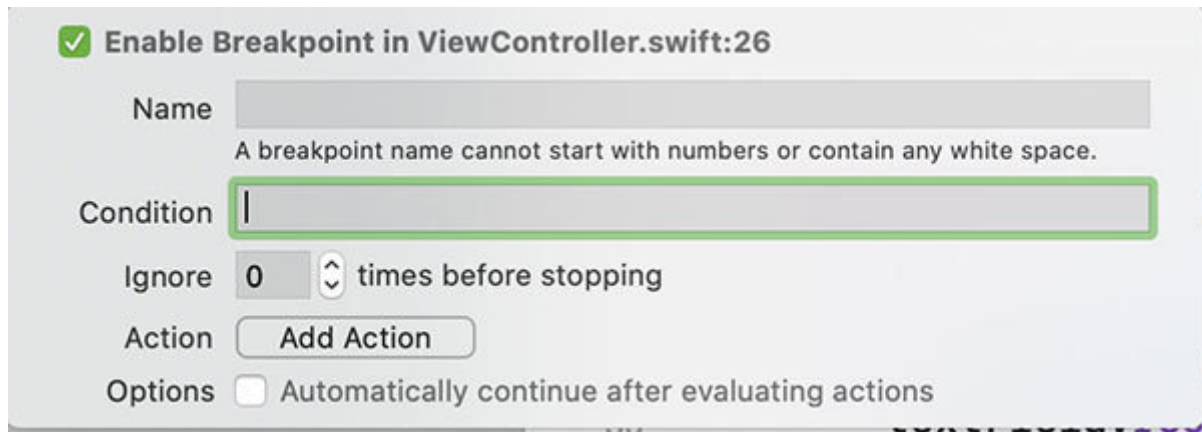


Figure 9.16: *Edit breakpoint menu*

If you click on the **Edit Breakpoint** option, you will see the window as shown in [Figure](#). Here, you can add condition like `if i == 3` and the debugger will stop when the condition is met. This can help you save time when you are debugging through long running loops. Or even if you want to stop at a line of code only when a condition is met.



The screenshot shows the 'Edit Breakpoint' dialog in Xcode. At the top, there is a green checkmark icon followed by the text 'Enable Breakpoint in ViewController.swift:26'. Below this, there are several fields and options: a 'Name' field with a placeholder text 'A breakpoint name cannot start with numbers or contain any white space.'; a 'Condition' field with a green border and a single character '|'; an 'Ignore' field with a value of '0' and a spinner icon, followed by the text 'times before stopping'; an 'Action' field with a button labeled 'Add Action'; and an 'Options' section with a checkbox labeled 'Automatically continue after evaluating actions'.

Figure 9.17: *Edit breakpoint options*

You can put multiple points in your code and watch them all at once when you navigate to the breakpoint navigator as shown in [Figure](#)

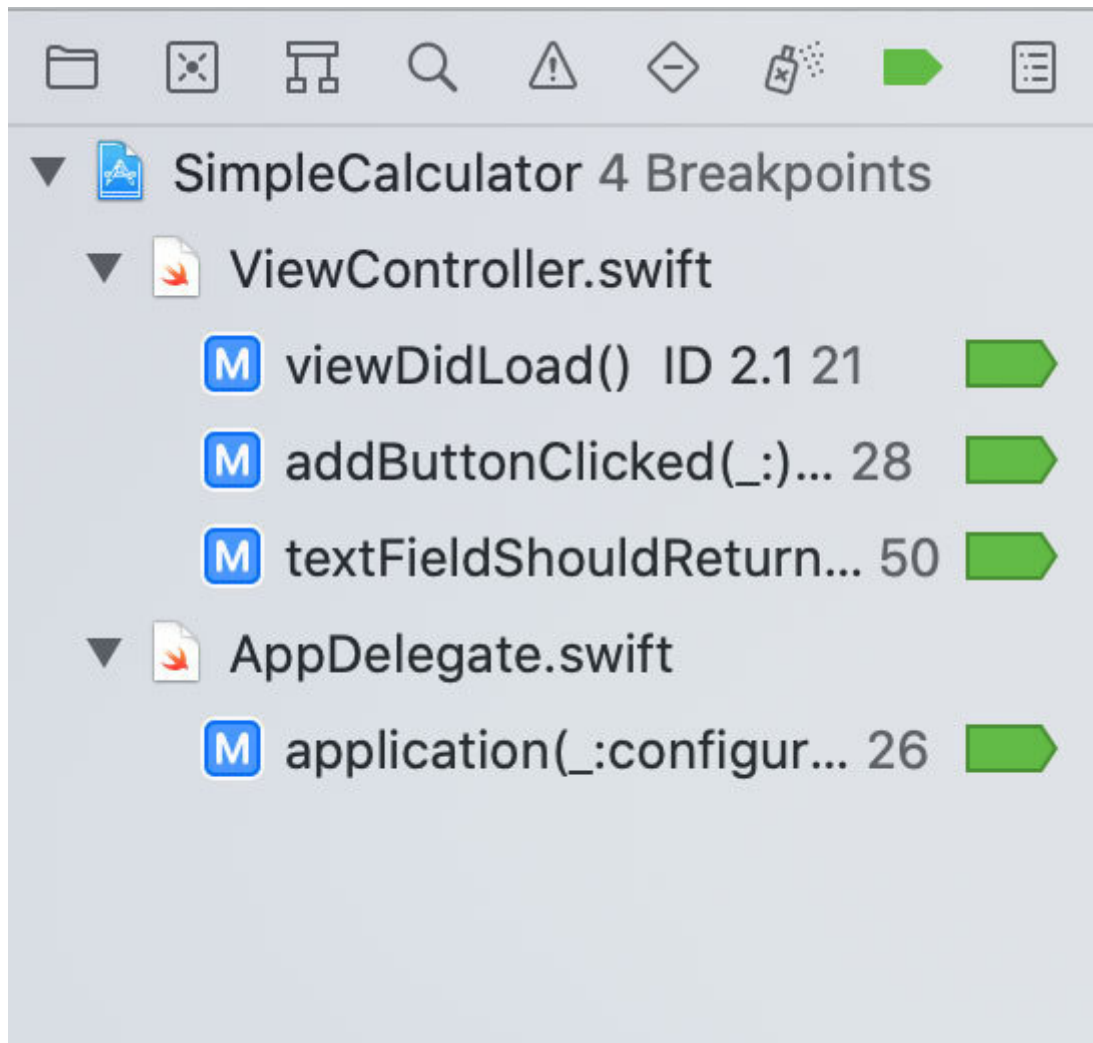


Figure 9.18: List of active breakpoints in navigation pane

Logging

Logging means recording the runtime behavior of your application. In the **try catch** section, you saw examples of code with print in the This is the simplest form of logging. You put print at different points to check the data. This record can be then persisted, saved, or sent to a server to find bugs or anomalies. As you saw in debugging, observing values can be achieved with the debug points too. But logging makes the functioning of your code quite visible. A well-logged app leaves behind a trail of information, which can then be used to trace a bug without even trying to reproduce it. For example, *File opened, Writing data, Data written, Closing file, Sending file, File* and so on. In these logs if you see an error at any point, you can figure out the problem right away.

You can even write your print statements to a file and then send it to a server to track the errors. But the files can get very huge over time. Also, writing to a file costs memory. Instead, there are many third-party analytics frameworks, which help to track the user activities. With Xcode 12, Swift has introduced the OS Framework, which can be used for logging.

OS With the OS framework, Apple aims at providing a lightweight logging system. This framework has many built-in features and is secure. The simplest way to import the OS log in your class is the statement. You can then start logging by using the following code:

```
os_log("Some message")
```

But during development, we often put logs at places where they are not necessary. Neither all logs need to be sent to the server nor all logs need to be. To overcome this problem, Apple introduced 4 types of logging levels:

This is useful when you are actively debugging. These logs are not saved. They are for pure debugging when you are actively working on the application. In priority level, they are the least important.

This is helpful for troubleshooting, but it is not essential. The Info messages are persisted only if they are generated right before the log collect command.

Notice The default used for troubleshooting. These logs are persisted depending upon the storage capacity.

The errors which we can see during execution. Like Notice messages, this log is too persisted depending on the storage capacity.

This is a bug in the program. These are the highest priority logs. They are persisted the longest.

The preceding messages are persisted in the order of least important to most important. The debug messages are never persisted. Once the storage has been reached, the logs are automatically purged. Fault logs are persisted the longest.

The performance of these levels is opposite of persistence, i.e., Debug being the most performant and Fault being the least performant. Though the OS framework offers very low overhead, persisting the logs comes at a performance cost. The syntax for logging with a type is as follows:

```
os_log(OSLogType, StaticString, CVarArg)
```

In the following example, we will log using a default **OSLog** instance. The type of the log is an error:

```
let logMessage: StaticString = "This is an example of a log message!"  
os_log(logMessage, log: OSLog.default, type: .error)
```

From iOS 14.x Apple wanted to provide a unified logging system and with the `Logger` struct we can do that. Here is an example of `Logger` used in the *Simple Calculator* app. As seen, we can have different categories for your subsystem. You can even customize the subsystems to get better results:

```
@IBAction func addButtonClicked(_ sender: Any) {  
    let logger = Logger(subsystem: "com.gaurang.SimpleCalculator",  
                        category: "addition")
```

```
if let firstValue = Int(firstNumberTextField.text ?? "0"), let
secondValue = Int(secondNumberTextField.text ?? "0") {
resultTextField.text = String(firstValue+secondValue)
logger.log("values added successfully")
}
}
```

Here is an example of Logger with the level of the log:

```
logger.log(level: .error, "Some message")
```

Logging Sometimes, there is data which is protected in the app. You want to log it, but do not want to display the info. You can then use the private and public keywords. Here is how the code looks like:

```
os_log("Second value is %{public}@", log: OSLog.default, type:
.info, secondValue)
os_log("Second value is %{private}@", log: OSLog.default, type:
.info, secondValue)
```

The second line will only log the word instead of the value of the variable

Tip: You can customize the configuration profiles to override the persistence behavior of each log level.

Using If you want to get the most out of the OS log, you should use it in combination with the Console app. Open the

Console.app on your mac machine. Its icon is shown in [Figure](#)

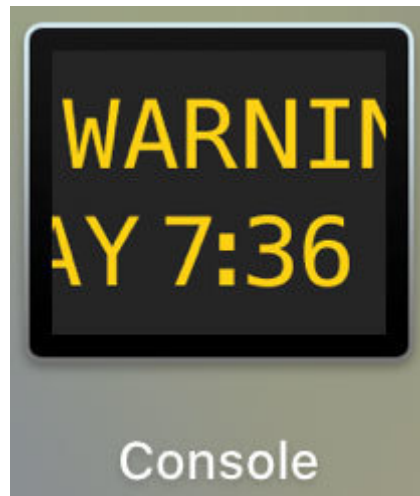


Figure 9.19: *Console app icon*

In the devices section on your left-hand side, you will see all the simulators and your devices connected to the machine:

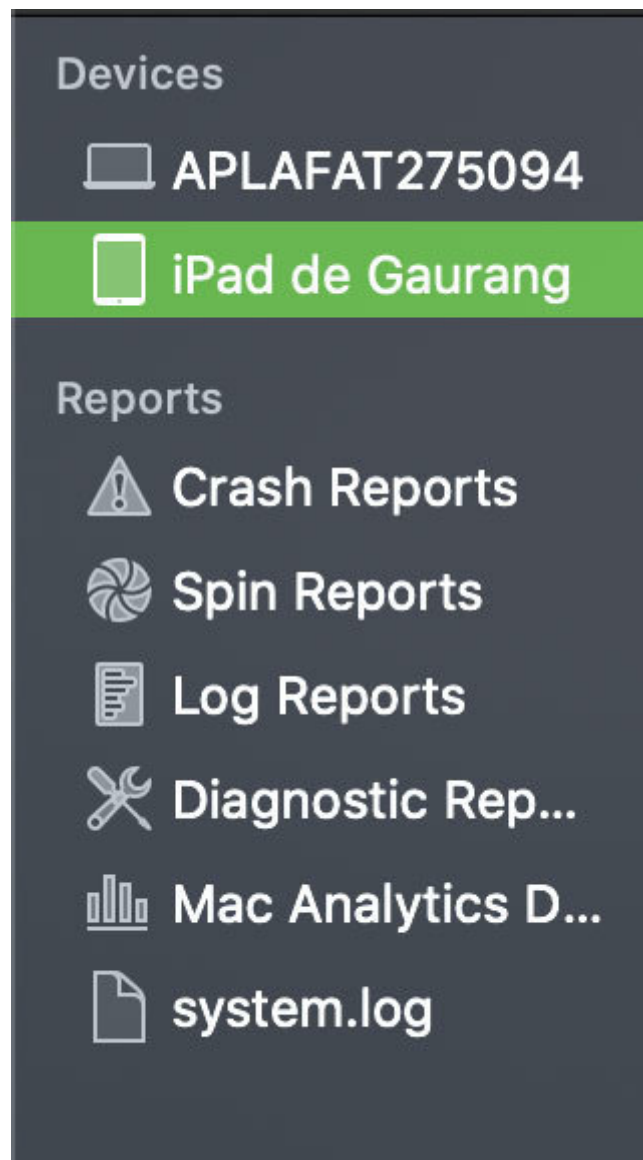


Figure 9.20: *Device selection for log analysis*

Next, in **Action**, you can enable what types of logs you want to observe as shown in [Figure](#)

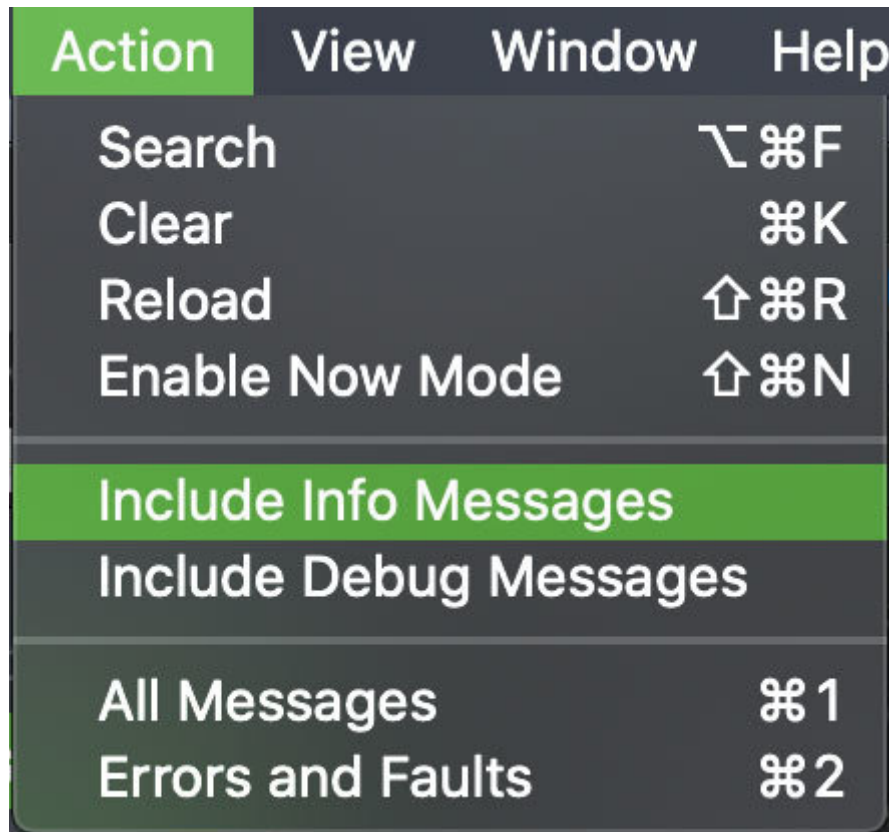


Figure 9.21: Log type selection from Actions Menu

Then, you can observe the logs in the log panel. Even if your app is not in the debugging mode, these logs are visible. So, happy logging!!

Conclusion

We completed an important chapter. Though we did not a lot in this chapter, we covered some important topics which are really helpful to be a good developer. These tools will help you in the real-world coding scenarios. We started with **memory** Then, we saw the different ways of **error handling** and This is where we ended the Swift iOS app development using the *imperative* In the next chapter, we will see something really interesting called SwiftUI, which uses the *declarative*

Key learnings

We started the chapter by learning about memory management.

We learned about the functioning of ARC, and how it manages memory by using reference counting.

We then saw an example about strong retain cycles and how we can break them using weak and unowned variables.

We also learned about optional chaining and how it can be used to avoid crashes.

We then learned about throwing functions and how we can use them to pass errors from one class to another. We also saw how to use try catch to handle them.

We then saw the debugging tool provided by Xcode and tested it using a simple example.

We ended the chapter by using the logging framework, which helps us persist and send logs from our app to the server.

Test your knowledge

How does ARC manage the memory for structures and enumerations?

Structures and enumerations are value types so they do not need memory management.

When you create a variable and assign an instance of a class to it, a strong reference is created. State true or false.

True

What is the difference between a weak reference and an unowned reference?

An unowned reference is always expected to have a value.

Can optional chaining be used on methods?

Yes, optional chaining can also be used for methods of the type.

```
if let sound = person.pet?.makeSound() {  
    print(sound)  
}
```

Which protocol does the type need to conform to for an error?

The Error protocol

Which button do we need to use to debug a function call from the line of debug point?

The step-in button

Which is the highest priority type of log?

Fault – It is also the one which is persisted the longest.

Which logging parameter can we use to stop logging private data?

The private logging parameter

```
os_log("Second value is %{private}@", log: OSLog.default, type:
.info, secondValue)
```

In this chapter, we will learn about SwiftUI. It is a framework used to build user interfaces. It was introduced by Apple in WWDC 2019 and since then, they are making a push for it. This can be the future of the Apple app development. It uses the declarative form of programming. This provides a very simple readable form of writing the user interface. It is a completely new approach and different from the usual storyboards and ViewControllers routine. The best part is that you can build an app for any Apple device using just one set of tools and APIs.

Structure

In this chapter, we will cover the following topics:

Introduction to the SwiftUI framework

Design and layout

User actions

List views

Navigation

SwiftUI with UIKit

Objective

The objective of this chapter is to introduce you to SwiftUI. As we can make apps for any Apple device with SwiftUI, we are possibly looking at the future of app development in the Apple environment. As an iOS app developer, it is imperative that you know about SwiftUI. Since 2019, this framework has evolved a lot and is more mature than its debutant version. *You will have to use Xcode 12 and above for this*

Introduction to the SwiftUI framework

The SwiftUI framework was introduced with iOS 13. It uses **declarative**. Before iOS 13, we used **imperative**. So far, we have been using the imperative programming style. Imperative coding style means we monitor the state of data to update our views. Remember MVC! How does the controller update the view, depending on the state of the model? For example, in a function, we have to write a code to monitor a value of a model; if it has a certain value, we change the image and color of a button. Otherwise, we apply some other colors and images. More the values, the more monitoring we will have to do. Thus, our code also becomes prone to errors. The declarative UI tries to break this coding style.

In the declarative UI, we tell the OS all possible rules before and the SwiftUI framework manages the view changes. So, it would be SwiftUI changing the colors and images of the button for us. We do not have to do it in any function; instead, we have to inform SwiftUI of the state change of our model.

With Swift UI, Apple is trying to create cross-platform layouts. You can create layouts for iOS, macOS, tvOS, and even watchOS. This simplifies the learning process. You are able to write code for any Apple platform. How neat is that!!

Design and layout in SwiftUI

We will learn SwiftUI through a step-by-step tutorial. In this section, we will understand how to build simple views and create different layouts.

Creating a project

Let's start creating a project as follows:

Open Xcode and click on create a new project.

Now, choose the app template.

In the next window, you will have to choose the options for your app. Give this app a name you like. I gave it SwiftUIViews.

Choose SwiftUI for Interface and Life Cycle as shown in [Figure](#)

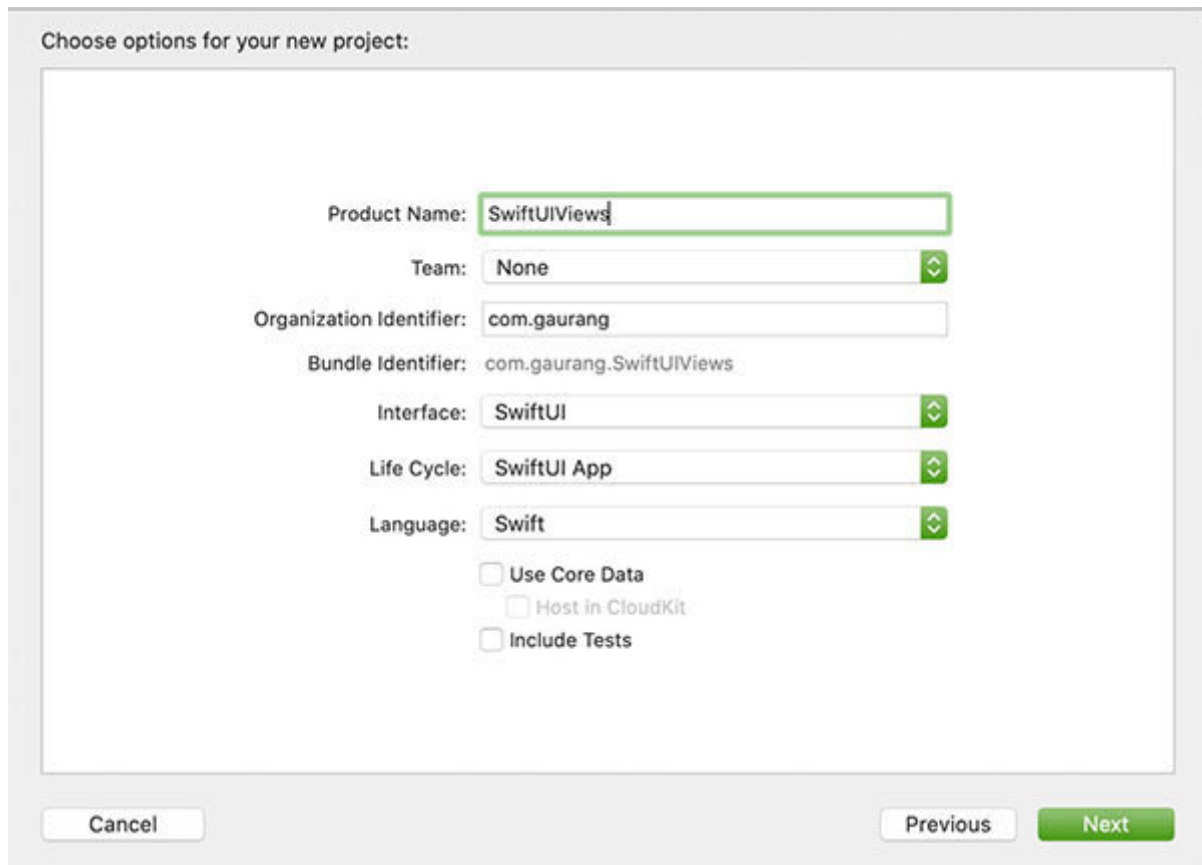


Figure 10.1: *Choosing SwiftUI for interface*

Open the You will see the following code:

```
import SwiftUI
```

```
@main
```

```
struct SwiftUIViewsApp: App {  
    var body: some Scene {  
        WindowGroup {
```

```
            ContentView()  
        }  
    }  
}
```

By now you should be able to read this code:

@main represents the entry point of the SwiftUI App life cycle: Because we chose

Our app conforms to the **App** protocol.

The body returns one or more scenes.

The scene contains our main content view.

Now, open the I won't be explaining the code, as by now, you should be able to understand it. On the right-hand side, turn on the preview update, by clicking on the **Resume** button as shown in [Figure](#)

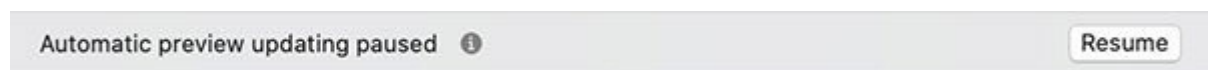


Figure 10.2: Turn on automatic preview

Wait for the preview to load. You should now have a screen similar to [Figure](#) Well done, our basic setup is done!

You can try and change the text to see if it is working.

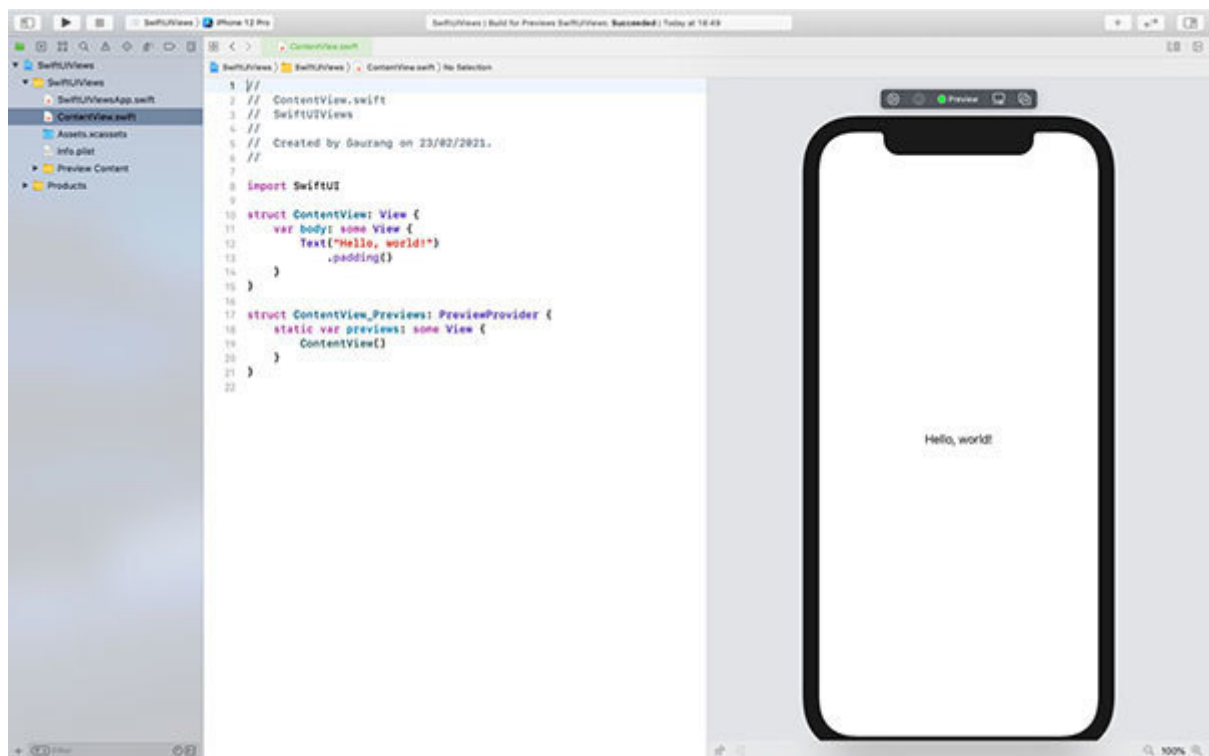


Figure 10.3: Automatic preview in action

Playing around with views

Let's try to customize the text view we see. We will learn more as we play around with the views.

Command + click on the text view to open up the structured editing popover as shown in [Figure](#)

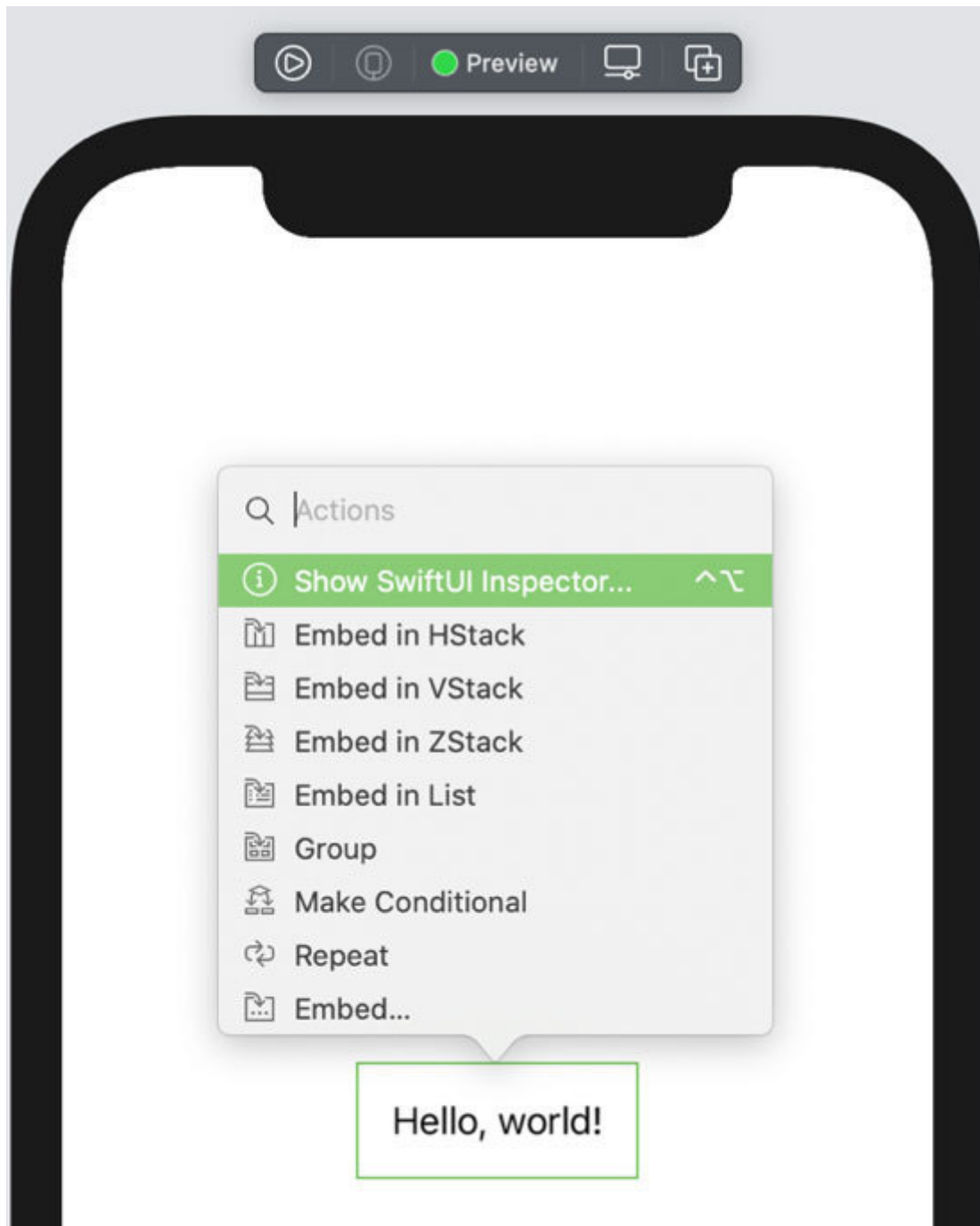


Figure 10.4: *Structured editing popover*

Then, click on **Show SwiftUI Inspector** and go ahead and change some properties you want. You can try to change the text, or alignment, or padding. The inspector is very intuitive as shown in [Figure](#)

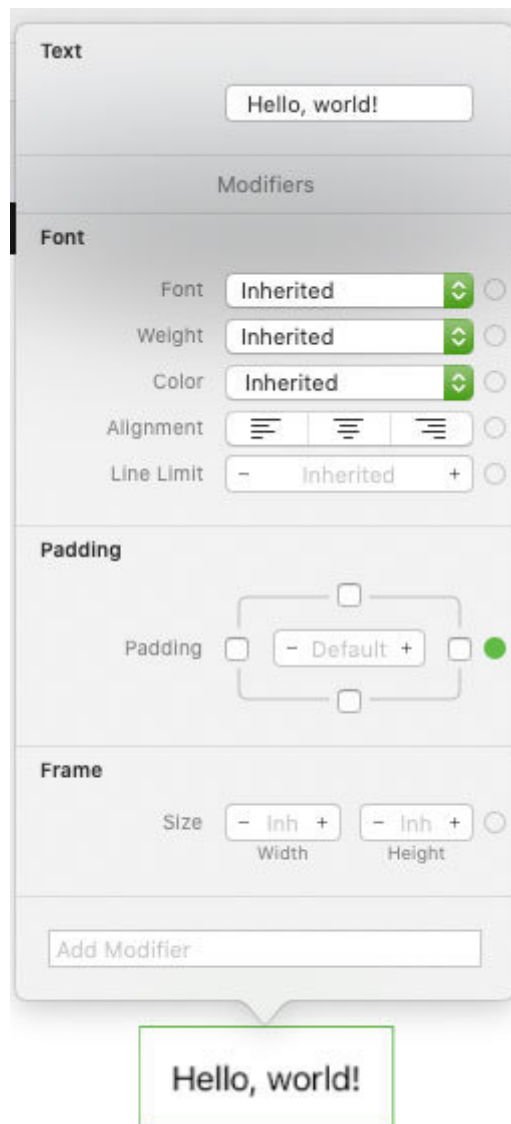


Figure 10.4.1: *Inspector popover*

You won't see a lot of options. But you can add a custom modifier you want in the field of the inspector. Modifiers are methods used to customize the SwiftUI view. The following code shows the use of a modifier to change the foreground color:

```
var body: some View {  
  Text("Hello, world!")  
  .padding()  
  .foregroundColor(.red) //this is a modifier  
}
```

You will notice that the code starts changing as you modify the parameters. When you start adding or changing properties in the Inspector, your code modifies in parallel. But if you know your modifiers well, you do not have to open the inspector, you can add modifiers in your code directly:

```
struct ContentView: View {  
  var body: some View {  
    Text("Hello, Reader!")  
    .font(.title)  
    .fontWeight(.semibold)  
    .multilineTextAlignment(.leading)  
    .lineLimit(4)  
    .padding()  
    .foregroundColor(.blue)  
  }  
}
```

As you cmd+clicked on the text view to open the inspector, you can even cmd+click on the code. [Figure 10.5](#) shows that the declarative UI is two ways and works perfectly:

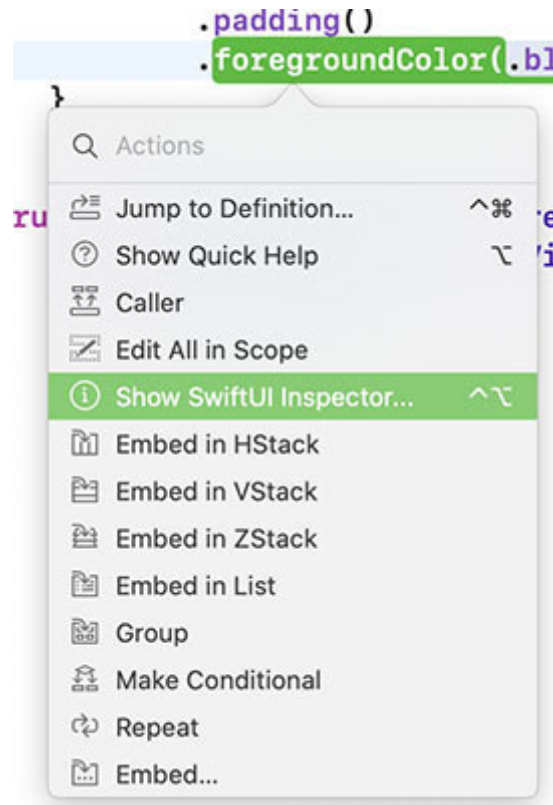


Figure 10.5: Code side inspector popover

The body property returns only If we need to add multiple views, we need to add them on a stack. There are two types of stacks such as **VStack** or **vertical stack** and **HStack** or **horizontal** We have already used stacks, so let us do the same. Cmd+click on your **Text** and embed it in a **VStack** as shown in [Figure](#)

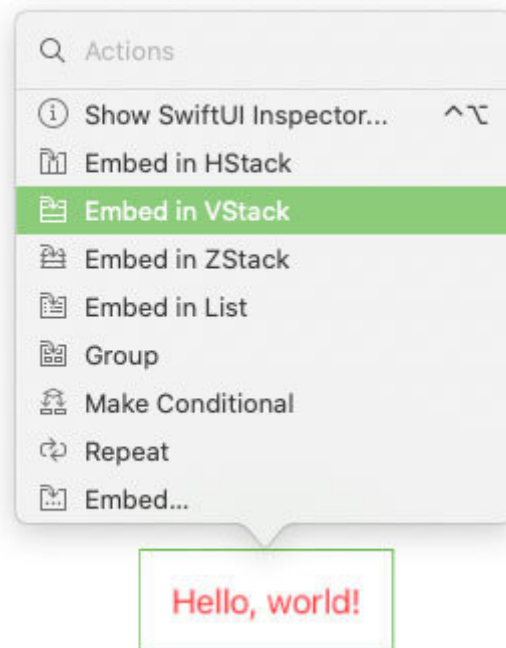


Figure 10.6: *Embedding in VStack*

Open the UI library and drag and drop another **Text** in your stack as shown in [Figure](#)

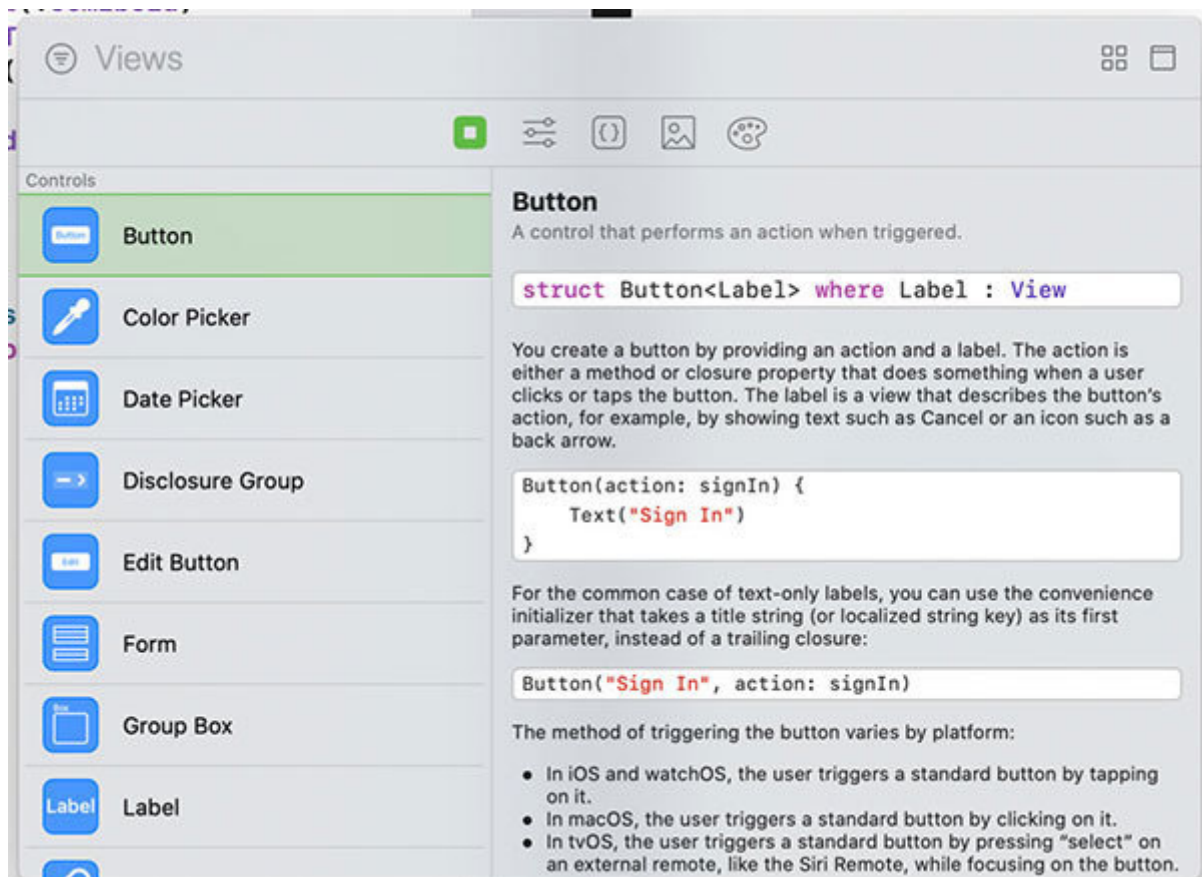


Figure 10.7: SwiftUI library

I added two more texts as shown in [Figure](#). Do you see the problem? They are very close to each other. How to manage this? This is when we use an element called

Hello, Reader!

Some new text
This is another text

Figure 10.8: Closely placed view elements

Spacers take up all the space. They stretch and push the items to take up the whole layout. Hence, as soon as you add a spacer, you must have seen a huge space between your two labels as shown in [Figure](#)

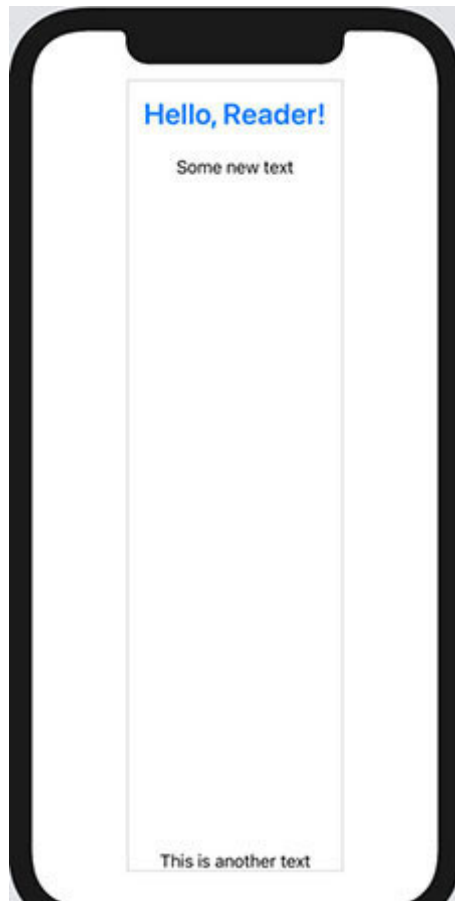


Figure 10.9: Views after adding spacers

This can be overcome by adding the frame modifier on your Space. Here is how the resulting code should look like:

```
struct ContentView: View {  
    var body: some View {  
        VStack() {
```



```
Text("Hello, Reader!")  
.font(.title)  
.fontWeight(.semibold)  
.lineLimit(4)  
.padding()  
.foregroundColor(.blue)
```

```
Text("Some new text")  
Spacer().frame(height:50)  
Text("This is another text")  
}  
}  
}
```

[Adding custom views](#)

Imagine we have many views. As we add them, our code will keep on getting bigger and probably ugly! We can add custom views to solve this problem. Let us add a custom view to our current App as follows:

Add an image set to your Assets library. (Refer to [Chapter 1, Building Your First Application](#) if you are unable to)

Now, right click on your **SwiftUIViews** folder and add a New File. Select a **SwiftUI View** and name it **ProfileImage** as shown in [Figure](#)

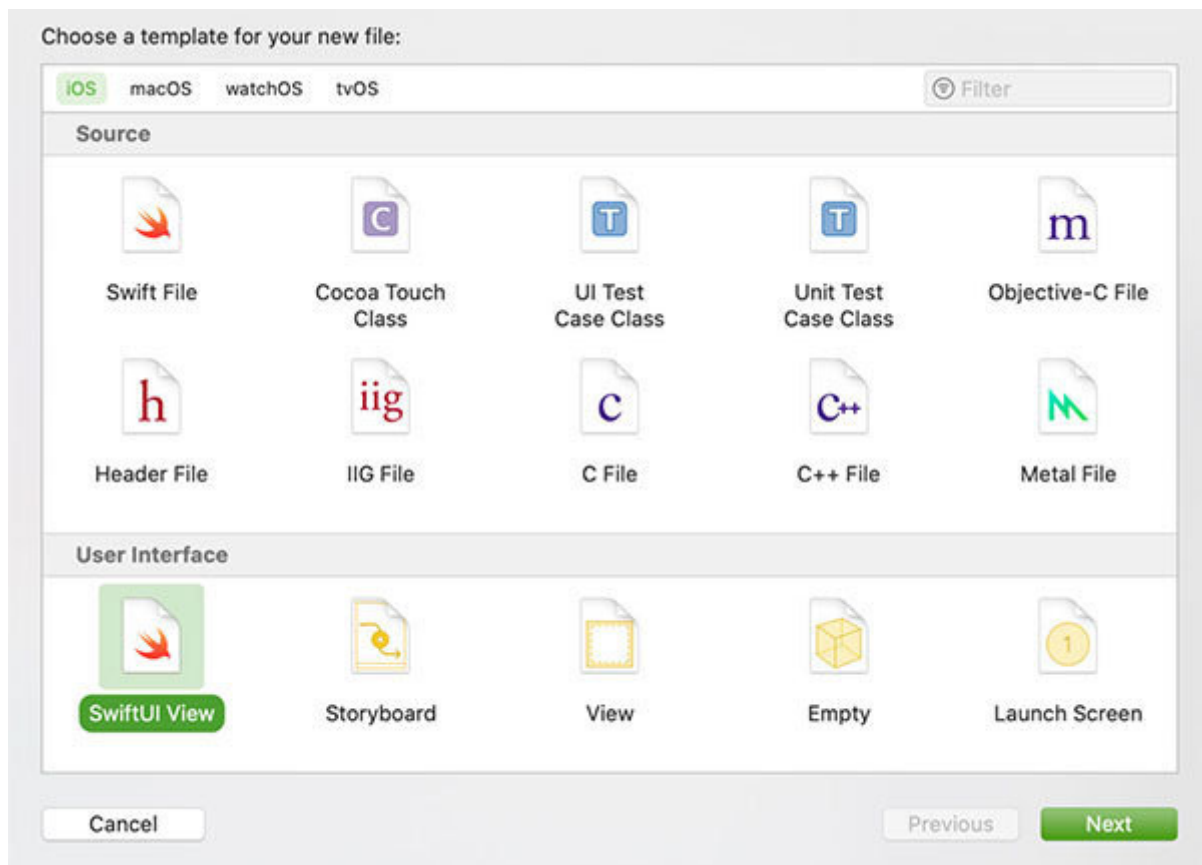


Figure 10.10: Adding a new view

In the body, remove the default Text and replace it with Image. Change the image name to the image from assets.

Cmd+click to open the inspector and check out the attributes you can change.

Try adding some modifiers. Your final code should look something like this:

```
struct ProfileImage: View {
    var body: some View {
        Image("Image")
```

```
.overlay(Circle().stroke(Color.gray, lineWidth: 3))
.shadow(radius:5)
}
}
```

Now, it's time to add this **ProfileImage** to your Look at the code carefully. **ProfileImage** is a struct. So, when we add it, we need to add an instance of this struct. Your final code should look like this:

```
struct ContentView: View {
  var body: some View {
    VStack() {
      ProfileImage() //this creates a new instance of ProfileImage
      Text("Hello, Reader!")
        .font(.title)
        .fontWeight(.semibold)
        .lineLimit(4)
        .padding()
        .foregroundColor(.blue)

      Text("Some new text")
      Spacer().frame(height:50)
      Text("This is another text")
    }
  }
}
```

Remember that this is just to show you how you can layout the views. Now, it is up to you to play around with this toolkit. Your

final preview should look something like [Figure](#)

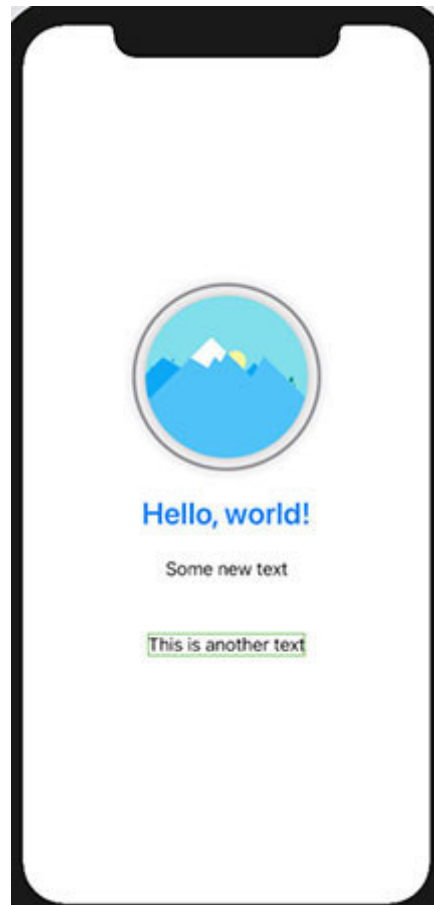


Figure 10.11: Screen after adding image view

User actions

We read in the introduction that buttons work differently in declarative programming. Let us see how that works. We will add a button and a simple alert action to it. But first, we will modify our **ContentView** to make it more sensible:

First, open our **ContentView** and change the text to something meaningful. I changed it to the following. This will be our details view. We will show the details of a person:

```
var body: some View {  
    VStack() {  
        ProfileImage()  
        Text("Hello, Tom!")  
        .font(.title)  
        .fontWeight(.semibold)  
        .lineLimit(4)  
        .padding()  
        .foregroundColor(.blue)  
        Text("Masters in Computers")  
        Spacer().frame(height:30)  
        Text("Currently living in Texas")  
    }  
}
```

Next, we will add a button. Add a spacer below the last text and then open the library and drag and drop a button below it. Here is the button that is added. Your final screen should look something like [Figure](#)

```
Button(action: {  
  
}, label: {  
Text("Add friend")  
.padding()  
.background(Color.green)  
.foregroundColor(Color.white)  
.cornerRadius(10)  
})
```

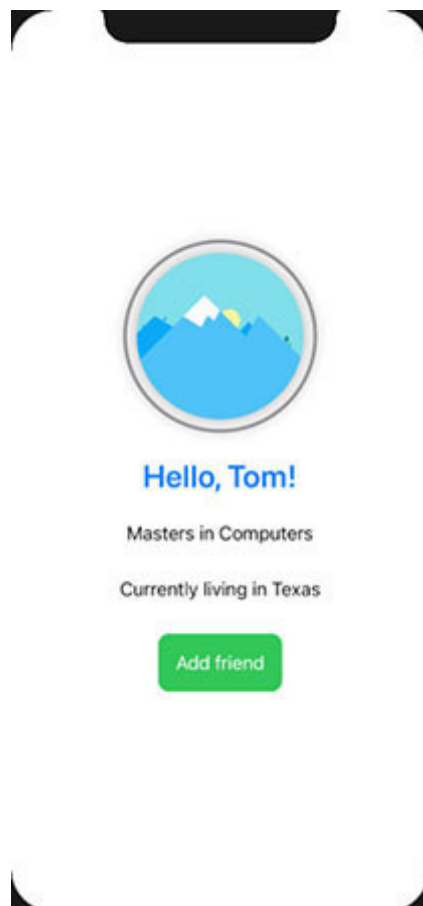


Figure 10.12: *View after adding button*

Let us now add an action to see our button working. We will show an alert on its click. We will first create a private variable as follows:

```
@State private var showingAlert = false
```

The keyword **State** is new. It is a property wrapper which monitors the state of the type. When we access the value of the type from within the body, we can access the wrapped value, otherwise a **State** instance is not a value in itself; it is a means of reading and writing the value. Here, we have initiated our Boolean **showingAlert** with an initial value of

We will then add an alert to our view. The code after adding an alert should look like the following code:

```
Button(action: {  
    showingAlert = true  
}, label: {  
    Text("Add friend")  
    .padding()  
    .background(Color.green)  
    .foregroundColor(Color.white)  
    .cornerRadius(10)  
}).alert(isPresented: $showingAlert) {
```



```
Alert(title: Text("Friend request sent"), message: nil, dismissButton:  
.default(Text("Ok")))  
}
```

Let us try to understand the code. In the action, we will change the value of the `Then`, we will add a modifier to present an alert:

```
.alert(isPresented: $showingAlert) {  
Alert(title: Text("Friend request sent"), message: nil, dismissButton:  
.default(Text("Ok")))  
}
```

Here, we bind the value of our bool **showingAlert** to the **isPresented** property of the alert. So, whenever the alert is dismissed, the value of **showingAlert** will be automatically reset. This is the beauty of declarative programming.

You can check whether your code is working by switching to live preview by clicking on the button on the top as shown in [Figure](#)

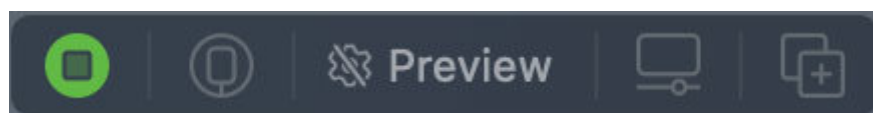


Figure 10.13: Live preview options

Next, we will check out how we can create lists in SwiftUI.

List views

Let us now try to create a List view in our app:

We have changed our ContentView to show details of a Person. We will now create a model to hold this data. First, we will add a new Swift file called You can define the new **struct** as you like. This would be our model:

```
struct Person {  
    var name: String  
    var city: String  
    var education: String  
    var imageName: String?  
}
```

Then, we will add a new file called This would provide us with a list of persons. Here is an example;

```
var persons: [Person] = [Person(name: "Tom", city: "Texas",  
    education: "Masters in Computers"),Person(name: "Jack", city:  
    "Tokyo", education: "Masters in Psychology"),Person(name: "Leticia",  
    city: "Madrid", education: "Masters in Big Data")]
```

For the list view, we need to repeat rows. Right click on the app folder and create a Row. Let's call it Design it to have an image

and a text in a Declare a stored property called **person** of the type This would be used by the row. Here is the resulting code:

```
struct ListRow: View {
var person: Person //every row represents a person
var body: some View {

HStack {
Image("Image").resizable()
.frame(width: 50, height: 50)
Text(person.name)
Spacer()
}
}
}
```

As soon as you declare the person, you will notice that the preview stops working because the **ListRow** will demand a person instance during initialization. Change the initializer to accept a person as seen in

```
struct ListRow_Previews: PreviewProvider {
static var previews: some View {
ListRow(person:persons[0])
}
}
```

Then, add a modifier on this initializer and you will see that we have created a preview layout of our row. You can even create multiple such layouts and put them together in a Your preview should now look like [*Figure*](#)

```
ListRow(person:persons[0]).previewLayout(.fixed(width: 300, height: 70))
```

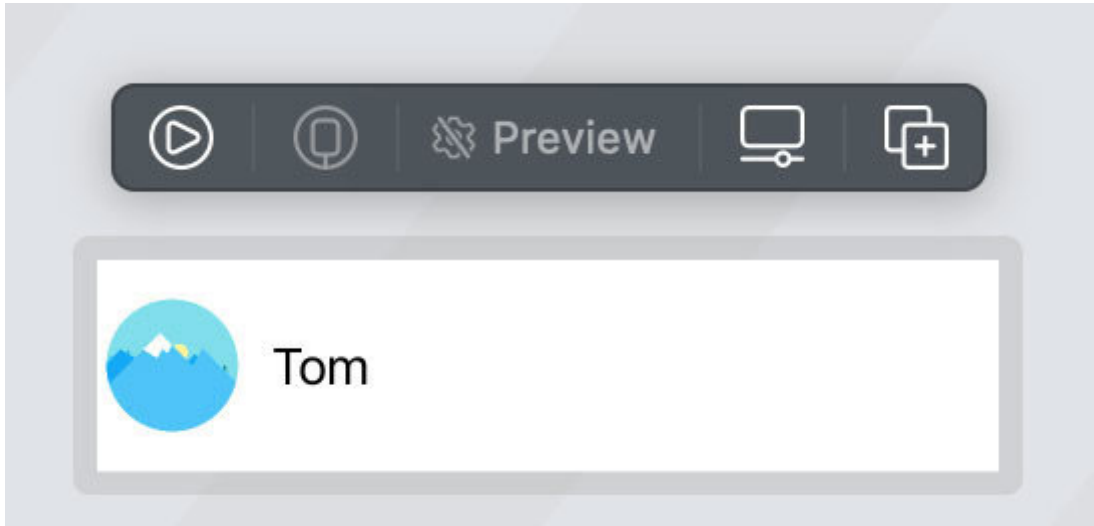


Figure 10.14: *ListRow layout preview*

The next step is to design a Create a new SwiftUI class called Add a list in its body. Now, try to add some items to your list. These items would be nothing but the instances of **ListRow** we created above. This should look as shown in [Figure](#)

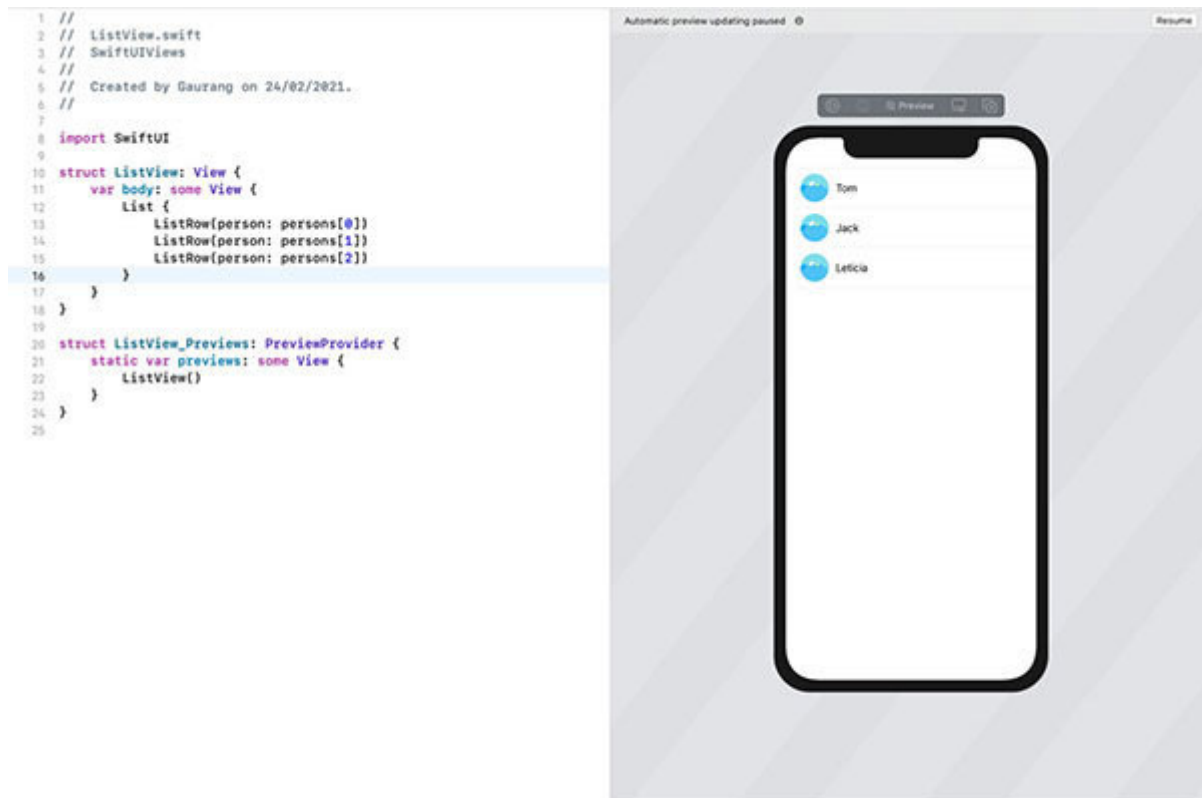


Figure 10.15: List with code and it's live preview

But we would rather build a list with dynamic elements. Let us try to fix that. For a list to repeat the elements, they need to be identifiable.

Your model can be identifiable, by either having a distinct key path (like id) or by conforming to the Identifiable protocol. Our type **Person** neither has the ID nor does it conform to the identifiable protocol. But for demo purpose, let us consider the *name* as our keypath as shown in [Figure](#)

```

 7
10 struct ListView: View {
11     var body: some View {
12         //id is the identifiable keypath
13         List(persons, id: \.name) { person in
14             ListRow(person: person)
15         }
16     }
17 }
18

```

Figure 10.16: Using name as the keypath

Let us try and understand the code:

```

List(persons, id: \.name) {person in
ListRow(person: person)
}

```

Here, we are iterating over the **persons** list we created. This will make our **ListView** dynamic. ID is the identifiable keypath which will be used by the list to check the uniqueness of the row. We have used the name keypath here. Then, we have used a closure and returned an instance of **ListRow** for every person in the list. You can try adding more rows, or change, or add an identifiable keypath.

Next, we will add a new view called the Our entry point of the app is the If we make a the body of our **ContentView** should now point towards it rather than showing the details. In the newly created cut and paste the contents of the Your **DetailsView** is ready already!! This is how simple declarative programming is!! Your code for **ContentView** should look like this:

```
struct ContentView: View {  
    var body: some View {  
        ListView()  
    }  
}
```

Tip: The Identifiable protocol requires you to have a property “id” in your type. This property helps the protocol to decode your type.

Navigation

We have still not added navigation to our list. You can click on the rows but nothing happens. Let us fix that.

Like how we embed our *views* in stack, you must embed your **ListView** in a **NavigationView**. The resulting view should look like [Figure](#)

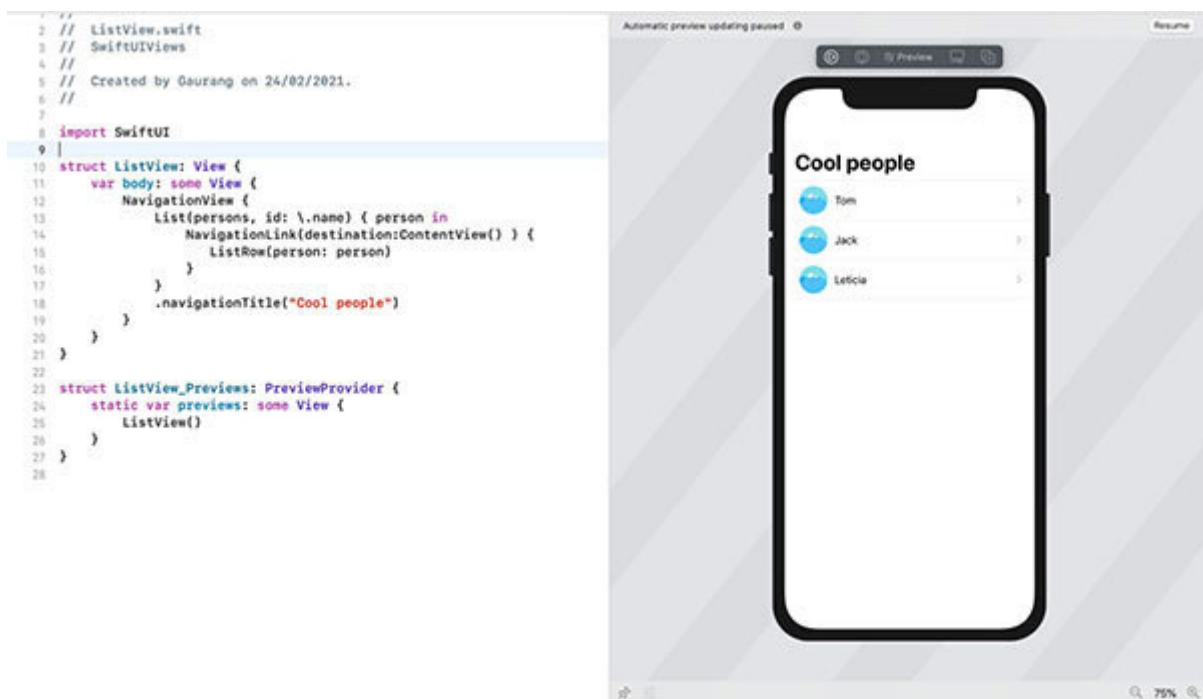


Figure 10.17: Embedding list in navigation

Let us look at the code changes we made. The code is quite self-explanatory. For each person, we create a navigation link in which we have our


```

var body: some View {
    NavigationView { //this embeds the List into a navigation view
        List(persons, id: \.name) {person in
            NavigationLink(destination:DetailsView(person: person)) {
                ListRow(person: person) //this creates a navigation link for each
                row
            }
        }
    }
    .navigationTitle("Cool people")
}
}

```

When we put our **List** in the curly braces of it gets embedded in the navigation stack. But that does not mean the navigation will work. We will have to tell SwiftUI that we want to navigate on the click of each row. Hence, we embed every **ListRow** in a Each **NavigationLink** has a destination; here, the You can even create different detail views and change the navigation destination depending on the condition.

You can check whether your code is working by switching to live preview by clicking the button on the top.

If you want to add more devices to your preview, you can do so by modifying the This will allow you to check your view on different device formats:

```

struct ListView_Previews: PreviewProvider {
    static var previews: some View {

```

```
ForEach(["iPhone 11 pro", "iPhone 12 mini", "iPhone SE (2nd  
generation)"], id: \.self) {deviceName in  
    ListView()  
    .previewDevice(PreviewDevice(rawValue: deviceName))  
    .previewDisplayName(deviceName)  
  
}  
  
}  
  
}
```

SwiftUI and UIKit

After learning this, you must be wondering do we need to change an existing project entirely to SwiftUI? This can be done for sure, but we can use SwiftUI views in existing UIKit projects and vice versa. We can do this with the use of extensions. Apple takes care of the transition from one to another.

SwiftUI in A SwiftUI view needs hosting in a UIKit. The hosting controller acts as a bridge. It is as simple as using and adding a view controller in a storyboard. Here is a sample code:

```
func presentSwiftUIView() {  
    let swiftUIView = SwiftUIView()  
    let hostingController = UIHostingController(rootView: swiftUIView)  
    present(hostingController, animated: true, completion: nil)  
}
```

The preceding code, if repeated, can be simplified by creating an extension on the UIView class. The function of this extension will look something like this:

```
func addSubSwiftUIView(_ swiftUIView: Content, to view: UIView)  
where Content : View {  
}
```

UIKit in We need a host to make SwiftUI work in a UIKit, but the other way around is easier. We have to import the **SwiftUIKitView** in our project:

```
import SwiftUIUIKitView
```

```
struct SwiftUIwithUIKitView: View {  
    var body: some View {  
        UILabel().swiftUIView(layout: .intrinsic)  
    }  
  
}
```

You will see in the preceding example that we are transforming the UIKit label to a Swift UI View. We are using the **swiftUIView** method for this. The type of layout can be intrinsic, fixed width, and fixed size.

Exercise: We have not shown the data in the This is for you to complete. First, try to pass the person instance to the like we did for the Then, add more images to have a different image for each entry.

Conclusion

This chapter was to give you a brief introduction to the future of Apple app development. Now, you know the basics of the SwiftUI framework, and this chapter can serve as a perfect starting point. Of course, there are many more details which you can dive into on your own.

This is the end of the book and now you know both Swift and SwiftUI, i.e., imperative and declarative programming styles. Your next task would be to dive deeper into the iOS app development world and try to explore frameworks like UIKit in depth and learn new frameworks like Your app development journey has just begun. All the best!!

Key learnings

This chapter gave you a brief introduction about the SwiftUI framework.

It uses the declarative programming as opposed to the imperative programming style which we had been using up until now.

We started by building simple views and learned about modifiers. These are methods used to customize views.

We then saw the use of stacks to combine views and spacers to add proper spaces between them.

We even learned how to create custom views to keep the code clean.

We also learned how to create a List in SwiftUI

We ended the chapter by adding navigation to our existing app.

Test your knowledge

What is the programming style used in SwiftUI called?

Declarative programming

What are the different types of stacks in SwiftUI?

VStack

HStack

ZStack

What are the methods used to customize views called?

Modifiers

Which element is used to add space between two UI elements?

Spacers

Which protocol should your model conform to in order to be used in a List?

Identifiable

What are the two key elements used for navigation?

NavigationView

NavigationLink

Where can you add more devices for preview?

PreviewProvider

A

access control

about [104](#)

module [104](#)

source file [104](#)

ambiguous layout [56](#)

Any keyword [158](#)

app delegate [32](#)

application

working [14](#)

app structure [32](#)

arguments [121](#)

arithmetic operator [30](#)

array

about [76](#)

accessing [77](#)

creating

modifying [77](#)

array literals [77](#)

assets catalog [19](#)

assignment operator (=) [30](#)

associated values [149](#)

asynchronous function [101](#)

auto layout

need for

source of changes [42](#)

Automatic Reference Counting (ARC)

B

base class [155](#)
behavioural design pattern [163](#)
binary operator [29](#)
bounds [68](#)

C

classes
about [150](#)
identity operators [154](#)
initializing [152](#)
reference types [154](#)
value types [153](#)
versus structures [151](#)
classes, functionalities
deinitializers [151](#)
inheritance [151](#)
reference counting [151](#)
reference types [151](#)
type casting [151](#)
closure expression
closures [125](#)
code debugging
collection type
about [76](#)
array [76](#)

dictionary [81](#)

sets [79](#)

commenting [29](#)

comparison operator [30](#)

compile time polymorphism [163](#)

compound assignment operator [30](#)

computed properties [132](#)

concurrency design pattern [164](#)

Console.app

using [216](#)

constants [27](#)

constraints

control flow

about [83](#)

for in statement [86](#)

if else statement [83](#)

switch statement [85](#)

while loop

controller [166](#)

controller communication [166](#)

control transfer statements

about [101](#)

break [102](#)

continue [102](#)

fallthrough [103](#)

return [103](#)

throw [103](#)

creational design pattern [163](#)

custom operator [163](#)

D

debugging

about [205](#)

UI debugging [206](#)

debug point

editing [213](#)

declarative programming [220](#)

default stored values, initializers

customizing [134](#)

deinitializer [135](#)

defer keyword [101](#)

deinitializer [135](#)

delegate [164](#)

delegate design pattern [178](#)

design patterns

about [163](#)

advantages [164](#)

behavioural design pattern [163](#)

concurrency design pattern [164](#)

creational design pattern [163](#)

disadvantages [164](#)

structural design pattern [163](#)

development environment

problem solving [21](#)

dictionary

about [81](#)

accessing [82](#)

creating [82](#)

modifying [82](#)

down casting operator [158](#)

E

EntryApp

setup

enumeration

about [146](#)

associated values [149](#)

iterating over [150](#)

raw values [147](#)

switch case [149](#)

error handling

about [204](#)

throw functions [204](#)

event driven programming [32](#)

existential type [175](#)

extended delimiters [71](#)

extensions

about [161](#)

computed properties [161](#)

initializer, extending [162](#)

F

failable initializer [135](#)

for in statement [85](#)

function

about [121](#)

calling [122](#)

defining [121](#)

method [125](#)
nested function [124](#),
parameters [123](#)
return values [123](#)

G

generics [160](#)
global function [125](#)
guard let

H

hashable [79](#),
hash value [79](#),
Herbivore class [155](#)
HStack [63](#)

I

IOutlets
if else statement [83](#)
if-let
immutability [27](#),
imperative programming [220](#)
inheritance
about [155](#)
base class [155](#)
overriding [157](#)

subclass [156](#)

initializer function [135](#)

initializers

about [134](#)

default stored values [134](#)

instance method [125](#)

Int64 [68](#)

Integrated Development Environment (IDE) [6](#)

interface

building

views, adding [18](#)

iOS application

environment, setting up [3](#)

hardware device, running

iOS simulator, using [11](#)

running [10](#)

Xcode project, creating

L

Last In First Out (LIFO) [120](#)

layout

working [43](#)

layout, types

auto layout [43](#)

frame based [43](#)

list view

Logger [214](#)

logging [213](#)

logging levels, types

- debug [214](#)
- error [214](#)
- fault [214](#)
- info [214](#)
- notice [214](#)
- logging parameters [215](#)
- logical operator [31](#)
- logical operator, rules and symbols
- AND [31](#)
- NOT [31](#)
- OR [31](#)

M

- memberwise initializer [134](#)
- memory management [196](#)
- method [125](#)
- method overloading [163](#)
- model [165](#)
- model communication [165](#)
- model layer [165](#)
- Model-View-Controller (MVC)
 - about [165](#)
 - drawbacks [166](#)
 - in iOS [166](#)
- multiple constraints
 - adding
 - ambiguous layout [56](#)
 - editing [54](#)
 - unsatisfiable layout [55](#)

N

naming convention [32](#)

navigation

about

delegates, using [140](#)

segue behavior [140](#)

nested function [125](#)

nib [35](#)

O

observers [132](#)

operator overloading [163](#)

operators [29](#)

operators, types

arithmetic operator [30](#)

assignment operator (=) [30](#)

binary operator [29](#)

comparison operator [30](#)

compound assignment operator [30](#)

logical operator [31](#)

ternary conditional operator [31](#)

ternary operator [29](#)

unary operator [29](#)

optional chaining [203](#)

optionals [97](#)

OS Framework [213](#)

overriding [157](#)

P

parameters [121](#)

playground tool [26](#)

products [35](#)

properties [131](#)

protocol

about [170](#)

defining [171](#)

initializer requirement [173](#)

method requirement [172](#)

property requirement [171](#)

syntax [170](#)

protocol extension [176](#)

protocol inheritance [177](#)

protocol-oriented programming (POP)

about [173](#)

features [174](#)

protocol extension [176](#)

protocol inheritance [177](#)

protocol type [175](#)

using, in Swift [174](#)

R

raw values

assigning [148](#)

initializing [148](#)

readability [28](#)

reference counting [196](#)

S

scene delegate [34](#)

second view, word bank app

images, adding [189](#)

new word, adding

updating [192](#)

segue [136](#)

setDefaultValues function [121](#)

sets

about [7.9](#)

accessing [80](#)

creating [7.9](#)

hash values [7.9](#)

modifying [80](#)

simple calculator

size classes

SophisticatedCalculator

building [106](#)

image icon, adding to app

layout, preparing [110](#)

working

source of changes

external changes [42](#)

internal changes [43](#)

steppers

using [212](#)

stored properties [131](#)

- storyboards [35](#)
- string
 - characters, counting [72](#)
 - comparison [74](#)
 - concatenation [71](#)
 - creating [70](#)
 - indices [73](#)
 - inserting [74](#)
 - interpolation [71](#)
 - iterating [72](#)
 - removing [74](#)
 - type conversion [75](#)
 - string manipulation [69](#)
 - strong reference [196](#)
 - structural design pattern [163](#)
 - structures
 - about [150](#)

- versus classes [151](#)
- subclass [155](#)
- subscript [77](#)
- subscripting [132](#)
- subscripts [133](#)
- substring [74](#)
- Swift [24](#)
- Swift types
 - about [68](#)
 - bool [68](#)
 - bounds [68](#)
 - double [69](#)
 - float [69](#)
 - floating point numbers [69](#)

- int [68](#)
- integers [68](#)
- SwiftUI
 - about [237](#)
 - custom views [229](#)
 - custom views, adding [228](#)
 - design [222](#)
 - in UIKit [237](#)
 - layout [222](#)
 - project, creating [221](#)
 - text view
- SwiftUI framework [220](#)
- switch statement [85](#)
- symbols [29](#)

T

- targets [35](#)
- ternary conditional operator [31](#)
- ternary operator [29](#)
- throw functions
 - about [204](#)
- try-catch, using [205](#)
- tuples [83](#)
- type alias [104](#)
- type casting
 - about [157](#)
- Any keyword [158](#)
- down casting operator [158](#)
- type checking [157](#)

type check operator [157](#)

type inference [77](#)

type method [125](#)

U

UI debugging [208](#)

UIKit

about [237](#)

in SwiftUI [238](#)

unary operator [29](#)

unwrapping [97](#)

user actions

V

variable [27](#)

view [165](#)

view communication [165](#)

ViewController [35](#)

view controller lifecycle [62](#)

views

about [35](#)

types [63](#)

void type function [122](#)

VStack [63](#)

W

while loop

windows [35](#)

word bank app

about [178](#)

features [193](#)

second view, configuring [185](#)

second view, creating [184](#)

table view, configuring

x

Xcode

about

debug pane [9](#)

editor pane [9](#)

inspector pane [9](#)

navigation pane [8](#)

status bar [8](#)