Pro iPhone Development with SwiftUI

Design and Manage Top Quality Apps

Third Edition

Wallace Wang

Apress®

Pro iPhone Development with SwiftUI

Design and Manage Top Quality Apps

Third Edition

Wallace Wang

Apress®

Pro iPhone Development with SwiftUI: Design and Manage Top Quality Apps

Wallace Wang
San Diego, CA, USA

ISBN-13 (pbk): 978-1-4842-7826-0 ISBN-13 (electronic): 978-1-4842-7827-7
https://doi.org/10.1007/978-1-4842-7827-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub: https://github.com/Apress/Pro-iPhone-Development-with-SwiftUI. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Wallace Wang is a former Windows enthusiast who took one look at Vista and realized that the future of computing belonged to the Mac. He's written more than 40 computer books, including *Microsoft Office for Dummies*, *Beginning Programming for Dummies*, *Steal This Computer Book*, *My New Mac*, and *My New iPad*. In addition to programming the Mac and iPhone/iPad, he also performs stand-up comedy, having appeared on A&E's "An Evening at the Improv" and having performed in Las Vegas at the Riviera Comedy Club at the Riviera Hotel and Casino. When he's not writing computer books or performing stand-up comedy, he enjoys blogging about screenwriting at his site, The 15 Minute Movie Method, where he shares screenwriting tips with other aspiring screenwriters who all share the goal of breaking into Hollywood.

About the Technical Reviewer

Wesley Matlock is a published author of books about iOS technologies. He has more than 20 years of development experience in several different platforms. He first started doing mobile development on the Compaq iPAQ in the early 2000s. Today, Wesley enjoys developing on the iOS platform and bringing new ideas to life for Major League Baseball in the Denver metro area.

CHAPTER 1

Organizing Code

Programs are rewritten and modified far more often than they are ever created. That means most of the time developers must change and modify existing code either written by someone else or written by you sometime in the past. Since you may be writing code that you or someone else will eventually modify in the future, you need to make sure you organize your code to make it easy to understand.

While every developer has their own programming style and no two programmers will write the exact same code, programming involves writing code that works and writing code that's easy to understand.

Writing code that works is hard. Unfortunately, once developers get their code to work, they rarely clean it up and optimize it. The end result is a confusing mix of code that works but isn't easy to understand. To modify that code, someone has to decipher how it works and then rewrite that code to make it cleaner to read while still working as well as the original code. Since this takes time and doesn't add any new features, it's often ignored.

Since few developers want to take time to clean up their code after they get it to work, it's best to get in the habit of writing clear, understandable code right from the start. That involves several tasks:

- Writing code in a consistent and understandable style
- Making the logic of your code clear so anyone reading it later can easily understand how it works
- Organizing code to make it easy to modify later

When writing code, focus on clarity and readability. It's possible to write code that works but is hard to understand. That makes modifying that code difficult. Many times, it can be easier to rewrite code from scratch rather than waste time trying to figure out how it works.

W. Wang, *Pro iPhone Development with SwiftUI*, https://doi.org/10.1007/978-1-4842-7827-7_1

Swift lets you choose any name for variables and constants. However, it's a good idea to use descriptive names to help you understand what type of data that variable or constant can hold. Consider the following variable names:

```
var x: Int = 8
var dgie83: Double = 13.48
var FLdkjep: String = "Right"
```

While valid, these names don't make it clear what type of data they hold. A far better solution is to use descriptive names like this:

```
var age: Int = 8
var weight: Double = 13.48
var direction: String = "Right"
```

Single word variable names can be fine, but you may want to use multiple words to make a variable or constant name even more descriptive. When combining multiple words to form a variable or constant name, Swift programmers commonly use camelCase, which uses lowercase letters for the first word and an uppercase letter for the first letter of each succeeding word like this:

```
var ageOfPet: Int = 8
var weightInKilograms: Double = 13.48
var directionToTurn: String = "Right"
```

Just as variable and constant names can be too short, they can also be too long. Ideally, use descriptive names that get their meaning across using as few words and characters as possible.

When declaring variables or constants, you can optionally define the data type they hold by adding a prefix or suffix that identifies the type of data they contain such as

```
var strName : String
var intAge : Int
var dblSalary : Double

var nameStr : String
var ageInt : Int
var salaryDbl : Double
```

Note The idea of adding data type prefixes to variable and constant names is known as Hungarian notation, which was invented by Charles Simonyi, who worked at Xerox PARC and Microsoft.

The ultimate goal is to write self-documenting code that makes it easy for anyone to understand at first glance. One huge trap that programmers often make is assuming they'll be able to understand their own code months or even years later. Yet even after a few weeks, your own code can seem confusing because you're no longer familiar with your assumptions and logic that you had when you wrote the code originally.

If you can't even understand your own code months or even weeks later, imagine how difficult other programmers will find your code when they have to modify it in your absence. Good code doesn't just work, but it's easy for other programmers to understand how it works and what it does as well.

When developing your own programming style, strive for consistency and organization. Consistency means you use the same convention for writing code whether it's naming variables with prefixes or suffixes that identify the data type or indenting code the same way to highlight specific steps.

Organization means using spacing and storing related code together such as putting variables and functions in the same location consistently. This can group chunks of code in specific places to make code easier to understand as shown in Figure 1-1.

```
import SwiftUI

struct ContentView: View {
    @State private var choiceArray = [Int]()
    @State private var randomNumber = 0
    @State private var message = ""
    @State private var arrayLength = 100
    @State private var maxNumber: Double = 100

    var body: some View {
        VStack {
            Slider(value: $maxNumber, in: 1...100)
            Text("Max value = \(Int(maxNumber))")
                .padding()
            Button(action: {
                choiceArray.removeAll()
                for _ in 0...arrayLength - 1 {
                    randomNumber = Int.random(in:
                        1...Int(maxNumber))
                    choiceArray.append(randomNumber)
                    message = "\(choiceArray)"
                }
            }) {
                Text("Create array")
            }
            TextEditor(text: $message)
        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Figure 1-1. *Grouping related code together makes it easy to know where to look for certain information*

The specific placement of code is arbitrary, but what's important is that you organize code so it's easy to find. The clearer your code, the easier it will be to fix and modify it later.

Using the // MARK: Comment

Besides physically grouping related items together such as functions and variables, you can also make searching for groups of related code easier by using the // MARK: comment. By placing a // MARK: comment, followed by descriptive text, you can make it easy to jump from one section of code to another through Xcode's pull-down menu as shown in Figure 1-2.

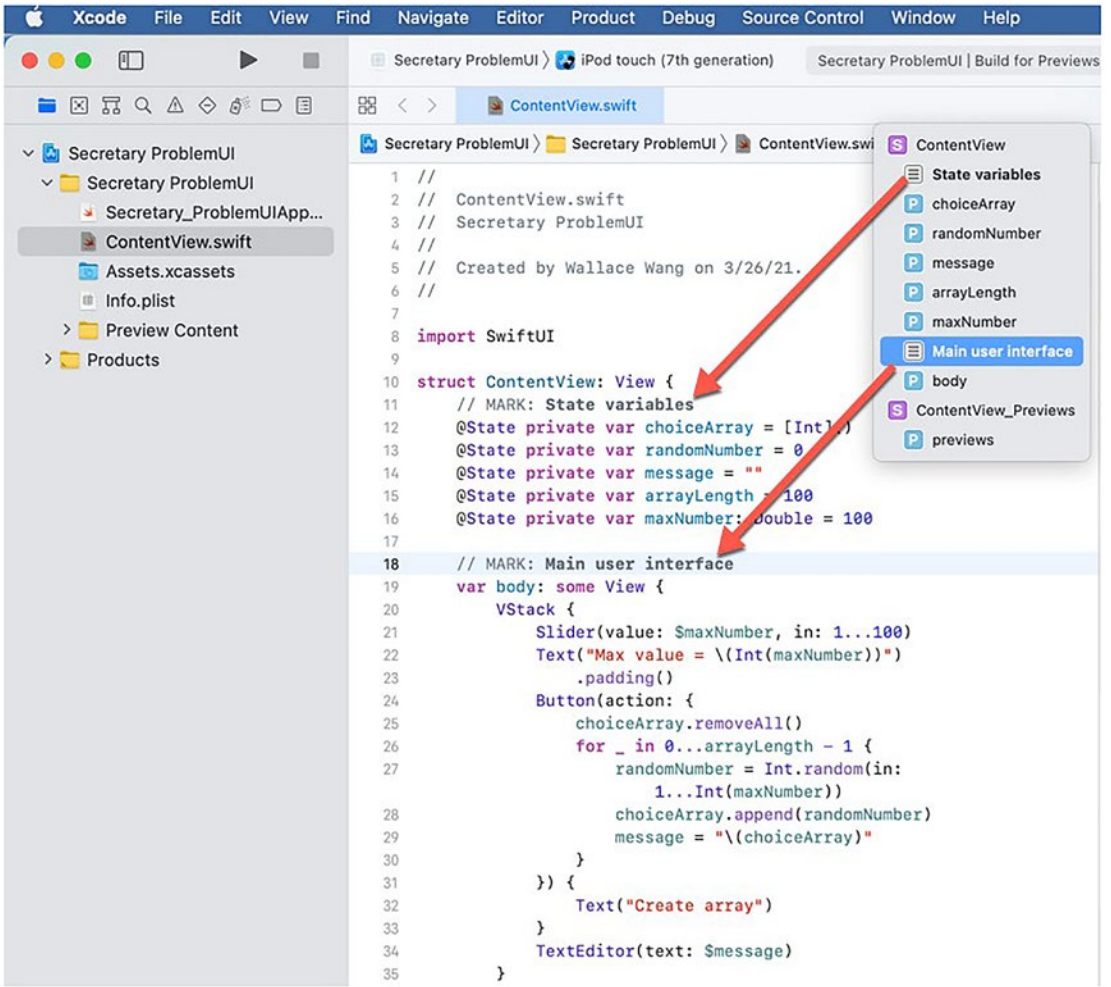

Figure 1-2. *The // MARK: comment creates categories in Xcode's pull-down menus*

The structure of the // MARK: comment looks like this:

```
// MARK: Descriptive text
```

The two // symbols define a comment. The MARK: text tells Xcode to create a pull-down menu category. The descriptive text can be any arbitrary text you want to identify the code that appears underneath.

Once you've defined one or more // MARK: comments, you can quickly jump to any of them by clicking on the last item displayed above Xcode's middle pane to open a pull-down menu as shown in Figure 1-3.

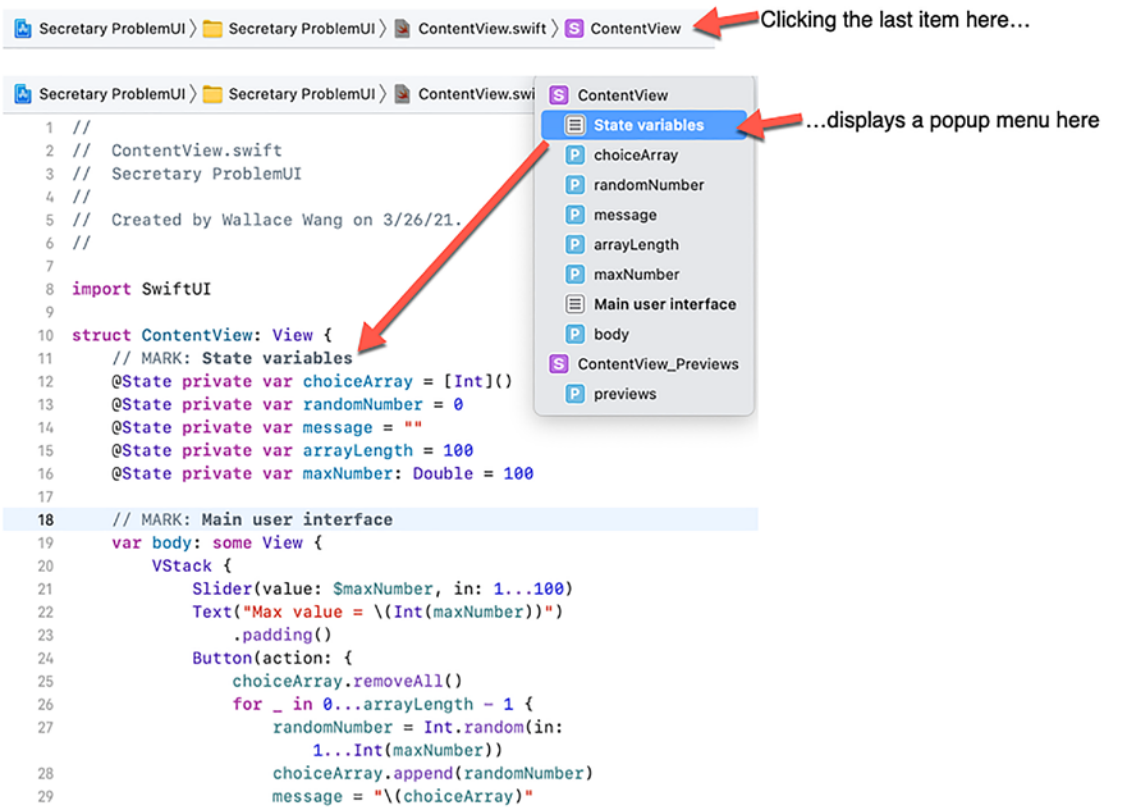

Figure 1-3. *Displaying Xcode's pull-down menu that lists all // MARK: comments*

If you choose Editor ➤ Minimap, you can toggle between opening or hiding the minimap, which displays a thumbnail view of your code. If you have // MARK: comments in your code, the minimap will display them, as shown in Figure 1-4, so you can click on any // MARK: comment to jump to that part of your code.

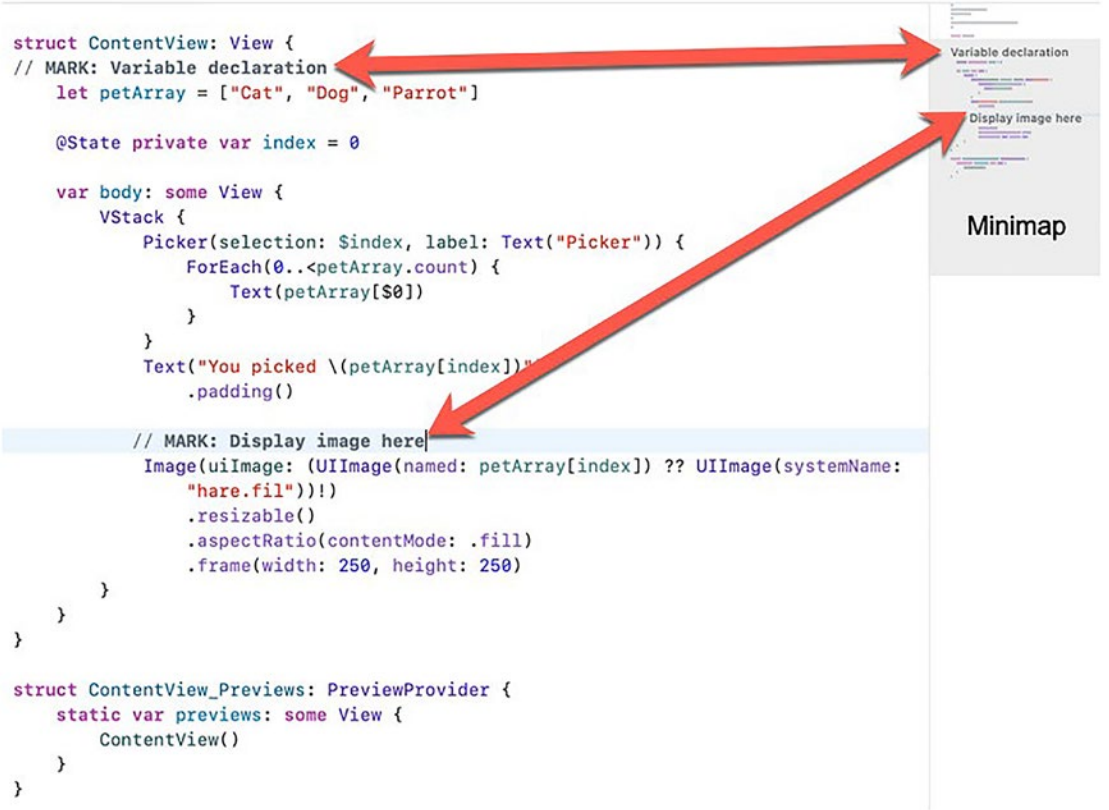

Figure 1-4. *The minimap displays // MARK: comments for easy navigation*

Use the // MARK: comment generously throughout each .swift file. This will make it easy to jump to different parts of your code to modify or study later.

Using Files and Folders

Theoretically, you could create a single file and cram it full of code. While this would work, it's likely to be troublesome to read and modify. A far better solution is to divide your project into multiple files and store those multiple files in separate folders in Xcode's Navigator pane.

Separate files and folders exist solely for your benefit to organize your project. Xcode ignores all folders and treats separate files as if they were all stored in a single file. When creating separate files, the two most common types of files to create are shown in Figure 1-5:

- SwiftUI View
- Swift File

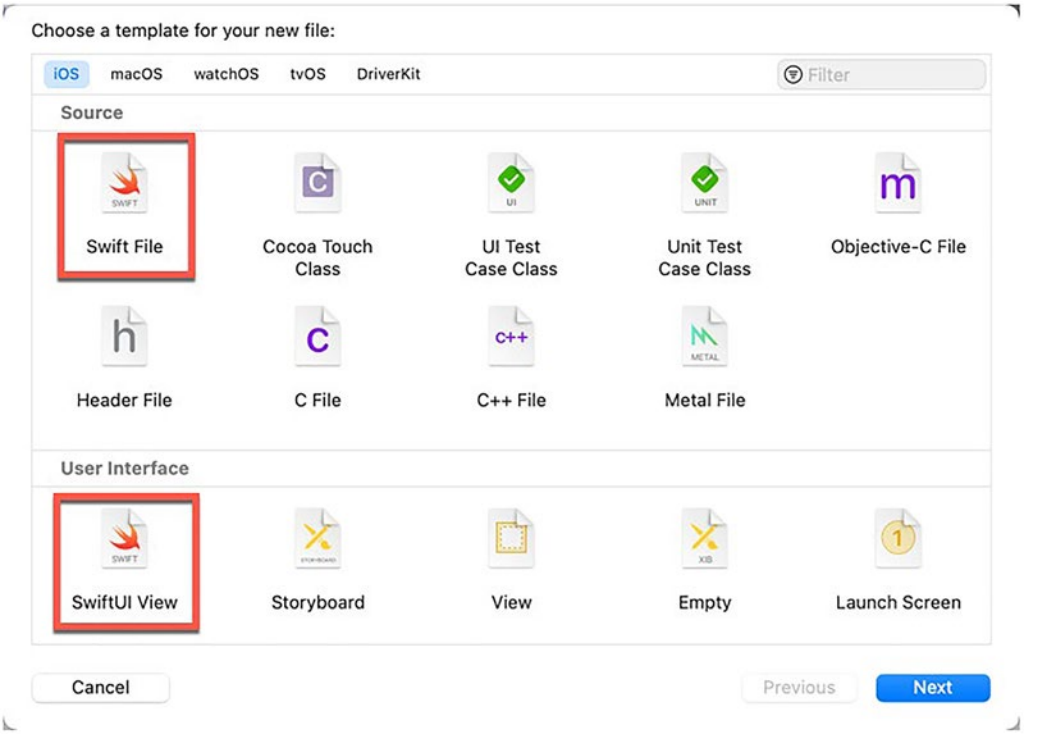

Figure 1-5. *The two most common types of .swift files in a project*

SwiftUI View files define user interfaces that appear on an iOS screen.

The Swift File option creates a blank .swift file for storing and isolating code such as defining a list of variables, data structures, or classes.

The more .swift files you add to a project, the harder it can be to find any particular file. To help organize all the files that make up a project, Xcode lets you create folders. By using folders, you can selectively hide or display the contents of a folder as shown in Figure 1-6.

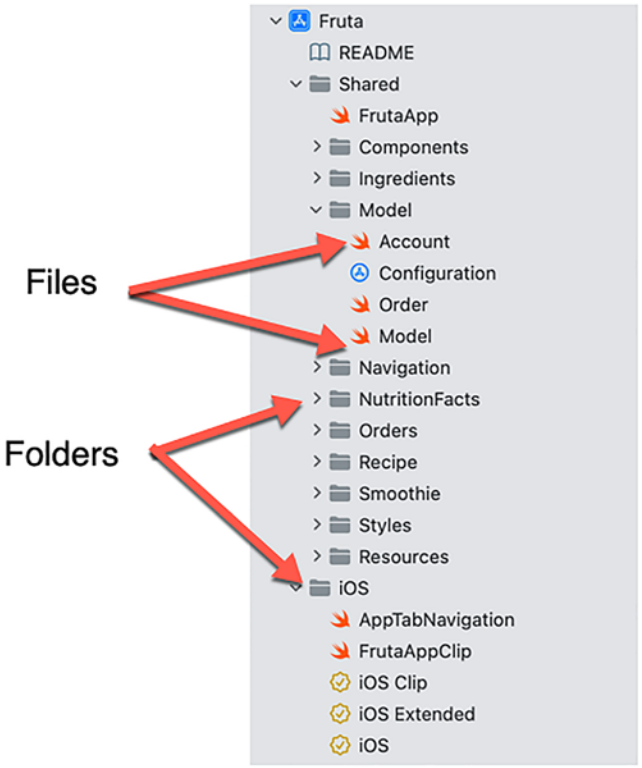

Figure 1-6. *Folders help organize all the files in a project*

To create an empty folder, choose File ➤ New ➤ Group. Once you've created an empty folder, you can drag and drop other folders or files into that empty folder.

Another option is to select one or more files and/or folders by holding down the Command key and clicking on a different file and/or folder. Then choose File ➤ New ➤ Group from Selection. This creates a new folder and automatically stores your selected items into that new folder.

You can also right-click in the Navigator pane to display a pop-up menu with the New Group or New Group from Selection commands as shown in Figure 1-7.

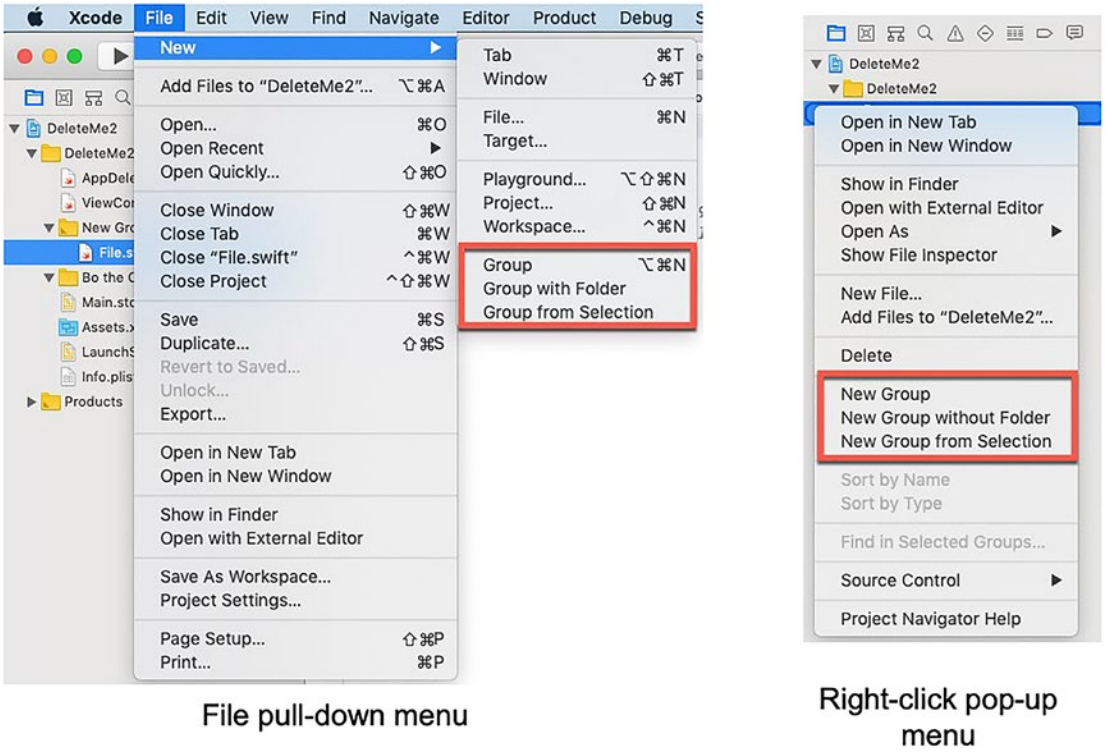

Figure 1-7. *Menu commands to create a new folder*

Note If the Group or Group from Selection commands are greyed out, click on a file to select it before choosing the File ➤ New ➤ Group or File ➤ New ➤ Group from Selection command.

Once you've created a folder, you can always delete that folder afterwards. To delete a folder, follow these steps:

1. Click on the folder you want to delete in the Navigator pane.
2. Choose Edit ➤ Delete, or right-click on the folder and when a pop-up menu appears, choose Delete. If the folder is not empty, Xcode displays a dialog to ask if you want to remove references to any stored files in that folder or just delete them all as shown in Figure 1-8.

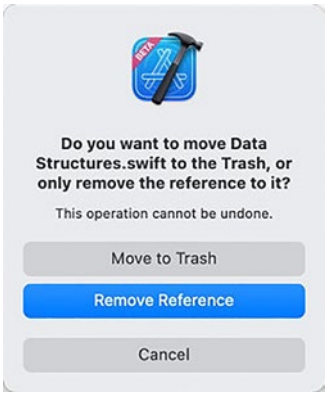

Figure 1-8. *Xcode alerts you if you're deleting a folder that contains files*

Note Deleting a folder also deletes its contents as well, which can include other folders and files.

---

3. Click the Move to Trash button to delete the files completely (or click Remove Reference to keep the file and disconnect the file from your project but without deleting it).

Use Code Snippets

Remembering the exact syntax to create switch statements or for loops in Swift can be troublesome. As a shortcut, Xcode offers code snippets, which let you insert generic code in your Swift files that you can customize afterwards. This lets you focus on the purpose of your code without worrying about the specifics of how Swift implements a particular way of writing branching or looping statements. In addition, code snippets help you write consistent code that's formatted the same way.

To use code snippets, follow these steps:

1. Click in the Swift file where you want to type code.
2. Click the Library icon. A window appears.
3. Click the Snippets icon to display code snippets as shown in Figure 1-9.

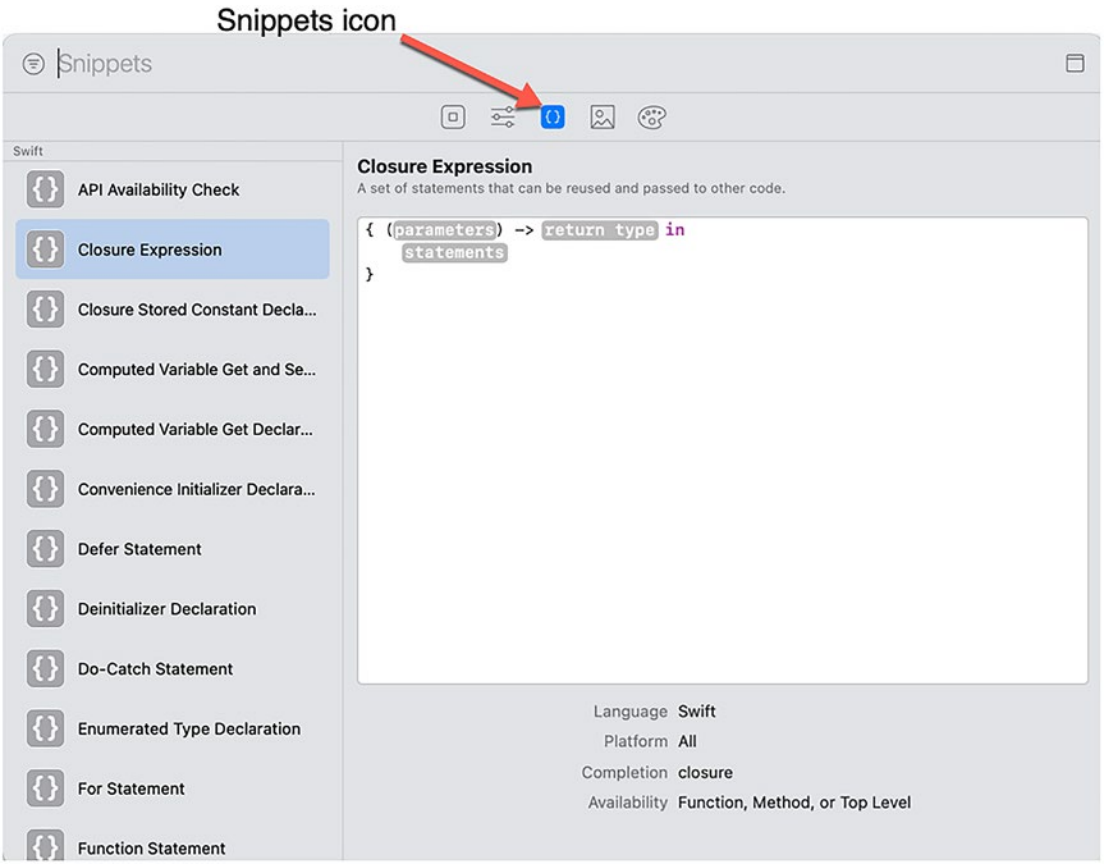

Figure 1-9. *The Code Snippets window*

4. Scroll through the Code Snippets window and click on a snippet you want to use. Xcode displays a brief description of that code snippet.

5. Drag a snippet from the Code Snippet window and drop it in your Swift file. Xcode displays your snippet with placeholders for customizing the code with your own data as shown in Figure 1-10.

```
do {
    try throwing expression
} catch pattern {
    statements
}
```

Figure 1-10. *A code snippet ready for customization*

Creating Custom Code Snippets

The Code Snippet window can make it easy to use common types of Swift statements without typing them yourself. However, you might create your own code that you might want to save and reuse between multiple projects. Rather than copy and paste from one project to another, you can store your own code in the Code Snippet window.

To store your own code as a snippet, follow these steps:

1. Select the code you want to store.
2. Choose Editor ➤ Create Code Snippet, or right-click on your selected code and when a pop-up menu appears, choose Create Code Snippet as shown in Figure 1-11. Xcode adds your selected code to the Code Snippet window as shown in Figure 1-12.

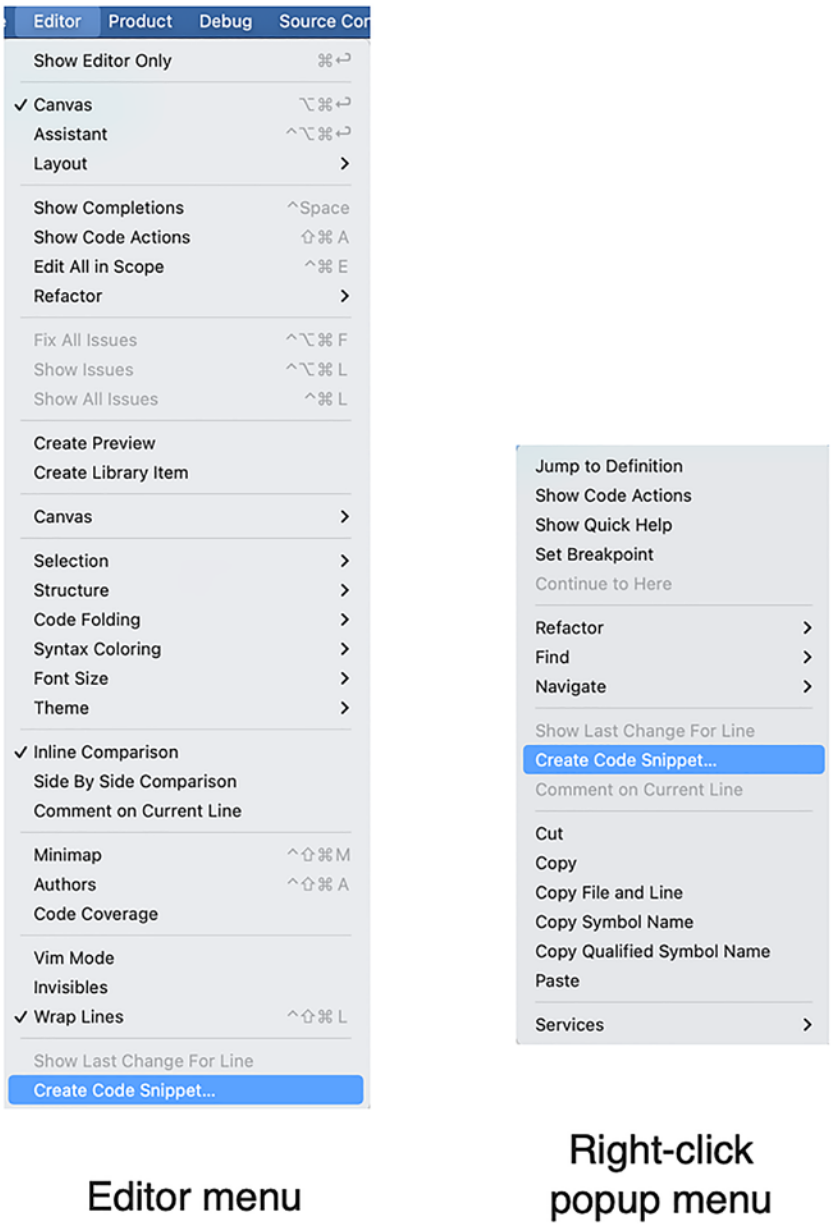

Figure 1-11. *The Create Code Snippet command for adding your own code to the Code Snippet library*

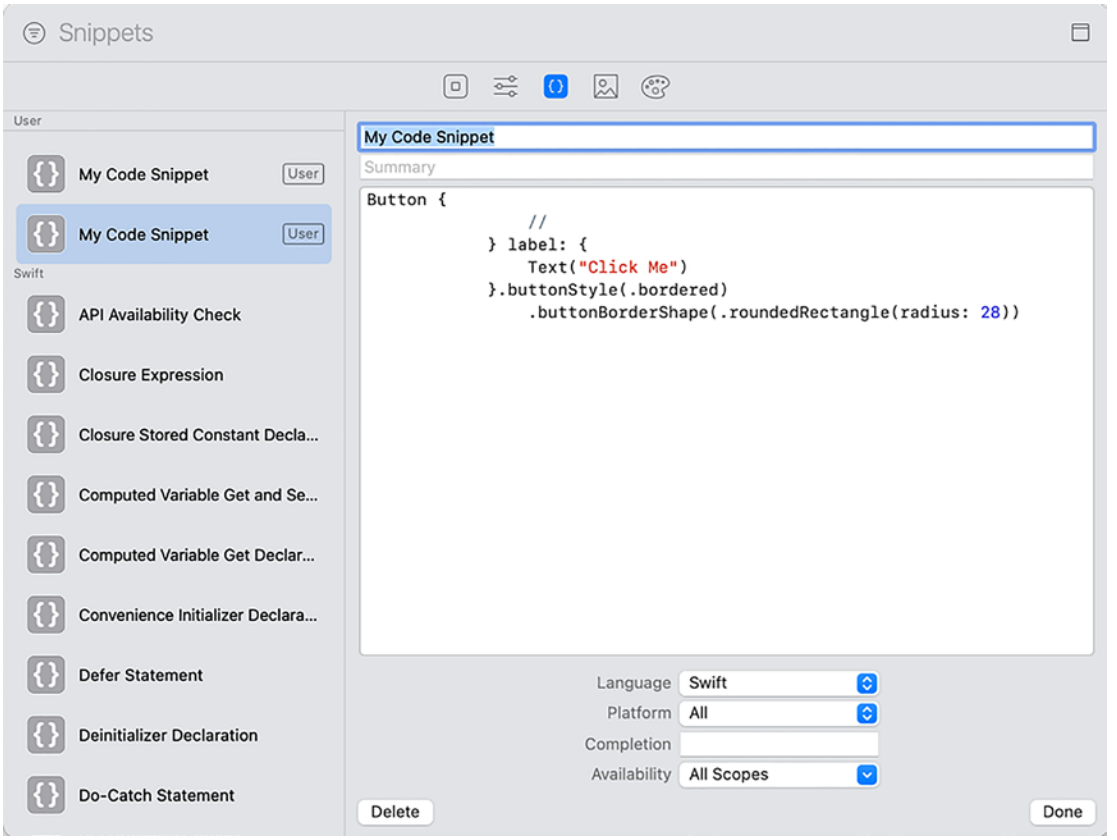

Figure 1-12. *Adding custom code to the Code Snippet window*

3. Click in the Title text field and type a descriptive name for your code snippet. You may also want to edit your code or modify other options as well.

4. Click Done. From now on, you'll be able to use your custom code snippet in any Xcode project.

Editing and Deleting Custom Code Snippets

After adding one or more code snippets, you may want to delete them. You can only delete any code snippets you added to Xcode; you can never delete any of Xcode's default code snippets. To delete a user-defined code snippet from the Code Snippet window, follow these steps:

1. Click on a Swift file in the Navigator pane.
2. Click the Library icon to open the library window.
3. Click the Snippets icon.
4. Click on the code snippet you want to edit or delete.
5. Click the Edit button. Now you can modify the code and when you're finished, click Done. (Or click the Delete button. When Xcode asks if you really want to delete the code snippet, click Delete.)

Use View Modifiers

When designing user interfaces in SwiftUI, you typically create a view (such as Text) and then apply modifiers. If you tend to use the same modifiers over and over again, you can duplicate these modifiers for multiple views. However, duplicating modifiers can take up space and make code harder to read as shown here:

```
Text ("This is the first line")
    .font(.title)
    .foregroundColor(.yellow)
    .background(Color.blue)
    .cornerRadius(6)
    .padding()
Text ("Second line here")
    .font(.title)
    .foregroundColor(.yellow)
    .background(Color.blue)
    .cornerRadius(6)
    .padding()
```

Besides taking up space with duplicate code, another problem is if you want to change one or more modifiers such as changing the font size or the background color. With duplicate modifiers, you need to modify every copy, increasing the chance you'll miss one or more copies.

A better solution is to store commonly used groups of modifiers together in a separate structure defined as a ViewModifier. Then you can apply this ViewModifier structure to multiple views. Now if you need to change these modifiers, you can change them in one place rather than in multiple places throughout your code.

In the preceding example, it makes sense to store the modifiers within a ViewModifier structure like this:

```
struct MyStyle: ViewModifier {
    func body(content: Content) -> some View {
        content
            .font(.title)
            .foregroundColor(.yellow)
            .background(Color.blue)
            .cornerRadius(6)
            .padding()
    }
}
```

This ViewModifier structure encloses all the modifiers, which can then be applied to any view by using .modifier followed by the name of your structure like this:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text ("This is the first line")
                .modifier(MyStyle())
            Text ("Second line here")
                .modifier(MyStyle())
        }
    }
}

struct MyStyle: ViewModifier {
    func body(content: Content) -> some View {
        content
            .font(.title)
            .foregroundColor(.yellow)
```

```
            .background(Color.blue)
            .cornerRadius(6)
            .padding()
    }
}
```

Just as functions let you isolate and reuse code, so can ViewModifiers let you isolate and reuse modifiers for different views.

Summary

Writing iOS apps involves writing new code and modifying existing code. To do both tasks, you need to understand how any existing code works so you don't accidentally duplicate or break it. In many cases, you'll have to edit other people's code, which may or may not have been written in a clear, understandable manner.

Although you can't control how other programmers write code, you can control how you write code. The general principle is to write code that's easy to understand. This can involve adding comments (especially // MARK: comments to make it easy to jump to specific parts of your code). You should also use descriptive variable names and organize the related code in logical groups. You can do that by storing different parts of your code together. You can also organize code by storing code in separate files that you can group in folders.

To insure you write common Swift statements in a consistent manner, you can use code snippets to insert the basic Swift code for you. Then you just have to customize it with your own data. For more flexibility, store your own code in the Code Snippet window. That way you can reuse your own code between multiple projects in Xcode.

Organizing code is never necessary, but since most programs are modified multiple times, proper organization ahead of time can make modifying code much easier. Always assume that someone else will modify your code and make it easy on that person for the future, especially because that person could be you.

CHAPTER 2

Debugging Code

In the professional world of software, you'll actually spend more time modifying existing programs than you ever will creating new ones. When writing new programs or editing existing ones, it doesn't matter how much experience or education you might have because even the best programmers can make mistakes. In fact, you can expect that you will make mistakes no matter how careful you may be. Once you accept this inevitable fact of programming, you need to learn how to find and fix your mistakes.

In the world of computers, mistakes are commonly called "bugs," which gets its name from an early computer that used physical switches to work. One day the computer failed and when technicians opened the computer, they found that a moth had been crushed within a switch, preventing the switch from closing. From that point on, programming errors have been called bugs and fixing computer problems has been known as debugging.

Three common types of computer bugs are

- Syntax errors – Occurs when you misspell something such as a keyword, variable name, function name, class name, or use a symbol incorrectly
- Logic errors – Occurs when you use commands correctly, but the logic of your code doesn't do what you intended
- Runtime errors – Occurs when a program encounters unexpected situations such as the user entering invalid data or when another program somehow interferes with your program unexpectedly

Syntax errors are the easiest to find and fix because they're merely misspellings of variable names that you created or misspelling of Swift commands that Xcode can help you identify. If you type a Swift keyword such as "var" or "let," Xcode displays that keyword in pink (or whatever color you specify for displaying keywords in the Xcode editor).

W. Wang, *Pro iPhone Development with SwiftUI*, https://doi.org/10.1007/978-1-4842-7827-7_2

Now if you type a Swift keyword and it doesn't appear in its usual identifying color, then you know you probably typed it wrong somehow. By coloring your code, Xcode's editor helps you visually identify common misspellings or typos.

Besides using color, the Xcode editor provides a second way to help you avoid mistakes when you need to type the name of a method or class. As soon as Xcode recognizes that you might be typing a known item, it displays a pop-up menu of possible options. Now instead of typing the entire command yourself, you can simply select a choice in the pop-up menu and press the Tab or Enter key to let Xcode type your chosen command correctly as shown in Figure 2-1.

Figure 2-1. *Xcode displays a menu of possible commands you might want to use*

Syntax errors often keep your program from running at all. When a syntax error keeps your program from running, Xcode can usually identify the line (or the nearby area) of your program where the misspelled command appears so you can fix it as shown in Figure 2-2.

Figure 2-2. *Syntax errors often keep a program from running, which allows Xcode to identify the syntax error*