



Pro iPhone Development with Swift 5

Design and Manage Top Quality Apps

—

Second Edition

—

Wallace Wang

Apress®

Pro iPhone Development with Swift 5

Design and Manage Top Quality Apps

Second Edition

Wallace Wang

Apress®

Pro iPhone Development with Swift 5: Design and Manage Top Quality Apps

Wallace Wang
San Diego, CA, USA

ISBN-13 (pbk): 978-1-4842-4943-7
<https://doi.org/10.1007/978-1-4842-4944-4>

ISBN-13 (electronic): 978-1-4842-4944-4

Copyright © 2019 by Wallace Wang

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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

Cover image designed by Freepik (www.freepik.com)

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 rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

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 via the book's product page, located at www.apress.com/978-1-4842-4943-7. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

The secret to success is persistence. Never give up, never doubt yourself. The path to any goal will never be easy, but that's exactly what makes striving for goals so rewarding.

Talent, intelligence, and skill are never as important as persistence. Remember, never be afraid of failure. Be afraid of giving up too soon and never realizing your true potential in whatever dream you want to achieve. You may not always reach your dreams, but pursuing big dreams will always give you a far richer life than if you never bothered trying at all.

Table of Contents

About the Author	xi
About the Technical Reviewer	xiii
Chapter 1: Organizing Code	1
Using the // MARK: Comment.....	3
Using Extensions.....	5
Using Files and Folders.....	11
Use Code Snippets.....	14
Creating Custom Code Snippets	17
Deleting Custom Code Snippets	18
Using @IBDesignable and @IBInspectable	19
Summary.....	26
Chapter 2: Debugging Code	27
Simple Debugging Techniques.....	30
Using the Xcode Debugger.....	34
Using Breakpoints	35
Stepping Through Code	36
Managing Breakpoints.....	41
Using Symbolic Breakpoints.....	45
Using Conditional Breakpoints	48
Summary.....	49

TABLE OF CONTENTS

- Chapter 3: Understanding Closures 51**
 - Closures with Multiple Parameters 54
 - Understanding Value Capturing 56
 - Using Closures Like Data 57
 - Summary 59

- Chapter 4: Multithreaded Programming Using Grand Central Dispatch 61**
 - Understanding Threads 62
 - Using Grand Central Dispatch 68
 - Displaying Feedback 77
 - Using Dispatch Groups 80
 - Summary 87

- Chapter 5: Understanding the Application Life Cycle 89**
 - Getting State-Change Notifications 90
 - Using Execution State Changes 98
 - Active ► Inactive 98
 - Inactive ► Background 98
 - Background ► Inactive 99
 - Inactive ► Active 99
 - Displaying the Launch Screen 99
 - Using the Notification Center 101
 - Summary 111

- Chapter 6: Understanding Data Persistence 113**
 - Storing Preferences in UserDefaults 114
 - Storing Preferences in UserDefaults in the AppDelegate File 120
 - Reading and Writing to Files 131
 - Using Core Data 136
 - Creating a Data Model File 137
 - Customizing a Data Model File 141

Designing the User Interface	145
Writing Swift Code.....	148
Summary.....	152
Chapter 7: Passing Data Between Files	155
Sharing Data with the AppDelegate.swift File.....	155
Sharing Data Between View Controllers	161
Passing Data Forward	163
Passing Data Backward with a Protocol.....	179
Passing Data Backward with a Delegate.....	187
Passing Data with the Notification Center	192
Summary.....	198
Chapter 8: Translating with Localization	201
Designing the User Interface.....	202
Creating a Localization File.....	205
Storing Text.....	209
Creating a Localized String File	213
Localizing Images	222
Customizing the App Name.....	226
Formatting Numbers and Dates.....	228
Summary.....	232
Chapter 9: Using 3D Touch.....	235
Understanding 3D Touch	236
Detecting 3D Touch Availability	239
Detecting Pressure.....	242
Creating Home Screen Quick Actions	245
Responding to Quick Action Items	252
Adding Dynamic Home Screen Quick Actions.....	259
Adding Peeking, Popping, and Previewing.....	266
Summary.....	276

TABLE OF CONTENTS

- Chapter 10: Detecting Motion and Orientation 277**
 - Detecting Shake Gestures..... 277
 - Understanding Core Motion 281
 - Detecting Acceleration 282
 - Detecting Rotation with the Gyroscope 285
 - Detecting Magnetic Fields..... 289
 - Detecting Device Motion Data..... 290
 - Summary..... 292

- Chapter 11: Using Location and Maps 293**
 - Using Core Location 293
 - Defining Accuracy..... 294
 - Defining a Distance Filter 295
 - Requesting a Location 296
 - Retrieving Location Data 296
 - Requesting Authorization 297
 - Adding a Map 298
 - Zooming in a Location..... 303
 - Adding Annotations 307
 - Summary..... 311

- Chapter 12: Playing Audio and Video 313**
 - Playing an Audio File..... 314
 - Playing Video..... 323
 - Playing Videos on the Internet 327
 - Summary..... 332

- Chapter 13: Using the Camera 333**
 - Setting Privacy Settings..... 333
 - Checking for a Camera..... 335
 - Designing a Simple User Interface..... 337
 - Taking a Picture 339

Saving a Picture	340
Summary.....	344
Chapter 14: Using WebKit	345
Displaying Web Pages from the Internet.....	345
Displaying HTML Files.....	351
Summary.....	356
Chapter 15: Displaying Animation	357
Moving Items with Animation.....	358
Customizing Animation with Delays and Options.....	363
Customizing Animation with Damping and Velocity	367
Resizing Items with Animation.....	369
Rotating Items with Animation.....	372
Changing Transparency with Animation.....	376
Animating Transitions Between View Controllers.....	379
Simple Animation Transition Between View Controllers.....	393
Summary.....	398
Chapter 16: Using Machine Learning.....	399
Understanding Machine Learning	400
Finding a Core ML Model	402
Image Recognition	403
Identifying Objects from the Camera	415
Analyzing an Image.....	424
Summary.....	432
Chapter 17: Using Facial and Text Recognition	433
Recognizing Faces in Pictures	433
Highlighting Faces in an Image.....	442
Highlighting Parts of a Face in an Image	450
Recognizing Text in an Image	458
Summary.....	469

TABLE OF CONTENTS

- Chapter 18: Using Speech..... 471**
 - Converting Speech to Text 471
 - Recognizing Spoken Commands..... 482
 - Turning Text to Speech..... 486
 - Summary..... 490

- Chapter 19: Understanding SiriKit..... 491**
 - How SiriKit Works 492
 - Defining How Siri Interacts with the User 497
 - Understanding the IntentHandler.swift File..... 500
 - Understanding the ExtensionUI Folder..... 504
 - Creating a Payment App with Siri 510
 - Summary..... 518

- Chapter 20: Understanding ARKit..... 519**
 - How ARKit Works..... 519
 - Drawing Augmented Reality Objects..... 528
 - Resetting the World Origin 531
 - Drawing Custom Shapes..... 537
 - Modifying the Appearance of Shapes 539
 - Playing with Lighting 548
 - Summary..... 553

- Chapter 21: Interacting with Augmented Reality 555**
 - Storing and Accessing Graphic Assets..... 556
 - Working with Touch Gestures..... 559
 - Detecting a Horizontal Plane..... 565
 - Modifying an Image 571
 - Creating Virtual Objects 572
 - Summary..... 584

- Index..... 585**

About the Author

Wallace Wang has written dozens of computer books over the years beginning with ancient MS-DOS programs like WordPerfect and Turbo Pascal, migrating to writing books on Windows programs like Visual Basic and Microsoft Office, and finally switching to Swift programming for Apple products like the Macintosh and the iPhone. He currently teaches iOS programming through UCSD Extension in San Diego.

When he's not helping people discover the fascinating world of programming, he performs stand-up comedy and appears on two radio shows on KNSJ in San Diego (<http://knsj.org>) called *Notes from the Underground* (with Dane Henderson, Jody Taylor, and Kristen Yoder) and *Laugh In Your Face Radio* (with Chris Clobber and Sarah Burford).

He also writes a screenwriting/storytelling blog called *The 15 Minute Movie Method* (<http://15minutemoviemethod.com>) designed for screenwriters and novelists. For fun, he also writes a blog about the latest cat news on the Internet called *Cat Daily News* (<http://catdailynews.com>).

About the Technical Reviewer

Massimo has more than 22 years of experience in Security, Web and Mobile Development, Cloud, and IT Architecture. His true IT passions are Security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science in Computing Science from the University of Salerno, Italy.

He has worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect for many years.

His technical skills include Security, Android, Cloud, Java, MySQL, Drupal, Cobol, Perl, Web and Mobile Development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc.

He currently works as Chief Information Security Officer (CISO) for Cargotec Oyj.

CHAPTER 1

Organizing Code

Programs are rewritten and modified far more often than they are ever created. That means most of the time developers will be changing and altering 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

Writing code in a consistent and understandable style means predictability. For example, some programmers give all IBOutlet variables a prefix of "IB" to stand for IBOutlet such as

```
@IBOutlet var IBtitleLabel: UILabel!
```

This type of programming style makes it easy to tell the difference between using an IBOutlet variable and an ordinary variable. Other programmers add a prefix or suffix to variable names to identify the type of data they contain such as

```
var nameStr : String  
var ageInt : Int  
var salaryDb1 : Double
```

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 IBOutlet variables near the top and placing IBAction methods at the bottom with ordinary functions in the middle. This can group chunks of code in specific places to make looking for specific code easier as shown in Figure 1-1.

IBOutlets and variables

Functions

IBAction methods

```
import Foundation
import UIKit

class ViewController: UIViewController, UICollectionViewDataSource, UICollectionViewDelegate {

    @IBOutlet weak var photosCollectionView:UICollectionView!

    @IBOutlet weak var maxDegreesValueLabel:UILabel!
    @IBOutlet weak var coverDensityValueLabel:UILabel!
    @IBOutlet weak var minOpacityValueLabel:UILabel!
    @IBOutlet weak var minScaleValueLabel:UILabel!

    @IBOutlet weak var maxDegreesSlider: UISlider!
    @IBOutlet weak var coverDensitySlider: UISlider!
    @IBOutlet weak var minScaleSlider: UISlider!
    @IBOutlet weak var minOpacitySlider: UISlider!

    var originalItemSize = CGSize.zero
    var originalCollectionViewSize = CGSize.zero

    // MARK: Lifecycle
    override func viewDidLoad() {
        super.viewDidLoad()

        originalCollectionViewSize = photosCollectionView.bounds.size
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + Double(Int64(2 * NSEC_PER_SEC)) * Double(NSEC_PER_SEC)) {
            self.photosCollectionView.reloadData()
        }
    }

    @IBAction func densitySliderValueChanged(_ sender:UISlider) {
        photosCollectionView.reloadData()
    }

    @IBAction func opacitySliderValueChanged(_ sender:UISlider) {
        photosCollectionView.reloadData()
    }

    @IBAction func scaleSliderValueChanged(_ sender:UISlider) {
        photosCollectionView.reloadData()
    }
}
```

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

The exact grouping of different parts of code is arbitrary, but what’s important is that you organize code so it’s easy to find what you want.

Using the // MARK: Comment

Besides physically grouping related items together such as IBOutlets 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.

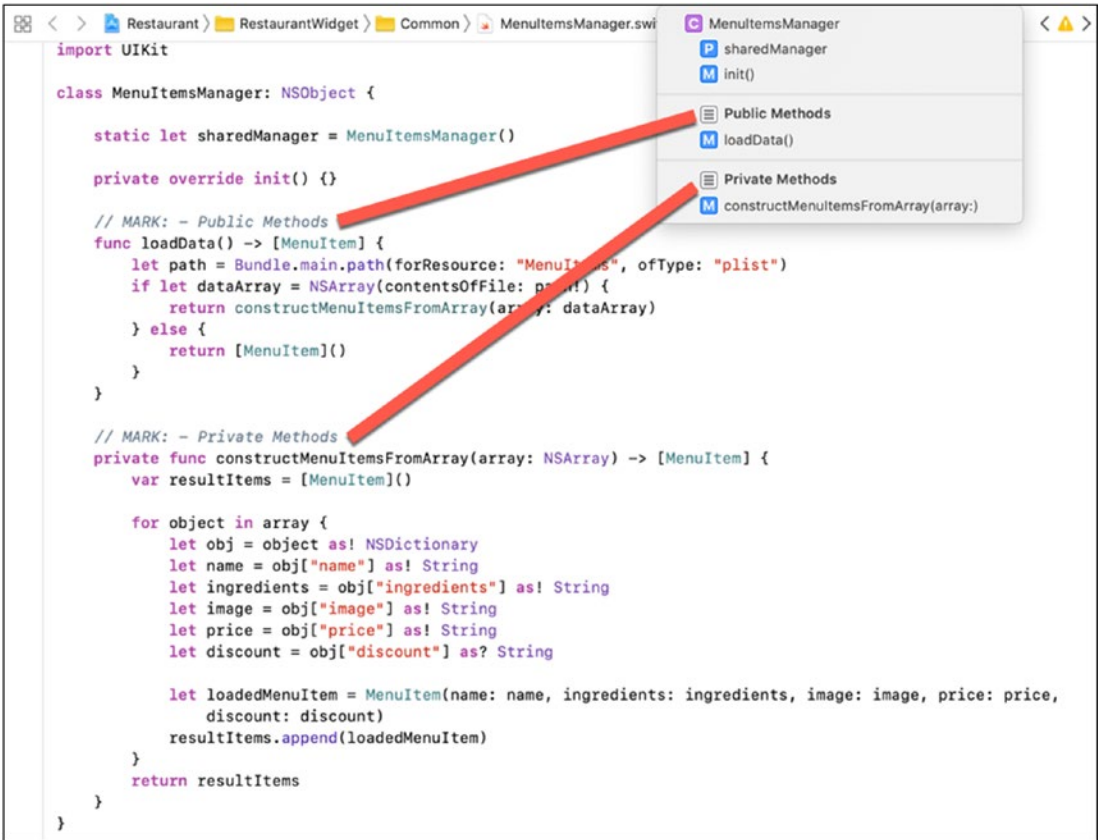


Figure 1-2. The // MARK: comment creates categories in Xcode’s pull-down menus

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 the last item displayed above Xcode’s middle pane to open a pull-down menu as shown in Figure 1-3.

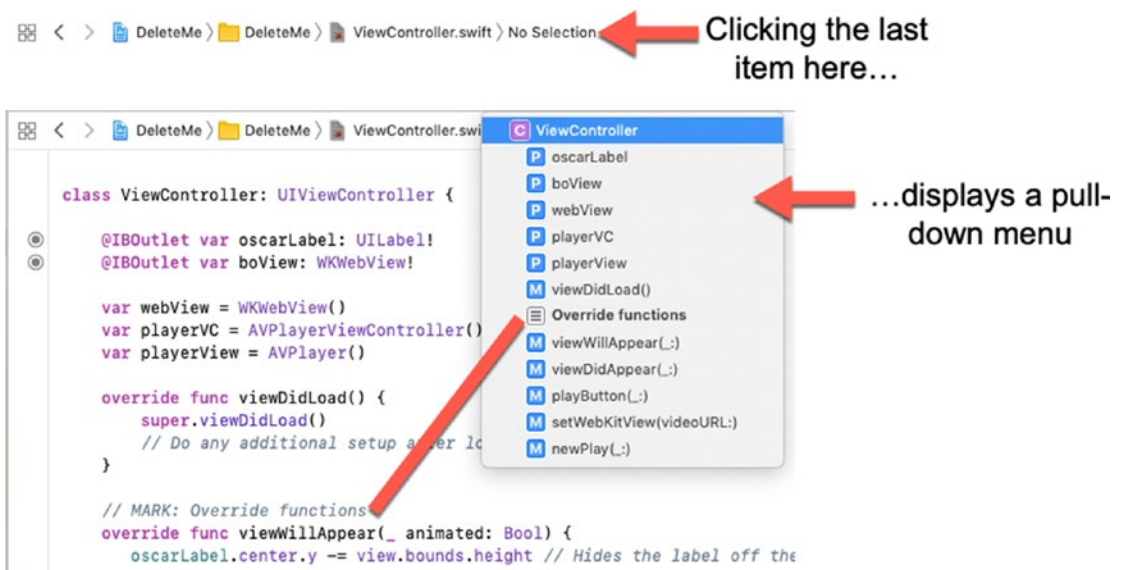


Figure 1-3. Displaying Xcode’s pull-down menu that lists all // MARK: comments

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

Using Extensions

When creating different classes, it’s likely you’ll need to extend them. For example, a class file that uses table views often needs to extend its class with UITableViewDataSource and UITableViewDelegate such as

```
class ViewController: UIViewController, UITableViewDelegate,
UITableViewDataSource {
```

Once you extend a class, you need to implement its required functions. For example, extending a class with UITableViewDataSource requires that you include the following two functions:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    // Code here
}
```

```

func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    // Code here
}

```

You can place these two functions anywhere in your .swift file, but it's generally a good idea to keep these two functions together. If you extend a ViewController class with UITableViewDelegate and UITableViewDataSource, the entire ViewController.swift file might look like this:

```

import UIKit

class ViewController: UIViewController, UITableViewDelegate,
UITableViewDataSource {

    @IBOutlet var petTable: UITableView!

    let petArray = ["cat", "dog", "parakeet", "parrot", "canary", "finch",
    "tropical fish", "goldfish", "sea horses", "hamster", "gerbil",
    "rabbit", "turtle", "snake", "lizard", "hermit crab"]

    let cellID = "cellID"

    override func viewDidLoad() {
        super.viewDidLoad()
        petTable.dataSource = self
        petTable.delegate = self
        // Do any additional setup after loading the view, typically from a nib.
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
        return petArray.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
        var cell = tableView.dequeueReusableCell(withIdentifier: cellID)
        if (cell == nil) {
            cell = UITableViewCell(

```

```

        style: UITableViewCell.CellStyle.default,
        reuseIdentifier: cellID)
    }
    cell?.textLabel?.text = petArray[indexPath.row]
    return cell!
}

func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    let selectedItem = petArray[indexPath.row]
    let alert = UIAlertController(title: "Your Choice", message: "\
(selectedItem)", preferredStyle: .alert)

    let okAction = UIAlertAction(title: "OK", style: .default, handler:
{ action -> Void in
        //Just dismiss the action sheet
    })
    alert.addAction(okAction)

    self.present(alert, animated: true, completion: nil)
}
}

```

While it's easy to identify the three tableView functions (numberOfRowsInSection, cellForRowAt, and didSelectRowAt), it's not easy to see which functions belong to the UITableViewDelegate and which belong to UITableViewDataSource. Even more troublesome is that it's possible to insert multiple functions in between all three tableView functions.

To make it much easier to see which required functions are required by which class, you can extend a class a second way by adding specific extension code at the end of a class file as follows:

```

import UIKit

class ViewController: UIViewController {
    @IBOutlet var petTable: UITableView!

```

```

let petArray = ["cat", "dog", "parakeet", "parrot", "canary", "finch",
  "tropical fish", "goldfish", "sea horses", "hamster", "gerbil",
  "rabbit", "turtle", "snake", "lizard", "hermit crab"]

let cellID = "cellID"

override func viewDidLoad() {
    super.viewDidLoad()
    petTable.dataSource = self
    petTable.delegate = self
    // Do any additional setup after loading the view.
}

}

extension ViewController: UITableViewDataSource {
    func tableView(_ tableView: UITableView, numberOfRowsInSection section:
    Int) -> Int {
        return petArray.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath:
    IndexPath) -> UITableViewCell {
        var cell = tableView.dequeueReusableCell(withIdentifier: cellID)
        if (cell == nil) {
            cell = UITableViewCell(
                style: UITableViewCell.CellStyle.default,
                reuseIdentifier: cellID)
        }
        cell?.textLabel?.text = petArray[indexPath.row]
        return cell!
    }
}

extension ViewController: UITableViewDelegate {
    func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
    IndexPath) {
        let selectedItem = petArray[indexPath.row]

```

```

let alert = UIAlertController(title: "Your Choice", message:
"\(selectedItem)", preferredStyle: .alert)

let okAction = UIAlertAction(title: "OK", style: .default, handler:
{ action -> Void in
    //Just dismiss the action sheet
})
alert.addAction(okAction)

self.present(alert, animated: true, completion: nil)
}
}

```

Notice that this method separates the tableView functions from the rest of the ViewController.swift code and explicitly shows that the numberOfRowsInSection and cellForRowAt tableView functions belong to the UITableViewDataSource while the didSelectRowAt tableView function belongs to the UITableViewDelegate.

By using the extension keyword at the end of .swift class files, it's much easier to group and organize related code. With the extension keyword, Xcode automatically identifies extensions in its pull-down menus to make it easier to find as shown in Figure 1-4.

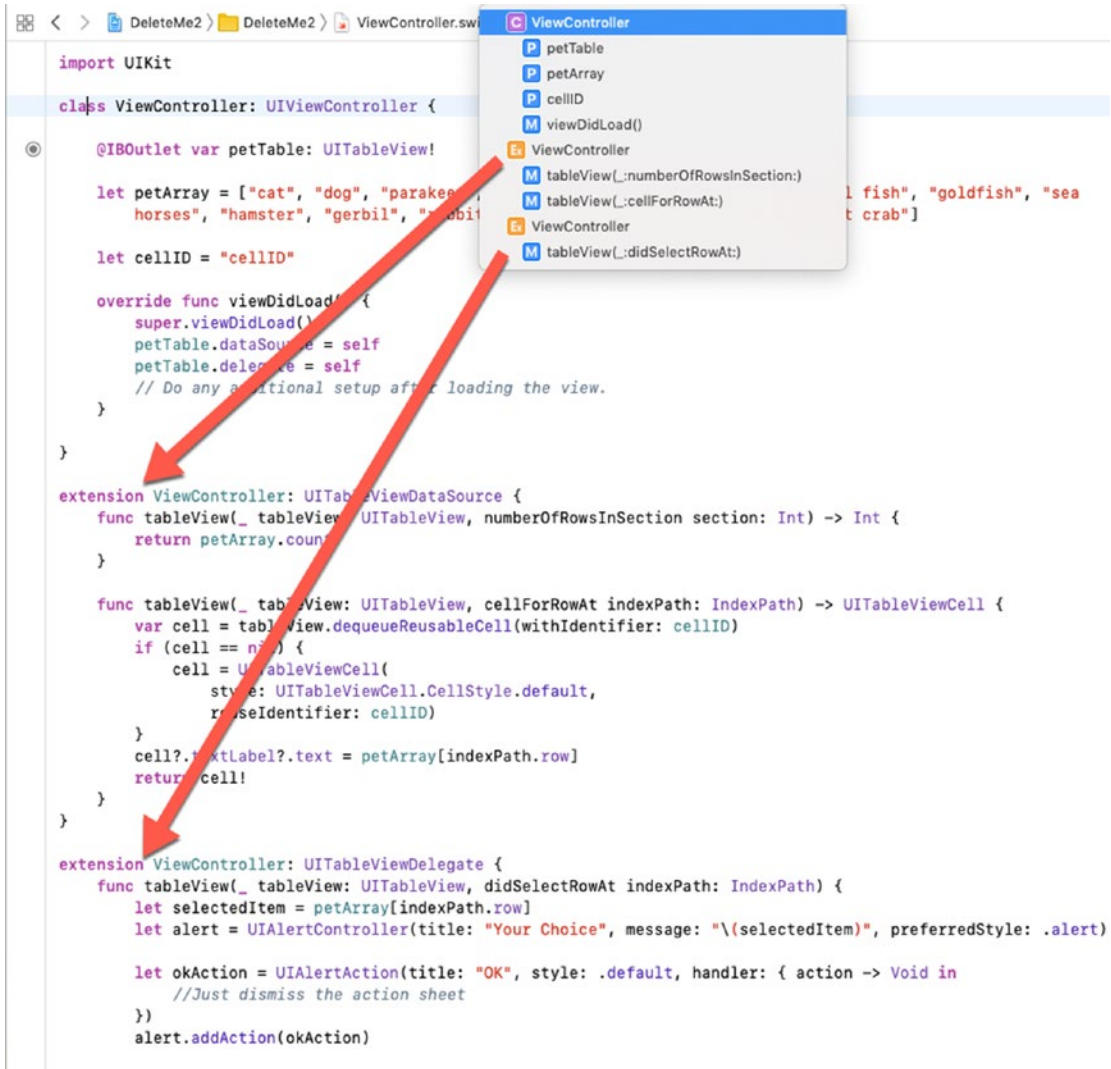


Figure 1-4. Displaying extensions in Xcode’s pull-down menu

The preceding two methods of extending a class are equivalent so it’s just a matter of using which method you like best. Just be aware that using the extension keyword to separate code can help you organize code without any extra work on your part.

Using Files and Folders

Theoretically, you could create a single `ViewController.swift` 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:

- Cocoa Touch Class
- Swift File

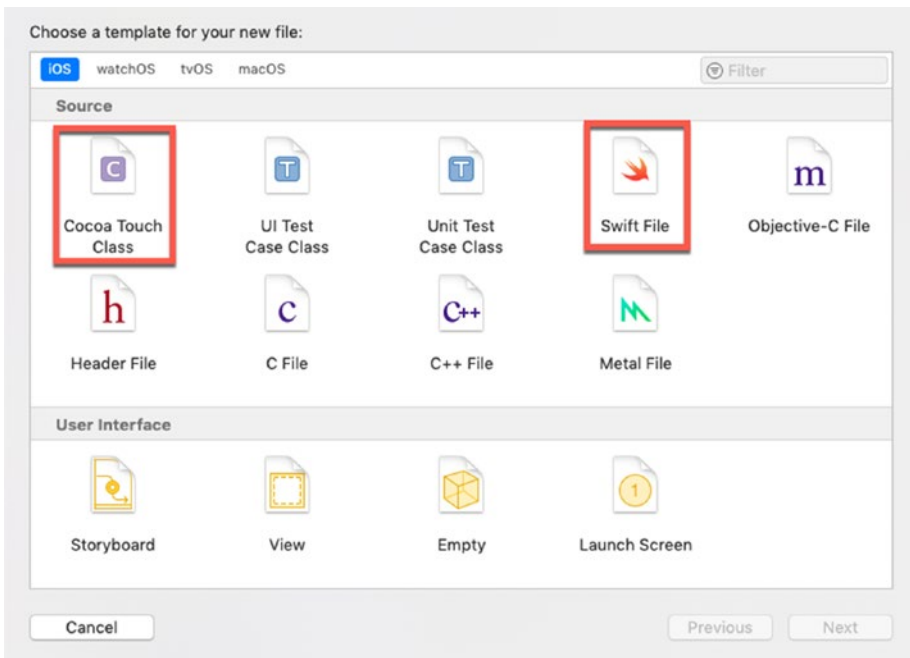


Figure 1-5. The two most common types of `.swift` files in a project

Cocoa Touch Class files are mostly used to connect to view controllers displayed in a storyboard. When you need a .swift file to control part of your app’s user interface, use a Cocoa Touch Class file.

The Swift File option creates blank .swift files which are most often used to store and isolate code that you don’t want to cram in an existing .swift file 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.

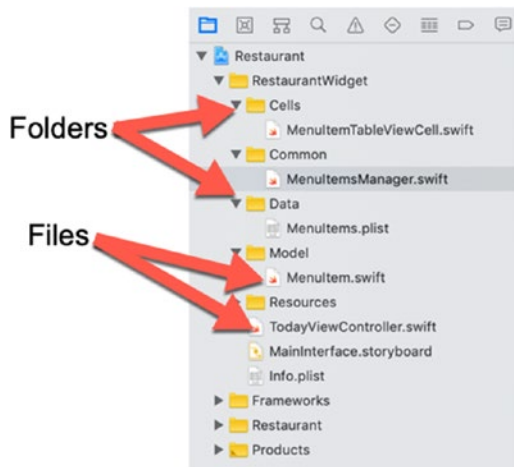


Figure 1-6. Folders help organize all the files in a project

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 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 the Navigator pane to display a popup menu with the New Group or New Group from Selection commands as shown in Figure 1-7.

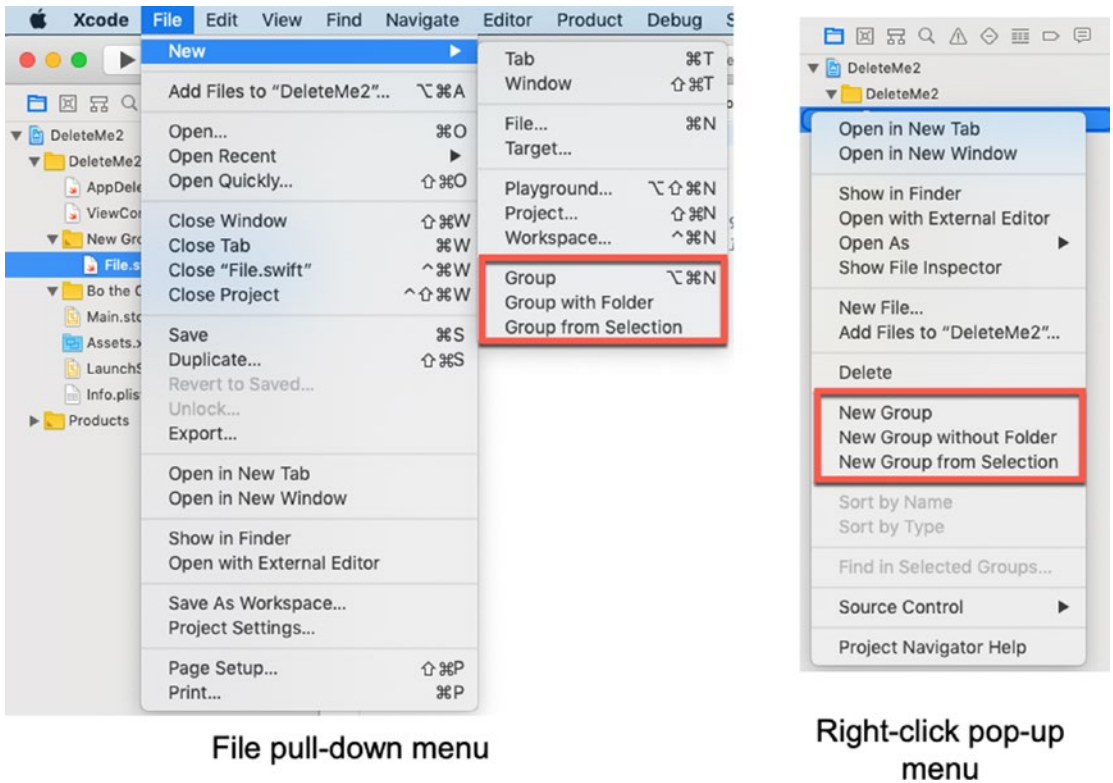


Figure 1-7. Menu commands to create a new folder

Note If the Group or Group from Selection commands are grayed out, click a .swift 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 afterward. To delete a folder, follow these steps:

1. Click the folder you want to delete in the Navigator pane.
2. Choose Edit ► Delete, or right-click the folder, and when a popup 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.

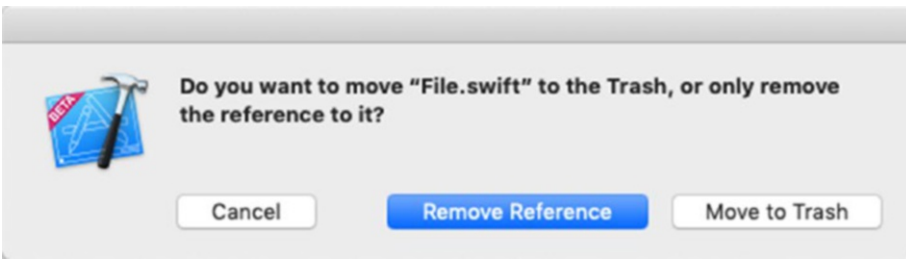


Figure 1-8. Xcode alerts you if you're deleting a folder that contains files

Note Deleting a folder also deletes its contents, 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 afterward. 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 the .swift file where you want to type code.
2. Click the Library icon. The Snippets window appears as shown in [Figure 1-9](#).

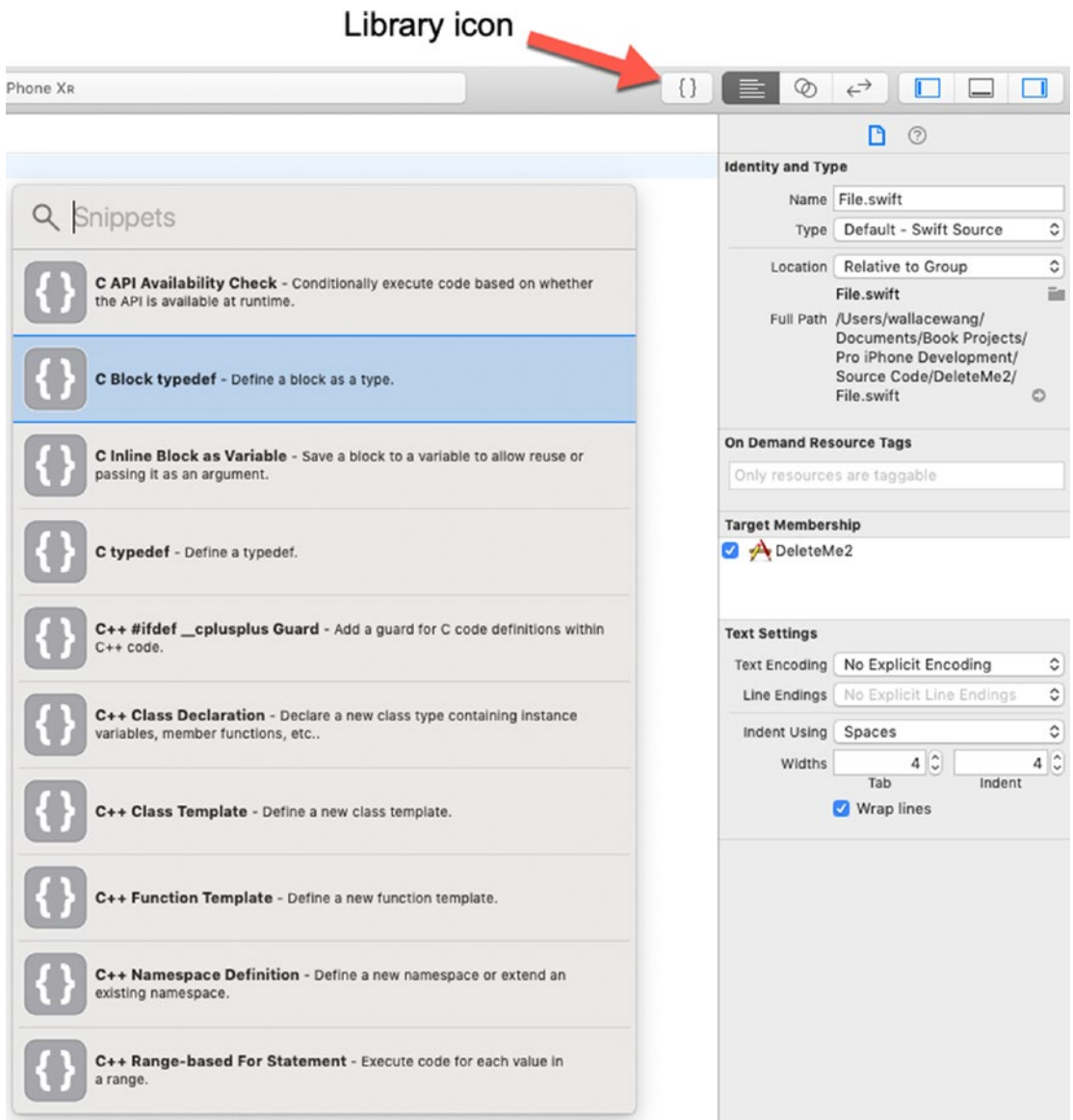


Figure 1-9. The Code Snippets window

3. Scroll through the Code Snippets window and click a snippet you want to use. Xcode displays a brief description of that code snippet as shown in Figure 1-10.

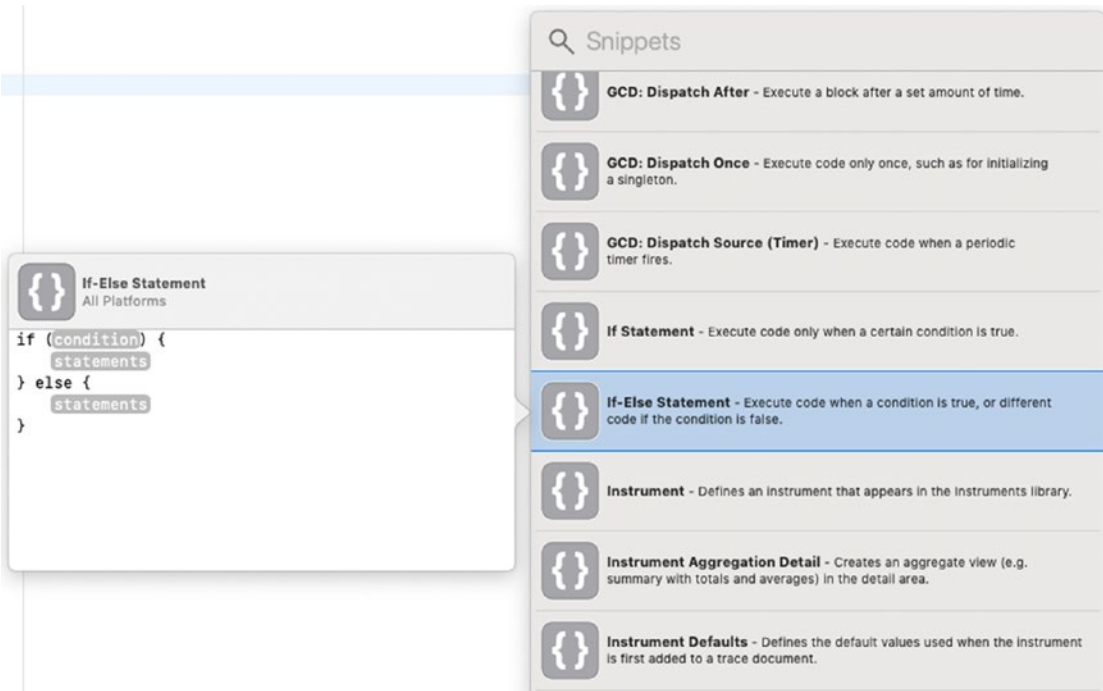


Figure 1-10. *The Code Snippets window*

4. 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 1-11.

```
if (condition) {  
    statements  
} else {  
    statements  
}
```

Figure 1-11. *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 your selected code, and when a popup menu appears, choose Create Code Snippet as shown in Figure 1-12. Xcode adds your selected code to the Code Snippet window as shown in Figure 1-13.

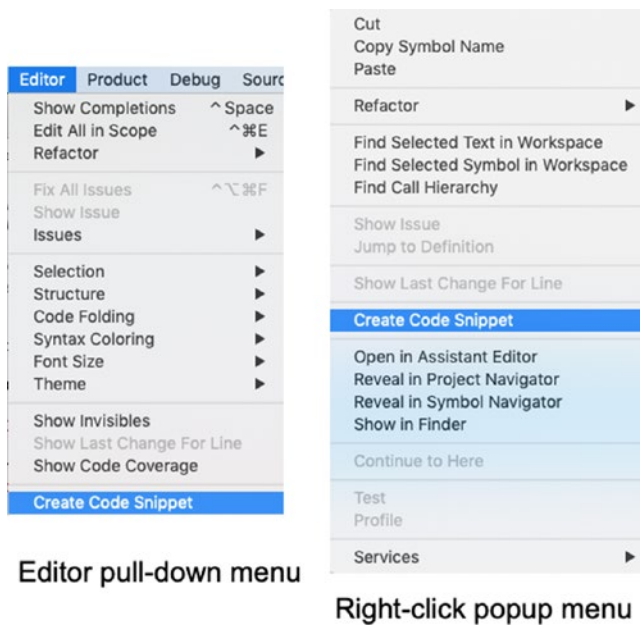


Figure 1-12. The *Create Code Snippet* command for adding your own code to the Code Snippet library

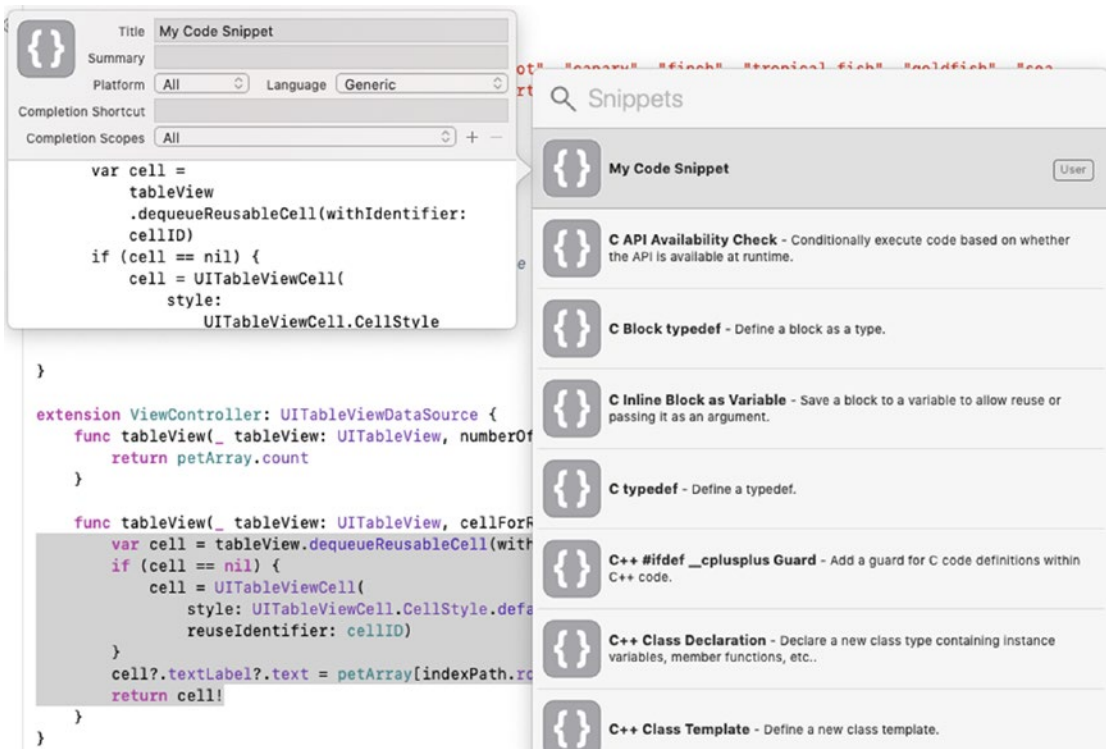


Figure 1-13. Adding custom code to the Code Snippet window

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

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 a .swift file in the Navigator pane.
2. Click the Library icon to open the Code Snippet library.
3. Click the code snippet you want to delete.