

Covers Swift 4, Xcode 9, and iOS 11



# iOS Development with Swift

Craig Grummitt

SAMPLE CHAPTER

 MANNING



*iOS Development with Swift*

by Craig Grummitt

**Sample Chapter 15**

Copyright 2018 Manning Publications

# *brief contents*

---

<b>PART 1</b>	<b>INTRODUCING XCODE AND SWIFT .....</b>	<b>1</b>
1	■ Your first iOS application	3
2	■ Introduction to Swift playgrounds	29
3	■ Swift objects	55
<b>PART 2</b>	<b>BUILDING YOUR INTERFACE .....</b>	<b>81</b>
4	■ View controllers, views, and outlets	83
5	■ User interaction	105
6	■ Adaptive layout	133
7	■ More adaptive layout	167
8	■ Keyboard notifications, animation, and scrolling	199
<b>PART 3</b>	<b>BUILDING YOUR APP .....</b>	<b>225</b>
9	■ Tables and navigation	227
10	■ Collections, searching, sorting, and tab bars	263
11	■ Local data persistence	297
12	■ Data persistence in iCloud	337
13	■ Graphics and media	371
14	■ Networking	409
15	■ Debugging and testing	439
<b>PART 4</b>	<b>FINALIZING YOUR APP .....</b>	<b>479</b>
16	■ Distributing your app	481
17	■ What's next?	513

# 15

## *Debugging and testing*

---

### ***This chapter covers***

- Debugging using different techniques, tools, gauges, and instruments in Xcode
- Testing your app
- Testing your app interface

All's well and good reading a book or following a tutorial, but in the real world things go wrong. And often! This is your chance to put your detective hat on and investigate.

In this chapter, we'll look at what to do when things go wrong by using debugging. We'll also look at how to prevent things from going wrong with testing.

Along the way, we'll explore additional concepts:

- The console
- Variables view
- Breakpoints and the breakpoint navigator
- The debug navigator and gauges
- Instruments
- Unit tests and UI tests



## 15.1 The setup

A friend has kindly offered to look at your app and see if they can find any bugs. You sent them a link to the GitHub repo for your Xcode project, and a few days later you got this email in return:

*Hey—I've had a look at the app for you. It's looking good, but I also found a few odd problems:*

- *The book edit form was working well to begin with, but then it started crashing. Don't know what that's about.*
- *The Cancel button in the book edit form crashes the app.*
- *After you add an image and save it, the next time you edit the book and save it, the book cover seems to disappear ... strange?*

*Oh, I also made a couple of little improvements here and there. Hope that's okay!*

- *I used a cool third-party framework to detect a nice color palette in the cover art of each book, to use in styling the table view cells and the book edit form. I've also added properties for these colors in the `Book` class. The app seems to freeze, though, for a couple of seconds when you add an image. Is there something you can do about that?*
- *I added a nice little three-page help section to onboard the app, using a page view controller. It automatically triggers when you first open the app, and you can reopen it with a Help button. There should be a title, blurb, and image, but for some weird reason, only the images are displaying.*

*Oh, and you should probably add some tests.*

*Sorry I ran out of time to fix everything up. All the best with it, I look forward to downloading it from the App Store!*

*Oh, here's the repo with my updates: <https://github.com/iOSAppDevelopmentwithSwift-inAction/Bookcase.git> (Chapter15.1.UpdatesNeedFixing).*

Well, that was a nice surprise. Your friend made a couple of nice additions to the app. Great! But it seems the app has been left in a buggy state. That email contains a lot of information; let's go through it step by step, check out what they've done, and explore what needs fixing.

## 15.2 Debugging mode

*The book edit form was working well to begin with, but then it started crashing. Don't know what that's about.*

Let's confirm what your friend is saying about the app crashing.

- 1 Download your friend's repo update.
- 2 As usual, run `carthage update` in the Terminal to update third-party code in the project.
- 3 Run the app. Your friend's onboarding section should appear.
- 4 Select the Skip button.
- 5 Select the + button to add a book.

Bam! Your friend was right—the app crashes!

When Xcode crashes, it automatically enters debugging mode (see figure 15.1). Debugging mode can be intimidating, especially at first. Let's break it down.

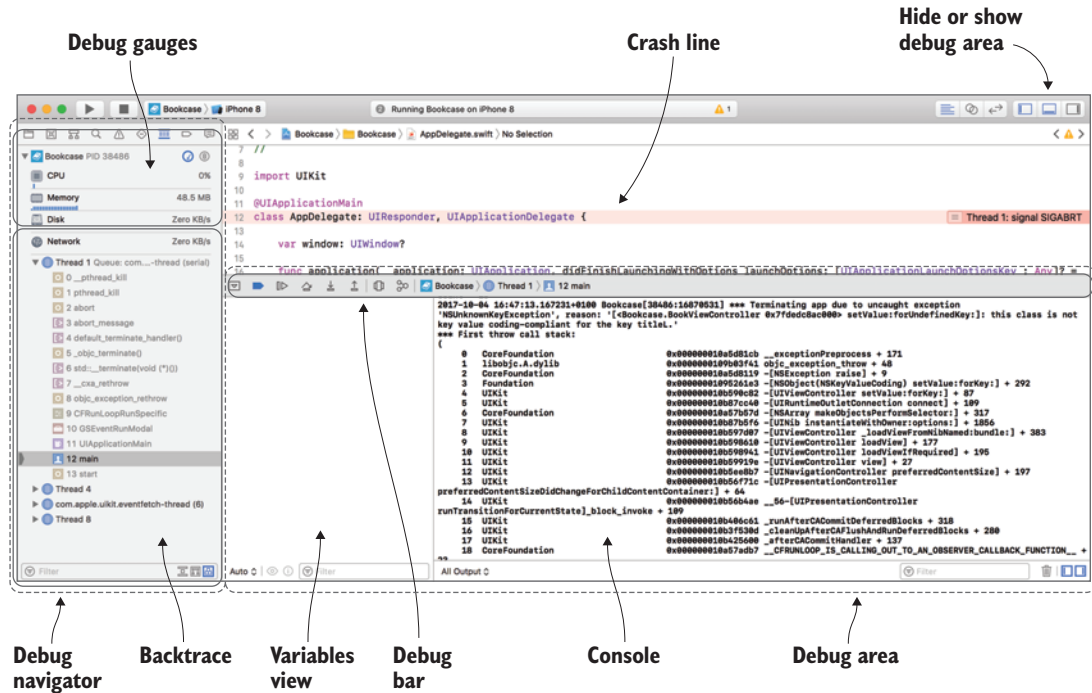


Figure 15.1 Xcode debugger in a crash

Debugging mode consists of

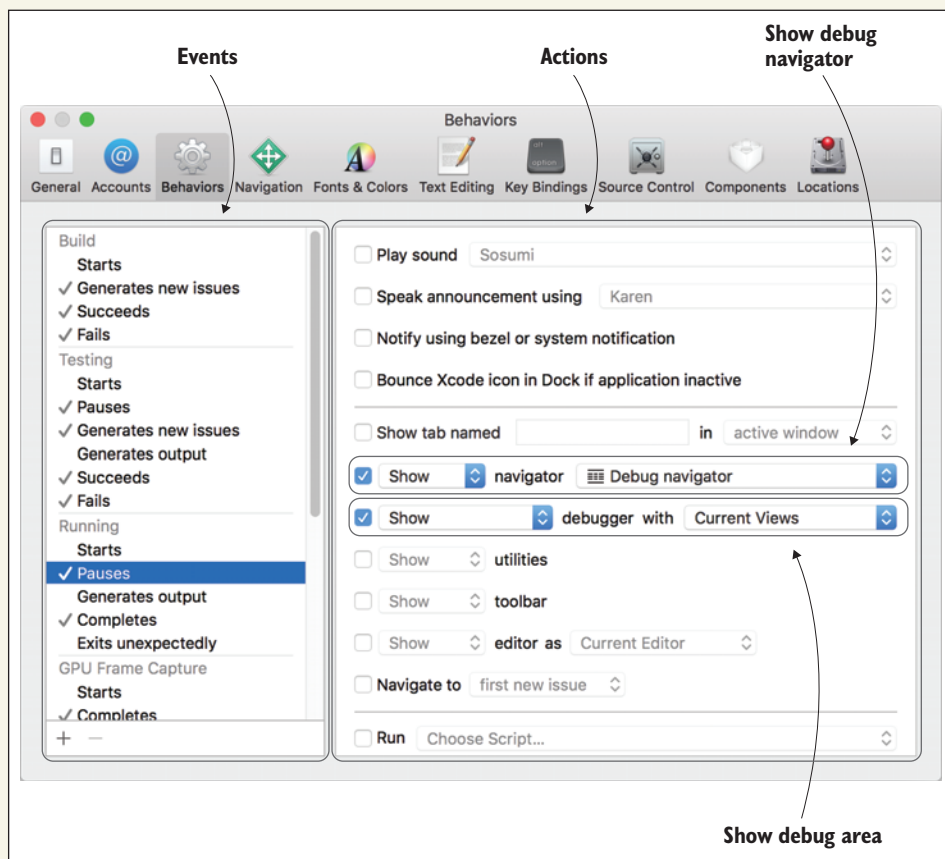
- A red line that appears in the source editor indicating the most recent line of your code that ran before the crash occurred.
- The *debug navigator* appears in the navigator panel, consisting of
  - Gauges for measuring the current state of your device or simulator's CPU, memory, disk, and network activity.
  - A path of how you arrived at the current line of code in each active thread. This is called the *backtrace* (people also call this the *call stack* or *stack trace*).
- The *debug area* appears below the source editor, consisting of
  - The *debug bar* with several debug controls including stepping through your app.
  - The *variables view* showing the current state of variables from the scope of the line in the source editor.
  - The *console*, which outputs the reason for the crash and a printed call stack.

Don't worry, this has only been a short summary of these tools. In a moment, we'll look at each in turn.

### Xcode behaviors

How does Xcode know to automatically show you the debug navigator and the debug area when the app crashes? Well, it's all defined in special Xcode preferences called *behaviors*. Use behaviors to request that Xcode performs specific actions when specific events occur. Xcode comes with certain behaviors already set up for you by default.

Let's look at the default behavior that opens the debug navigator and debug area. Select Xcode > Behaviors > Edit Behaviors. In the events menu on the left, select Running > Pauses. This behavior is triggered when a running app is paused, such as when the app crashes! In the actions menu on the right, you can specify actions to perform when this event occurs. In addition to showing the debug navigator and debug area, you could, for example, play a sound, display a system notification, or even have an announcement spoken to you.



Sometimes, such as in this case, the red line freezes on your AppDelegate class, indicating that the problem probably occurred in initial setup. One common reason for this is a problem with the storyboard. Let's look at the console for clues.

## 15.3 Debugging crash logs in the console

At first glance, the output in the console after a crash looks crazy complicated. To give yourself a shock, take a glance at figure 15.2. But don't panic! You'll see a number of strange symbols, numbers, and unfamiliar syntax. Where to start?

```

2017-10-04 16:54:52.138072+0100 Bookcase[38828:16887511] *** Terminating app due to uncaught exception 'NSUnknownKeyException', reason:
'[[Bookcase.BookViewController 0x7fd4a307e200] setValue:forUndefinedKey:]: this class is not key value coding-compliant for the key titleL.'
*** First throw call stack:
(
  0 CoreFoundation          0x000000010d7ab1cb __exceptionPreprocess + 171
  1 libobjc.A.dylib          0x000000010ccd6f41 objc_exception_throw + 48
  2 CoreFoundation          0x000000010d7ab119 -[NSException raise] + 9
  3 Foundation              0x000000010c6f91e3 -[NSObject(NSKeyValueCoding) setValue:forKey:] + 292
  4 UIKit                   0x000000011054fc82 -[UIViewController setValue:forKey:] + 87
  5 UIKit                   0x000000011083bc40 -[UIRuntimeOutletConnection connect] + 109
  6 CoreFoundation          0x000000010d74e57d -[NSArray makeObjectsPerformSelector:] + 317
  7 UIKit                   0x000000011083a5f6 -[UINib instantiateWithOwner:options:] + 1856
  8 UIKit                   0x0000000110556d07 -[UIViewController loadViewFromNibNamed:bundle:] + 383
  9 UIKit                   0x0000000110557610 -[UIViewController loadView] + 177
  10 UIKit                   0x0000000110557941 -[UIViewController loadViewIfRequired] + 195
  11 UIKit                   0x000000011055819e -[UIViewController view] + 27
  12 UIKit                   0x00000001105ad8b7 -[UINavigationController preferredContentSize] + 197
  13 UIKit                   0x000000011052e71c -[UIPresentationController preferredContentSizeDidChangeForChildContentContainment] + 168
  14 UIKit                   0x000000011052a4ae -[UIPresentationController runTransitionForCurrentState]_block_invoke + 14
  15 UIKit                   0x00000001103c5c61 -[UIViewController runAfterCACommitDeferredBlocks] + 318
  16 UIKit                   0x00000001103b430d -[UIViewController cleanUpAfterCAFlushAndRunDeferredBlocks] + 280
  17 UIKit                   0x00000001103e4600 -[UIViewController _afterCACommitHandler] + 137
  18 CoreFoundation          0x000000010d74ddb7 __CFRunLoop_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ + 23
  19 CoreFoundation          0x000000010d74dd0e __CFRunLoopDoObservers + 430
  20 CoreFoundation          0x000000010d732324 __CFRunLoopRun + 1572
  21 CoreFoundation          0x000000010d731a89 CFRunLoopRunSpecific + 409
  22 GraphicsServices        0x0000000114e029c6 GSEventRunModal + 62
  23 UIKit                   0x00000001103b9d30 UIApplicationMain + 159
  24 Bookcase                 0x000000010c2d1557 main + 55
  25 libdyld.dylib            0x00000001102b0d81 start + 1
  26 ???                     0x0000000000000001 0x0 + 1
)
libc++abi.dylib: terminating with uncaught exception of type NSException
(11db)

```

Figure 15.2 Crash log in console

The trick in interpreting this output is learning what you can ignore 90% of the time and where to find the most relevant information.

The text that automatically outputs to the console when your app crashes is made of two main parts that answer two important questions:

- *Exception information*—What caused the problem?
- *Call stack*—What was happening at the time?

I've organized the console output in figure 15.3. I separated the two main parts and emphasized part of the output to help you focus on what's most important.

First, what caused the problem? The *exception information* should answer this important question, and ironically, it's often scrolled offscreen by the call stack! Ignore the time codes and memory addresses and look for the description of the exception in English. According to the exception information in this case, there was an `NSUnknownKeyException` for the key `titleL` in the `BookViewController`.



Figure 15.3 Crash log in console

Great—the English description of the exception information is often all you’ll need to look at after a crash, but sometimes it helps to also look at what was happening at the time of the crash. The call stack is a path of method calls called *frames* that lead to a certain location in the code. You can use the call stack to trace the path backward from the most recent frame marked with a 0 at the top, down to the least recent frame at the bottom.

To identify each frame in the call stack, each line gives you the framework, origin (usually object and method), line number, and even the memory address of each call. See figure 15.4 for a close-up of frame 5.

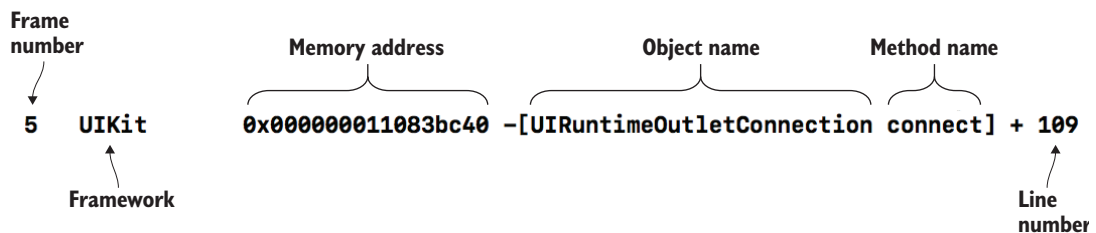


Figure 15.4 Frame in call stack

Calls originating from your own code will have your project name at the left. Note in the call stack that only one call originates from your project, indicated by the project name Bookcase. Look for *main* at line 29 of figure 15.3.

**NOTE** The *main* call is a special one—*main* represents the main entry point for your app, which in your project (and most others as well) is the AppDelegate class. If you take a close look at the AppDelegate class in your project, you'll notice that it's preceded by the keyword `@UIApplicationMain`. This keyword defines the AppDelegate as your app's entry point. You'll find this in the call stack too, at line 28.

Sometimes the call stack can give you a peek behind the curtain of certain classes in the iOS SDK that aren't available to developers. If you look through the objects and method calls in the call stack, you might get an idea as to what was happening when the unknown key exception occurred. Perhaps the connect call to the `UIRuntimeOutletConnection` object at line 5 could be a clue. Although you don't have documentation for this object, you could make a reasonable guess by its name that this object is involved in connecting outlets, and perhaps this has something to do with your crash. The plot thickens!

### 15.3.1 Solving a crash caused by an outlet

Let's revise your clues. You know that an outlet problem likely exists in `BookViewController` related to the key `titleLabel`. Let's look at the storyboard and try to dig deeper.

- 1 Open the storyboard, and select the book edit form scene.
- 2 Open the Connections Inspector to explore problems with outlets. As expected, it appears there's a problem with the `titleLabel` property—the Connections Inspector shows it with an exclamation mark within a yellow triangle, indicating a broken connection (see figure 15.5).

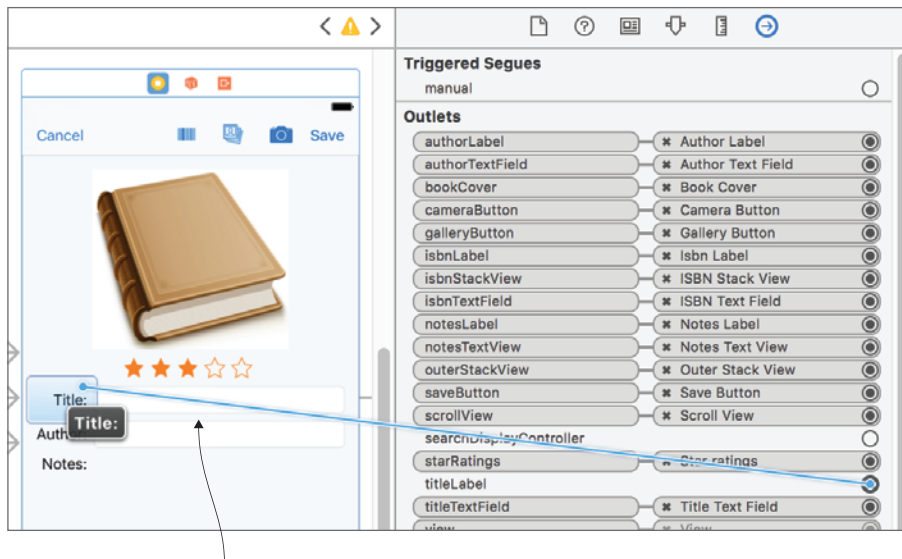


Figure 15.5 IOutlet issues

Below the broken outlet connection is another outlet called `titleLabel` with a hollow circle, indicating that a property in the `BookViewController` class called `titleLabel` has been defined with the `@IBOutlet` keyword, but hasn't been connected to a view in the storyboard.

It appears that your friend set up an outlet called `titleLabel` and then decided to give it the name `titleLabel`, probably to ensure good naming practices. They renamed it in the code, but didn't update the connections! Let's fix it and see if that resolves the crash.

- 3 Remove the old connection by selecting the X next to `TitleL`.
- 4 Now, set up a new connection to `titleLabel` in the Connections Inspector. You could do this in the Assistant Editor as you've done previously, but since you're already in the Connections Inspector, drag from the circle beside `titleLabel` to the title label in the storyboard (see figure 15.6).



**Drag from Connections Inspector to appropriate element in storyboard**

**Figure 15.6** Connect IBOutlet into the Connections Inspector.

You should see the title label with a filled circle in the Connections Inspector, indicating that it's now connected to a view in the storyboard. If you open the `BookViewController` class, you'll see the same filled circle indicator there as well (see figure 15.7).



**Figure 15.7** IBOutlet connected in the source editor



Now, all that's left is to run the app and see if you've solved the problem!

- 5 Run the app, select or add a book, and ... no crash!

First problem solved, what's next?

### 15.3.2 Solving a crash caused by an action

*The Cancel button in the book edit form crashes the app.*

With the app running and the book form open, select the Cancel button. Your friend was right!

Another long crash log fills the console, but this time you have a better idea of what to look for. Let's start with what caused the problem. With memory addresses removed, the exception information reads thus:

```
- [Bookcase.BookViewController touchCancel:]: unrecognized selector sent to
  instance
```

It appears that in the BookViewController class, a selector (that is, a method) called touchCancel is being called but not recognized. Why would that be, and what was happening at the time? You probably have enough information to take a good, educated guess, but let's look at a portion of the call stack for more clues. See figure 15.8—again, I've emphasized part of the output to help you focus on more-interesting details.

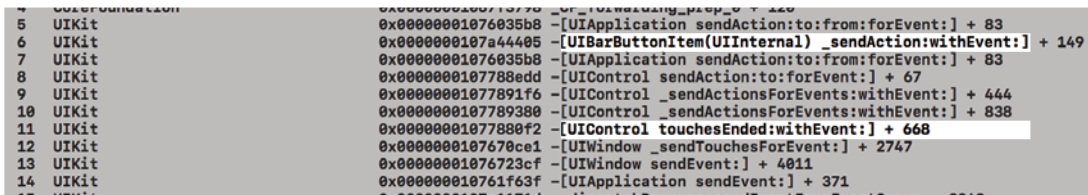


Figure 15.8 Crash log in the console

Note that sending an action for an event triggered by a UIControl seems to be a theme. The event itself seems to be a touch, according to frame 11, and the control seems to be a UIBarButtonItem.

Let's revise all of our clues again. When a bar button item in the scene connected to the BookViewController class (assumedly the Cancel button) tries to call the touchCancel method, it's not recognized. Let's look at the storyboard to get a clearer idea of the problem.

- 1 Open the storyboard, select the book edit form scene, and open the Connections Inspector to explore problems with actions. Similar to earlier, there seems to be a problem with the touchCancel method (see figure 15.9).



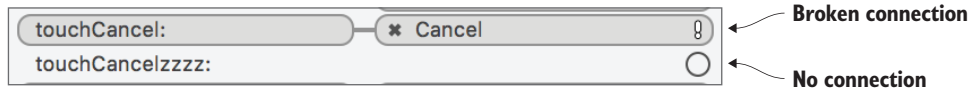


Figure 15.9 IBAction issues

There seems to be a broken connection between the Cancel button and the touchCancel action method. Curiously, there seems to be an unconnected action method called touchCancelzzzz!

- 2 Open the BookViewController class and see what's going on in the code (see figure 15.10).

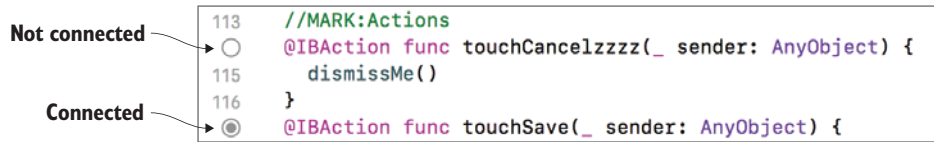


Figure 15.10 IBActions in the source editor

It's true! There's a touchCancelzzzz method in the BookViewController, and there isn't a touchCancel method to be seen. Your "helpful" friend must have leaned on the keyboard and inadvertently renamed the method. As the hollow circle indicates, this caused the touchCancelzzzz method to disconnect from the storyboard.

- 3 Remove the extra z's from the method name and rebuild the project. The circle should fill in, indicating that all is well in the world again, and the Cancel button in the storyboard is reconnected with the touchCancel action in your BookViewController class.
- 4 To be sure, rerun the app, open a book, and select Cancel.

This time, the app should act as expected, closing the book edit form scene. What's next, detective?

## 15.4 Examining variables and breakpoints

*After you add an image and save it, the next time you edit the book and save it, the book cover seems to disappear ... strange?*

First, check that you can replicate the problem.

Run the app, open a book with a cover image (you'll have to add a cover image for a book first if none of your sample books have cover art), and select Save. The book image returns to the default cover image. "Strange" is right! What could be happening?

Your immediate suspicion is that for some reason, an existing book cover isn't being used when the `BookViewController` generates a book to save. Let's confirm that by examining the `bookToSave` variable in the `BookViewController` class in the `touchSave` method.

As is so often the case in Xcode, there are many different ways to examine the contents of a variable. Let's look at a few now, beginning with a method that you've seen before, the `print` method.

### 15.4.1 Examining a variable with `print`

To examine the `bookToSave` variable, let's print its contents to the console with the `print` method.

- 1 Before the `touchSave` method calls `dismissMe`, print the `bookToSave` variable.

```
print("Saving book: \(bookToSave)")
```

- 2 Run the app again, once again open a book with a cover image, and select `Save`. This time, the book object should print to the console, looking something like this:

```
Saving book: Book(title: "Five on Brexit Island", author: "Enid
Blyton", rating: 3.0, isbn: " 9781786488077", notes: "", image:
Optional(<UIImage: 0x1c02aeb20>, {128, 202})), backgroundColor:
UIExtendedGrayColorSpace 1 1, primaryColor: UIExtendedGrayColorSpace 0
1, detailColor: UIExtendedGrayColorSpace 0 1)
```

Well, that's great. By default you're seeing the value of every property of the object, down to its background color. Sometimes, however, when you print an object, you might not need to see its every last detail. You might prefer to see just the important stuff. It would probably be sufficient detail to identify a book, for example, by the title and author. To resolve this bug, you might also want to see whether or not this book has a cover image.

There's a neat little trick for adjusting the string that's output when you print an object. If your custom type adopts the `CustomStringConvertible` protocol, you can provide a `description` property that describes your object as a `String`, and it will automatically be used by `print`.

- 3 Add a `description` property to the `Book` class that returns the title, author, and a message about whether the book has a cover image.

```
override var description: String {
    return "\(title) by \(author) :
        ➡ \(hasCoverImage ? "Has" : "No") cover image"
}
```

- 4 Run the app again, and save a book with a cover image. This time, you should see more meaningful information about the book being saved in the console:

```
Saving book: Five on Brexit Island by Enid Blyton : No cover image
```

It appears that your suspicion was correct. For some unknown reason, the book object to be saved isn't being generated with its cover image.

**TIP** Classes that subclass `NSObject`, such as `UIView`, automatically adopt the `CustomStringConvertible` protocol and contain a `description` property. To provide your own description, you'll have to override the default `description` property.

Sometimes, adding `print` statements everywhere in your code to help diagnose a problem can get out of hand, and more-sophisticated debugging techniques would be more appropriate.

**TIP** An alternative approach to `print` that certain developers prefer is the `NSLog` statement. While `NSLog` is a little slower, it does add a timestamp to the log and stores logging data to disk. Having a log history can be useful, but makes it even more important to ensure you remove all `NSLog` calls from your code before publishing your app to the App Store.

Remove the `print` statement now. We're going to explore other debugging techniques to diagnose the source of this problem further.

#### 15.4.2 *Pausing your app with a breakpoint*

To diagnose problems in your app, sometimes it can help to use a file and line breakpoint to pause execution at a line in your code. File and line breakpoints are ultra-useful for

- *Checking the current state of the app.* This is useful for taking a closer look at variables, the call stack, threads, the user interface (UI), or the app's use of system resources at a specific point in time.
- *Stepping through your app.* You can use the step controls to run your app step by step and diagnose any problems with the flow of your app.

You'll use file and line breakpoints to analyze why books aren't being saved with their images. Let's start by looking at right after a book object is generated for saving data from the book edit form.

- 1 Add a breakpoint to your code after setting the `bookToSave` variable in the `touchSave` method in `BookViewController`. Adding a breakpoint is simple; click to the left of the line where you want execution to be paused. A dark blue pointed rectangle should appear where you clicked, indicating an active breakpoint (see figure 15.11).

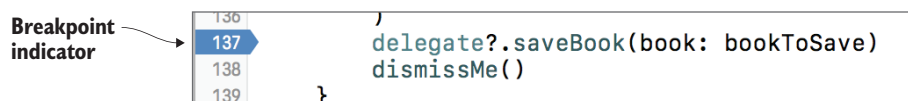


Figure 15.11 File and line breakpoint

**NOTE** Be careful not to click on the breakpoint again; this will cause the indicator to turn light blue and the breakpoint will toggle to a disabled state.

Another place that could be interesting to analyze is when a view is loaded and the `BookViewController` class receives a `Book` object to edit.

- 2 Using the same technique, add a second breakpoint to the `viewDidLoad` method of `BookViewController` after unwrapping the book object.
- 3 Run your app again, and this time tap on a book that does *not* have a cover image. The app should pause immediately at the breakpoint you specified in the `viewDidLoad` method.

The same way it did earlier when the app crashed, the Running > Pauses behavior launches into action, automatically opening the debug navigator and debug area for you. One difference you may notice is that the paused line of execution is green this time (see figure 15.12).



Figure 15.12 Breakpoint pausing execution

### Advanced breakpoints

Most commonly, you'll use breakpoints to pause execution at a specific line of code, but they're capable of doing so much more.

For example, *exception breakpoints* break execution whenever specific types of exceptions occur, and *symbolic breakpoints* break execution whenever a specific *method* is called on all subclasses of a certain type of class. You have to add these types of breakpoints in the *breakpoint navigator*.

Your breakpoint could be set up to trigger only if a certain condition is true or after a certain number of times. Breakpoints can also be set up to perform one or more *actions*, such as output to the console or play a sound. Ironically, breakpoints don't necessarily *break* execution. If you like, after performing an action, a breakpoint can automatically continue.

Edit your breakpoints by double-clicking on the breakpoint indicator in the source editor or the breakpoint navigator.

Now that your app has paused execution, you can examine the state of the app's variables. Checking the book object at this point may help diagnose the problem with saving a book cover.

You can use several approaches for examining the state of variables while the app is paused:

- The variables view
- Quick Look
- Print description
- Command line in the lower-level debugger
- Datatips

We'll look at each of these in turn. Let's look first at the variables view.

### 15.4.3 Examining a variable with the variables view

The *variables view* contains variables in the context of where the app is currently paused. Instance variables of `BookViewController` will be contained within the `self` property, while local variables are shown at the top level. As the book object is unwrapped with optional binding, it's considered a local variable.

At the left of several variables, you'll see a *disclosure triangle*, indicating that you can “open up” the variable to have a closer look at its contents.

- 1 Click on the disclosure triangle for the book object to inspect the value of its properties (see figure 15.13).

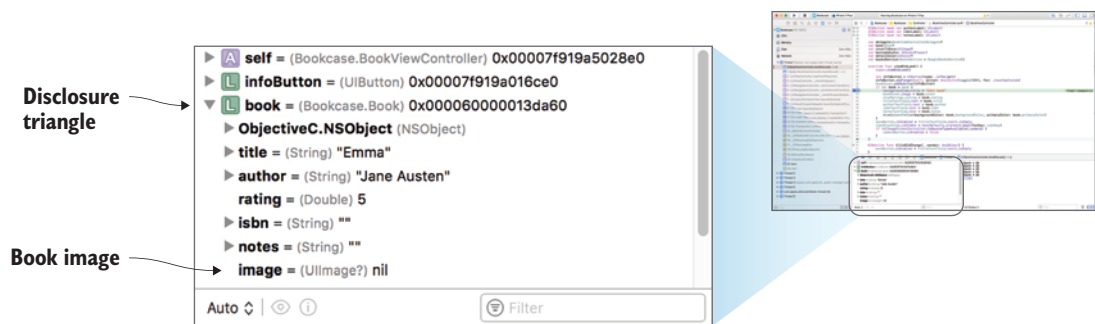


Figure 15.13 Variables view

- 2 Note that the book image is nil.

This makes sense, as you selected a book with no cover.

Now, let's resume execution so that you can add an image to this book.

### 15.4.4 Controlling the app's execution using the debug bar

Above the variables view, you'll find the debug bar, which contains several controls useful for controlling the execution of your app (see figure 15.14).

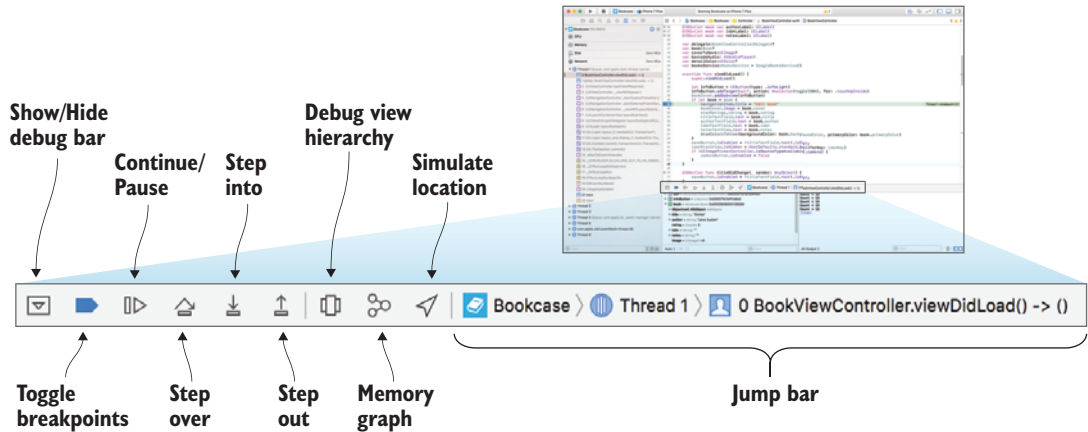


Figure 15.14 Debug bar

Table 15.1 lists several elements that could use extra explanation.

Table 15.1 Debug bar elements

Element	Description
Toggle breakpoints	For convenience, toggle all breakpoints on or off.
Continue/Pause	Continue execution of the app.
Step buttons	Three skip buttons allow you to execute your code step by step. <i>Step over</i> and <i>step into</i> differ as to how they act when there's a method call in the current line. <i>Step into</i> will step through every line of the method, whereas <i>step over</i> will interpret the entire method as one step. <i>Step out</i> , on the other hand, executes the rest of the current function as one step and pauses execution again when it exits the function.
Debug view hierarchy	View the hierarchy of views in the app. We'll come back to this soon.
Memory graph	Visualize the memory allocations in the app.
Simulate location	Simulate that your app is running from an alternative location.
Jump bar	Use the jump bar to examine your app state from the context of different threads and stack frames.

Let's use the controls in the debug bar to resume execution of the app.

- 1 Tap the Continue button.
- 2 Add a cover image to the app.
- 3 Save the book with the new image by tapping the Save button.

The app should pause execution again after generating a new book to save in the local `bookToSave` variable. Let's examine this variable for more clues.

### 15.4.5 Examining a variable with Quick Look

Let's explore examining variables using another technique, called *Quick Look*.

- 1 First, focus once again on the variables view, and select the disclosure triangle beside the `bookToSave` variable to open it up.
- 2 Note that this time, the book image shows a memory address. You can reasonably assume that this means that your book contains an image, but how can you know which?

Certain variables are visual in nature, and the variables view may not be sufficient to describe a variable. *Quick Look* provides you with a *visualization* of the contents of a variable. (You may remember Quick Look from playgrounds, way back in chapter 2.)

- 3 Select the image property of `bookToSave`.
- 4 To open a visualization of the image property and select the button that looks like an eye, located below the variables view (see figure 15.15).

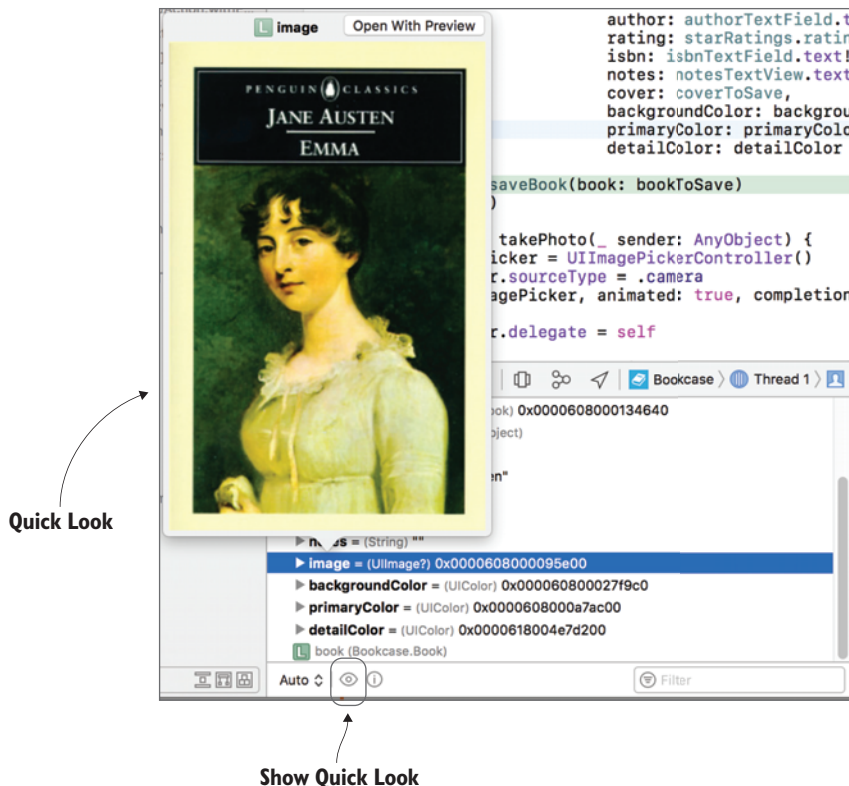


Figure 15.15  
Quick Look

Well, that seems to have worked correctly. The image you added to the book edit form is stored in the image you're saving. But the problem was presenting itself in books that already have an image. You'll need to go through this process again, with the same book now that you know it contains an image, and find the source of this problem.

- 5 Tap the Continue button, which should return you to the main screen.
- 6 Choose the same book you added a cover image to.

The app should pause once again at the breakpoint in the `viewDidLoad` method after unwrapping the book object to edit.

Let's use yet another technique for examining the contents of the book object.

#### 15.4.6 Examining a variable with print description

Next to the *Show Quick Look* button, is another useful button that appears as an "i" in a circle. This is called the *Print Description* button. If you select a variable in the variables view, and select the Print Description button, you get exactly the same output in the console as you did earlier when you printed a variable in code.

This time, you'll examine the contents of the book object with the Print Description button.

- 1 Select the book object in the variables view.
- 2 Select the Print Description button.

The description property of the `Book` object that you set up earlier will output to the console (see figure 15.16). Covering all bases, the properties of the `Book` object also output to the console.

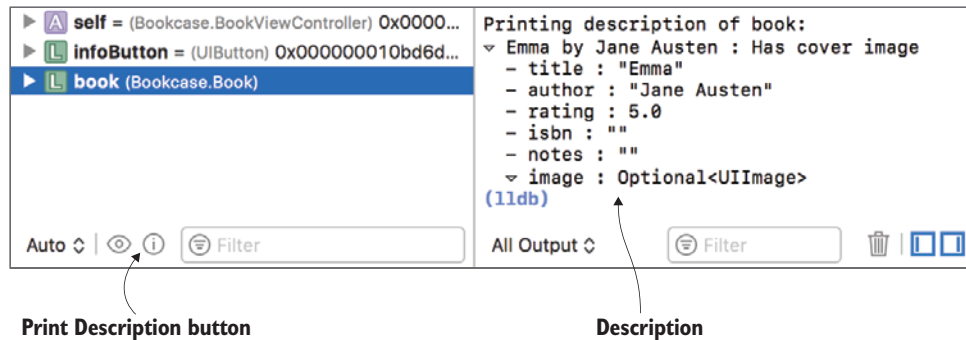


Figure 15.16 Print variable description

Well, according to the output, it seems no problem exists with the book object. You'll have to continue execution and save the book to see if the problem is happening there.

But first, what's that strange `11db` message that crops up in the console?



### 15.4.7 Examining a variable with LLDB

The console is much more than an area for receiving debug logs and outputting print messages. It's a window into the powerful command-line debugger called *lower-level debugger* (LLDB), and the `lldb` message is a prompt for you to enter commands.

Many debugging features in this chapter are GUI representations of lower-level commands that are available to you as command-line commands in the console.

For example, the Print Description button you used to explore details on the book object uses the LLDB `po` command under the hood.

- 1 Use the `po` command to examine the book variable. Type the following after the `lldb` prompt and press the Return key:

```
po book
```

You should see the same description appear for Book that you saw for Print Description (see figure 15.17).

If you want to go beyond the default description of a variable and print the underlying implementation of an object, use the `p` command.

- 2 Use the `p` command on the book variable.

```
p book
```

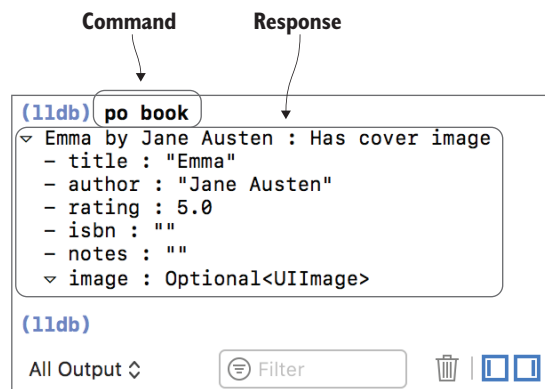


Figure 15.17 LLDB command `po` in the console

See figure 15.18 for the result from the `p` command. This time, you should see a much more detailed output of the contents of the book variable.

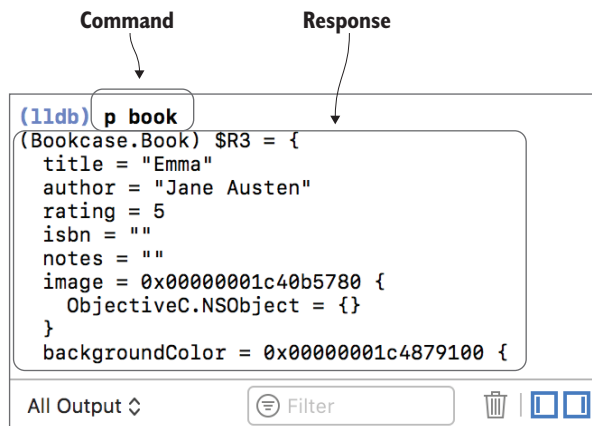


Figure 15.18 LLDB command `p` in the console

We've barely scratched the surface of what's possible with LLDB. Apart from online documentation, you can use LLDB's help command to get a comprehensive listing of debugger commands.

For a change, let's use LLDB to resume program execution.

- 3 Type `c` after the `lldb` prompt, and press Return. The program should continue.
- 4 Tap Save, to test saving this book.

Once again, the app should pause execution right after generating a book to save. Let's use one final technique to examine the contents of the book to save.

### 15.4.8 Examining a variable with data tips

Believe it or not, there's yet another way to examine the contents of your variable, and this time, you don't even need the variables view or the console!

With app execution paused, you can point your cursor in the source editor at a variable you want to examine, and a *data tip* for that variable will pop up. From there, you can open the variable the way you did in the variables view, select to show Quick Look, or select the Print Description button.

- 1 Point to the `bookToSave` variable now. A data tip for the variable should appear.
- 2 Select the disclosure triangle, open the variable, and examine its contents (see figure 15.19).

Notice that this time, the `image` property of `bookToSave` is equal to `nil`. You seem to be getting closer to the problem!

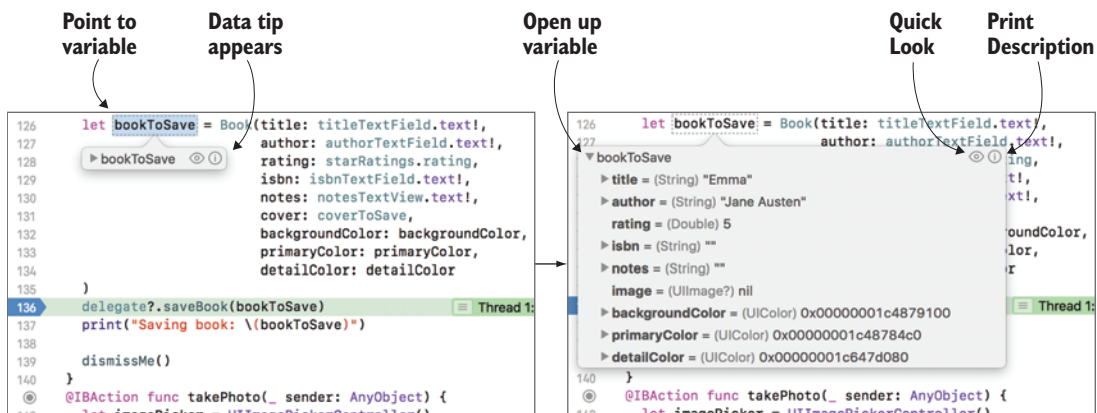


Figure 15.19 Examine a variable with data tips

### 15.4.9 Solving the save problem

Why would the `image` property be `nil`? Look at how the `bookToSave` object is generated—the cover image comes from the `coverToSave` property. Okay, where’s this property set?

A quick search for the `coverToSave` property uncovers the problem. The `coverToSave` property is only set in two places: when the user selects a photo or image for the book or when the `booksService` returns an image after the user scans a barcode. What about books that already have an image? The `coverToSave` property is never set.

- 1 In the `viewDidLoad` method, after unwrapping the book object, set the `coverToSave` property. Check first that the book *has* a cover image, to avoid setting the default cover to the `coverToSave` property.

```
if let book = book {
    navigationItem.title = "Edit book"
    bookCover.image = book.cover
    if book.hasCoverImage {
        coverToSave = book.cover
    }
    ...
}
```

- 2 Run the app again, select a book with a cover image, and save it. This time (fingers crossed!) the book cover image should stick around. Hooray! Good job, detective—problem solved. You can remove your two breakpoints now.
- 3 To remove the breakpoints, click on them, and drag them to the right. They should disappear—in a puff of smoke!

### 15.4.10 Examining a variable in summary

Many methods exist for examining the contents of a variable, each with their own advantages, as shown in table 15.2.

**Table 15.2** Examining a variable

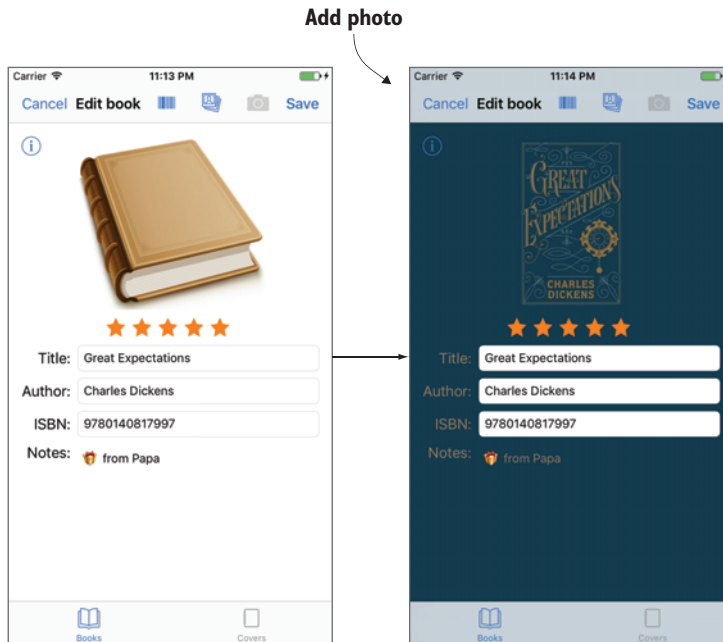
Element	Best for
<code>print</code>	If you prefer to not pause execution of your app
<code>NSLog</code>	If you want timestamps on your console logs and a log history
Quick Look	If you want a visualization of the variable’s contents
Data tips	If you’re short on screen space and prefer to hide the debug area, or if you prefer to explore variables in the context of your source code
<code>p</code> command in LLDB	If you need information beyond what the default description returns for the variable
Variables view	If you want a visual representation of the hierarchy of variables in your app

## 15.5 Debugging playback with gauges and instruments

Let's check out your friend's next piece of feedback.

*I used a cool third-party framework to detect a nice color palette in the cover art of each book, to use in styling the table view cells and the book edit form. I've also added properties for these colors in the `Book` class. The app seems to freeze, though, for a couple of seconds when you add an image. Is there something you can do about that?*

Sounds like quite an interesting addition to the app that your friend has contributed; see figure 15.20 to see it in action.



**Figure 15.20**  
Color detection  
of the book image

The freezing interface isn't so useful, though!

If your app is having playback problems such as a stuttering or freezing interface, the cause may be that you're performing long operations in the main thread and therefore blocking your interface from updating.

Let's explore this theory with the debug gauges.

### 15.5.1 Debugging playback with debug gauges

If your app is experiencing performance issues, it can be a good idea to look at your app's use of system resources. One way to do this is with the *debug gauges* that you can find in the debug navigator. The debug gauges give you a good summary of how your app is using the device's CPU, memory, disk access, and network calls. You can click on a gauge to get a more detailed report on your app's use of this system resource.

You're going to examine your app's use of the CPU when adding an image to diagnose why the user interface is freezing temporarily.

- 1 Run your app, and select the Debug Navigator.
- 2 Select the CPU gauge from the debug gauges, to display the CPU report.
- 3 Select a book, and add an image. You should see something like figure 15.21.

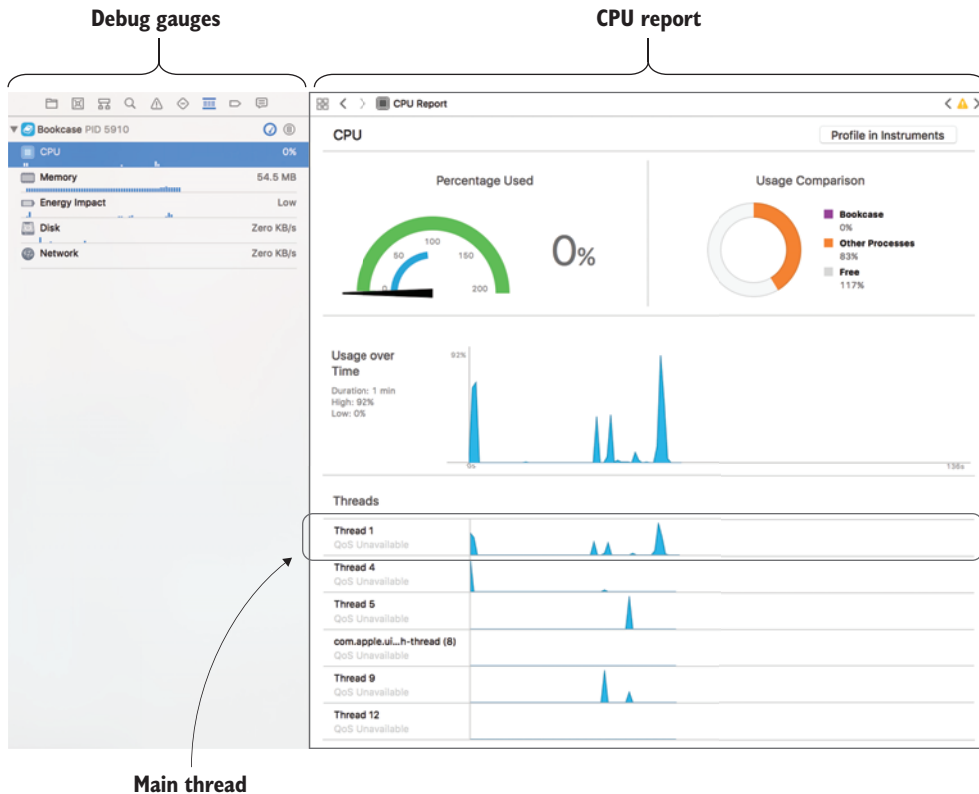


Figure 15.21 Debug gauges and CPU report

Note that the majority of the work is going on in thread 1. Thread 1 is also known as the main thread and is where the user interface is updated. As you've seen, if your app is busy working on a time-consuming algorithm such as image color detection in the main thread, the app's user interface will be prevented from updating and responding to user interaction.

It has become clear that a certain operation that your friend introduced needs to be moved to a background thread. But which operation? You could spend time hunting down this method in the code, but you have yet another debugging trick up your sleeve!

### 15.5.2 Debugging playback with instruments

Xcode provides developers with a library containing debugging tools called *instruments* that build on and supplement the performance and testing tools that are available in debug gauges.

To get a feel for instruments, we'll have a look at the *time profiler* instrument. The time profiler measures how frequently your app performs different processes. You could use the time profiler to find any long-running processes that could be holding up the main thread.

Although you could open the time profiler up by selecting Xcode > Open Developer Tool > Instruments > Time Profiler, you have a shortcut right in front of you in the CPU debug report—at the top-right corner is a *Profile in Instruments* button.

- 1 Select the Profile in Instruments button. Xcode will offer to transfer or restart the debug session.
- 2 Select Transfer.
- 3 The time profiler opens and automatically begins recording the time spent on various processes in your app.
- 4 Back in the simulator, add an image to a book again.
- 5 Once the image has been added to the book, you can select the Stop button in the time profiler. The processes that you want to debug have been profiled, and now you can explore the time profiler (see figure 15.22).

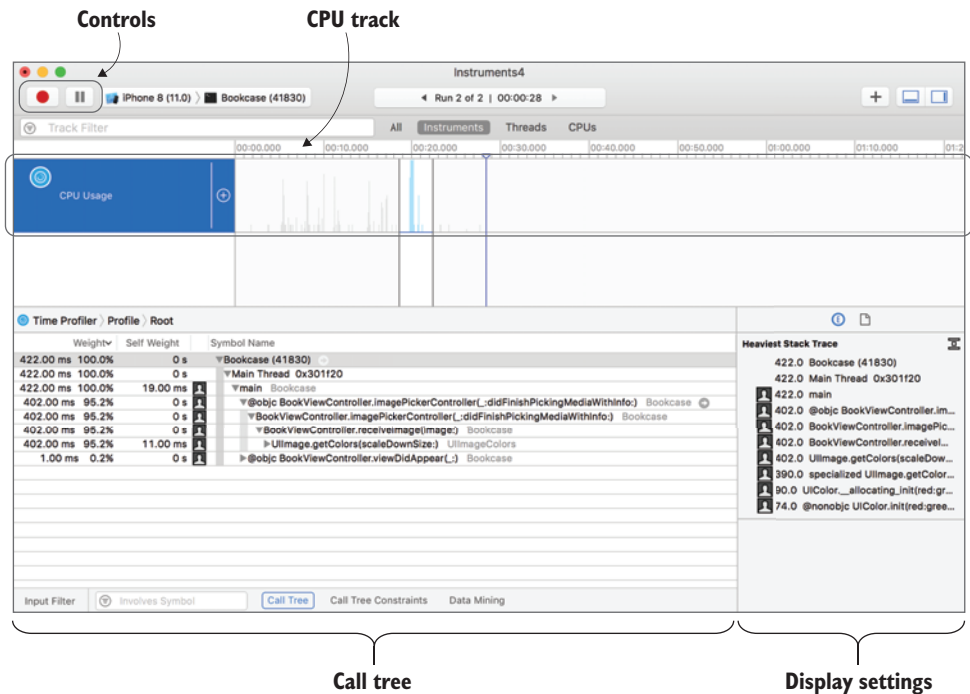


Figure 15.22 Time profiler

While you were recording your app, the time profiler sampled CPU percentage usage (indicated in the CPU track) and call stacks (detailed in the call tree) at regular intervals. Each call in the call tree indicates what's called a *weight*, which is an approximation of the amount of time spent in this process.

- 6 Because you're interested in finding problems in your own code rather than Apple's, select the Call Tree menu in the bar along the bottom, and check Hide System Libraries.
- 7 Now, your detective work involves digging down through the call tree hierarchy, following the process with the greatest amount of sample time. You should find a clear path in the main thread down to the `receiveImage` method in the `BookViewController` class, which in turn calls the `UIImage` object's `getColors` method.
- 8 Double-click the line that reads `BookViewController.receiveImage`. This will show you the problem line of code, indicating the number of samples recorded containing this process (see figure 15.23).



Figure 15.23 Time profiler

If there was any question which line of code was taking up processing time, it seems to be resolved now! This line definitely needs to be moved to a background thread.

- 9 Select the Open in Xcode button at the top right of the time profiler. This should take you straight to the problem line of code, ready for you to solve the problem.

### 15.5.3 Solving the playback problem

Now that you know for sure what was causing the app to freeze, let's move it to a background thread.

- 1 Move the `getColors` call to a background thread using Grand Central Dispatch.
- 2 Move the `receiveColors` call to the main thread, so that it can update the user interface.

```
DispatchQueue.global().async {
    let colors = image.getColors()
    DispatchQueue.main.async {
        self.receiveColors(colors:colors)
    }
}
```

- 3 Run your app again and add an image to a book. You should find that the app no longer freezes while the colors are being detected in the image. You're free to interact with the app, and when the algorithm has finished its work on a background thread, the colors in the interface smoothly animate to the colors detected in the image. Nice!

I think you're ready for your final debugging challenge!

## 15.6 Debugging the user interface

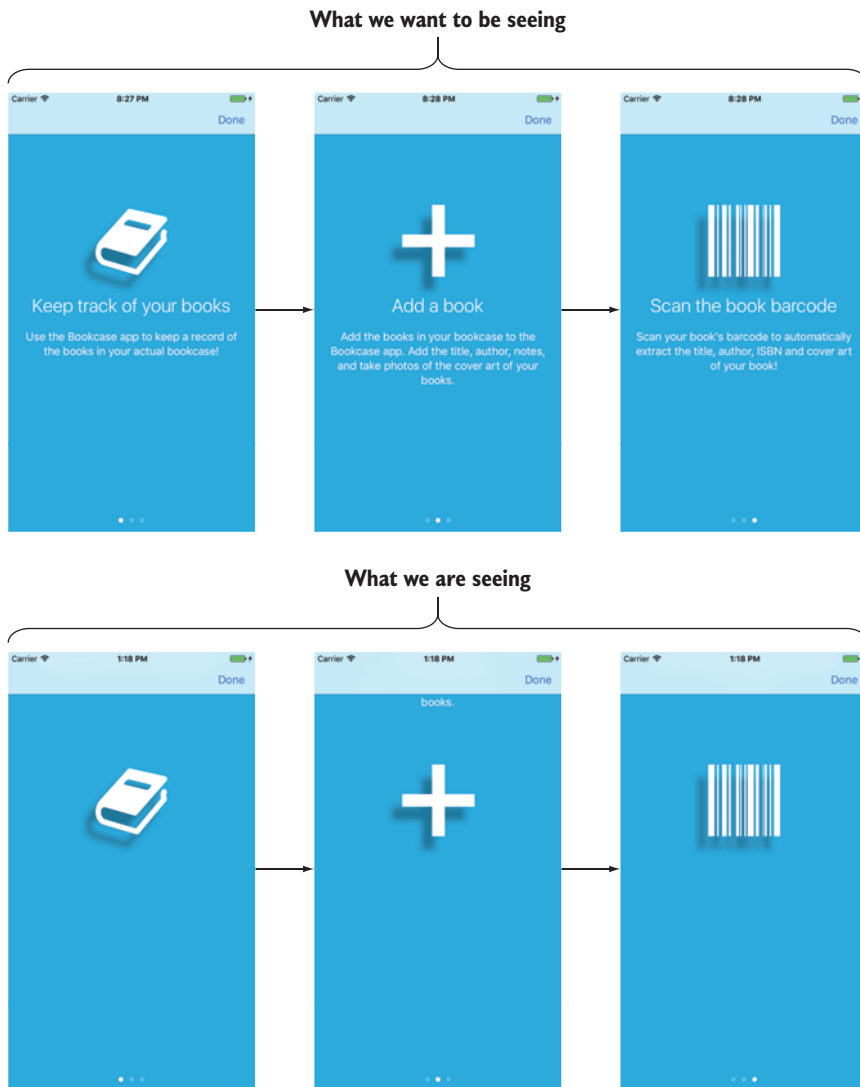
*I added a nice little three-page help section to onboard the app using a page view controller. It automatically triggers when you first open the app, and you can re-open it with a Help button. There should be a title, blurb, and image, but for some weird reason though, only the images are displaying.*

Again, this is a nice improvement that your friend has contributed. However, as mentioned, there's a visual issue—the title and blurb for each page aren't appearing. Your friend sent through an image showing how the help pages *should* look, and how they *do* look (see figure 15.24).

Your friend isn't a fan of the storyboard and has set up the three pages entirely in code. These three view controllers make use of convenience methods in a structure called `InstructionFactory` to perform the repetitive tasks of building their interface. They then use a convenience method in another structure called `ContentLayoutMachine` that automatically sets up their auto layout constraints.

It's all sophisticated, but what's going wrong—where's the title and blurb?





**Figure 15.24** Help page view controller

### Onboarding and page view controllers

It's a good idea to walk your users through how to use your app. This sort of introduction is called *onboarding* your users. Frequently, onboarding requires multiple pages, and the most common approach for displaying these pages is with a page view controller. Rather than the default page turn, it's more common to use a scroll transition style and a page control at the bottom of the screen, indicating the page you're currently viewing.

Pages are represented by view controllers, and the next and previous pages are loaded, ready for the user to scroll to them.

Your friend has been kind enough to set up such a page view controller for you in the Bookcase project, but for future reference, these are the general steps you'd take:

- 1 Add a page view controller to the storyboard that's connected to a custom class that subclasses `UIPageViewController`.
- 2 In the `viewDidLoad` method, set the initial view controller to display with the `setViewControllers` method.
- 3 Adopt the `UIPageViewControllerDataSource` protocol, set the data source, and implement data source methods that return the next and previous view controllers.
- 4 Also implement data source methods that return the number of pages, and the number of the initial page.

#### 15.6.1 Debugging the user interface with the Debug View Hierarchy

When there's a visual problem with your app, a good place to look for answers is the *Debug View Hierarchy*. The Debug View Hierarchy helps you visualize your app's interface and interact with it by separating the layers of the interface and rotating them in 3D space.

You'll use the Debug View Hierarchy to see if you can get a better idea of what's going on in the interface of the help pages.

- 1 Run the app, and select the Help button.
- 2 Back in Xcode, select the Debug View Hierarchy button in the debug bar (see figure 15.25).



Figure 15.25 Debug View Hierarchy button in the debug bar

The app will automatically pause. A rendering of the views in your app will appear in the editor window with controls below it for adjusting the view. A hierarchy of views will appear on the left in the Debug Navigator. The object and size inspectors become available in the inspector panel, with additional information on currently selected views (see figure 15.26).

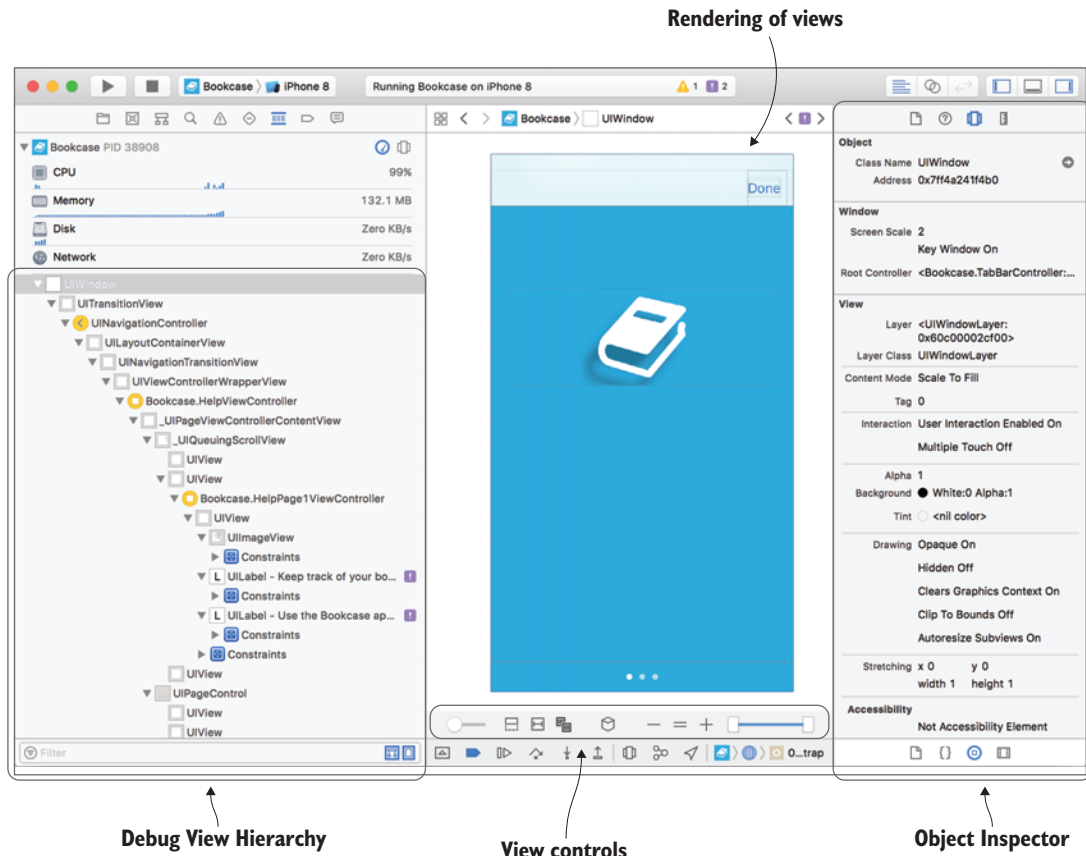
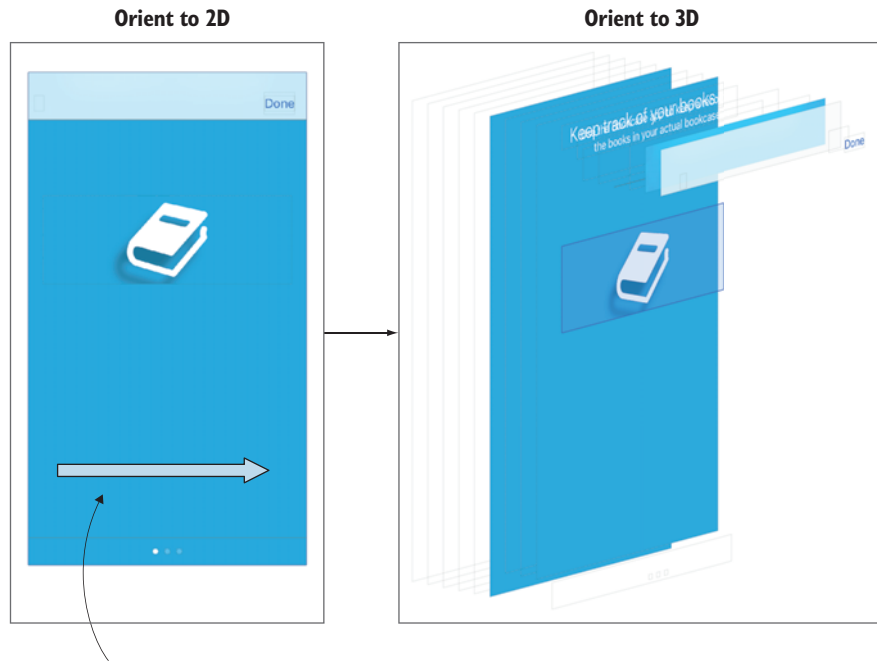


Figure 15.26 Debug View Hierarchy

This is where it gets interesting!

- 3 Click on the rendering of views and drag to the right. The layers will separate and rotate in 3D orientation, giving you a clearer perspective on what's happening in the scene (see figure 15.27).



Click-drag 2D view to right

Figure 15.27 Debug view oriented to 3D

That's interesting! Two text labels are hiding behind the navigation bar. They must be the title and blurb that you're looking for! But what could be causing the layout issue?

- 4 Select one of the labels. If you find it difficult to select, you can use one of the two sliders in the view controls. The slider on the left adjusts the spacing between views, and the slider on the right adjusts the range of visible views.

The label should automatically highlight in the view hierarchy. Notice the purple exclamation mark beside the view. This indicates a runtime issue with this view.

- 5 To get more clues on this issue, open the Issue Navigator.

### 15.6.2 Debugging the user interface with runtime issues

The Issue Navigator gives you more detail on any pending issues. Until now, you've probably only noticed build-time issues, but Xcode can also report *runtime* issues. Ambiguous layouts, problems with threading, and problems with memory allocation can all trigger runtime issues.

Let's examine the runtime issues to further diagnose the problem with your app's layout.

Select the Runtime Issues tab in the Issue Navigator. You should find that several labels have ambiguous vertical positions (see figure 15.28).

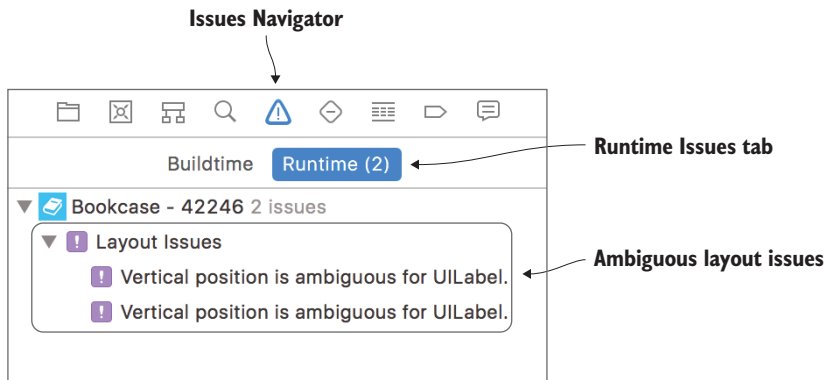


Figure 15.28 Runtime issues

Select one of the issues and open the Size Inspector. Look at the *Constraints* section. In addition to reiterating the layout issue, the existing constraints are specified. The description of the ambiguous layout issue makes sense; there doesn't appear to be a constraint specified for vertical position! See figure 15.29.

Now that you know that certain views aren't being provided with vertical position constraints, you have an idea of the problem to look for in the layout code.



Figure 15.29 Constraints in the Size Inspector

### 15.6.3 Solving the user interface problem

- 1 Open the `ContentLayoutMachine.swift` file where your friend defined the layout for the help pages.

It appears that the `verticalLayout` method your friend wrote loops through all the views in the page, attaching their `topAnchor` to the `bottomAnchor` of the `previousView`:

```
static func verticalLayout(to rootView: UIView, views: [UIView]) {
    ...
    var previousView: UIView?
    ...
    for view in views {
        if let previousView = previousView {
            ...
        }
    }
}
```

Loops through views →

← Declares previousView optional

← Unwraps previousView

```

constraints += [view.topAnchor.constraint(
    equalTo: previousView.bottomAnchor) ]
    }
    ...
  }
  ...
}

```

Attaches top anchor as  
to previousView

Going through the logic, you see a significant problem. The `previousView` is never set, so the constraint is never added!

- 2 Set the `previousView` at the end of the for loop:

```

static func verticalLayout(to rootView:UIView,views:[UIView]) {
    var previousView:UIView?
    for view in views {
        if let previousView = previousView {
            constraints += [view.topAnchor.constraint(
                equalTo: previousView.bottomAnchor) ]
        }
        previousView = view
    }
}

```

Sets previous  
View

Vertical constraints should be added to views now, pinning them to the previous view.

- 3 Run the app to check, and select Help. The help pages should appear as expected, and if you open the debug view hierarchy, you shouldn't find any run-time issues. Hooray!



**CHECKPOINT** If you'd like to compare your project with mine at this point, you can check mine out at <https://github.com/iOSApp-DevelopmentwithSwiftinAction/Bookcase.git> (Chapter15.2.Debugged). Don't forget to run `carthage update` to update third-party code.

Well, you solved all the bugs your friend reported in their email, detective. Congratulations! But what was that your friend said about testing?

## 15.7 Testing your app

It's so easy to make changes to your app to make a minor fix or improvement, only to realize later that you've inadvertently caused a major problem elsewhere in your app. Solving one problem can create another, or, like your friend earlier in this chapter, even resting your hand on the Z key for a second could cause it to crash!

Testing your app manually but comprehensively after every small change would be a tedious prospect. Xcode provides you with the tools for automating this testing process.

Xcode can perform two types of tests:

- *Unit tests* test that your code is doing what it's intended to do.

- *UI tests* test that your app is doing as expected from the perspective of the user interface.

Within both categories, Xcode can focus from two perspectives:

- *Functional*—Is it working correctly? For example, in a calculator app, does  $2+2=4$ ?
- *Performance*—Is its performance acceptable compared against a benchmark time? For example, in a calculator app, is a complex calculation taking a reasonable time to process?

Let's add tests to the Bookcase app to help prevent the sort of bugs you've seen so far in this chapter and to keep the app working in tip-top shape!

### 15.7.1 Testing for functionality

Let's start by adding unit tests to test that the BooksManager is sorting and searching the books array correctly.

Tests are performed in special targets in your project: one test target for unit tests and another test target for UI tests. Targets can contain multiple test *classes*, which are useful for grouping related tests. Each test class can contain multiple test *methods*, each performing a single test.

When you create a project, the project option screen gives you two checkboxes to set up your project with unit tests and UI tests. Selecting these checkboxes automatically adds appropriate testing *targets* to your project and a test *class* containing test *methods*.

Open the Test Navigator to see the tests that come in your project by default (see figure 15.30).

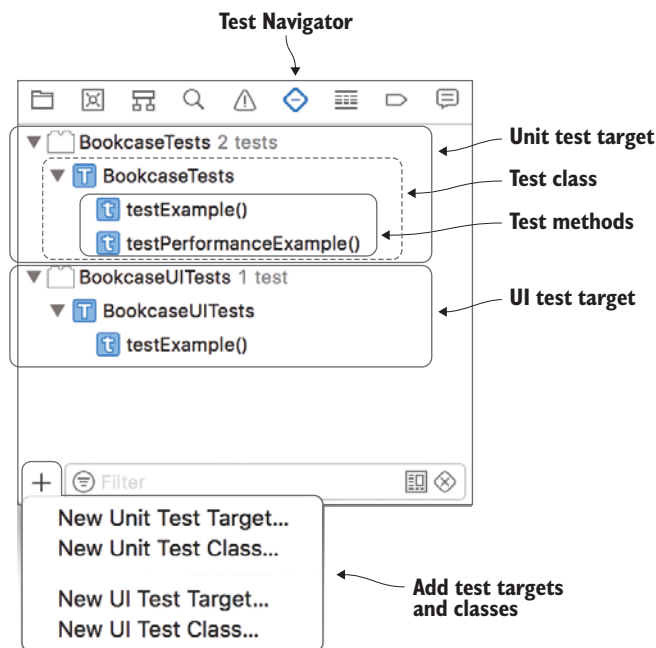


Figure 15.30 Default tests in the Test Navigator

If by chance you didn't select the testing checkboxes when you created your app, don't despair—it's easy enough to add test targets to your project. Select the + symbol at the bottom of the Test Navigator, give the target a name, and select the target to be tested. A test class will automatically be created with the same name as the target.

Let's use the same menu to add another test class (see figure 15.30) to test the `BooksManager` class.

- 1 Select the + symbol, and then select New Unit Test Class.
- 2 Name the test class "`BooksManagerTests`." A unit test class will appear with two default test methods: `testExample` and `testPerformanceExample`.
- 3 You can delete these two default test methods.

### SETTING UP YOUR TEST CLASS

To perform tests on the `BooksManager` class, you first need to set it up. To have complete control over the test data, it'd be a good idea to set that up in the test class, too.

You may have noticed your test class has a `setUp` method. This is a good place to specify any code that you want to run before each test method. This'll be the perfect place to instantiate the `BooksManager` and pass in test data to the `books` array. Because you know that these variables will necessarily be instantiated prior to the test methods, you can confidently set these to implicitly unwrapped optionals.

- 1 Set up the `BooksManager` and test data.

```
var booksManager: BooksManager!
var bookDaVinci: Book!
var bookGulliver: Book!
var bookOdyssey: Book!

override func setUp() {
    super.setUp()
    bookDaVinci = Book(title: "The Da Vinci Code",
        author: "Dan Brown", rating: 5, isbn: "", notes: "")
    bookGulliver = Book(title: "Gulliver's Travels",
        author: "Jonathan Swift", rating: 5, isbn: "", notes: "")
    bookOdyssey = Book(title: "The Odyssey",
        author: "Homer", rating: 5, isbn: "", notes: "")
    booksManager = BooksManager()
    booksManager.addBook(bookDaVinci)
    booksManager.addBook(bookGulliver)
    booksManager.addBook(bookOdyssey)
}
```

**NOTE** You've probably noticed a `tearDown` method as well. You can specify any code you want to run *after* each test method here.

You'll see errors basically on every line, for example: *Use of undeclared type 'BooksManager'.*

By default, files in one target don't have access to files in another. If you select the `BooksManager` file in the Project Navigator, and select the File Inspector,



you'll find that this file is only set to be accessible from within the Bookcase target (see figure 15.31).

You *could* add test target membership checking the checkboxes in figure 15.31 for *every file* your test class needs to access, but there's a much quicker and easier solution! You can give your test class access to your app target files by simply *importing* the app target with a `@testable` attribute.

- 2 Add a *testable import* at the top of your `BooksManagerTests` file to make classes in the Bookcase target visible to your test target.

```
@testable import Bookcase
```

The errors should go away, and you're ready to start filling out your test methods.

### ADDING TESTS TO YOUR TEST CLASS

Let's start by creating a test method that tests that the `booksManager` is sorting the books correctly by *title*.

- 1 Add a method called `testSortTitle`.

```
func testSortTitle() {  
}
```

- 2 Because you want to test sorting by *title* in this method, set the `sortOrder` property in the `BooksManager` to `title`.

```
booksManager.sortOrder = .title
```

Great, so your test method is set up, but how does it perform a test?

To create a test, first consider what you're expecting as the correct result. In this case, after sorting by title, you would expect that the `books` array will be sorted in a certain order: "Gulliver's Travels," "The Da Vinci Code," then "The Odyssey."

In Xcode, you express this expectation with what's called an *assertion*. The basic assertion is expressed with the `XCTAssert` method. This method requires a Boolean expression—if it returns `true`, the test has passed. Conversely, if it returns `false`, the test has failed.

- 3 Assert the order of the sorted array:

```
XCTAssert(booksManager.getBook(at: 0) == bookGulliver)  
XCTAssert(booksManager.getBook(at: 1) == bookDaVinci)  
XCTAssert(booksManager.getBook(at: 2) == bookOdyssey)
```

That's it—you're ready to run your test! Because your method starts with the word "test," Xcode automatically recognizes that it's a test method and indicates this with a diamond beside the method.

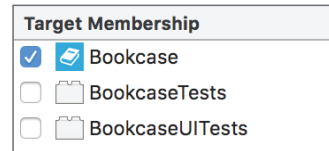


Figure 15.31 BooksManager.swift target

- 4 Hover over this diamond, and it should become a Play button. Click on this Play button, and the test method you just created should run.

If the test is successful, the diamond will display a green tick, while an unsuccessful test will display a red cross (see figure 15.32).

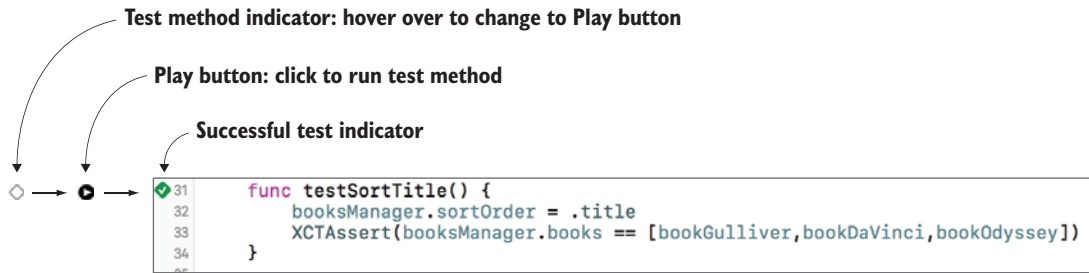


Figure 15.32 Test method

Several assertion methods expand on the basic `XCTAssert` method, performing various common test assertions such as equality, inequality, greater than, less than, and so on.

- 5 Add another test method to test the sort by author function. This time, use the `XCTAssertEqual` method:

```

func testSortAuthor() {
    booksManager.sortOrder = .author
    XCTAssertEqual(booksManager.getBook(at: 0) == bookDaVinci)
    XCTAssertEqual(booksManager.getBook(at: 1) == bookOdyssey)
    XCTAssertEqual(booksManager.getBook(at: 2) == bookGulliver)
}

```

- 6 This time, run both tests in this class by selecting the Run test button next to the class declaration. You should end up with two successful tests. You can also see your successful and unsuccessful tests in the test navigator.

**CHALLENGE** Create a functional test method to test searching the books array. You'll find my solution in the repo coming later in this chapter!

Great! If you make changes to your app now, you can be sure by running your tests that your books should still sort and search correctly.

## 15.7.2 Testing for performance

Unit tests aren't only about whether a unit of code is correct or incorrect—*performance* unit tests permit you to accurately analyze the efficiency of a unit of code. Performance tests run a unit of code 10 times and give you the average execution time.

Let's add a performance unit test to analyze the efficiency of the image color detection algorithm that your friend introduced.

- 1 As you did in the previous section, add a new unit test class called `UIImage-ColorDetectionTests` to test the `UIImageColors` framework, and remove the default test methods.
- 2 You're going to need an image to detect colors. Add an image variable and set it up in the `setUp` method.

```
var image: UIImage!
override func setUp() {
    super.setUp()
    image = UIImage(named: "book")
}
```

To analyze the performance of a unit of code, run it in a closure passed to the `measure` method.

- 3 Create the `testColorDetection` test method, and measure the performance of the `getColors` method.

```
func testColorDetection() {
    self.measure {
        self.image.getColors()
    }
}
```

Because this `UIImage` extension comes from a third-party binary framework that's not compiled by Carthage for testing, the `@testable` attribute won't work.

- 4 Instead, select the `UIImageColors` framework in the Project Navigator, and check the `BookcaseTests` target in the File Inspector to make this framework available to your unit tests.
- 5 Run the test by clicking the Play button beside the test method. An average time will appear after the `measure` closure, along with a gray diamond.
- 6 Click to the left of the Play button for more information about performance (see figure 15.33).
- 7 Select the Set Baseline button in the performance result.

Future tests will now be based on this baseline. If something changes in this third-party code in the future, and it becomes significantly less efficient than this baseline, you'll know about it when this performance test fails.



Figure 15.33 Performance result

### Silence the warning!

Because you're only testing the performance of the method, you aren't interested in the returned result. The Xcode compiler finds this strange and warns you of the unnecessary function call. To silence the warning, you can explicitly ignore the result by assigning it to an underscore:

```
_ = self.image.getColors()
```

### 15.7.3 Testing your user interface

User interface testing tests your app from a different perspective than unit testing. While functionality and performance can still be tested, UI testing shifts the focus from testing units of code to testing the user experience of your app.

Let's explore UI tests by creating one to test a user experience in your app. If you select the Info button in the book edit form, the ISBN field should appear. If you select the Info button again, it should disappear. Let's test that this functionality is working correctly.

UI tests are created in a separate target to the app and unit tests.

- 1 Find the BookcaseUITests test target that was generated when the Bookcase project was created, and open the default test class BookcaseUITests.
- 2 Create a new test method called testToggleISBN.

Your test class accesses the application via the `XCUIApplication` object, which is launched by default in the `setUp` method. You can use this object to access

interface elements in various ways. For example, to get a reference to the Add button in the navigation bar, you could type

```
let addButton = XCUIApplication().navigationBars["Books"].buttons["Add"]
```

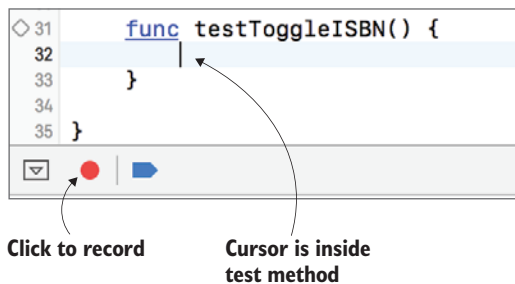
This gets a reference in the application to the navigation bar with the title Books, and then within the navigation bar finds a reference to the Add button. With this reference, you can now simulate the user tapping the button.

```
addButton.tap()
```

This is great, but with all this syntax, all you've achieved is a button tap. What happens when you want to test a longer and more complex user experience with multiple interactions? Setup would be a time-consuming and frustrating process.

Fortunately, Xcode allows you to *record* a user experience live and automatically convert to UI test sequences of code. If you entered the `addButton` code, delete it now. You're going to set up this UI test by recording it!

- 3 Ensure your cursor is inside the `testToggleISBN` method, and press the Record button (see figure 15.34).



**Figure 15.34** Record UI test

The app will launch, and the Stop Recording button will replace the Record button in the debug bar.

- 4 Select the Add button. A UI test action will automatically be added to the `testToggleISBN` method:

```
XCUIApplication().navigationBars["Books"].buttons["Add"].tap()
```

- 5 Now that you're in the book edit form, select the Info button. Again, Xcode will automatically add this action to your test, even refactoring the first line to set up a convenience variable to hold the application object:

```
let app = XCUIApplication()
app.navigationBars["Books"].buttons["Add"].tap()
app.scrollViews.otherElements.buttons["More Info"].tap()
```

To check that the ISBN field has been toggled, you'll need a reference to the ISBN field.

- 6 To find how to reference the ISBN label, click on it. You'll find that Xcode once again has refactored your code, setting up a property to hold the elements in the interface:

```
let elementsQuery = app.scrollViews.otherElements
elementsQuery.buttons["More Info"].tap()
elementsQuery.staticTexts["ISBN:"].tap()
```

Great, with little effort on your part, you know how to reference the ISBN field! You can stop the recording now, because you're going to finish writing the test yourself!

- 7 Press the Stop Recording button. You're going to refactor the test yourself. You only tapped the ISBN field to get a reference to it.
- 8 Remove the line tapping the ISBN label and instead use the reference to determine whether the ISBN label exists in the interface prior to tapping the Info button. You can do this with the `exists` method:

```
elementsQuery.staticTexts["ISBN:"].tap()  
let isbnExists = elementsQuery.staticTexts["ISBN:"].exists  
elementsQuery.buttons["More Info"].tap()
```

Now, you're ready to make an assertion. Tapping the Info button should have toggled the existence of the ISBN field in the interface.

- 9 Confirm that the ISBN field's existence has toggled with a call to `XCTAssertNotEqual`.

```
XCTAssertNotEqual(elementsQuery.staticTexts["ISBN:"].exists, isbnExists)
```

You've set up your first UI test!

- 10 As you did with unit tests earlier in the chapter, run the test by tapping the Play button beside the method.

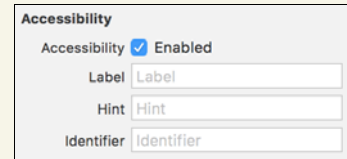
The app will run in the simulator, automatically performing the actions defined in the test method. With any luck, it should eventually highlight a successful test with a green tick.

### Accessibility

For a user interface to be testable, its interface elements need to have *accessibility* enabled. But even if accessibility wasn't required for UI testing, it's still best practice to ensure that your interface is accessible.

**(continued)**

Select an interface element and open the Identity Inspector. There, you'll find the accessibility panel. Here, you can provide a *label* to describe the element, a *hint* to describe the result of interacting with the element, and a unique *identifier* for the element.



Beneath these properties are a number of trait checkboxes, such as *Button*, *Selected*, *Image*, *Search Field*, and *Static Text*. These properties give the operating system a better understanding of how the element is expected to behave.

Adding accessibility properties to the visual elements in your app will open them up to be described by the VoiceOver accessibility app, and enable users with impaired vision to use your app.



**CHECKPOINT** If you'd like to compare your project with mine at this point, you can check mine out at <https://github.com/iOSApp-DevelopmentwithSwiftinAction/Bookcase.git> (Chapter15.3.Tested).

## 15.8 Summary

In this chapter, you learned the following:

- Different methods exist for examining the contents of a variable, each with their own advantages. Check table 15.2 for a summary.
- Debugging in Xcode is a massive topic, and the tools available for exploring your app are extensive. One chapter can't cover everything—if you'd like to explore further, check out the memory graph debugger, instruments tools, and type “help” into the lldb command line.
- Use *functional* tests to test that something does what it should, and use *performance* tests to confirm that a process is taking an appropriate amount of time, compared with a baseline.
- Unit tests test from the perspective of units of code, while UI tests test from the perspective of the user experience of your app.
- Ensure that the elements in your app are accessible.
- For further reading on testing, check out Apple's documentation on testing at [https://developer.apple.com/library/content/documentation/Developer-Tools/Conceptual/testing\\_with\\_xcode](https://developer.apple.com/library/content/documentation/Developer-Tools/Conceptual/testing_with_xcode). Look at how to perform *asynchronous* testing.

# iOS Development with Swift

Craig Grummitt



**O**ne billion iPhone users are waiting for the next amazing app. It's time for you to build it! Apple's Swift language makes iOS development easier than ever, offering modern language features, seamless integration with all iOS libraries, and the top-notch Xcode development environment. And with this book, you'll get started fast.

**iOS Development with Swift** is a hands-on guide to creating iOS apps. It takes you through the experience of building an app—from idea to App Store. After setting up your dev environment, you'll learn the basics by experimenting in Swift playgrounds. Then you'll build a simple app layout, adding features like animations and UI widgets. Along the way, you'll retrieve, format, and display data; interact with the camera and other device features; and touch on cloud and networking basics.

## What's Inside

- Create adaptive layouts
- Store and manage data
- Learn to write and debug Swift code
- Publish to the App Store

Written for intermediate web or mobile developers. No prior experience with Swift assumed.

**Craig Grummitt** is a successful developer, instructor, and mentor. His iOS apps get over 100,000 downloads.

“A practical approach, with lots of real-world examples.”

—Andrea Prearo, Capital One

“More than just a guide to learning Swift, this book demonstrates concepts useful for any language.”

—Becky Huett, Big Shovel Labs

“A self-contained step-by-step tutorial with plenty of examples.”

—Ghita Kouadri  
University College London

“Provides comprehensive knowledge of Swift 4 combined with clear explanations of iOS key concepts and APIs.”

—Žarko Jović, Quandoo Berlin

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit  
[www.manning.com/books/ios-development-with-swift](http://www.manning.com/books/ios-development-with-swift)

ISBN-13: 978-1-61729-407-5  
ISBN-10: 1-61729-407-1



9 781617 294075



\$49.99 / Can \$65.99 [INCLUDING eBook]