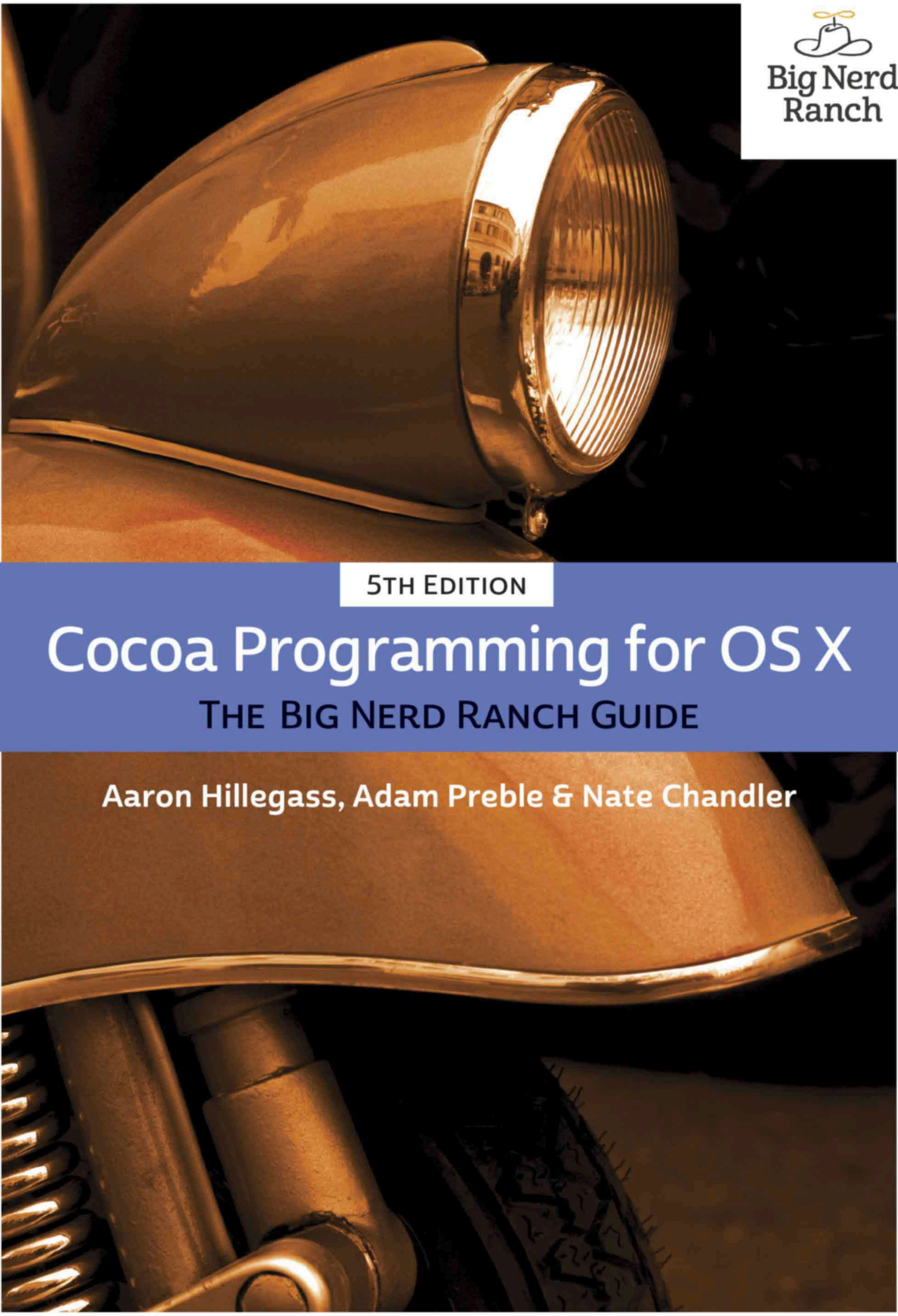


Big Nerd Ranch

5th Edition

Cocoa Programming for OS X

The Big Nerd Ranch Guide

Aaron Hillegass, Adam Preble & Nate Chandler

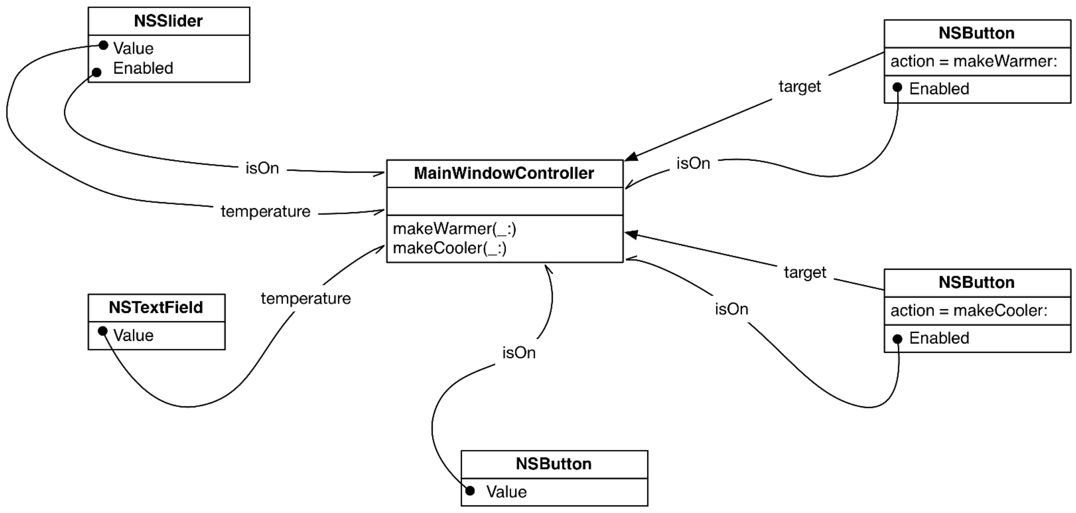
Cocoa Programming for OS X: The Big Nerd Ranch Guide

by Aaron Hillegass, Adam Preble and Nate Chandler

ISBN-10 0134077113
ISBN-13 978-0134077116

Fifth edition, first printing, April 2015
Release D.5.1.1

Figure 8.8 Object diagram with bindings



You are now likely wondering when to use bindings and when to update the view layer in your controller code. The answer, of course, is "it depends." Some Cocoa APIs require bindings, so you must use them there. If you have a relatively straightforward UI, bindings make an excellent choice. However, as you will see in a moment, bindings can be difficult to debug – there is no code to step through! We suggest that for more complicated user interfaces, you do things the hard way. As you gain more experience with bindings – and there is a lot more beyond what this book can cover – try using them in more complicated settings. One aspect of bindings that is easy to forget is that they are difficult to understand when you are new to a code base. It is hard to see them all at once (there are no diagrams like Figure 8.8), so learning what bindings are present can take a lot of poking around.

KVC and Property Accessors

KVC will call setters and getters for properties, if they are present. To see this in action, add accessors for `temperature`:

```
~~dynamic var temperature = 68~~
private var privateTemperature = 68
dynamic var temperature: Int {
    set {
        println("set temperature to \(newValue)")
        privateTemperature = newValue
    }
    get {
        println("get temperature")
        return privateTemperature
    }
}
```

Run the application and move the slider to change the temperature. Notice how the setter is called, followed by the getter, the latter of which corresponds to bindings updating the label in response to the KVO notification. If you use the buttons to change the temperature you will see the setter (property access) and two getter calls (updating the slider and label).

In this example you changed `temperature` to a computed property. In Swift, computed properties have no storage, so an additional property was needed to replicate the existing behavior of this application.

Why is `privateTemperature` marked `private`? Usually this pattern of a computed property with separate backing storage (the `privateTemperature` property) is used to control access to a value, so it makes sense to mark it as private to make sure that external code uses the intended interface for changing it – in this case the computed property.

KVC and nil

What do you think happens if you use KVC to set a numeric type, such as a `Float` or `Int`, to `nil`?

```
setValue(nil, forKey: "temperature")
```

Instead of making an assumption, such as assigning the value `0`, KVC's designers decided to let the object decide what to do. When `nil` is set for numeric type, KVC calls the method **`setNilValueForKey(_:)`**. The default implementation on **`NSObject`** throws an exception.

Although it is not needed in this application – none of the controls will try to set it to `nil` – you could implement **`setNilValueForKey(_:)`** on **`MainWindowController`** to handle this situation:

```
    override func setNilValueForKey(key: String) {
        switch key {
        case "temperature":
            temperature = 68
        default:
            super.setNilValueForKey(key)
        }
    }
```

Under what circumstances could this be an issue? One case is a text field bound to a numeric property. If the user deletes all of the text and hits Return, the text field interprets this as `nil`. You will see another solution to this problem, as well as learn about a related topic, Key-Value Validation, in Chapter 10.

What about setting `nil` for other types? Because KVC depends on the Objective-C runtime, this can be tricky. Optional reference types will work as expected: the property will be set to `nil`. *Some* optional value types will work as well, such as `String?`, array, set, and dictionary optionals. This works because all of these value types have Objective-C object counterparts. Although this may change in future versions of Swift and OS X, setting a numeric optional, `Int?` for example, to `nil` via KVC presently results in a crash. You can work around this by using the optional Objective-C counterpart, `NSNumber?`.

Debugging Bindings

Bindings are a powerful feature of Cocoa, but they are notoriously difficult to debug. To a new Cocoa developer (and indeed even to the experienced), they can be quite mysterious. Because they are generally configured in the XIB file it can be difficult to track down the source of a problematic binding. Additionally, because they use strings, the compiler cannot help you by checking that the key actually exists on the object you are attempting to bind to.

The most common problem in using bindings is a key name typo, followed closely by binding to the wrong object. You will see this error at runtime when the related NIB file is opened: an exception will be logged to the console. There will be a couple dozen lines of output, but the most interesting line will be at the top of the output and look something like this:

```
[<Thermostat.MainWindowController 0x610000023260> valueForUndefinedKey:]:
    this class is not key value coding-compliant for the key temperatur.
```

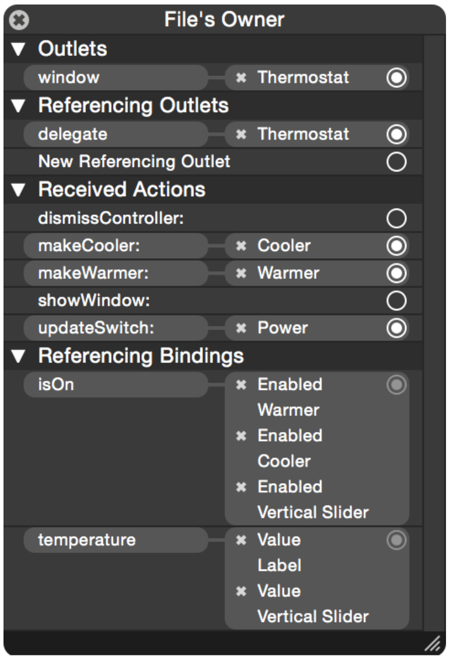
By pulling this line apart you can learn a few things about the error:

1. A binding was made to a non-existent key (**`valueForUndefinedKey:`**).
2. The target of the binding is **`MainWindowController`**.
3. The key string is `"temperatur"`.

When you see an exception like this, think through the components of the exception message. Did you expect to be binding to that key on that particular object? Is the object or the key wrong? Comparing the answers to those questions with your intentions will lead you to the source of the problem.

To see a list of all the bindings for a particular object, reveal its connections panel by Control-clicking it (or its placeholder) in the document outline. The bindings are listed at the bottom with the key in the left column and the object binding to it in the right column (Figure 8.9). Think about each of the bindings and make sure that it is what you expect.

Figure 8.9 Connections panel for MainWindowController


File's Owner

▼ Outlets

window	Thermostat

▼ Referencing Outlets

delegate	Thermostat
New Referencing Outlet	

▼ Received Actions

dismissController:	
makeCooler:	Cooler
makeWarmer:	Warmer
showWindow:	
updateSwitch:	Power

▼ Referencing Bindings

isOn	Enabled Warmer
	Enabled Cooler
	Enabled Vertical Slider
temperature	Value Label
	Value Vertical Slider

Using the Debugger

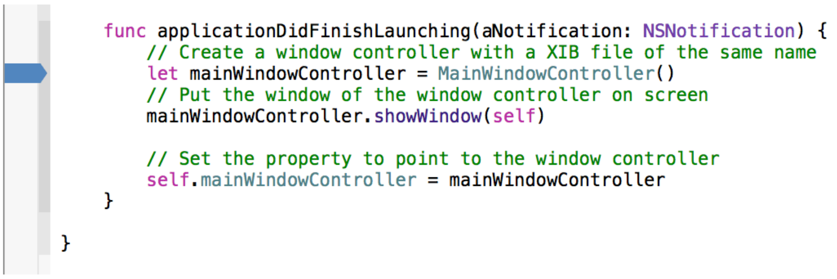
When an application is launched from Xcode, the debugger is attached to it. The debugger monitors the current state of the application, like what method it is currently executing and the values of the variables that are accessible from that method. Using the debugger can help you understand what an application is actually doing, which, in turn, helps you find and fix bugs.

Using breakpoints

One way to use the debugger is to set a *breakpoint*. Setting a breakpoint on a line of code pauses the execution of the application at that line (before it executes). Then you can execute the subsequent code line by line. This is useful when your application is not doing what you expected and you need to isolate the problem.

Open `AppDelegate.swift` and find the **`applicationDidFinishLaunching(_:)`** method. Set a breakpoint by clicking in the gutter (the lightly shaded bar on the left side of the editor area) next to the first line of code in the method (Figure 8.10). The blue indicator shows where the application will "break" the next time you run it.

Figure 8.10 A breakpoint

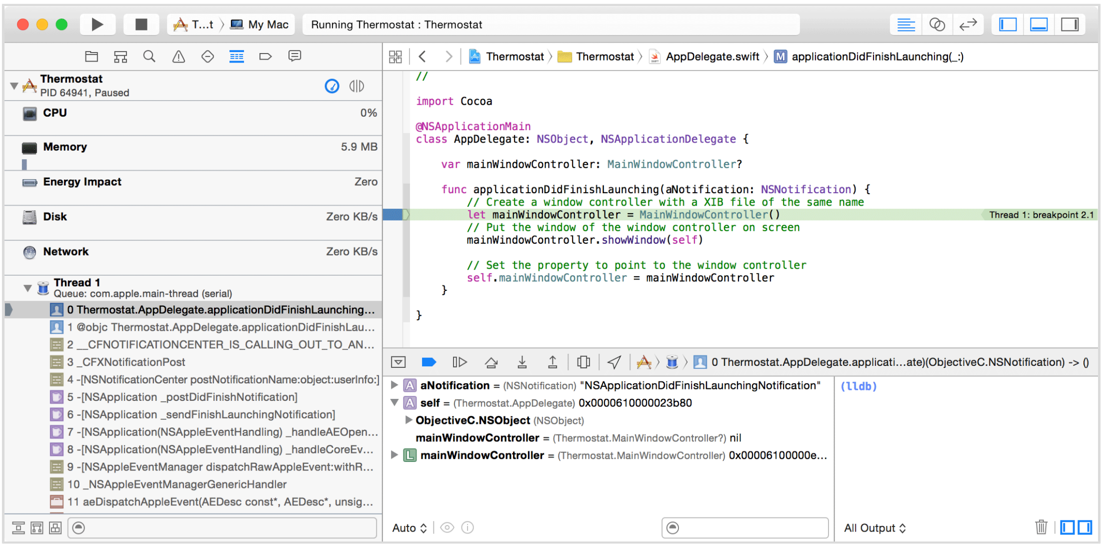


```
    func applicationDidFinishLaunching(aNotification: NSNotification) {
        // Create a window controller with a XIB file of the same name
        let mainWindowController = MainWindowController()
        // Put the window of the window controller on screen
        mainWindowController.showWindow(self)

        // Set the property to point to the window controller
        self.mainWindowController = mainWindowController
    }

}
```

Run the application. The application will stop execution before the line of code where you put the breakpoint is executed. Notice the light green indicator and shading that appear on that line to show the current point of execution.

Now your application is temporarily frozen in time, and you can examine it more closely. In the navigator area, click the 𝌆 tab to open the *debug navigator*. This navigator shows a *stack trace* of where the breakpoint stopped execution (Figure 8.11). A stack trace shows you the methods and functions whose frames were in the stack when the application halted.

Figure 8.11 The debug navigator and debug area


T...t
My Mac
Running Thermostat : Thermostat
Thermostat
Thermostat
AppDelegate.swift
applicationDidFinishLaunching(_:)
Auto
All Output

```
Thermostat
PID 64941, Paused
CPU                 0%
Memory              5.9 MB
Energy Impact       Zero
Disk                Zero KB/s
Network             Zero KB/s
Thread 1
Queue: com.apple.main-thread (serial)
0 Thermostat.AppDelegate.applicationDidFinishLaunching...
1 @objc Thermostat.AppDelegate.applicationDidFinishLau...
2 __CFNOTIFICATIONCENTER_IS_CALLING_OUT_TO_AN...
3 _CFXNotificationPost
4 -[NSNotificationCenter postNotificationName:object:userInfo:]
5 -[NSApplication _postDidFinishNotification]
6 -[NSApplication _sendFinishLaunchingNotification]
7 -[NSApplication(NSAppleEventHandling) _handleAEOpen...
8 -[NSApplication(NSAppleEventHandling) _handleCoreEv...
9 -[NSAppleEventManager dispatchRawAppleEvent:withR...
10 _NSAppleEventManagerGenericHandler
11 aeDispatchAppleEvent(AEDesc const*, AEDesc*, unsig...
```

```
//

import Cocoa

@NSApplicationMain
class AppDelegate: NSObject, NSApplicationDelegate {

    var mainWindowController: MainWindowController?

    func applicationDidFinishLaunching(aNotification: NSNotification) {
        // Create a window controller with a XIB file of the same name
        let mainWindowController = MainWindowController()        Thread 1: breakpoint 2.1
        // Put the window of the window controller on screen
        mainWindowController.showWindow(self)

        // Set the property to point to the window controller
        self.mainWindowController = mainWindowController
    }

}
```

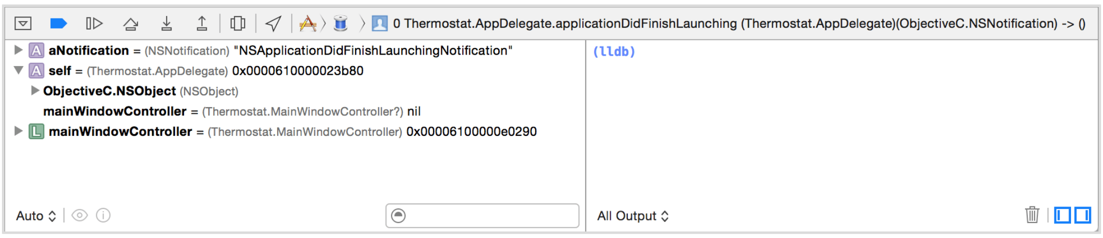
```
0 Thermostat.AppDelegate.applicati...ate)(ObjectiveC.NSNotification) -> ()
▶ aNotification = (NSNotification) "NSApplicationDidFinishLaunchingNotification"
▼ self = (Thermostat.AppDelegate) 0x0000610000023b80
  ▶ ObjectiveC.NSObject (NSObject)
    mainWindowController = (Thermostat.MainWindowController?) nil
▶ mainWindowController = (Thermostat.MainWindowController) 0x00006100000e...
```

```
(lldb)
```

The method where execution stopped is at the top of the stack trace. It was called by the method just below it, which was called by the method just below it, and so on. Notice that the methods that you have written code for are in black text while the methods belonging to Apple are in gray.

Select the method at the top of the stack, **`applicationDidFinishLaunching(_:)`**. In the debug area below the editor area, check out the variables view to the left of the console. This area shows the variables within the scope of **`MainWindowController`**'s **`makeWarmer(_:)`** method along with their current values. (Figure 8.12).

Figure 8.12 Debug area with variables view



```
0 Thermostat.AppDelegate.applicationDidFinishLaunching (Thermostat.AppDelegate)(ObjectiveC.NSNotification) -> ()
```

```
▶ aNotification = (NSNotification) "NSApplicationDidFinishLaunchingNotification"
▼ self = (Thermostat.AppDelegate) 0x0000610000023b80
  ▶ ObjectiveC.NSObject (NSObject)
    mainWindowController = (Thermostat.MainWindowController?) nil
▶ mainWindowController = (Thermostat.MainWindowController) 0x00006100000e0290
```

```
(lldb)
```

(If you do not see the variables view, find the ◫ button in the bottom-right corner of the debug area. Click the left icon to show the variables view.)

In the variables view you will see three variables: `aNotification`, `self`, and `mainWindowController`. Reference types are typically shown with their address – location in memory – while value types show a representation of their value. You can click the disclosure triangle to see more about non-scalar types. In Auto mode, the variables view attempts to show the most relevant variables. In this case that is the parameter of this method (`aNotification`), a reference to the current instance, `self`, since the debugger is in the context of an instance method, and the local variable `mainWindowController`.

You may have noticed that the local variable `mainWindowController` already has an address shown, but the line initializing it has not yet executed. This is simply because the compiler did not clear the memory, as an optimization.

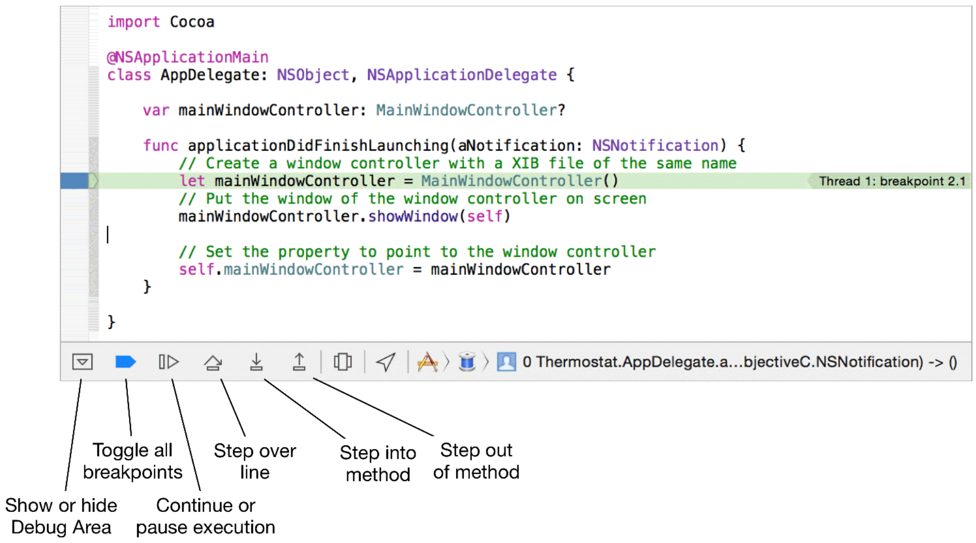
Click the disclosure button next to `self`. The first item under `self` is the superclass, in this case **`NSObject`**. The Swift module name, **`ObjectiveC`**, is also shown. Clicking the disclosure button next to a superclass will show the properties inherited from the superclass.

`AppDelegate` has one property shown, `mainWindowController`. Its value appears as nil because it is an optional that has not been assigned.

Stepping through code

In addition to giving you a snapshot of the application at a given point, the debugger also allows you to *step through* your code line by line and see what your application does as each line executes. The buttons that control the execution are on the *debugger bar* that sits between the editor area and the debug area (Figure 8.13). Keyboard shortcuts for these are shown in the Debug menu.

Figure 8.13 Debugger bar



```
import Cocoa

@NSApplicationMain
class AppDelegate: NSObject, NSApplicationDelegate {

    var mainWindowController: MainWindowController?

    func applicationDidFinishLaunching(aNotification: NSNotification) {
        // Create a window controller with a XIB file of the same name
        let mainWindowController = MainWindowController()
        // Put the window of the window controller on screen
        mainWindowController.showWindow(self)

        // Set the property to point to the window controller
        self.mainWindowController = mainWindowController
    }

}
```

When the line the debugger is stopped on is executed, a new instance of **`MainWindowController`** will be created and the local variable `mainWindowController` will be set. Click the Step Over button on the debugger bar. The debugger will stop again. Expand `mainWindowController`. The `internalTemperature` is shown, as is `isOn`.

If you expand its superclass, **`AppKit.NSWindowController`**, you will see that `_window` is `0x0`. You are now looking at the details of an Objective-C class. The `_window` *instance variable* (the backing storage for a property) has not yet been initialized because the window is not loaded from the NIB until the `window` property is accessed, which happens in **`showWindow(_:)`**.

Click Step Over once again to step over the call to **`showWindow(_:)`**. Note that `_window` now has an address.

Back at the top level of the variables view, if you expand `self` you will see that the `mainWindowController` property (which corresponds to `self.mainWindowController`) is still `nil`. Step over the next line, the assignment of that property, to see it change.

At this point, you could continue stepping through the code to see what happens. Or you could click the Continue button to let your program run until the next breakpoint. Or you could step into a method ( ). Stepping into a method takes you to the method that is called by the line of code that currently has the green execution indicator. Once you are in the method, you have the chance to step through its code in the same way.

When you step out of a method, you are taken to the method that called it. To try this, with Thermostat still stopped in the debugger, open `MainWindowController.swift` and add a breakpoint in the setter for the computed property `temperature`. Hit Continue . The app window will appear. Change the temperature using the Warmer or Colder button, and execution will stop again, this time in the setter. Click Step Out and you will find yourself in the action method that called the setter.