Extracted from:

# Xcode Treasures

## Master the Tools to Design, Build, and Distribute Great Apps



The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

# Xcode Treasures

## Master the Tools to Design, Build, and Distribute Great Apps

Chris Adamson
*edited by Tammy Coron*

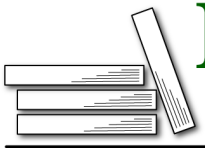
# Xcode Treasures

## Master the Tools to Design, Build, and Distribute Great Apps

Chris Adamson

The Pragmatic Bookshelf

Raleigh, North Carolina





Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Development Editor: Tammy Coron
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.



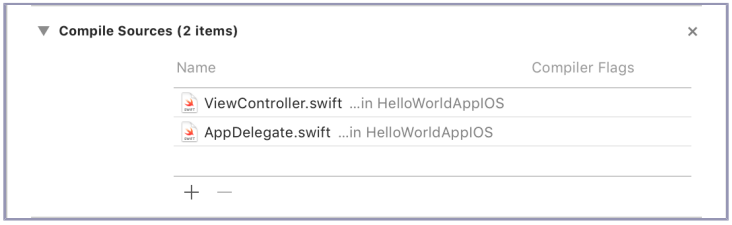
ISBN-13: 978-1-68050-586-3
Book version: P1.0—October 2018

## Build Phases

Build Settings are great and all, but they don't actually *do* anything in and of themselves. The next tab over, Build Phases, is where the action takes place. This tab describes each step of the build process in order.

Depending on what type of project you created, several phases will already be present. For an iOS or Mac app, there are default build phases to do the following:

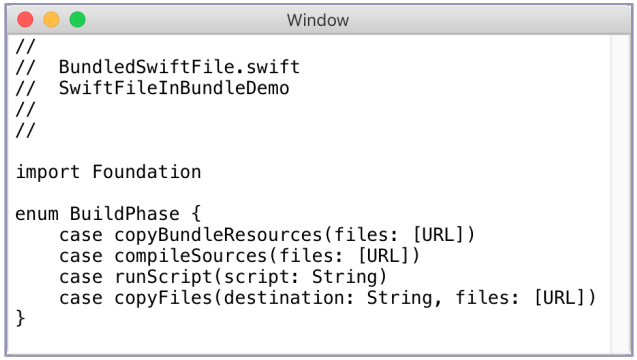
- Build any dependencies first, which are other targets in the project that need to be built before this one.
- Compile your source files into executable code.
- Link your executables with system libraries.
- Copy resource files, like storyboards, asset catalogs, other kinds of media, and so on.




You can expand a phase with the disclosure triangle on the left. Each phase contains a list of files that the phase applies to, along with + and - buttons to add and remove files from the phase. So, for the Compile Sources phase, all .swift, .c, .m (Objective-C) and other source files are automatically added to the compile phase as you add them to the project. Same goes for resource files in the Copy Bundle Resources phase.

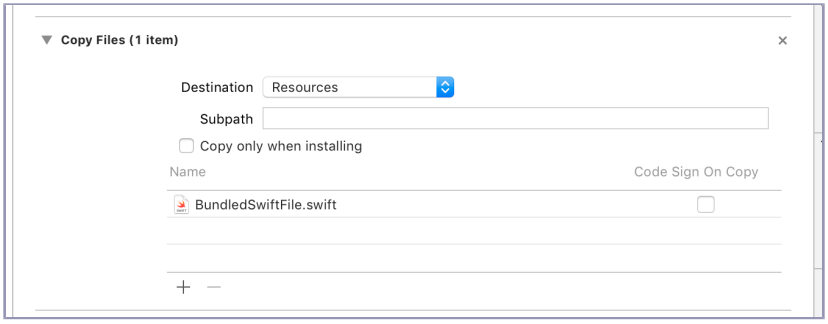
### Copy File Phases

In rare occasions, you might want to tweak this. Imagine, for example, if you had a programming-tutorial app that needed to bundle .swift files for the user to view and edit. You wouldn't want to build these files, but instead copy them to the app-bundle as-is. In the following figure, BundledSwiftFile.swift is meant to be shown in a window, and not to actually be built as part of the app:




To do this, you need to do two things. First, go to the Compile Sources build phase and use the minus button to remove BundledSwiftFile.swift from the set of files to compile. Next, since the file isn't of a resource type (storyboards, asset catalogs, images, etc.), you can't just add it to the files in the Copy Bundle Resources rule. Instead, click the + at the top of the Build Phases tab and choose "New Copy Files Phase" (or use the menu item Editor -> New Build Phase -> Add Copy Files Build Phase).

This copy-files phase is different from the default Copy Bundle Resources phase, because it lets you specify a known destination within the app bundle, along with an optional subpath if you want to get fancy with your file structure. To read the file at runtime, the best place is the Resources folder, since that's in the search path used by the Bundle class' url(forResource:withExtension:) and similar resource-loading methods.




With your file copied to this expected location, reading it at runtime is easy:

building/SwiftFileInBundleDemo/SwiftFileInBundleDemo/ViewController.swift

```
guard let sourceURL = Bundle.main.url(
    forResource: "BundledSwiftFile", withExtension: "swift"),
    let sourceText = try? String(contentsOf: sourceURL) else { return }
textView.string = sourceText
```

### Build Rules

To the right of the Build Phases tab, there's one more tab that you'll probably never need to use, but which answers an important question: *how does Xcode know what to do with each file type?* The *Build Rules* tab is a list of file types and actions to take on them. You can search through the list to find the rule that manages the existing types. You can also use the + button to define a new rule, which matches files either by known types or by a substring you choose (like a file extension), and can then run one of several dozen built-in actions, or run an arbitrary script.

This lets you add pretty much any kind of processing to an Xcode build. For example, if you had a compiler for some arbitrary language that produced code in an appropriate binary format (x86_64 for Mac, ARM for iOS devices), you could add a rule to run a script to call that compiler, and then write app sources in that language. And that's great news for anyone with legacy FORTRAN code from 1965 that they want to embed in an iPhone app, right?

## Run Script Phases

Along with tweaking the files processed by the built-in rules, and having the ability to copy files to the target, there's one more option to really open up what you can do with builds: the Run Script phase. This allows you to write a shell script that can do, well, anything a shell script on your Mac can do. For example, the SwiftLint[1] code checker uses custom scripts to scan your source code and enforce good Swift coding style.

As an example, I once hid an Easter egg in one of my apps that would show what I was playing in iTunes at the time the build was performed. You can get the current song title with a three-line AppleScript:

```
tell application "iTunes"
    get the name of the current track
end tell
```

Next, with the command-line utility `osascript`, you can run AppleScripts by either separating each line with the `-e` flag, or providing a source file as an argument. So you can get the iTunes current song title written to standard out like this (note that this listing uses the `\` line-wrap operator to split the command over several lines to fit the book's formatting; you can write it all as one line):

```
⇛ osascript -e "tell application \"iTunes\"" \
⇛ -e "get the name of the current track" \
```

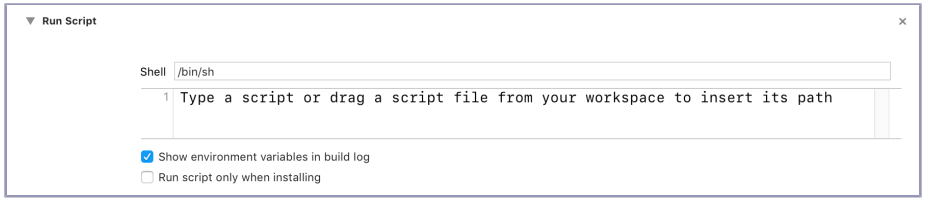
1. https://github.com/realm/SwiftLint

```
⇛ -e "end tell"
❮ Gonna Needa Pasteboard
```

So, you can probably imagine that with a series of osascript calls, you can extract whatever you need from iTunes. Now you'd need a way to get them into a file that could then be copied into the build. You could write out a simple text file, but for build scripts, you can make use of a wonderful command-line utility called PListBuddy. This executable, which lives in /usr/libexec, can read and write individual entries from plist files, which in turn can be easily read into memory as NSArray and NSDictionary objects.

PListBuddy's commands can by shown with its -h command. For this demo, all you need to know is that you can say PListBuddy -c "Add *key-name value-type value file-name*" to provide the value of a key in a given .plist file, which will be created automatically if it doesn't already exist.

With these two tools, you have everything you need to write a script to set up the needed file inside the bundle. Start by clicking the + button and choose "New Run Script Phase".




There's one other thing you need to know for this script to work: where to write the .plist file. Fortunately, all the build settings described earlier are available in scripts, so $TARGET_BUILD_DIR contains the path to the directory where the app is being built. And that means you can write a file inside the bundle by using this path and appending the app name. Then you need to know where to put a file inside the app bundle so the Bundle class can find it at run-time. On iOS, just put your file in the top-level directory, and on macOS, put it in Contents/Resources.

So here's a script to create the Easter egg file (like before, this has to use bash's line-wrap syntax to fit the formatting of the book):

```
tmpfile=$(mktemp /tmp/tunes.txt)
rm $TARGET_BUILD_DIR/BuildScriptEasterEggDemo.app/BuildTunes.plist > \
/dev/null 2>&1
osascript -e "tell application \"iTunes\"" \
-e "get the name of the current track" \
-e "end tell" > $tmpfile
/usr/libexec/PListBuddy -c "Add :SongTitle string $(cat $tmpfile)" \
$TARGET_BUILD_DIR/BuildScriptEasterEggDemo.app/BuildTunes.plist
osascript -e "tell application \"iTunes\"" \
-e "get the artist of the current track" -e \
"end tell" > $tmpfile
/usr/libexec/PListBuddy -c "Add :SongArtist string $(cat $tmpfile)" \
$TARGET_BUILD_DIR/BuildScriptEasterEggDemo.app/BuildTunes.plist
rm "$tmpfile"
```

This script starts by creating a temporary file descriptor and deletes any BuildTunes.plist file left over from an earlier run (writing any output or errors to /dev/null so Xcode doesn't see them as errors and stop the build). Then it does a call to osascript that writes the song title to the temp file, and a call to PListBuddy to write the temp value to the BuildTunes.plist file. Next, it repeats these steps with the song artist. Finally, it deletes the temp file.

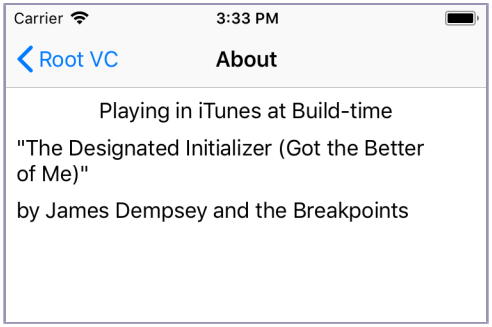
Try a build, look in the package contents of the app file, and you'll see the BuildTunes.plist file. Now all you need to do is read it at runtime:

building/BuildScriptEasterEggDemo/BuildScriptEasterEggDemo/AboutViewController.swift

```
guard let songInfoURL = Bundle.main.url(forResource: "BuildTunes",
                                        withExtension: "plist"),
    let songNSArray = NSDictionary(contentsOf: songInfoURL),
    let title = songNSArray["SongTitle"] as? String,
    let artist = songNSArray["SongArtist"] as? String else {
        titleLabel.text = "Didn't find"
        artistLabel.text = "Didn't find"
        return
}
titleLabel.text = "\"\(title)\""
artistLabel.text = "by \(artist)"
```

And that's all it takes. Just like the custom-copy-phase Swift file in the previous section, the newly created .plist file is waiting for you to find and use at runtime, as shown in the figure on page 10.

Granted, this is a silly exercise, but it should drive home the point that anything you can do on the command line—which pretty much means *anything*—can be part of your build process, thanks to the run script build phase. All you need are some mad bash skills, and all those Xcode build variables mentioned earlier.


Carrier
3:33 PM
Root VC
About
Playing in iTunes at Build-time
"The Designated Initializer (Got the Better of Me)"
by James Dempsey and the Breakpoints