

THE COMPLETE PRACTICAL GUIDE

Objective-C for Modern iOS and macOS Developers

Maintaining, Modernizing, and Migrating Legacy Code to Swift

ARC · Blocks · GCD · Runtime · UIKit · Core Data · Testing · Swift Interop · Incremental Migration · SwiftUI

28 CHAPTERS · 4 APPENDICES · CASE STUDIES

THENEXBOOKS

Objective-C for Modern iOS and macOS Developers

Maintaining, Modernizing, and Migrating Legacy Code to Swift

Understand It · Fix It · Modernize It · Move It Forward

First Edition

THENEXBOOKS

Objective-C for Modern iOS and macOS Developers
Maintaining, Modernizing, and Migrating Legacy Code to Swift

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Apple's tools, frameworks, and deprecation policies change over time; always consult Apple's current documentation.

Apple, iOS, iPadOS, macOS, Xcode, Swift, SwiftUI, Cocoa, Cocoa Touch, and Instruments are trademarks of Apple Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with or endorsed by Apple Inc.

Fieldbook, the example app in this book, is fictional. Any resemblance to real apps or companies is coincidental.

Preface

Swift has been Apple's recommended language for new code for years, so why a book about Objective-C now? Because a great deal of valuable software is still written in it. Many successful iOS and macOS apps were built before Swift existed, and much of that code still runs, still earns money, and still needs to be fixed, extended, and kept compatible with each new release of the operating system. Apple's own frameworks were designed around Objective-C, and their conventions still shape how Swift talks to them.

If you have just inherited one of these codebases, the experience can be disorienting. The square brackets, the header files, the `@property` attributes, the delegates and notifications, the runtime tricks that seem to work by magic — none of it looks like the Swift you know. And the people who wrote it may have left years ago.

This book is a practical guide for exactly that situation. We teach Objective-C from the ground up, with an eye on why it works the way it does. Then we go further: how to read and understand an unfamiliar legacy codebase, how to make it safer with tests, how to modernize it with nullability, generics, and current APIs, how Objective-C and Swift work together, and how to migrate to Swift and SwiftUI gradually — one feature at a time, without a risky rewrite.

Who This Book Is For

- **Swift developers** who need to work on an existing Objective-C codebase and want to do it confidently.
- **Developers joining teams** with long-lived iOS or macOS apps.
- **Experienced Objective-C developers** who want to modernize their code and plan a safe migration to Swift.
- **Tech leads** deciding what to keep, what to refactor, and what to migrate.

What You Will Learn

- Objective-C syntax, classes, properties, memory management with ARC, and the Foundation framework.
- Protocols, categories, blocks, Grand Central Dispatch, error handling, and the Objective-C runtime.
- Delegation, target-action, notifications, key-value coding, and key-value observing.
- Building apps with UIKit (and AppKit), networking with NSURLSession, and persistence with Core Data.
- How to read, test, debug, and modernize legacy Objective-C code safely.
- How Objective-C and Swift interoperate in both directions.
- How to plan and carry out an incremental migration to Swift and SwiftUI.

How This Book Is Organized

Part I — Objective-C Foundations (Chapters 1–5) teaches the language and Foundation. **Part II — The Language in Depth** (Chapters 6–10) covers protocols, blocks, concurrency, errors, the runtime, and Cocoa's communication patterns. **Part III — Building Apps with UIKit and AppKit** (Chapters 11–15) shows how classic Apple apps are put together. **Part IV — Maintaining Legacy Code** (Chapters 16–20) is about understanding, testing, debugging, and modernizing existing code. **Part V — Interoperability and Migration to Swift** (Chapters 21–25) connects the two languages and moves code across. **Part VI — Case Studies** (Chapters 26–28) brings it all together. The appendices provide a cheat sheet, an Objective-C-to-Swift translation reference, a checklist, and a glossary. Each chapter ends with a **Key Takeaways** box.

The Running Example: Fieldbook

Throughout the book we work on **Fieldbook**, a fictional field-research notebook app first released about ten years ago. Researchers record observations with notes, locations, tags, and photos, organize them into projects, sync with a server, and export data. Fieldbook has all the traits of real legacy code: a giant data-manager singleton, massive view controllers, hand-written JSON parsing, and very few tests. Over the course of the book we learn its code, make it safer, modernize it, and migrate key features to Swift and SwiftUI.

Conventions Used in This Book

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, history, or a useful aside.

> **WARNING**
> A common mistake that causes crashes, leaks, or subtle bugs.

Code appears in `monospace`, with the file name above each example. Objective-C examples use ARC. Where Swift is shown, it uses current Swift syntax.

Contents

Preface

PART I — OBJECTIVE-C FOUNDATIONS

1 Why Objective-C Still Matters

2 The Toolchain: Xcode, Clang, Projects, and Your First Program

3 Syntax Essentials: C Roots, Messages, Classes, Methods, and Headers

4 Objects, Properties, and Memory Management with ARC

5 The Foundation Framework: Strings, Collections, Numbers, and Dates

PART II — THE LANGUAGE IN DEPTH

6 Protocols, Categories, and Class Extensions

7 Blocks and Grand Central Dispatch

8 Error Handling: NSError, Exceptions, and Assertions

9 The Objective-C Runtime: Messaging, Introspection, and Method Swizzling

10 Communication Patterns: Delegation, Target-Action, Notifications, KVC, and KVO

PART III — BUILDING APPS WITH UIKIT AND APPKIT

11 UIKit Fundamentals: Lifecycle, View Controllers, and the Responder Chain

12 Views, Auto Layout, Table Views, and Collection Views

13 Navigation, Storyboards, XIBs, and Building UI in Code

14 Networking with NSURLSession and JSON

15 Persistence: NSUserDefaults, Files, NSSecureCoding, and Core Data

PART IV — MAINTAINING LEGACY CODE

16 Reading and Understanding a Legacy Codebase

17 Modern Objective-C: Nullability, Lightweight Generics, and Modern Syntax

18 Testing Legacy Objective-C with XCTest

19 Debugging and Instruments: Leaks, Retain Cycles, Zombies, and Crashes

20 Modernizing the Project: Build Settings, Warnings, Dependencies, and Deprecated APIs

PART V — INTEROPERABILITY AND MIGRATION TO SWIFT

21 Using Objective-C from Swift

22 Using Swift from Objective-C

23 Planning an Incremental Migration

24 Migrating Models, Networking, and Persistence to Swift

25 Migrating UI: From UIKit View Controllers to SwiftUI

PART VI — CASE STUDIES

26 Case Study: Modernizing Fieldbook Without a Rewrite

27 Case Study: Migrating the Observations Feature End to End

28 Living with a Mixed Codebase: Long-Term Maintenance and Team Practices

APPENDICES

A Objective-C Cheat Sheet

B Objective-C to Swift Translation Reference

C Legacy Modernization and Migration Checklist

D Glossary

PART I

Objective-C Foundations

Why so much important code is still written in Objective-C, how the toolchain works, and the core of the language: messages, classes, properties, ARC, and Foundation.

1 · Why Objective-C Still Matters
2 · The Toolchain
3 · Syntax Essentials
4 · Objects, Properties, and ARC
5 · The Foundation Framework

CHAPTER 1 · PART I — OBJECTIVE-C FOUNDATIONS

Why Objective-C Still Matters: Legacy Apps, Apple Frameworks, and Your Career

Objective-C is no longer the language Apple promotes on stage, yet a great deal of working software, and much of the platform itself, still speaks it. This chapter explains why that is, what it means for you as a developer, and how this book will take you from reading your first square bracket to migrating a real codebase to Swift without breaking it.

A Language That Refuses to Disappear

When Apple introduced Swift, many developers assumed Objective-C would fade away within a few years. Swift was safer, more expressive, and clearly the future of the platform. New tutorials, new sample code, and eventually new frameworks such as SwiftUI arrived in Swift first, and often in Swift only. It would be easy to conclude that Objective-C is a historical curiosity, something you might read about but never need to touch.

Reality is more interesting. Software that works tends to stay in production for a long time, and companies do not rewrite working products simply because a newer language exists. An app that has served its users well for a decade represents years of bug fixes, edge cases, customer feedback, and hard-won domain knowledge. All of that knowledge is encoded in its source code. If the source code is Objective-C, then someone has to be able to read it, change it safely, and decide how much of it to move to Swift and when.

That someone may well be you. Perhaps you are a Swift developer who has just joined a team and discovered that the networking layer, the data model, and half the view controllers are written in a language you have never studied. Perhaps you have inherited a macOS utility that a colleague wrote years ago and that customers still depend on. Perhaps you wrote Objective-C yourself, long ago, and now need to bring an aging project up to modern standards and plan its journey to Swift. This book is written for all three situations.

We will treat legacy code with respect throughout. Code that has been in production for years is not bad code simply because it is old. It survived contact with real users, real devices, and real deadlines. It may have rough edges, outdated patterns, and missing tests, but it also pays the bills. Our goal is never to sneer at it; our goal is to understand it well enough to improve it with confidence.

Where Objective-C Came From

To understand why Objective-C looks the way it does, it helps to know a little of its history. Objective-C was created in the 1980s as a small set of extensions to the C programming language. Its designers wanted to bring the message-passing object model of Smalltalk to C, without giving up C's speed, portability, and access to the operating system. The result is a language that is a strict superset of C: every valid C program is, in principle, also a valid Objective-C program, and Objective-C adds classes, objects, and messages on top.

The language became the foundation of NeXTSTEP, the operating system developed by NeXT. When Apple acquired NeXT in the late 1990s, NeXTSTEP's technologies became the basis of Mac OS X, and later of iOS. That lineage is still visible every day. The `NS` prefix you see on `NSString`, `NSArray`, and `NSObject` comes from NeXTSTEP. The Foundation framework and the AppKit framework descend directly from those NeXT libraries, and UIKit was designed in the same tradition.

The object part of Objective-C is also unusual. In many languages, calling a method is resolved largely at compile time. In Objective-C, you *send a message* to an object, and the decision about which code runs is made at run time by the Objective-C runtime. That dynamism makes features such as delegation, target-action, Key-Value Observing, and Interface Builder connections natural. It also enables risky tricks, such as replacing method implementations at run time, which we will examine with appropriate caution in Chapter 9.

What Objective-C code looks like

If you have only seen Swift, your first encounter with Objective-C can feel like reading a familiar language through a funhouse mirror. Here is a small class from Fieldbook, the app we will use throughout this book. First, the header file, which declares the public interface:

`Fieldbook/Model/FBObservation.h`

```
#import <Foundation/Foundation.h>

NS_ASSUME_NONNULL_BEGIN

@interface FBObservation : NSObject

@property (nonatomic, copy) NSString *title;
@property (nonatomic, copy, nullable) NSString *notes;
@property (nonatomic, strong) NSDate *timestamp;

- (instancetype)initWithTitle:(NSString *)title;
- (NSString *)summaryLine;

@end

NS_ASSUME_NONNULL_END
```

Next, the implementation file, which contains the code:

Fieldbook/Model/FBObservation.m

```
#import "FBObservation.h"

@implementation FBObservation

- (instancetype)initWithTitle:(NSString *)title {
    self = [super init];
    if (self) {
        _title = [title copy];
        _timestamp = [NSDate date];
    }
    return self;
}

- (NSString *)summaryLine {
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    formatter.dateStyle = NSDateFormatterShortStyle;
    formatter.timeStyle = NSDateFormatterShortStyle;
    NSString *when = [formatter stringFromDate:self.timestamp];
    return [NSString stringWithFormat:@"%@ (%@)", self.title, when];
}

@end
```

For comparison, a roughly equivalent Swift class looks like this:

Swift equivalent (for comparison only)

```
import Foundation

final class Observation {
    var title: String
    var notes: String?
    var timestamp: Date

    init(title: String) {
        self.title = title
        self.timestamp = Date()
    }

    func summaryLine() -> String {
        let formatter = DateFormatter()
        formatter.dateStyle = .short
        formatter.timeStyle = .short
        return "\(title) (\(formatter.string(from: timestamp)))"
    }
}
```

Several differences jump out immediately. Objective-C splits a class across two files: a header ending in `.h` that other files import, and an implementation ending in `.m`. Object types are written as pointers, such as `NSString *`. String literals start with an at sign, as in `@"%@ (%@)"`. Method calls use

square brackets, with the receiver first and the message second: `[formatter stringFromDate:self.timestamp]`. Keywords such as `@interface`, `@implementation`, and `@property` all begin with an at sign, which marks them as Objective-C additions to C.

None of this is difficult once you know the rules, and Chapter 3 walks through each piece in detail. For now, notice how much the two versions have in common. The class has the same properties, the same initializer, and the same method. The Foundation types map one to one: `NSDate` becomes `Date`, `NSDateFormatter` becomes `DateFormatter`. That close correspondence is not an accident. Swift was designed to work with the existing Apple frameworks, which were written for Objective-C, and that design is what makes gradual migration possible.

Why Objective-C Still Matters Today

There are several distinct reasons why Objective-C remains relevant. They overlap, but it is worth separating them, because each one affects a different kind of developer.

1. Long-lived applications

Many apps on the App Store and on the Mac were started before Swift existed or before it was mature enough for the team to adopt. Some of those apps have since been fully migrated. Many others are mixed: new features are written in Swift, while older modules remain in Objective-C because they work, they are well understood by the people who wrote them, and rewriting them would carry risk without delivering new value to users.

Enterprise apps, internal tools, medical and scientific software, point-of-sale systems, and mature consumer products often fall into this category. Their owners care deeply about stability. A rewrite that introduces regressions can cost far more than it saves. In these environments, the developer who can confidently maintain Objective-C code, and who can migrate it one safe step at a time, is extremely valuable.

2. The frameworks were designed for it

Even in a pure Swift project, you are surrounded by Objective-C ideas. UIKit and AppKit were designed around Objective-C patterns: delegates, data sources, target-action, notifications, selectors, and Key-Value Coding. When Swift code uses `#selector`, marks a method `@objc`, subclasses `NSObject`, or works with `NSNotificationCenter`, it is participating in the Objective-C runtime.

Understanding Objective-C therefore makes you a better Swift developer on Apple platforms. Error messages about unrecognized selectors, crashes involving KVO, and API names that look odd in Swift all make more sense when you know how the Objective-C side works. Apple's documentation still lets you switch many API references between Swift and Objective-C, and reading the Objective-C declaration often clarifies nullability, ownership, and naming decisions that appear surprising in Swift.

3. Interoperability is a daily task

Swift and Objective-C interoperate through bridging headers, which expose Objective-C declarations to Swift, and through a generated header ending in `-Swift.h`, which exposes compatible Swift declarations to Objective-C. In any mixed project, someone must understand both sides of that bridge. They must know why an Objective-C method appears in Swift as returning an implicitly unwrapped optional, why a Swift struct cannot be used from Objective-C, and how annotations such as `NS_SWIFT_NAME` improve the result. Part V of this book is devoted to exactly these questions.

> **NOTE**
>
> This book makes no claims about how many apps use Objective-C, how many jobs require it, or what such jobs pay. Those numbers change and are difficult to measure honestly. What we can say with confidence is that mixed Objective-C and Swift codebases are common in long-lived products, and that the skills to maintain and migrate them are practical and transferable.

Objective-C and Your Career

It is natural to wonder whether time spent learning an older language is time well spent. We believe it is, for reasons that go beyond any single job listing.

First, the ability to work in an unfamiliar, long-lived codebase is one of the most valuable skills a developer can have, whatever the language. Most professional work is maintenance and extension of existing systems, not greenfield development. The techniques you will practice in this book, such as reading code systematically, writing characterization tests before changing behavior, introducing seams, and migrating incrementally, apply far beyond Objective-C.

Second, teams with mixed codebases need people who can move comfortably between languages. A developer who can review an Objective-C pull request, explain why a retain cycle occurs in a block, and then design the Swift replacement for that module is a rare and useful colleague. Teams often have plenty of people who know modern Swift, and fewer who can safely change the older code that everything else depends on.

Third, Objective-C deepens your understanding of the platform. Concepts such as reference counting, message dispatch, autorelease pools, and the responder chain are not merely Objective-C trivia. They explain behavior you will observe in Swift apps too. When a Swift app leaks memory through a closure, the underlying problem is the same retain cycle that Objective-C developers have been fixing with weak references for years.

> **TIP**
>
> If you are a Swift developer, try reading the Objective-C declaration of an Apple API you use every day, such as `UITableViewDataSource`. Seeing the original selectors, the `@optional` sections, and the nullability annotations often explains why the Swift version looks the way it does.

Meet Fieldbook

Learning a language through disconnected snippets is tiresome, and it hides the most important lessons about maintenance, which only appear when code has history. Throughout this book we will therefore work with a single fictional application called **Fieldbook**.

Fieldbook is a field-research notebook first released about ten years ago. Researchers use it to record observations while working outdoors. Each observation has a title, free-form notes, GPS coordinates, a timestamp, tags, and optional photos. Observations are grouped into projects. The app syncs with a REST API at `https://api.fieldbook.example/v2/` and can export a project as a CSV file. The main app is written in Objective-C with UIKit for iOS, and there is a smaller macOS companion built with AppKit that researchers use to review and export their data at a desk.

Like many real apps of its age, Fieldbook has accumulated some characteristic legacy traits:

- **Massive view controllers.** `FBObservationListViewController` handles table display, searching, syncing, sorting, and exporting, all in one very long file.
- **A singleton data manager.** `FBDataManager` is reachable from everywhere through `[FBDataManager sharedManager]`, which makes testing hard and dependencies invisible.
- **Manual JSON parsing.** `FBSyncClient` converts dictionaries into model objects by hand, with type checks scattered throughout.
- **Persistence history.** The first versions archived objects with `NSCoding`; a later release migrated the data to Core Data, and some archive-reading code remains for older installations.
- **Delegate and notification spaghetti.** Changes propagate through a mixture of delegates, notifications, and occasional Key-Value Observing.
- **Missing nullability annotations** in many older headers, which makes those APIs awkward to use from Swift.
- **Few tests.** A handful of unit tests exist, mostly for date formatting.

Every class in the project uses the `FB` prefix: `FBObservation`, `FBProject`, `FBSyncClient`, `FBDataManager`, and so on. Objective-C has no namespaces, so prefixes are the conventional way to avoid name collisions between your classes, Apple's classes, and those of third-party libraries.

The fictional team that maintains Fieldbook wants to keep shipping features while gradually modernizing the code. Over the course of the book they will add nullability annotations, break up the massive view controller, introduce tests, replace deprecated APIs, and migrate the model, networking, and eventually parts of the user interface to Swift and SwiftUI. You will follow that journey chapter by chapter.

A first look at legacy Fieldbook code

To give you a flavor of what we will be working with, here is a fragment of the original `FBDataManager` header as it existed before the modernization effort began:

Fieldbook/Data/FBDataManager.h (legacy version)

```
#import <Foundation/Foundation.h>

@class FBObservation;
@class FBProject;

extern NSString * const FBDataManagerDidChangeNotification;

@interface FBDataManager : NSObject

+ (FBDataManager *)sharedManager;

- (NSArray *)allProjects;
- (NSArray *)observationsForProject:(FBProject *)project;
- (FBObservation *)observationWithIdentifier:(NSString *)identifier;
- (void)saveObservation:(FBObservation *)observation;
- (void)deleteObservation:(FBObservation *)observation;

@end
```

There is nothing wrong with this header from the compiler's point of view, and it has served the app for years. Yet a modern reader immediately has questions. What kind of objects does `allProjects` return? The plain `NSArray` does not say. Can `observationWithIdentifier:` return `nil` when no observation matches? Is it safe to pass `nil` to `saveObservation:`? What happens to callers when the data changes?

Here is the same header after the modernization work described in Chapter 17:

Fieldbook/Data/FBDataManager.h (modernized version)

```
#import <Foundation/Foundation.h>

@class FBObservation;
@class FBProject;

NS_ASSUME_NONNULL_BEGIN

extern NSNotificationName const FBDataManagerDidChangeNotification;

@interface FBDataManager : NSObject

@property (class, nonatomic, readonly) FBDataManager *sharedManager;

- (NSArray<FBProject *> *)allProjects;
- (NSArray<FBObservation *> *)observationsForProject:(FBProject *)project;
- (nullable FBObservation *)observationWithIdentifier:(NSString *)identifier;
- (void)saveObservation:(FBObservation *)observation;
- (void)deleteObservation:(FBObservation *)observation;

@end

NS_ASSUME_NONNULL_END
```

The behavior has not changed at all, but the header now documents itself. Lightweight generics state the element types of the arrays. Nullability annotations make it explicit that the lookup method can return `nil` and that every other pointer must be non-nil. The class property lets Swift callers write `FBDataManager.shared`, because Swift trims the repeated type name from the property, and the `NSNotificationName` type makes the notification constant appear as a proper notification name in Swift. These are small, safe, mechanical improvements, and they are exactly the kind of change this book will teach you to make with confidence.

From Swift, the modernized header imports as clean, idiomatic API:

How Swift sees the modernized header (simplified)

```
class FBDataManager : NSObject {
    class var shared: FBDataManager { get }
    func allProjects() -> [FBProject]
    func observations(for project: FBProject) -> [FBObservation]
    func observation(withIdentifier identifier: String) -> FBObservation?
    func save(_ observation: FBObservation)
    func delete(_ observation: FBObservation)
}
```

Notice that Swift renamed several methods automatically, such as turning `observationsForProject:` into `observations(for:)`. The exact names Swift chooses depend on its import rules, which we cover in Chapter 21, along with how to override them with `NS_SWIFT_NAME` when the automatic result is not what you want.

What Maintaining Objective-C Actually Involves

Before we start learning syntax, it is worth being honest about what day-to-day work on an Objective-C codebase looks like. It is rarely about writing large amounts of new Objective-C. More often it involves tasks like these:

- **Fixing bugs** in existing code: a crash report points to a line in a view controller, and you must understand the surrounding code well enough to fix it without introducing new problems.
- **Adding small features** to existing screens, where writing the change in Objective-C is simpler and safer than migrating the whole screen first.
- **Updating deprecated APIs**, such as replacing `NSURLConnection` with `NSURLSession` or `UIWebView` with `WKWebView`.
- **Responding to new platform requirements**, such as adopting the scene-based lifecycle, supporting new screen sizes, or handling new privacy permissions.
- **Improving the code for Swift callers** by adding nullability annotations, generics, and better names.
- **Finding and fixing memory problems**, such as retain cycles involving delegates, blocks, or timers.

- **Migrating modules to Swift** when the business case and the test coverage justify it.

Each of these tasks requires reading before writing. You need to understand the flow of control, the ownership of objects, and the conventions the original developers followed. Objective-C codebases often rely on conventions that the compiler does not enforce, such as method naming rules that indicate ownership in manual memory management code, or notification names that tie distant parts of the app together. Part IV of this book focuses on exactly this kind of detective work.

An example of a small, safe fix

Here is a realistic example. A crash report shows that Fieldbook sometimes crashes when a researcher exports a project that contains an observation with no notes. The export code looks like this:

Fieldbook/Export/FBCSVExporter.m (excerpt, before)

```
- (NSString *)rowForObservation:(FBObservation *)observation {
    NSMutableArray *fields = [NSMutableArray array];
    [fields addObject:observation.title];
    [fields addObject:observation.notes];
    [fields addObject:[self.dateFormatter stringFromDate:observation.timestamp]];
    return [fields componentsJoinedByString:@","];
}
```

In Objective-C, adding `nil` to an `NSMutableArray` raises an exception, which terminates the app if it is not caught. When `notes` is `nil`, the second `addObject:` call crashes. A minimal fix substitutes an empty string:

Fieldbook/Export/FBCSVExporter.m (excerpt, after)

```
- (NSString *)rowForObservation:(FBObservation *)observation {
    NSMutableArray<NSString *> *fields = [NSMutableArray array];
    [fields addObject:observation.title];
    [fields addObject:observation.notes ?: @""];
    [fields addObject:[self.dateFormatter stringFromDate:observation.timestamp]];
    return [fields componentsJoinedByString:@","];
}
```

The `?:` operator, a widely supported C extension available in Clang, returns the left side if it is non-nil and otherwise the right side, much like Swift's `??`. The fix is one line, but finding it required knowing three Objective-C facts: that `nil` cannot be stored in Foundation collections, that this rule is enforced at run time rather than compile time, and that the `notes` property can legitimately be `nil`. Notice also that this is not a complete CSV implementation; real CSV export must quote fields that contain commas, quotes, or line breaks. The Fieldbook team will address that properly when they write characterization tests for the exporter in Chapter 18.

> **WARNING**
>
> Do not "fix" crashes like this by wrapping code in `@try` and `@catch`. In Objective-C, exceptions signal programmer errors, not recoverable conditions, and much of the framework code is not designed to continue safely after an exception passes through it. Chapter 8 explains the difference between exceptions and `NSError` in detail.

Objective-C and Swift Side by Side

Because many readers come from Swift, it is useful to set expectations about how the two languages differ at a high level. The table below is not exhaustive, and each row is explained properly in later chapters.

Concept	Objective-C	Swift
Files per class	Header (.h) and implementation (.m)	One .swift file, no headers
Calling a method	`[list reloadData]`	`list.reloadData()`
Missing values	`nil` pointers, any object type	Optionals, checked by the compiler
Messaging nil	Allowed; returns zero or nil	Requires unwrapping or optional chaining
Memory	ARC (manual retain/release in old code)	ARC
Value types	C structs and primitives	Structs and enums with rich features
Generics	Lightweight, for collections and classes	Full generics
Errors	`NSError **` out-parameters	`throws` and `try`
Closures	Blocks: `^{ ... }`	Closures: `{ ... }`
Dispatch	Dynamic messaging by default	Static or vtable by default
Namespacing	Class prefixes such as `FB`	Modules

One row deserves special mention because it surprises almost every Swift developer: in Objective-C, sending a message to `nil` is perfectly legal. It does nothing and returns zero, `NO`, or `nil` depending on the return type. That rule makes some code wonderfully concise and some bugs wonderfully hard to find:

Messaging nil

```
FBProject *project = nil;
NSString *name = [project name];          // name is nil, no crash
NSUInteger count = [project.observations count]; // count is 0
BOOL archived = [project isArchived];      // archived is NO

if (count == 0) {
```

```
    // Was the project empty, or was there no project at all?
    // The code cannot tell the difference.
}
```

The silence of nil messaging is one reason why legacy Objective-C bugs often appear far from their cause. A property that should never be `nil` quietly becomes `nil`, and nothing fails until much later, when the empty value finally reaches a collection or a C API that cannot accept it. Nullability annotations, assertions, and tests, all covered later in this book, are the main tools for pushing such failures back toward their origin.

How Migration Fits In

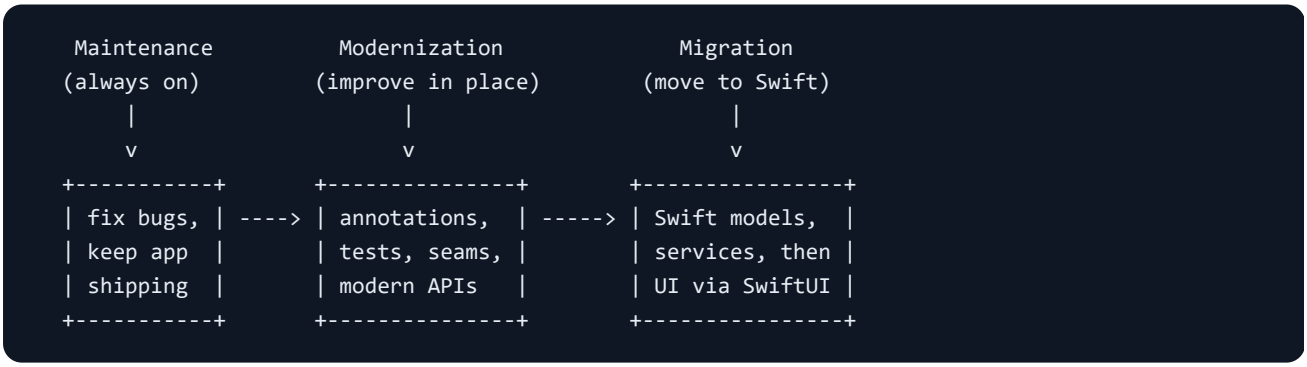
The subtitle of this book promises maintenance, modernization, and migration. These are three different activities, and keeping them distinct will help you make better decisions.

Maintenance keeps the existing app working: fixing bugs, responding to platform changes, and making small improvements. It is ongoing and unavoidable.

Modernization improves the existing Objective-C code without changing its language: adopting ARC where old files still use manual memory management, adding nullability and generics, replacing deprecated APIs, enabling stricter compiler warnings, breaking up massive classes, and adding tests. Modernization usually delivers immediate benefits, and it also makes later migration much easier, because well-annotated, well-tested Objective-C code bridges cleanly into Swift.

Migration moves code from Objective-C to Swift. Done well, it happens incrementally, one class or one feature at a time, with the app remaining shippable at every step. Done badly, it becomes a long-running rewrite that competes with feature work and introduces regressions.

A recurring theme of this book is that you should modernize before you migrate. Converting a poorly understood, untested Objective-C class directly to Swift often reproduces its bugs in a new language while adding new ones from the translation. Adding tests and annotations first gives you a safety net and a clear specification of the behavior you must preserve.



Chapter 23 presents a full strategy for planning a migration: choosing an order, managing risk, and deciding which parts of the code may reasonably stay in Objective-C indefinitely. Not every line needs to move. A stable, well-tested Objective-C module with a clean interface can live alongside Swift code

for a very long time.

> **NOTE**
>
> A full rewrite is sometimes the right answer, for example when the product itself is changing so much that little of the old behavior matters. This book does not argue that rewrites are always wrong. It argues that the decision should be made deliberately, with an honest view of the risk, rather than by default.

How This Book Is Organized

The book has six parts. **Part I** (Chapters 1 to 5) teaches the language from the ground up: toolchain, syntax, memory management, and Foundation. **Part II** (Chapters 6 to 10) goes deeper into protocols, categories, blocks, error handling, the runtime, and communication patterns. **Part III** (Chapters 11 to 15) covers UIKit and AppKit apps, networking, and persistence. **Part IV** (Chapters 16 to 20) focuses on legacy work: reading, modernizing, testing, and debugging. **Part V** (Chapters 21 to 25) covers interoperability and gradual migration to Swift, and **Part VI** (Chapters 26 to 28) brings everything together in Fieldbook case studies. The appendices provide a cheat sheet, a translation reference, a checklist, and a glossary.

What You Need to Follow Along

To build and run the examples, you need a Mac with a recent version of Xcode installed. We deliberately avoid quoting specific Xcode, macOS, or iOS version numbers, because they change frequently and the material in this book does not depend on any particular release. Where an API's availability matters, we say so and recommend that you check the current documentation.

You also need some general programming experience. We assume you know what variables, functions, loops, and classes are. You do not need to know C, but some familiarity helps, because Objective-C is built on it. Chapter 3 introduces the C features that matter most in day-to-day Objective-C code.

Many examples in early chapters can be run as small command-line programs, which keeps them free of user-interface boilerplate. Here is the kind of program you will build in Chapter 2:

`FieldbookCLI/main.m`

```
#import <Foundation/Foundation.h>
#import "FBObservation.h"

int main(int argc, const char *argv[]) {
    @autoreleasepool {
        FBObservation *observation =
            [[FBObservation alloc] initWithTitle:@"Heron at north pond"];
        observation.notes = @"Fishing near reeds, approx. 40 m from path.";
        NSLog(@"%@", [observation summaryLine]);
```