BUILDING macOS APPLICATIONS WITH SWIFT

FROM FIRST PROGRAM TO PRODUCTION APPLICATION

LEARN SWIFT
Step by step

BUILD MAC APPS
With real-world projects

FOLLOW BEST PRACTICES
Write clean, maintainable, production-ready code

DEPLOY WITH CONFIDENCE
Distribute on the Mac App Store or directly to users

SCOTT HANSMANN

Building macOS Applications with Swift

From First Program to Production Application

Steve Publications

This book is available at
https://leanpub.com/buildingmacosapplicationswithswift

This version was published on 2026-07-25

Contents

From First Program to Production Application

Swift is the modern programming language Apple created to replace Objective-C as the primary language for building applications on its platforms. This book takes you from writing your first line of Swift code to shipping professional-grade macOS applications. You will learn not only Swift syntax and features but also the design philosophy behind the language, the macOS ecosystem it serves, and the engineering practices that separate working prototypes from production software. Every concept is explained with real-world context, practical examples, and complete source code you can study and adapt. By the end, you will have the knowledge and confidence to build anything from simple utilities to sophisticated desktop applications distributed through the Mac App Store or directly to users worldwide.

Chapter 1: The Swift Ecosystem and macOS Development Landscape

The Story of Swift: From C++ Roots to Modern Language

In the summer of 2014, Apple announced something that would fundamentally reshape software development on its platforms. At the Worldwide Developers Conference, the company unveiled Swift, a new programming language designed to replace Objective-C as the primary language for building iOS, macOS, watchOS, and tvOS applications. The announcement stunned the developer community not because creating a new language was unprecedented, but because Apple had kept the project secret from virtually everyone outside the company. Prior to the WWDC reveal, only about 250 internal employees knew Swift existed [1].

The story of Swift began four years earlier, in 2010, when Chris Lattner, a senior engineer at Apple who had previously created LLVM (a compiler infrastructure project that would become foundational to the open-source compiler ecosystem), started designing the language as a solo project. By late 2011, a small team of engineers had joined him: Doug Gregor, John McCall, Ted Kremenek, and Joe Groff. Together, they worked to create a language that addressed the shortcomings of Objective-C while remaining compatible with the existing Apple ecosystem [2].

Objective-C, the language Swift was designed to replace, had served Apple well for decades. It was an extension of C that added Smalltalk-style messaging and dynamic runtime features. However, by 2010, it showed its age. The syntax was verbose, the type system was weak, and the language lacked many modern programming constructs. Memory management through manual retain/release calls was error-prone, and the dynamic nature of the language made large codebases difficult to maintain and refactor safely.

Swift was designed with three core goals in mind: safety, performance, and expressiveness. Lattner drew inspiration from a wide range of languages including Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, and others [3]. The

result was a language that combined the low-level control and performance of compiled systems programming with high-level abstractions that made code easier to write, read, and maintain.

In December 2015, Apple open-sourced Swift under the Apache 2.0 license with a Runtime Library Exception, making it available to developers beyond the Apple ecosystem [4]. This decision was significant for several reasons. It allowed the broader developer community to contribute to the language's evolution, enabled Swift to be used on Linux and other platforms, and signaled Apple's long-term commitment to the language. Today, Swift is maintained by the open-source community through the Swift Evolution process, with proposals reviewed and approved by a technical oversight committee that includes representatives from Apple and the wider Swift community.

Swift has progressed rapidly since its initial release. Each major version has introduced new features and improvements: Swift 2 added error handling and protocol-oriented programming capabilities; Swift 3 brought a more consistent API design; Swift 4 improved interoperability with Objective-C and enhanced the standard library; Swift 5 delivered ABI stability, meaning binaries compiled with Swift 5 would continue to work on future versions without recompilation; Swift 5.5 introduced modern concurrency with async/await and actors; and Swift 6, released in September 2024, added strict concurrency checking that diagnoses potential data races at compile time [5].

Chris Lattner left Apple in January 2017 to join Tesla, where he led the Autopilot software team. His departure did not slow Swift's progress. The language continues to evolve through the collaborative efforts of the open-source community and Apple's ongoing investment in its development.

Design Philosophy: Safety, Performance, and Expressiveness

Swift is described on its official website as a "safe, fast, and expressive general-purpose programming language" [6]. These three words capture the essence of Swift's design philosophy. Understanding them helps explain many of the language's specific features and why they work the way they do.

Safety means that Swift eliminates entire classes of bugs that were common in C-based languages. Variables must be initialized before use, preventing the subtle errors caused by reading uninitialized memory. Arrays and integers

are checked for overflow, catching boundary errors at runtime rather than allowing silent corruption. Memory is managed automatically through Automatic Reference Counting (ARC), removing the need for manual memory management while still providing performance comparable to manual approaches. Most importantly, Swift's optional type system forces developers to explicitly handle the absence of values, eliminating the notorious "nil pointer exception" that has caused countless crashes in other languages [7].

Performance is not sacrificed for safety. Swift is compiled to native machine code using the LLVM compiler infrastructure, the same technology used to compile C and C++ programs. This means Swift programs run at speeds comparable to those written in lower-level languages. The language provides fine-grained control over performance-critical code when needed, while also offering high-level abstractions that make everyday programming easier. Value types, such as structs and enums, are passed by copy and can be optimized aggressively by the compiler, often resulting in better performance than reference types [8].

Expressiveness refers to how clearly and concisely Swift allows developers to communicate their intent. The language includes features like type inference, which lets the compiler determine variable types from context so you write less boilerplate. Closures provide a concise way to define anonymous functions inline. Pattern matching with switch statements is powerful and exhaustive, ensuring all cases are handled. Protocols and extensions enable flexible code organization without deep inheritance hierarchies. These features combine to make Swift code readable and maintainable, reducing the gap between what a developer intends and what the code actually says [9].

Beyond these three pillars, Swift embraces several additional design principles:

- **Progressive disclosure of complexity**: Simple things should be simple, and complex things should be possible. Beginners can write useful programs using basic features, while experts have access to advanced capabilities when needed.
- **One way to do things**: Wherever possible, Swift provides a single idiomatic approach to common tasks, reducing confusion and making code more consistent across projects.
- **Compatibility with existing ecosystems**: Swift interoperates seamlessly with C, C++, and Objective-C, allowing gradual migration of existing codebases and access to vast libraries of existing code.

These principles are not merely aspirational; they guide the Swift Evolution process, through which all new language features are proposed, debated, and refined before being added to the language.

macOS as a Development Platform

macOS is more than just an operating system; it is a comprehensive development platform that provides the tools, frameworks, and runtime environment needed to build professional applications. Understanding macOS and its relationship to Swift is essential for becoming an effective developer on Apple's desktop platform.

At its core, macOS is built on top of Darwin, an open-source Unix-like operating system that combines the Mach kernel with components from BSD (Berkeley Software Distribution). This foundation gives macOS strong POSIX compliance, a robust file system, and powerful command-line tools. For developers, this means many concepts and tools from the Unix world apply directly to macOS development.

macOS provides several key frameworks that Swift developers use extensively:

- **AppKit**: The traditional framework for building macOS user interfaces, providing windows, views, controls, menus, and document management. AppKit has been part of macOS since its early days and remains fully supported and actively maintained.
- **SwiftUI**: Apple's modern declarative UI framework, introduced in 2019 and now mature enough for production use on all Apple platforms including macOS. SwiftUI uses a declarative syntax that describes what the interface should look like rather than how to construct it imperatively.
- **Foundation**: The core framework providing fundamental data types, collection classes, file and URL handling, networking, and other essential utilities. Foundation is implemented in Swift and Objective-C and serves as the backbone for most Swift applications.
- **Core Data**: An object graph and persistence framework for managing model layer objects in applications. Core Data handles saving and loading data, undo management, and complex queries.
- **SwiftData**: A newer persistence framework built on top of Core Data that uses modern Swift features like macros to simplify data modeling and querying. Available on macOS 14 and later [10].

macOS also provides a rich ecosystem of system services that applications can integrate with, including notifications, printing, accessibility support, drag-and-drop operations, and integration with other applications through services and automation.

One important consideration for macOS developers is the platform's security architecture. Modern macOS versions enforce Gatekeeper, which checks applications for valid code signatures and notarization before allowing them to run. Applications distributed through the Mac App Store must run in a sandbox that restricts their access to system resources and user data. Even applications distributed outside the App Store are increasingly expected to adopt security best practices including code signing, notarization, and the hardened runtime [11].

macOS also supports universal binaries, allowing applications to run natively on both Intel-based Macs and Apple silicon (M-series) Macs. Since the transition to Apple silicon began in 2020, all new Swift and macOS development should target universal builds to ensure compatibility across the Mac ecosystem.

Xcode: Your Primary Development Environment

Xcode is Apple's integrated development environment (IDE) for building applications on all Apple platforms. It combines a code editor, interface designer, debugger, profiler, and distribution tools into a single application. While it is possible to develop Swift applications using alternative editors and command-line tools, Xcode provides the most complete and integrated experience for macOS development.

The current version of Xcode (as of mid-2026) is Xcode 26, which includes Swift 6.2 and SDKs for the latest versions of all Apple operating systems [12]. Xcode is available free of charge from the Mac App Store, though distributing applications requires enrollment in the Apple Developer Program, which costs $99 per year.

Xcode's key features include:

- **Code editor**: A powerful editor with syntax highlighting, code completion, refactoring tools, and inline documentation. The editor understands Swift's type system and can suggest completions based on context.

- **Interface Builder**: A visual design tool for creating user interfaces using storyboards and xib files, primarily used with AppKit. SwiftUI provides an alternative approach with live previews that render your declarative UI directly in the editor.
- **Build system**: Xcode uses the Swift build system (swift build) under the hood, providing fast incremental builds and parallel compilation.
- **Debugger**: Integration with LLDB, the LLVM-based debugger, allowing you to set breakpoints, inspect variables, and step through code execution.
- **Instruments**: A suite of profiling tools for analyzing performance, memory usage, energy consumption, and other runtime characteristics of your application.
- **Test runner**: Built-in support for running unit tests, UI tests, and performance tests using XCTest and Swift Testing frameworks.
- **Distribution tools**: Integrated workflows for archiving applications, generating distribution packages, and submitting to the Mac App Store.

Xcode projects organize code, resources, and build settings into a structured format. A typical macOS application project contains source files, asset catalogs for images and other resources, entitlements files for sandbox permissions, and configuration files that specify build targets and deployment settings.

While Xcode is the standard tool for Apple development, Swift can also be developed using alternative editors such as Visual Studio Code with the Swift Language Support extension, or command-line tools including the Swift compiler (swift), the build tool (swift build), and the package manager (swift package). These alternatives are useful for server-side Swift development, continuous integration pipelines, and developers who prefer non-Apple IDEs.

Swift Toolchain Beyond Xcode

The Swift toolchain consists of several components that work together to compile, run, and manage Swift code. Understanding these tools is valuable even if you primarily use Xcode, as they provide insight into what happens behind the scenes and enable development workflows outside the IDE.

The core components include:

- **swift**: The Swift compiler and interpreter. It can compile Swift source files to executables or libraries, run Swift scripts directly, and provide a REPL (read-eval-print loop) for interactive exploration of the language.
- **swiftc**: A lower-level interface to the Swift compiler that provides more fine-grained control over compilation options, useful for custom build systems and advanced use cases.
- **swift build**: The build tool used by Swift Package Manager to compile packages and their dependencies. It handles dependency resolution, incremental builds, and linking.
- **swift package**: The command-line interface to Swift Package Manager, used to create new packages, add dependencies, resolve versions, and manage package metadata.
- **swift test**: Runs tests defined in a Swift package using the XCTest or Swift Testing frameworks.
- **lldb**: The LLVM-based debugger that provides source-level debugging for Swift, C, C++, and Objective-C programs.

Swift can be installed on macOS either through Xcode (which includes the full toolchain) or through the command-line tools package, which is a smaller installation suitable for developers who do not need the full IDE. On Linux and Windows, Swift can be installed directly from swift.org using pre-built binaries or package managers [13].

The Swift REPL provides an interactive environment for experimenting with the language. You can launch it by running `swift` in the terminal without any arguments, then typing Swift code directly and seeing immediate results. This is particularly useful for learning the language, testing small snippets of code, and exploring APIs before incorporating them into larger projects.

Swift Playgrounds, while primarily designed as an educational tool on iPad and Mac, also serves as a lightweight environment for experimenting with Swift code without the overhead of a full Xcode project. For quick prototyping and learning exercises, Playgrounds can be faster and more convenient than creating complete application projects.

Installing and Configuring Your Development Environment

Before writing Swift code, you need to set up your development environment. The process is straightforward for macOS development, as Apple provides all necessary tools through official channels.

First, ensure your Mac meets the system requirements for the latest version of Xcode. As of 2026, Xcode requires macOS 15 (Sequoia) or later and runs natively on both Intel and Apple silicon Macs. Check your system version by clicking the Apple menu and selecting About This Mac.

Download Xcode from the Mac App Store. Search for "Xcode" and click Get to begin the download. The installation is substantial, typically over 40 gigabytes, so ensure you have adequate disk space and a stable internet connection. Alternatively, you can download specific versions of Xcode from Apple's developer website if you need an older version for compatibility reasons [14].

After installation, launch Xcode for the first time. You will be prompted to install additional components, including command-line tools and platform SDKs. Accept these installations to ensure you have the complete development environment.

Next, configure your preferences:

- Open Xcode Preferences (Xcode > Settings or Command+,).
- In the Accounts tab, sign in with your Apple ID. This is required for distributing applications but not for development and testing on your local machine.
- In the Locations tab, verify that the correct command-line tools are selected.
- In the Text Editing tab, configure code formatting preferences such as indentation style and tab width. Swift's standard convention uses four-space indentation with no tabs.

If you plan to distribute applications, enroll in the Apple Developer Program through developer.apple.com. The $99 annual membership provides access to distribution certificates, provisioning profiles, TestFlight beta testing, and the ability to submit applications to the Mac App Store.

For command-line development without the full Xcode IDE, install the Command Line Tools by running `xcode-select --install` in Terminal. This installs the Swift compiler, build tools, and essential libraries with a much smaller footprint than the complete Xcode installation.

With your environment configured, you are ready to begin writing Swift code. The next chapter will guide you through creating and running your first Swift programs, introducing the fundamental syntax and concepts that form the foundation of everything that follows.

Chapter 2: Your First Swift Programs

Running Your First Swift Program

The fastest way to experience Swift is to write and run a small program. There are several approaches depending on your setup and goals. This section walks through each method so you can choose what works best for your workflow.

Using Xcode Playground: Playgrounds provide an interactive environment where you can write Swift code and see results immediately without creating a full project. Open Xcode, select File > New > Playground, and choose macOS as the platform. Name your playground "HelloSwift" and save it. Delete the default code and type:

```
import Foundation

print("Hello, world!")
```

The right side of the playground window displays output in real time. You will see "Hello, world!" appear almost instantly after typing. Playgrounds are excellent for experimenting with language features, testing small code snippets, and learning at your own pace. They execute code line by line and show the result of each expression, making them ideal for exploration.

Using a Command Line Script: For quick experiments outside Xcode, you can write Swift scripts directly in the terminal. Create a file named `hello.swift` using any text editor:

```
#!/usr/bin/env swift
import Foundation

print("Hello from the command line!")
```

Make the file executable with `chmod +x hello.swift`, then run it with `./hello.swift`. The shebang line at the top tells the system to use the Swift

interpreter to execute the file. This approach is useful for automation scripts, quick utilities, and learning Swift without opening Xcode.

Using a Swift Package: For more structured projects, Swift Package Manager provides a clean way to organize code. Create a new package by running:

```
swift package init --type executable
```

This generates a standard project structure with a `Package.swift` manifest file and a `Sources/Hello/Swift.swift` source file. Run the program with `swift run`. The output shows "Hello, world!" from the default template. Packages scale from simple scripts to complex applications and are the recommended approach for any project beyond experimentation.

Using Xcode Project: For full macOS applications, create an Xcode project. Select File > New > Project, choose macOS > App, and ensure the interface is set to SwiftUI or AppKit depending on your preference. This creates a complete application skeleton with build settings, entitlements, and a main entry point. While this is the most involved approach, it is necessary for building distributable applications.

For the remainder of this chapter, we will focus on fundamental Swift concepts using playground and script examples. These concepts apply equally to all project types.

Variables and Constants: let and var

In Swift, you store values in variables and constants. The distinction between them is fundamental to Swift's design philosophy of safety and clarity.

Use `let` to declare a constant, a value that cannot be changed after it is set:

```
let appName = "My Application"
let maxRetries = 3
let pi = 3.14159
```

Once you assign a value to a constant, any attempt to modify it results in a compile-time error. This is not merely a convention; the compiler enforces

immutability. Using constants wherever possible makes your code safer because it eliminates an entire class of bugs caused by unintended modification of values that should remain fixed.

Use `var` to declare a variable, a value that can be changed:

```
var currentUserCount = 0
var isLoading = false
var selectedTabIndex = 0
```

Variables are necessary when you need to track changing state, such as counters, user input, or data that updates over time. However, Swift encourages you to default to constants and only use variables when mutation is genuinely required. This practice makes code easier to reason about because you can trust that a `let` value will not change unexpectedly.

A key principle in Swift: declare values as constants unless you have a specific reason they must be mutable. When reading code, seeing `let` immediately tells you the value is stable, reducing cognitive load. This philosophy extends throughout the language and influences how you design functions, data structures, and entire applications.

Swift also enforces that all variables and constants must be initialized before use. Unlike some languages that allow uninitialized variables with undefined or default values, Swift requires you to provide an initial value:

```
var count: Int // Error: missing initial value for variable 'count'
var count = 0  // Correct: initialized to zero
```

This rule eliminates bugs caused by reading uninitialized memory and forces you to think explicitly about what initial state makes sense for each value.

Type System Basics: Int, Double, String, Bool

Swift is a statically typed language, meaning every value has a specific type known at compile time. Types define what operations are valid on a value and help catch errors before your program runs. Swift's type system is powerful yet approachable, providing safety without excessive verbosity.

The basic types you will use most frequently are:

Int: Whole numbers, both positive and negative. Swift's `Int` type automatically matches the native word size of the platform (64-bit on modern macOS), so you do not need to choose between Int32 and Int64 in most cases.

```
let numberOfUsers = 150
let temperatureDelta = -5
let zeroValue = 0
```

Double: Floating-point numbers with decimal precision. Use Double for values that require fractional parts:

```
let price = 29.99
let ratio = 0.75
let scientificNotation = 1.5e2 // Equals 150.0
```

Swift also provides `Float` (32-bit) for cases where memory is constrained, but Double is the default choice for most applications due to its greater precision.

String: Sequences of characters representing text. Swift strings are Unicode-aware and handle international text correctly:

```
let greeting = "Hello, world!"
let name = "Alex"
let emptyString = ""
```

Strings can be created using double quotes and support a variety of operations including concatenation, interpolation, and searching.

Bool: Boolean values representing true or false. Booleans are essential for control flow and conditional logic:

```
let isEnabled = true
let hasErrors = false
let isEmpty = [] == [] // Evaluates to true
```

Swift does not allow implicit conversion between types. You cannot use an integer where a boolean is expected, or treat a string as a number without explicit conversion. This strictness prevents subtle bugs:

```
let count = 5
// if count { } // Error: cannot convert value of type 'Int' to expected
↪  condition type 'Bool'
if count > 0 { } // Correct: comparison produces a Bool
```

Swift provides additional numeric types for specific needs: `UInt` for unsigned integers, `Int8`, `Int16`, `Int32`, and `Int64` for fixed-size integers, and `Float80` for extended precision on some platforms. For most applications, however, `Int` and `Double` cover the vast majority of numeric needs.

Type Inference and Explicit Annotations

Swift's type inference is one of its most convenient features. When you initialize a variable or constant with a value, the compiler automatically determines the appropriate type:

```
let name = "Alex"        // Compiler infers String
let count = 42           // Compiler infers Int
let price = 19.99        // Compiler infers Double
let isActive = true      // Compiler infers Bool
```

Type inference works because Swift is statically typed; the compiler still knows the exact type of every value. It simply saves you from writing the type explicitly when it can be determined from context. This makes code more concise without sacrificing type safety.

You can also specify types explicitly using a colon followed by the type name:

```
let name: String = "Alex"
let count: Int = 42
let price: Double = 19.99
```

Explicit annotations are useful in several situations:

- When the initial value does not clearly indicate the intended type:

```swift
var temperature: Double = 0 // Without annotation, compiler might infer Int
```

- When declaring a variable before assigning its value:

```swift
var result: String?
// ... later ...
result = "Success"
```

- When you want to document your intent for readers of the code:

```swift
let userId: Int32 = 12345 // Documents that this must be Int32 specifically
```

- In function signatures and API definitions, where types are always explicit.

The general guideline is to rely on type inference for local variables and constants where the type is obvious, and use explicit annotations when clarity, documentation, or specific type requirements demand it. Over-annotating code with obvious types adds noise without benefit; under-annotating when types are ambiguous creates confusion.

Basic Operators and Expressions

Swift provides a comprehensive set of operators for performing calculations, comparisons, and logical operations. Most operators behave as you would expect from other programming languages, but Swift adds some conveniences and safety features.

Arithmetic operators work on numeric types:

```swift
let sum = 10 + 5          // 15
let difference = 10 - 5   // 5
let product = 10 * 5      // 50
let quotient = 10 / 5     // 2 (integer division)
let remainder = 10 % 3    // 1

let doubleDivision = 10.0 / 3.0 // 3.333...
```

Note that integer division truncates the result toward zero. If you need a fractional result, ensure at least one operand is a floating-point type.

Comparison operators produce boolean values:

```swift
let isEqual = 5 == 5           // true
let isNotEqual = 5 != 3        // true
let isGreater = 10 > 5         // true
let isLess = 3 < 7             // true
let isGreaterOrEqual = 5 >= 5  // true
let isLessOrEqual = 2 <= 4     // true
```

Logical operators combine boolean values:

```swift
let andResult = true && false    // false (both must be true)
let orResult = true || false     // true (at least one must be true)
let notResult = !true            // false (inverts the value)
```

Swift uses short-circuit evaluation for && and ||, meaning the right side is only evaluated if the left side does not determine the result. This is important when the right side has side effects or might cause errors:

```swift
let count = 0
if count > 0 && (10 / count) > 2 {
    // The division never executes because count > 0 is false
}
```

Compound assignment operators combine an operation with assignment:

```swift
var score = 10
score += 5   // Equivalent to score = score + 5, result is 15
score -= 3   // Result is 12
score *= 2   // Result is 24
score /= 4   // Result is 6
```

Ternary conditional operator provides a concise way to write simple conditionals:

```swift
let age = 20
let status = age >= 18 ? "adult" : "minor" // "adult"
```

This is equivalent to writing an if-else statement but more compact for straightforward cases. Use it sparingly; complex nested ternary expressions become difficult to read.

Swift does not allow implicit type mixing in operations. You cannot add an Int and a Double without explicit conversion:

```swift
let intVal = 5
let doubleVal = 2.5
// let result = intVal + doubleVal // Error: binary operator '+' cannot be
↪  applied
let result = Double(intVal) + doubleVal // Correct: 7.5
```

This strictness prevents accidental loss of precision and makes type conversions explicit and visible.

Strings and Character Handling

Strings in Swift are value types, meaning they are copied when passed around rather than shared by reference. This design choice has important implications for safety and performance that we will explore later. For now, focus on how to work with strings effectively.

Creating strings uses double quotes:

```swift
let message = "Hello, world!"
let emptyString1 = ""
let emptyString2 = String()
```

Both `""` and `String()` create empty strings; the literal syntax is more common.

String concatenation combines strings using the `+` operator or string interpolation:

```swift
let firstName = "Alex"
let lastName = "Johnson"
let fullName = firstName + " " + lastName

// String interpolation is cleaner and more readable
let greeting = "Hello, \(firstName) \(lastName)!"
let calculation = "The sum of 2 and 3 is \(2 + 3)"
```

String interpolation, using `\()` syntax, allows you to embed expressions directly within string literals. The expression inside the parentheses is evaluated and its result is converted to a string. This works with any type that can be described as text.

Multiline strings use triple quotes:

```swift
let poem = """
Roses are red,
Violets are blue,
Swift is modern,
And so am I.
"""
```

Multiline string literals preserve formatting, including indentation and line breaks. They are particularly useful for templates, error messages, and any text that spans multiple lines.

String operations include checking length, searching, and transforming:

```swift
let text = "Hello, Swift!"
print(text.count)                    // 13 (number of characters)
print(text.isEmpty)                  // false
print(text.hasPrefix("Hello"))       // true
print(text.hasSuffix("Swift!"))      // true
print(text.uppercased())             // "HELLO, SWIFT!"
print(text.lowercased())             // "hello, swift!"
print(text.trimmingCharacters(in: .whitespaces)) // Removes leading/trailing
↪  spaces
```

Substring vs String: When you extract part of a string, Swift returns a `Substring`, not a `String`. Substrings share storage with the original string for efficiency. If you need to store the result long-term, convert it to a String:

```swift
let text = "Hello, world!"
let sub = text[..<5]           // Substring: "Hello"
let str = String(sub)          // String: "Hello"
```

This distinction prevents accidental retention of large strings when you only need a small portion. The compiler reminds you to convert to String when storing substrings in properties.

Unicode support: Swift strings are fully Unicode-aware. They correctly handle characters from any writing system, including emoji and combining characters:

```swift
let greeting = "Hello 🌍"
let accented = "café"
print(greeting.count) // 7 (each emoji counts as one character)
```

Characters in Swift are extended grapheme clusters, which means a visually single character may consist of multiple Unicode code points. Swift handles this complexity transparently, so your string operations behave intuitively regardless of the text content.

Comments and Code Documentation

Comments are notes in your code that the compiler ignores. They help explain your thinking, document complex logic, and communicate with other developers reading your code.

Single-line comments start with `//`:

```swift
let maxAttempts = 3 // Maximum number of retry attempts allowed
```

Everything after `//` on that line is treated as a comment.

Multi-line comments use `/* */`:

```swift
/*
 This function calculates the total price including tax.
 It handles edge cases for negative quantities and zero prices.
 */
func calculateTotalPrice(_ quantity: Int, pricePerUnit: Double, taxRate:
↪  Double) -> Double {
    // Implementation here
    return 0.0
}
```

Multi-line comments can span multiple lines and even nest within each other, making them useful for temporarily disabling blocks of code during development.

Documentation comments use `///` or `/** */` and generate formatted documentation that Xcode displays in code completion and the documentation viewer:

```swift
/// Calculates the total price including tax.
///
/// - Parameters:
///   - quantity: Number of units to purchase. Must be non-negative.
///   - pricePerUnit: Price for a single unit.
///   - taxRate: Tax rate as a decimal (e.g., 0.08 for 8 percent).
/// - Returns: Total price including tax, or zero if quantity is negative.
func calculateTotalPrice(_ quantity: Int, pricePerUnit: Double, taxRate:
↪  Double) -> Double {
    guard quantity >= 0 else { return 0.0 }
    let subtotal = Double(quantity) * pricePerUnit
    return subtotal * (1.0 + taxRate)
}
```

Documentation comments follow a structured format that Xcode recognizes. The summary line describes what the function does, parameters are documented with their names and meanings, and the return value is explained. This documentation appears when you type the function name in code, helping developers understand how to use it correctly.

Good commenting practice:

- Explain why, not what. The code shows what it does; comments should explain non-obvious reasoning or constraints.
- Keep comments close to the code they describe.
- Update comments when code changes. Outdated comments are worse than no comments.
- Use documentation comments for public APIs and complex logic that others will need to understand.

With these fundamentals in place, you have the basic building blocks for writing Swift programs. The next chapter explores how to control program flow, make decisions, and repeat operations using Swift's control flow constructs.

Chapter 3: Control Flow and Decision Making

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Conditional Statements with if and guard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Switch Statements and Exhaustive Pattern Matching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Loop Constructs: for-in, while, repeat-while

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Range Operators and Sequence Iteration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Break, Continue, and Labelled Statements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Early Exit Patterns with guard

Chapter 4: Functions and Closures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Defining and Calling Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Parameter Labels and Argument Names

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Return Types and Multiple Returns with Tuples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Default Parameters and Variadic Arguments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Closures: Syntax and Capture Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Higher-Order Functions: map, filter, reduce

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Function Types as First-Class Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 5: Collections and Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Arrays: Ordered Collections and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Sets: Unique Elements and Set Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Dictionaries: Key-Value Storage and Lookup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Ranges and Partial Ranges

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Collection Protocols and Conformance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Common Collection Patterns and Algorithms

Chapter 6: Structs, Classes, and Object-Oriented Programming

Value Types vs Reference Types: The Fundamental Distinction

Defining Structs for Data Modeling

Classes and Reference Semantics

Properties: Stored, Computed, and Property Observers

Methods and Self

Initializers and Deinitializers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Inheritance and Class Hierarchies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Type Casting and the is/as Operators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 7: Protocol-Oriented Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

What Are Protocols and Why They Matter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Defining and Conforming to Protocols

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Protocol Extensions with Default Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Protocol Composition and Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Associated Types and Generic Protocols

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Choosing Between Inheritance and Protocols

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Real-World Protocol Design Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 8: Optionals, Error Handling, and Defensive Programming

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Understanding Nil and the Optional Type

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Unwrapping Optionals Safely

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Optional Chaining and the Nil-Coalescing Operator

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Defining Custom Error Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Throwing and Catching Errors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

do-catch and Propagating Failures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Defensive Programming Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 9: Memory Management in Swift

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

How Automatic Reference Counting Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Strong References and Retain Counts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Weak and Unowned References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Strong Reference Cycles and How to Break Them

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Memory Management with Closures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Class vs Struct Implications for Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Diagnosing Memory Issues with Instruments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 10: Generics and Advanced Type Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

What Are Generics and Why They Exist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Generic Functions and Type Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Generic Types: Structs, Classes, and Enums

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Type Constraints and Protocol Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Associated Types in Generic Contexts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Where Clauses for Complex Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Real-World Generic Design Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 11: Concurrency in Swift

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

The Problems with Traditional Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Async Functions and Awaiting Results

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Tasks and Structured Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Actors and Isolated State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Task Groups for Concurrent Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Serial and Concurrent Actors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Interacting with Legacy Completion Handlers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 12: SwiftUI for macOS Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Declarative UI and the SwiftUI Philosophy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Views, Modifiers, and Composition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

State Management: @State, @Binding, @ObservedObject

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Layouts and Geometry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Lists, Forms, and Common Controls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Navigation and Window Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Building a Complete SwiftUI Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 13: AppKit and Traditional macOS Development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

AppKit Architecture and the Responder Chain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Windows, Views, and Controllers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Menus, Toolbars, and Status Bars

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Document-Based Applications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Table Views and Advanced Controls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Mixing SwiftUI and AppKit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

When to Choose AppKit Over SwiftUI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 14: Swift Package Manager and Project Organization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Understanding Swift Packages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Package.swift Manifest Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Targets, Dependencies, and Products

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Creating and Publishing Your Own Packages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Module Design Principles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Organizing Large Projects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 15: Networking and Web Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

URLSession and Network Requests

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Async/Await with URLSession

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Working with JSON and Codable

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

HTTP Methods and Headers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Authentication and API Keys

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Error Handling for Network Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Building a Complete API Client

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 16: Data Persistence and File Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

The macOS File System and Directories

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Reading and Writing Files

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

UserDefaults for Simple Settings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Property Lists and Binary Archives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Core Data for Complex Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Choosing a Persistence Strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 17: System Integration and Advanced Frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

User Notifications and Alerts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Accessibility and VoiceOver Support

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Drag and Drop Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Printing and PDF Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Working with the Clipboard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

System Extensions and Services

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 18: Testing, Debugging, and Profiling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Unit Testing with XCTest

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Test Organization and Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Mocking Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Debugging with LLDB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Profiling with Instruments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Memory Leak Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Performance Analysis Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 19: Security, Sandboxing, and Code Signing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

macOS Security Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

App Sandbox and Entitlements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Code Signing with Apple Certificates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Notarization for Distribution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Secure Storage of Sensitive Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Network Security and TLS

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Common Vulnerabilities and Mitigations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 20: Distribution and Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Distribution Options for macOS Apps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Building for Release

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Mac App Store Submission Process

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Direct Distribution and DMGs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Software Updates and Versioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

TestFlight and Beta Distribution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Post-Launch Maintenance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Chapter 21: Conclusion and Next Steps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Core Principles of Swift Development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Staying Current with Swift Evolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Contributing to the Swift Community

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

Advanced Topics for Further Study

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/buildingmacosapplicationswithswift.