SWIFT PROGRAMMING

FROM FUNDAMENTALS TO MASTERY

A COMPLETE GUIDE TO MODERN SWIFT DEVELOPMENT

START STRONG
From your first line of Swift code to real apps

MODERN CONCURRENCY
Master structured concurrency with Swift 6's strict model

POWERFUL PATTERNS
Protocol-oriented design, generics, extensions and more

PRODUCTION READY
Build scalable, maintainable apps with proven architecture

UP TO DATE FOR 2026
Swift 6, typed throws, approachable concurrency defaults and the latest features

ETHAN HARRISON

Swift Programming: From Fundamentals to Mastery

A Complete Guide to Modern Swift Development

Steve Publications

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

This version was published on 2026-08-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Complete Guide to Modern Swift Development

Swift is a modern, fast and safe programming language that has transformed how developers build software for Apple platforms and beyond. This book takes you from your very first line of Swift code through advanced topics like structured concurrency, protocol-oriented design, generics and production-grade architecture. Each chapter builds on the last with clear explanations, working code examples and practical guidance on when to use each feature and why. Whether you are new to programming or an experienced developer mastering a new language, this book will give you deep, accurate knowledge of Swift as it stands in 2026 - including Swift 6's strict concurrency model, typed throws, approachable concurrency defaults and the latest language features.

Introduction: Why Swift Matters

The Birth of a Language - Apple's Challenge and Chris Lattner's Vision

In 2010, Chris Lattner, creator of the LLVM compiler infrastructure, began developing a new programming language at Apple. At the time, iOS and macOS development relied on Objective-C, a language that had served Apple well for decades but carried significant baggage from its C heritage. Null pointer crashes, memory management bugs, verbose syntax and a runtime model that made performance optimization difficult were constant sources of frustration for developers building complex applications [1].

Lattner's goal was ambitious: create a language that combined the safety and expressiveness of modern languages with the raw performance of systems programming. The result was Swift, first unveiled at Apple's Worldwide Developers Conference on June 2, 2014, alongside iOS 8 [2]. The announcement stunned the developer world - Apple had secretly built an entirely new language over four years and was asking millions of developers to adopt it.

Swift's lineage is eclectic. It draws from Objective-C for interoperability, Rust for memory safety concepts, Haskell for functional programming features like higher-order functions and pattern matching, Python and Ruby for syntax clarity, C# for properties and generics and CLU for abstract data types [3]. This diverse influence shaped Swift into a multi-paradigm language that supports procedural, object-oriented, protocol-oriented and functional programming styles.

In March 2016, Apple open-sourced Swift under the Apache 2.0 license, inviting contributions from developers worldwide [4]. This decision transformed Swift from an Apple-only tool into a general-purpose language with active communities building server-side applications, command-line tools, embedded systems and cross-platform software on Linux, Windows, Android and WebAssembly.

Design Principles - Safety, Speed, Interoperability, Expressiveness

Swift's design is guided by four core principles that influence every feature in the language:

Safety. Swift eliminates entire categories of bugs through its type system. Optionals force you to handle the absence of values explicitly instead of silently crashing with null pointer exceptions. Value semantics prevent unexpected mutation through shared references. The compiler catches mismatched types, unhandled errors and unreachable code before your program ever runs. In Swift 6, strict concurrency checking extends this safety to concurrent code by preventing data races at compile time [5].

Performance. Swift compiles to highly optimized machine code using the LLVM backend, producing binaries that rival C++ in performance-critical scenarios [6]. The language is designed so that safe abstractions can be as fast as low-level code. Features like value semantics with copy-on-write optimization, whole-module optimization and generic specialization allow the compiler to eliminate overhead that other languages bake into their runtime.

Interoperability. Swift was designed to coexist with Objective-C from day one. You can call Objective-C APIs directly from Swift code, use existing Cocoa and Cocoa Touch frameworks and gradually migrate legacy codebases without rewriting everything at once [7]. This pragmatic approach made Swift adoption practical for millions of developers who had invested years in Objective-C projects.

Expressiveness. Swift's syntax is designed to be readable and concise without sacrificing precision. Type inference lets you write `let name = "Swift"` instead of explicitly declaring the type. String interpolation embeds values directly into strings with `\()` syntax. Pattern matching with switch statements handles complex data structures elegantly. These features reduce boilerplate and let developers focus on solving problems rather than fighting the language.

Swift vs. Other Languages - Where Swift Fits in the Ecosystem

Understanding Swift requires understanding how it differs from languages you may already know:

Swift vs. Objective-C. Swift is not a minor improvement over Objective-C; it is a fundamentally different language. Swift uses static typing instead of dynamic typing, value types alongside reference types instead of only reference types and compile-time safety checks instead of runtime message dispatch for most operations. Swift code is generally 20 to 40 percent shorter than equivalent Objective-C code for the same functionality [8].

Swift vs. C++. Both languages target high-performance scenarios and compile to machine code. However, Swift prioritizes developer safety over raw control. Swift uses automatic reference counting instead of manual memory management, prevents data races through its concurrency model instead of relying on lock primitives and provides a much simpler standard library. Where C++ gives you many ways to shoot yourself in the foot, Swift constrains your choices to safe patterns.

Swift vs. Rust. Rust and Swift share similar goals: memory safety without garbage collection and fearless concurrency. Both use ownership concepts and compile to native code. However, Rust's borrow checker is more strict and requires explicit lifetime annotations in many cases. Swift trades some of Rust's guarantees for developer ergonomics, using reference counting and a simpler ownership model that most developers find easier to learn [9].

Swift vs. Kotlin. Both languages target modern development with null safety, coroutines or structured concurrency, and strong standard libraries. Kotlin focuses on JVM interoperability and Android development, while Swift targets Apple platforms and native performance. Their syntax is surprisingly similar in many areas - both use data classes/structs with value semantics, extension functions/methods, and expressive collection operations.

How to Use This Book - Structure, Conventions, and What You'll Learn

This book is organized as a progressive journey from fundamentals to mastery. Here is how to navigate it:

Chapters 1 through 3 cover the absolute basics: variables, types, operators, control flow, collections and strings. If you are new to programming, start here and move forward at your own pace. If you already know another language, you may skim these chapters but should still read them - Swift has unique behaviors that differ from most languages.

Chapters 4 through 6 dive into functions, closures, and optionals - the core building blocks of Swift code. Optionals in particular deserve careful study; they are Swift's most distinctive feature for preventing null-related crashes.

Chapters 7 through 10 cover type system fundamentals: structs, classes, enums, protocols and generics. This is where Swift's design philosophy becomes clearest - the emphasis on value semantics, protocol-oriented design over class inheritance and type-safe abstractions through generics.

Chapters 11 through 13 tackle advanced topics: error handling, memory management, access control, opaque and existential types, property wrappers, result builders and modern concurrency with async/await, actors and Sendable. These chapters assume comfort with earlier material and are essential for professional-level Swift development.

Chapter 14 ties everything together with real-world software engineering: testing, debugging, performance optimization, package management, API design, architecture patterns and idiomatic Swift conventions. This chapter is designed as a long-term reference for production development.

Throughout the book, you will find complete, compilable code examples that demonstrate each concept in context. Code blocks are written for modern Swift (targeting Swift 6.x) and follow current Apple style guidelines unless noted otherwise. When features differ between Swift versions, I note the version requirements explicitly.

A note on frameworks vs. language. Swift is a programming language; SwiftUI, Foundation, UIKit, Combine, and other Apple technologies are frameworks built using Swift. This book focuses on the Swift language itself - its syntax,

type system, concurrency model and design patterns. Where framework-specific features like @State or @Published are relevant to understanding language concepts like property wrappers, I include them with clear attribution to their framework origins.

Your First Swift Program - A Taste of Modern Swift

Let us start with a complete Swift program that demonstrates several modern language features:

```swift
import Foundation

// A simple value type representing a person
struct Person {
    let name: String
    let age: Int

    // Computed property with a getter
    var decade: String {
        let decadeStart = (age / 10) * 10
        return "\(decadeStart)s"
    }

    // Method that returns an optional
    func greeting(to other: Person?) -> String? {
        guard let other = other else { return nil }
        return "Hello \(other.name), I'm \(name)"
    }
}

// An enum with associated values for different message types
enum Message {
    case text(String)
    case image(url: URL)
    case unknown

    // Computed property using pattern matching
    var description: String {
        switch self {
        case .text(let content):
            return "Text: \(content)"
        case .image(let url):
            return "Image from \(url.absoluteString)"
        case .unknown:
            return "Unknown message type"
        }
```

```
    }
}

// Async function demonstrating modern concurrency
func fetchUserProfile(for person: Person) async -> String {
    // Simulate network delay without blocking
    try? await Task.sleep(for: .milliseconds(100))
    return "Profile loaded for \(person.name), age \(person.age)"
}

// Main entry point - async @main is supported in modern Swift
@main
struct App {
    static func main() async {
        let alice = Person(name: "Alice", age: 32)
        let bob = Person(name: "Bob", age: 28)

        // Value semantics: copying alice creates an independent value
        var aliceCopy = alice
        print("Original decade: \(alice.decade)")   // Output: Original decade:
        ↪  30s

        // Optional handling with guard let
        if let greeting = alice.greeting(to: bob) {
            print(greeting)  // Output: Hello Bob, I'm Alice
        }

        // Pattern matching on enum
        let message = Message.text("Swift is amazing")
        print(message.description)  // Output: Text: Swift is amazing

        // Async/await for concurrent operations
        let profile = await fetchUserProfile(for: alice)
        print(profile)  // Output: Profile loaded for Alice, age 32
    }
}
```

This short program touches on many concepts we will explore in depth: value types with structs, computed properties, optionals, enums with associated values, exhaustive pattern matching with switch, async functions and structured concurrency. By the end of this book, every line here will be familiar territory - and you will understand exactly why Swift was designed this way.

The journey begins now.

Chapter 1: Foundations - Variables, Types, and Basic Values

Every Swift program is built from values stored in variables and constants. Understanding how Swift manages these values - their types, mutability and the rules that govern them - is the foundation for everything else in the language. This chapter covers the basics you need to start writing correct Swift code.

Constants and Variables - let vs. var and Immutability by Default

In Swift, you declare constants with `let` and variables with `var`. A constant holds a single value that cannot be changed after it is set. A variable can be reassigned to a new value of the same type.

```swift
// Constants - values that never change
let appName = "MyApp"
let maxRetries = 3
let pi = 3.14159

// Variables - values that may change
var score = 0
var isLoggedIn = false
var currentLevel = 1
```

Swift encourages immutability by default. You should declare a value as a constant unless you have a clear reason for it to change. This practice makes code easier to reason about because you can trust that a `let` binding will always hold the same value throughout its lifetime. It also helps the compiler optimize your code more aggressively.

Attempting to modify a constant produces a compile-time error:

```
let version = "1.0"
version = "2.0"  // Error: cannot assign to value: 'version' is a 'let' constant
```

This behavior differs from many other languages where `var` or equivalent is the default and immutability requires extra keywords like `final`, `const` or `readonly`. Swift flips this convention: mutability is explicit, and you must opt into it with `var`.

When to use let vs. var. Use `let` for values that conceptually do not change: configuration settings, computed results, function parameters (which are always constants unless marked otherwise) and data loaded from external sources that your code should not modify. Use `var` for state that genuinely changes over time: counters, user input, mutable caches and accumulators in algorithms.

A common beginner mistake is declaring everything as a variable out of habit. Resist this urge. If you find yourself reaching for `var`, ask whether the value truly needs to change. Often it does not.

Type Inference and Explicit Typing - How Swift Deduces Types

Swift is a statically typed language, which means every value has a type that is known at compile time. The compiler uses these types to catch errors before your program runs - for example, adding a string to an integer or calling a method that does not exist on a given type.

Unlike some statically typed languages that require you to declare types explicitly everywhere, Swift supports powerful type inference. When you initialize a variable or constant with a value, the compiler deduces its type from that value:

```
let name = "Alice"        // Compiler infers String
let age = 30              // Compiler infers Int
let height = 5.9          // Compiler infers Double
let isActive = true       // Compiler infers Bool
let items = [1, 2, 3]     // Compiler infers [Int]
```

Type inference works because the initial value provides enough information for the compiler to determine the exact type. You can still write explicit type

annotations when you want to be clear about your intent or when the compiler cannot infer the type:

```
let name: String = "Alice"        // Explicit type annotation
let age: Int = 30                 // Redundant but allowed
let count: Int                    // Declaration without initialization
count = 5                         // Type must be known before assignment
```

When you declare a variable without initializing it, you must provide an explicit type because the compiler has no value to infer from. You can then assign a value of that type later:

```
var temperature: Double
temperature = 98.6    // OK - matches declared type
temperature = "hot"   // Error: cannot assign value of type 'String' to type
↪  'Double'
```

When to use explicit types. Use explicit type annotations in these situations:

- When declaring a variable without an initial value
- When the inferred type might be ambiguous or surprising to readers
- In public APIs where the type is part of the contract
- When you want to document your intent more clearly than inference alone provides

For local variables and internal implementation details, trust Swift's type inference. Over-annotating types adds noise without adding value.

Basic Data Types - Integers, Floats, Booleans and Their Ranges

Swift provides a rich set of built-in types for representing numeric values. Unlike many languages that use a single integer type, Swift offers multiple integer types with different sizes to match your precision and memory requirements.

Integer types. Swift has signed integers (which can represent negative values) and unsigned integers (which cannot). Each comes in four sizes: 8-bit, 16-bit, 32-bit and 64-bit.

Type	Description	Range
Int8	Signed 8-bit integer	-128 to 127
UInt8	Unsigned 8-bit integer	0 to 255
Int16	Signed 16-bit integer	-32,768 to 32,767
UInt16	Unsigned 16-bit integer	0 to 65,535
Int32	Signed 32-bit integer	-2,147,483,648 to 2,147,483,647
UInt32	Unsigned 32-bit integer	0 to 4,294,967,295
Int64	Signed 64-bit integer	-9.2 quintillion to 9.2 quintillion
UInt64	Unsigned 64-bit integer	0 to 18.4 quintillion

In practice, you will rarely use the specific-sized types directly. Swift provides a platform-independent `Int` type that matches the native word size of the current platform: 64 bits on modern 64-bit systems and 32 bits on 32-bit systems [10]. For most counting, indexing and general-purpose integer work, use `Int`:

```
let numberOfItems = 42           // Int - the default integer type
let fileSize: UInt64 = 1_073_741_824   // Use specific types when needed
```

Swift also supports digit separators for readability in large numeric literals:

```
let population = 8_000_000_000    // Much easier to read than 8000000000
let timestamp = 1_725_000_000
```

Floating-point types. Swift provides two floating-point types: `Float` and `Double`. The `Float` type is a 32-bit floating-point number with about 6 to 9 decimal digits of precision. The `Double` type is a 64-bit floating-point number with about 15 to 17 digits of precision [10].

```
let piFloat: Float = 3.14159
let piDouble: Double = 3.141592653589793
let temperature = 98.6    // Inferred as Double by default
```

Always prefer `Double` over `Float` unless you have a specific reason to use the smaller type, such as interfacing with C APIs that expect `Float` or working in memory-constrained environments. Swift infers floating-point literals as

`Double` by default because the extra precision is usually worth the minimal additional memory cost.

Floating-point arithmetic follows IEEE 754 rules, which means you may encounter small rounding errors:

```
let result = 0.1 + 0.2    // Result is approximately 0.30000000000000004
print(result == 0.3)      // Prints: false
```

When comparing floating-point values, use a tolerance-based comparison instead of exact equality:

```
func approximatelyEqual(_ a: Double, _ b: Double, tolerance: Double = 1e-10) ->
↪  Bool {
    return abs(a - b) < tolerance
}

print(approximatelyEqual(0.1 + 0.2, 0.3))    // Prints: true
```

Boolean type. Swift's `Bool` type has exactly two values: `true` and `false`. Unlike some languages that treat zero, empty strings, or nil as falsy, Swift requires explicit boolean values in conditional contexts:

```
let isLoggedIn = true
let hasPermission = false

if isLoggedIn {
    print("Welcome back!")
}

// This does NOT work - Swift requires an actual Bool
let count = 0
if count {
    // Error: Cannot convert value of type 'Int' to expected condition type
    ↪  'Bool'
}

// Write this instead
if count > 0 {
    print("You have items")
}
```

This strictness prevents a common class of bugs where developers accidentally write `if x = y` (assignment) instead of `if x == y` (comparison). Swift's compiler catches this error immediately.

Type conversion. Swift does not perform implicit type conversions between different numeric types. You must convert explicitly to avoid accidental loss of precision or unexpected behavior:

```
let integerValue: Int = 42
let doubleValue: Double = 3.14

// This does NOT work - no implicit conversion
// let sum = integerValue + doubleValue    // Error

// Convert explicitly
let sum = Double(integerValue) + doubleValue    // 45.14
let roundedInt = Int(doubleValue)               // 3 (truncates toward zero)
```

Explicit conversion makes your intent clear and forces you to consider whether the conversion is safe. For example, converting a large `Double` to an `Int` can overflow or lose precision:

```
let hugeNumber: Double = 1e20
let truncated = Int(hugeNumber)    // May overflow on 32-bit platforms
```

Strings and Characters - Unicode, Concatenation, Interpolation and String Literals

Swift's `String` type is fully Unicode-compliant and designed for correctness with international text. Every character in a Swift string is represented as a UTF-8 encoded scalar value, which means strings can represent any character from any writing system [10].

Creating strings. You create strings using string literals enclosed in double quotes:

```
let greeting = "Hello, world!"
let emptyString = ""
let alsoEmpty = String()    // Equivalent to ""
```

Swift supports multiline string literals using triple quotes. These preserve line breaks and indentation exactly as written:

```
let poem = """
Roses are red,
Violets are blue,
Swift is type-safe,
And so am I too.
"""
```

Multiline strings are especially useful for templates, SQL queries, configuration text and any content that naturally spans multiple lines.

String concatenation. You combine strings using the + operator or by appending with +=:

```
let firstName = "Alice"
let lastName = "Smith"
let fullName = firstName + " " + lastName    // "Alice Smith"

var message = "Hello"
message += ", world!"    // "Hello, world!"
```

String interpolation. Swift's string interpolation lets you embed values directly into strings using `\()` syntax. The value inside the parentheses is converted to a string and inserted at that position:

```
let name = "Alice"
let age = 30
let message = "\(name) is \(age) years old"    // "Alice is 30 years old"

// Complex expressions are supported
let total = 19.99 * 1.08
let receipt = "Total: $\(String(format: "%.2f", total))"    // "Total: $21.59"
```

String interpolation works with any type that conforms to the `CustomStringConvertible` protocol, which includes all standard library types. You can also make your own types interpolatable by implementing this protocol.

Characters. A Swift `Character` represents a single extended grapheme cluster - what users perceive as one character, even if it requires multiple Unicode scalar values to represent:

```swift
for char in "👋🏽" {
    print(char)    // Prints: 👋🏽 (one Character, despite multiple scalars)
}

let emojiCount = "Hello 👋".count    // 7 characters
```

This behavior matters for international text. An accented character like “é” can be represented as a single precomposed scalar or as “e” plus a combining accent mark. Swift treats both as one `Character` because users see them the same way.

String operations. Swift strings provide many useful methods:

```swift
let text = "  Hello, World!  "

// Trimming whitespace
print(text.trimmingCharacters(in: .whitespaces))    // "Hello, World!"

// Case conversion
print(text.uppercased())    // "  HELLO, WORLD!  "
print(text.lowercased())    // "  hello, world!  "

// Searching and replacing
print(text.contains("World"))        // true
print(text.replacingOccurrences(of: "World", with: "Swift"))    // "  Hello,
↪  Swift!  "

// Splitting
let words = text.trimmingCharacters(in: .whitespaces).split(separator: ",")
print(words)    // ["Hello", " World!"]

// Prefix and suffix checks
print(text.hasPrefix("He"))    // false (leading spaces matter)
print(text.hasSuffix("!"))     // true
```

String performance considerations. Swift strings use copy-on-write semantics, which means copying a string does not immediately duplicate its underlying data. The actual copy happens only when one of the copies is modified. This optimization makes passing strings around very efficient [11].

However, building large strings through repeated concatenation in a loop can be inefficient because each + operation creates a new string:

```
// Inefficient - creates many intermediate strings
var result = ""
for i in 0..<1000 {
    result += "\(i)"    // Each iteration may allocate a new buffer
}

// Efficient - use join or write to a buffer
let efficient = (0..<1000).map(String.init).joined()
```

For most everyday use, string concatenation is fast enough that premature optimization is unnecessary. Profile your code before worrying about edge cases.

Tuples - Grouping Values Without Defining New Types

Tuples let you group multiple values into a single compound value without defining a custom type. They are useful for returning multiple values from a function, grouping related data temporarily, or passing several arguments as a single unit.

Creating tuples. You create a tuple by enclosing comma-separated values in parentheses:

```
let httpStatusCode = (200, "OK")
let address = (street: "123 Main St", city: "San Francisco", zip: "94105")
let dimensions = 1920, 1080    // Tuple without parentheses in some contexts
```

You can access tuple elements by index or by name if you provided names when creating the tuple:

```
// Access by index
print(httpStatusCode.0)    // 200
print(httpStatusCode.1)    // "OK"

// Access by name
print(address.city)        // "San Francisco"
print(address.zip)         // "94105"
```

Destructuring tuples. You can unpack a tuple's values into individual constants or variables using pattern matching:

```
let (code, message) = httpStatusCode
print(code)      // 200
print(message)   // "OK"

// Ignore specific values with underscore
let (street, _, zipCode) = address
print(street)    // "123 Main St"
print(zipCode)   // "94105"
```

Tuples and functions. Tuples are commonly used to return multiple values from a function:

```
func divide(_ a: Double, by b: Double) -> (quotient: Double, remainder:
↪  Double)? {
    guard b != 0 else { return nil }
    let quotient = (a / b).rounded(.down)
    let remainder = a - (quotient * b)
    return (quotient, remainder)
}

if let result = divide(17.5, by: 4.0) {
    print("Quotient: \(result.quotient), Remainder: \(result.remainder)")
    // Output: Quotient: 4.0, Remainder: 1.5
}
```

When to use tuples vs. structs. Tuples are great for temporary grouping of values within a limited scope. However, they have limitations that make them unsuitable for representing domain concepts:

- Tuple elements are accessed by position or local names, which can be confusing when passed around
- You cannot add methods or computed properties to a tuple
- Tuple types become verbose and hard to read with many elements
- Different tuples with the same element types are indistinguishable to the compiler

When you find yourself using a tuple with three or more elements, or when the grouped values represent a meaningful concept in your domain, define a struct instead:

```swift
// Prefer this over a tuple for domain concepts
struct Address {
    let street: String
    let city: String
    let zipCode: String

    var formatted: String {
        "\(street), \(city) \(zipCode)"
    }
}
```

Type Safety - What the Compiler Guards Against and Why It Matters

Swift's type system is one of its strongest features. The compiler checks types at compile time, catching errors that would otherwise cause crashes or subtle bugs at runtime. This safety comes with minimal overhead in well-written code.

Type checking in action. Consider this example:

```swift
func calculateTotal(price: Double, quantity: Int) -> Double {
    return price * Double(quantity)
}

let total = calculateTotal(price: 19.99, quantity: 3)
// Total is correctly typed as Double

// The compiler prevents obvious mistakes
// let wrong = calculateTotal(price: "nineteen", quantity: 3)
// Error: Cannot convert value of type 'String' to expected argument type
↪  'Double'
```

The compiler ensures that functions receive arguments of the correct type, that you do not assign incompatible values and that operations are valid for the types involved. These checks happen before your code runs, which means many bugs are caught during development rather than in production.

Strong typing vs. dynamic typing. Swift is strongly typed: every value has a specific type, and the compiler enforces type rules strictly. This differs from dynamically typed languages like Python or JavaScript where variables can hold values of any type and type errors surface only at runtime.

The trade-off is clear: strong typing requires more upfront effort to declare types correctly, but it prevents entire classes of bugs and makes refactoring safer. When you rename a type or change a function's parameter type, the compiler shows you every place in your codebase that needs updating. In a dynamically typed language, those errors would only appear when someone runs the affected code path.

Type aliases. Swift allows you to create alternative names for existing types using `typealias`. This is useful for improving readability or providing abstractions:

```
typealias UserID = Int
typealias Price = Double
typealias ResultHandler = (Result<Data, Error>) -> Void

func fetchUser(id: UserID) async throws -> User { ... }
func calculateDiscount(price: Price) -> Price { ... }
```

Type aliases do not create new types - they are purely for documentation and convenience. The compiler treats `UserID` exactly the same as `Int`. For true type safety where you want to prevent mixing different concepts that share the same underlying type, use a newtype wrapper (a struct with a single stored property).

Summary. This chapter covered the fundamentals of Swift's value system: constants and variables, type inference, numeric types, strings, tuples and the role of type safety. These building blocks appear in every Swift program you will write. In the next chapter, we will explore how to combine these values into expressions, make decisions with control flow and iterate over collections - the tools you need to write programs that do useful work.

Chapter 2: Operators, Expressions and Control Flow

Values alone are not very interesting. Programs become useful when you combine values with operators, make decisions based on conditions and repeat operations over collections of data. This chapter covers Swift's operators, conditional statements, loops and the pattern matching capabilities that set it apart from many other languages.

Arithmetic, Comparison and Logical Operators - Precedence and Associativity

Swift provides a familiar set of operators for arithmetic, comparison and logical operations. Understanding how these operators interact through precedence rules is essential for writing correct expressions without excessive parentheses.

Arithmetic operators. Swift supports the standard arithmetic operators: addition (+), subtraction (-), multiplication (*), division (/) and remainder (%). These operators work on numeric types with the expected behavior:

```
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

// Floating-point division preserves decimals
let preciseQuotient = 10.0 / 3.0    // 3.333...
```

Swift also provides compound assignment operators that combine an operation with assignment:

```swift
var count = 5
count += 3     // Equivalent to count = count + 3, now 8
count -= 2     // Now 6
count *= 4     // Now 24
count /= 6     // Now 4
count %= 3     // Now 1
```

The unary minus operator negates a value and the unary plus operator explicitly indicates a positive value (though it is rarely needed):

```swift
let negative = -5
let explicitPositive = +5    // Same as just 5
```

Important: no implicit overflow. By default, Swift checks for integer overflow at runtime and triggers a trap if an arithmetic operation exceeds the type's range. This behavior prevents silent wraparound bugs that are common in languages like C and C++:

```swift
var maxUInt8: UInt8 = 255
// maxUInt8 += 1    // RuntimeError: Arithmetic overflow

// Use wrapping operators when overflow is intentional
maxUInt8 &+= 1    // Wraps around to 0 (no error)
```

Wrapping operators (`&+`, `&-`, `&*`, `&/`, `&%`) perform arithmetic without overflow checks and wrap around on overflow. Use them only when you explicitly want wrapping behavior, such as in cryptographic algorithms or bit manipulation.

Comparison operators. Swift's comparison operators return boolean values:

```swift
let a = 5
let b = 10

print(a == b)    // false - equal to
print(a != b)    // true - not equal to
print(a < b)     // true - less than
print(a > b)     // false - greater than
print(a <= b)    // true - less than or equal to
print(a >= b)    // false - greater than or equal to
```

These operators work on any type that supports comparison, including integers, floating-point numbers, strings (using lexicographic ordering), and characters. Custom types can support comparison by conforming to the `Comparable` protocol.

Logical operators. Swift provides three logical operators for combining boolean values:

- Logical NOT (`!`) - inverts a boolean value
- Logical AND (`&&`) - true only if both operands are true
- Logical OR (`||`) - true if at least one operand is true

```swift
let isLoggedIn = true
let hasPermission = false

if !isLoggedIn {
    print("Please log in")
}

// Short-circuit evaluation: && stops at first false, || stops at first true
if isLoggedIn && hasPermission {
    print("Access granted")    // Not executed - hasPermission is false
}

if isLoggedIn || hasPermission {
    print("At least one condition met")    // Executed - isLoggedIn is true
}
```

Short-circuit evaluation means that Swift does not evaluate the right side of `&&` if the left side is false, and does not evaluate the right side of `||` if the left side is true. This behavior allows you to write safe conditional expressions:

```
let numbers = [1, 2, 3]
if !numbers.isEmpty && numbers[0] > 0 {
    print("First number is positive")
}
// If we reversed the conditions, this would crash when numbers is empty:
// if numbers[0] > 0 && !numbers.isEmpty { ... }  // Crash on empty array
```

Operator precedence. When an expression contains multiple operators, Swift applies them in order of precedence. Higher-precedence operators are evaluated first:

```
let result = 2 + 3 * 4    // 14, not 20 - multiplication before addition
let booleanResult = true && false || true    // true - && before ||
```

The full precedence hierarchy is complex, but the key rules to remember are: unary operators bind tightest, then multiplication/division/modulo, then addition/subtraction, then comparison, then logical AND, then logical OR. When in doubt, use parentheses to make your intent explicit:

```
let clearResult = (2 + 3) * 4    // 20 - parentheses override precedence
```

Range Operators - Closed, Half-Open and One-Sided Ranges

Swift's range operators are a distinctive feature that creates intervals of values for use in loops, collection subscripting, and pattern matching. Understanding the differences between range types is essential for writing correct code.

Closed range operator (`...`). The closed range includes both endpoints:

```
let closedRange = 1...5    // Includes 1, 2, 3, 4, and 5

for number in 1...5 {
    print(number)    // Prints 1 through 5
}
```

Half-open range operator (`..<`). The half-open range includes the start but excludes the end:

```swift
let halfOpenRange = 1..<5    // Includes 1, 2, 3, and 4 - NOT 5

for index in 0..<array.count {
    print(array[index])    // Safe: stops before array.count
}
```

Half-open ranges are the most commonly used range type in Swift because they align naturally with zero-based indexing. When iterating over a collection's indices, you want to go from 0 up to but not including `count`, which is exactly what `0..<count` expresses.

One-sided ranges. One-sided ranges omit one endpoint and extend indefinitely in that direction:

```swift
let array = ["a", "b", "c", "d", "e"]

// From index 2 to the end
let suffix = array[2...]    // ["c", "d", "e"]

// From the start up to (but not including) index 3
let prefix = array[..<3]    // ["a", "b", "c"]

// Up to and including index 2
let throughTwo = array[...2]    // ["a", "b", "c"]
```

One-sided ranges are especially useful for slicing collections without needing to know their length. They also work in pattern matching with switch statements, which we will explore later.

Range types. Swift has several range types depending on the bounds and whether they are closed or half-open:

- `ClosedRange` (`...`) - both endpoints included
- `Range` (`..<`) - start included, end excluded
- `PartialRangeFrom` (`n...`) - from n to infinity
- `PartialRangeUpTo` (`..<n`) - from negative infinity up to but not including n
- `PartialRangeThrough` (`...n`) - from negative infinity through and including n

Ranges can be created with any comparable type, not just integers:

```swift
let alphabetRange = "a"..."z"
if alphabetRange.contains("m") {
    print("m is in the alphabet")
}
```

Conditional Statements - if, guard, switch and Their Distinct Roles

Swift provides three primary conditional constructs: `if` for branching based on conditions, `guard` for early exits when preconditions fail, and `switch` for exhaustive pattern matching. Each serves a distinct purpose in writing clear control flow.

if statements. The `if` statement executes code when a condition is true:

```swift
let temperature = 25

if temperature > 30 {
    print("It's hot outside")
} else if temperature > 20 {
    print("It's warm outside")
} else {
    print("It's cool outside")
}
```

Swift does not require parentheses around the condition, and curly braces are mandatory even for single-line bodies. This consistency prevents a common source of bugs in languages where omitting braces leads to subtle errors:

```swift
// Always use braces - this is required in Swift
if x > 0 {
    print("Positive")
} else {
    print("Zero or negative")
}
```

The ternary conditional operator. For simple conditions with two outcomes, Swift provides the ternary operator as a compact alternative to if-else:

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

Use the ternary operator sparingly. It is appropriate for simple, obvious conditions but becomes hard to read when nested or used with complex expressions.

guard statements. The `guard` statement is one of Swift's most valuable features for writing clear control flow. It requires a condition to be true and exits the current scope immediately if the condition fails:

```swift
func greetUser(_ name: String?) {
    guard let name = name, !name.isEmpty else {
        print("No valid name provided")
        return
    }

    // From here on, name is guaranteed to be a non-empty String
    print("Hello, \(name)!")
}
```

The `guard` statement has two key characteristics:

1. The else branch must exit the current scope using `return`, `break`, `continue`, `throw`, or `fatalError()` - it cannot fall through to code after the guard statement.
2. Any bindings created in the guard condition (like `let name = name` or `var x = y`) are available in the rest of the scope after the guard passes.

When to use guard vs. if. Use `guard` when you need to validate preconditions and exit early if they fail. Use `if` when you have two branches of equal importance that both continue within the same scope:

```swift
// Guard for validation - "fast fail" pattern
func processUser(_ user: User?) {
    guard let user = user else { return }
    guard user.isActive else { return }
    guard !user.banned else { return }

    // Main logic here - all preconditions satisfied
    performAction(for: user)
}

// If for branching - both paths are valid continuations
func describeTemperature(_ temp: Int) -> String {
    if temp > 30 {
        return "Hot"
    } else {
        return "Not hot"
    }
}
```

Using `guard` for validation keeps your main logic unindented and easy to read. This pattern is sometimes called the "railway orientation" of code: guard statements act as switches that route invalid inputs away from the happy path, which runs straight down the page without nesting.

switch statements. Swift's `switch` statement is far more powerful than in most languages. It supports pattern matching on enums with associated values, ranges, tuples, and custom patterns. Most importantly, switch statements must be exhaustive: you must handle every possible case, or the compiler produces an error.

```swift
let httpStatusCode = 200

switch httpStatusCode {
case 200:
    print("Success")
case 404:
    print("Not found")
case 500:
    print("Server error")
default:
    print("Unknown status code: \(httpStatusCode)")
}
```

The `default` case handles any values not explicitly matched. However, when switching on an enum with a known set of cases, Swift requires you to handle every case and does not allow a default:

```swift
enum Direction {
    case north, south, east, west
}

let direction = Direction.north

switch direction {
case .north:
    print("Going up")
case .south:
    print("Going down")
case .east:
    print("Going right")
case .west:
    print("Going left")
// No default allowed - all cases are handled
}
```

This exhaustiveness checking is a powerful safety feature. When you add a new case to an enum, the compiler shows you every switch statement in your codebase that needs updating. This prevents silent bugs where new cases fall through unhandled.

Loops and Iteration - for-in, while, repeat-while and When to Use Each

Swift provides three loop constructs: `for-in` for iterating over collections and sequences, `while` for repeating while a condition is true, and `repeat-while` for executing at least once before checking a condition.

for-in loops. The `for-in` loop is the most commonly used loop in Swift. It iterates over elements of a collection, range, sequence, or any type that conforms to the `Sequence` protocol:

```swift
// Iterating over an array
let names = ["Alice", "Bob", "Charlie"]
for name in names {
    print("Hello, \(name)")
}

// Iterating over a range
for i in 1...5 {
    print(i)    // Prints 1 through 5
}

// Iterating with indices
for index in names.indices {
    print("\(index): \(names[index])")
}

// Iterating with both index and value using enumerated()
for (index, name) in names.enumerated() {
    print("Person \(index) is \(name)")
}
```

The `enumerated()` method returns a sequence of tuples containing the index and element, which is cleaner than manually tracking an index variable.

Iterating over dictionaries. When iterating over a dictionary, each iteration produces a tuple with the key and value:

```swift
let scores = ["Alice": 95, "Bob": 87, "Charlie": 92]

for (name, score) in scores {
    print("\(name): \(score)")
}

// Note: dictionary iteration order is not guaranteed
```

while loops. Use `while` when you need to repeat code while a condition remains true, and the number of iterations is not known in advance:

```swift
var number = 1
while number <= 100 {
    print(number)
    number *= 2    // 1, 2, 4, 8, 16, 32, 64
}
```

The condition is checked before each iteration. If it is false initially, the loop body never executes.

repeat-while loops. Use `repeat-while` when you need to execute code at least once and then repeat while a condition is true:

```
var guess = 0
let secretNumber = 7

repeat {
    print("Guess a number: ")
    // In a real app, read user input here
    guess = Int.random(in: 1...10)
} while guess != secretNumber

print("You guessed it!")
```

The condition is checked after each iteration, guaranteeing that the body executes at least once. This pattern is useful for input validation, retry loops, and game loops.

Loop labels. When you have nested loops, you can use labels to specify which loop a `break` or `continue` statement applies to:

```
outerLoop: for i in 1...3 {
    for j in 1...3 {
        if i == 2 && j == 2 {
            break outerLoop    // Breaks out of both loops
        }
        print("i=\(i), j=\(j)")
    }
}
```

Use loop labels sparingly. Deeply nested loops often indicate that your code would be clearer if refactored into separate functions.

Avoiding C-style for loops. Swift does not support traditional C-style for loops with initialization, condition, and increment expressions (`for (int i = 0; i < n; i++)`). Instead, use `for-in` with a range:

```swift
// Prefer this
for i in 0..<10 {
    print(i)
}

// Over this pattern from other languages
// for var i = 0; i < 10; i++ { ... }  // Not valid Swift
```

The `for-in` approach is safer because it eliminates off-by-one errors and mutation of loop variables.

Early Returns with guard - Writing Clear Control Flow

We covered `guard` briefly in the conditional statements section, but it deserves deeper treatment because it fundamentally changes how you structure functions in Swift. The guard pattern promotes a style of programming where invalid inputs are rejected immediately and the main logic proceeds without nesting.

The problem with nested if statements. Consider a function that needs to validate several conditions before performing its main work:

```swift
// Nested if - hard to read, deeply indented
func processOrder(_ order: Order?) -> String {
    if let order = order {
        if !order.items.isEmpty {
            if order.customer != nil {
                if order.total > 0 {
                    // Main logic buried in nesting
                    return "Processing order for \(order.customer!.name)"
                } else {
                    return "Order total must be positive"
                }
            } else {
                return "No customer specified"
            }
        } else {
            return "Order has no items"
        }
    } else {
        return "No order provided"
    }
}
```

This code is hard to read because the main logic is buried four levels deep. Each additional validation requirement adds another level of nesting, making the function increasingly difficult to understand.

The guard solution. Using `guard` statements flattens this structure:

```
// Guard - clean, linear flow
func processOrder(_ order: Order?) -> String {
    guard let order = order else {
        return "No order provided"
    }
    guard !order.items.isEmpty else {
        return "Order has no items"
    }
    guard let customer = order.customer else {
        return "No customer specified"
    }
    guard order.total > 0 else {
        return "Order total must be positive"
    }

    // Main logic at the same indentation level
    return "Processing order for \(customer.name)"
}
```

Each guard statement handles one failure case and returns immediately. By the time you reach the main logic, all preconditions are satisfied and the code runs without further checks. This pattern scales gracefully: adding a new validation requirement means adding one more guard statement at the top, not wrapping existing code in another if block.

Guard with binding. Guard statements can bind optional values to constants or variables that are available after the guard passes:

```swift
func findUser(named name: String, in users: [User]?) -> User? {
    guard let users = users else { return nil }

    for user in users {
        if user.name == name {
            return user
        }
    }

    return nil
}

// Multiple bindings in one guard
func connect(to host: String?, port: Int?) -> Bool {
    guard let host = host, let port = port, port > 0 && port <= 65535 else {
        print("Invalid connection parameters")
        return false
    }

    // Both host and port are non-optional here
    return establishConnection(host: host, port: port)
}
```

Guard in loops. Guard statements work inside loops to skip iterations when conditions fail:

```swift
func sumPositiveNumbers(_ numbers: [Int?]) -> Int {
    var total = 0

    for number in numbers {
        guard let value = number, value > 0 else {
            continue    // Skip nil and non-positive values
        }
        total += value
    }

    return total
}
```

Summary. This chapter covered Swift's operators, control flow statements and iteration patterns. Key takeaways: prefer `guard` for validation and early returns to keep main logic unindented; use half-open ranges (`..<`) for zero-based indexing; leverage `switch` exhaustiveness checking as a safety net when working with enums and favor `for-in` loops over manual index management. In the next chapter, we will explore Swift's collection types - arrays, sets and dictionaries - and the powerful higher-order functions that make working with collections concise and expressive.

Chapter 3: Collections - Arrays, Sets, Dictionaries and Sequences

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

Arrays - Ordered Collections, Performance Characteristics and Common Operations

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

Sets - Unordered Unique Values, Set Algebra and Hash Requirements

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

Dictionaries - Key-Value Pairs, Lookup Performance and Mutation Patterns

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

Sequence and Collection Protocols - The Foundation of Iteration

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

Higher-Order Functions - map, filter, reduce, compactMap and Their Combinatorics

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

Choosing the Right Collection - Trade-offs Between Arrays, Sets and Dictionaries

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

Chapter 4: Functions - Building Reusable Code

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

Defining Functions - Parameters, Return Types and Calling Conventions

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

Parameter Names and Labels - Swift's Unique Approach to Readability

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

Default Argument Values - Flexible APIs Without Overloading

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

Variadic Parameters - Handling Unknown Numbers of Arguments

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

In-Out Parameters - Mutating Arguments and When to Use Them

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

Nested Functions and Function Scope - Encapsulation Within Functions

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

Chapter 5: Closures and Functional Patterns

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

What Are Closures - Self-Contained Blocks of Reusable Code

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

Closure Syntax - From Full Form to Shorthand Expressions

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

Trailing Closure Syntax - Making APIs Read Like Natural Language

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

Capturing Values - How Closures Reference Their Surrounding Context

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

Autoclosures - Deferring Execution for Performance and Clarity

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

Functional Programming with Swift - Immutability, Purity and Composition

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

Chapter 6: Optionals - Handling Absence Safely

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

The Optional Type - Modeling “Maybe a Value” Without Null

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

Unwrapping Optionals - force unwrap, if let, guard let, and Their Risks

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

Implicitly Unwrapped Optionals - When (Rarely) to Use Them

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

Optionals in API Design - Communicating Intent Through Types

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

Chapter 7: Structs, Classes, and Value Semantics

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

Defining Structs - Value Types with Copy-on-Write Semantics

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

Defining Classes - Reference Types and Shared Identity

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

Properties - Stored, Computed, Type Properties and Property Observers

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

Methods - Instance Methods, Type Methods and Mutating Methods

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

Initialization - Designated, Convenience and Required Initializers

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

Value vs. Reference Semantics - The Core Design Decision

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

Chapter 8: Enums and Pattern Matching

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

Basic Enumerations - Named Constants with Type Safety

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

Associated Values - Embedding Data Within Enum Cases

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

Raw Values - Enums Backed by Integers, Strings, or Other Types

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

Recursive Enums - Modeling Tree Structures and Nested Data

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

Pattern Matching with switch - Exhaustive Coverage as a Safety Guarantee

Computed Properties and Methods on Enums - Making Enums Powerful

Chapter 9: Protocols, Extensions, and Protocol-Oriented Design

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

Defining Protocols - Declaring Requirements Without Implementation

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

Conforming to Protocols - Satisfying Protocol Requirements

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

Protocol Properties and Methods - Computed vs. Stored, Mutating Requirements

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

Extensions - Adding Functionality to Existing Types

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

Protocol Composition and Inheritance - Combining Multiple Protocols

Protocol-Oriented Design - Why Protocols Over Classes in Modern Swift

Chapter 10: Generics - Writing Reusable Type-Safe Code

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

Generic Functions - Type Parameters and Constraints

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

Generic Types - Structs, Classes, Enums, and Protocols That Work With Any Type

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

Where Clauses - Expressing Complex Type Relationships

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

Associated Types - Abstract Types Within Protocols

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

Protocol Conformance Requirements - Constraining Generic Parameters

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

Real-World Generics - Building Flexible, Reusable APIs

Chapter 11: Error Handling and Result Types

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

Throwing Functions - Declaring and Propagating Errors

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

do-catch Blocks - Catching and Handling Errors Locally

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

Custom Error Types - Designing Meaningful Error Hierarchies

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

The Result Type - Functional Error Handling Without Throwing

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

Defer Statements - Guaranteed Cleanup Regardless of Control Flow

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

Error Handling Strategies - Retry, Fallback, and Propagation Patterns

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

Chapter 12: Advanced Language Features

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

Access Control - Public, Internal, Private, Fileprivate, and Package

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

Memory Management - ARC, Strong References, Weak and Unowned References

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

Circular References - Detecting and Breaking Retain Cycles

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

Opaque Types vs. Existential Types - some Protocol vs. any Protocol

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

Property Wrappers - Encapsulating Property Behavior

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

Result Builders - Building DSLs with Declarative Syntax

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

Type Erasure - Hiding Concrete Types Behind Protocols

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

Chapter 13: Concurrency - Async, Actors, and Structured Tasks

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

Async Functions - Writing Asynchronous Code That Reads Like Synchronous Code

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

Await and Task Suspension - Non-Blocking Execution

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

Tasks and Task Groups - Structured Concurrency and Parallelism

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

Actors - Isolated State and Thread Safety by Design

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

Main Actor and @MainActor - UI Updates and Global Dispatch Queues

Sendable Protocol - Compile-Time Guarantees for Thread Safety

Race Conditions and Data Races - Understanding and Preventing Them

Chapter 14: Production Swift - Tooling, Testing, and Best Practices

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

Swift Package Manager - Dependencies, Targets, and Build Configuration

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

Writing Tests - XCTest, Test Structure, and Mocking Strategies

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

Debugging - Breakpoints, Logging, LLDB, and Crash Analysis

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

Performance Optimization - Profiling, Algorithms, Memory, and Compiler Flags

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

API Design Guidelines - Naming, Consistency, and Developer Experience

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

Architecture Patterns - MVVM, Clean Architecture, and Dependency Injection

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

Security Best Practices - Cryptography, Secrets, and Input Validation

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

Idiomatic Swift - Conventions, Style, and What Makes Code “Swift-like”

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

Conclusion: The Future of Swift

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

Swift as an Open Source Project - Community Governance and Evolution

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

Cross-Platform Swift - Server-Side, Linux, WASM, and Beyond

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

The Road Ahead - Upcoming Features and Language Directions

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

Becoming a Swift Expert - Continuous Learning and Contribution

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

Glossary

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

Index

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

References

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