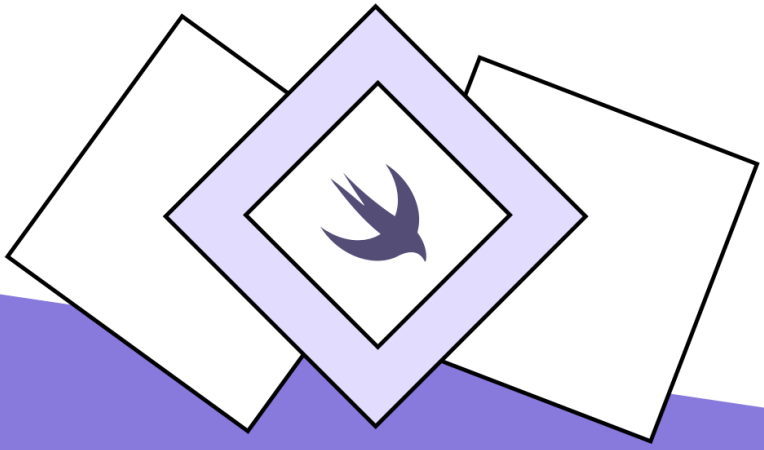
FREE SAMPLE

Swift Gems

100+ tips to take your Swift code to the next level

Natalia Panferova

Updated - November 2025



Introduction

Welcome to “Swift Gems”! This book contains a collection of concise, easily digestible tips and techniques designed for experienced developers looking to advance their Swift expertise.

As an enthusiastic Swift developer with extensive experience in creating robust applications across multiple platforms, I have had the privilege to witness and contribute to the evolution of Swift. This journey has provided me with invaluable insights and advanced techniques that I am eager to share with the community. Swift’s versatility allows it to be used effectively for both small-scale applications and complex enterprise systems, making it a top choice for developers looking to push the boundaries of what’s possible in software development.

Recognizing that many experienced developers are constantly seeking ways to refine their skills and expand their toolkit, this book is designed to serve as a resource for enhancing the quality and efficiency of your Swift code. Whether you are developing for iOS, macOS, watchOS, or tvOS, or even writing Swift on the server, the tips and techniques compiled here will provide you with new perspectives and approaches to tackle everyday coding challenges and innovate in your projects.

“Swift Gems” aims to bridge the gap between intermediate knowledge and advanced mastery, offering a series of easily implementable, bite-sized strategies that can be directly applied to improve your current projects. Each chapter is crafted with precision, focusing on a specific aspect of Swift development, from pattern matching and asynchronous programming to data management and beyond. Practical examples and concise explanations make complex concepts accessible and actionable.

Furthermore, to ensure that you can immediately put these ideas into practice, each concept is accompanied by code snippets and examples that illustrate how to integrate these enhancements effectively. The book is structured to facilitate quick learning and application, enabling you to integrate advanced features and optimize your development workflow efficiently.

Whether you're looking to optimize performance, streamline your coding process, or simply explore new features in Swift, this book will provide you with the tools and knowledge necessary to elevate your coding skills and enhance your development projects.

The content of the book is copyright Nil Coalescing Limited. You can't share or redistribute it without a prior written permission from the copyright owner. If you find any issues with the book, have any general feedback or have suggestions for future updates, you can send us a message to `support@nilcoalescing.com`. I will be updating the book when there are changes to the APIs and to incorporate important feedback from readers. You can check back on `books.nilcoalescing.com/swift-gems` to see if there have been new releases and view the release notes.

I hope you enjoy the book!

Pattern matching and control flow

Let's explore some useful techniques that can be applied in pattern matching and control flow in Swift.

We'll delve into the powerful capabilities of Swift's pattern matching, a feature that goes beyond the basic switch-case structure familiar in many programming languages. We'll look into how to use various Swift constructs to elegantly handle conditions and control flows, including working with enums and optional values. We'll see how to perform operations on continuous data ranges and manage optional values with precision. The techniques we'll cover can simplify our code and enhance its robustness and readability.

This chapter is designed for experienced Swift developers who are looking to refine their skills in efficiently managing program flow and handling data based on specific patterns. By mastering these techniques, you will be equipped to write cleaner, more expressive, and more efficient Swift code, harnessing the full potential of pattern matching and advanced control flow to tackle real-world programming challenges more effectively.

The free sample includes 4 tips from this chapter. To read the remaining 11 tips in this chapter you need to purchase the book from:

https://books.nilcoalescing.com/swift-gems

Overload the pattern matching operator for custom matching behavior

Swift's pattern matching is an exceptionally flexible technique predominantly used in `switch` statements to accommodate a wide range of patterns. In this context, an expression pattern within a `switch` case represents the value of an expression. The core of this functionality hinges on the pattern matching operator (~=), which Swift utilizes behind the scenes to assess whether a pattern corresponds with the value. Typically, ~= performs comparisons between two values of the same type using ==. However, the pattern matching operator can be overloaded to enable custom matching behaviors, offering enhanced control and adaptability in how data is evaluated and handled.

Let's consider a custom type `Circle` and demonstrate how to implement custom pattern matching for it. We'll define a simple `Circle` struct and overload the ~= operator to match a `Circle` with a specific radius. This overload will allow us to use a `Double` in a `switch` statement case to match against a `Circle`.

```
struct Circle {
    var radius: Double
}

func ~= (pattern: Double, value: Circle) → Bool {
    return value.radius == pattern
}


let myCircle = Circle(radius: 5)

switch myCircle {
case 5:
    print("Circle with a radius of 5")
case 10:
    print("Circle with a radius of 10")
default:
    print("Circle with a different radius")
}
```

We can add as many overloads as we need. For example, we can define custom logic to check whether the `Circle`'s radius falls within a specified range. The `switch` statement will now be able to match `myCircle` against `Double` values and ranges, thanks to our custom implementations of the ~= operator.

```
func ~= (pattern: ClosedRange<Double>, value: Circle) → Bool {
    return pattern.contains(value.radius)
}

switch myCircle {
case 0:
    print("Radius is 0, it's a point!")
case 1...10:
    print("Small circle with a radius between 1 and 10")
default:
    print("Circle with a different radius")
}
```

Custom pattern matching in Swift opens up a lot of possibilities for handling complex types more elegantly. By overloading the ~= operator, we can tailor the pattern matching process to suit our custom types. As with any powerful tool, we should use it wisely to enhance our code without compromising on readability.

Switch on multiple optional values simultaneously

Utilizing tuple patterns in `switch` statements offers a robust way to handle multiple optional values simultaneously, allowing for clean and concise management of various combinations of those values.

Consider a scenario where we have two optional integers, `optionalInt1` and `optionalInt2`. Depending on their values, we might want to execute different actions. Here's how we can use a tuple pattern to elegantly address each possible combination of these optional integers.

```
var optionalInt1: Int? = 1
var optionalInt2: Int? = nil

switch (optionalInt1, optionalInt2) {
case let (value1?, value2?):
    print("Both have values: \(value1) and \(value2)")
case let (value1?, nil):
    print("First has a value: \(value1), second is nil")
case let (nil, value2?):
    print("First is nil, second has a value: \(value2)")
case (nil, nil):
    print("Both are nil")
}
```

In this example, the `switch` statement checks the tuple `(optionalInt1, optionalInt2)`. The first case matches when both elements in the tuple are non-nil. Here, each value is unwrapped and available for use within the case block. The second and third cases handle the scenarios where one of the optionals is `nil`, and the other contains a value. The final case addresses the situation where both optionals are `nil`, allowing us to define a clear action for this scenario.

By structuring the `switch` this way, each combination of presence and absence of values is handled explicitly, making the code both easier to follow and maintain. This method significantly reduces the complexity that typically arises from nested if-else

conditions and provides a straightforward approach to branching logic based on multiple optional values.

Iterate over items and indices in collections

Iterating over both the items and their indices in a collection is a common requirement in Swift programming. While the `enumerated()` method might seem like the obvious choice for this task, it's crucial to understand its limitations. The integer it produces starts at zero and increments by one for each item, which is perfect for use as a counter but not necessarily as an index, especially if the collection isn't zero-based or directly indexed by integers.

Here's a typical example using `enumerated()`.

```
var ingredients = ["potatoes", "cheese", "cream"]

for (i, ingredient) in ingredients.enumerated() {
    // The counter helps us display the sequence number, not the index
    print("ingredient number \(i + 1) is \(ingredient)")
}
```

For a more accurate way to handle indices, especially when working with collections that might be subsets or have non-standard indexing, we can use the `zip()` function. This method pairs each element with its actual index, even if the collection has been modified.

```
// Array<String>
var ingredients = ["potatoes", "cheese", "cream"]

// Array<String>.SubSequence
var doubleIngredients = ingredients.dropFirst()

for (i, ingredient) in zip(
    doubleIngredients.indices, doubleIngredients
) {
    // Correctly use the actual indices of the subsequence
    doubleIngredients[i] = "\(ingredient) x 2"
}
```

This approach ensures that we are using the correct indices corresponding to the actual positions in the modified collection.

For a better interface over `zip()`, we can also use the `indexed()` method from Swift Algorithms. It's equivalent to `zip(doubleIngredients.indices, doubleIngredients)` but might be more clear.

```
import Algorithms

// Array<String>
var ingredients = ["potatoes", "cheese", "cream"]

// Array<String>.SubSequence
var doubleIngredients = ingredients.dropFirst()

for (i, ingredient) in doubleIngredients.indexed() {
    // Do something with the index
    doubleIngredients[i] = "\(ingredient) x 2"
}
```

Label loop statements to control execution of nested loops

One of the lesser-known yet incredibly useful features in Swift is the concept of named loops. This feature enhances the control flow in our code, making complex loop structures more manageable and readable.

Named loops are not a separate type of loop but rather a way of labeling loop statements. In Swift, we can assign a name to loops (`for`, `while`, or `repeat-while`) and use this name to specifically control the flow of the program. This is particularly useful when dealing with nested loops and we need to break out of or continue an outer loop from within an inner loop.

The syntax for naming a loop is straightforward. We simply precede the loop with a label followed by a colon. Let's consider a practical example. Suppose we are working with a two-dimensional array and want to search for a specific value. Once the value is found, we'd typically want to break out of both loops. Here's how we can do it with named loops.

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

let valueToFind = 5
searchValue: for row in matrix {
    for num in row {
        if num == valueToFind {
            print("Value \(valueToFind) found!")
            break searchValue
        }
    }
}
```

In this example, `searchValue` is the label for the outer loop. When `valueToFind` is found, `break searchValue` terminates the outer loop, preventing unnecessary

iterations.

In nested loops, it's often unclear which loop the `break` or `continue` statement is affecting. Named loops remove this ambiguity, making our code more readable and maintainable.

Functions, methods, and closures

Let's delve into the world of functions, methods, and closures in Swift.

In this chapter, we explore the sophisticated mechanisms of Swift's functions and closures, key to crafting flexible and reusable code. You'll learn about enhancing function definitions, handling complex closures, and using advanced techniques to make your functions more expressive and robust. We will also touch on how certain features can simplify your code's structure and increase its maintainability, such as using OptionSet for configuration or employing type aliases to clarify complex closures.

Designed for experienced Swift developers, this chapter aims to deepen your understanding of function and closure mechanisms. By mastering these advanced techniques in functions, methods, and closures, you will significantly enhance the flexibility and efficiency of your Swift code.

To read all 14 tips included in this chapter you need to purchase the book from:

https://books.nilcoalescing.com/swift-gems

Custom types: structs, classes, enums

In Swift, the power to define and manipulate custom types—such as structs, enums, and classes—is fundamental to building robust and efficient applications. This chapter dives into advanced techniques for defining and enhancing custom types, aimed at improving their functionality and efficiency in your applications. You'll explore how to make your types more expressive, such as by customizing their string representations or enabling initialization from literal values, which makes them as straightforward to use as built-in types.

We'll cover key strategies for balancing custom behavior with automated features, like preserving memberwise initializers while adding custom ones. Efficiency is another major focus, with techniques like implementing copy-on-write to manage memory effectively and ensuring types use only the necessary resources.

Additionally, you'll learn how to use enumerations for modeling complex data structures and managing instances with precision, such as through factory methods or by simplifying comparisons with automatic protocol conformance.

By mastering these techniques, you'll be able to create sophisticated, efficient custom types that leverage Swift's type system, enhancing both the performance and clarity of your code.

To read all 17 tips included in this chapter you need to purchase the book from:

https://books.nilcoalescing.com/swift-gems

Advanced property management strategies

This chapter explores advanced techniques for effective property management in Swift, providing you with strategies to enhance code robustness and efficiency. It covers essential topics such as dynamic property initialization, access control, and the use of property wrappers to encapsulate behavior. You will learn to optimize property initialization to prevent resource wastage and ensure that properties maintain a consistent state through computed values and property observers.

The chapter also delves into modern Swift features like asynchronous property getters and error handling in property contexts, equipping you with the skills to handle complex scenarios and improve the maintainability of your code.

Through these insights, you will master the nuanced management of properties in Swift, enabling you to build more reliable and scalable applications.

To read all 11 tips included in this chapter you need to purchase the book from:

https://books.nilcoalescing.com/swift-gems

Protocols and generics

This chapter explores the sophisticated use of protocols and generics in Swift, which are essential for developing flexible and secure software components. You'll learn how to customize protocols for specific class types and set precise requirements for them. Additionally, you'll discover how to boost protocol capabilities with associated types and streamline your code with default implementations in protocol extensions.

These techniques enhance the simplicity and organization of your code while maintaining its adaptability. You'll also learn to build strong and organized interfaces through protocol inheritance and the use of factory methods, which simplify the creation of types that conform to protocols.

With practical examples to guide you, this chapter will arm you with the skills to effectively use protocols and generics, greatly improving the strength and ease of maintenance of your Swift applications.

To read all 10 tips included in this chapter you need to purchase the book from:

https://books.nilcoalescing.com/swift-gems

Collection transformations and optimizations

In this chapter, we'll delve into the art of manipulating and optimizing collections in Swift. From arrays to dictionaries, you'll explore a variety of techniques that enhance the functionality and performance of these fundamental data structures. We'll cover everything from sorting arrays with closures and mapping with key paths to efficiently transforming dictionary values and utilizing lazy collections for resource management.

This chapter will equip you with the skills to handle large datasets, implement type-safe operations, and ensure your collections are both efficient and easy to manage. By the end of this chapter, you'll be well-versed in leveraging Swift's powerful features to manipulate collections in a way that is both memory-efficient and optimized for performance, making your applications faster and more reliable.

To read all 19 tips included in this chapter you need to purchase the book from:

https://books.nilcoalescing.com/swift-gems

String manipulation techniques

This chapter explores foundational techniques for manipulating strings in Swift, essential for creating effective and user-friendly applications. You'll learn how to handle string indices for precise manipulations, manage multiline strings to avoid unwanted newlines, and use raw strings to simplify handling special characters.

Additionally, the chapter delves into enhancing expressiveness and functionality through custom string interpolation, allowing for more flexible and tailored text output. We also cover dynamic string concatenation techniques that adapt to various programming needs.

By mastering these string handling techniques, you will enhance the readability and maintainability of your code, ensuring that your applications can efficiently process and display text data. These practices provide the essential tools for any Swift developer aiming to build robust, navigable, and efficient text-based features in their projects.

To read all 10 tips included in this chapter you need to purchase the book from:

https://books.nilcoalescing.com/swift-gems

Asynchronous programming and error handling

This chapter delves into advanced techniques for managing asynchronous operations in Swift, a crucial aspect for building responsive and efficient applications. You'll explore how to effectively bridge traditional completion handlers with modern `async/await` syntax and execute main-actor-isolated functions within `DispatchQueue.main.async` for UI safety. We'll also discuss how to use `yield()` to allow other tasks to proceed during lengthy operations and implement task cancellation mechanisms to stop unnecessary work, optimizing resource usage.

Additionally, you'll learn how to manage the execution of concurrent tasks with priorities and delay tasks using modern clock APIs. Handling and refining error management in asynchronous code is also covered, including strategies for rethrowing errors with added context and defining the scope of `try` explicitly. Finally, we'll look at transforming and converting the results of asynchronous operations, enhancing error handling and the usability of the `Result` type.

These advanced strategies will provide you with the tools to write cleaner, more robust asynchronous code, improving the performance and reliability of your Swift applications.

To read all 19 tips included in this chapter you need to purchase the book from:

https://books.nilcoalescing.com/swift-gems

Logging and debugging

This chapter dives into essential debugging and logging techniques in Swift, focusing on tools and practices that enhance error detection and program analysis. You'll learn how to use assertions to catch logical errors early in the development process and enrich debug logs with contextual information for clearer insights. We'll explore how to implement custom nil-coalescing in debug prints and customize output with `CustomDebugStringConvertible` for more informative debugging. Additionally, we'll look into techniques for introspecting properties and values at runtime, along with utilizing the `dump()` function for in-depth analysis.

These strategies will equip you to effectively debug and refine your Swift applications, ensuring they run smoothly and efficiently.

To read all 5 tips included in this chapter you need to purchase the book from:

https://books.nilcoalescing.com/swift-gems

Code organization

This chapter explores effective strategies for organizing code in Swift to enhance readability and maintainability. Learn how to manage shared resources, encapsulate utilities, and use enums as namespaces to keep your codebase clean and modular. We'll also touch on important practices for maintaining code quality and adaptability, such as highlighting code for review and managing framework compatibility.

These techniques are essential for building a structured, navigable, and efficient codebase in your Swift projects.

To read all 8 tips included in this chapter you need to purchase the book from:

https://books.nilcoalescing.com/swift-gems