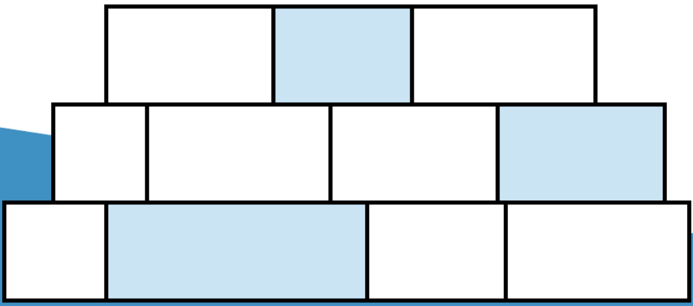
# SwiftUI Fundamentals

The essential guide to SwiftUI core concepts and APIs

Natalia Panferova

2025



# Preface

Welcome to SwiftUI Fundamentals!

This book distills the core principles and foundational concepts behind SwiftUI, the modern UI framework for building apps across Apple platforms. My goal is to help you go beyond surface-level understanding and develop a deep, practical knowledge of how SwiftUI works under the hood.

SwiftUI provides a simple, declarative way to build user interfaces, but behind that simplicity is a carefully designed architecture that can be tricky to grasp without the right perspective. In this book, we'll explore the key APIs and design patterns that power SwiftUI, giving you a solid foundation to write more efficient, maintainable, and expressive code.

I've been working with SwiftUI since it was first released and later had the privilege of contributing to its development as a member of the core SwiftUI team at Apple, where I helped design and build some of the framework's most widely used APIs. I wrote this book to share the insights I gained during that time, insights that will help you better understand how SwiftUI is built and how you can use it more effectively in your own projects.

This is not a beginner's tutorial. It's written for developers who have already experimented with SwiftUI and have a working knowledge of the Swift language. The examples here are designed to demonstrate how the framework's principles apply in real-world scenarios, equipping you with the understanding you need to solve problems independently and confidently.

SwiftUI is evolving every year, and having a solid grasp of its core principles is the best way to stay ahead. By the end of this book, you'll not only know how to use SwiftUI, you'll understand why it works the way it does.

# Text and localization

SwiftUI provides powerful tools for displaying and adapting text in user interfaces. The `Text` view is at the core of this system, handling everything from simple labels to dynamically formatted content. Combined with SwiftUI's automatic support for localization, it ensures that user-facing text adapts seamlessly to different languages, regions, and device settings.

This chapter examines how the `Text` view resolves its content and adapts to its environment. We'll explore the nuances of text initializers, how localization impacts text rendering, and ways to use formatting and styling to create clear and dynamic text-based interfaces.

## Text initializers and contextual behavior

When we talk about text in SwiftUI, we might immediately think of the `Text` view, which explicitly displays non-editable text to the user. However, many built-in SwiftUI components also include text elements. When we initialize common views and controls, we often provide a label, a string, or a string literal. Passing a string is simply a convenience for creating a `Text` label for the control. Since these labels are `Text` views internally, all styling and localization techniques that apply to standalone `Text` views also apply to control labels.

Like all SwiftUI views, `Text` is a struct. It stores information internally and resolves that information into a string at render time. The final string that appears on the screen depends on how we create the text and the context in which it is placed.

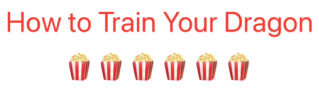
For example, if we initialize a `Text` view and assign to a variable, it remains unresolved until it's added to the view hierarchy.

```
let movieTitle = Text("How to Train Your Dragon")
```

To be rendered as a string on screen, `Text` requires an environment. Some of this environment may be defined by developers using view modifiers and environment values, while other parts are determined by SwiftUI itself based on the text's placement. Device settings, such as dark mode, accessibility text size, and locale, will also impact how `Text` is resolved.

SwiftUI automatically adapts text styling based on the environment it inherits. For example, inserting the `movieTitle` text into a `VStack` and applying a red foreground style to the hierarchy results in it being displayed in red.

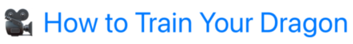
```
VStack(spacing: 12) {
    movieTitle
    Text("🍿🍿🍿🍿🍿🍿")
}
.foregroundStyle(.red)
```



*Text reading 'How to Train Your Dragon' in red, displayed above six popcorn emoji*

However, the same `movieTitle` text behaves differently when placed inside a `Link`. If no explicit foreground style is applied, SwiftUI automatically styles it using the default link color.

```
Link(destination: trailerURL) {
    HStack {
        Text("🎥")
        movieTitle
    }
}
```



*A movie camera emoji next to the text 'How to Train Your Dragon' in blue*

Even the actual string that is displayed can change based on the localization settings of the device.

## Choosing the right Text initializer

So far, we know that the `Text` struct stores data and resolves it into a string at render time. The way this data is stored directly impacts how it is rendered. SwiftUI provides multiple initializers for `Text`, it can be created from a string literal or variable, a date with a specified format, an image, or an attributed string. Choosing the right initializer helps prevent unexpected behavior.

One of the most subtle but crucial differences in text initialization is between using a string variable and a string literal. Consider the following:

```
let favoriteMovie = "The Lion King"
Text(favoriteMovie)

Text("The Lion King")
```

These two examples seem identical, but SwiftUI treats them differently. When `Text` is initialized with a string variable, the framework calls the generic initializer that accepts a value conforming to `StringProtocol`. This means SwiftUI stores the text as a standard Swift `String`. However, when `Text` is initialized with a string literal, SwiftUI uses the `LocalizedStringKey` initializer. The `LocalizedStringKey` type conforms to `ExpressibleByStringLiteral`, and it takes priority over the `StringProtocol` initializer. SwiftUI assumes that string literals should be localized.

The path SwiftUI follows to resolve text into a string differs based on the initializer used. If `Text` stores a `String`, the framework displays it as-is without looking for localization. If `Text` stores a `LocalizedStringKey`, SwiftUI searches for a translation in the main bundle. If it doesn't find the appropriate localization, it falls back to displaying the key itself.

If we are not localizing the app, we might not notice the difference in behavior immediately. However, understanding this distinction is crucial to avoid surprises, as `LocalizedStringKey` has additional functionality. For example, `LocalizedStringKey` supports interpolating more types, such as images and dates, and also parses Markdown, enabling richer text rendering directly within SwiftUI.

## Localization in SwiftUI

Localization ensures that user-facing text adapts to the user's language, region, and culture. SwiftUI simplifies this process by automatically treating string literals in many built-in components, such as `Text`, `Label`, `Button`, and others, as localizable. These components use `LocalizedStringKey` for string literals, ensuring translations are applied at runtime based on the user's locale without requiring additional configuration in most cases.

```
Button(
    "Watch trailer", // LocalizedStringKey
    action: playTrailer
)
```

### Xcode integration

When the project is built, Xcode automatically includes localizable strings from SwiftUI views in the string catalog, provided one is present in the project. The string catalog offers a centralized and efficient way to manage translations and organize all user-facing text.

SwiftUI supports adding comments to provide translators with additional context. These comments can be included when initializing `Text`, ensuring translations are accurate and aligned with the intended usage.

```
Text(
    "Showtimes",
    comment: "Header for the list of movie showtimes"
)
```

If we need to provide a comment for the label of another SwiftUI component, such as a button, we can use the initializer that accepts a view instead of the convenience one that takes a string literal directly.

```
Button(action: displayShowtimes) {
    Text(
        "View showtimes",
        comment: "Button to display showtimes"
    )
}
```

When exporting the localization catalog, Xcode includes the comment alongside the string, ensuring that translators have all the necessary information.

For larger projects, translations can be organized by splitting them into multiple catalogs. We can reference specific catalogs using the `tableName` parameter, which helps maintain scalability and clarity in projects with numerous localizable strings.

```
Text("Explore movies", tableName: "Navigation")
```

For advanced scenarios, such as strings defined in external frameworks or modules, we can specify additional parameters like bundle to control where the system looks up translations.

```
Text("Cast & Crew", bundle: Bundle(for: MovieDetails.self))
```

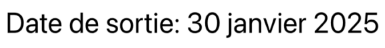
## String interpolation

SwiftUI allows us to include variables within text using string interpolation. This makes it easy to combine dynamic content, such as dates and numbers, with static text while ensuring the entire string is properly localized.

```
Text("Release date: \(movie.releaseDate, style: .date)")
```

When we embed variables into a `Text` view, SwiftUI automatically converts them into format specifiers in the exported localizable strings files and catalog. The resulting key for the example above would be `Release date: %@`.

At runtime, SwiftUI replaces the placeholder with the appropriately formatted value, such as in the French translation shown below, ensuring the string adapts to the user's locale.

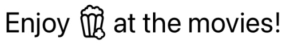

Date de sortie: 30 janvier 2025

*French text displaying 'Date de sortie: 30 janvier 2025'*

Localized string interpolation in SwiftUI supports a wide range of types, including numeric values, Foundation types, and even SwiftUI components like `Text` and `Image`.

```
Text("Enjoy \(Image(systemName: "popcorn")) at the movies!")
```

In this example, the `Image` is embedded directly into the `Text` view. At runtime, the image appears inline with the text, maintaining the visual and contextual consistency of the string.



*Text reading 'Enjoy popcorn at the movies' with a popcorn icon displayed inline with the text*

## Explicitly localizing strings

For strings that are not directly tied to SwiftUI views, such as those used in custom models or general Swift structures, we can use the `String(localized:)` initializer. This explicitly marks the string as localizable and ensures it is included in the localization catalog.

```
let movieTitle = String(
    localized: "Spirited Away",
    comment: "Title of the featured movie in the app"
)
```

This initializer also allows us to provide comments for translators, similar to how comments are added in SwiftUI `Text` views.

By leveraging SwiftUI's built-in localization features and seamless integration with Xcode's string catalog, we can efficiently adapt our apps to different languages and regions, providing users with a tailored and intuitive experience.

## Text formatting

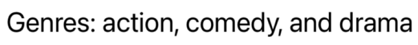
SwiftUI makes it simple to format and present a wide range of dynamic content in `Text` views. Whether displaying lists, dates, measurements, or other types of information, SwiftUI provides flexible formatting options that adapt naturally to the user's locale and context. This ensures that text remains clear, readable, and appropriate across different regions and languages.

### Format styles

SwiftUI's `Text` views provide flexible options for formatting data using the format parameter, which takes a `FormatStyle`. This enables us to present structured data directly within a text view, adapting its appearance to suit different contexts and locales.

For instance, an array can be formatted as a grammatically correct list.

```swift
let genres = ["action", "comedy", "drama"]
Text("Genres: \(genres, format: .list(type: .and))")
```



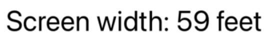
Genres: action, comedy, and drama

*Text displaying 'Genres: action, comedy, and drama' formatted as a grammatically correct list*

The list format style will adjust automatically when the array is modified, ensuring the text is updated appropriately as items are added or removed.

Measurements, such as distances or dimensions, can be displayed in a format appropriate for the user's locale. SwiftUI handles unit conversions seamlessly, presenting values in units familiar to the user.

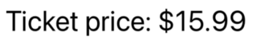
```swift
let screenWidth = Measurement(
    value: 18, unit: UnitLength.meters
)
Text("""
Screen width: \(
    screenWidth,
    format: .measurement(width: .wide)
)
""")
```


Screen width: 59 feet

*Text displaying 'Screen width: 59 feet' with a measurement formatted for the user's locale*

Numeric values, such as prices or totals, can be formatted to include a currency symbol and displayed in the appropriate monetary style.

```swift
let price = 15.99
Text("""
Ticket price: \(price, format: .currency(code: "USD"))
""")
```


Ticket price: $15.99

*Text displaying 'Ticket price: $15.99' formatted with a currency symbol*

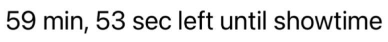
By using `FormatStyle`, we can manage how data is presented in `Text` views with clarity and precision, tailoring the output to suit different requirements and regional conventions.

## Dynamic dates

`Text` view allows us to display dates and times that update dynamically as time passes. By interpolating a date with styles like `relative`, `offset`, or `timer`, we can present time-sensitive information that remains accurate without additional code.

For instance, to show the remaining time until a specific event, we can use the relative style. The `Text` view will automatically calculate the difference between the current date and the target date, and update the displayed value in real time.

```swift
Text("\(showtime, style: .relative) left until showtime")
```


59 min, 53 sec left until showtime

*Text displaying '59 min, 31 sec left until showtime'*

By default, digits in the text use proportional widths, meaning their spacing can vary depending on the numbers displayed. This might cause the text to shift slightly as it updates. To avoid this, we can apply the `monospacedDigit()` modifier to ensure all numeric characters occupy the same width, while leaving other characters unaffected.

```
Text("\(showtime, style: .relative) left until showtime")
    .monospacedDigit()
```

This approach ensures the text remains visually stable as the numeric values change, improving the overall appearance of the interface.

## Plurals

SwiftUI leverages the Foundation framework's automatic grammar agreement feature to simplify handling plurals and reduce the need for extensive localization strings. This ensures that text follows grammatical rules, such as pluralization, without requiring additional logic in our code.

To automatically adjust text for plural values, we can use the inflection rule and define the scope of the adjustment with the following syntax:

```
Text("^[\(ticketsSold) ticket](inflect: true) sold")
```

SwiftUI parses this syntax as a custom Markdown attribute and applies the appropriate pluralization using Foundation's grammar engine. For instance, if `ticketsSold` is `1`, the text will display "1 ticket sold". When the value increases to `2` or more, it will adjust to "2 tickets sold", "3 tickets sold", etc., ensuring grammatical accuracy.

2 tickets sold

*Text displaying '2 tickets sold'*

The grammar engine supports multiple languages, applying the correct pluralization rules for each. Initially introduced in iOS 15 with support for English and Spanish, it has since been expanded. As of iOS 18, it includes languages such as German, French, Italian, Portuguese, Hindi, and Korean, making it a versatile solution for multilingual apps.

## Styling Text views

SwiftUI provides several approaches to styling text, allowing us to create visually rich and dynamic content. By leveraging view modifiers, text-specific modifiers, Markdown, or attributed strings, we can tailor text to fit a variety of design requirements.

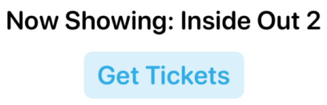
### View modifiers

We can style text in SwiftUI using view modifiers, such as `font(_:)`, `foregroundStyle(_:)`, and many others. These modifiers can be applied directly to a `Text` view to customize its appearance or to a container view to apply styling across multiple text elements.

Since string labels in SwiftUI controls, such as buttons, are internally represented as `Text`, these modifiers also apply to them. This approach simplifies styling and maintains visual consistency within the interface.

```
VStack {
    Text("Now Showing: Inside Out 2")
    Button("Get Tickets", action: purchaseTickets)
        .buttonStyle(.bordered)
}
.font(.headline)
.fontDesign(.rounded)
```

Here, both the standalone `Text` and the button label adopt the headline font with a rounded design, creating a unified appearance for all text elements in the container.



*Text and a button styled with a headline font and rounded font design*

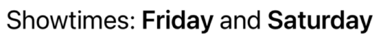
## Text modifiers

In addition to standard view modifiers, which can be applied anywhere in the view hierarchy, SwiftUI provides text-specific modifiers tailored for styling individual `Text` views. Unlike view modifiers, which apply to any SwiftUI view and return `some View`, text modifiers return another `Text` instance when applied directly to `Text`. This distinction allows text modifiers to target specific parts of a string, making them particularly effective when working with interpolated or concatenated text.

Text modifiers are ideal for emphasizing particular words or phrases within a sentence. By using interpolation, we can apply styles to individual segments while leaving the surrounding text unchanged.

```
Text("""
Showtimes: \(Text("Friday").bold()) \
and \(Text("Saturday").bold())
""")
```

In this example, “Friday” and “Saturday” are styled in bold, visually distinguishing them from the rest of the text, which retains the default font weight.

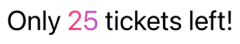

Showtimes: **Friday** and **Saturday**

*Text reading ‘Showtimes: Friday and Saturday’ with ‘Friday’ and ‘Saturday’ styled in bold*

Modifiers like `foregroundStyle(_:)` can also be applied inline to add custom styles, such as colors or gradients.

```
Text("""
Only \(
    Text("25")
        .foregroundStyle(
            .linearGradient(
                colors: [.pink, .purple],
                startPoint: .leading,
                endPoint: .trailing
            )
        )
) tickets left!
""")
```

Here, the number `25` is displayed in a pink and purple gradient, drawing attention to the numeric value while keeping the rest of the text un-styled.


Only 25 tickets left!

*Text reading 'Only 25 tickets left!' with the number 25 styled in a pink and purple gradient*

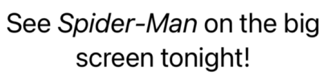
Text interpolation combined with text modifiers provides precise control over formatting, enabling selective styling of text content. This flexibility is invaluable when emphasizing specific details and creating visually engaging text layouts.

## Markdown

SwiftUI supports Markdown formatting within text views when `Text` is initialized from a string literal. This feature allows for easy application of text styles such as bold, italic, strikethrough, and monospace. By using Markdown, we can directly embed these styles into the text, eliminating the need for additional modifiers and simplifying the styling process.

Here is how we can emphasize a word with italic style:

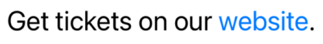
```
Text("See *Spider-Man* on the big screen tonight!")
```


See *Spider-Man* on the big
screen tonight!

*Text saying 'See Spider-Man on the big screen tonight!' with the title Spider-Man in italics*

We can also embed links directly in text, and SwiftUI will automatically render them as interactive elements. When a URL is included in a string literal using Markdown syntax, the `Text` view detects it and makes it tappable without requiring additional code to handle interactions.

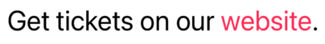
```
Text("Get tickets on our [website](https://example.com).")
```


Get tickets on our website.

*Text saying 'Get tickets on our website' with the word 'website' styled as a blue clickable link*

The link adopts the default system styling for interactive text. If necessary, the `tint(_:)` modifier can be used to adjust its color.

```
Text("Get tickets on our [website](https://example.com).")
    .tint(.pink)
```


Get tickets on our website.

*Text saying 'Get tickets on our website' with the word 'website' styled as a pink clickable link*

While SwiftUI supports inline Markdown styles and links, it does not recognize paragraph-level formatting such as headers or code blocks. Markdown in `Text` is best suited for simple formatting and inline enhancements.

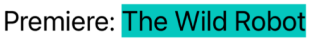
## Attributed strings

Another way to style portions of text in SwiftUI is by using the `AttributedString` APIs. These provide a modern and strictly typed way to apply attributes to text.

We can define an `AttributedString` and pass it to a `Text` view to style specific parts of the text. For instance, we can set a background color and a foreground color for a substring, and SwiftUI will render it accordingly.

```
struct PremiereView: View {
    var attrString: AttributedString {
        var attrString = AttributedString(
            "Premiere: The Wild Robot"
        )
        if let range = attrString.range(
            of: "The Wild Robot"
        ) {
            attrString[range].backgroundColor = .mint
            attrString[range].foregroundColor = .black
        }
        return attrString
    }

    var body: some View {
        Text(attrString)
    }
}
```


Premiere: The Wild Robot

*Text saying 'Premiere: The Wild Robot' with 'The Wild Robot' highlighted in mint green*

Attributes for `AttributedString` are organized into scopes, with each UI framework, such as UIKit, AppKit, and SwiftUI, defining its own. The SwiftUI scope includes attributes such as `foregroundColor`, `backgroundColor`, `font`, `kern`, `tracking`, `underlineStyle`, `strikethroughStyle`, and `baselin-eOffset`. Any attributes not supported by SwiftUI are ignored when a `Text` view renders an `AttributedString`.

SwiftUI also recognizes some Foundation attributes, such as links and inline presentation intent. Accessibility attributes are included as well, enabling customization for assistive technologies to improve the user experience.