Structure

Text

A view that displays one or more lines of read-only text.

iOS 13.0+ | iPadOS 13.0+ | Mac Catalyst 13.0+ | macOS 10.15+ | tvOS 13.0+ | visionOS 1.0+ | watchOS 6.0+

```
@frozen
struct Text
```

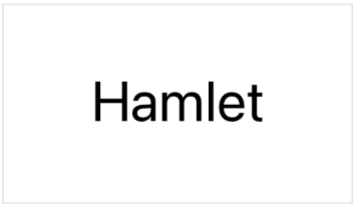
Mentioned in

- Configuring views
- Building layouts with stack views
- Declaring a custom view
- Laying out a simple view
- Displaying data in lists

Overview

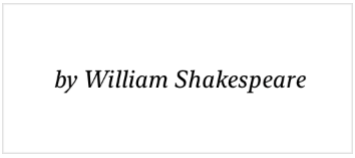
A text view draws a string in your app's user interface using a body font that's appropriate for the current platform. You can choose a different standard font, like `title` or `caption`, using the `font(_:)` view modifier.

```
Text("Hamlet")
    .font(.title)
```


Hamlet

If you need finer control over the styling of the text, you can use the same modifier to configure a system font or choose a custom font. You can also apply view modifiers like `bold()` or `italic()` to further adjust the formatting.

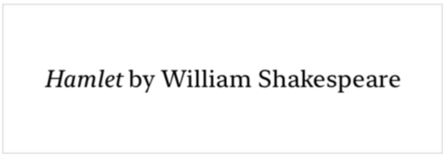
```
Text("by William Shakespeare")
    .font(.system(size: 12, weight: .light, design: .serif))
    .italic()
```


by William Shakespeare

To apply styling within specific portions of the text, you can create the text view from an `AttributedString`, which in turn allows you to use Markdown to style runs of text. You can mix string attributes and SwiftUI modifiers, with the string attributes taking priority.

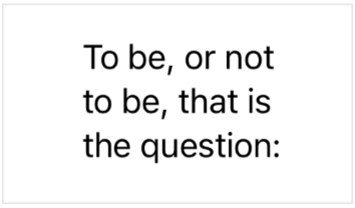
```
let attributedString = try! AttributedString(
    markdown: "_Hamlet_ by William Shakespeare")

var body: some View {
    Text(attributedString)
        .font(.system(size: 12, weight: .light, design: .serif))
}
```


Hamlet by William Shakespeare

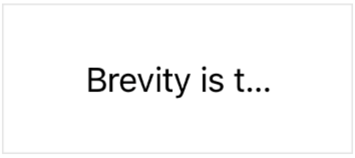
A text view always uses exactly the amount of space it needs to display its rendered contents, but you can affect the view's layout. For example, you can use the `frame(width:height:alignment:)` modifier to propose specific dimensions to the view. If the view accepts the proposal but the text doesn't fit into the available space, the view uses a combination of wrapping, tightening, scaling, and truncation to make it fit. With a width of `100` points but no constraint on the height, a text view might wrap a long string:

```
Text("To be, or not to be, that is the question:")
    .frame(width: 100)
```


To be, or not
to be, that is
the question:

Use modifiers like `lineLimit(_:)`, `allowsTightening(_:)`, `minimumScaleFactor(_:)`, and `truncationMode(_:)` to configure how the view handles space constraints. For example, combining a fixed width and a line limit of 1 results in truncation for text that doesn't fit in that space:

```
Text("Brevity is the soul of wit.")
    .frame(width: 100)
    .lineLimit(1)
```


Brevity is t...

Localizing strings

If you initialize a text view with a string literal, the view uses the `init(_:tableName:bundle:comment:)` initializer, which interprets the string as a localization key and searches for the key in the table you specify, or in the default table if you don't specify one.

```
Text("pencil") // Searches the default table in the main bundle.
```

For an app localized in both English and Spanish, the above view displays "pencil" and "lápiz" for English and Spanish users, respectively. If the view can't perform localization, it displays the key instead. For example, if the same app lacks Danish localization, the view displays "pencil" for users in that locale. Similarly, an app that lacks any localization information displays "pencil" in any locale.

To explicitly bypass localization for a string literal, use the `init(verbatim:)` initializer.

```
Text(verbatim: "pencil") // Displays the string "pencil" in any locale.
```

If you initialize a text view with a variable value, the view uses the `init(_:)` initializer, which doesn't localize the string. However, you can request localization by creating a `LocalizedStringKey` instance first, which triggers the `init(_:tableName:bundle:comment:)` initializer instead:

```
// Don't localize a string variable...
Text(writingImplement)

// ...unless you explicitly convert it to a localized string key.
Text(LocalizedStringKey(writingImplement))
```

When localizing a string variable, you can use the default table by omitting the optional initialization parameters — as in the above example — just like you might for a string literal.

When composing a complex string, where there is a need to assemble multiple pieces of text, use string interpolation:

```
let name: String = //…
Text("Hello, \(name)")
```

This would look up the `"Hello, %@"` localization key in the localized string file and replace the format specifier `%@` with the value of `name` before rendering the text on screen.

Using string interpolation ensures that the text in your app can be localized correctly in all locales, especially in right-to-left languages.

If you desire to style only parts of interpolated text while ensuring that the content can still be localized correctly, interpolate `Text` or `AttributedString`:

```
let name = Text(person.name).bold()
Text("Hello, \(name)")
```

The example above uses `appendInterpolation(_:)` and will look up the `"Hello, %@"` in the localized string file and interpolate a bold text rendering the value of `name`.

Using `appendInterpolation(_:)` you can interpolate `Image` in text.

Topics

Creating a text view

`init(LocalizedStringKey, tableName: String?, bundle: Bundle?, comment: StaticString?)`

Creates a text view that displays localized content identified by a key.

`init(_:)`

Creates a text view that displays styled attributed content.

`init(verbatim: String)`

Creates a text view that displays a string literal without localization.

`init(Date, style: Text.DateStyle)`

Creates an instance that displays localized dates and times using a specific style.

`init(_:format:)`

Creates a text view that displays the formatted representation of a nonstring type supported by a corresponding format style.

`init(_:formatter:)`

Creates a text view that displays the formatted representation of a Foundation object.

`init(timerInterval: ClosedRange<Date>, pauseTime: Date?, countsDown: Bool, showsHours: Bool)`

Creates an instance that displays a timer counting within the provided interval.

Choosing a font

`func font(Font?) -> Text`

Sets the default font for text in the view.

`func fontWeight(Font.Weight?) -> Text`

Sets the font weight of the text.

`func fontDesign(Font.Design?) -> Text`

Sets the font design of the text.

`func fontWidth(Font.Width?) -> Text`

Sets the font width of the text.

Styling the view’s text

`func foregroundStyle<S>(S) -> Text`

Sets the style of the text displayed by this view.

```
func bold() -> Text
```

Applies a bold or emphasized treatment to the fonts of the text.

```
func bold(Bool) -> Text
```

Applies a bold font weight to the text.

```
func italic() -> Text
```

Applies italics to the text.

```
func italic(Bool) -> Text
```

Applies italics to the text.

```
func strikethrough(Bool, color: Color?) -> Text
```

Applies a strikethrough to the text.

```
func strikethrough(Bool, pattern: Text.LineStyle.Pattern, color: Color
?) -> Text
```

Applies a strikethrough to the text.

```
func underline(Bool, color: Color?) -> Text
```

Applies an underline to the text.

```
func underline(Bool, pattern: Text.LineStyle.Pattern, color: Color?) ->
Text
```

Applies an underline to the text.

```
func monospaced(Bool) -> Text
```

Modifies the font of the text to use the fixed-width variant of the current font, if possible.

```
func monospacedDigit() -> Text
```

Modifies the text view’s font to use fixed-width digits, while leaving other characters proportionally spaced.

```
func kerning(CGFloat) -> Text
```

Sets the spacing, or kerning, between characters.

```
func tracking(CGFloat) -> Text
```

Sets the tracking for the text.

```
func baselineOffset(CGFloat) -> Text
```

Sets the vertical offset for the text relative to its baseline.

`enum Case`

A scheme for transforming the capitalization of characters within text.

`struct DateStyle`

A predefined style used to display a `Date`.

`struct LineStyle`

Description of the style used to draw the line for `StrikethroughStyleAttribute` and `UnderlineStyleAttribute`.

Fitting text into available space

`func textScale(Text.Scale, isEnabled: Bool) -> Text`

Applies a text scale to the text.

`struct Scale`

Defines text scales

`enum TruncationMode`

The type of truncation to apply to a line of text when it's too long to fit in the available space.

Localizing text

`func typesettingLanguage(_:isEnabled:)`

Specifies the language for typesetting.

Configuring voiceover

`func speechAdjustedPitch(Double) -> Text`

Raises or lowers the pitch of spoken text.

`func speechAlwaysIncludesPunctuation(Bool) -> Text`

Sets whether VoiceOver should always speak all punctuation in the text view.

`func speechAnnouncementsQueued(Bool) -> Text`

Controls whether to queue pending announcements behind existing speech rather than interrupting speech in progress.

`func speechSpellsOutCharacters(Bool) -> Text`

Sets whether VoiceOver should speak the contents of the text view character by character.

Providing accessibility information

`func accessibilityHeading(AccessibilityHeadingLevel) -> Text`

Sets the accessibility level of this heading.

`func accessibilityLabel(_:)`

Adds a label to the view that describes its contents.

`func accessibilityTextContentType(AccessibilityTextContentType) -> Text`

Sets an accessibility text content type.

Combining text views

~~`static func + (Text, Text) -> Text`~~

Concatenates the text in two text views in a new text view.

Deprecated

Deprecated symbols

~~`func foregroundColor(Color?) -> Text`~~

Sets the color of the text displayed by this view.

Deprecated

Structures

`struct AlignmentStrategy`

The way SwiftUI infers the appropriate text alignment if no value is explicitly provided.

`struct Layout`

A value describing the layout and custom attributes of a tree of `Text` views.

`struct LayoutKey`

A preference key that provides the `Text.Layout` values for all text views in the queried subtree.

`struct WritingDirectionStrategy`

The way SwiftUI infers the appropriate writing direction if no value is explicitly provided.

Instance Methods

`func customAttribute<T>(T) -> Text`

Adds a custom attribute to the text view.

`func textVariant<V>(V) -> some View`

Controls the way text size variants are chosen.

Relationships

Conforms To

`Copyable`
`Equatable`
`Sendable`
`SendableMetatype`
`View`

See Also

Displaying text

`struct Label`

A standard label for user interface items, consisting of an icon with a title.

`func labelStyle<S>(S) -> some View`

Sets the style for labels within this view.