Swift / String

Structure

String

A Unicode string value that is a collection of characters.

iOS 8.0+ | iPadOS 8.0+ | Mac Catalyst 13.0+ | macOS 10.10+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
@frozen
struct String
```

Overview

A string is a series of characters, such as `"Swift"`, that forms a collection. Strings in Swift are Unicode correct and locale insensitive, and are designed to be efficient. The `String` type bridges with the Objective-C class `NSString` and offers interoperability with C functions that works with strings.

You can create new strings using string literals or string interpolations. A *string literal* is a series of characters enclosed in quotes.

```
let greeting = "Welcome!"
```

String interpolations are string literals that evaluate any included expressions and convert the results to string form. String interpolations give you an easy way to build a string from multiple pieces. Wrap each expression in a string interpolation in parentheses, prefixed by a backslash.

```
let name = "Rosa"
let personalizedGreeting = "Welcome, \(name)!"
// personalizedGreeting == "Welcome, Rosa!"

let price = 2
let number = 3
let cookiePrice = "\(number) cookies: $\(price * number)."
// cookiePrice == "3 cookies: $6."
```

Combine strings using the concatenation operator (+).

```
let longerGreeting = greeting + " We're glad you're here!"
// longerGreeting == "Welcome! We're glad you're here!"
```

Multiline string literals are enclosed in three double quotation marks ("""), with each delimiter on its own line. Indentation is stripped from each line of a multiline string literal to match the indentation of the closing delimiter.

```
let banner = """
          __,
         (           o  /) _/_
          `.  , , , ,  //  /
        (___)(_(_/_(_ //_ (__
                     /)
                    (/
        """
```

Modifying and Comparing Strings

Strings always have value semantics. Modifying a copy of a string leaves the original unaffected.

```
var otherGreeting = greeting
otherGreeting += " Have a nice time!"
// otherGreeting == "Welcome! Have a nice time!"

print(greeting)
// Prints "Welcome!"
```

Comparing strings for equality using the equal-to operator (==) or a relational operator (like < or >=) is always performed using Unicode canonical representation. As a result, different representations of a string compare as being equal.

```
let cafe1 = "Cafe\u{301}"
let cafe2 = "Café"
```

```
print(cafe1 == cafe2)
// Prints "true"
```

The Unicode scalar value `"\u{301}"` modifies the preceding character to include an accent, so `"e\u{301}"` has the same canonical representation as the single Unicode scalar value `"é"`.

Basic string operations are not sensitive to locale settings, ensuring that string comparisons and other operations always have a single, stable result, allowing strings to be used as keys in `Dictionary` instances and for other purposes.

Accessing String Elements

A string is a collection of *extended grapheme clusters*, which approximate human-readable characters. Many individual characters, such as "é", "김", and "🇮🇳", can be made up of multiple Unicode scalar values. These scalar values are combined by Unicode's boundary algorithms into extended grapheme clusters, represented by the Swift `Character` type. Each element of a string is represented by a `Character` instance.

For example, to retrieve the first word of a longer string, you can search for a space and then create a substring from a prefix of the string up to that point:

```
let name = "Marie Curie"
let firstSpace = name.firstIndex(of: " ") ?? name.endIndex
let firstName = name[..<firstSpace]
// firstName == "Marie"
```

The `firstName` constant is an instance of the `Substring` type—a type that represents substrings of a string while sharing the original string's storage. Substrings present the same interface as strings.

```
print("\(name)'s first name has \(firstName.count) letters.")
// Prints "Marie Curie's first name has 5 letters."
```

Accessing a String's Unicode Representation

If you need to access the contents of a string as encoded in different Unicode encodings, use one of the string's `unicodeScalars`, `utf16`, or `utf8` properties. Each property provides access to a view of the string as a series of code units, each encoded in a different Unicode encoding.

To demonstrate the different views available for every string, the following examples use this `String` instance:

```
let cafe = "Cafe\u{301} du 🌍"
print(cafe)
// Prints "Café du 🌍"
```

The `cafe` string is a collection of the nine characters that are visible when the string is displayed.

```
print(cafe.count)
// Prints "9"
print(Array(cafe))
// Prints "["C", "a", "f", "é", " ", "d", "u", " ", "🌍"]"
```

Unicode Scalar View

A string's `unicodeScalars` property is a collection of Unicode scalar values, the 21-bit codes that are the basic unit of Unicode. Each scalar value is represented by a `Unicode.Scalar` instance and is equivalent to a UTF-32 code unit.

```
print(cafe.unicodeScalars.count)
// Prints "10"
print(Array(cafe.unicodeScalars))
// Prints "["C", "a", "f", "e", "\u{0301}", " ", "d", "u", " ", "\u{0001F30D}"]"
print(cafe.unicodeScalars.map { $0.value })
// Prints "[67, 97, 102, 101, 769, 32, 100, 117, 32, 127757]"
```

The `unicodeScalars` view's elements comprise each Unicode scalar value in the `cafe` string. In particular, because `cafe` was declared using the decomposed form of the `"é"` character, `unicodeScalars` contains the scalar values for both the letter `"e"` (101) and the accent character `"´"` (769).

UTF-16 View

A string's `utf16` property is a collection of UTF-16 code units, the 16-bit encoding form of the string's Unicode scalar values. Each code unit is stored as a `UInt16` instance.

```
print(cafe.utf16.count)
// Prints "11"
print(Array(cafe.utf16))
// Prints "[67, 97, 102, 101, 769, 32, 100, 117, 32, 55356, 57101]"
```

The elements of the `utf16` view are the code units for the string when encoded in UTF-16. These elements match those accessed through indexed `NSString` APIs.

```
let nscafe = cafe as NSString
print(nscafe.length)
// Prints "11"
print(nscafe.character(at: 3))
// Prints "101"
```

UTF-8 View

A string's `utf8` property is a collection of UTF-8 code units, the 8-bit encoding form of the string's Unicode scalar values. Each code unit is stored as a `UInt8` instance.

```
print(cafe.utf8.count)
// Prints "14"
print(Array(cafe.utf8))
// Prints "[67, 97, 102, 101, 204, 129, 32, 100, 117, 32, 240, 159, 140, 141]"
```

The elements of the `utf8` view are the code units for the string when encoded in UTF-8. This representation matches the one used when `String` instances are passed to C APIs.

```
let cLength = strlen(cafe)
print(cLength)
// Prints "14"
```

Measuring the Length of a String

When you need to know the length of a string, you must first consider what you'll use the length for. Are you measuring the number of characters that will be displayed on the screen, or are you measuring the amount of storage needed for the string in a particular encoding? A single string can have greatly differing lengths when measured by its different views.

For example, an ASCII character like the capital letter *A* is represented by a single element in each of its four views. The Unicode scalar value of *A* is 65, which is small enough to fit in a single code unit in both UTF-16 and UTF-8.

```
let capitalA = "A"
print(capitalA.count)
// Prints "1"
print(capitalA.unicodeScalars.count)
// Prints "1"
print(capitalA.utf16.count)
// Prints "1"
print(capitalA.utf8.count)
// Prints "1"
```

On the other hand, an emoji flag character is constructed from a pair of Unicode scalar values, like `"\u{1F1F5}"` and `"\u{1F1F7}"`. Each of these scalar values, in turn, is too large to fit into a single UTF-16 or UTF-8 code unit. As a result, each view of the string "🇵🇷" reports a different length.

```
let flag = "🇵🇷"
print(flag.count)
// Prints "1"
print(flag.unicodeScalars.count)
// Prints "2"
print(flag.utf16.count)
// Prints "4"
print(flag.utf8.count)
// Prints "8"
```

To check whether a string is empty, use its `isEmpty` property instead of comparing the length of one of the views to `0`. Unlike with `isEmpty`, calculating a view's `count` property requires iterating through the elements of the string.

Accessing String View Elements

To find individual elements of a string, use the appropriate view for your task. For example, to retrieve the first word of a longer string, you can search the string for a space and then create a new string from a prefix of the string up to that point.

```swift
let name = "Marie Curie"
let firstSpace = name.firstIndex(of: " ") ?? name.endIndex
let firstName = name[..<firstSpace]
print(firstName)
// Prints "Marie"
```

Strings and their views share indices, so you can access the UTF-8 view of the `name` string using the same `firstSpace` index.

```swift
print(Array(name.utf8[..<firstSpace]))
// Prints "[77, 97, 114, 105, 101]"
```

Note that an index into one view may not have an exact corresponding position in another view. For example, the `flag` string declared above comprises a single character, but is composed of eight code units when encoded as UTF-8. The following code creates constants for the first and second positions in the `flag.utf8` view. Accessing the `utf8` view with these indices yields the first and second code UTF-8 units.

```swift
let firstCodeUnit = flag.startIndex
let secondCodeUnit = flag.utf8.index(after: firstCodeUnit)
// flag.utf8[firstCodeUnit] == 240
// flag.utf8[secondCodeUnit] == 159
```

When used to access the elements of the `flag` string itself, however, the `secondCodeUnit` index does not correspond to the position of a specific character. Instead of only accessing the specific UTF-8 code unit, that index is treated as the position of the character at the index's encoded offset. In the case of `secondCodeUnit`, that character is still the flag itself.

```swift
// flag[firstCodeUnit] == "🇵🇷"
// flag[secondCodeUnit] == "🇵🇷"
```

If you need to validate that an index from one string's view corresponds with an exact position in another view, use the index's `samePosition(in:)` method or the `init(_:within:)` initializer.

```
if let exactIndex = secondCodeUnit.samePosition(in: flag) {
    print(flag[exactIndex])
} else {
    print("No exact match for this position.")
}
// Prints "No exact match for this position."
```

Performance Optimizations

Although strings in Swift have value semantics, strings use a copy-on-write strategy to store their data in a buffer. This buffer can then be shared by different copies of a string. A string's data is only copied lazily, upon mutation, when more than one string instance is using the same buffer. Therefore, the first in any sequence of mutating operations may cost O(*n*) time and space.

When a string's contiguous storage fills up, a new buffer must be allocated and data must be moved to the new storage. String buffers use an exponential growth strategy that makes appending to a string a constant time operation when averaged over many append operations.

Bridging Between String and NSString

Any `String` instance can be bridged to `NSString` using the type-cast operator (`as`), and any `String` instance that originates in Objective-C may use an `NSString` instance as its storage. Because any arbitrary subclass of `NSString` can become a `String` instance, there are no guarantees about representation or efficiency when a `String` instance is backed by `NSString` storage. Because `NSString` is immutable, it is just as though the storage was shared by a copy. The first in any sequence of mutating operations causes elements to be copied into unique, contiguous storage which may cost O(*n*) time and space, where *n* is the length of the string's encoded representation (or more, if the underlying `NSString` has unusual performance characteristics).

For more information about the Unicode terms used in this discussion, see the Unicode.org glossary. In particular, this discussion mentions extended grapheme clusters, Unicode scalar values, and canonical equivalence.

Topics

Creating a String

In addition to creating a string from a single string literal, you can also create an empty string, a string containing an existing group of characters, or a string repeating the contents of another string.

`init(decoding: FilePath)`

Creates a string by interpreting the file path's content as UTF-8 on Unix and UTF-16 on Windows.

`init()`

Creates an empty string.

`init(Character)`

Creates a string containing the given character.

`init<S>(S)`

Creates a new string containing the characters in the given sequence.

`init<S>(S)`

Creates a new instance of a collection containing the elements of a sequence.

`init<S>(S)`

Creates a new string containing the characters in the given sequence.

`init(Substring)`

Creates a new string from the given substring.

`init(repeating: String, count: Int)`

Creates a new string representing the given string repeated the specified number of times.

`init(repeating: Character, count: Int)`

Creates a string representing the given character repeated the specified number of times.

`init(unsafeUninitializedCapacity: Int, initializingUTF8With: (UnsafeMutableBufferPointer<UInt8>) throws -> Int) rethrows`

Creates a new string with the specified capacity in UTF-8 code units, and then calls the given closure with a buffer covering the string's uninitialized memory.

Inspecting a String

`var isEmpty: Bool`

A Boolean value indicating whether a string has no characters.

`var count: Int`

The number of characters in a string.

Creating a String from Unicode Data

`init(Unicode.Scalar)`

`init?(data: Data, encoding: String.Encoding)`

Returns a `String` initialized by converting given `data` into Unicode characters using a given `encoding`.

`init?(validatingUTF8: UnsafePointer<CChar>)`

Creates a new string by copying and validating the null-terminated UTF-8 data referenced by the given pointer.

`init?<Encoding>(validating: some Sequence, as: Encoding.Type)`

Creates a new string by copying and validating the sequence of code units passed in, according to the specified encoding.

`init?<Encoding>(validating: some Sequence<Int8>, as: Encoding.Type)`

Creates a new string by copying and validating the sequence of code units passed in, according to the specified encoding.

`init?(utf8String: [CChar])`

Creates a string by copying the data from a given null-terminated array of UTF8-encoded bytes.

`init?(utf8String: UnsafePointer<CChar>)`

Creates a string by copying the data from a given null-terminated C array of UTF8-encoded bytes.

`init(utf16CodeUnits: UnsafePointer<unichar>, count: Int)`

Creates a new string that contains the specified number of characters from the given C array of Unicode characters.

~~`init(utf16CodeUnitsNoCopy: UnsafePointer<unichar>, count: Int, freeWhenDone: Bool)`~~

Creates a new string that contains the specified number of characters from the given C array of UTF-16 code units.

Deprecated

`init<C, Encoding>(decoding: C, as: Encoding.Type)`

Creates a string from the given Unicode code units in the specified encoding.

Creating a String Using Formats

`init(format: String, any CVarArg...)`

Returns a `String` object initialized by using a given format string as a template into which the remaining argument values are substituted.

`init(format: String, arguments: [any CVarArg])`

Returns a `String` object initialized by using a given format string as a template into which the remaining argument values are substituted according to the user's default locale.

`init(format: String, locale: Locale?, any CVarArg...)`

Returns a `String` object initialized by using a given format string as a template into which the remaining argument values are substituted according to given locale information.

`init(format: String, locale: Locale?, arguments: [any CVarArg])`

Returns a `String` object initialized by using a given format string as a template into which the remaining argument values are substituted according to given locale information.

`static func localizedStringWithFormat(String, any CVarArg...) -> String`

Returns a string created by using a given format string as a template into which the remaining argument values are substituted according to the user's default locale.

Creating a Localized String

`init(localized: String.LocalizationValue, table: String?, bundle: Bundle?, locale: Locale, comment: StaticString?)`

Creates a localized string from an interpolated string.

`init(localized: String.LocalizationValue, options: String.LocalizationOptions, table: String?, bundle: Bundle?, locale: Locale, comment: StaticString?)`

Creates a localized string from an interpolated string, applying the specified options.

`struct LocalizationValue`

A reference to a localizable string, with optional string interpolation.

`struct LocalizationOptions`

Options to apply when initializing a localized string.

`init(localized: StaticString, defaultValue: String.LocalizationValue, table: String?, bundle: Bundle?, locale: Locale, comment: StaticString?)`

Creates a localized string from an arbitrary static string key.

`init(localized: StaticString, defaultValue: String.LocalizationValue, options: String.LocalizationOptions, table: String?, bundle: Bundle?, locale: Locale, comment: StaticString?)`

Creates a localized string from an arbitrary static string key, applying the specified options.

`init(localized: LocalizedStringResource)`

Creates a localized string from a localized string resource.

`init(localized: LocalizedStringResource, options: String.LocalizationOptions)`

Creates a localized string from a localized string resource, applying the specified options.

Converting Numeric Values

`init<T>(T, radix: Int, uppercase: Bool)`

Creates a string representing the given value in base 10, or some other specified base.

Converting a C String

`init?<S>(bytes: S, encoding: String.Encoding)`

Creates a new string equivalent to the given bytes interpreted in the specified encoding. Note: This API does not interpret embedded nulls as termination of the string. Use `String?(validatingCString:)` instead for null-terminated C strings.

~~`init?(bytesNoCopy: UnsafeMutableRawPointer, length: Int, encoding: String.Encoding, freeWhenDone: Bool)`~~

Creates a new string that contains the specified number of bytes from the given buffer, interpreted in the specified encoding, and optionally frees the buffer.

Deprecated

`init?(validatingCString: UnsafePointer<CChar>)`

Creates a new string by copying and validating the null-terminated UTF-8 data referenced by the given pointer.

`init?(validatingCString: [CChar])`

Creates a new string by copying and validating the null-terminated UTF-8 data referenced by the given array.

`init(cString: UnsafePointer<CChar>)`

Creates a new string by copying the null-terminated UTF-8 data referenced by the given pointer.

`init(cString: UnsafePointer<UInt8>)`

Creates a new string by copying the null-terminated UTF-8 data referenced by the given pointer.

`init?(cString: [CChar], encoding: String.Encoding)`

Produces a string by copying the null-terminated bytes in a given array, interpreted according to a given encoding.

`init?(cString: UnsafePointer<CChar>, encoding: String.Encoding)`

Produces a string by copying the null-terminated bytes in a given C array, interpreted according to a given encoding.

`init<Encoding>(decodingCString: [Encoding.CodeUnit], as: Encoding.Type)`

Creates a new string by copying the null-terminated sequence of code units referenced by the given array.

```
static func decodeCString<Encoding>(UnsafePointer<Encoding.CodeUnit>?,
as: Encoding.Type, repairingInvalidCodeUnits: Bool) -> (result: String,
repairsMade: Bool)?
```

Creates a new string by copying the null-terminated data referenced by the given pointer using the specified encoding.

Converting Other Types to Strings

`init<T>(T)`

Creates an instance from the description of a given `LosslessStringConvertible` instance.

`init<Subject>(describing: Subject)`

Creates a string representing the given value.

`init<Subject>(describing: Subject)`

Creates a string representing the given value.

`init<Subject>(describing: Subject)`

Creates a string representing the given value.

`init<Subject>(describing: Subject)`

Creates a string representing the given value.

`init<Subject>(reflecting: Subject)`

Creates a string with a detailed representation of the given value, suitable for debugging.

Creating a String from a File or URL

~~`init(contentsOf: URL) throws`~~ Deprecated

`init(contentsOf: URL, encoding: String.Encoding) throws`

Produces a string created by reading data from a given URL interpreted using a given encoding.

`init(contentsOf: URL, usedEncoding: inout String.Encoding) throws`

Produces a string created by reading data from a given URL and returns by reference the encoding used to interpret the data.

~~`init(contentsOfFile: String) throws`~~ Deprecated

`init(contentsOfFile: String, encoding: String.Encoding) throws`

Produces a string created by reading data from the file at a given path interpreted using a given encoding.

`init(contentsOfFile: String, usedEncoding: inout String.Encoding) throws`

Produces a string created by reading data from the file at a given path and returns by reference the encoding used to interpret the file.

Writing to a File or URL

`func write(String)`

Appends the given string to this string.

`func write<Target>(to: inout Target)`

Writes the string into the given output stream.

Appending Strings and Characters

`func append(String)`

Appends the given string to this string.

`func append(Character)`

Appends the given character to the string.

`func append(contentsOf: String)`

`func append(contentsOf: Substring)`

`func append<S>(contentsOf: S)`

Appends the characters in the given sequence to the string.

`func append<S>(contentsOf: S)`

Adds the elements of a sequence or collection to the end of this collection.

`func reserveCapacity(Int)`

Reserves enough space in the string's underlying storage to store the specified number of ASCII characters.

`static func + (String, String) -> String`

`static func += (inout String, String)`

`static func + <Other>(Other, Self) -> Self`

Creates a new collection by concatenating the elements of a sequence and a collection.

`static func + <Other>(Self, Other) -> Self`

Creates a new collection by concatenating the elements of a collection and a sequence.

`static func + <Other>(Self, Other) -> Self`

Creates a new collection by concatenating the elements of two collections.

`static func += <Other>(inout Self, Other)`

Appends the elements of a sequence to a range-replaceable collection.

Inserting Characters

`func insert(Character, at: String.Index)`

Inserts a new character at the specified position.

`func insert(Self.Element, at: Self.Index)`

Inserts a new element into the collection at the specified position.

`func insert<C>(contentsOf: C, at: Self.Index)`

Inserts the elements of a sequence into the collection at the specified position.

`func insert<S>(contentsOf: S, at: String.Index)`

Inserts a collection of characters at the specified position.

Replacing Substrings

`func replaceSubrange<C>(Range<String.Index>, with: C)`

Replaces the text within the specified bounds with the given characters.

`func replaceSubrange<C, R>(R, with: C)`

Replaces the specified subrange of elements with the given collection.

Removing Substrings

`func remove(at: String.Index) -> Character`

Removes and returns the character at the specified position.

`func remove(at: Self.Index) -> Self.Element`

Removes and returns the element at the specified position.

`func removeAll(keepingCapacity: Bool)`

Replaces this string with the empty string.

`func removeAll(where: (Self.Element) throws -> Bool) rethrows`

Removes all the elements that satisfy the given predicate.

`func removeFirst() -> Self.Element`

Removes and returns the first element of the collection.

`func removeFirst(Int)`

Removes the specified number of elements from the beginning of the collection.

`func removeLast() -> Self.Element`

Removes and returns the last element of the collection.

`func removeLast(Int)`

Removes the specified number of elements from the end of the collection.

`func removeSubrange(Range<String.Index>)`

Removes the characters in the given range.

`func removeSubrange(Range<Self.Index>)`

Removes the elements in the specified subrange from the collection.

`func removeSubrange<R>(R)`

Removes the elements in the specified subrange from the collection.

`func filter((Self.Element) throws -> Bool) rethrows -> Self`

Returns a new collection of the same type containing, in order, the elements of the original collection that satisfy the given predicate.

`func drop(while: (Self.Element) throws -> Bool) rethrows -> Self.SubSequence`

Returns a subsequence by skipping elements while `predicate` returns `true` and returning the remaining elements.

`func dropFirst(Int) -> Self.SubSequence`

Returns a subsequence containing all but the given number of initial elements.

`func dropLast(Int) -> Self.SubSequence`

Returns a subsequence containing all but the specified number of final elements.

`func popLast() -> Self.Element?`

Removes and returns the last element of the collection.

Changing Case

`func lowercased() -> String`

Returns a lowercase version of the string.

`func uppercased() -> String`

Returns an uppercase version of the string.

Comparing Strings Using Operators

Comparing strings using the equal-to operator (==) or a relational operator (like < and >=) is always performed using the Unicode canonical representation, so that different representations of a string compare as being equal.

`static func == (String, String) -> Bool`

Returns a Boolean value indicating whether two values are equal.

`static func == <RHS>(Self, RHS) -> Bool` `static func != (Self, Self) -> Bool`

Returns a Boolean value indicating whether two values are not equal.

`static func != <RHS>(Self, RHS) -> Bool`

`static func ~= (String, Substring) -> Bool`

Comparing Characters

`func elementsEqual<OtherSequence>(OtherSequence) -> Bool`

Returns a Boolean value indicating whether this sequence and another sequence contain the same elements in the same order.

`func elementsEqual<OtherSequence>(OtherSequence, by: (Self.Element, OtherSequence.Element) throws -> Bool) rethrows -> Bool`

Returns a Boolean value indicating whether this sequence and another sequence contain equivalent elements in the same order, using the given predicate as the equivalence test.

`func starts<PossiblePrefix>(with: PossiblePrefix) -> Bool`

Returns a Boolean value indicating whether the initial elements of the sequence are the same as the elements in another sequence.

`func starts<PossiblePrefix>(with: PossiblePrefix, by: (Self.Element, PossiblePrefix.Element) throws -> Bool) rethrows -> Bool`

Returns a Boolean value indicating whether the initial elements of the sequence are equivalent to the elements in another sequence, using the given predicate as the equivalence test.

`func lexicographicallyPrecedes<OtherSequence>(OtherSequence) -> Bool`

Returns a Boolean value indicating whether the sequence precedes another sequence in a lexicographical (dictionary) ordering, using the less-than operator (<) to compare elements.

`func lexicographicallyPrecedes<OtherSequence>(OtherSequence, by: (Self.Element, Self.Element) throws -> Bool) rethrows -> Bool`

Returns a Boolean value indicating whether the sequence precedes another sequence in a lexicographical (dictionary) ordering, using the given predicate to compare elements.

Creating and Applying Differences

`func applying(CollectionDifference<Self.Element>) -> Self?`

Applies the given difference to this collection.

```
func difference<C>(from: C) -> CollectionDifference<Self.Element>
```

Returns the difference needed to produce this collection's ordered elements from the given collection.

```
func difference<C>(from: C, by: (C.Element, Self.Element) -> Bool) ->
CollectionDifference<Self.Element>
```

Returns the difference needed to produce this collection's ordered elements from the given collection, using the given predicate as an equivalence test.

Finding Substrings

```
func hasPrefix(String) -> Bool
```

```
func hasSuffix(String) -> Bool
```

Finding Characters

```
func contains(Self.Element) -> Bool
```

Returns a Boolean value indicating whether the sequence contains the given element.

```
func allSatisfy((Self.Element) throws -> Bool) rethrows -> Bool
```

Returns a Boolean value indicating whether every element of a sequence satisfies a given predicate.

```
func contains(where: (Self.Element) throws -> Bool) rethrows -> Bool
```

Returns a Boolean value indicating whether the sequence contains an element that satisfies the given predicate.

```
func first(where: (Self.Element) throws -> Bool) rethrows -> Self.
Element?
```

Returns the first element of the sequence that satisfies the given predicate.

```
func firstIndex(of: Self.Element) -> Self.Index?
```

Returns the first index where the specified value appears in the collection.

```
func firstIndex(where: (Self.Element) throws -> Bool) rethrows -> Self.
Index?
```

Returns the first index in which an element of the collection satisfies the given predicate.

```
func last(where: (Self.Element) throws -> Bool) rethrows -> Self.
Element?
```

Returns the last element of the sequence that satisfies the given predicate.

`func lastIndex(of: Self.Element) -> Self.Index?`

Returns the last index where the specified value appears in the collection.

`func lastIndex(where: (Self.Element) throws -> Bool) rethrows -> Self.Index?`

Returns the index of the last element in the collection that matches the given predicate.

`func max() -> Self.Element?`

Returns the maximum element in the sequence.

`func max<T>(T, T) -> T`

`func max(by: (Self.Element, Self.Element) throws -> Bool) rethrows -> Self.Element?`

Returns the maximum element in the sequence, using the given predicate as the comparison between elements.

`func min() -> Self.Element?`

Returns the minimum element in the sequence.

`func min<T>(T, T) -> T`

`func min(by: (Self.Element, Self.Element) throws -> Bool) rethrows -> Self.Element?`

Returns the minimum element in the sequence, using the given predicate as the comparison between elements.

Getting Substrings

`subscript(Range<String.Index>) -> Substring`

Accesses a contiguous subrange of the collection's elements.

`subscript<R>(R) -> Self.SubSequence`

Accesses the contiguous subrange of the collection's elements specified by a range expression.

`subscript((UnboundedRange_) -> ()) -> Self.SubSequence`

`func prefix(Int) -> Self.SubSequence`

Returns a subsequence, up to the specified maximum length, containing the initial elements of the collection.

`func prefix(through: Self.Index) -> Self.SubSequence`

Returns a subsequence from the start of the collection through the specified position.

`func prefix(upTo: Self.Index) -> Self.SubSequence`

Returns a subsequence from the start of the collection up to, but not including, the specified position.

`func prefix(while: (Self.Element) throws -> Bool) rethrows -> Self.SubSequence`

Returns a subsequence containing the initial elements until `predicate` returns `false` and skipping the remaining elements.

`func suffix(Int) -> Self.SubSequence`

Returns a subsequence, up to the given maximum length, containing the final elements of the collection.

`func suffix(from: Self.Index) -> Self.SubSequence`

Returns a subsequence from the specified position to the end of the collection.

Splitting a String

`func split(separator: Self.Element, maxSplits: Int, omittingEmptySubsequences: Bool) -> [Self.SubSequence]`

Returns the longest possible subsequences of the collection, in order, around elements equal to the given element.

`func split(maxSplits: Int, omittingEmptySubsequences: Bool, whereSeparator: (Self.Element) throws -> Bool) rethrows -> [Self.SubSequence]`

Returns the longest possible subsequences of the collection, in order, that don't contain elements satisfying the given predicate.

Getting Characters and Bytes

`subscript(String.Index) -> Character`

Accesses the character at the given position.

`var first: Self.Element?`

The first element of the collection.

`var last: Self.Element?`

The last element of the collection.

`func randomElement() -> Self.Element?`

Returns a random element of the collection.

`func randomElement<T>(using: inout T) -> Self.Element?`

Returns a random element of the collection, using the given generator as a source for randomness.

Working with Encodings

`static var availableStringEncodings: [String.Encoding]`

An array of the encodings that strings support in the application's environment.

`static var defaultCStringEncoding: String.Encoding`

The C-string encoding assumed for any method accepting a C string as an argument.

`static func localizedName(of: String.Encoding) -> String`

Returns a human-readable string giving the name of the specified encoding.

`var isContiguousUTF8: Bool`

Returns whether this string's storage contains validly-encoded UTF-8 contents in contiguous memory.

`func makeContiguousUTF8()`

If this string is not contiguous, make it so. If this mutates the string, it will invalidate any pre-existing indices.

```
func withUTF8<R>((UnsafeBufferPointer<UInt8>) throws -> R) rethrows ->
R
```

Runs body over the content of this string in contiguous memory. If this string is not contiguous, this will first make it contiguous, which will also speed up subsequent access. If this mutates the string, it will invalidate any pre-existing indices.

Working with String Views

`var unicodeScalars: String.UnicodeScalarView`

The string's value represented as a collection of Unicode scalar values.

`init(String.UnicodeScalarView)`

Creates a string corresponding to the given collection of Unicode scalars.

`init(Substring.UnicodeScalarView)`

Creates a String having the given content.

`var utf16: String.UTF16View`

A UTF-16 encoding of `self`.

`init(String.UTF16View)`

Creates a string corresponding to the given sequence of UTF-16 code units.

`init?(Substring.UTF16View)`

Creates a String having the given content.

`var utf8: String.UTF8View`

A UTF-8 encoding of `self`.

`init(String.UTF8View)`

Creates a string corresponding to the given sequence of UTF-8 code units.

`init?(Substring.UTF8View)`

Creates a String having the given content.

Transforming a String's Characters

`func compactMap<ElementOfResult>((Self.Element) throws -> ElementOfResult?) rethrows -> [ElementOfResult]`

Returns an array containing the non-`nil` results of calling the given transformation with each element of this sequence.

`func flatMap<SegmentOfResult>((Self.Element) throws -> SegmentOfResult) rethrows -> [SegmentOfResult.Element]`

Returns an array containing the concatenated results of calling the given transformation with each element of this sequence.

`func flatMap<ElementOfResult>((Self.Element) throws -> ElementOfResult?) rethrows -> [ElementOfResult]`

`func reduce<Result>(Result, (Result, Self.Element) throws -> Result) rethrows -> Result`

Returns the result of combining the elements of the sequence using the given closure.

`func reduce<Result>(into: Result, (inout Result, Self.Element) throws -> ()) rethrows -> Result`

Returns the result of combining the elements of the sequence using the given closure.

`var lazy: LazySequence<Self>`

A sequence containing the same elements as this sequence, but on which some operations, such as `map` and `filter`, are implemented lazily.

Iterating over a String’s Characters

`func forEach((Self.Element) throws -> Void) rethrows`

Calls the given closure on each element in the sequence in the same order as a `for-in` loop.

`func enumerated() -> EnumeratedSequence<Self>`

Returns a sequence of pairs (*n*, *x*), where *n* represents a consecutive integer starting at zero and *x* represents an element of the sequence.

`func makeIterator() -> String.Iterator`

Returns an iterator over the elements of the collection.

`var underestimatedCount: Int`

A value less than or equal to the number of elements in the collection.

Reordering a String’s Characters

`func sorted() -> [Self.Element]`

Returns the elements of the sequence, sorted.

`func sorted(by: (Self.Element, Self.Element) throws -> Bool) rethrows -> [Self.Element]`

Returns the elements of the sequence, sorted using the given predicate as the comparison between elements.

`func reversed() -> ReversedCollection<Self>`

Returns a view presenting the elements of the collection in reverse order.

`func shuffled() -> [Self.Element]`

Returns the elements of the sequence, shuffled.

`func shuffled<T>(using: inout T) -> [Self.Element]`

Returns the elements of the sequence, shuffled using the given generator as a source for randomness.

Getting C Strings

`var utf8CString: ContiguousArray<CChar>`

A contiguously stored null-terminated UTF-8 representation of the string.

`func withCString<Result>((UnsafePointer<Int8>) throws -> Result) rethrows -> Result`

Calls the given closure with a pointer to the contents of the string, represented as a null-terminated sequence of UTF-8 code units.

`func withCString<Result, TargetEncoding>(encodedAs: TargetEncoding.Type, (UnsafePointer<TargetEncoding.CodeUnit>) throws -> Result) rethrows -> Result`

Calls the given closure with a pointer to the contents of the string, represented as a null-terminated sequence of code units.

Working with Paths

~~`init(FilePath)`~~ Deprecated

~~`init?(validatingUTF8: FilePath)`~~ Deprecated

Manipulating Indices

`var startIndex: String.Index`

The position of the first character in a nonempty string.

`var endIndex: String.Index`

A string's "past the end" position—that is, the position one greater than the last valid subscript argument.

`func index(after: String.Index) -> String.Index`

Returns the position immediately after the given index.

`func formIndex(after: inout Self.Index)`

Replaces the given index with its successor.

`func index(before: String.Index) -> String.Index`

Returns the position immediately before the given index.

`func formIndex(before: inout Self.Index)`

Replaces the given index with its predecessor.

`func index(String.Index, offsetBy: Int) -> String.Index`

Returns an index that is the specified distance from the given index.

`func index(String.Index, offsetBy: Int, limitedBy: String.Index) -> String.Index?`

Returns an index that is the specified distance from the given index, unless that distance is beyond a given limiting index.

`func formIndex(inout Self.Index, offsetBy: Int)`

Offsets the given index by the specified distance.

`func formIndex(inout Self.Index, offsetBy: Int, limitedBy: Self.Index) -> Bool`

Offsets the given index by the specified distance, or so that it equals the given limiting index.

`func distance(from: String.Index, to: String.Index) -> Int`

Returns the distance between two indices.

`var indices: DefaultIndices<Self>`

The indices that are valid for subscripting the collection, in ascending order.

Creating a Range Expression

`static func ... (Self, Self) -> ClosedRange<Self>`

Returns a closed range that contains both of its bounds.

`static func ... (Self) -> PartialRangeThrough<Self>`

Returns a partial range up to, and including, its upper bound.

`static func ... (Self) -> PartialRangeFrom<Self>`

Returns a partial range extending upward from a lower bound.

Encoding and Decoding

`func encode(to: any Encoder) throws`

Encodes this value into the given encoder.

`init(from: any Decoder) throws`

Creates a new instance by decoding from the given decoder.

Describing a String

`var description: String`

The value of this string.

`var debugDescription: String`

A representation of the string that is suitable for debugging.

`var customMirror: Mirror`

A mirror that reflects the `String` instance.

`var hashValue: Int`

The hash value.

`func hash(into: inout Hasher)`

Hashes the essential components of this value by feeding them into the given hasher.

Infrequently Used Functionality

`func index(of: Self.Element) -> Self.Index?`

Returns the first index where the specified value appears in the collection.

`init(NSString)`

`init(stringInterpolation: DefaultStringInterpolation)`

Creates a new instance from an interpolated string literal.

`init(stringLiteral: String)`

Creates an instance initialized to the given string value.

`init(unicodeScalarLiteral: Self.ExtendedGraphemeClusterLiteralType)`

`init(extendedGraphemeClusterLiteral: Self.StringLiteralType)`

~~`var customPlaygroundQuickLook: _PlaygroundQuickLook`~~

A custom playground Quick Look for the `String` instance.

Deprecated

`func withContiguousStorageIfAvailable<R>((UnsafeBufferPointer<Self.Element>) throws -> R) rethrows -> R?`

Executes a closure on the sequence’s contiguous storage.

Reference Types

Use bridged reference types when you need reference semantics or Foundation-specific behavior.

`class NSString`

A static, plain-text Unicode string object.

`class NSMutableString`

A dynamic plain-text Unicode string object.

Related String Types

`struct Substring`

A slice of a string.

`protocol StringProtocol`

A type that can represent a string as a collection of characters.

`struct Index`

A position of a character or code unit in a string.

`struct UnicodeScalarView`

A view of a string's contents as a collection of Unicode scalar values.

`struct UTF16View`

A view of a string's contents as a collection of UTF-16 code units.

`struct UTF8View`

A view of a string's contents as a collection of UTF-8 code units.

`struct Iterator`

A type that provides the collection's iteration interface and encapsulates its iteration state.

`struct Encoding`

Structures

`struct Comparator`

A `String` comparison performed using the given comparison options and locale.

`struct IntentInputOptions`

`struct StandardComparator`

Compares `String`s using one of a fixed set of standard comparison algorithms.

Initializers

`init(URL.Template.VariableName)`

`init(Slice<AttributedString.CharacterView>)`

~~`init(cString: inout CChar)`~~ Deprecated

~~`init(cString: inout UInt8)`~~ Deprecated

`init(cString: [UInt8])`

Creates a new string by copying the null-terminated UTF-8 data referenced by the given array.

`init(cString: [CChar])`

Creates a new string by copying the null-terminated UTF-8 data referenced by the given array.

~~`init(cString: String)`~~ Deprecated

~~`init?(cString: inout CChar, encoding: String.Encoding)`~~ Deprecated

~~`init?(cString: String, encoding: String.Encoding)`~~ Deprecated

`init(copying: UTF8Span)`

Creates a new string, copying the specified code units.

`init(decoding: FilePath.Root)`

On Unix, creates the string `"/"`

`init(decoding: FilePath.Component)`

Creates a string by interpreting the path component's content as UTF-8 on Unix and UTF-16 on Windows.

~~`init<Encoding>(decodingCString: String, as: Encoding.Type)`~~ Deprecated

~~`init<Encoding>(decodingCString: inout Encoding.CodeUnit, as: Encoding.Type)`~~ Deprecated

`init(describingForTest: some Any)`

Initialize this instance so that it can be presented in a test's output.

~~`init(platformString: String)`~~ Deprecated

~~`init(platformString: inout CInterop.PlatformChar)`~~ Deprecated `init(platformString: UnsafePointer<CInterop.PlatformChar>)`

Creates a string by interpreting the null-terminated platform string as UTF-8 on Unix and UTF-16 on Windows.

`init(platformString: [CInterop.PlatformChar])`

Creates a string by interpreting the null-terminated platform string as UTF-8 on Unix and UTF-16 on Windows.

~~`init?(utf8String: String)`~~ Deprecated

~~`init?(utf8String: inout CChar)`~~ Deprecated

`init?(validating: FilePath.Root)`

On Unix, creates the string `"/"`

`init?(validating: FilePath.Component)`

Creates a string from a path component, validating its contents as UTF-8 on Unix and UTF-16 on Windows.

`init?(validating: FilePath)`

Creates a string from a file path, validating its contents as UTF-8 on Unix and UTF-16 on Windows.

~~`init?(validatingCString: inout CChar)`~~ Deprecated

~~`init?(validatingCString: String)`~~ Deprecated

`init?(validatingPlatformString: UnsafePointer<CInterop.PlatformChar>)`

Creates a string by interpreting the null-terminated platform string as UTF-8 on Unix and UTF-16 on Windows.

~~`init?(validatingPlatformString: inout CInterop.PlatformChar)`~~ Deprecated

`init?(validatingPlatformString: [CInterop.PlatformChar])`

Creates a string by interpreting the null-terminated platform string as UTF-8 on Unix and UTF-16 on Windows.

~~`init?(validatingPlatformString: String)`~~ Deprecated

`init?(validatingUTF8: [CChar])`

Creates a new string by copying and validating the null-terminated UTF-8 data referenced by the given array.

~~`init?(validatingUTF8: inout CChar)`~~ Deprecated

~~`init?(validatingUTF8: String)`~~ Deprecated

Instance Properties

`var characters: String`

A view of the string’s contents as a collection of characters.

`var utf8Span: UTF8Span`

A UTF8span over the code units that make up this string.

Instance Methods

`func data(using: String.Encoding, allowLossyConversion: Bool) -> Data?`

`func withMutableCharacters<R>((inout String) -> R) -> R`

Applies the given closure to a mutable view of the string’s characters.

`func withPlatformString<Result>((UnsafePointer<CInterop.PlatformChar>) throws -> Result) rethrows -> Result`

Calls the given closure with a pointer to the contents of the string, represented as a null-terminated platform string.

Type Aliases

`typealias CharacterView`

A view of a string’s contents as a collection of characters.

`typealias CompareOptions`

`typealias EncodingConversionOptions`

`typealias EnumerationOptions`

~~`typealias IndexDistance`~~

A type that represents the number of steps between two `String.Index` values, where one value is reachable from the other.

Deprecated

`typealias Output`

`typealias Specification`

`typealias UnicodeScalarIndex`

The index type for a string’s `unicodeScalars` view.

```
typealias UnwrappedType
```

```
typealias ValueType
```

Type Properties

```
static var defaultResolverSpecification: some ResolverSpecification
```

Type Methods

~~static func decodeCString<Encoding>(inout Encoding.CodeUnit, as: Encoding.Type, repairingInvalidCodeUnits: Bool) -> (result: String, repairsMade: Bool)?~~

Deprecated

```
static func decodeCString<Encoding>([Encoding.CodeUnit], as: Encoding
.Type, repairingInvalidCodeUnits: Bool) -> (result: String, repairsMade
: Bool)?
```

~~static func decodeCString<Encoding>(String, as: Encoding.Type, repairingInvalidCodeUnits: Bool) -> (result: String, repairsMade: Bool)?~~

Deprecated

Default Implementations

- BidirectionalCollection Implementations
- CodingKeyRepresentable Implementations
- Collection Implementations
- Comparable Implementations
- CustomDebugStringConvertible Implementations
- CustomReflectable Implementations
- CustomStringConvertible Implementations
- Decodable Implementations
- Encodable Implementations
- Equatable Implementations
- ExpressibleByExtendedGraphemeClusterLiteral Implementations

- ExpressibleByStringInterpolation Implementations
- ExpressibleByStringLiteral Implementations
- ExpressibleByUnicodeScalarLiteral Implementations
- Hashable Implementations
- LosslessStringConvertible Implementations
- RangeReplaceableCollection Implementations
- Sequence Implementations
- StringProtocol Implementations
- TextOutputStream Implementations
- TextOutputStreamable Implementations

Relationships

Conforms To

`Attachable`
`BidirectionalCollection`
`BindableData`
`CKRecordValueProtocol`
`CVarArg`
`CodingKeyRepresentable`
`Collection`
`Comparable`
`ConvertibleFromGeneratedContent`
`ConvertibleToGeneratedContent`
`Copyable`
`CustomDebugStringConvertible`
`CustomReflectable`
`CustomStringConvertible`
`CustomTestStringConvertible`
`CustomURLRepresentationParameterConvertible`
`Decodable`
`Encodable`
`EntityIdentifierConvertible`

```
Equatable
ExpressibleByExtendedGraphemeClusterLiteral
ExpressibleByStringInterpolation
ExpressibleByStringLiteral
ExpressibleByUnicodeScalarLiteral
Generable
Hashable
InstructionsRepresentable
LosslessStringConvertible
MLDataValueConvertible
MLIdentifier
MirrorPath
MusicLibraryRequestFilterValueEquatable
Plottable
PrimitivePlottableProtocol
PromptRepresentable
RangeReplaceableCollection
RegexComponent
Sendable
SendableMetatype
Sequence
StringProtocol
TextOutputStream
TextOutputStreamable
Transferable
```

See Also

Standard Library

`struct Int`

A signed integer value type.

`struct Double`

A double-precision, floating-point value type.

`struct Array`

An ordered, random-access collection.

`struct Dictionary`

A collection whose elements are key-value pairs.

Swift Standard Library

Solve complex problems and write high-performance, readable code.