orphan:

Swift String Design

> **This Document**
>
> - contains interactive HTML commentary that does not currently appear in printed output. Hover your mouse over elements with a dotted pink underline to view the hidden commentary.
> - represents the intended design of Swift strings, not their current implementation state.
> - is being delivered in installments. Content still to come is outlined in Coming Installments.

> **Warning**
>
> This document was used in planning Swift 1.0; it has not been kept up to date and does not describe the current or planned behavior of Swift.

Contents

Introduction

Like all things Swift, our approach to strings begins with a deep respect for the lessons learned from many languages and libraries, especially Objective-C and Cocoa.

Goals

`String` should:

- honor industry standards such as Unicode
- when handling non-ASCII text, deliver "reasonably correct" results to users thinking only in terms of ASCII
- when handling ASCII text, provide "expected behavior" to users thinking only in terms of ASCII
- be hard to use incorrectly
- be easy to use correctly
- provide near-optimal efficiency for 99% of use cases
- provide a foundation upon which proper locale-sensitive operations can be built

Non-Goals

`String` need not:

- have behavior appropriate to all locales and contexts

- be an appropriate type (or base type) for all text storage applications

Overview By Example

In this section, we'll walk through some basic examples of Swift string usage while discovering its essential properties.

`String` is a First-Class Type

```
(swift) var s = "Yo"
// s: String = "Yo"
```

Unlike, say, C's `char*`, the meaning of a Swift string is always unambiguous.

Strings are Efficient

The implementation of `String` takes advantage of state-of-the-art optimizations, including:

- Storing short strings without heap allocation
- Sharing allocated buffers among copies and slices
- In-place modification of uniquely-owned buffers

As a result, copying and slicing strings, in particular, can be viewed by most programmers as being "almost free."

Strings are Mutable

```
(swift) extension String {
          func addEcho() {
            self += self
          }
        }
(swift) s.addEcho()
(swift) s
// s: String = "YoYo"
```

> **Why Mention It?**
>
> The ability to change a string's value might not be worth noting except that *some languages make all strings immutable*, as a way of working around problems that Swift has defined away--by making strings pure values (see below).

Strings are Value Types

Distinct string variables have independent values: when you pass someone a string they get a copy of the value, and when someone passes you a string *you own it*. Nobody can change a string value "behind your back."

```
(swift) class Cave {
          // Utter something in the cave
          func say(_ msg: String) -> String {
            msg.addEcho()
            self.lastSound = msg
            return self.lastSound
          }

          var lastSound: String   // a Cave remembers the last sound made
        }
(swift) var c = Cave()
// c: Cave = <Cave instance>
(swift) s = "Hey"
(swift) var t = c.say(s)
// t: String = "HeyHey"
(swift) s
// s: String = "Hey"
(swift) t.addEcho()
(swift) [s, c.lastSound, t]
// r0: [String] = ["Hey", "HeyHey", "HeyHeyHeyHey"]
```

Strings are Unicode-Aware

Swift applies Unicode algorithms wherever possible. For example, distinct sequences of code points are treated as equal if they represent the same character: [2]

```
(swift) var n1 = "\u006E\u0303"
// n1 : String = "Ñ"
(swift) var n2 = "\u00F1"
// n2 : String = "Ñ"
(swift) n1 == n2
// r0 : Bool = true
```

Note that individual code points are still observable by explicit request:

```
(swift) n1.codePoints == n2.codePoints
// r0 : Bool = false
```

> **Deviations from Unicode**
>
> Any deviation from what Unicode specifies requires careful justification. So far, we have found two possible points of deviation for Swift `String`:
>
> 1. The Unicode Text Segmentation Specification says, "do not break between CR and LF." However,

breaking extended grapheme clusters between CR and LF may necessary if we wish `String` to "behave normally" for users of pure ASCII. This point is still open for discussion.

2. The Unicode Text Segmentation Specification says, "do not break between regional indicator symbols." However, it also says " (Sequences of more than two RI characters should be separated by other characters, such as U+200B ZWSP)." Although the parenthesized note probably has less official weight than the other admonition, breaking pairs of RI characters seems like the right thing for us to do given that Cocoa already forms strings with several adjacent pairs of RI characters, and the Unicode spec *can* be read as outlawing such strings anyway.

Strings are Locale-Agnostic

Strings neither carry their own locale information, nor provide behaviors that depend on a global locale setting. Thus, for any pair of strings `s1` and `s2`, "`s1 == s2`" yields the same result regardless of system state. Strings *do* provide a suitable foundation on which to build locale-aware interfaces.[3]

Strings are Containers

```
(swift) var s = "Strings are awesome"
// s : String = "Strings are awesome"
(swift) var r = s.find("awe")
// r : Range<StringIndex> = <"...are aÌ²wÌ²eÌ²some">
(swift) s[r.start]
// r0 : Character = Character("a")
```

String Indices

`String` implements the `Container` protocol, but **cannot be indexed by integers**. Instead, `String.IndexType` is a library type conforming to the `BidirectionalIndex` protocol.

This might seem surprising at first, but code that indexes strings with arbitrary integers is seldom Unicode-correct in the first place, and Swift provides alternative interfaces that encourage Unicode-correct code. For example, instead of `s[0] == 'S'` you'd write `s.startsWith("S")`.

Strings are Composed of `Character`s

`Character`, the element type of `String`, represents a **grapheme cluster**, as specified by a default or tailored Unicode segmentation algorithm. This term is precisely defined by the Unicode specification, but it roughly means what the user thinks of when she hears "character". For example, the pair of code points "LATIN SMALL LETTER N, COMBINING TILDE" forms a single grapheme cluster, "Ã±".

Access to lower-level elements is still possible by explicit request:

```
(swift) s.codePoints[s.codePoints.start]
// r1 : CodePoint = CodePoint(83) /* S */
(swift) s.bytes[s.bytes.start]
// r2 : UInt8 = UInt8(83)
```

Strings Support Flexible Segmentation

The `Characters` enumerated when simply looping over elements of a Swift string are extended grapheme clusters as determined by Unicode's Default Grapheme Cluster Boundary Specification. [5]

This segmentation offers naÃ¯ve users of English, Chinese, French, and probably a few other languages what we think of as the "expected results." However, not every script can be segmented uniformly for all purposes. For example, searching and collation require different segmentations in order to handle Indic scripts correctly. To that end, strings support properties for more-specific segmentations:

> **Note**
>
> The following example needs a more interesting string in order to demonstrate anything interesting. Hopefully Aki has some advice for us.

```
(swift) for c in s { print("Extended Grapheme Cluster: (c)") }
Extended Grapheme Cluster: f
Extended Grapheme Cluster: o
Extended Grapheme Cluster: o
(swift) for c in s.collationCharacters {
          print("Collation Grapheme Cluster: (c)")
        }
Collation Grapheme Cluster: f
Collation Grapheme Cluster: o
Collation Grapheme Cluster: o
(swift) for c in s.searchCharacters {
          print("Search Grapheme Cluster: (c)")
        }
Search Grapheme Cluster: f
Search Grapheme Cluster: o
Search Grapheme Cluster: o
```

Also, each such segmentation provides a unique `IndexType`, allowing a string to be indexed directly with different indexing schemes

> **System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\(swift-main)(docs)StringDesign.rst, line 392)**
>
> Cannot analyze code. No Pygments lexer found for "swift-console".
>
> ```
> .. code-block:: swift-console
>
>    |swift| var i = s.searchCharacters.startIndex
>    `// r2 : UInt8 = UInt8(83)`
> ```

Strings are Sliceable

```
(swift) s[r.start...r.end]
// r2 : String = "awe"
(swift) s[r.start...]
// r3 : String = "awesome"
(swift) s[...r.start]
// r4 : String = "Strings are "
(swift) s[r]
// r5 : String = "awe"
(swift) s[r] = "hand"
(swift) s
// s : String = "Strings are handsome"
```

Strings are Encoded as UTF-8

```
(swift) for x in "bump".bytes {
          print(x)
        }
98
117
109
112
```

> **Encoding Conversion**
>
> Conversion to and from other encodings is out-of-scope for `String` itself, but could be provided, e.g., by an `Encoding` module.

Coming Installments

- Reference Manual
- Rationales
- Cocoa Bridging Strategy
- Comparisons with NSString
  - High Level

- Member-by-member

Reference Manual

- s.bytes
- s.indices
- s[i]
- s[start...end]
- s == t, s != t
- s < t, s > t, s <= t, s >= t
- s.hash()
- s.startsWith(), s.endsWith()
- s + t, s += t, s.append(t)
- s.split(), s.split(n), s.split(sep, n)
- s.strip(), s.stripStart(), s.stripEnd()
- s.commonPrefix(t), s.mismatch(t)
- s.toUpper(), s.toLower()
- s.trim(predicate)
- s.replace(old, new, count)
- s.join(sequenceOfStrings)

Cocoa Bridging Strategy

Rationales

Why a Built-In String Type?

> **DaveZ Sez**
>
> In the "why a built-in string type" section, I think the main narrative is that two string types is bad, but that we have two string types in Objective-C for historically good reasons. To get one string type, we need to merge the high-level features of Objective-C with the performance of C, all while not having the respective bad the bad semantics of either (reference semantics and "anarchy" memory-management respectively). Furthermore, I'd write "value semantics" in place of "C++ semantics". I know that is what you meant, but we need to tread carefully in the final document.

`NSString` and `NSMutableString`--the string types provided by Cocoa--are full-featured classes with high-level functionality for writing fully-localized applications. They have served Apple programmers well; so, why does Swift have its own string type?

- ObjCMessageSend
- Error Prone Mutability Reference semantics don't line up with how people think about strings
- 2 is too many string types. two APIs duplication of effort documentation Complexity adds decisions for users etc.
- ObjC needed to innovate because C strings suck O(N) length no localization no memory management no specified encoding
- C strings had to stay around for performance reasons and interoperability

Want performance of C, sane semantics of C++ strings, and high-level goodness of ObjC.

> The design of `NSString` is *very* different from the string designs of most modern programming languages, which all tend to be very similar to one another. Although existing `NSString` users are a critical constituency today, current trends indicate that most of our *future* target audience will not be `NSString` users. Absent compelling justification, it's important to make the Swift programming environment as familiar as possible for them.

How Would You Design It?

> **DaveZ Sez**
>
> In the "how would you design it" section, the main narrative is twofold: how does it "feel" and how efficient is it? The former is about feeling built in, which we can easily argue that both C strings or Cocoa strings fail at for their respective semantic (and often memory management related) reasons. Additionally, the "feel" should be modern, which is where the Cocoa framework and the Unicode standard body do better than C. Nevertheless, we can still do better than Objective-C and your strong work at helping people reason about grapheme clusters instead of code points (or worse, units) is wonderful and it feels right to developers. The second part of the narrative is about being efficient, which is where arguing for UTF8 is the non-obvious but "right" answer for the reasons we have discussed.

- It'd be an independent *value* so you don't have to micromanage sharing and mutation
- It'd be UTF-8 because:
  - UTF-8 has been the clear winner among Unicode encodings since at least 2008; Swift should interoperate smoothly and

efficiently with the rest of the world's systems
    - UTF-8 is a fairly efficient storage format, especially for ASCII but also for the most common non-ASCII code points.
    - This posting elaborates on some other nice qualities of UTF-8:
        1. All ASCII files are already UTF-8 files
        2. ASCII bytes always represent themselves in UTF-8 files. They never appear as part of other UTF-8 sequences
        3. ASCII code points are always represented as themselves in UTF-8 files. They cannot be hidden inside multibyte UTF-8 sequences
        4. UTF-8 is self-synchronizing
        5. CodePoint substring search is just byte string search
        6. Most programs that handle 8-bit files safely can handle UTF-8 safely
        7. UTF-8 sequences sort in code point order.
        8. UTF-8 has no "byte order."
- It would be efficient, taking advantage of state-of-the-art optimizations, including:
    - Storing short strings without heap allocation
    - Sharing allocated buffers among copies and slices
    - In-place modification of uniquely-owned buffers

Comparisons with `NSString`

High-Level Comparison with `NSString`

> **DaveZ Sez**
>
> I think the main message of the API breadth subsection is that URLs, paths, etc would be modeled as formal types in Swift (i.e. not as extensions on String). Second, I'd speculate less on what Foundation could do (like extending String) and instead focus on the fact that NSString still exists as an escape hatch for those that feel that they need or want it. Furthermore, I'd move up the "element access" discussion above the "escape hatch" discussion (which should be last in the comparison with NSString discussion).

API Breadth

The `NSString` interface clearly shows the effects of 20 years of evolution through accretion. It is broad, with functionality addressing encodings, paths, URLs, localization, and more. By contrast, the interface to Swift's `String` is much narrower.

Of course, there's a reason for every `NSString` method, and the full breadth of `NSString` functionality must remain accessible to the Cocoa/Swift programmer. Fortunately, there are many ways to address this need. For example:

- The `Foundation` module can extend `String` with the methods of `NSString`. The extent to which we provide an identical-feeling interface and/or correct any `NSString` misfeatures is still TBD and wide open for discussion.
- We can create a new modular interface in pure Swift, including a `Locale` module that addresses localized string operations, an `Encoding` module that addresses character encoding schemes, a `Regex` module that provides regular expression functionality, etc. Again, the specifics are TBD.
- When all else fails, users can convert their Swift `String`s to `NSString`s when they want to access `NSString`-specific functionality:

```
NSString(mySwiftString).localizedStandardCompare(otherSwiftString)
```

For Swift version 1.0, we err on the side of keeping the string interface small, coherent, and sufficient for implementing higher-level functionality.

Element Access

`NSString` exposes UTF-16 code units as the primary element on which indexing, slicing, and iteration operate. Swift's UTF-8 code units are only available as a secondary interface.

`NSString` is indexable and sliceable using `Int`s, and so exposes a `length` attribute. Swift's `String` is indexable and sliceable using an abstract `BidirectionalIndex` type, and does not expose its length.

Sub-Strings

Creating substrings in Swift is very fast. Therefore, Cocoa APIs that operate on a substring given as an `NSRange` are replaced with Swift APIs that just operate on `String`s. One can use range-based subscripting to achieve the same effect. For example: `[str doFoo:arg withRange:subrange]` becomes `str[subrange].doFoo(arg)`.

`NSString` Member-by-Member Comparison

Notes:

- The following are from public headers from public frameworks, which are AppKit and Foundation (verified).
- Deprecated Cocoa APIs are not considered
- A status of "*Remove*" below indicates a feature whose removal is anticipated. Rationale is provided for these cases.

Indexing

Cocoa:

> **Why doesn't `String` support `.length`?**
>
> In Swift, by convention, `x.length` is used to represent the number of elements in a container, and since `String` is a container of abstract `Character`s, `length` would have to count those.
>
> This meaning of `length` is unimplementable in O(1). It can be cached, although not in the memory block where the characters are stored, since we want a `String` to share storage with its slices. Since the body of the `String` must already store the `String`'s *byte length*, caching the `length` would increase the footprint of the top-level String object. Finally, even if `length` were provided, doing things with `String` that depend on a specific numeric `length` is error-prone.

```
- (NSUInteger)length
- (unichar)characterAtIndex:(NSUInteger)index;
```

Swift:

not directly provided, but similar functionality is available:

```
for j in 0...s.bytes.length {
  doSomethingWith(s.bytes[j])
}
```

Cocoa:

```
- (NSRange)rangeOfComposedCharacterSequenceAtIndex:(NSUInteger)index;
- (NSRange)rangeOfComposedCharacterSequencesForRange:(NSRange)range;
```

Swift:

```
typealias IndexType = ...
func indices() -> Range<IndexType>
subscript(i: IndexType) -> Character
```

> **Usage**
>
> ```
> for i in someString.indices() {
>   doSomethingWith(someString[i])
> }
>
> var (i, j) = someString.indices().bounds
> while (i != j) {
>   doSomethingElseWith(someString[i])
>   ++i
> }
> ```

Slicing

Cocoa:

```
- (void)getCharacters:(unichar *)buffer range:(NSRange)aRange;
```

Swift:

```
typealias IndexType = ...
subscript(r: Range<IndexType>) -> Character
```

Indexing

Cocoa:

```
- (NSString *)substringToIndex:(NSUInteger)to;
- (NSString *)substringFromIndex:(NSUInteger)from;
- (NSString *)substringWithRange:(NSRange)range;
```

Swift:

```
subscript(range : Range<IndexType>) -> String
```

> **Example**
>
> ```
> s[beginning...ending] // [s substringWithRange: NSMakeRange(beginning, end
> s[beginning...]       // [s substringFromIndex: beginning]
> s[...ending]          // [s substringToIndex: ending]
> ```
>
> **Note:** Swift may need additional interfaces to support `index...` and `...index` notations. This part of the `Container` protocol design isn't worked out yet.

Comparison

Cocoa:

```
- (BOOL)isEqualToString:(NSString *)aString;
- (NSComparisonResult)compare:(NSString *)string;
```

Swift:

```
func == (lhs: String, rhs: String) -> Bool
func != (lhs: String, rhs: String) -> Bool
func <  (lhs: String, rhs: String) -> Bool
func >  (lhs: String, rhs: String) -> Bool
func <= (lhs: String, rhs: String) -> Bool
func >= (lhs: String, rhs: String) -> Bool
```

`NSString` comparison is "literal" by default. As the documentation says of `isEqualToString`,

> "Ã–" represented as the composed character sequence "O" and umlaut would not compare equal to "Ã–" represented as one Unicode character.

By contrast, Swift string's primary comparison interface uses Unicode's default collation algorithm, and is thus always "Unicode-correct." Unlike comparisons that depend on locale, it is also stable across changes in system state. However, *just like* `NSString`'s `isEqualToString` and `compare` methods, it should not be expected to yield ideal (or even "proper") results in all contexts.

Cocoa:

```
- (NSComparisonResult)compare:(NSString *)string options:(NSStringCompareOptions)mask;
- (NSComparisonResult)compare:(NSString *)string options:(NSStringCompareOptions)mask range
- (NSComparisonResult)caseInsensitiveCompare:(NSString *)string;
```

Swift: *various compositions of primitive operations* / TBD

- As noted above, instead of passing sub-range arguments, we expect Swift users to compose slicing with whole-string operations.
- Other details of these interfaces are distinguished by an `NSStringCompareOptions` mask, of which `caseInsensitiveCompare:` is essentially a special case:

  `NSCaseInsensitiveSearch`: Whether a direct interface is needed at all in Swift, and if so, its form, are TBD. However, we should consider following the lead of Python 3, wherein case conversion also normalizes letterforms. Then one can combine `String.toLower()` with default comparison to get a case-insensitive comparison:

  ```
  { $0.toLower() == $1.toLower() }
  ```

  `NSLiteralSearch`: Though it is the default for `NSString`, this option is essentially only useful as a performance optimization when the string content is known to meet certain restrictions (i.e. is known to be pure ASCII). When such optimization is absolutely necessary, Swift standard library algorithms can be used directly on the `String`'s UTF8 code units. However, Swift will also perform these optimizations automatically (at the cost of a single test/branch) in many cases, because each `String` stores a bit indicating whether its content is known to be ASCII.

  `NSBackwardsSearch`: It's unclear from the docs how this option interacts with other `NSString` options, if at all, but basic cases can be handled in Swift by `s1.endsWith(s2)`.

  `NSAnchoredSearch`: Not applicable to whole-string comparisons

  `NSNumericSearch`: While it's legitimate to defer this functionality to Cocoa, it's (probably--see <rdar://problem/14724804>) locale-independent and easy enough to implement in Swift. TBD

  `NSDiacriticInsensitiveSearch`: Ditto; TBD

  `NSWidthInsensitiveSearch`: Ditto; TBD

`NSForcedOrderingSearch`:
Ditto; TBD. Also see <rdar://problem/14724888>

`NSRegularExpressionSearch`:
We can defer this functionality to Cocoa, or dispatch directly to ICU as an optimization. It's unlikely that we'll be building Swift its own regexp engine for 1.0.

Cocoa:

```
- (NSComparisonResult)localizedCompare:(NSString *)string;
- (NSComparisonResult)localizedCaseInsensitiveCompare:(NSString *)string;
- (NSComparisonResult)localizedStandardCompare:(NSString *)string;
- (NSComparisonResult)compare:(NSString *)string options:(NSStringCompareOptions)mask range
```

Swift:
As these all depend on locale, they are TBD

Searching

Cocoa:

> **Rationale**
>
> Modern languages (Java, C#, Python, Ruby...) have standardized on variants of `startsWith`/`endsWith`. There's no reason Swift should deviate from de-facto industry standards here.

```
- (BOOL)hasPrefix:(NSString *)aString;
- (BOOL)hasSuffix:(NSString *)aString;
```

Swift:

```
func startsWith(_ prefix: String)
func endsWith(  suffix: String)
```

Cocoa:

```
- (NSRange)rangeOfString:(NSString *)aString;
```

Swift:

```
func find(_ sought: String) -> Range<String.IndexType>
```

> **Note**
>
> Most other languages provide something like `s1.indexOf(s2)`, which returns only the starting index of the first match. This is far less useful than the range of the match, and is always available via `s1.find(s2).bounds.0`

Cocoa:

```
- (NSRange)rangeOfCharacterFromSet:(NSCharacterSet *)aSet;
```

Swift:

> **Naming**
>
> The Swift function is just an algorithm that comes from conformance to the `Container` protocol, which explains why it doesn't have a `String`-specific name.

```
func find(  match: (Character) -> Bool) -> Range<String.IndexType>
```

> **Usage Example**
>
> The `NSString` semantics can be achieved as follows:
>
> ```
> someString.find( {someCharSet.contains($0)} )
> ```

Cocoa:

```
- (NSRange)rangeOfString:(NSString *)aString options:(NSStringCompareOptions)mask;
- (NSRange)rangeOfString:(NSString *)aString options:(NSStringCompareOptions)mask range:(NSI
- (NSRange)rangeOfString:(NSString *)aString options:(NSStringCompareOptions)mask range:(NSI

- (NSRange)rangeOfCharacterFromSet:(NSCharacterSet *)aSet options:(NSStringCompareOptions)ma
- (NSRange)rangeOfCharacterFromSet:(NSCharacterSet *)aSet options:(NSStringCompareOptions)ma
```

These functions

Swift:
various compositions of primitive operations / TBD

Building

Cocoa:

```
- (NSString *)stringByAppendingString:(NSString *)aString;
```

Swift:

> **`append`**
>
> the `append` method is a consequence of `String`'s conformance to `TextOutputStream`. See the *Swift formatting proposal* for details.

```
func + (lhs: String, rhs: String) -> String
func [infix, assignment] += (lhs: [inout] String, rhs: String)
func append(_ suffix: String)
```

Dynamic Formatting

Cocoa:

```
- (NSString *)stringByAppendingFormat:(NSString *)format, ... NS_FORMAT_FUNCTION(1,2);
```

Swift: *Not directly provided*--see the *Swift formatting proposal*

Extracting Numeric Values

Cocoa:

```
- (double)doubleValue;
- (float)floatValue;
- (int)intValue;
- (NSInteger)integerValue;
- (long long)longLongValue;
- (BOOL)boolValue;
```

Swift: Not in `String`--It is up to other types to provide their conversions to and from String. See also this rationale

Splitting

Cocoa:

```
- (NSArray *)componentsSeparatedByString:(NSString *)separator;
- (NSArray *)componentsSeparatedByCharactersInSet:(NSCharacterSet *)separator;
```

Swift:

```
func split(_ maxSplit: Int = Int.max()) -> [String]
func split(_ separator: Character, maxSplit: Int = Int.max()) -> [String]
```

The semantics of these functions were taken from Python, which seems to be a fairly good representative of what modern languages are currently doing. The first overload splits on all whitespace characters; the second only on specific characters. The universe of possible splitting functions is quite broad, so the particulars of this interface are **wide open for discussion**. In Swift right now, these methods (on `CodePoints`) are implemented in terms of a generic algorithm:

```
func split<Seq: Sliceable, IsSeparator: Predicate
    where IsSeparator.Arguments == Seq.Element
>(_ seq: Seq, isSeparator: IsSeparator, maxSplit: Int = Int.max(),
  allowEmptySlices: Bool = false) -> [Seq]
```

Splitting

Cocoa:

```
- (NSString *)commonPrefixWithString:(NSString *)aString options:(NSStringCompareOptions)mas
```

Swift:

```
func commonPrefix(_ other: String) -> String
```

Upper/Lowercase

Cocoa:

```
- (NSString *)uppercaseString;
- (NSString *)uppercaseStringWithLocale:(NSLocale *)locale;
- (NSString *)lowercaseString;
- (NSString *)lowercaseStringWithLocale:(NSLocale *)locale;
```

Swift:

> **Naming**
>
> Other languages have overwhelmingly settled on `upper()` or `toUpper()` for this functionality

```
func toUpper() -> String
func toLower() -> String
```

Capitalization

Cocoa:

```
- (NSString *)capitalizedString;
- (NSString *)capitalizedStringWithLocale:(NSLocale *)locale;
```

Swift: **TBD**

> **Note**
>
> `NSString` capitalizes the first letter of each substring separated by spaces, tabs, or line terminators, which is in no sense "Unicode-correct." In most other languages that support a `capitalize` method, it operates only on the first character of the string, and capitalization-by-word is named something like "`title`." If Swift `String` supports capitalization by word, it should be Unicode-correct, but how we sort this particular area out is still **TBD**.

Cocoa:
```
- (NSString *)stringByTrimmingCharactersInSet:(NSCharacterSet *)set;
```

Swift:
```
trim trim(match: (Character) -> Bool) -> String
```

> **Usage Example**
>
> The `NSString` semantics can be achieved as follows:
>
> ```
> someString.trim( {someCharSet.contains($0)} )
> ```

Cocoa:
```
- (NSString *)stringByPaddingToLength:(NSUInteger)newLength withString:(NSString *)padStrin
```

Swift: *Not provided.* It's not clear whether this is useful at all for non-ASCII strings, and

---

Cocoa:
```
- (void)getLineStart:(NSUInteger *)startPtr end:(NSUInteger *)lineEndPtr contentsEnd:(NSUInt
```

Swift: **TBD**

Cocoa:
```
- (NSRange)lineRangeForRange:(NSRange)range;
```

Swift: **TBD**

Cocoa:
```
- (void)getParagraphStart:(NSUInteger *)startPtr end:(NSUInteger *)parEndPtr contentsEnd:(NS
```

Swift: **TBD**

Cocoa:
```
- (NSRange)paragraphRangeForRange:(NSRange)range;
```

Swift: **TBD**

Cocoa:
```
- (void)enumerateSubstringsInRange:(NSRange)range options:(NSStringEnumerationOptions)opts u
```

Swift: **TBD**

Cocoa:
```
- (void)enumerateLinesUsingBlock:(void (^)(NSString *line, BOOL *stop))block;
```

Swift: **TBD**

Cocoa:
```
- (NSString *)description;
```

Swift: **TBD**

Cocoa:
```
- (NSUInteger)hash;
```

Swift: **TBD**

Cocoa:
```
- (NSStringEncoding)fastestEncoding;
```

Swift: **TBD**

Cocoa:
```
- (NSStringEncoding)smallestEncoding;
```

Swift: **TBD**

Cocoa: `- (NSData *)dataUsingEncoding:(NSStringEncoding)encoding allowLossyConversion:(BOOL)lossy;`

Swift: **TBD**

Cocoa: `- (NSData *)dataUsingEncoding:(NSStringEncoding)encoding;`

Swift: **TBD**

- (BOOL)**canBeConvertedToEncoding:**(NSStringEncoding)encoding;

Cocoa: `- (__strong const char *)cStringUsingEncoding:(NSStringEncoding)encoding NS_RETURNS_INNER_P(`

Swift: **TBD**

Cocoa: `- (BOOL)getCString:(char *)buffer maxLength:(NSUInteger)maxBufferCount encoding:(NSStringEnc`

Swift: **TBD**

Cocoa: `- (BOOL)getBytes:(void *)buffer maxLength:(NSUInteger)maxBufferCount usedLength:(NSUInteger`

Swift: **TBD**

Cocoa: `- (NSUInteger)maximumLengthOfBytesUsingEncoding:(NSStringEncoding)enc;`

Swift: **TBD**

Cocoa: `- (NSUInteger)lengthOfBytesUsingEncoding:(NSStringEncoding)enc;`

Swift: **TBD**

Cocoa: `- (NSString *)decomposedStringWithCanonicalMapping;`

Swift: **TBD**

Cocoa: `- (NSString *)precomposedStringWithCanonicalMapping;`

Swift: **TBD**

Cocoa: `- (NSString *)decomposedStringWithCompatibilityMapping;`

Swift: **TBD**

Cocoa: `- (NSString *)precomposedStringWithCompatibilityMapping;`

Swift: **TBD**

Cocoa: `- (NSString *)stringByFoldingWithOptions:(NSStringCompareOptions)options locale:(NSLocale *)`

Swift: **TBD**

Cocoa: `- (NSString *)stringByReplacingOccurrencesOfString:(NSString *)target withString:(NSString *`

Swift: **TBD**

Cocoa: `- (NSString *)stringByReplacingOccurrencesOfString:(NSString *)target withString:(NSString *`

Swift: **TBD**

Cocoa: `- (NSString *)stringByReplacingCharactersInRange:(NSRange)range withString:(NSString *)repla`

Cocoa: `- (__strong const char *)UTF8String NS_RETURNS_INNER_POINTER;`

Swift: **TBD**

Cocoa: `+ (NSStringEncoding)defaultCStringEncoding;`

Swift: **TBD**

Cocoa: `+ (const NSStringEncoding *)availableStringEncodings;`

Swift: **TBD**

Cocoa: `+ (NSString *)localizedNameOfStringEncoding:(NSStringEncoding)encoding;`

Constructors

Cocoa: `- (instancetype)init;`

Cocoa: `- (instancetype)initWithString:(NSString *)aString;`

Cocoa: `+ (instancetype)string;`

Cocoa: `+ (instancetype)stringWithString:(NSString *)string;`

Not available (too error prone)

Cocoa: `- (instancetype)initWithCharactersNoCopy:(unichar *)characters length:(NSUInteger)length fre`

Swift: **TBD**

Cocoa: `- (instancetype)initWithCharacters:(const unichar *)characters length:(NSUInteger)length;`

Swift: **TBD**

Cocoa: `- (instancetype)initWithUTF8String:(const char *)nullTerminatedCString;`

Swift: **TBD**

Cocoa: `- (instancetype)initWithFormat:(NSString *)format, ... NS_FORMAT_FUNCTION(1,2);`

Swift: **TBD**

Cocoa: `- (instancetype)initWithFormat:(NSString *)format arguments:(va_list)argList NS_FORMAT_FUNC`

Swift: **TBD**

Cocoa: `- (instancetype)initWithFormat:(NSString *)format locale:(id)locale, ... NS_FORMAT_FUNCTION`

Swift: **TBD**

Cocoa: `- (instancetype)initWithFormat:(NSString *)format locale:(id)locale arguments:(va_list)argL`

Swift: **TBD**

Cocoa: `- (instancetype)initWithData:(NSData *)data encoding:(NSStringEncoding)encoding;`

Swift: **TBD**

Cocoa: `- (instancetype)initWithBytes:(const void *)bytes length:(NSUInteger)len encoding:(NSStringE`

Swift: **TBD**

Cocoa: `- (instancetype)initWithBytesNoCopy:(void *)bytes length:(NSUInteger)len encoding:(NSStringE`

Swift: **TBD**

Cocoa: `+ (instancetype)stringWithCharacters:(const unichar *)characters length:(NSUInteger)length;`

Swift: **TBD**

Cocoa: `+ (instancetype)stringWithUTF8String:(const char *)nullTerminatedCString;`

Swift: **TBD**

Cocoa: `+ (instancetype)stringWithFormat:(NSString *)format, ... NS_FORMAT_FUNCTION(1,2);`

Swift: **TBD**

Cocoa: `+ (instancetype)localizedStringWithFormat:(NSString *)format, ... NS_FORMAT_FUNCTION(1,2);`

Swift: **TBD**

Cocoa: `- (instancetype)initWithCString:(const char *)nullTerminatedCString encoding:(NSStringEncod`

Swift: **TBD**

Cocoa: `+ (instancetype)stringWithCString:(const char *)cString encoding:(NSStringEncoding)enc;`

Linguistic Analysis

Cocoa:
```
- (NSArray *)linguisticTagsInRange:(NSRange)range scheme:(NSString *)tagScheme options:(NSL
- (void)enumerateLinguisticTagsInRange:(NSRange)range scheme:(NSString *)tagScheme options:
```

Swift: **TBD**

Unavailable on Swift Strings

URL Handling

```
- (instancetype)initWithContentsOfURL:(NSURL *)url encoding:(NSStringEncoding)enc error:(NSError **)error;
+ (instancetype)stringWithContentsOfURL:(NSURL *)url encoding:(NSStringEncoding)enc error:(NSError **)error;
- (instancetype)initWithContentsOfURL:(NSURL *)url usedEncoding:(NSStringEncoding *)enc error:(NSError **)error
+ (instancetype)stringWithContentsOfURL:(NSURL *)url usedEncoding:(NSStringEncoding *)enc error:(NSError **)err
- (BOOL)writeToURL:(NSURL *)url atomically:(BOOL)useAuxiliaryFile encoding:(NSStringEncoding)enc error:(NSError
- (NSString *)stringByAddingPercentEncodingWithAllowedCharacters:(NSCharacterSet *)allowedCharacters;
- (NSString *)stringByRemovingPercentEncoding;
- (NSString *)stringByAddingPercentEscapesUsingEncoding:(NSStringEncoding)enc;
- (NSString *)stringByReplacingPercentEscapesUsingEncoding:(NSStringEncoding)enc;
```

See: class File

```
- (instancetype)initWithContentsOfFile:(NSString *)path encoding:(NSStringEncoding)enc error:(NSError **)error;
+ (instancetype)stringWithContentsOfFile:(NSString *)path encoding:(NSStringEncoding)enc error:(NSError **)erro
- (instancetype)initWithContentsOfFile:(NSString *)path usedEncoding:(NSStringEncoding *)enc error:(NSError **)
+ (instancetype)stringWithContentsOfFile:(NSString *)path usedEncoding:(NSStringEncoding *)enc error:(NSError *
- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)useAuxiliaryFile encoding:(NSStringEncoding)enc error:(NS
```

Path Handling

```
+ (NSString *)pathWithComponents:(NSArray *)components;
- (NSArray *)pathComponents;
- (BOOL)isAbsolutePath;
- (NSString *)lastPathComponent;
- (NSString *)stringByDeletingLastPathComponent;
- (NSString *)stringByAppendingPathComponent:(NSString *)str;
- (NSString *)pathExtension;
- (NSString *)stringByDeletingPathExtension;
- (NSString *)stringByAppendingPathExtension:(NSString *)str;
- (NSString *)stringByAbbreviatingWithTildeInPath;
- (NSString *)stringByExpandingTildeInPath;
- (NSString *)stringByStandardizingPath;
- (NSString *)stringByResolvingSymlinksInPath;
- (NSArray *)stringsByAppendingPaths:(NSArray *)paths;
- (NSUInteger)completePathIntoString:(NSString **)outputName caseSensitive:(BOOL)flag matchesIntoArray:(NSArray
- (__strong const char *)fileSystemRepresentation NS_RETURNS_INNER_POINTER;
- (BOOL)getFileSystemRepresentation:(char *)cname maxLength:(NSUInteger)max;
```

Property Lists

Property lists are a feature of Cocoa.

```
- (id)propertyList;
- (NSDictionary *)propertyListFromStringsFileFormat;
Not applicable. Swift does not provide GUI support.
```

```
- (NSSize)sizeWithAttributes:(NSDictionary *)attrs;
- (void)drawAtPoint:(NSPoint)point withAttributes:(NSDictionary *)attrs;
- (void)drawInRect:(NSRect)rect withAttributes:(NSDictionary *)attrs;
- (void)drawWithRect:(NSRect)rect options:(NSStringDrawingOptions)options attributes:(NSDictionary *)attributes
- (NSRect)boundingRectWithSize:(NSSize)size options:(NSStringDrawingOptions)options attributes:(NSDictionary *)
- (NSArray *)writableTypesForPasteboard:(NSPasteboard *)pasteboard;
- (NSPasteboardWritingOptions)writingOptionsForType:(NSString *)type pasteboard:(NSPasteboard *)pasteboard;
- (id)pasteboardPropertyListForType:(NSString *)type;
+ (NSArray *)readableTypesForPasteboard:(NSPasteboard *)pasteboard;
+ (NSPasteboardReadingOptions)readingOptionsForType:(NSString *)type pasteboard:(NSPasteboard *)pasteboard;
- (id)initWithPasteboardPropertyList:(id)propertyList ofType:(NSString *)type;
```

Deprecated APIs

Already deprecated in Cocoa.

```
- (const char *)cString;
- (const char *)lossyCString;
- (NSUInteger)cStringLength;
- (void)getCString:(char *)bytes;
- (void)getCString:(char *)bytes maxLength:(NSUInteger)maxLength;
- (void)getCString:(char *)bytes maxLength:(NSUInteger)maxLength range:(NSRange)aRange remainingRange:(NSRangeP
- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)useAuxiliaryFile;
- (BOOL)writeToURL:(NSURL *)url atomically:(BOOL)atomically;
- (id)initWithContentsOfFile:(NSString *)path;
- (id)initWithContentsOfURL:(NSURL *)url;
+ (id)stringWithContentsOfFile:(NSString *)path;
+ (id)stringWithContentsOfURL:(NSURL *)url;
- (id)initWithCStringNoCopy:(char *)bytes length:(NSUInteger)length freeWhenDone:(BOOL)freeBuffer;
- (id)initWithCString:(const char *)bytes length:(NSUInteger)length;
- (id)initWithCString:(const char *)bytes;
+ (id)stringWithCString:(const char *)bytes length:(NSUInteger)length;
+ (id)stringWithCString:(const char *)bytes;
- (void)getCharacters:(unichar *)buffer;
```

Why YAGNI

- Retroactive Modeling
- Derivation
- ...

[1] Unicode specifies default ("un-tailored") locale-independent collation and segmentation algorithms that make reasonable sense in most contexts. Using these algorithms allows strings to be naturally compared and combined, generating the expected results when the content is ASCII

[2] Technically, == checks for Unicode canonical equivalence

[3] We have some specific ideas for locale-sensitive interfaces, but details are still TBD and wide open for discussion.

[4] Collections that automatically re-sort based on locale changes are out of scope for the core Swift language

[5] The type currently called `Char` in Swift represents a Unicode code point. This document refers to it as `CodePoint`, in anticipation of renaming.

[6] When the user writes a string literal, she specifies a particular sequence of code points. We guarantee that those code points are stored without change in the resulting `String`. The user can explicitly request normalization, and Swift can use a bit to remember whether a given string buffer has been normalized, thus speeding up comparison operations.

[7] Since `String` is locale-agnostic, its elements are determined using Unicode's default, "un-tailored" segmentation algorithm.