

## In this chapter, you'll learn:

- ## Binary representations

Character representation has changed so much over the years, starting from **ASCII** (American Standard Code for Information Interchange), which represents English numbers and characters using up to seven bits.

[illegible]

Then, Extended ASCII came along, which used the remaining 128 values representable by a single byte.

000	(nul)	016	► (dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001	Ⓔ (soh)	017	◄ (dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	Ⓕ (stx)	018	↑ (dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	♥ (etx)	019	!! (dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	♦ (eot)	020	⌘ (dc4)	036	\$	052	4	068	D	084	T	100	d	116	t
005	♣ (enq)	021	§ (nak)	037	%	053	5	069	E	085	U	101	e	117	u
006	♠ (ack)	022	— (syn)	038	&	054	6	070	F	086	V	102	f	118	v
007	• (bel)	023	⚡ (etb)	039	'	055	7	071	G	087	W	103	g	119	w
008	▣ (bs)	024	↑ (can)	040	(	056	8	072	H	088	X	104	h	120	x
009	(tab)	025	↓ (em)	041	)	057	9	073	I	089	Y	105	i	121	y
010	(lf)	026	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	♂ (vt)	027	← (esc)	043	+	059	;	075	K	091	[	107	k	123	{
012	♀ (np)	028	L (fs)	044	,	060	<	076	L	092	\	108	l	124	
013	(cr)	029	↔ (gs)	045	-	061	=	077	M	093	]	109	m	125	}
014	♯ (so)	030	▲ (rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	✱ (si)	031	▼ (us)	047	/	063	?	079	O	095	_	111	o	127	␣

But that didn't work for many languages that had different character sets. So another standard came out, called **ANSI**. Which is also the name of the entity that created this standard. American National Standards Institute.

Unlike ASCII, ANSI's not a single character set. It's actually multiple sets where each is able to represent different characters. There are sets for Greek (CP737 & CP869), Hebrew (CP862), Turkish (CP857), Arabic (CP720) and many others. Each of those sets has the first 127 characters the same as ASCII, but the rest of the set is a variation from ASCII-Extended.

Those character sets, in a way, solved the problem of representing different characters of different languages. But another problem came up! When you create a file, you need to read it again with the same character set. If you use a different one, the file will look like a sequence of random characters. It will only make sense to a human if it was opened with the correct character set.

For example, the character of byte hex value `0x9C`, when read with character set CP-852, aka Latin-2, will show the character `č` (Lower case t with caron). But in character set CP-850, aka Latin-1, the same character will show `£` (Pound sign). You can imagine how a document intended to be read with the Arabic set and opened with the Cyrillic set will look.

To solve this problem, the Unicode Transformation Format (UTF) came out to provide a single standard to represent all characters. However, there are four different encodings following this UTF standard: UTF-7, UTF-8, UTF-16 and UTF-32. Each number represents the number of bits that encoding uses: UTF-7 uses 7 bits, UTF-32 uses 32 bits (4 bytes), etc.

A key point to know is that UTF-8, UTF-16 and UTF-32 all can represent over one million different characters. It is clear that the latter of the group has a large range. As for the first, it's not limited to 8 bits only — it can expand over 4 bytes. To cover all possible values in the UTF standard requires 21 bits.

## UTF-8 binary representation

Each character in UTF-8 varies in size from 1 byte to 4 bytes. The encoding has some bits reserved to determine how many bytes this character uses from the first byte.



A byte with its most significant bit having **0** value is, on its own, a character. The character is 1 byte.



A byte with its three most significant bits having **110** value, along with the following byte, represent a character. The character is 2 bytes.



A byte with its four most significant bits having **1110** value, along with the two following bytes, represent a character. The character is 3 bytes.



A byte with its five most significant bits having **11110** value, along with the three following bytes, represent a character. The character is 4 bytes.



Any byte with its two most significant bits having **10** value is a byte that is part of a character (following byte). It doesn't provide enough information on its own without the leading byte.

The number of bits available to store the value for UTF-8 is calculated as follows:

- 1 byte: 8 bits - 1 reserved = 7 available bits
- 2 bytes: 16 bits - 5 reserved = 11 available bits
- 3 bytes: 24 bits - 8 reserved = 16 available bits

- 4 bytes: 32 bits - 11 reserved = 21 available bits

## UTF-16 binary representation

UTF-16 is another variable-length encoding format. A character can be 2 bytes or 4 bytes. Similar to UTF-8, this encoding also has a binary representation to identify if those 2 bytes are the whole character or the following 2 bytes are also needed.

If the 2 bytes start with `0xD8` (`110110` in binary), those two bytes complete a character. This character is 4 bytes in size.

1	1	0	1	1	0	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The following 2 bytes will start with `0xDC` (`110111` in binary). This makes for 4-byte values, with 12 bits reserved and 20 bits to define the value.

1	1	0	1	1	1	Y	Y	Y	Y	Y	Y	Y	Y	Y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

With those reserved values, characters can't be represented with values in the range between `0xD800` to `0xDFFF`, because doing so would confuse their values with length extensions.

## UTF-32 binary representation

It's obvious how UTF-32 works. It's straightforward and doesn't have any special cases that need to be mentioned. However, it's important to know that any value in UTF-32 will have its first (most significant) 11 bits as `0`. UTF possible values cover only 21 bits, and those 11 bits are never used.

It's worth noting that UTF-16 and UTF-32 aren't backward compatible with ASCII, but UTF-8 is. That means a file saved with ASCII encoding can still be read with UTF-8 encoding, but can't in the other two encodings.

## Human representation

Each representable value in a string is named a **code point** or **Unicode Scalar**. Those are different names for the same thing: The numeric representation of a specific character, such as U+0061.

Each number is represented by a different drawing, which is called **Character Glyph**. UTF, with all its variations, has the same mapping for each Unicode scalar to a glyph. The standards differ only in how the machine represents that scalar value.



For example, the Unicode scalar U+0061 represents the letter `a` (Latin lowercase letter "a"), and U+00E9 represents `é` (Latin lowercase letter "e" with acute).

Fonts are a pallet of glyphs that have a different drawing. Each glyph/letter is drawn in a different style, but in the end, they all have maps to a binary representation. A font affects only the rendering; it changes nothing in the stored information.

## Grapheme cluster

Knowing how UTF-8 and UTF-16 work to represent variable sizes, you can imagine that knowing the length of a string isn't as straightforward as it is for ASCII and ANSI representations. For the latter, an array of 100 bytes is simply 100 characters. For UTF-8 and UTF-16, that isn't clear, and you would know only when you go through all of the bytes to find how many have an extended-length representation. For UTF-32, this isn't an issue. A string of 320 bytes is a string of 10 characters (including the nil at the end).

To make it a little more complicated, say you have 4 bytes for a UTF-16 string, and there are no extended lengths. You would think that this means you have a string of length two. The answer is: *not necessarily!*

Take the character U+00E9  (Latin lowercase letter “e” with acute) as an example. It can be represented like that or by two Unicode scalar values of the standard letter  U+0065 (Latin lowercase letter “e”) followed by U+0301 (combining acute accent).

Open a new playground project and try the following:

```
import Foundation

let eAcute = "\u{E9}"
let combinedEAcute = "\u{65}\u{301}"
```

COPY



Those are the two representations, and they both represent .

```
eAcute.count // 1
combinedEAcute.count // 1
```

COPY



In Swift, both of those strings have a length of 1, although they have different binary sizes. Also, those strings are equal:

```
eAcute == combinedEAcute // true
```

COPY



When different Unicode sequences form the same result, those results are said to have **canonical equivalence**. **Swift equality checks the canonical equivalence of the content and not the absolute equality of the content.**

Try the same with Objective-C types:

```
let eAcute_objC: NSString = "\u{E9}"
let combinedEAcute_objC: NSString = "\u{65}\u{301}"

eAcute_objC.length // 1
combinedEAcute_objC.length // 2
```

COPY



```
eAcute_objC == combinedEAcute_objC // false
```

Objective-C String didn't read any of that. It simply compared the contents of the bytes. It didn't check its contents and didn't figure out that both represent the same thing.

A character in Swift doesn't represent a byte as in Objective-C. It represents a **grapheme cluster**, which can be one or more scalar values combined to represent a single glyph.

If you keep two characters separate that normally would form a grapheme cluster if merged together, they'll both be treated as normal characters. It's only when you merge them that they become a different character:

```
let acute = "\u{301}"
let smallE = "\u{65}"

acute.count // 1
smallE.count // 1

let combinedEAcute2 = smallE + acute

combinedEAcute2.count // 1
```

COPY



Now that you understand how characters are represented in zeros and ones, let's see how Swift works with them and how all those "under the hood" details affect how you can work with strings.

## UTF in Swift

Until Swift 4.2, Swift used UTF-16 as the preferred encoding. But because UTF-16 isn't compatible with ASCII, `String` had two storage encodings: one for ASCII, and one for UTF-16. Swift 5 and later versions use only UTF-8 storage encoding.

UTF-8 is the most common server-side encoding: Over 95% of the internet uses it. You might think for a moment that the internet isn't only English and UTF-16 is the more logical choice because less extended byte values will be used. But most of a webpage is HTML, and HTML can be completely represented in ASCII. This makes the usage of UTF-8 for internet content a better choice for size and transfer speed. That said, the change to UTF-8 storage encoding made any communication between Swift and a server straightforward, because they use the same encoding and therefore require no conversion.

## Collection protocol conformance

`String` conforms to the two collection protocols: `BidirectionalCollection` and `RangeReplaceableCollection`:

```
var sampleString = "Lōr'em, ìpsum, dōl,ōr' sīt 'a~m,et "
```

```
sampleString.last
```

COPY



```
// t'ḙḙ, a~ t'fī s r'ōl, o'd m, usp_ī m, er'ōl
let reversedString = String(sampleString.reversed())

if let rangeToReplace = sampleString.range(of: "Loṛ'ḙḙ") {
    // Lorem īp_sum, dōl, o'r' sīt' a~m, et'
    sampleString.replaceSubrange(rangeToReplace,
        with: "Lorem")
}
```

You can traverse a Swift String in either direction, and you can also replace a range of values. But it doesn't conform to `RandomAccessCollection`.

```
sampleString[2] ❌ 'subscript(_:)' is unavailable: cannot subscript String with an Int, use a String.Index instead.
```

You could extend `String` with `subscript(_:)` so you can easily access characters by their index:

```
extension String {
    subscript(position: Int) -> Self.Element {
        get {
            let characters = Array(self)
            return characters[position]
        }
        set(newValue) {
            let startIndex = self.index(self.startIndex,
                offsetBy: position)
            let endIndex = self.index(self.startIndex,
                offsetBy: position + 1)
            let range = startIndex..

```

COPY



Then the following code will work:

```
sampleString[2] // r
sampleString[2] = "R"

sampleString // LoRem īp_sum, dōl, o'r' sīt' a~m, et'
```

COPY



In the code above, there doesn't seem to be a problem. Try the following code:

```
for i in 0..

```

COPY



With a quick look, you would think this code has a complexity of  $O(n)$ , but that is incorrect. In the `subscript(_:)` implementation, you converted the string to an array to get the index you want. That itself is an  $O(n)$  operation, giving the loop you added a complexity of  $O(n^2)$ .



You can't reach the `n`th character directly without passing by the `n-1` characters first. A character — aka grapheme cluster — can be a long sequence of Unicode scalars, making the operation of reaching the `n`th character one of  $O(n)$ , not  $O(1)$ , thus not meeting the requirement of `RandomAccessCollection`.

Although the extension you created simplifies and shortens your code, it also affects the performance:

```
for element in sampleString {
    element.uppercased()
}
```

COPY



This code is the same. It didn't use the subscript approach and traverses the collection once. Using the subscript approach will often seem appealing, but that approach causes you to do many more operations than you think. Thus, understanding how the `String` class works, as well as what `Character` is and how Swift treats it, can make a huge difference in how you approach challenges and implement solutions.

## String ordering

You're already well acquainted with string comparison. The default sorting in a string ignores localization preference.

String comparison is always consistent, as it should be. However, for different locales, it should be different.

For example, the ordering of `ö` is different than `z` between German and Swedish:

```
let OwithDiaersis = "Ö"
let zee = "Z"

OwithDiaersis > zee // true

// German ☐☐
OwithDiaersis.compare(
    zee,
    locale: Locale(identifier: "DE")) == .orderedAscending // true

// Sweden ☐☐
OwithDiaersis.compare(
    zee,
    locale: Locale(identifier: "SE")) == .orderedAscending // false
```

COPY



When you're ordering text for internal use in the system, the locale must not affect it. But if you're ordering it to show it to the user, you must be aware of the differences.

Also, there is a notorious problem that arises when strings have numbers. A string with value `"11"` should be higher than a string of value `"2"`. But this isn't the case unless it is a comparison that is considering the locale:

```
"11".localizedCompare("2") == .orderedAscending // true
```

COPY





```
"11".localizedStandardCompare("2") == .orderedAscending // false
```

## String folding

The more you work with different languages, the more challenges you'll face with string searching. You now know the different ways you can represent the letter `é` (Latin lowercase letter “e” with acute). But the word `"Café"` doesn't match `"Cafe"`:

```
"Café" == "Cafe" // false
```

COPY



And checking if it contains the letter `e` (Latin lowercase letter “e”) will return false:

```
"Café".contains("e") // false
```

COPY



Using diacritics on a character transforms it into a different character. Although it originates from the same, comparing it to the original will fail — almost the same idea behind different cases:

```
"Café" == "café" // false  
"Café".contains("c") // false
```

COPY



When you want to compare strings and ignore casing, you convert the original string and keyword to the same casing, upper or lower. This is called **String Folding**, where you remove distinctions on the strings to make them suitable for comparison.

In the case of diacritics, you want to remove all of the marks and return all of the characters to their original letter to simplify comparison. To continue with our example, this would return `Café`, or any other diacritic variation of it, to `Cafe`.

Consider the following example:

```
let originalString = "H'ello_W'orld!"  
originalString.contains("Hello") // false
```

COPY



`originalString` contains a combining character for each letter in the string `Hello World!`. That makes it very hard to search for any words. Luckily, `String` provides a mechanism for folding so you can specify what distinctions you want to remove. Cases, diacritics, or both:

```
let foldedString = originalString.folding(  
  options: [.caseInsensitive, .diacriticInsensitive],  
  locale: .current)  
foldedString.contains("hello") // true
```

COPY



The options parameter in `folding(options:locale:)` gives you that control. In this

example, it removed both cases and diacritics. The resulting string is `hello world!`

Another, shorter way to do the same is by using `localizedStandardContains(_:)`:

```
originalString.localizedStandardContains("hello") // true
```

COPY



This method does the same. It performs a case- and diacritic-insensitive, locale-aware comparison. Without **folding** the string to remove the diacritics, you'll have a very hard time searching for text, or you'll give the user a very unpleasant experience.

## String and Substring in memory

Another tricky point related to performance in `String` is `Substring`. Just as how `String` conforms to `StringProtocol`, so does `Substring`.

As you can see from its name, a substring is a part of a string. And it is a very fast and optimized datatype when you're breaking down a large string. However, there is a key point that you should be aware of, especially when working with large strings:

```
func doSomething() -> Substring {
    let largeString = "Lorem ipsum dolor sit amet"
    let index = largeString.firstIndex(of: " ") ??
largeString.endIndex
    return largeString[..<index]
}
```

COPY



The code above returns the first word of a large string.

What you expect with a quick look is that you worked with the large string, finished using it, and returned only the small part of the string you need:

```
let subString = doSomething() // Lorem
subString.base // "Lorem ipsum dolor sit amet"
```

COPY



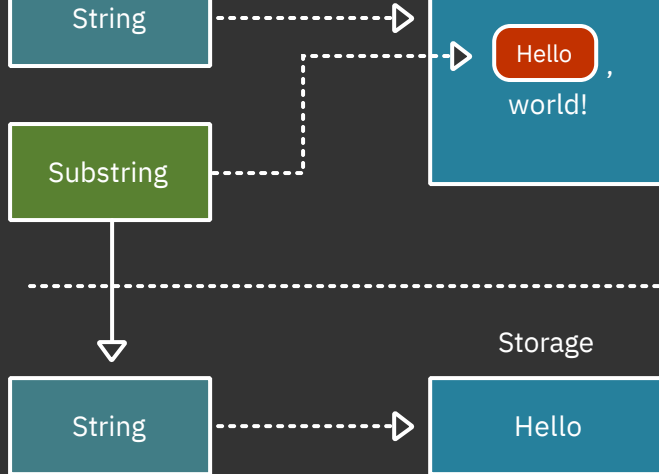
You still have the large string loaded in memory. `Substring` shares memory with the original string. If you're working with a large string and need a lot of smaller strings from it, *while still using the large string*, there will be no additional memory cost. But if you want to just break it and remove the large string from memory, then you need to create a new string object from your substring right away:

```
let newString = String(subString)
```

COPY



If you don't, the original string will stay in memory for much longer without your awareness.



That was a lot of info about `String`. The next part will cover a very interesting perk from Swift that you've been using frequently. You'll know how it works under the hood and builds on top of it.

## Custom string interpolation

String interpolation is a powerful tool for creating strings. But it's not narrowed to the creation of strings. Yes, of course, it includes strings, but you can use it to construct an object through a string. Yes, I know it's confusing.

Consider the following type:

```
struct Book {
    var name: String
    var authors: [String]
    var fpe: String
}
```

COPY



Wouldn't it be super cool if you could define a new instance from `Book` with a string like `"Expert Swift by: Ehab Amer, Marin Bencevic, Ray Fix, Shai Mishali"`?

Swift allows you to define any type by a string literal by conforming to the protocol `ExpressibleByStringLiteral`, and implementing `init(stringLiteral value: String)`.

Add this extension:

```
extension Book: ExpressibleByStringLiteral {
    public init(stringLiteral value: String) {
        let parts = value.components(separatedBy: " by: ")
        let bookName = parts.first ?? ""
        let authorNames = parts.last?.components(separatedBy: ",") ?? []
        self.name = bookName
        self.authors = authorNames
        self.fpe = ""
    }
}
```

COPY



You break down the string into two parts with the `" by: "` separator: The first part is the book name, and the second part is the author names, comma-separated. Ignore the “fpe” (final pass editor) for now, but you’ll use this property later.

The string defining the book should be **[Book name] + by: + Author1,Author2,Author3, ....**:

```
var book: Book = """
Expert Swift by: Ehab Amer,Marin Bencevic,\
Ray Fix,Shai Mishali
"""

book.name // Expert Swift
book.authors.first // Ehab Amer
```

COPY



This is a very human-friendly way to construct your object, but if anything changed in this format, unexpected data will be saved in the object!

```
var invalidBook: Book = """
Book name is `Expert Swift`. \
Written by: Ehab Amer, Marin Bencevic, \
Ray Fix & Shai Mishali
"""

invalidBook.name // Book name is `Expert Swift`. Written
invalidBook.authors.last // Ray Fix & Shai Mishali
```

COPY



Now, the name contains invalid information, and the last author is actually two of them together. You can fix this by improving the implementation of `init(stringLiteral value: String)`, but will you ever be able to expect all possible inputs to make sure that the string will be parsed properly?

There is another way you can construct `Book`: using string interpolation. To do this, you define a string that has clear, explicit mention of the book name and the array of authors:

```
extension Book: ExpressibleByStringInterpolation { // 1
    struct StringInterpolation: StringInterpolationProtocol { // 2
        var name: String // 3
        var authors: [String]
        var fpe: String

        init(literalCapacity: Int, interpolationCount: Int) { // 4
            name = ""
            authors = []
            fpe = ""
        }

        mutating func appendLiteral(_ literal: String) { // 5
            // Do something with the literals?
        }

        mutating func appendInterpolation(_ name: String) { // 6
            self.name = name
        }
    }
}
```

COPY



```

mutating func appendInterpolation(
    authors list: [String]) { // 7
    authors = list
}

init(stringInterpolation: StringInterpolation) { // 8
    self.authors = stringInterpolation.authors
    self.name = stringInterpolation.name
    self.fpe = stringInterpolation.fpe
}
}

```

1. To use custom interpolation to define a `Book`, you need it to conform to `ExpressibleByStringInterpolation`.
2. This requires defining a struct with the name `StringInterpolation` that conforms to `StringInterpolationProtocol`. The visibility of this struct is only from within the `Book` type.
3. The new struct must carry properties to store the values that will be provided in the string. For this example, `name` and `authors` will do. You can also have any properties you may need. Ignore `fpe` for now.
4. The string will contain literal strings and interpolations. This initializer is the first that gets called. It provides the counts of every character in the literal and the number of interpolations present.
5. This gets called for literals in the string. For this example, do nothing with them. This method declaration identified the generic type for the literal as `String`.
6. This added an interpolation with a string that defines the name of the book. Interpolation should look like `"\ (String) "`
7. This added an interpolation signature that looks like `"\ (authors: [String]) "`. This is a labeled interpolation for the authors list.
8. You define a new initializer with a parameter of type `StringInterpolation`, which is the same struct you defined.

Now you can create an instance of `Book` like this:

```

var interpolatedBook: Book = """
The awesome team of authors \ (authors:
    ["Ehab Amer", "Marin Bencevic", "Ray Fix", "Shai Mishali"]) \
wrote this great book. Titled \ ("Expert Swift")
"""

```

COPY



The book was defined with a lot more description. Even the list of authors came before the name of the book. But because each interpolation has its form, either through a label and/or data-type, there was no mixup.

What actually happened behind the scenes is as follows:

```

let stringInterpolation = StringInterpolation(

```

COPY



```

literalCapacity: 59,
interpolationCount: 2)

stringInterpolation.appendLiteral("he awesome team of authors ")

stringInterpolation.appendInterpolation(
    authors: ["Ehab Amer",
              "Marin Bencevic",
              "Ray Fix",
              "Shai Mishali"])

stringInterpolation
    .appendLiteral(" wrote this great book. Titled ")

stringInterpolation
    .appendInterpolation("Expert Swift")

Book(stringInterpolation: stringInterpolation)

```

`init(literalCapacity: Int, interpolationCount: Int)` is called with the number of total character literals and the number of interpolations.

Then, for each literal sequence, `appendLiteral(_:)` is called. After that, for each interpolation, its appropriate method is called. Finally, the initializer is called with the interpolation object.

Notice that each interpolation was translated to a method. `\(_:)` was translated to `appendLiteral(_:)`, and `\(authors:)` was translated to `appendLiteral(authors:)`.

Remember the `fpe` that you didn't use? So far, you focused only on the title and authors of the book. But at the point of creating the interpolation object, you had no use for this property and left it empty.

Add an extension to `StringInterpolation` defined inside `Book`:

```

extension Book.StringInterpolation {
    mutating func appendInterpolation(fpe name: String) {
        fpe = name
    }
}

```

COPY



Then, define a new book with this interpolation:

```

var interpolatedBookWithFPE: Book = ""
\("Expert Swift") had an amazing \
final pass editor \(fpe: "Eli Ganim")
""

```

COPY



This created a new instance of a book and used the interpolation you identified in the extension to set `fpe`. You can define as many additional interpolation formats as you wish:

```

extension Book.StringInterpolation {

```

COPY



```
mutating func appendInterpolation(bookName name: String) {
    self.name = name
}

mutating func appendInterpolation(anAuthor name: String) {
    self.authors.append(name)
}
}
```

This added an alternative way to define the name of the book and a way to add authors one by one:

```
var interpolatedBook2: Book = ""
\ (anAuthor: "Ray Fix") & \ (anAuthor: "Shai Mishali") \
were authors in \ (bookName: "Expert Swift")
""
```

The type `String` is no different from `Book`. You have already been using its `StringInterpolation` subtype for some time with your standard interpolations, such as including a number in a string:

```
var num = 1234
var string = "The number is: \ (num) "
```

Just as you added a new interpolation for `fpe` on `Book`, you can do the same on `String` to interpolate `Book`.

Try to include the first book instance into a string:

```
var string = "\ (book) "
// Book(name: "Expert Swift", authors: ["Ehab Amer", "Marin
Bencevic", "Ray Fix", "Shai Mishali"], fpe: "")
```

The string doesn't have a friendly representation of the book. But you can control that. Add an extension to `StringInterpolation` inside `String`:

```
extension String.StringInterpolation {
    mutating func appendInterpolation(_ book: Book) {
        appendLiteral("The Book ")
        appendLiteral(book.name)
        appendLiteral("\ ")

        if !book.authors.isEmpty {
            appendLiteral(" Authored by: ")
            for author in book.authors {
                if author == book.authors.first {
                    appendLiteral(author)
                } else {
                    if author == book.authors.last {
                        appendLiteral(", & ")
                        appendLiteral(author)
                        appendLiteral(".")
                    }
                }
            }
        }
    }
}
```



```

    } else {
        appendLiteral(", ")
        appendLiteral(author)
    }
}
}

if !book.fpe.isEmpty {
    appendLiteral(" Final Pass Edited by: ")
    appendLiteral(book.fpe)
}
}
}

```

Add the `fpe` to `interpolatedBook` object you defined earlier, and convert it to a string:

```

interpolatedBook.fpe = "Eli Ganim"
var string2 = "\(interpolatedBook)"
// The Book "Expert Swift" Authored by: Ehab Amer, Marin Bencevic,
Ray Fix, & Shai Mishali. Final Pass Edited by: Eli Ganim

```

COPY



Now, this is a much more friendly way to describe a book.

In the extension, you had full control over how the fields were printed, their order and what user-friendly text to precede and/or follow each property.

The reason `appendLiteral(_:)` was heavily used here is that you don't know the internal implementation of `String.StringInterpolation`, and you don't know what temporary fields it has to store the information. But it's not like `Book.StringInterpolation`. The literals are stored just like interpolations and in order, so you can safely convert an interpolation to a series of literals. In the end, it is only **one** string. Not **multiple** fields like in `Book`.

## Key points

- ASCII was the first standard for storing characters, and it evolved to UTF to represent all the possible characters in one single standard.
- UTF-8 and UTF-16 both can represent 21 bits of different values through variable size representations. A UTF-8 character can take up to 4 bytes.
- UTF-16 and UTF-32 aren't backward compatible with ASCII.
- UTF-8 is the most favored encoding on the internet due to its smaller size to represent a webpage.
- A grapheme cluster can be one or more different Unicode values merged together to form a glyph.
- A character in Swift is a grapheme cluster, not a Unicode value. And the same cluster can be represented in different ways. This is called canonical equivalence.
- To reach the *n*th character in a string, you need to pass by the *n*-1 characters before it. It is **not** an  $O(1)$  operation.

- The order of strings can vary based on the locale.
- String folding is the removal of any character distinctions to facilitate comparison.
- `Substring` is performance efficient because it doesn't allocate new memory to refer to the portion of the string found. However, this means that the original string is still present in memory.
- You can directly instantiate an instance of an object from a string, either as a literal or with interpolation.
- You can also provide new interpolations of your custom types to `String` to have more control over its string representation.

✓ Completed

---

## 8. Codable



## 6. Sequences, Collections & Algorithms

---

**Have a technical question? Want to report a bug?** You can ask questions and report bugs to the book authors in our official book forum [here](#).

**Have feedback to share about the online reading experience?** If you have feedback about the UI, UX, highlighting, or other features of our online readers, you can send them to the design team with the form below:

Feedback about the UI, UX, or other features of the online reader? Leave them here!

Send Feedback