GUIDE TO SWIFT STRINGS

// VOL 3 // MATTT

flight School

Flight School
An imprint of Read Evaluate Press, LLC
Portland, Oregon
https://readeval.press

Publisher	Mattt Zmuda
Illustrator	Lauren Mendez
Cover Designer	Mette Hornung Rankin
Editor	Morgan Tarling

ISBN 978-1-949080-06-3

Printed in IBM Plex, designed by Mike Abbink at IBM in collaboration with Bold Monday.
Additional glyphs provided by Google Noto Sans.
Twemoji graphics made by Twitter and other contributors.
Cover type set in Tablet Gothic, designed by José Scaglione & Veronika Burian.

Contents


salve
ようこそ
Welcome
BIENVENUE
Valkommen
welkom
ARRIVALS

Chapter 1:
Introduction to Unicode

Unicode is the standard way for computers to encode, represent, and process text written in most written languages.[1]

The Unicode Standard encodes *characters* rather than *glyphs*, which is to say the idea rather than any particular representation. For example, the code point U+0041 always represents Latin Capital Letter A, no matter which font is used to render it.

- **Abstract Characters** are the smallest meaningful units within a writing system
- **Coded Characters** assign a name and a code point to identify each abstract character
- **Character Encoding Forms** determine how coded characters are represented by computers
- **Character Encoding Schemes** specify how encoded representations are serialized into bytes

Abstract Characters

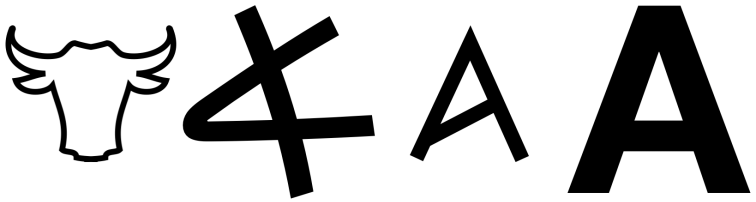
Nearly every writing system attempts to solve the same problem: representing language separately from its essential, spoken (or signed), utterance.

There's a fascinating amount of variation throughout history and across cultures. Many cultures independently arrive at the same patterns; others come up with entirely unique and surprising results. As a broad generalization, many writing systems began as pictorial representations of objects in the world. Over time, these

1. Unicode® is a registered trademark of Unicode, Inc.

pictures start to be more closely associated with the pronunciation of words than the words themselves, and then get abstracted and used to represent the sound alone.

For example, this is true of the writing systems that produced modern Latin script.



We went from a pictorial prototype of an ox (𓃾) to the Phoenician notation (𐤀), which eventually became associated with the pronunciation of the word (*'alp*), as in the case of Greek alpha (Α), until it was purely phonetic with Latin Capital Letter A.[2]

Eastern writing systems evolved in a similar manner:

However, in the case of Chinese, these pictorial representations didn't develop into an alphabet. Instead of representing a particular sound, each character represents an individual word.

A *character*[3] is the atomic unit of a writing system, we can categorize all writing systems by what that unit signifies:

---

2. The word for ox was also used to mean "leader", because oxen were used to lead plows and wagons. This is why A is the first letter of the alphabet. (If you've ever wondered why it's called an *alphabet*, it comes from the first two Greek letters: *alpha* and *beta*)
3. The OED credits William Caxton with the first use of the word "character" in 1490 to describe written or printed letters.

- In *alphabets* like Latin, Greek, and Cyrillic, each character represents a consonant or a vowel sound, or *phoneme*.
- In *abjads*, including Arabic, Hebrew, and Urdu, only *consonants* are written.
- In *abugidas* such as Devanagari, Telugu, and Ge'ez, a character represents a *consonant-vowel cluster*.
- In *syllabaries* including Japanese Hiragana and Katakana, a character represents a *syllable*.[4]
- In *logographic* scripts like Simplified and Traditional Chinese, each character represents a *word*.[5]
- In *ideographic* scripts like emoji and various technical notations, each character represents an *idea*.

One or more characters can combine to form words and sentences as well as paragraphs and other forms of discourse. In computing, we refer to a sequence of characters as a *string*.[6]

Coded Characters

Unicode assigns each abstract character a unique *name* and *code point*. For example, the letter "A" is assigned the name LATIN CAPITAL LETTER A and the code point U+0041.

---

4. Or more precisely, a *mora*; the long vowels in the word 東京 (とうきょう, tōkyō) are each counted as one syllable, but comprise two moras.
5. If you learn Chinese as a second language, you may be "tricked" into thinking that the writing system is easier than it actually is by learning the simple ideographic characters first, like 上 (shàng) meaning "up" or 下 (xià) meaning "down", or semantic compounds like the word 休 (xiū) "rest" deriving from 人 (rén) "person" leaning on a 木 (mù) "tree". However, the majority of Chinese characters consist of a mix of phonosemantic compounds that can't be understood entirely from their written form.
6. The etymology for "string" in this sense is uncertain. One theory is that it comes from an early 19th century expression concerning honorific letters at the end of a personal name. Another possibility is that it has to do with how typesetters would bill according to the length of string used to bind a block of text. However, we can definitively point to "strings" and "characters" used in the context of computing as early as 1949 in George Patterson's paper from the Second Symposium on Large-Scale Digital Calculating Machinery.

Note: Each coded character maps to a single abstract character, but an abstract character can map to multiple coded characters for compatibility with previous standards.

Character Names

A Unicode character name consists of uppercase letters (A–Z), digits (0–9), hyphens, and spaces. Most characters are assigned descriptive names, like GREEK SMALL LETTER BETA (β), HANGUL SYLLABLE CAEG (책), and OPEN BOOK (📖). However, some character names include their code point, like CJK UNIFIED IDEOGRAPH-672C (本); this is most often the case for ideographs and historical scripts, which can be challenging to name for lack of a single known meaning or pronunciation.

Once a name is assigned, it never changes — even if the original name was misspelled or a more descriptive name is later determined. Instead, any corrections are issued by assigning a *formal alias*.[7]

In addition to its normative name, a character may have one or more *informal names* assigned. For example, the character U+1F41E LADY BEETLE (🐞) has the assigned name "lady beetle" but provides aliases for its British name (ladybird), American name (ladybug), and Scientific name (coccinellids).

Code Points

A code point is a unique number assigned to a Unicode character. Code points are typically represented by "U+" followed by 4 – 6 hexadecimal digits (smaller values are padded with leading zeroes). For example, the character U+0041 LATIN CAPITAL LETTER A is assigned a code point with numeric value of sixty-five, which is represented in hexadecimal (base-16) as 41.

7. For a list of known anomalies in Unicode character names, see Unicode Technical Note #27

The Unicode Code Space and its Constituent Planes

Unicode defines a *code space*, or range of code points, between U+0000 – U+10FFFF. As of the Unicode 11.0 release in June 2018, a total of 137,439 characters have been encoded so far, or roughly 10% of the allocated space.

These 1,114,112 code points are divided into 17 *planes*, each consisting of 2^{16}, or 65,536 characters.

- **Plane 0** (U+0000 – U+FFFF) is called the Basic Multilingual Plane (BMP) and it contains the most frequently used characters, including Latin, Aramaic, Brahmic, Cyrillic, CJK scripts and symbols.
- **Plane 1** (U+10000 – U+1FFFF) is called the Supplementary Multilingual Plane (SMP). It contains historical scripts like Egyptian Hieroglyphs, pictorial symbols like musical notation and playing cards, as well as emoji.
- **Plane 2** (U+20000 - U+2FFFF) is called the Supplementary Ideographic Plane (SIP) and it contains CJK ideographs that weren't part of earlier character encoding standards.
- **Planes 3 to 13** (U+30000 – U+DFFFF) are currently empty, with no characters yet assigned to them.
- **Plane 14** (U+E0000 - U+EFFFF) is called the Supplementary Special-Purpose Plane (SSP). This, like its preceding blocks, is mostly unassigned, but it does, notably, contain the regional indicator characters used to form country flag emoji.
- **Planes 15 and 16** (U+F0000 - U+10FFFF) are designated as Private Use Areas (PUA) that vendors can use to refer to auxiliary glyphs.[8][9]

---

8. Unicode defines three PUA blocks, of which two comprise the entirety of Planes 15 and 16. The eponymous Private Use Area block (U+E000 – U+F8FF) can be found in the BMP, at the edge of Unicode's original 16-bit code space.
9. For example, the key combination ⇧ ⌥ K produces the character on macOS, because Apple designates the private-use area character U+F8FF to be the Apple logo.

Surrogate Code Points

The first version of Unicode defined a 16-bit code space, which allowed for up to 2^{16}, or 65,536 characters, total. Over time, this was found to be insufficient, and the size was increased to the 21-bit code space we know today.

To extend the original coding scheme, two blocks of 10-bit (1,024) unused code points were reserved:

- High Surrogates (U+D800 – U+DBFF)
- Low Surrogates (U+DC00 – U+DFFF)

These are collectively known as *surrogate code points*, and their purpose will become clearer in the next section when we explain character encoding forms (specifically, UTF-16).

One other thing to mention here is the term *scalar value*, which is defined as any Unicode code point that *isn't* a surrogate.

Character Encoding Forms

Having assigned a unique number to each character, the next question is how to represent those numbers on a computer. This is the responsibility of a *character encoding form*, defines how a scalar value is mapped onto one or more *code units*.

Computers store and process information in 8-bit chunks called *bytes*. Unicode has a code space 21-bits in size, and defines three different methods for representing characters: UTF-8, UTF-16, and UTF-32.[10] Each transformation format differs by the size of code unit (8, 16, and 32 bits, respectively) as well as the conversion algorithm.

---

10. Collectively referred to as UTF, or Unicode Transformation Format.

UTF-8 and UTF-16 are *variable-width encodings*, and UTF-32 is a *fixed-width* encoding; a code point can be encoded with between 1 to 4 UTF-8 code units, and either 1 or 2 UTF-16 code units, or a single UTF-32 code unit.

You can represent every scalar value with any of Unicode encoding forms — which is to say that you wouldn't choose one over another strictly based on which characters you want to be able to represent. Instead, each encoding form comes with its own set of trade-offs.

Tip:

To better understand why one encoding form might be used instead of another, consider the analogy of shipping cargo by plane: If you want to send a message from one location to another, you must send each character in sequence from the origin to the destination. Imagine that, to send a character, you need to pack up each of the hexadecimal digits in its code point (excluding leading zeroes). So U+0041 LATIN CAPITAL LETTER A has a payload size of 2 and U+3042 HIRAGANA LETTER A has a payload size of 4. The choice between UTF-8, UTF-16, and UTF-32 would be analogous to having a fleet of small, mid-sized, and jumbo jets with cargo capacity of 2, 4, and 8 hexadecimal code point digits. (Let's say that, for balance reasons, digits must be transported in pairs)

When you send a character, you divide up the cargo across as many planes as it takes, and then fly them together in formation. Each plane in this analogy is equivalent to a code unit.

"A" has a relatively small payload and can be transported by a single plane of any size. "あ", meanwhile, is too big for a single small plane, but can fit in a mid-sized one. Flying a smaller plane is more efficient than flying one that's mostly empty, but efficiency may not always be what you're optimizing for. You might decide that a jumbo jet is sensible if you value the simplicity of 1 plane = 1 character, no matter what (besides, it can be tricky to figure out which order each of the smaller planes arrived). Or maybe you don't have a say in which aircraft you use, and you've inherited a mid-sized fleet from the 1990's.

UTF-8

UTF-8 represents each Unicode character using between 1 to 4 8-bit code units. The following table shows how ranges of scalar values are represented in UTF-8:

Unicode Scalar Value	Unit 1	Unit 2	Unit 3	Unit 4
U+0000 – U+007F	0xxxxxxx			
U+0080 – U+07FF	110xxxxx	10xxxxxx		
U+0800 – U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000 – U+1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

For example, the code point for U+1F6EB AIRPLANE TAKING OFF (🛫) is expressed in binary as `0b11111011011101011`. It falls into the range of the final row, and is therefore encoded by 4 UTF-8 code units. The binary representation of the code point value fills the allocated digits from least to most significant bit (right-to-left):

Format	11110___	10_____	10_____	10_____
Code Point	_____000	__011111	__011011	__101011
UTF-8 Encoding	11110000	10011111	10011011	10101011

UTF-8 coincides with ASCII — which is to say that if a string contains only ASCII characters, its representation in either encodings is identical. This backwards compatibility offers a built-in advantage that has made UTF-8 the most popular encoding on the web (used by over 90% of all websites with known encodings).

UTF-16

UTF-16 represents each Unicode character in 1 or 2 16-bit code units. The following table shows how ranges of scalar values are represented in UTF-16:

Unicode Scalar Value	Unit 1	Unit 2
U+0000 – U+D7FF	xxxxxxxxxxxxxxxx	
U+E000 – U+10FFFF	110110xxxxxxxxxx	110111xxxxxxxxxx

For example, the code point for U+1F6EC AIRPLANE LANDING (🛬) is expressed in binary as `0b11111011011101100`. It falls into the

range of the final row, and is therefore encoded by 2 UTF-16 code units. The binary representation of the code point value fills the allocated digits from least to most significant bit (right-to-left):

Format	110110_____	110111_____
Code Point	_____0000111101	_____1011101100
UTF-8 Encoding	1101100000111101	1101111011101100

UTF-16 has the advantage of being able to represent the entire Basic Multilingual Plane (BMP) in a single code unit. This would make UTF-16 quite efficient for encoding most human communication... that is, if emoji hadn't come to dominate all human communication 🙃.

UTF-16 is the native encoding for many String implementations in languages and platforms that were around for Unicode 1.0, which was originally a 16-bit encoding standard.

UTF-32

UTF-32 represents each Unicode character in a single 32-bit code unit. The following table shows how ranges of scalar values are represented in UTF-32:

Unicode Scalar Value	Unit 1
U+0000 – U+10FFFF	00000000000xxxxxxxxxxxxxxxxxxxx

As you can see, the table is mostly a formality here. 32 bits can comfortably fit any of the 21-bit Unicode scalar values with room to spare. However, because of this, UTF-32 isn't a particularly efficient encoding format. For instance, a text document consisting entirely of ASCII characters would require 4 times as much memory when encoded in UTF-32 compared to UTF-8.

Character Encoding Schemes

When the code units for a character encoding are larger than a single byte, the computer has to decide which order to send those bytes. If you send the most significant byte first, that's called *big-endian* format; starting with the least significant byte is called *little-endian* format. There's no inherent benefit to doing it one way or another — it's just a matter of convention.

UTF-8 has a code size equal to a single byte, so there's only one way to serialize that encoding. UTF-16 and UTF-32, on the other hand, have three different *character encoding schemes*.

- UTF-16LE and UTF-32LE are little-endian
- UTF-16BE and UTF-32BE are big-endian
- UTF-16 and UTF-32 without an explicit designation can be either little-endian or big-endian

Most languages and frameworks abstract these kinds of details from developers, but it's helpful to at least be aware of the possible distinction.

Now that we've constructed the entire character model, let's look at a few examples:

Abstract Character	The letter "A" in the Latin alphabet
Coded Character	U+0041 LATIN CAPITAL LETTER A
UTF-8 Encoding Form	41

The abstract character A from Latin script is represented by the coded character LATIN CAPITAL LETTER A and its code point (U+0041) is encoded in UTF-8 by a single byte.

Abstract Character	The letter "あ" in the Hiragana syllabary
Coded Character	U+3042 HIRAGANA LETTER A
UTF-8 Encoding Form	E3 81 82

The abstract character あ from Hiragana script is represented by the coded character HIRAGANA LETTER A and its code point (U+3042) is encoded in UTF-8 by three bytes.

Abstract Character	The letter "𐤀" in Phoenician script
Coded Character	U+10900 PHOENICIAN LETTER ALF
UTF-8 Encoding Form	`F0 90 A4 80`

The abstract character 𐤀 from Phoenician script is represented by the coded character PHOENICIAN LETTER ALF and its code point (U+10900) is encoded in UTF-8 by four bytes.

Note: Phoenician script has its own Unicode block (U+10900 – U+1091F) in the SMP.

Character Properties

So far, we've discussed how the Unicode standard encodes and represents sequences of characters. But more information is necessary for software to display and process text correctly — to know, for example, that Hebrew letters are written right-to-left or how Korean letters are grouped into syllabic blocks or even how uppercase and lowercase Latin letters map to one another.

To facilitate this, the Unicode Character Database (UCD) maintains a set of properties for each character. These properties are used by different algorithms to determine their behavior during processing.

At a high level, characters can be classified according in the following way:

- *Graphic characters* including letters, numbers, symbols, marks, punctuation, and whitespace
- *Format characters* are invisible but affect neighboring characters, like line and paragraph separators
- *Control characters* which are inherited from ASCII
- ...and the rest: *Private Use Area*, *noncharacter scalar values*, *surrogate code points*, and *reserved scalar values*

Composition

Linguistics has a term, *markedness*, that describes how a word like "pilots" can be seen as a variant, or *marked* form of the word "pilot". The "-s" ending isn't itself a word, but when combined with another word in English, can change the meaning of the original. It is, to use another linguistics term, *productive*.

Marked forms exist in writing systems as well, including accents in Latin script (e -> é), nuqtā in Indic scripts like Devanagari (क -> क़), and annotations for mathematical symbols (x -> x̄)

Rather than attempting to encode every possible combination, Unicode provides a mechanism called *dynamic composition* that encodes marks as combining characters. Additional marks can be attached, stacking outwards from the base form. For example, the character é can be encoded as the base character U+0065 LATIN SMALL LETTER E followed by U+0301 COMBINING ACUTE ACCENT.

Precomposition

For compatibility with existing standards, some of these combinations are encoded as as single characters. These are called *precomposed characters*. From the previous example, the letter é can also be expressed by the single character U+00E9 LATIN SMALL LETTER E WITH ACUTE.

Canonical Equivalence

To deal with ambiguity between dynamic composition and precomposed characters, Unicode defines a notion of *canonical equivalence*. The character é when expressed as a single character, is canonically equivalent to é when it's formed with a combining mark.

Another example where canonical equivalence is used is the Korean alphabet, *Hangul*. Hangul consists of 24 letters called *jamo*: 14 consonants and 10 vowels, which are grouped into syllabic blocks. For example, given the consonant ㄱ (*g*) and vowel ㅏ (*a*), the sequence of ㄱ, ㅏ, and ㄱ is written in a single character, 갂 (*gag*), rather than ㄱㅏㄱ. Unicode encodes both the individual jamo as well as the most common combinations that form syllables.

We'll see how these variations interact when we discuss Normalization Forms.

Character Algorithms

Unicode specifies a number of algorithms that use the aforementioned character properties for performing common operations. These include *normalization*, *case mapping*, *collation*, and *bidirectionality*.

Unless you're specifically working on the text system for a language or framework, this functionality isn't something you need to implement yourself. However, knowing about how these algorithms work generally is important to working with text correctly and efficiently.

Normalization

As discussed previously, there are sometimes multiple ways to express equivalent text. Normalizing text to use the same kind of representation consistently can make it easier and faster to perform

certain operations. For example, you can determine if two strings are equal by normalizing their representations and performing a binary comparison.

There are two basic *normalization forms*: *NFC*, in which characters are precomposed, and *NFD*, in which characters are decomposed.[11] [12]

Case Mapping

Basic latin script has 26 letters, A through Z with capital and miniscule forms, or *cases*. Other scripts, including Cyrillic and Greek also have lowercase and uppercase letter variations. Unicode provides algorithms for mapping the correspondence between these cases.

> **Note:** The origin of terms lowercase and uppercase comes from printing terminology When compositors arranged type, they would traditionally store capital letters in the upper drawer or *case*, and miniscule letters below them.

Case mapping is deceptively complicated. There are *simple mappings*, such as in Latin script, where characters have bidirectional correspondences independent of any context: A ↔ a, B ↔ b, C ↔ c, and so on. But some characters require *complex mappings*. For example, the lowercase Greek letter sigma has different forms depending on whether they come at the end of a word, as in the first and last letter in the word σας (meaning “you” or “your”).

Unicode actually defines three different cases: in addition to lowercase and uppercase, there’s titlecase as well. For basic Latin characters, the titlecase form of a word is derived by making the

11. Unicode defines two additional normal forms: NFKD and NFKC. As their names imply, they’re derived from NFD and NFC; the K in the name is meant to denote that they apply compatibility decompositions as their first step.

12. Unicode also specifies two “fast” algorithms: FCD and FCC. For more information, see Unicode Technical Note #5.

first letter uppercase and the rest lowercase. The extended Latin character block includes digraphs like dž which have distinct forms for uppercase and titlecase (DŽ versus Dž).

Some locales may specify *tailoring* conventions for titlecase. For example, the English title of this book is *Guide to Strings* (with the word "to" lowercased). A hypothetical French language translation might by titled *Le guide des chaînes de caractères* (with only the first word capitalized).

Be mindful of these complexities and use provided methods for case transformation and comparison rather than attempting to implement this functionality yourself.

Collation

Sorting is another important text processing operation.

We use the term *collation* to describe the assembly of text into a standard order for a given locale.

Here are some examples of the variation in lexicographic ordering conventions found around the world:

- In English, words that differ only by accents are compared only by the first accent difference, whereas in French only the last accent difference is compared.
- In German, the letter "ö" is equivalent to the letters "oe" and therefore sorts between "od" and "of" — but only in dictionaries; phone books use a different convention.
- In Swedish, the letter "ö" comes after the letter z.
- In Spanish, the characters "ch" can be considered to be a single letter that's sorted between "c" and "d".

And that's to say nothing of ideographic characters in East Asian scripts, which are ordered by stroke count, stroke order, and radicals.

We could spend the dedicate of the book attempting to cover all of the intricacies in the Unicode collation algorithm,[13] but all you really need to know about it is that collation order depends on locale, but Unicode provides a fast algorithm that works well the vast majority of the time. As with case mapping, be sure to use system-provided methods anytime you sort or compare strings — especially within the context of the user's locale.

Bidirectionality

Unicode text is stored in logical order, which is to say the order in which characters are read. This may be different from how they appear written on a page or displayed on a screen. Latin, Cyrillic, Devanagari, and other scripts are written left-to-right, whereas Arabic, Hebrew, and other scripts are written right-to-left.

The *bidirectional algorithm* determines the visual order of text using the *directionality* property of characters. For example, Latin characters have strong Left-to-Right directionality, whereas Arabic letters have strong Right-to-Left directionality. Some characters, like numbers, have weak directionality and instead adapt to their surroundings. Other characters, like paragraph separators, have neutral directionality. You can also implicitly or explicitly modify the directional context of text using one of the invisible directional formatting characters.[14]

Now that we have a solid foundation for understanding Unicode, we're ready to learn how strings work in Swift. As we'll see, the design of Swift's `String` type reinforces the concepts discussed in this chapter, including extended grapheme clusters, scalar values, encoding forms, and canonical equivalence.

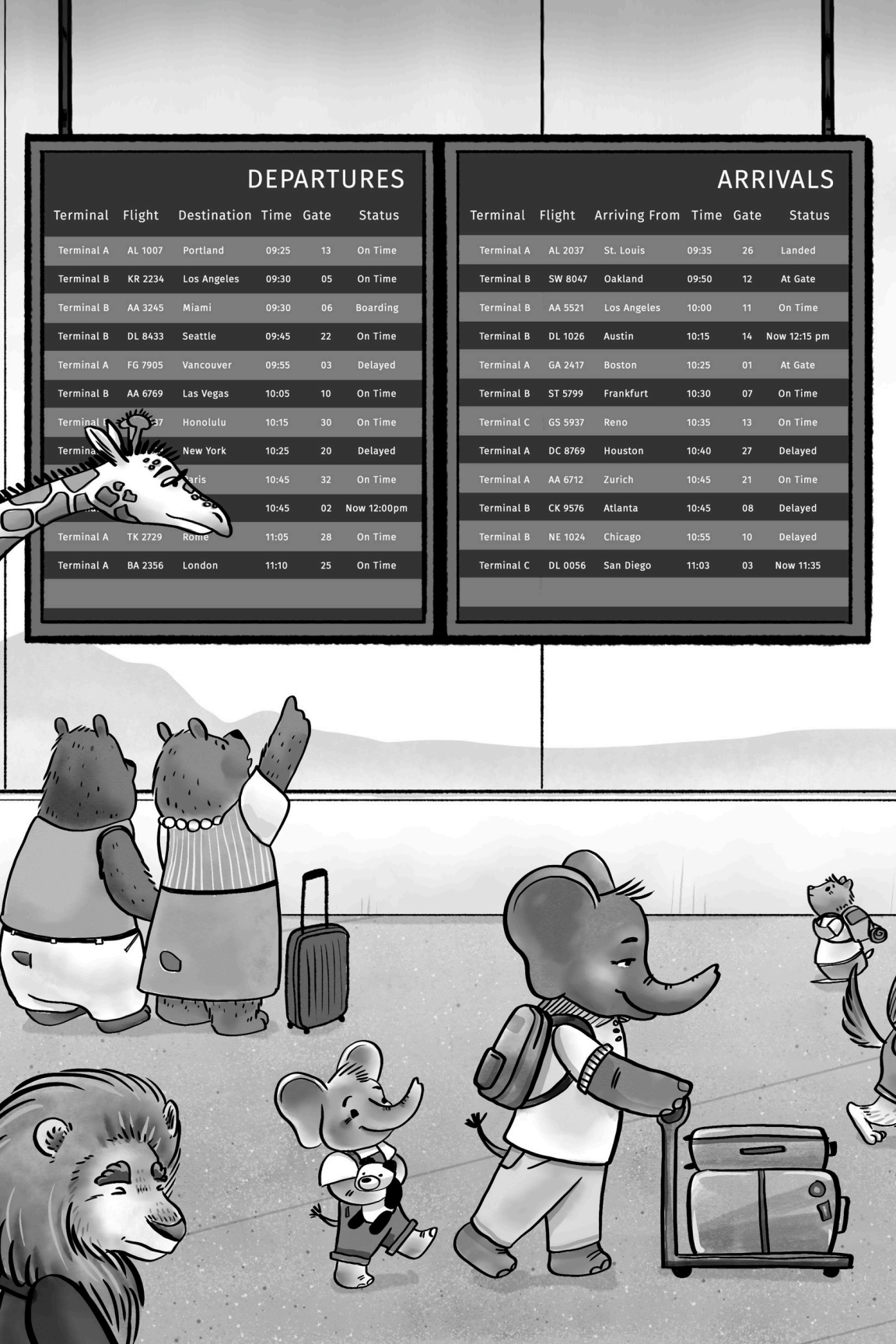
---

13. If you're interested in learning more, check out Unicode Technical Report #10: https://unicode.org/reports/tr10/

14. For a complete description of the bidi algorithm, check out Unicode Standard Annex #9.

Recap

- Unicode is the standard way for computers to encode, represent, and process text written in most scripts.
- Unicode encodes characters rather than glyphs; it's concerned about the meaning and behavior of characters, not their appearance.
- The character model for Unicode can be separated into four parts: abstract characters, coded characters, character encoding forms, and character encoding schemes.
- Coded characters assign a name and code point to an abstract character.
- Unicode defines three character encoding forms: UTF-8, UTF-16, and UTF-32, each of which specifies how code points are represented.
- Character encoding schemes specify the byte order of encoded forms; specifically, whether UTF-16 or UTF-32 send code units starting with the least significant byte (little-endian, LE) or the most significant byte (big-endian, BE).
- In addition to a name and code point, each Unicode character has several properties, including information about case, classification, and directionality.
- Character properties are used by Unicode algorithms like normalization, case mapping, collation, bidirectionality and others.
- Unicode's standard algorithms and character database are used by Swift and other programming languages to provide a standard way to interpret and transform text.


DEPARTURES
Terminal Flight Destination Time Gate Status
Terminal A AL 1007 Portland 09:25 13 On Time
Terminal B KR 2234 Los Angeles 09:30 05 On Time
Terminal B AA 3245 Miami 09:30 06 Boarding
Terminal B DL 8433 Seattle 09:45 22 On Time
Terminal A FG 7905 Vancouver 09:55 03 Delayed
Terminal B AA 6769 Las Vegas 10:05 10 On Time
Terminal B 37 Honolulu 10:15 30 On Time
Termina New York 10:25 20 Delayed
ris 10:45 32 On Time
10:45 02 Now 12:00pm
Terminal A TK 2729 Rome 11:05 28 On Time
Terminal A BA 2356 London 11:10 25 On Time
ARRIVALS
Terminal Flight Arriving From Time Gate Status
Terminal A AL 2037 St. Louis 09:35 26 Landed
Terminal B SW 8047 Oakland 09:50 12 At Gate
Terminal B AA 5521 Los Angeles 10:00 11 On Time
Terminal B DL 1026 Austin 10:15 14 Now 12:15 pm
Terminal A GA 2417 Boston 10:25 01 At Gate
Terminal B ST 5799 Frankfurt 10:30 07 On Time
Terminal C GS 5937 Reno 10:35 13 On Time
Terminal A DC 8769 Houston 10:40 27 Delayed
Terminal A AA 6712 Zurich 10:45 21 On Time
Terminal B CK 9576 Atlanta 10:45 08 Delayed
Terminal B NE 1024 Chicago 10:55 10 Delayed
Terminal C DL 0056 San Diego 11:03 03 Now 11:35

Chapter 2:

Working with Strings in Swift

Federal Aviation Regulations require air carriers to give passenger safety briefings before take-off.[1] As annoying as they may be to frequent fliers, these announcements play an important role in keeping the skies safe for passengers and flight personnel alike. The results of this focus on safety speak for themselves: in the past two decades, the rate of commercial aviation fatalities in the United States has decreased by 95%.[2]

By contrast, the rapid globalization of communication on the Internet over the same time period can be correlated with a sharp increase in the incidence and severity of software bugs related to text handling.

In February 2018, it was discovered that an iMessage containing a particular combination of characters in Telugu script caused iOS devices to crash.[3]

In May 2018, just a few months later, a message went viral that solicited the user to tap the dot in the sequence "<⚫> 👈". However, the message *also* contained many invisible directional control characters that caused iOS to crash when attempting to map touch input to the character position in the text.

These two bugs alone affected over a billion devices worldwide. According to the National Vulnerability Database (NVD) from the U.S. Department of Commerce's National Institute of Standards

---

1. *United States Code, Title 49 from the U.S. Government Publishing Office.*
2. As measured by fatalities per 100 million passengers. Source: *"Fact Sheet – Out Front on Airline Safety: Two Decades of Continuous Evolution" accessed October, 2018.*
3. CVE-2018-4124

and Technology (NIST), there have been at least dozens of other vulnerabilities with Medium, High, or Critical severity related to Unicode.[4]

Safety should be just as important to software developers as it is to pilots and flight attendants. And proper text handling is among the most important steps we can take to make our software more robust for an increasingly global marketplace.

Unicode isn't a silver bullet for all of the problems faced by our industry. There's no magic wand emoji for us to wave over our code. Instead, understanding how Unicode works is the first step to designing more correct, more resilient software.

The second step, then, is to choose suitable technologies based on our understanding. In this respect, Swift is an excellent choice.

The Swift standard library boasts one of the most capable `String` types that you'll find in a programming language. Its design balances correctness with ergonomics and its implementation manages reasonable performance for most operations.

The first chapter provided an overview of how the Unicode Standard encodes, represents, and processes text. With that, we have the necessary framework for understanding how Swift strings work and how that functionality relates to Unicode specifications.

So, without further ado, let's dive in and see how to work with `String` effectively, applying the concepts we've learned so far:

In Swift, a `String` is a collection of `Character` values. Each `Character` represents a *grapheme cluster*, or a single, user-perceived element consisting of one or more `Unicode.Scalar` values.

4. Using NVD CVSS v3 metrics.

Creating Strings

You can create a string using a *string literal*, or a representation of a string value in source code enclosed by double quotes (").

```
let string = "Welcome aboard!"
```

A string literal may contain one or more *escape sequences*, which begin with a backslash (\) and represent characters that otherwise can't be expressed verbatim. This includes whitespace and newlines, double quotes, and, consequently, the escaping backslash character itself.

Escape Sequence	Meaning
\t	Horizontal Tab
\n	Line Feed
\r	Carriage Return
\"	Double Quote
\\	Backslash

Alternatively, you can use *multi-line string literals*, which are enclosed by three double quotes ("""), to express text containing double quotes and newlines verbatim — without the need for escape sequences.

```
let multilineString = """
Thank you for choosing Air Swift.
We wish you all a pleasant flight.
"""
```

Note: Multi-line string literals are especially useful for embedding HTML and JSON into Swift source code.

Unicode Scalar Escape Sequences

The Unicode Standard encodes hundreds of characters and sequences that may be difficult or problematic to express literally in source code, including nonprinting characters like U+200D ZERO WIDTH JOINER and ambiguous characters like U+2013 EN DASH. In Swift, a string literal may include a Unicode character using the `\u{n}` escape sequence, where n is a 1–8 digit hexadecimal number corresponding to the scalar value.

```
// U+3030 WAVY DASH
"\u{3030}" // "〰"
```

String Interpolation

However, the most capable escape sequence of them all is `\()`, which evaluates the expression enclosed in the parentheses and replaces the sequence with a `String` describing the result. This process is called *string interpolation*.

```
"7 + 4 + 7 = \(7 + 4 + 7)"
// "7 + 4 + 7 = 18"
```

Raw String Literals

Escape sequences allow you to express strings that you wouldn't otherwise be able to using literals. The irony, however, is that the one thing escape sequences can't express are themselves.

In most cases, this doesn't present a significant problem: the inability to express escape sequences directly in string literals is a

small price to pay for what escape sequences allow us to do. But it does hamper efforts to embed Swift code *in* Swift code — something particularly useful when developing tooling around the language.[5]

To solve this, Swift 5 introduces syntax for *raw string literals*, which set custom boundary delimiters with one or more pound signs (#). For example, a raw string literal beginning with "pound-quote" (#") treats all content literally until the terminating "quote-point" ("#). In order for an escape sequence to be interpreted, it would have to include the same number of pound signs used by the opening and closing delimiters.

```
#"7 + 4 + 7 = \(7 + 4 + 7)"#
// "7 + 4 + 7 = \\(7 + 4 + 7)"

#"7 + 4 + 7 = \#(7 + 4 + 7)"#
// "7 + 4 + 7 = 18"
```

You can also create raw multi-line string literals using the same syntax.

```
let asciiArt = #"""
 \-----------------------\
  \                       \
   \       ___             \
    \     (_  /'_ /_/        \         __
     \    /  ((///)/          \        | \
      >       _/               >------|  \        ______
     /      __                 /       --- \_____/**|_|_\____  |
    /      (  _ /      /       /          \_______ --------- __>-}
   /      __)( /)()()(       /              /  \_____|_____/   |
  /                         /               *          |
 /-----------------------/                            {o}
"""#
```

5. Raw string literals are also helpful for regular expressions, which make extensive use of their own escape sequences. We'll talk more about regexes in Chapter 6.

Concatenating Strings

You can *concatenate*, or join the contents of strings using the addition operator, +.

```
let name = "Swift"
"Hello, " + name // "Hello, Swift"
```

String variables (declared with `var` instead of `let`) can also use mutating methods like `append(_:)` or the addition assignment operator (+=).

```
var welcome = "Welcome"
welcome.append(" back") // "Welcome back"
welcome += "!" // "Welcome back!"
```

Alternatively, you can combine strings by string interpolation.

```
let name = "Swift"
"Hello, \(name)" // "Hello, Swift"
```

Accessing Characters in a String

You can iterate the contents of a string using a `for-in` loop:

```
let string = "Hi!"
for character in string {
    print(character)
}
// Prints:
// H
// i
// !
```

To get the length of a string, use the `count` property.

```
"Hi!".count // 3
```

Like any collection, each element in a string has an associated *index*. To create its indices, a string must determine its character boundaries by inspecting each scalar value individually from start to finish.[6]

`String.Index` is an opaque type that stores the physical offset for each logical member of the collection.[7] This is arguably the most important thing to know about working with `String` values, and it bears repeating: **Swift strings aren't indexed by integer values.**

So common is this misconception that attempting to access strings by integer subscript generates a specific compiler warning:

```
"Hello"[0]
// error: 'subscript(_:)' is unavailable:
// cannot subscript String with an Int,
// see the documentation comment for discussion
```

Problem is, we have no idea what documentation comment this warning is referring to. We couldn't find anything in Apple's docs, and the header docs just seem to be a Markdown version of the website.

So why doesn't Swift allow integer subscripting for strings?
To prevent developers from writing quadratic, `O(n²)` code.

---

6. Guidelines for determining the boundaries of user-perceived characters are provided by Unicode Standard Annex #29.

7. The standard print output for a string index isn't particularly helpful, so we use the encoded `Offset` property in the example to get equivalent, encoded integer values.

For collections like `Array` and `Dictionary`, subscript access is a constant-time, `O(1)` operation. But if you were to ask for the n^{th} character in a string, that would be a linear-time, `O(n)` operation because it requires the string to process each scalar value up to that position in order to determine character boundaries. Do that in a loop, and you've made what should be a quick, linear operation into a slow, quadratic operation.

```
// 😱 Accidentally quadratic
for i in 0..<string.count {
    let character = string[i]
}
```

For accessing individual indices outside of iteration, Swift strings have `startIndex` and `endIndex` properties that correspond to the positions of the first and last character. You can seek to other indices from those positions using the methods `index(after:)` and `index(offsetBy:)`.

```
let string = "Hello"
string[string.startIndex] // "H"
string[string.index(after: string.startIndex)] // "e"
string[string.index(string.startIndex, offsetBy: 4)] // "o"
string[string.index(string.endIndex, offsetBy: -1)] // "l"
```

Another approach is to use the `firstIndex(of:)` or `lastIndex(of:)` method to get the index of the first or last occurrence of a particular character, if present.

```
string.firstIndex(of: "l")?.encodedOffset // 2
string.lastIndex(of: "l")?.encodedOffset // 3
string.firstIndex(of: "!") // nil
```

Note: Behind the scenes, a `Character` is actually backed by a `String` value with API guarantees for its length to always be 1 — that is, representing a single grapheme cluster.

You can construct a *range* from two indices using the *closed range operator* (`...`) and *half-open range operator* (`..<`). A closed range

includes both the left- and right-hand side values, whereas a half-open range includes only the left-hand side value. Both operators require that the left-hand side value is less than the right-hand side value.

The full range of a string can be expressed using a half-open range from its `startIndex` to its `endIndex`. Subscripting by range returns a `Substring` of the original `String`.

```
let range = string.startIndex..<string.endIndex
string[range] // "Hello"
```

These binary operators have prefix and postfix variants that allow you to refer to ranges in a breezy, offhand manner. *"Why yes, everything up to here is fine, thanks."*

```
let index = string.index(after: string.startIndex)
string[...index] // "He"
string[..<index] // "H"
string[index...] // "ello"
```

Or if you want to be ***really*** casual about it, go ahead and omit both ends of a closed range operator to have the subscript return the entire string as a `Substring`.

```
string[...] // "Hello"
```

Note: A `Substring` shares a buffer with its originating `String` value. Both types have a common API provided by the `StringProtocol` type, which lets you to use them interchangeably in many circumstances.

Comparing Strings

In Swift, two strings are considered equal if they're canonically equivalent.

For example, the “é” in the string “exposé” can be either precomposed into a single character or decomposed into a letter and a combining mark. In Swift, both forms produce strings comprising the same characters.

```
// U+00E9 LATIN SMALL LETTER E WITH ACUTE (é)
let precomposed = "expos\u{00E9}"

// U+0301 COMBINING ACUTE ACCENT (´)
let decomposed = "expose\u{0301}"

precomposed == decomposed // true
precomposed.count == decomposed.count // true
```

If you want to distinguish between canonically equivalent strings, you can use one of the provided Unicode encoding view properties to compare their underlying representations.

```
precomposed.utf8.elementsEqual(decomposed.utf8) // false
```

Accessing Unicode Encoding Forms

Swift strings expose Unicode encoding forms through instance properties: `utf8` for UTF-8 encoding, `utf16` for UTF-16 encoding, and `unicodeScalars` for UTF-32 encoding — so named because 32-bits are sufficient for each scalar value in the 21-bit Unicode code space.

You can use these properties individually or together to get a complete understanding of the contents for any string.

```
let string = "東京 🇯🇵"

for character in string {
  print("\(character)")

  for unicodeScalar in character.unicodeScalars {
    print(unicodeScalar)
  }

  for utf16CodeUnit in "\(character)".utf16 {
      print(utf16CodeUnit)
  }

  for utf8CodeUnit in "\(character)".utf8 {
      print(utf8CodeUnit)
  }
}
```

Character	東		京			🇯🇵			
Code Point	U+6771		U+4EAC	U+0020		U+1F1EF		U+1F1F5	
UTF-16	6771		4EAC	0020		D83C	DDEF	D83C	DDF5
UTF-8	E6	9D B1	E4 BA AC	20		F0 9F 87 AF		F0 9F 87 B5	

Accessing Unicode Character Properties

Swift 5 adds a `properties` property to `Unicode.Scalar` values that provides a wealth of information about code points from the Unicode Character Database such as name, general category, and case mapping. Exposing this information through built-in APIs allows you to work with text comprehensively without duplicating information from the Unicode Character Database.

For example, the `isEmoji` property offers a more convenient, correct, and future-proof way to tell if a character is emoji than checking for membership in a code point range.

```
func filterNonEmoji<T>(from string: T) -> String
    where T: StringProtocol
{
  return String(string.filter { character in
    character.unicodeScalars.allSatisfy { unicodeScalar in
      unicodeScalar.properties.isEmoji
    }
  })
}

filterNonEmoji(from: "EMOJI 👏 MAKE 👏 STRINGS 👏 BETTER 👏")
// "👏👏👏👏"
```

Many of these properties are also exposed to `Character` values — in this case, directly rather than through an intermediate `properties` member.

For example, each entry in the Unicode Character Database has a field for whether that character represents a number (and if so, what that value is). Beyond the ten Hindu-Arabic digits we're all familiar with, there are Arabic-Indic numerals, Chinese characters, enclosed digits, and fractions.[8] Rather than hard-coding these relationships yourself in a lookup table, you can use the `isNumber`, `isWholeNumber`, and `wholeNumberValue` properties provided by the `Character` type.

```
// U+0031 DIGIT ONE
("1" as Character).wholeNumberValue // 1

// U+0661 ARABIC-INDIC DIGIT ONE
("١" as Character).wholeNumberValue // 1

// U+4E00 CJK UNIFIED IDEOGRAPH-4E00
("一" as Character).wholeNumberValue // 1

// U+2460 CIRCLED DIGIT ONE
("①" as Character).wholeNumberValue // 1

// U+00BD VULGAR FRACTION ONE HALF
("½" as Character).isNumber // true
("½" as Character).isWholeNumber // false
("½" as Character).wholeNumberValue // nil
```

The Unicode properties added in Swift 5 offer a glimpse of the future direction for the language and standard library.

Currently, string APIs in Swift are divided equally between the standard library and the Foundation framework. Swift strings offer a solid, albeit limited core of functionality. Everything else, from locale-sensitive operations and data encoding to transformations and regular expressions, are delegated to APIs inherited from `NSString`. Over time, more and more of Foundation will be pulled into the Swift standard library. But for now, most Swift developers will be reliant on Foundation for essential string functionality.

---

8. Fun fact: there are three characters whose numeric values overflow 32-bit integers:
U+16B60 PAHAWH HMONG NUMBER TEN BILLIONS (equal to 10^{10}),
U+16B61 PAHAWH HMONG NUMBER TRILLIONS (equal to 10^{12}), and
U+5146 CJK UNIFIED IDEOGRAPH-5146 (equal to 10^{12}).

We'll discuss all the ways that Foundation augments the Swift `String` type in Chapter 4.

But before we do that, let's first take a closer look at how string functionality is organized within the Swift standard library itself — from general-purpose, functional protocols like `Sequence` and `Collection` to those specific to text, like `StringProtocol` and `TextOutputStreamable`. That's up next, in Chapter 3.

Recap

- `String` is a `Collection` of `Character` elements.
- `Character` represents a single user-perceived grapheme cluster consisting of one or more `Unicode.Scalar` values.
- `Unicode.Scalar` represents a Unicode scalar value.
- You can create strings using string literals, which come in single-line, multi-line, and (in Swift 5) raw varieties.
- String literals may contain interpolated expressions that are evaluated and inserted in the resulting string value.
- Determining the character boundaries in a string is an operation with linear-time complexity (`O(n)`) that requires enumeration of each code unit from start to finish.
- String indices are opaque structures that cannot be directly initialized by integer value.
- Two `String` values are considered equal if they are canonically equivalent.
- You can access the encoding forms of a `String` with the `utf8`, `utf16` and `unicodeScalars` properties.
- Most `String` functionality is defined by inherited protocols like `Sequence`, `Collection`, and `StringProtocol`.
- Locale-dependent String operations like collation, searching, and transformation aren't available in the standard library, but are instead provided by the Foundation framework.

Chapter 3:

Swift String Protocols and Supporting Types

A string is a collection of characters. This is true in most programming languages at a high level, however Swift takes this idea quite literally: `String` conforms to the `Collection` protocol and its elements are `Character` objects.

On one hand, this kind of conceptual purity is laudable — impressive, even — and provides a consistent, functional interface across the standard library. On the other hand, this approach creates some friction between the inherited semantics of collections and the unique semantics of text.

For example, when you add an element to a `Collection`, you expect the resulting collection to contain that element. And yet, adding a combining character like U+0301 COMBINING ACUTE ACCENT (◌́) instead changes the value of the last character.

```
var string = "expose"
let newElement: Character = "\u{0301}"
string.append(newElement) // exposé
string.contains(newElement) // false (!)
```

Another difference between `String` and other `Collection` types is that two strings may be considered equal despite having different underlying contents:

```
// é LATIN SMALL LETTER E WITH ACUTE
let precomposed = "expos\u{00E9}"

// e LATIN SMALL LETTER E +
// ◌́ COMBINING ACUTE ACCENT
let decomposed = "expose\u{0301}"

precomposed == decomposed // true
```

Conversely, there are several operations that make sense for collections but not for strings. For example, `String` is a `Collection` whose `Element` conforms to `Comparable` and therefore inherits the `sorted()` method that produces an array of characters with a particular ordering.

```
"exposé".sorted() // ["e", "o", "p", "s", "x", "é"] (?)
```

It's hard to imagine when this method would be useful, so its inclusion in the API is potentially confusing for developers.

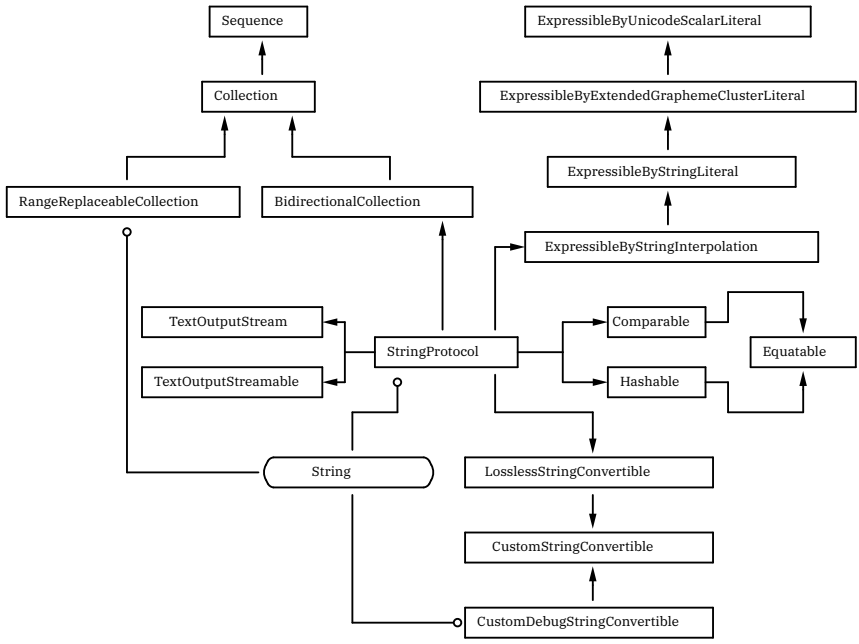
> **Note:** Swift has struggled with this tension throughout its development. Swift 2 changed `String` to remove conformance to `Collection`, instead exposing a collection of characters through a `characters` property (similar to the Unicode views provided by `utf8`, `utf16`, and `unicodeScalars`). However, this change was later reverted in Swift 3.

Another downside to the protocol-oriented design of the Swift standard library is that it's often difficult to determine where a particular method or property is defined and how it's expected to behave. Everything just kind of gets thrown together into a mish-mash of APIs.

In this chapter, we'll untangle `String` from each of its adopted and inherited protocols. By understanding the protocol-oriented nature of `String`, you can leverage its functionality more effectively.

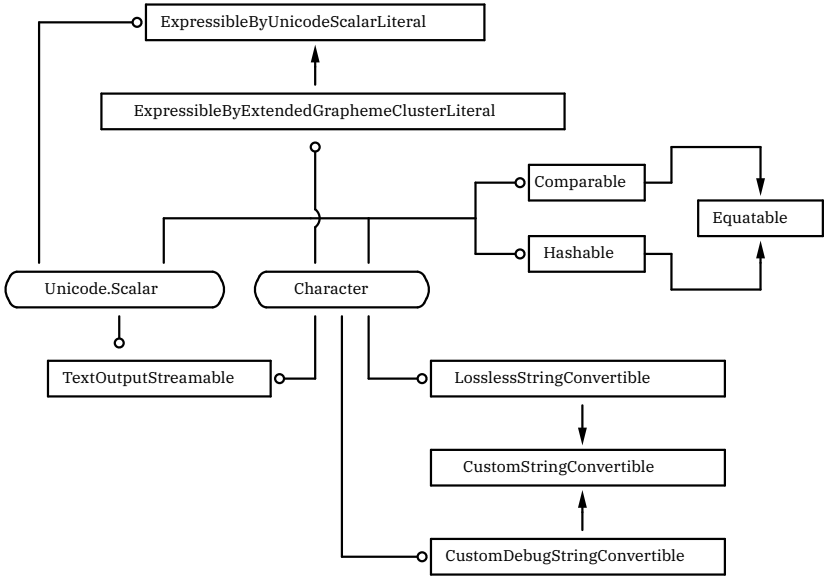
`String` is one of the most connected types in the standard library, inheriting dozens of methods from over a dozen protocols in total. To help us make sense of all of this, it's helpful to visualize `String` in relation to its adopted and inherited protocols:[1]

1. For clarity, some protocols are omitted from this diagram, including `CustomPlaygroundQuickLookable`, `CustomReflectable`, `MirrorPath`, and `Codable`.



`String` directly conforms to the `CustomDebugStringConvertible`, `RangeReplaceableCollection`, and `StringProtocol` protocols and inherits an additional 14 — all of them, incidentally, from `StringProtocol`.

In addition to `String`, the `Character` and `Unicode.Scalar` play important roles in text processing. In many ways, the equivalent diagram for these types is much the same as for `String` if you were to remove `StringProtocol`:



Each protocol contributes a particular set of methods and properties that come together to form `String` as we know it. Let's look at each individually in terms of the behaviors they provide.

Sequence

A *sequence* is a list of values that provides sequential, iterated access to elements one at a time.

Iterating the Characters in a String

One of the defining characteristics of a Swift sequence is that its elements can be iterated in a `for-in` loop. Because `String` inherits the `Sequence` protocol, you can iterate a Swift string character by character.

```
let string = "Hello, again!"
for character in string {
    print(character)
}
```

Creating Arrays of Characters

Collections like `String` and `Array` have initializers from sequences, which allow you to go back and forth like this:

```
let string = "Hello!"
Array(string) // ["H", "e", "l", "l", "o", "!"]
String(Array(string)) // "Hello!"
```

Functional Programming

`Sequence` provides essential functional programming methods like `map(_:)`, `compactMap(_:)`, `filter(_:)`, and `reduce(_:_:)`, which allow you to apply the same functional paradigm to strings that you would for any other data processing task.

```
"Boeing 737-800".filter { $0.isCased }
                .map { $0.uppercased() }
// ["B", "O", "E", "I", "N", "G"]
```

Splitting

`split(separator:maxSplits:omittingEmptySubsequences:)` divides a sequence into an array of subsequences around a given separator element. In the case of a `String`, the subsequence is a `Substring` and the separator is a `Character`.

A common use for the `split` method is to tokenize comma-delimited values in a string.

```
let seats = "7A,7B,10F,13C"
seats.split(separator: ",")
// ["7A", "7B", "10F", "13C"]
```

Note: Don't use `split` to tokenize natural language text into words, sentences, or paragraphs. Instead, use the `NLTokenizer` class described in Chapter 7.

Finding Characters

`Sequence` provides the `contains(where:)` and `first(where:)` methods, which you can use to determine whether a string contains a particular character.

```
let isVowel: (Character) -> Bool = {
    "aeiouy".contains($0)
}

"pilot".first(where: isVowel) // "i"
```

Finding Prefix Substrings

Use the `prefix(_:)` method to get a substring of a given length from the start of a string, or the `prefix(while:)` method to get everything from the beginning until a particular condition isn't met.

```
"airport".prefix(3) // "air"
"airport".prefix(while: { $0 != "r"}) // "ai"
```

Tip: You can use Swift's *trailing closure* syntax when passing a closure as the final argument to a function. In the case of functions like `filter(_:)` and `map(_:)` the meaning of this closure is clear, so this alternative syntax is preferred.

```
"Boeing 737-800".filter { $0.isCased }
                .map { $0.uppercased() }
```

However, when the semantics of the passed closure isn't as clear, such as in the case `prefix(while:)`, forego the fancier syntax in favor of keeping the final parameter label.

```
string.prefix { $0 != " "} // 🤷‍♀️
string.prefix(while: { $0 != " "}) // 🤦‍♀️
```

Excluding Characters

The `dropFirst(_:)` and `dropLast(_:)` methods offer another approach to accessing substrings by excluding a given number of characters from the start or end of a string.

```
let string = "ticket counter"
string.dropFirst(7) // "counter"
string.dropLast(7) // "ticket"
```

Along the same lines, the `drop(while:)` method produces a substring starting from the first character that doesn't satisfy a given condition — essentially the inverse of `prefix(while:)`.

```
string.prefix(while: { $0 != " "}) // "ticket"
string.drop(while: { $0 != " "}) // " counter"
```

Warning: Avoid using the following `Sequence` methods on `String` values:

- `shuffled(using:)` randomly orders the elements in a `Sequence`, which is rarely useful in the context of a `String`.
- Similarly, `sorted()` produces a sorted array of elements, which is unlikely to be helpful for text processing.
- Likewise, `min()` and `max()` are unlikely to produce meaningful results.
- `lexicographicallyPrecedes(_:)`, despite its implied dictionary ordering, doesn't support the Unicode Collation Algorithm. If you want to sort strings, use the localized string comparison methods described in Chapter 4 instead.
- `underestimatedCount` is equivalent to `count` for `Collection` types (including `String`), and serves primarily as a way for types conforming to `Sequence` to reserve storage capacity more efficiently.

Collection

A *collection* is an indexed sequence that can be traversed multiple times from any index.

Accessing Indices

The extent of a collection is signified by its `startIndex` and `endIndex` properties. Collections also have an `indices` property that returns a `DefaultIndices` collection of each index in order.

To iterate over each character along with its associated index, create a zip sequence with the `zip(_:_:)` function, passing the the string and string indices.

```
for (index, character) in zip(string.indices, string) {
    print(index.encodedOffset, character)
}
// Prints:
// 0 H
// 1 i
// 2 !
```

Warning: Swift sequences have an `enumerated()` method, which returns an enumerated sequence that produces tuples containing collection elements with integer offsets. Developers frequently mistake these offsets to be collection indices *(understandably so, as arrays are frequently iterated in `for-in` loops and their indices are equivalent to these integer offsets)*.

```
// ⚠️ offset is an `Int`, not a `String.Index`
for (offset, character) in string.enumerated() {
    // ...
}

To prevent potential confusion,
avoid calling the `enumerated()` method on `String` values.
```

Determining the Length of a String

A `String` is a `Collection` of `Character` elements. And as for any other `Collection` type, you use the `count` property to get the number of `Character` elements a `String` contains.

```
let string = "Hello!"
string.count // 6
```

A common mistake for developers is to use `count == 0` to determine if a string is empty, which iterates through the entire sequence of characters. It's much faster to use the `isEmpty` property instead.

BidirectionalCollection

A *bidirectional collection* is a collection that can be traversed backward as well as forward from any index.

Reversing a String

Although `Sequence` provides an implementation of `reversed(_:)` that creates an array populated with each element in reverse order, the `BidirectionalCollection` override lets you reverse a collection without allocating additional memory.

```
func isPalindrome(_ string: String) -> Bool {
    let normalized = string.filter { $0.isLetter }
                           .lowercased()
    return normalized.elementsEqual(normalized.reversed())
}

isPalindrome("I'm a fool; aloof am I.") // true
```

Finding Characters (Starting from the End)

From `BidirectionalCollection`, `String` gets a complement to `first(where:)`: `last(where:)`. You can use it to determine whether a string contains a particular character, returning the last occurrence instead.

```
"pilot".last(where: isVowel) // "o"
```

Traversing backwards from the end lets you perform operations like `lastIndex(of:)` and `lastIndex(where:)` without iterating through the entire sequence first.

Finding Suffix Substrings

Use the `suffix(_:)` method to get a substring of a given length from the end of a string.

```
"airport".suffix(4) // "port"
```

RangeReplaceableCollection

A *range-replaceable collection* is a collection that can add, remove, and replace elements.

Adding Characters

Use `append(_:)`, `insert(_:at:)` to add characters to a string variable.

```
var string = "plane"
string.append("s") // "planes"

let index = string.index(string.endIndex, offsetBy: -1)
string.insert("t", at: index) // "planets"
```

Tip: If you're building up a string value and know what the resulting length will be ahead of time, you can call the `reserveCapacity(_:)` method after initializing your string to reduce the number of additional memory allocations. However, as for any optimization, it's important to benchmark before and after to understand the impact of such a change.

Removing Characters

Use the `removeFirst(_:)`, `removeLast(_:)`, `remove(at:)`, and `removeSubrange(_:)` methods to remove characters from the start, middle, and end of a string variable. Or use the `removeAll(keepingCapacity:)` method to remove them all.

```
var string = "international concourse"
string.removeFirst(5) // national concourse
string.removeLast(6) // national con

let index = string.lastIndex(of: " ")!
string.remove(at: index) // nationalcon

let range = string.firstIndex(of: "o")!..<string.lastIndex(of:
"o")!
string.removeSubrange(range) // "nation"
```

Replacing Characters

Use the `replaceSubrange(_:with:)` method to replace a range of characters those from a different string.

```
var string = "lost baggage"
let index = string.firstIndex(of: " ")!
string.replaceSubrange(..<index, with: "found")
// "found baggage"
```

StringProtocol

`StringProtocol` provides a common set of APIs to `String` and `Substring` including initialization from C strings, non-localized case transformation, and access to Unicode views.

Converting with C Strings

In C, a string is represented by an array of code units terminated by a zero value (`NUL`). Swift strings can be created from a pointer to a byte array using the `String(cString:)`, `String(validating UTF8:)`, and `String(decodingCString:as:)` initializers.

```
var ascii: [UInt8] = [0x43, 0x00]
let string = String(cString: &ascii) // "C"
```

You can also go the reverse direction, and obtain a pointer to a C string using the `withCString(_:)` and `withCString(encoded As:)` method.

```
import func Darwin.strlen

string.withCString { pointer in
    strlen(pointer)
} // 1
```

Use these methods when interacting with C APIs that consume or produce strings.

Transforming Case

You can use the `uppercased()` and `lowercased()` methods to obtain the UPPERCASE and lowercase forms of strings.[2]

```
"hello".uppercased() // "HELLO"
"HELLO".lowercased() // "hello"
```

2. Characters are case mapped in a context-independent manner according to the guidelines provided by Unicode Standard Annex #21.

Checking Prefix and Suffix

`StringProtocol` adds complementing `hasPrefix(_:)` and `hasSuffix(_:)` methods to the sequence methods `prefix(_:)` and `suffix(_:)`.

```
let prefix = "airport".prefix(3) // "air"
"airplane".hasPrefix(prefix) // true

let suffix = "airport".suffix(4) // "port"
"airplane".hasSuffix(suffix) // false
```

Extending the Functionality of Strings

The most important aspect of `StringProtocol` is its role as the extension point for Swift strings. If you want to add a method or property to `String`, define it in an extension to `StringProtocol` instead to expose the same functionality to `Substring`.

```
extension StringProtocol {
    var isPalindrome: Bool {
        let normalized = self.lowercased()
                             .filter { $0.isLetter }

        return normalized.elementsEqual(normalized.reversed())
    }
}

let string = "I'm a fool; aloof am I."
string.isPalindrome // true

let range = string.firstIndex(of: "a")!...string.lastIndex(of:
"a")!
string[range].isPalindrome // true
```

You can also use `StringProtocol` as a constraint for generic types or extensions with conditional conformance.

```
struct Wrapper<Value> where Value: StringProtocol { }

extension Array where Element: StringProtocol { }
```

`StringProtocol` doesn't cover the entire `String` API surface area. So if you want to use a method like `removeSubrange(_:)` in your extension, you can define a generic `where` clause to pull in requirements from the `RangeReplaceableCollection` protocol.

```
extension StringProtocol where Self: RangeReplaceableCollection
{}
```

Warning: At the time of writing, Apple's documentation for `StringProtocol` lists dozens of methods that are inherited from Foundation, and aren't part of the Swift standard library.

CustomStringConvertible & CustomDebugStringConvertible

The Swift standard library provides three distinct functions for writing textual representations of values:

- `print`, which uses the `String(describing:)` initializer
- `debugPrint`, which uses the `String(reflecting:)`
- `dump`, which constructs a string representation using the `Mirror(reflecting:)` initializer

To accommodate as many built-in and user-defined types as possible, each function has an elaborate "trial-and-error" approach for finding a suitable candidate representation. However, these candidates can (and often do) overlap, and cause, for example, `print` and `debugPrint` having the same behavior for a given value.

If we were to generalize, we could associate each of these functions with a particular protocol in the following way:

Function	Protocol	Required Property
print	CustomStringConvertible	description
debugPrint	CustomDebugStringConvertible	debugDescription
dump	CustomReflectable	customMirror

Types can customize their string representation used by the `String(describing:)` initializer and `print(_:)` function by adopting the `CustomStringConvertible` protocol and implementing the required `description` property.

```
struct FlightCode {
    let airlineCode: String
    let flightNumber: Int
}

let flight = FlightCode(airlineCode: "AA",
                        flightNumber: 1)
print(flight)
// FlightCode(airlineCode: "AA", flightNumber: 1)

extension FlightCode: CustomStringConvertible {
    var description: String {
        return "\(self.airlineCode) \(self.flightNumber)"
    }
}

print(flight)
// AA 1
```

Types can also customize how their debugging representation is used by the `String(reflecting:)` initializer and `debugPrint(_:)` function by conforming to the `CustomDebugStringConvertible` and implementing the `debugDescription` property.

```
extension FlightCode: CustomDebugStringConvertible {
    var debugDescription: String {
        return """
        Airline Code: \(self.airlineCode)
        Flight Number: \(self.flightNumber)
        """
    }
}
```

In the case of `String`, conformance to both types is trivial — instances simply return themselves.

LosslessStringConvertible

`String` has more initializers than any other type in the standard library. In previous versions of Swift, the `String` initializer that is used to create textual representations of values took a single, unlabeled argument that could be difficult to distinguish from other custom initializers.

Swift 3 solved this by renaming this initializer to `String(describing:)` and introducing a replacement that took a single `LosslessStringConvertible` argument.[3]

3. SE-0089: "Renaming `String.init<T>(_: T)`"

Values conforming to `LosslessStringConvertible` can be represented as a string unambiguously without loss of information. Examples of such types in the standard library include `String`, `Character`, `Unicode.Scalar`, `Bool`, `Int`, and `Float`.

```
extension FlightCode: LosslessStringConvertible {
    public init?(_ description: String) {
        let components = description.split(separator: " ")
        guard components.count == 2,
            let airlineCode = components.first,
            let number = components.last,
            let flightNumber = Int(number)
        else {
            return nil
        }

        self.airlineCode = String(airlineCode)
        self.flightNumber = flightNumber
    }
}

let flight = FlightCode(airlineCode: "AA",
                        flightNumber: 1)

String(flight)
// "AA 1"

FlightCode(String(flight))
// FlightCode(airlineCode: "AA", flightNumber: 1)
```

Note: Parsing values from strings is covered in greater depth in Chapter 6.

TextOutputStreamable & TextOutputStream

The standard library `print(_:)` and `dump(_:)` functions write values of any type conforming to the `TextOutputStreamable` to standard output.

You can use the `print(_:to:)` and `dump(_:to:)` functions to write to any value whose type conforms to the `TextOutputStream` protocol.

For example, you could create a custom stream type to write to `stderr` instead of `stdout`.

```
import func Darwin.fputs
import var Darwin.stderr

struct StderrOutputStream: TextOutputStream {
    mutating func write(_ string: String) {
        fputs(string, stderr)
    }
}

var standardError = StderrOutputStream()
print("Error!", to: &standardError)
```

However, a more compelling use case might be found as an alternative to adopting `CustomStringConvertible` for types that have multiple valid or configurable representations.

For example, you could create a `UnicodeLogger` type conforming to `TextOutputStream` that, instead of writing a string verbatim, writes the name and code point for each of its characters.

```
struct UnicodeLogger: TextOutputStream {
  mutating func write(_ string: String) {
    guard !string.isEmpty && string != "\n" else {
        return
    }

    for (index, unicodeScalar) in
        string.unicodeScalars.lazy.enumerated()
    {
      let codePoint = String(format: "U+%04X",
unicodeScalar.value)
      let name = unicodeScalar.name ?? ""
      print("\(index): \(unicodeScalar) \(codePoint)\t\(name)")
    }
  }
}
```

A custom logger type offers a compelling alternative to a conventional top-level function that calls `print(_:)` internally.

```
print("👯")
// Prints: "👯"

var logger = UnicodeLogger()
print("👯", to: &logger)
// Prints:
// 0: 👨 U+1F468    MAN
// 1:    U+200D     ZERO WIDTH JOINER
// 2: 👩 U+1F469    WOMAN
// 3:    U+200D     ZERO WIDTH JOINER
// 4: 👧 U+1F467    GIRL
// 5:    U+200D     ZERO WIDTH JOINER
// 6: 👧 U+1F467    GIRL
```

ExpressibleByUnicodeScalarLiteral, ExpressibleByExtendedGraphemeClusterLiteral, and ExpressibleByStringLiteral

In Swift, string literals are enclosed double quotes ("). The same syntax can also express grapheme cluster literals when the enclosed values comprise a single grapheme cluster or Unicode scalar literals when the enclosed values comprise a single scalar value.

Initialization from string, grapheme cluster, and Unicode scalar literals is available to types conforming to the `ExpressibleByStringLiteral`, `ExpressibleByExtendedGraphemeClusterLiteral`, and `ExpressibleByUnicodeScalarLiteral` protocols, respectively.

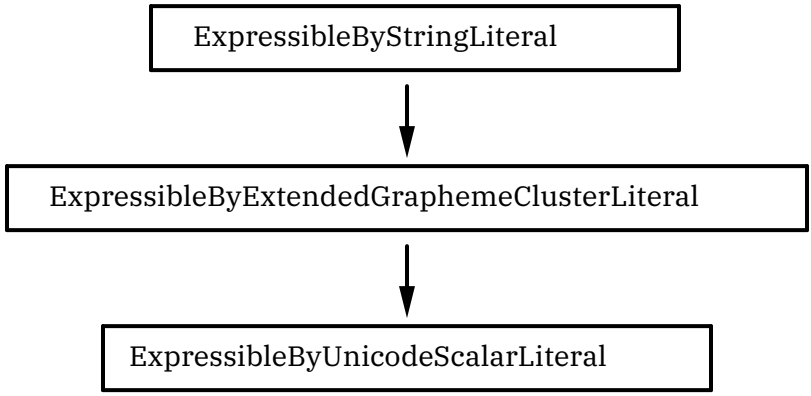
```
// ExpressibleByUnicodeScalarLiteral
let unicodeScalar: Unicode.Scalar = "A"

// ExpressibleByExtendedGraphemeClusterLiteral
let character: Character = "A"

// ExpressibleByStringLiteral
let string: String = "A"
```

By default, literals enclosed by double quotes (") are inferred to be `String` values.

```
"A" is String // true
```

A string consists of one or more grapheme clusters and a grapheme cluster consists of one or more Unicode scalar values. Consequently, any grapheme cluster literal is a valid string literal and any Unicode scalar literal is a valid grapheme cluster literal (however the converse isn't true).

For this reason, the `ExpressibleByStringLiteral` inherits `ExpressibleByExtendedGraphemeClusterLiteral`, which in turn inherits `ExpressibleByUnicodeScalarLiteral`.



Attempting to initialize a `Unicode.Scalar` with a literal containing multiple scalar values or a `Character` with a literal containing multiple characters results in an error.

```
("ABC" as Unicode.Scalar) // Error
// Cannot convert value of type 'String' to specified type
'Unicode.Scalar'

("ABC" as Character) // Error
// Cannot convert value of type 'String' to specified type
'Character'
```

Extending Custom Types to Be Expressible By String Literals

You can improve the ergonomics of custom types by adding support for initialization with string literals. For example, consider the following type representing an airline fare basis code:

```
public enum BookingCode: String {
    case firstClass = "F"
    case businessClass = "J"
    case premiumEconomy = "W"
    case economy = "Y"
}
```

Enumerations with `String` raw values can be trivially extended to support initialization from string literals.

Tip: Although each of the defined enumeration cases are expressible by Unicode scalar literals, it's generally advisable for custom types to adopt `ExpressibleByStringLiteral` even when `ExpressibleByExtendedGraphemeClusterLiteral` or `ExpressibleByUnicodeScalarLiteral` would be sufficient.

Because of protocol inheritance, `ExpressibleByStringLiteral` requires initializers for string, grapheme cluster, and Unicode scalar literals. In most cases, you can simply delegate the latter two to the string initializer.

```
extension BookingCode: ExpressibleByStringLiteral {
    public init(stringLiteral value: String) {
        self.init(rawValue: value)!
    }

    public init(extendedGraphemeClusterLiteral value: String) {
        self.init(stringLiteral: value)
    }

    public init(unicodeScalarLiteral value: String) {
        self.init(stringLiteral: value)
    }
}
```

Doing this lets you initialize `BookingClass` values directly from a string literal.

```
// Before
BookingClass(rawValue: "J")

// After
("J" as BookingClass) // Business Class
```

This can be especially convenient in the context of an array or dictionary literal.

```
let nowBoarding: Set<BookingClass> = ["F", "J"]
```

ExpressibleByStringInterpolation

One of the most anticipated new features in Swift 5 is the ability to extend string interpolation to support configurable representations of values.[4]

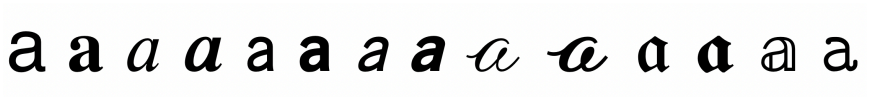
Normally, interpolated values in string literals are converted to strings using the `String(describing:)` initializer. By conforming to the `ExpressibleByStringInterpolation` protocol (which inherits from `ExpressibleByStringLiteral`), types can define a `StringInterpolation` type to change and parameterize interpolation behavior in string literals.

To understand what this feature allows for and how it works, consider the following example:

Unicode encodes several alternate forms of Latin letters for use in mathematical formulas and other presentation contexts, including bold, italic, script, and fraktur.

---

4. SE-0228: “Fix ExpressibleByStringInterpolation”



We could use `ExpressibleByStringInterpolation` to support interpolation of strings with a style parameter.

```
let name = "Johnny"
let styled: StyledString = """
    Hello, \(name, style: .fraktur(bold: true))!
"""
```

Important: Don't use mathematical alphanumeric symbols as stylistic alternatives to conventional characters. They make text inaccessible to screen readers and aren't guaranteed to render on all devices.

First, define each available style in an enumeration.[5]

```
enum Style {
    case none
    case bold
    case italic
    case boldItalic
    case sansSerif(bold: Bool, italic: Bool)
    case script(bold: Bool)
    case fraktur(bold: Bool)
    case doubleStruck
    case monospace
}
```

5. Although the majority of these variants have code points that can be computed by adding a constant offset, there are sufficiently many exceptions to justify the use of a lookup table.

Each of the variants are consolidated into a single tuple with named fields for each, which are keyed in a dictionary according to their base character.

```
typealias Variants = (
    bold: Character,
    italic: Character,
    boldItalic: Character,
    sansSerif: Character,
    sansSerifBold: Character,
    sansSerifItalic: Character,
    sansSerifBoldItalic: Character,
    script: Character,
    scriptBold: Character,
    fraktur: Character,
    frakturBold: Character,
    doubleStruck: Character,
    monospace: Character
)

let characterVariants: [Character: Variants] = ...
```

Next, define a `StyledString` structure to manage an array of styled components.

```
struct StyledString {
    private var components: [(String, Style)] = []

    init(_ value: String, style: Style = .none) {
        self.append(value, style: style)
    }

    mutating func append(_ string: String, style: Style) {
        self.components.append((string, style))
    }
}
```

In preparation for adopting `ExpressibleByStringInterpolation`, conform `StyledString` to the prerequisite `ExpressibleByStringLiteral` protocol.

```
extension StyledString: ExpressibleByStringLiteral {
    init(stringLiteral value: String) {
        self.init(value)
    }

    init(extendedGraphemeClusterLiteral value: String) {
        self.init(stringLiteral: value)
    }

    init(unicodeScalarLiteral value: String) {
        self.init(stringLiteral: value)
    }
}
```

Finally, conform `StyledString` to the `ExpressibleByStringInterpolation` protocol in an extension that implements the required initializer.

When a string literal is used to initialize `StyledString`, its split before and after each interpolation segment and appended — in order — to the nested, associated `StringInterpolation` type.

Each uninterpolated literal value is passed to the append `Literal(_:)` method. Interpolations are forwarded to the append `Interpolation` method whose signature matches that of the interpolation.

```
extension StyledString: ExpressibleByStringInterpolation {
  init(stringInterpolation: StringInterpolation) {
      self.components =
        stringInterpolation.styledString.components
  }

  struct StringInterpolation: StringInterpolationProtocol {
    private var styledString: StyledString

    init(literalCapacity: Int, interpolationCount: Int) {
      styledString = StyledString()
    }

    mutating func appendLiteral(_ literal: String) {
      styledString.append(literal, style: style)
    }

    mutating func appendInterpolation<T>(_ value: T,
                                         style: Style)
    {
        styledString.append(String(describing: value),
                            style: style)
    }
  }
}
```

After all of that, we can now interpolate our strings *in style*:

```
let name = "Johnny"
let styled: StyledString = """
Hello, \(name, style: .fraktur(bold: true))!
"""
```

Hello, Johnny

It's a heck of a lot easier than if we were to try to replicate each step of the interpolation ourselves:

```
let approximate = "Hello, \(name)!"
var interpolation =
    StyledString.StringInterpolation(
        literalCapacity: approximate.count,
        interpolationCount: 1
    )
interpolation.appendLiteral("Hello, ")
interpolation.appendInterpolation(
    name, style: .fraktur(bold: true)
)
let styled = StyledString(stringInterpolation: interpolation)
```

The new `ExpressibleByStringInterpolation` protocol in Swift 5 lends an unprecedented level of expressiveness to `String`. Look for it to enable all kinds of new functionality, both in the standard library and in the broader developer ecosystem.

Swift's `String` type has a reputation for being difficult to work with.

As we've seen in this chapter, some of the blame can be placed on how functionality is inherited from so many different protocols, like `Sequence` and `Collection`. Combined with poor treatment in Apple's documentation, it can be difficult to determine where a particular method or property is defined and how it's expected to behave.

Another factor is the seemingly magical relationship between `String` and the Foundation framework. Because `String` is bridged with `NSString`, importing Foundation automatically exposes the `String` type to the entire surface area of `NSString`. This interoperability with Foundation APIs is the subject of the next chapter.

Putting all of that aside, you might argue that this negative reputation is mostly undeserved. Any perceived difficulty to work

with `String` may stem more from mismatched expectations based on a developer's experience with string types in other languages. In many cases, "easy-to-use" APIs come at the cost of correctness, which is ultimately the wrong trade-off to make. The Swift core team articulated this nicely in their so-called "Strings Manifesto":[6]

> *[...] Ergonomics and correctness are mutually-reinforcing. An API that is easy to use — but incorrectly — cannot be considered an ergonomic success. Conversely, an API that's simply hard to use is also hard to use correctly. Achieving optimal performance without compromising ergonomics or correctness is a greater challenge.*

All things considered, Swift provides one of the most correct, usable, and performant string types available in any programming language.

6. Abrahams, Dave, and Cohen, Ben. "String Processing For Swift 4". (2017) https://github.com/apple/swift/blob/master/docs/StringManifesto.md, accessed October 2018.

Recap

- Most `String` functionality is defined by inherited protocols; however, some inherited methods have limited usefulness as String operations and shouldn't be called.
- `Sequence` provides the `map(_:)`, `prefix(_:)`, `drop(while:)`, and `split(separator:maxSplits:omittingEmptySubsequences:)` methods as well as character iteration in `for-in` loops.
- `Collection` provides the `count` and `isEmpty` properties as well as character indices and subscript access.
- `BidirectionalCollection` provides the `suffix(_:)`, `reversed(_:)`, `lastIndex(of:)`, and `lastIndex(where:)` methods.
- `RangeReplaceableCollection` provides concatenation functionality like the `append(_:)` and `insert(_:at:)` methods and the + operator as well as methods for removing, including `removeFirst(_:)`, `removeLast(_:)`, `removeSubrange(_:)`, and `removeAll(keepingCapacity:)`.
- `StringProtocol` provides a shared interface between `String` and support types like `Substring` and `StaticString` as well as conversion with C strings, access to Unicode encoded forms, locale-insensitive `lowercased()` and `uppercased()` methods, as well as the `hasPrefix(_:)` and `hasSuffix(_:)` methods.
- `CustomStringConvertible` provides the `description` property, which is used by the `String(describing:)` initializer and the `print(_:)` method as well as for interpolation in string literals.
- `CustomDebugStringConvertible` provides the `debugDescription` property, which is used by the `String(reflecting:)` initializer and the `debugPrint(_:)` function.
- `TextOutputStream` and `TextOutputStreamable` provide mutating and nonmutating forms of the `write(_:)` method, respectively.
- `ExpressibleByUnicodeScalarLiteral`, `ExpressibleByExtendedGraphemeClusterLiteral`, and `ExpressibleByStringLiteral` provide initialization by literals.
- In Swift 5, `ExpressibleByStringInterpolation` allows types to extend the functionality of string interpolation.

Chapter 4:

Working with Foundation String APIs

`NSString` is a Foundation class that represents a Unicode-compliant text string. It's the *de facto* standard string type for Objective-C and used throughout the system.

By importing the Foundation framework, `String` imports dozens of useful `NSString` APIs and unlocks interoperability with the rest of the platform SDKs.

Interoperability with Foundation and Objective-C APIs

For better interoperability between Swift and Objective-C, most Apple platform SDKs, including Foundation, AppKit, and UIKit, have refinements for Swift that change `NSString` arguments and return values to instead take and receive Swift `String` values.

For example, in Objective-C, `NSDateFormatter` has a `dateFormat` property with type `NSString`, a `-dateFromString:` method that takes an `NSString` parameter, and a `-stringFromDate:` method that returns an `NSString` value.

```
NSString *timestamp = @"1969-02-09";

NSDateFormatter *formatter = [NSDateFormatter new];
formatter.dateFormat = @"yyyy-MM-dd";
NSDate *date = [formatter dateFromString:timestamp];

formatter.dateStyle = NSDateFormatterLongStyle;
NSString *string = [formatter stringFromDate:date];

NSLog(@"%@", string); // February 9, 1969
```

In Swift, `NSDateFormatter` is imported as `DateFormatter` and provides a `dateFormat` property and `date(from:)` / `string(from:)` methods, each of which take and receive `String` values rather than `NSString` objects.

```
import Foundation

let timestamp = "1969-02-09"

let formatter = DateFormatter()
formatter.dateFormat = "yyyy-MM-dd"
let date = formatter.date(from: timestamp)!

formatter.dateStyle = .long
let string = formatter.string(from: date)

print(string) // Prints "February 9, 1969"
```

You don't typically need to reference `NSString` in Swift code. But if you do, `NSString` is bridged to Swift's `String` type, which allows you to cast from one to another using the `as` operator.

```
let string = "Hello!"
let nsstring = string as NSString
```

Note: As a performance optimization, Swift copies the contents of a bridged `NSString` value into contiguous storage only when that value is mutated. This operation typically has linear performance characteristics, corresponding to the length of the string's encoded representation.

Interoperability with Core Foundation and C APIs

`NSString` is *toll-free bridged* with its Core Foundation counterpart, `CFString`. This means that you can pass an `NSString` object to any function argument that takes a `CFString` reference. By the transitive property, `String` is therefore convertible to `CFString` (but you'll need to import Foundation in order for this to work).

For example, the first argument in the Core Text function `CTFontCreateWithName` takes a `CFString` value for the font name parameter. After importing Foundation, you can pass a Swift `String` cast as either a `CFString` or `NSString`.

```
import Foundation
import CoreText

CTFontCreateWithName("Helvetica" as CFString, 16, nil)
CTFontCreateWithName("Helvetica Neue" as NSString, 16, nil)
```

Tip: Another type that you may encounter when working with low-level C APIs is `UniChar` (a.k.a. `unichar`). These values are equivalent to UTF-16 code units.

For example, the second argument in the `CTFontGetGlyphsForCharacters` function takes an array of `UniChar` values, which can be constructed from the `utf16` view of a Swift `String`.

```
func glyphAvailable(for string: String,
                    in font: CTFont) -> Bool
{
    let unichars: [UniChar] = Array(string.utf16)
    var glyphs: [CGGlyph] = [0, 0]
    return CTFontGetGlyphsForCharacters(font,
                                        unichars,
                                        &glyphs,
                                        unichars.count)
}
```

String vs. NSString

`NSString` represents strings using UTF-16 encoding. All positions, ranges, and lengths are expressed in terms of UTF-16 code units. **This is the single most important difference between Foundation's `NSString` and the standard `String` type in Swift.**

As discussed in Chapter 1, the first publication of the Unicode standard in 1991 was restricted to a 16-bit code space. It was only in the 2.0 release in 1996 that Unicode introduced a surrogate character mechanism to expand its code space to 21 bits.

Until recently, most developers wouldn't need to concern themselves with the distinction between `NSString` code units and grapheme clusters. Text so rarely included characters outside the Basic Multilingual Plane (BMP) that you could get away with thinking that each pair of bytes was one character. And if text did wade into the Supplementary Multilingual Plane (SMP) or beyond, any resulting errors were just as often blamed on the user than seen as a programmer error.[1]

"Of course that broke. What, did you expect Egyptian hieroglyphs to work?"

But then Emoji happened. And now billions of people around the world include hieroglyphs *(of a sort)* in their everyday communication.[2] Characters outside the BMP are the norm rather than the exception.

Some text processing APIs in Foundation express index and length in terms of `NSString` UTF-16 code points by way of `NSRange` and `NSUInteger` values.

You can convert a Swift `String` range to an equivalent `NSRange` value using the `NSRange(_:in:)` initializer.

```
import Foundation

let string = "Hello, world!"
let nsRange = NSRange(string.startIndex..<string.endIndex,
                      in: string)
let range = Range(nsRange, in: string)
```

---

1. *[citation needed]*
2. Emoji went mainstream in 2011 when iOS 5 included Emoji as a standard international keyboard. Once people saw that you could add hearts and smiley faces to text messages, this previously obscure feature instantly went viral.

To get the length of a Swift `String` in terms of UTF-16 code units, you can get the `count` property of its `utf16` view.

```
let length = string.utf16.count
```

To convert indexes into `NSString`-compatible UTF-16 code unit positions, use the `samePosition(in:)` method and call the `encodedOffset` property on the result.

```
let index = string.firstIndex(of: " ")!
let position = index.samePosition(in: string.utf16)?
                    .encodedOffset
```

Warning:

Semantic differences between Strings in Foundation and the Swift standard library can lead to surprising behavior. For example, `padding(toLength:withPad:startingAt:)` pads a string according to its length in UTF-16 code units rather than extended grapheme clusters, which isn't apparent by looking at the documentation alone:

> ***`padding(toLength:withPad:startingAt:)`***
> *Returns a new string formed from the receiver by either removing characters from the end, or by appending as many occurrences as necessary of a given pad string.*

```
import Foundation

// Family With Mother and Daughter
// (U+1F469 U+200D U+1F467)
let string = "\u{1F469}\u{200D}\u{1F467}"
let length = string.count // 1
string.padding(toLength: 2,
               withPad: " ",
               startingAt: 0) // "👩" (!)
```

Exercise caution when interacting with Foundation text processing APIs that take `NSUInteger` or `NSRange` parameters to express string lengths or positions. Use comments and unit tests to communicate and verify intent — or avoid using them altogether.

Now that we have a solid understanding of the ins and outs of string interoperability between Swift and Objective-C APIs, let's dive into all of the functionality provided by Foundation.

Localized String Operations

Swift's `String` type is Unicode-compliant but locale-insensitive. Meaning that it handles text correctly in general, but can't accommodate differences in how text is handled in particular languages and regions. By importing Foundation, `String` gains the localized string operations necessary to develop apps for an international audience.[3]

But perhaps the phrase "international audience" is misleading in that it presents an "us" vs. "them" mentality. When you use the localized string operations provided by Foundation, you're writing code to accommodate everyone — and that includes yourself.

For instance, every locale has some convention for how strings should be collated.

Collation and Comparison

Most of us would expect a list of items containing numbers, like "File 3.txt", "File 7.txt", and "File 36.txt", to be ordered just like so, in ascending, numerical value. However, unlocalized string comparison is performed on a character by character basis; because "3" precedes "7" and numerals precede ".", that same list would be ordered "File 3.txt", "File 36.txt", "File 7.txt".

We can use one of the Foundation string comparison methods to collate strings in a way that follows user expectation. If we had to

3. Localized string operations use the Unicode Collation Algorithm as well as information for tailoring to different languages provided by the Common Locale Data Repository (CLDR).

choose just one of these methods, it would have to be `localizedStandardCompare(_:)`. Its documentation describes it as being able to "[Sort] strings like Finder", which is pretty hard to fault.[4]

```
import Foundation

let files: [String] = [
    "File 3.txt",
    "File 7.txt",
    "File 36.txt"
]

let order: ComparisonResult = .orderedAscending

files.sorted { lhs, rhs in
    lhs.compare(rhs) == order
} // => ["File 3.txt", "File 36.txt", "File 7.txt"]

files.sorted { lhs, rhs in
    lhs.localizedStandardCompare(rhs) == order
} // => ["File 3.txt", "File 7.txt", "File 36.txt"]
```

Other collation rules differ between different locales. For example, many English-speaking locales are insensitive to diacritics like ring (˚) and diaeresis (¨), and therefore treat A and Å as the same letter for the purposes of collation. However, in Swedish, characters like Å and Ö are distinct letters that appear at the end of the alphabet.

4. You can also produce this behavior by passing the `numeric` comparison option to the `compare(_:options:range:locale:)` method.

We can demonstrate this by using the `compare(_:options:range:locale:)` method to order a list of city names in both American and Swedish locales and comparing the results:

```
import Foundation

let cities: [String] = [
    "Albuquerque",
    "Ålesund",
    "Östersund",
    "Reno",
    "Tallahassee"
]

let 🇺🇸 = Locale(identifier: "en_US")
let 🇸🇪 = Locale(identifier: "sv_SE")

let order: ComparisonResult = .orderedAscending

for locale in [🇺🇸, 🇸🇪] {
    cities.sorted { lhs, rhs in
      lhs.compare(rhs,
                  options: [],
                  range: nil,
                  locale: locale) == order
    }
}
```

English (United States)	Swedish (Sweden)
Albuquerque	Albuquerque
Ålesund	Reno
Östersund	Tallahassee
Reno	Ålesund
Tallahassee	Östersund

Searching

"Does this string contain a given substring?" You might think this to have a straightforward answer (just compare them like you would any other collection, right?). But, alas, like so many things involving text, the correct answer is "it depends". Specifically, it depends on which locale you're asking about.

For example, diacritics are typically ignored for the purpose of comparison in English-speaking locales, so the word “Éclair” is said to contain the character “E” despite it not having the code point U+0045 LATIN CAPITAL LETTER E.

```
import Foundation

"Éclair".contains("E") // false
"Éclair".localizedStandardContains("E") // true
```

There are two flavors of string searching: those that return a Boolean value (“contains” methods) and those that return a range (“range” methods). Both flavors come in several varieties with configurable sensitivity to differences in case (a ≠ A), diacritics (a ≠ á), and width (a ≠ ａ), as well as options to anchor search to boundaries or search starting from the end of the string.

Unless you need a particular combination of these behaviors or want to specify a locale other than the current one, you should use the `localizedStandardContains(_:)` and `localizedStandardRange(of:)` methods.

Case Mapping

Although rare, some case mapping rules are language sensitive. In those locales, these distinctions can be a matter of life and death — literally.[5] So unless you have a good reason not to, make a practice of using `localizedLowercase()` and `localizedUppercase()` instead of the unlocalized `lowercased` and `uppercased` properties.

5. In 2007, a typo in a text message of the dotless “ı” in the Turkish word “sıkısınca” was cause for a violent misunderstanding that resulted in the death of two people and imprisonment of three others. (Source: http://www.hurriyet.com.tr/gundem/kucucuk-bir-nokta-tam-5-kisiyi-yakti-8748359)

For instance, Turkish and Azeri loses the dot in dotted capital i (İ) when lowercased unless it precedes a dotted character.

```
import Foundation

let turkish = Locale(identifier: "tr")
"İ".lowercased(with: turkish) // "\u{0131}"
"İ".lowercased() // "\u{0069}\u{U+0307}"
```

Contrapositively, Lithuanian retains the dot in accented lowercase i (ĩ) when uppercased.

```
import Foundation

let lithuanian = Locale(identifier: "lt")
"ĩ".uppercased(with: lithuanian) // "\u{0049}\u{0300}"
"ĩ".uppercased() // "\u{0049}\u{0307}\u{0300}"
```

Tip: A good rule of thumb for localized string operations in Foundation is to prefer the “localized standard” version of string search and comparison methods and the “localized” version of other string methods, where available. The other variations should be used only to achieve a specific behavior according to the requirements of your app.

Normalizing Strings

Normalization forms are, in some ways, similar to byte order. Normally you don’t have to be concerned about them… at least until you reach an API boundary. Even then, a well-designed, high-level API should take care of this for you.

If string normalization is, in fact, something that your app needs to be concerned about, you’ll be glad to know that Foundation provides access to the following Unicode Normalization Form properties:

Property	Unicode Normalization Form
`precomposedStringWithCanonicalMapping`	C
`decomposedStringWithCanonicalMapping`	D
`precomposedStringWithCompatibilityMapping`	KC
`decomposedStringWithCompatibilityMapping`	KD

Although the Swift standard library doesn't provide any APIs for accessing normalization forms for strings, you can still verify the results by inspecting their unicode scalars.

```
import Foundation

let string = "ümlaut"

let nfc = string.precomposedStringWithCanonicalMapping
nfc.unicodeScalars.first
// U+00FC LATIN SMALL LETTER U WITH DIAERESIS

let nfd = string.decomposedStringWithCanonicalMapping
nfd.unicodeScalars.first
// U+0075 LATIN SMALL LETTER U
```

Initializing Strings from Data

Another key component provided by Foundation is the `Data` type. Importing the framework lets you, for example, create a `String` from a text file. This can be done either directly from a URL using the `String(contentsOf: url)` initializer or by first populating a `Data` object from the contents of the file and then calling `String(data :encoding:)`.

```
import Foundation

let url = Bundle.main.url(forResource: "file", withExtension:
"txt")!
try String(contentsOf: url) // "Hello!"

let data = try Data(contentsOf: url)
String(data: data, encoding: .utf8) // "Hello!"
```

Important: Unless explicitly provided in a `String` initializer, the system attempts to infer the encoding of data. ASCII, UTF-8, and encodings using a *byte-order mark* (BOM) can be reliably detected with an algorithm, whereas other encodings rely on statistics-based heuristics to make this determination. If you know the encoding of a file ahead of time, specify it in code or risk getting back a steaming pile of *mojibake*.

Conversion Between String Encodings

Swift was created recently enough that its standard library doesn't need to handle text encodings other than UTF. Foundation, on the other hand, predates Unicode, and supports numerous legacy encodings, including Mac OS Roman, Japanese Shift JIS, and Windows code pages. You can use the `String(data:encoding:)` initializer and `data(using:)` method to read and write text into encoding formats for legacy systems as needed.

```
import Foundation

let string = "Hello, Macintosh!"
string.data(using: .macOSRoman) // 17 bytes
```

If ASCII compatibility is a concern, you may elect to use `data(using:allowLossyConversion:)` to convert text with some loss of fidelity.

```
import Foundation

let string = "Avión 🛩"
string.canBeConverted(to: .ascii) // false
string.data(using: .ascii) // nil
if let data = string.data(using: .ascii,
                          allowLossyConversion: true) {
    String(data: data, encoding: .ascii) // "Avion ??"
}
```

For a given string, some encodings may offer faster retrieval of characters whereas others may result in a smaller encoded

representation. Although the decision of which encoding to use for transport or storage isn't typically made by the client on an ad hoc basis, Foundation provides the `fastestEncoding` and `smallestEncoding` properties to help you make such determinations.

```
import Foundation

"Airplane".fastestEncoding // ASCII
"飛行機".smallestEncoding // UTF-16
```

Transforming Strings

In the previous example, the lossy encoding of the string "Avión" resulted in "Avion". A much better way to strip accents and other diacritics is to apply a string transform using the `applying Transform(_:reverse:)` method.[6]

```
import Foundation

"Avión".applyingTransform(.stripDiacritics, reverse: false)
// "Avion"
```

Several different transforms are available, including XML hexadecimal escaping ("©" -> "©"), Unicode character naming ("🛂" -> "\\N{PASSPORT CONTROL}") half-width and full-width form conversion for CJK characters ("マット" → "ﾏｯﾄ"), and a variety of transliterations between different scripts.

Transliteration is the process of transforming text to another equivalent or derived representation. One of the most useful ones available in Foundation is `toLatin`, which converts most scripts into its equivalent representation in Latin.

6. Foundation's string transformation functionality is provided by International Components for Unicode (ICU), an open-source library for software internationalization.

For example, you can use the `toLatin` transform to transliterate Korean text into something more easily readable as an English speaker.[7]

```
import Foundation

"""
안전 벨트를 휘게하십시오
""".applyingTransform(.toLatin, reverse: false)
// "anjeon belteuleul hwigehasibsio"
```

Tip: You can use Latin transliteration to normalize full-text indexes and queries to get a rough approximation of multilingual search functionality in your app.

Formatting Values

Foundation has a wealth of APIs for presenting strings and representations of data in a format that's suitable for the end user. Between formatters like `NumberFormatter` that provide comprehensive internationalization support and the localization functionality offered by `NSLocalizedString` and `Bundle`, the Foundation framework gives you everything you need to make software for a global audience.

The topics of internationalization and localization fall beyond the scope of what we can cover here, but it's important to preface our discussion of Foundation's string format initializers with the following guidance:

7. Though in this case, it may be just as easy to learn how to do it yourself. Hangul (한글), the official writing system of the Korean language, is widely celebrated by linguists and linguaphiles for its unique aesthetic, phonographic consistency, simplicity, and efficiency. It's also one the easiest script to learn — anyone can learn how to read and write it in a matter of minutes. *Give it a try!*

Important: Use `NSLocalizedString` and `NumberFormatter` instead of `String` initializers when creating strings to be presented to the user.

`String(format:locale:arguments:)` takes a format string, optional locale, and a variable list of arguments to be formatted. The format string uses the same syntax as `printf(3)`,[8] which replaces tokens beginning with a percent sign (%) according to their *format specifiers*.

For instance, here are the format specifiers for text and some examples of how they're used:

Format Specifier	Description
`c`	UTF-8 code unit
`C`	UTF-16 code unit
`@`	String
`%`	Percent Sign[9]

```
String(format: "%c", 0x41)      // A
String(format: "%C", 0x3042)    // "あ"
String(format: "%@", "Hello")   // "Hello"
String(format: "%d%%", 100)     // "100%"
```

For strings and characters, Swift's string interpolation offers a safer and more conventional way to produce formatted output. The only real advantage that old-school format specifiers have over string interpolation is more fine-grained control over how numbers are formatted.[10]

8. With slight differences — most notably the `@` specifier for strings.

9. Locales have different conventions for formatting percentages, with variation in whether the percent sign comes before or after the number and whether a space separates them. Use `NumberFormatter` with `percent` style to ensure the correct presentation for your users.

10. And even then, `ExpressibleByStringInterpolation` in Swift 5 will likely remove any need for `printf`-style format strings in the future.

Here's a list of the format specifiers related to numbers and a few examples of how they're used:

Format Specifier	Description
`d`	signed decimal integer
`u`	unsigned decimal integer
`x`	unsigned hexadecimal integer, lowercase
`X`	unsigned hexadecimal integer, uppercase
`o`	unsigned octal integer
`f`	floating-point decimal, lowercase
`F`	floating-point decimal, uppercase
`e`	scientific notation, lowercase
`E`	scientific notation, uppercase
`g`	floating-point or scientific notation, lowercase
`G`	floating-point or scientific notation, uppercase
`a`	scientific notation hexadecimal, lowercase
`A`	scientific notation hexadecimal, uppercase

```
String(format: "%d", 123)       // "123"
String(format: "%X", 127)       // "7F"
String(format: "%f", 1234.56)   // "1234.560000"
```

Length Modifiers

A format specifier may also include a *length modifier*, which changes the range of the formatted output.

These modifiers follow integer specifiers (d, o, u, x, X):

Modifier	Description
`hh`	signed or unsigned char (8 bits in size)
`h`	short signed or unsigned integer (at least 16 bits in size)
`l`	long signed or unsigned integer (at least 32 bits in size)
`ll`, `q`	long long signed or unsigned integer (at least 64 bits in size)

```
String(format: "%hhu", Int(UInt8.max))      // "255"
String(format: "%u", Int(UInt8.max) + 1)    // "256"
String(format: "%hhu", Int(UInt8.max) + 1)  // "0"
```

The only modifier for floating-point specifiers (`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`) is `L`, which extends the precision of floating-point representations:

Modifier	Description
`L`	extended precision double (at least 80 bits in size)

```
String(format: "%LF", Float80(6.02214086e23))
// "602214086000000000000000.000000"
```

Formatting Flags

In addition to the length modifiers, format specifiers may contain the following *formatting flags*, each of which alter the resulting output in their own particular ways.

Zero Padding (0)

For integer and floating-point formats, the `0` flag pads the output with leading zeroes to the specified width.

```
String(format: "%04d", 1)       // "0001"
String(format: "%02X", 0x0F)    // "0F"
```

Left Justification (-)

The `-` flag left-justifies the output with spaces to the specified width.

```
String(format: "%-5d", 10)  // "10   "
```

Signedness (+)

The + flag adds an explicit sign ("+" or "-") to the output — even when the number is positive.

```
String(format: "%+d", 1)    // "+1"
String(format: "%+d", -1)   // "-1"
String(format: "%d", 1)     // "1"
```

Alternative Forms (#)

The # flag causes the output to produce alternative forms, which vary across different format specifiers.

For octal integer specifiers (o), the alternative form has a leading zero before the first digit.

```
String(format: "%#o", 0o744)    // "0744"
String(format: "%o", 0o744)     // "744"
```

For hexadecimal integer specifiers (x and X), the alternative form prepends "0x" or "0X".

```
String(format: "%#X", 0xF7)     // "0XF7"
String(format: "%X", 0xF7)      // "F7"
```

For conditional floating-point specifiers (g and G), the alternative form doesn't remove trailing zeroes or radix point as it would normally.

```
String(format: "%#g", 123.0)    // "123.000"
String(format: "%g", 123.0)     // "123"
```

Locale Arguments

You can also pass a `Locale` argument to the second parameter of the `String(format:locale:arguments:)` initializer to produce output suitable for the specified locale.

For example, some countries like the United States use a comma as the thousands separator and a period as the radix / decimal point, whereas other countries, including France, use a comma instead for the radix / decimal point, and space for the thousands separator.

```
let 🇺🇸 = Locale(identifier: "en_US")
let 🇫🇷 = Locale(identifier: "fr_FR")

String(format: "%f", locale: 🇺🇸, 12345.67)
// "12,345.670000"

String(format: "%f", locale: 🇫🇷, 12345.67)
// "12 345,670000"
```

Another example of formatting differences across locales can be seen by comparing the conventions for scientific notation in the United States, Sweden, and Egypt.

```
let 🇸🇪 = Locale(identifier: "sv_SE")
let 🇪🇬 = Locale(identifier: "ar_EG")

String(format: "%e", locale: 🇺🇸, 12345.67)
// "1.234567E+04"

String(format: "%e", locale: 🇸🇪, 12345.67)
// "1,234567×10^+04"

String(format: "%e", locale: 🇪🇬, 12345.67)
// "+٠٤ س١٠,٢٣٤٥٦٧"
```

But again — despite what the `locale` parameter may indicate — you shouldn't rely on this method to provide localized output. If you're formatting numbers that will be seen by users, you'll find `NumberFormatter` to be a far more capable way to do that.[11]

```
import Foundation

let formatter = NumberFormatter()
let styles: [NumberFormatter.Style] = [
    .decimal, .scientific, .currency
]

for style in styles {
    formatter.numberStyle = style
    print(formatter.string(for: 12345.67)!)
}
// Prints:
// 12,345.67
// 1.234567E4
// $12,345.67
```

As we've seen in this chapter, Foundation provides a full complement of functionality to the Swift `String` type. Between normalization, conversion, transformation, and localized string operations, the APIs imported from `NSString` are must-haves for any app that works with text in a meaningful way.

Of course, there's much more to Foundation beyond `NSString`, and we'll take a look at some of those other APIs in the remaining chapters. In Chapter 5, we'll learn about the Base64 encoding functionality inherited from `NSString`. In Chapter 6, we'll show how to use `NSRegularExpression` and `Scanner` to parse information from text. Finally, Chapter 7 is a deep-dive into the Natural Language framework, which reorganizes and expands on functionality found in Foundation's `NSLinguisticTagger`.

11. For a comprehensive look at `NumberFormatter`, please refer to the Flight School Guide to Swift Numbers.

Recap

- `NSString` represents strings using UTF-16 encoding; all positions, ranges, and lengths are expressed in terms of UTF-16 code units.
- Importing the Foundation framework exposes `NSString` APIs to the `String` type; these methods, properties, and initializers offer locale-sensitive string operations like comparing strings, searching for substrings, and mapping case that aren't provided by the Swift standard library.
- Get the NFD and NFC forms of a string using the `decomposedStringWithCanonicalMapping` and `precomposedStringWithCanonicalMapping` properties.
- Use the `String(contentsOf: url)` to create a string from the contents of a file.
- Use the `data(using:)` and `canBeConverted(to:)` methods to convert between different string encodings.
- Use the `applyingTransform(reverse:)` method to transliterate and perform other ICU transforms on strings.
- Use the `trimmingCharacters(in:)` method to remove leading and trailing whitespace or other characters from a string.
- Use `NSLocalizedString` and `NumberFormatter` to create localized strings and representations of data; for machine-readable output, you can use the `String(format:locale:arguments:)` initializer.


09:00 LONDON
09:00 SYDNEY
09:30 MIAMI
09:45 HONOLUL

Chapter 5:
Binary-to-Text Encoding

Border crossings are inherently precarious. Whether international or regional, digital or analog; transitioning from one context to another necessarily entails some degree of risk that your passage will be obstructed or you'll lose something along the way.

Flyers traveling through a strange airport-of-call may elect to wrap luggage in a few layers of cellophane to mitigate the chance of their suitcase going on a diet prior to arrival at its final destination.

An analogous practice in computing can be found in programs that pass information across API boundaries, namely: converting binary data to text. Like a protective outer layer of shrink wrap, this *"ASCII armor"* guards against uncooperative software as well as confusion about byte order during transport.[1]

In this chapter, we'll explore a variety of different binary-to-text encoding schemes, with varying degrees of efficiency (and as you'll see, usefulness).

But the thread that ties them all together is how they relate the concept of string encoding described so far in this book with the more general concept of information encoding.

To kick things off, let's look at perhaps the simplest way to express binary data as text: with binary (Base2) numerals.

1. Use of the phrase "ASCII armor" was codified by RFC 4880, the specification for PGP.

Base2

The most direct binary-to-text encoding method is available through the `String(_:radix:uppercase:)` initializer, which produces a representation of fixed-width binary integers with the specified *radix*, or base. Specifying a base of 2 gives us a binary representation.

```
let byte: UInt8 = 0xF0

let encodedString = String(byte, radix: 2) // "11110000"
```

Going the other direction looks much the same, with the `init(_:radix)` initializer for `FixedWidthInteger` (which produces an `Optional`, because a string may not express an integer value).

```
let decodedByte = UInt8(encodedString, radix: 2)! // 240 (0xF0)
```

However, with a compression ratio at a measly 1 byte ∶ 8 characters (12.5%), Base2 isn't a tenable way to transmit information.

Let's see what other encoding forms offer us, keeping in mind the the age-old adage:

> *With great compression comes great complexity.*
>
> — Claude Shannon, probably

Base16

Base16, or *hexadecimal* is a popular format for data because it conveniently expresses the 8 bits in a byte with two characters (0–F).

To get the hexadecimal representation of a number, use the same `String(_:radix:uppercase:)` initializer from before, but pass 16 for the `radix` parameter. When omitted, the last parameter (`uppercase`) is set to `false`, which is clearly a bad default. As a person with good taste and high moral standing, you'll surely remember to pass `true` whenever you invoke this, *won't you?*

```
let byte: UInt8 = 0xF0
let encodedString = String(byte, radix: 16, uppercase: true) //
"F0"
let decodedByte = UInt8(encodedString, radix: 16)! // 240 (0xF0)
```

You can pass this into a the `map(_:)` method on a `Data` object to dump the contents of a buffer for debugging purposes.

```
let data = Data(bytes: [0xC0, 0xFF, 0xEE])
data.map { String($0, radix: 16, uppercase: true) } // C0 FF EE
```

2 characters to 1 byte translates to 50% efficiency, which is a big improvement over straight 1's and 0's. But we can do better — certainly in terms of efficiency, but also in terms of formality. The interchangeability of lowercase and uppercase letters is not only leaving bits on the table, so to speak, but a potential source of ambiguity.

Rather than leave that hanging out in the open, why not take full advantage of the range of ASCII characters available to us?

Base64

The general strategy for binary-to-text representations of arbitrary data is to choose characters that are both printable and common to most encodings. Nearly all computer systems today support the Latin alphabet (a-z) and Hindu-Arabic numerals (0-9), which makes Base64 a great choice for an encoding. Altogether, 26 uppercase

letters, 26 lowercase letters and 10 numerals gets us to 62 — just shy of the nearest power of two: 64. Standard Base64 encoding uses + and / to round out the character set.

Here's what the standard encoding table for Base64 looks like:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	A	B	C	D	E	F	G	H
1	I	J	K	L	M	N	O	P
2	Q	R	S	T	U	V	W	X
3	Y	Z	a	b	c	d	e	f
4	g	h	i	j	k	l	m	n
5	o	p	q	r	s	t	u	v
6	w	x	y	z	0	1	2	3
7	4	5	6	7	8	9	+	/

Foundation provides methods for Base64 encoding strings in Swift. You get the Base64 encoding of a data object by calling the `base64EncodedString(options:)` method, and go the opposite direction using the `Data(base64Encoded:options:)` initializer.

Here's how you complete a round trip — encoding the UTF-8 data for a string in Base64 and decoding back to the original string:

```
import Foundation

let string = "Hello!"

let data = string.data(using: .utf8)!
let encodedString = data.base64EncodedString()
// "SGVsbG8h"

let decodedData = Data(base64Encoded: encodedString)!
let decodedString = String(data: decodedData, encoding: .utf8)
// "Hello!"
```

At 4 encoded characters for every 3 bytes, Base64 encoding achieves 75% efficiency. Its efficient compression rate and relatively simple implementation has made Base64 the most ubiquitous, general-purpose, binary-to-text encoding for computers in the Internet age.

So what are some practical applications?

Attaching Files in Email

Email is the *king* of problematic API boundaries.

For starters, most email sent today *still* uses ASCII encoding.

Seriously.

As a workaround, email messages can set the `Content-Type` header field to specify a character set and the `Content-Transfer-Encoding` header field to `quoted-printable`. For UTF-8, each non-ASCII character is expressed by a sequence of hexadecimal code units, each prepending an equals sign (=).

For example, here's the raw source of an email message saying "Hello" in Japanese (こんにちは):

```
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: quoted-printable

=E3=81=93=E3=82=93=E3=81=AB=E3=81=A1=E3=81=AF
```

Another limitation with email is that the SMTP protocol, which defines how messages are sent, limits how many characters can appear in a single line.[2]

Seriously.

But perhaps the most egregious thing about email are its Mbox file formats.[3] Mbox delimits messages with lines beginning with the string `"From "`, so if a user wants to start a new paragraph with the word "From", the mail program must first change "From" to ">From" to avoid being misinterpreted as the start of a new message.

---

2. Per RFC 2821 4.5.3.1 Size limits and minimums: "The maximum total length of a text line including the <CRLF> is 1000 characters"

3. Because, of course, Mbox isn't just one format, but a family of mutually-incompatible variants.

Seriously.

Email has enough of a problem sending plain text, let alone binary data. So Base64, in addition to the aforementioned `quoted-printable`, is one of the standard encodings suitable for messages sent via SMTP, as defined by the Multipurpose Internet Mail Extensions (MIME) Internet standard.

Here's an example of an email message with a Base64-encoded text file attachment:

```
From: from@swift.org
To: to@swift.org
Subject: MIME Email Example
MIME-Version: 1.0
Content-Type: multipart/alternative;boundary = "+++"
Message-Id: <20181210000000.123456789@iMac.local>
Date: Wed, 5 Sep 2018 16:10:00 -0500 (EST)

--+++
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: base64

SGVsbG8sIHdvcmxkIQ==
```

Encoding Assets for Web Development

A common optimization for speeding up websites is to reduce the number of requests made to the server. One approach is to embed assets like fonts or images directly in HTML or CSS files using Base64 encoding.

```
<img src="…" />
```

```
@font-face {
  font-family: "CustomTypeface";
  src: url(data:font/woff2;base64,…) format("woff2");
  font-weight: normal;
  font-style: normal;
}
```

As a general observation, a surprisingly large portion of iOS developers are skittish about web views — especially if they're served locally. Maybe it's something to do with the confusing relationship between the Asset Catalog and relative paths in web resources.

If that sounds at all familiar, you can use Base64 to be more confident about static content you present in web views. Inline your CSS and JavaScript by using `data:` URIs with Base64-encoded assets instead of paths to files in your app bundle.

Embedding Resources in Code

If you're averse to using the Asset Catalog or the Resources folder in Playgrounds, or — for whatever reason — just *need* to have everything in a single file, you can embed assets into code directly using Base64 encoding.

For example, you might store cached data encoded as a binary property list in a Base64-encoded multi-line string:

```
import Foundation

let base64bplist = """
YnBsaXN0MDDRAQJRYRABCAsNAAAAAAAAAQEAAAAAAAAAAwAAAAAAAAAAAAAAAAAA
AA8=
"""

let data = Data(base64Encoded: base64bplist)!

data.base64EncodedString(options: .lineLength64Characters)

let decoder = PropertyListDecoder()
try decoder.decode([String: Int].self, from: data) // ["a": 1]
```

Base64 is a solved problem. Most languages provide some implementation in their standard library — Swift included. There's really no reason for implementing it yourself.

That said, the underlying mechanisms are really interesting and worth exploring in one way or another. What if we reimagine Base64 for a post-apocalyptic future in which emoji is the sole mode of communication? Well, it might look a little something like this…

Base 👱

Base👱 is a novel binary-to-text encoding that represents 3 bytes with 4 people emoji characters distinguished by their individual appearances.

Unicode 8.0 adds five skin tone modifier characters based on the Fitzpatrick scale, a recognized standard in dermatology. These could be combined with human emoji to form diverse color images. For example, the combination of U+1F9D2 CHILD (🧒) followed by U+1F3FD EMOJI MODIFIER FITZPATRICK TYPE-4 can be rendered on supported platforms as a child with medium skin tone.

Unicode 11.0 extends this combinatoric approach to diversity by introducing hair components that can be used in ZWJ sequences to indicate hair colors or styles, including red-haired, white-haired, curly-haired, and bald.

Enumerating over each combination of gender, skin tone, and hairstyle, we get a total of 60 distinct human adult characters. To round that number out to 64, we can use the gender-neutral U+1F9D1 ADULT (🧓) character and four of the skin tone modifiers. Altogether, these form the base of our encoding form.

```
let base: [Character] = [
    "🧓", "🧓🏻", "🧓🏼", "🧓🏽", "🧓🏾",
    "👨🦱", "👨🏻🦱", "👨🏼🦱", "👨🏽🦱", "👨🏾🦱",
    "👨🦲", "👨🏻🦲", "👨🏼🦲", "👨🏽🦲", "👨🏾🦲",
    "👨🦰", "👨🏻🦰", "👨🏼🦰", "👨🏽🦰", "👨🏾🦰",
    "👨🦳", "👨🏻🦳", "👨🏼🦳", "👨🏽🦳", "👨🏾🦳",
    "👱", "👱🏻", "👱🏼", "👱🏽", "👱🏾",
    "👩🦱", "👩🏻🦱", "👩🏼🦱", "👩🏽🦱", "👩🏾🦱",
    "👱‍♀️", "👱🏻‍♀️", "👱🏼‍♀️", "👱🏽‍♀️", "👱🏾‍♀️",
    "👩🦲", "👩🏻🦲", "👩🏼🦲", "👩🏽🦲", "👩🏾🦲",
    "👩🦰", "👩🏻🦰", "👩🏼🦰", "👩🏽🦰", "👩🏾🦰",
    "👩🦳", "👩🏻🦳", "👩🏼🦳", "👩🏽🦳", "👩🏾🦳",
    "👲", "👲🏻", "👲🏼", "👲🏽", "👲🏾"
]
```

Our colorful cast of characters effectively replaces the alphanumeric characters found in traditional Base64 encoding. We can reuse the same approach here to implement our own encoding and decoding functionality.

As discussed before, Base64 encodes 3 bytes of data with 4 characters. As an example, let's use the UTF-8 representation of the string `"Fly"`:

Source	**Text**	**F**	**l**		**y**
	Octets	0x46	0x6C		0x79
	Bits	0 1 0 0 0 1 1 0	0 1 1 0 1 1 0 0		0 1 1 1 1 0 0 1
	Sextets	17	38	49	57
Base64 encoded	**Character**	Man: Medium Skin Tone, Curly Hair	Woman: Light Skin Tone, Bald	Woman: Medium Skin Tone, White Hair	Woman: Dark Skin Tone, Curly Hair

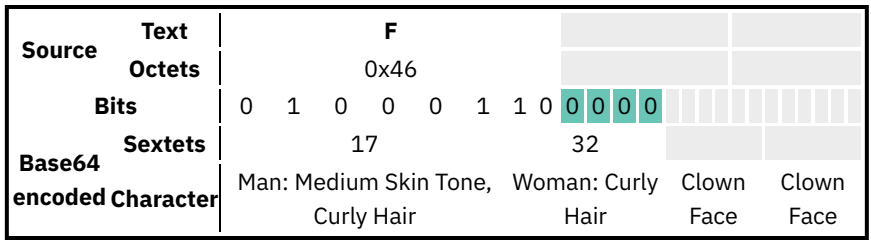
In our implementation, let's enumerate over our data in groups of 3 bytes. For each group, we initialize an integer by bit-shifting and adding each of the octets. We can then apply mask to that value, bit-shifting the other direction for each of the sextets.

```
extension Data {
    func base👴EncodedString() -> String {
        var output: String = ""

        let mask = 0b00111111

        for index in stride(from: startIndex,
                            to: endIndex - (endIndex % 3),
                            by: 3)
        {
            let value = Int(self[index]) << 16 +
                        Int(self[index + 1]) << 8 +
                        Int(self[index + 2])

            output.append(base[((value >> 18) & mask)])
            output.append(base[((value >> 12) & mask)])
            output.append(base[((value >> 6) & mask)])
            output.append(base[(value & mask)])
        }
        // ...
```

Of course, not all payloads come in sizes divisible by three. For example, if we pop off the "y" from our original string, the resulting `"Fl"` completely fills two sextets, partially fills the third sextet, and leaves the final sextet empty.

Source	**Text**	**F**								**l**													
	Octets	0x46								0x6C													
	Bits	0	1	0	0	0	1	1	0	0	1	1	0	1	1	0	0	0	0				
	Sextets	17								38								48					
Base64 encoded	**Character**	Man: Medium Skin Tone, Curly Hair								Woman: Light Skin Tone, Bald								Woman: Medium Skin Tone, Bald					Clown Face

Similarly, if we pop another character and leave just `"F"`, we're left with a filled sextet, a partially-filled sextet, and two empty sextets at the end.



We can set the least significant bits of the partially-filled sextet to 0 — no sweat. But for empty sextets, we'll need to add a padding character to differentiate missing bits from zeroed-out bits.

By convention, Base64 uses = as the padding character. In our case, we'll use something equally conspicuous: U+1F921 CLOWN FACE (🤡).

```
extension Data {
    func base👴EncodedString() -> String {
        // ...

        let padding: Character = "🤡"

        switch endIndex % 3 {
        case 2:
            let value = Int(self[endIndex - 2]) << 16 +
                        Int(self[endIndex - 1]) << 8

            output.append(base[((value >> 18) & mask)])
            output.append(base[((value >> 12) & mask)])
            output.append(base[((value >> 6) & mask)])
            output.append(padding)
        case 1:
            let value = Int(self[endIndex - 1]) << 16

            output.append(base[((value >> 18) & mask)])
            output.append(base[((value >> 12) & mask)])
            output.append(padding)
            output.append(padding)
        default:
            break
        }

        return output
    }
}
```

With a complete implementation, let's test out our new method:

```
let data = "Fly".data(using: .utf8)!

let encodedString = data.base👴EncodedString()
// Man: Medium Skin Tone, Curly Hair
// Woman: Light Skin Tone, Bald
// Woman: Medium Skin Tone, White Hair
// Woman: Dark Skin Tone, Curly Hair
```

Looking good!

Like Base64, we're representing 3 bytes of data with 4 characters.

```
encodedString.count // 4
```

However, not all characters are encoded equal. How many bytes does it take to encode this string in UTF-8?

```
encodedString.data(using: .utf8)!.count // 60 (!)
```

Yikes. 60 bytes to encode 3 bytes. That's, what, 5% efficiency? We're better off using Base2 and writing out 1's and 0's!

In fairness, byte count isn't the only metric for representing binary with text. Sometimes it's more important to be memorable than concise. So let's round out our discussion with a look at something more human, and a little less... *heady.*

Human Readable

The Internet Engineering Task Force (IETF) is a standards body for Internet technologies. Their Request for Comment memoranda (RFCs) are responsible for the alphabet soup of technologies we rely on every day, including TCP, DNS, and HTTP.

The focus of our discussion here is RFC 1751, "A Convention for Human-Readable 128-bit Keys".

For context, this document was published back in 1994 a time when, *"The Internet community [was beginning] to address matters of security."*[4] Around this time, a one-time password system called S/KEY™ devised a technique for mapping the 64-bit keys it produced to a memorable sequence of six short English words (each between one and four characters) from a 2,048-word dictionary.

For example, the following 64-bit key:

```
B203 E28F A525 BE47
```

...could be represented by the following six words:

```
LONG IVY JULY AJAR BOND LEE
```

You could also go the opposite direction, decoding the original 64-bit key from the six words.

Humans are better equipped to memorize words than random hexadecimal numbers. We can tell ourselves a story about the long ivy in the month of July covering the Lee residence and creeping into the joint of the cellar door, left ajar in the summer heat.

It's unlikely that you'll interact with S/KEY, much less write code to interface with it. But we can learn a lot by implementing this for ourselves in Swift.

Let's start by snagging that word list.

4. How different things were back then...

The RFC conveniently provides the 2048-word dictionary as a C array literal, which we can copy-paste verbatim into Swift code.

```
let words: [String] = [
    "A", "ABE", "ACE", "ACT", "AD", "ADA", "ADD",
    …,
    "YEAR", "YELL", "YOGA", "YOKE"
]
```

Each word in this list is equivalent to a single character in our Base64 encoding table. As the name implies, Base64 uses 64 characters; 64 is equal to 2^6, or 6 bits. By comparison, this word list has 2048, or 2^{11}, elements.

11 bits is a lot harder to work with than 6. Whereas our previous implementation took advantage of the convenient 3 : 4 ratio between code units and bytes, no such factor exists for 11.

Although we could white-knuckle the whole mess of manual bit-shifting needed to make this work (as seen in the RFC's reference implementation), let's take a step back and consider how we can take advantage of Swift language and standard library features to create something more understandable.

Fundamentally, the task at hand is to take a byte stream and divide it up into chunks of 11 bits. Swift makes it easy to enumerate by chunks of 8 (bytes), but not individual bits.

What if we implemented this ourselves?

Iterating Bit by Bit

In Swift, a *sequence* is a list of values. You can step through each *element* in a sequence using a `for-in` loop. Inside each sequence is an *iterator*, which is responsible for maintaining the state of the sequence and providing the next element, if available.

Tip: To satisfy the `Sequence` protocol, a type must implement the `makeIterator()` method by returning an instance of an iterator type. If that sequence can act as its own iterator, you can skip the additional step and have it conform to `IteratorProtocol` directly (as we do here).

`Data` is a `Sequence` whose `Element` is a byte (`UInt8`), which is why you can iterate each byte of a data object in a `for-in` loop.

To iterate over each *bit* of a `Data` object, we can implement a custom type that conforms to `Sequence`. In the initializer, the specified data object is used to make an iterator, which will be used to access bytes one at a time as needed; that iterator is then used immediately to set the initial state of the sequence.

```
struct BitSequence: Sequence, IteratorProtocol {
    private var base: Data.Iterator
    private var current: Data.Iterator.Element? = nil
    private var index: Int = 0

    init(_ data: Data) {
        self.base = data.makeIterator()
        self.current = self.base.next()
    }

    // ...
}
```

The `next()` method is responsible for returning the next bit in the sequence (1 or 0), or `nil` if we've run out of bytes in the base `Data` iterator. The `index` value cycles from 0 to 7 and determines which bit is returned next.

```
mutating func next() -> Int? {
    guard let current = self.current else {
        return nil
    }

    defer {
        self.index += 1

        if self.index == current.bitWidth {
            self.current = self.base.next()
            self.index = 0
        }
    }

    return current[bit: self.index]
}
```

Making Things a Bit More Convenient

You may have noticed that the last line of code from the previous example is aspirational — that is, not provided by the standard library. We need a way to access an individual bit by position, so let's make that easier for ourselves by creating an extension on `FixedWidthInteger` and defining a subscript method.

```
extension FixedWidthInteger {
    subscript(bit position: Int) -> Int? {
        guard position >= 0 && position < self.bitWidth else {
            return nil
        }

        return Int(self >> (self.bitWidth - position - 1) & 1)
    }
}
```

For our purposes, we'll index bits starting from the left-most, most significant bit at position 0. Asking for an index outside the valid range returns `nil`.

Now that we we can produce a sequence of bits, let's create a new sequence to batch those bits into groups of a particular size.

Iterating Multi-bit by Multi-bit

Just as before, `MultiBitSequence` is a custom sequence type whose initializer takes a `Data` object; the initializer also takes the number of bits included with each value.

```
struct MultiBitSequence: Sequence, IteratorProtocol {
    let numberOfBits: Int
    private var base: BitSequence.Iterator

    init(_ data: Data, numberOfBits: Int) {
        precondition(numberOfBits > 0 &&
                     numberOfBits <= Int.bitWidth)
        self.base = BitSequence(data).makeIterator()
        self.numberOfBits = numberOfBits
    }
```

The `next()` method is responsible for accumulating a given number of bits and reducing them into a single value. If, at any point, the method's loop runs ouf of bits before getting as many as needed, the method returns early with `nil`.

```
mutating func next() -> Int? {
    var value: Int = 0

    for _ in 0..<self.numberOfBits {
        guard let bit = self.base.next() else {
            return nil
        }

        value = (value << 1) + bit
    }

    return value
}
```

At this point, we're two abstractions deep, so you may have forgotten why we're building all of this to begin with. Recall that the human readable encoding described by RFC 1751 operates on 11 bit

chunks. Because of how cumbersome it is to divide multiples of 8 by 11, we've created an abstraction to take care of that for us: `MultiBitSequence`. That type is built on top of another abstraction, `BitSequence`, which takes in a `Data` object and yields each bit (whereas the standard `Data` iterator operates on bytes).

But enough about that. Let's get to the interesting bits.

Providing a Human Readable Interface

Declare a `humanReadableEncodedString()` method (to match the naming of the Foundation Base64 APIs, why not) in an extension to the `Data` type.

The actual implementation of this method is rather anticlimactic, as the toughest part was actually getting chunks of 11 bits in the first place. Each 11 bits maps to a number between 0 and 2047, which we use to find the corresponding word.

```
extension Data {
    func humanReadableEncodedString() -> String {
        let sequence = MultiBitSequence(self, numberOfBits: 11)
        return sequence.map { words[$0] }.joined(separator: " ")
    }
}
```

Let's give this a try with the example data provided in the RFC itself:

```
let data = Data(bytes: [0xB2, 0x03,
                        0xE2, 0x8F,
                        0xA5, 0x25,
                        0xBE, 0x47])

data.humanReadableEncodedString()
// LONG IVY JULY AJAR BOND LED
```

Looks good! ...wait, "*LED*"? What happened to the Lee family and their lovely ivy-covered home?

Skims through RFC

2 bits for parity, huh? Well I guess we need *something* to fill out to 66 bits...

What does the standard say about calculating those parity bits? Nothing much. All we really have to go on is the reference C implementation:

```
/* Encode 8 bytes in 'c' as a string of English words.
 * Returns a pointer to a static buffer
 */
char *
btoe(engout,c)
char *c, *engout;
{
    char cp[9];     /* add in room for the parity 2 bits*/
    /* compute parity */
    for(p = 0,i = 0; i < 64;i += 2)
        p += extract(cp,i,2);
    cp[8] = (char)p << 6;
```

By the looks of it, parity is computed by summing each pair of bits and then left-shifting the result by 6 places. Easy enough. Let's whip out `MultiBitSequence` one last time to bring this example home:

```
extension Data {
    func humanReadableEncodedString() -> String {
        var parity: Element {
            let sum = MultiBitSequence(self, numberOfBits:
2).reduce(0, +)
            return Element(truncatingIfNeeded: sum << 6)
        }

        let sequence = MultiBitSequence(self + [parity],
numberOfBits: 11)
        return sequence.map { words[$0] }.joined(separator: " ")
    }
}
```

With this last missing piece, we're able to recreate that unforgettable mnemonic:

```
data.humanReadableEncodedString()
// LONG IVY JULY AJAR BOND LEE
```

Recap

- Binary-to-text encoding formats represent binary data with text, which can be helpful for moving data across API boundaries.
- Use the `String(_:radix:uppercase:)` initializer to get binary, octal, and hexadecimal representations of fixed integers.
- Use Foundation APIs to encode and decode Base64 strings.
- You can create custom, domain-specific binary-to-text encoding formats that optimize for certain criteria, such as message size, clarity, or usability.

Chapter 6:

Parsing Data From Text

The Aeronautical Fixed Telecommunication Network (AFTN) is a worldwide system of communication used in aviation. Stations can communicate with one another by transmitting and receiving messages in a protocol-defined format.[1]

Here's an example of what an AFTN message looks like:

```
ZCZC NRA062 270930
GG KHIOYYYX
311521 KTTDZTZX

AIR SWIFT FLIGHT 42
CANCELED

NNNN
```

AFTN messages consist of capital letters, digits, punctuation, and spaces,[2] Messages are delimited by a start-of-message signal ("ZCZC") and an end-of-message signal ("NNNN"); both character sequences were chosen because they're unlikely to appear in a message otherwise.

Each message comprises a *heading*, its *destination* and *origin*, and the *text* itself.

The **heading** contains a six-character *transmission identifier* followed by a space and an optional *additional service indication*.

The **destination** contains a two-character *priority indicator* (DD) and an eight-letter sequence designating the *destination address*.

---

1. AFTN message syntax is defined by the International Civil Aviation Organization (ICAO) in Annex 10 of *Convention on International Civil Aviation - Aeronautical Telecommunications Volume II, Sixth Edition* (2001).

2. AFTN messages may look like they're ASCII, but they're actually encoded in ITA-2, a 5-bit teleprinter code introduced in 1924 — long before ASCII in 1963.

Messages are transmitted in order of priority, starting with distress (KK) messages, followed by urgent (DD) and safety (FF) messages, and finally informational (GG) and administrative (KK) messages. An address consists of a four-letter ICAO location indicator, followed by a three-letter organization indicator, and a single letter specifying a department within that organization (if unspecified, the letter X is used).

The **origin** contains a six-digit sequence indicating the *filing time* of the message and an eight-letter sequence designating the *origin address*. The filing time consists of three groups of 2-digits each for the *day of the month*, *hour*, and *minute*. The origin address follows the same structure as the destination address in the previous line.

Looking back at the example, we see that this was a informational (GG) message sent on the 10th day of the month at 9:41AM GMT from the aerodrome control tower (ZTZ) at Portland–Troutdale Airport (KTTD) to the aircraft operating agency (YYY) at Portland-Hillsboro Airport (KHIO), with a message that Air Swift flight 42 has been canceled.

By comparison, when apps communicate over the internet, messages are most often encoded in formats like JSON and XML, for which there are…

And indeed, there have been longstanding discussions about transitioning to industry standards like SMTP and XML.

It's rare that you get an opportunity to do this yourself.

Scanners

`Scanner` is a Foundation class that you can use to scan through a string from start to finish and extract substrings and numbers. It has robust support for parsing numbers from decimal, hexadecimal, and

floating-point representations — both localized and non-localized — as pragmatic affordances that make it a compelling choice for parsing semi-structured information from strings.

...or at least it would be, if it weren't awful to use from Swift.

Apple goes to extraordinary lengths make system frameworks look and feel like conventional Swift modules. A great example of this is how ~~Foundation drops its NS prefix~~ Objective-C methods that return an `NSError` by reference to indicate failure are imported as Swift methods that throw. For example, the Objective-C method `NSFileManager -removeItemAtURL:error:` is imported by Swift as `FileManager.removeItem(at:)`:

```
- (BOOL)removeItemAtURL:(NSURL *)URL
                  error:(NSError **)error;
```

```
func removeItem(at: URL) throws
```

Unfortunately, `NSScanner` methods don't follow this pattern, and consequently get a more *"by-the-books"* treatment with it has its APIs imported. Contrast the previous example with how `NSScanner` fares in its transition to Swift.

```
- (BOOL)scanUpToString:(NSString *)string
            intoString:(NSString * _Nullable *)result;
```

```
func scanUpTo(_ string: String,
    into result: AutoreleasingUnsafeMutablePointer<NSString?>?)
-> Bool
```

`AutoreleasingUnsafeMutablePointer<NSString?>??` 🤢😤!

As if it weren't bad enough that `scanUpTo(_:into:)` returns a `Bool` instead of throwing an error, it produces an `NSString` instead of a Swift `String`. And to add insult to injury, it's actually a pointer to an *optional* `NSString`, so we don't even get the benefit of knowing that a scan result is guaranteed even if successful.

> **Tip:** If the declaration for an Objective-C method gets *longer* when imported into Swift, you're gonna have a bad time.

For better or for worse, this pattern, whereby a function takes a pointer to a variable and populates it with a result of performing its operation, is known as *return-by-reference*. Thanks to tuples and language-level error handling, you don't often see this pattern for multivalued return in Swift code. To understand why let's see it in action as we use `Scanner` to parse our AFTN message.

Setting up a `Scanner` is pretty much what you expect. After passing the string to be scanned to the initializer, you have the opportunity to configure behavior like case sensitivity and characters to be skipped.

```
let scanner = Scanner(string: string)
scanner.caseSensitive = true
scanner.charactersToBeSkipped = .whitespacesAndNewlines
```

Scanners start at the beginning of a string and advance forward each time you scan for a value. AFTN messages start with the character sequence `ZCZC`, so our first step is to scan for that.

```
scanner.scanString("ZCZC", into: nil)
```

This method call returns `true` if `string` starts with `"ZCZC"` and advances the scan location to the position immediately following the second letter `C`. We pass `nil` to ignore the value produced by `scanString(_:into)` because it will be the same as the string we're scanning for.

Though when you *are* interested in the scanned value, that's when things get more annoying.

The next piece of information to parse after the start-of-message signal (ZCZC) is the transmission identifier, which consists of six alphanumeric characters.

To have a method return a value by reference in Swift, you must first declare a variable (so `var`) with an optional type. In our case, we declare an optional `NSString` variable *(thanks again, Objective-C*) and pass it by reference into the second parameter using the & prefix.

```
var transmission: NSString?
scanner.scanCharacters(from: .alphanumerics,
                       into: &transmission)
```

And actually, if we want to do this the right way, we need to add some error handling, like in a `guard` statement

```
var transmission: NSString?
guard scanner.scanCharacters(from: .alphanumerics,
                             into: &transmission)
else {
    // ...
}
```

In isolation, this is annoying but tolerable. However, if you're scanning a string with nontrivial length or complexity — as we are in our example — these minor annoyances add up quick.

This is the point where most would-be `Scanner` users give up and try another approach. Which is a shame, because with a little bit of TLC, we can turn it into a surprisingly capable API.

Scanner, *à la* Swift

As a first step, let's give `Scanner` the `Error` type it deserves. Whereas normally you might declare an `Error` type with an enumeration, `Scanner` doesn't really distinguish any failure cases beyond returning `false`, so here we use a structure with a property to track the location of the scan failure.[3]

```
import Foundation

extension Scanner {
    struct Error: Swift.Error {
        let location: Int
    }
}
```

Note: You can set the `scanLocation` property on a `Scanner` to backtrack after an error.

A proper `Error` type allows for a proper set of `scan` methods that either return the scanned value (directly, not by reference) or throws an error.

```
extension Scanner {
    func scan(_ characters: CharacterSet) throws -> String {
        var result: NSString?
        guard scanCharacters(from: characters,
                             into: &result),
            case let string? = (result as String?)
        else {
            throw Error(location: self.scanLocation)
        }

        return string
    }
}
```

3. Another option would be to define an `Error` enumeration with cases for `failed` and `endOf String`, the latter being thrown if the `isAtEnd` property returns `false` at the start of a refined `scan` method.

This more conventional approach offers more than just syntactic sugar: it goes a long way to ensure correct behavior.

Parsing AFTN Messages with Scanner

Let's start by initializing our original message into a Swift `String` with a multi-line string literal.

```
let message = """
ZCZC NRA062 270930
GG KHIOYYYX
311521 KTTDZTZX

AIR SWIFT FLIGHT 42
CANCELED

NNNN
"""
```

Thanks to our refactoring efforts, we're able to knock out each item line-by-line and consolidate error handling within a single do statement.

The result? A totally reasonable approach to parse our original AFTN message, if we might say so ourselves.

```
import Foundation

let scanner = Scanner(string: message)
scanner.charactersToBeSkipped = .whitespacesAndNewlines

do {
  try scanner.scan("ZCZC")
  let transmission = try scanner.scan(.alphanumerics)
  let additionalServices = try scanner.scan(.decimalDigits)
  let priority = try scanner.scan(.letters)
  let destination = try scanner.scan(.uppercaseLetters)
  let time = try scanner.scan(.decimalDigits)
  let origin = try scanner.scan(.uppercaseLetters)
  let text = try scanner.scan(upTo: "NNNN")
} catch {
  // ...
}
```

Tip: We can reduce the amount of boilerplate code in our extension on `Scanner` through clever use of *partial function application*.[4] In Swift, instance methods can be treated like functions that take an instance and return a function type. Therefore, methods that share a method signature — taking a pointer to a type and returning a `Bool` — such as `scanFloat` and `scanDouble` can be passed as an argument to a private method (see the `Scan<T>` type alias). The only other step is to manually allocate a pointer large enough to hold the result of calling the passed instance method.

```
extension Scanner {
  private typealias Scan<T> =
    (Scanner) -> ((UnsafeMutablePointer<T>?) -> Bool)

  private func scan<T>(_ partial: Scan<T>) throws -> T {
    let capacity = MemoryLayout<T>.stride
    let pointer = UnsafeMutablePointer<T>
                    .allocate(capacity: capacity)
    defer { pointer.deallocate() }

    guard partial(self)(pointer) else {
      throw Error(location: self.scanLocation)
    }

    return pointer.pointee
  }

  func scan(_ type: Float.Type) throws -> Float {
    return try scan(Scanner.scanFloat)
  }

  func scan(_ type: Double.Type) throws -> Double {
    return try scan(Scanner.scanDouble)
  }

  // etc.
}
```

You can find the full implementation of our `Scanner` overhaul in the sample code for this chapter.

4. Not to be conflated with *currying*, which is the process of converting an *n*-ary function into *n* unary functions (`f(x,y,z)` → `f(x)(y)(z)`). You can partially-apply both curried and non-curried functions by using some, but not all, of the resulting functions.

With a modicum of refinements to bring it in line with Swift best-practices, `Scanner` proves itself to be a delight to work with. It may not be the most sophisticated option available to us, but it's convenient to use and good enough for a variety of tasks — a pragmatic tool that gets the job done.

The next approach we'll talk about is also well-known for its ability to produce quick-and-dirty solutions to problems: *regular expressions*.

Regular Expressions

A *regular expression* is a mechanism for searching text that uses a pattern language consisting of characters and symbols. Regular expressions can match individual characters, sets of characters, or predefined-categories in sequence, alternation, or repetition, and capture grouped values by name or position.

Although regular expressions are often talked about as a single concept, there's no single authoritative standard or implementation that everyone agrees on.[5]

IEEE POSIX defines standards for Basic Regular Expressions (BRE) and Extended Regular Expressions (ERE), but they hold surprisingly little influence over most implementations, which instead model their functionality after Perl's regular expression engine. Non-standard features like extended character classes (`\d` for digits, etc.), multiline matching metacharacters (^ and $ for the beginning and end of lines), and minimal ("ungreedy") matching have gained such widespread adoption as to become a *de facto* standard, most commonly by way of the C library Perl Compatible Regular Expressions (PCRE).[6]

---

5. Among these points of contentions is the proper abbreviation of the term "regular expression": *"regex"* or *"regexp"*? For what it's worth, Google Trends shows a steady decline in the popularity of the term "regexp" since 2004, relative to an (albeit sporadic) rise for "regex" during that period.

6. Based on what our discussion of regular expressions so far, it shouldn't be surprising that there are nontrivial differences in behavior between regular expressions in PCRE and its ostensible namesake, Perl.

And yet, in spite of this chaotic patchwork of competing standards, regular expressions remain an invaluable tool for programmers across different languages and platforms. There's sufficient overlap across the various regex engines to satisfy the vast majority of users and use cases — and for anyone and anything else, most are content to make the most of the implementation native to their particular platform.

For Swift and Foundation, that implementation is `NSRegularExpression`.

NSRegularExpression

As a general observation, Objective-C and Swift code seems to use regular expressions less frequently than other, comparable languages.[7] It's unclear whether this effect (insofar as it might be observed) is caused by fundamental differences in apps compared to other programs or better explained by cultural trends within Apple. But here's one hypothesis: *nobody uses `NSRegularExpression` because it's no fun to use.*

In most other programming languages, regular expressions are a quick one-liner. For example, here's what it's like in Ruby:

```
"Boeing 747".scan(/\d+/) // ["747"]
```

7. Admittedly, this claim is based primarily on anecdata and an extremely cursory analysis of GitHub code search results.

But with `NSRegularExpression`, the entire process is bogged down by pointless ceremony. Every operation requires an options argument and a range; you can't omit either if you just want to match on the whole string normally. Here's the equivalent code in Swift:

```
import Foundation

let regex = try NSRegularExpression(pattern: #"\d+"#, options:
[])

let string = "Boeing 747"
let stringRange = NSRange(string.startIndex..<string.endIndex,
                          in: string)
regex.matches(in: string, options: [], range: stringRange)

if let match = regex.firstMatch(in: string,
                                options: [],
                                range: stringRange),
   let matchRange = Range(match.range, in: string)
{
   print(string[matchRange])
   // Prints "747"
}
```

Tip: In fairness, there is *some* joy to be found in Foundation, but it's not well-known. You can bypass `NSRegularExpression` entirely for lightweight search and replace operations by calling the Foundation-provided `String` methods `range(of:options:)` and `replacingOccurrences(of:with:options:)` with the `.regularExpression` option.

```
import Foundation

"Boeing 747".range(of: #"\d+"#,
                   options: [.regularExpression])
// {location: 6, length: 3}
```

Like `Scanner`, `NSRegularExpression` is a Foundation class that suffers limited usage due to poor usability. But if you can look past its cosmetic flaws, you'll be rewarded with a powerful tool for working with text.

Speaking of inner-beauty, let's talk about the engine that powers `NSRegularExpression`: *ICU*.

ICU Regular Expressions

ICU, or International Components of Unicode, is the C largely responsible for internationalization on Apple platforms, including localized string operations, date and number formatting, string transforms, and — of course — regular expressions.

To our earlier point about "regular expressions" meaning different things on different platforms, it's quite helpful to know what we have to work with in Swift. Reasonably assured that `NSRegularExpression` supports everything ICU does, let's take a look at some of the nonstandard features that we can take advantage of for our task at hand:

Named Capture Groups

Regular expressions may contain one or more *capture groups* that let you extract individual values from values.

The most common form of capture groups are *positional*, enclosed by parentheses (`(...)`); when a match is found, each captured values is accessible in the order that they're declared. For example, when matching the regular expression `/(Hello|Hi), (\w+)!/`, the first group captures the greeting ("Hello" or "Hi") and the second group captures the name.

Positional capture groups are convenient for simple expressions but are often difficult to keep track of for more complex patterns — particularly those containing nesting, alternation, or conditionals.

In those cases, it can be helpful to use *named capture groups* instead, which allow captured values to be extracted by a string key rather than a positional index. We could rewrite the previous example to use named captured groups with `/(?<greeting>Hello|Hi), (?<name>\w+)!/`.

Named capture groups aren't a feature of all regular expression engines, but, fortunately for us, they are supported by ICU. We'll take advantage of them to make sense of the various parts of our parsed AFTN message.

Metacharacters

Regular expressions contain characters, which are matched literally, and *metacharacters*, which match a set or category of characters.

Programmers are most familiar with metacharacters like `\w` for word characters, `\d` for digits, `\s` for whitespace characters, and the capitalized variants of each to denote their negation (such as `\S` for non-whitespace characters), not to mention conventional escape sequences like `\n` for newlines and `\t` for tabs.

However, there are several, more obscure, less standardized metacharacters made available to us by the ICU regular expression engine. Here are some of the most interesting and useful ones for us to be aware of:

Metacharacter	Description
`\A`	Match at the beginning of the input.[8]
`\b` `\B`	Match if the current position is / isn't a word boundary.
`\h` `\H`	Match horizontal whitespace character / any other characters
`\N{}`	Match the character with the specified name.
`\p{}` `\P{}`	Match any character with / without the specified Unicode Property.
`\x{}`	Match the character with the specified code point.
`\X`	Match a grapheme cluster

Note: For a full list of supported metacharacters, see the ICU User Guide.

Because AFTN is strictly alphanumeric, we won't have occasion to break out any of the Unicode metacharacters this time. However, `\h` will prove invaluable because of the final feature of ICU in our discussion: *flag settings*.

8. `\A` matches the beginning of a string, whereas `^` matches the beginning of a line. Despite `^` being more well-known, `\A` is most often the correct choice for anchoring matches.

Matching Options and Flag Settings

`NSRegularExpression` objects are initialized with a pattern along with any combination of the matching options provided by the `NSRegularExpression.Options` type. These matching options change the behavior of the regular expression. For example, the `.caseInsensitive` option allows letters in the pattern to match independent of case.

Most of these options can be instead set directly in the regular expression itself using the flag settings operator, `(?ismwx-ismwx)`:

Matching Option	Equivalent ICU Flag Setting
caseInsensitive	i
allowCommentsAndWhitespace	x
ignoreMetacharacters	
dotMatchesLineSeparators	s
anchorsMatchLines	m
useUnixLineSeparators	
useUnicodeWordBoundaries	w

For example, specifying the `.caseInsensitive` matching option has equivalent behavior to including the pattern `(?i)` at the beginning of a regular expression.

```
// Regex with options
try NSRegularExpression(pattern: #"hello"#,
                        options: [.caseInsensitive])

// Equivalent regex using flags
try NSRegularExpression(pattern: #"(?i)hello"#,
                        options: [])
```

An argument could be made for the first example being easier to understand (it is English, after all), but we generally recommend flag settings over `NSRegularExpression` matching options, if not for the sake of portability and self-descriptiveness, but then for the ability to enabled and disable flag settings within a pattern, either globally or within a parenthesized expression `(?ismwx-ismwx: ...)`.

Parsing AFTN Messages with NSRegularExpression

Applying the features of ICU regular expressions described above, we're now able to construct a pattern that matches our original message that meets our standards for clarity and usability.

As a refresher, here's the original AFTN message that we want to parse:

```
let message = """
ZCZC NRA062 270930
GG KHIOYYYX
311521 KTTDZTZX

AIR SWIFT FLIGHT 42
CANCELED

NNNN
"""
```

We could, ostensibly, match up the structure of our pattern with the message line-for-line, but that's likely more trouble than its worth; applying the `(?x)` flag setting improves readability and makes significant whitespace more explicit (thanks to our new friend, the `\h` horizontal whitespace metacharacter). And because we're extracting (at least) eight different values from a match, it'll be helpful to use named capture groups. Last, but not least, we'll take advantage Swift 5's new raw multi-line string literal, which is a perfect fit for this use case; now we can write all of our metacharacters without extra escape sequences.

```
let pattern = #"""
(?x-i)
ZCZC \h
(?<transmission>[A-Z]{3}[0-9]{3}) \h
(?<additionalService>[0-9]{0,8}) \n

(?<priority>[A-Z]{2}) \h
(?<destination>[A-Z]{8}) \n

(?<time>[0-9]{6}) \h
(?<origin>[A-Z]{8}) \n+

(?<text>[[A-Z][0-9]\h\n]+) \s*
NNNN
"""#
```

Beautiful. So beautiful, in fact, that we don't even mind all of the setup needed to feed this into `NSRegularExpression`.

```
let regex = try NSRegularExpression(pattern: pattern,
                                    options: [])

let stringRange = NSRange(string.startIndex..<string.endIndex,
in: string)
if let match = regex.firstMatch(in: string,
                                options: [],
                                range: stringRange)
{
    for name in [
        "transmission", "additionalService",
        "priority", "destination",
        "time", "origin",
        "text"
    ] {
        let nameRange = match.range(withName: name)
        if nameRange.location != NSNotFound,
            let range = Range(nameRange, in: string)
        {
            print("\(name): \(string[range])")
        }
    }
}
```

Note: The `range(withName:)` method is a relative newcomer, added to `NSTextCheckingResult` in iOS 11 and macOS High Sierra. If you need support older platforms, you can access captured groups by position using the `range(at:)` method.

Regular expressions are an indispensible tool for text processing. However, there are fundamental limitations to what you can do with them. To understand what those are, let's take a look at our third and final approach to parsing: *parser generators*

Parser Generators

Writing parser code by hand can be a time-consuming and error-prone process. Fortunately, it's also one that lends itself to automation. For this reason, many developers choose instead to define a formal grammar separately from any specific implementation and use a *parser generator* tool to write the code for them.

One such tool is ANTLR.[9]

ANTLR v4 natively supports Swift as a runtime target, meaning that it can read a grammar file and generate Swift parser code for that grammar.

...though, to be clear, the code that ANTLR *does* generate isn't going to win any beauty contests: Abstract base classes, functions masquerading as computed properties, verbose and inconsistent API naming... nothing about it is particularly *"Swifty"*. But the good news is that you can hide any ugliness as an implementation detail, and provide a clean, conventional interface to this underlying functionality.

Regular vs. Context-Free Grammars

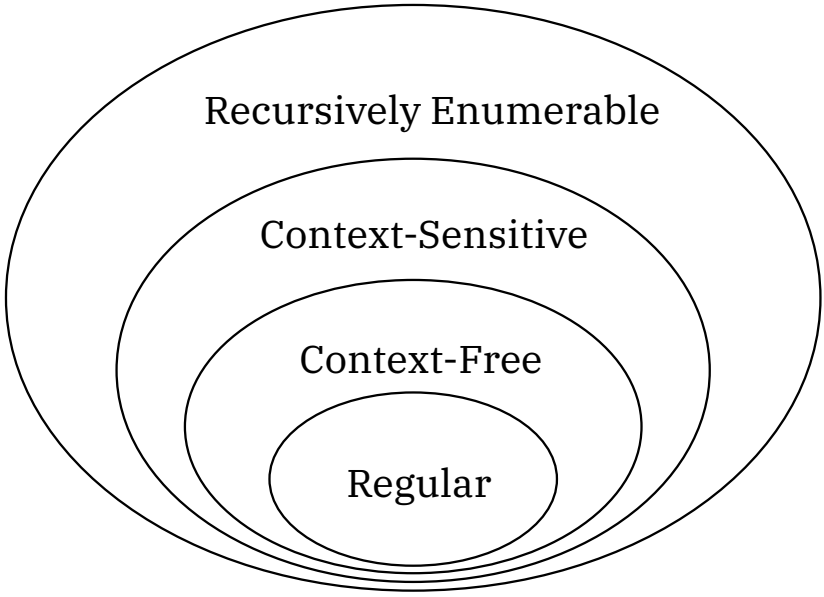
There's a well-known post on Stack Overflow explaining why attempting to parse HTML with a regular expression will have undesirable consequences. but it's the second-most-upvoted post that really answers the question:

> *HTML is a Chomsky Type 2 grammar (context-free grammar) and RegEx is a Chomsky Type 3 grammar (regular grammar).*

9. Pronounced "antler".

Since a Type 2 grammar is fundamentally more complex than a Type 3 grammar (see the Chomsky hierarchy), it is mathematically impossible to parse XML with RegEx.

In his 1956 paper *"Three models for the description of language"*, Noam Chomsky constructs a containment hierarchy describing the different categories of formal grammars.[10]



Within this classification, regular grammars are the simplest grammar. Formally, a regular language is one that can be matched by a *finite state machine* — specifically, a *deterministic finite automaton*. Informally, a regular language is one that can be matched by a regular expression (hence its name).

Regular expressions can match relationships like "A followed by B," "Either A or B," or "A followed by one or more instances of B."

---

10. Chomsky, Noam (1956). "Three models for the description of language". IRE Transactions on Information Theory (2): 113–124. doi:10.1109/TIT.1956.1056813.

However, they can't express "Every A is eventually followed by a B" or "Some instances of A followed by the same number of instances of B."[11]

Expressing those more complex relationships requires a more complex grammar: *context-free grammars*.

Formally, a context-free language can be matched by a sequence of replacement rules. Informally, a context-free language is one that can be parsed by an *LL parser*.

LL Parsers

An LL parser is a top-down parser that parses the input from **L**eft to right, always expanding **L**eftmost non-terminals.

ANTLR, like many parser generators, creates an LL(*) parser based on the replacement rules defined in the provided grammar.

Before we dive into the nitty-gritty of using ANTLR to generate an LL parser, a disclaimer about the motivating example:

We saw in the previous section that it's possible to parse AFTN messages with regular expressions, which means that AFTN is a regular language. Because every regular language is also context-free, we can use an LL parser to parse AFTN messages just as well.

As we'll see, this approach is significantly more involved than using a regular expression and is — strictly speaking — overkill in this situation. But after learning this technique with a simple example, you'll be able to apply it to solve problems that require context-free parsing.

So keep that in mind as we wade into the world of generated parser code.

---

11. This is why — to the point of the Stack Overflow Question — you can't write a regular expression to determine if nested tags are matching closing tags in the correct order.

Packaging Antlr4 Runtime

As we mentioned before, ANTLR provides a Swift runtime. However, in order to be usable from the Swift Package Manager, it needs to be extracted into a separate, semantically-versioned Git repository. We can accomplish this using the `git subtree split` command, which isolates all of the activity in the specified directory (`runtime/Swift`) and surfaces it as a separate repository.

```
$ git clone https://github.com/antlr/antlr4.git
$ cd antlr4
$ git subtree split --prefix=runtime/Swift/
$ tree . -L 2
.
├── Package.swift
├── Sources
│   ├── Antlr4
│   ├── Info-IOS.plist
│   └── Info-OSX.plist
├── Tests
│   ├── Antlr4Tests
│   ├── Info.plist
│   └── LinuxMain.swift
└── boot.py
```

Note: A mirror of the ANTLR v4 Swift runtime suitable for use from Swift Package Manager is provided by on GitHub

Creating a Swift Library Package

Now that we're able to pull in the ANTLR v4 Swift runtime as a dependency, the next step is to create a package using the `swift package init` command.

```
$ mkdir AFTN
$ cd AFTN
$ swift package init --type library
```

In our `Package.swift` file, declare a dependency for `Antlr4` and associate it with the `AFTN` target.

```
// swift-tools-version:4.2

import PackageDescription

let package = Package(
  name: "AFTN",
  dependencies: [
    .package(
      url: "https://github.com/Flight-School/SwiftAntlr4.git",
      .branch("master")
    )
  ],
  targets: [
    .target(
        name: "AFTN",
        dependencies: ["Antlr4"]
    )
  ]
)
```

Writing a Grammar

ANTLR generates LL parsers based on a grammar file written in ANTLR Meta-Language.

Create a new file named `aftn.grammar`, located in a `Resources` directory at the top level of our package. In the first line, write the `grammar` keyword followed by the name of our grammar: AFTN.

```
grammar AFTN;
```

A grammar file contains a set of lexer rules and a set of parser rules. A *lexer* takes the stream of characters and transforms them into individual tokens. These tokens are then handled by a *parser*.

Tip: ANTLR lexer and parser rules can be split into separate files, which may be helpful when working with large or complex grammars.

Lexer Tokens

AFTN messages consist of digits, uppercase letters, newlines, and spaces We can encode these constraints as lexical rules using a syntax similar to regular expression character sets.[12]

```
DIGIT: [0-9];
ALPHA: [A-Z];
CRLF: ('\r\n' | '\n');
SP: ' ';
```

We also create lexical rules for the start-of-message signal (`ZCZC`) and end-of-message signal (`NNNN`).

```
SOM: 'ZCZC';
EOM: 'NNNN' EOF;
```

Parser Rules

Parser rules define the tokens that will be processed by the parser.

The process for breaking down the message into rules is much the same as how we described them in prose: we start with the top-level message and identify each part of the message.

```
message:
    SOM
    heading CRLF
    destination CRLF
    origin CRLF*
    text CRLF+
    EOM;
```

We then subdivide each of these parts into their constituents...

```
heading: SP transmission SP additionalService?;
transmission: (ALPHA | DIGIT)+;
additionalService: DIGIT+;
```

---

12. By convention, lexical rules have uppercase names and parser rules have lowercase names.

...and keep subdividing until each parser rule can be described by a sequence of lexer tokens.

```
destination: priority SP address;
origin: filingTime SP address;

priority: ALPHA ALPHA;

filingTime: day hour minute;
day: DIGIT DIGIT;
hour: DIGIT DIGIT;
minute: DIGIT DIGIT;

address: location organization department;
location: ALPHA ALPHA ALPHA ALPHA;
organization: ALPHA ALPHA ALPHA;
department: ALPHA;

text: (CRLF? (ALPHA | DIGIT | SP)+)+;
```

Compiling the Grammar

If you don't already have ANTLR on your system, you can install it using Homebrew with the following command:

```
$ brew install antlr
```

Once installed, you should have the `antlr` executable accessible from your `$PATH`. Run the following command to have ANTLR read the grammar file and generate Swift code to the `Sources/AFTN` directory in our package.

```
$ antlr -Dlanguage=Swift \
        -visitor \
        -listener \
        -message-format gnu \
        -o Sources/AFTN \
        Resources/aftn.g4
```

Altogether, the `antlr` command generates the following files:

```
$ tree ./Sources/AFTN/Resources
├── AFTN.interp
├── AFTN.tokens
├── AFTNBaseListener.swift
├── AFTNBaseVisitor.swift
├── AFTNLexer.interp
├── AFTNLexer.swift
├── AFTNLexer.tokens
├── AFTNLexerATN.swift
├── AFTNListener.swift
├── AFTNParser.swift
├── AFTNParserATN.swift
└── AFTNVisitor.swift
```

Tip: Use a Makefile to automate the process of generating source code from your grammar.

```
Sources/AFTN/Resources: Resources/AFTN.g4
	@antlr -Dlanguage=Swift \
		-visitor \
		-listener \
		-message-format gnu \
		-o Sources/AFTN \
		Resources/aftn.g4

.PHONY: clean
clean:
	@rm -rf Sources/AFTN/Resources
```

Implementing the Message Type

Up until this point, we haven't done much in the way of data modeling. In the previous examples, all of the information being parsed out were `String` values. Let's fix that by declaring a proper `Message` type:

```
struct Message: {
    enum Priority: String {
        case distress = "SS"
        case urgency = "DD"
        case safety = "FF"
        case information = "GG"
        case administrative = "KK"
    }

    struct Address: {
        let location: String
        let organization: String
        let department: String
    }

    let transmission: String
    let additionalService: String?
    let priority: Priority
    let destination: Address
    let origin: Address
    let filingTime: (day: Int, hour: Int, minute: Int)
    let text: String
}
```

Extending Message to Include Parsing

You may be disappointed to learn that the generated Swift code doesn't magically adapt the parsed results to the structure of the `Message` type.[13] The good news is that it's not too much extra work — all you need to do is create a subclass of the generated `AFTNBase Listener` class and override the `enter` methods corresponding to each parser token to extract the text value from the token.

```
extension Message {
    final class Listener: AFTNBaseListener {
        var transmission: String?
        // ...

        override func enterTransmission(_ ctx:
AFTNParser.TransmissionContext) {
            self.transmission = ctx.getText()
        }

        // ...
```

Because address tokens occur in both the origin and destination. we need to consult the location context to determine which property to assign.

```
override func enterLocation(_ ctx: AFTNParser.LocationContext) {
    switch ctx.parent?.parent {
    case is AFTNParser.OriginContext:
        self.originLocation = ctx.getText()
    case is AFTNParser.DestinationContext:
        self.destinationLocation = ctx.getText()
    default:
        return
    }
}
```

13. I mean, not every API can be as amazing as `Codable`.

`exitMessage(_:)` is the last method called by our listener when a message is parsed, making it the ideal place to collect the extracted values and attempt to construct a valid `Message` object.

```
override func exitMessage(_ ctx: AFTNParser.MessageContext) {
    guard let transmission = self.transmission,
          let priorityIndicator = self.priorityIndicator,
          let priority = Message.Priority(rawValue:
priorityIndicator),
          // ...
    else {
        return nil
    }

    self.message = Message(transmission: transmission, /* ...
*/)
}
```

Hiding Complexity in the Initializer

Our ideal interface for parsing a message would be an optional initializer that takes a single `String` argument and throws any parsing errors.

```
extension Message {
    init?(_ string: String) throws {}
}
```

But really, we'd settle for anything that abstracts away the implementation details of parsing with ANTLR's generated interface:

```
import Antlr4

let data = Array(string)
let input = ANTLRInputStream(data, data.count)
let lexer = AFTNLexer(input)
let tokens = CommonTokenStream(lexer)
let parser = try AFTNParser(tokens)
let listener = Listener()
let walker = ParseTreeWalker()
try walker.walk(listener, parser.message())
guard let message = listener.message else {
    return nil
}
```

Granted, parsing is a complex problem, and it makes sense to keep all of the different steps separate. But at a certain point, you really start to wonder how many of those really needed to be there.

To summarize, parsing a message with Antlr involves the following:

1. Converting a string into an array of characters
2. Reading the array of characters into an input stream
3. Feeding the input stream into a lexer
4. Constructing a feed of tokens produced by the lexer
5. Initializing a parser with the lexer tokens
6. Creating a walker to traverse the AST generated by the parser
7. Creating a listener to relay the start and end of each parser token
8. Extracting the parsed message, if it was valid

Then again, that's not *too* bad considering how much is done for us by way of generating the interface in the first place.

Parsing the Message, From the Top

Putting aside all of our whinging, looking at the final interface, from tip to tail, it's hard not to come away with a sense of satisfaction.

```
import AFTN

let string = """
ZCZC NRA062 270930
GG KHIOYYYX
311521 KTTDZTZX

AIR SWIFT FLIGHT 42
CANCELED

NNNN
"""
```

```
do {
  let message = try Message(string)
  message.priority // GG
  message.destination.location // KGIO
  message.destination.organization // YYY
  message.destination.department // X
  message.filingTime // (day: 31, hour: 15, minute: 21)
  message.text // "AIR SWIFT FLIGHT 42\nCANCELED"
} catch {
  // ...
}
```

In this chapter, we looked at three different ways to parse a real-world message format. If you ever encounter a text format that you need to parse, but a parser doesn't yet exist, you should be able to adapt at least one of these approaches into a workable solution.

But what if your app needs to understand text that doesn't have a defined structure?

In Chapter 7, we'll take an even broader look at how we can write programs to derive meaning from text written by humans.

Recap

- With some modification, Foundation's `Scanner` class offers a flexible and convenient solution for parsing substrings and numbers from text.
- `NSRegularExpression` provides an interface to the ICU regular expression engine.
- You can use a parser generator like ANTLR to generate code to lex and parse input from a grammar as an alternative to writing this code by hand.


4
Your gate has changed to it 15.
That flight is delayed by an hour.
S
NP
VP
D
N
V
N
The bear
reads a book

Chapter 7:

Natural Language Processing

Language is humanity's greatest invention, with computers not too far behind. It's only natural that we should use computers to improve how we understand and communicate with one another.

Up until this point, we've concerned ourselves only with how computers encode language in its written form, without concern for what that's being communicated. In this chapter, we enter into the domain of Linguistics to study the structure and meaning of language.

The term *natural language* describes any text written by or for humans. The use of computers to analyze, transform, or understand written human language, is known as *natural language processing*, or NLP for short.

New in iOS 12 and macOS Mojave, the Natural Language framework reorganizes and expands on existing NLP APIs with smarter machine-learning based internals.

We'll start this chapter by looking at the three main components of the framework: `NLTokenizer`, `NLTagger` and `NLLanguageRecognizer`. From there, we'll discuss additional natural language processing techniques and how to accomplish them in Swift, sometimes in combination with functionality provided by the Foundation and Natural Language frameworks.

Tokenization

The first step in most natural language processing tasks is to *tokenize* text into paragraphs, sentences, and words.

If you're an English speaker, you'd be forgiven for thinking that tokenization is a relatively simple, straightforward process:

"Split on newlines to get paragraphs, on periods to get sentences, and on spaces to get words — no problem!"

You might even pat yourself on the back for being able to express this with functional programming, like so:

```
let string = """
   Welcome to New York, where the local time is 9:41 AM.
"""

let paragraphs = string.split(separator: "\n")
let sentences = paragraphs.compactMap{ $0.split(separator: ".")
}
let words = sentences.compactMap{ $0.split(separator: " ") }
```

However elegant this solution is and how well it appears to work, consider what happens if you try to use it to tokenize Chinese text:

```
let string = """
   欢迎来到北京，当地时间是上午9:41。
"""
```

Like many Asian scripts, Chinese doesn't delimit words with spaces. It also uses a different character to terminate sentences.[1] Even determining paragraph boundaries within a document has its quirks; Unix-like systems, including macOS, end lines with a single line feed character (␊ or `"\n"`), whereas Windows and other systems end lines with a carriage return and line feed (␍␊, or `"\r\n"`).

1. U+3002 IDEOGRAPHIC FULL STOP instead of U+002E FULL STOP.

The reality is that tokenization is a very difficult task in NLP. Fortunately, you can leverage one such system on macOS and iOS by way of the `NLTokenizer` class:

```
let tokenizer = NLTokenizer(unit: .word)
tokenizer.string = string

let stringRange = string.startIndex..<string.endIndex
tokenizer.enumerateTokens(in: stringRange) { (tokenRange,
attributes) in
    let token = string[tokenRange]
    // ...
    return true // continue processing
}
```

You initialize `NLTokenizer` by specifying a tokenization unit — document, paragraph, sentence, word — and then set the `string` property to the text you want to segment. Next, call the `enumerateTokens(in:using:)` method, passing the range of the string to process (usually the entire string range) and a closure to execute for each token. The closure takes two arguments, the range of the token, as well as an `NLTokenizer.Attributes` value, which can tell you whether then token contains numbers, symbols, and/or emoji.

As we mentioned before, tokenization is often a starting point for other kinds of NLP tasks, and we'll see it come up throughout the rest of this chapter.

Tagging

Tokenization can tell you where word boundaries are, but it can't tell you anything about the words themselves. The process of associating information with tokens is called *tagging*.

There are many different ways in which natural language text can be tagged. Text can be tagged syntactically, differentiating between words, punctuation, and whitespace. Text can be tagged grammatically, distinguishing among nouns, adjectives, and verbs.

For multilingual text, you might tag each unit with its language or script, or you may use tags to identify people, places, and organizations.

All of this functionality (and more) is provided by the `NLTagger` class. Functionally, it works much the same as `NLTokenizer` with a `string` property and an enumeration method. But instead of initializing with a linguistic unit, you pass `NLTagger` an array of tag schemes that you are interested in:

```
let schemes: [NLTagScheme] = [.language, .script]
let tagger = NLTagger(tagSchemes: schemes)
tagger.string = string
```

Important: Each additional tag scheme increases the amount of processing time, so specify only schemes you're interested in.

Part of Speech Tagging

Among the most common uses of natural language tagging is to tag the *lexical class*, or part of speech of each word in a sentence. This lets you to differentiate between nouns, verbs, adjectives, and other grammatical entities.

Here's an example of how you use `NLTagger` to do part of speech tagging:

```
let string = """
   The sleek white jet soars over the hazy fog.
"""

let tagger = NLTagger(tagSchemes: [.lexicalClass])
tagger.string = string

let options: NLTagger.Options = [.omitWhitespace,
                                 .omitPunctuation]
tagger.enumerateTags(in: string.startIndex..<string.endIndex,
                     unit: .word,
                     scheme: .lexicalClass,
                     options: options) { (tag, tagRange) in
    if let partOfSpeech = tag?.rawValue {
        print("\(string[tagRange]): \(partOfSpeech)")
    }

    return true // continue processing
}
```

Running this code prints the following correspondences:

Token	Tag
The	Determiner
sleek	Adjective
white	Adjective
jet	Noun
soars	Verb
over	Preposition
the	Determiner
hazy	Adjective
fog	Noun

Named Entity Recognition

Another common form of tagging is *named entity recognition*, or NER. Using `NLTagger`, you can extract the names of people, places, and organizations from natural language text in several different languages.

The usage is nearly identical to the example above for part of speech tagging, except that instead of passing `.lexicalClass` you use `.nameType`, and in addition to the options to omit whitespace and punctuation, you can choose to join related name tokens, such as a person's given and family names,
into a single, tagged entity.

```
import NaturalLanguage

let string = """
   Fang Liu of China is the current Secretary General of ICAO.
"""

let tagger = NLTagger(tagSchemes: [.nameType])
tagger.string = string

let options: NLTagger.Options = [.omitWhitespace,
                                 .omitPunctuation,
                                 .joinNames]
tagger.enumerateTags(in: string.startIndex..<string.endIndex,
                     unit: .word,
                     scheme: .nameType,
                     options: options) { (tag, tagRange) in
    if let nameType = tag?.rawValue, tag != .otherWord {
        print("\(string[tagRange]): \(nameType)")
    }

    return true // continue processing
}
```

Here are the results of running the code above:

Token	Tag
Fang Liu	PersonalName
China	PlaceName
ICAO	OrganizationName

Note: Dr. Fang Liu began her second consecutive term as Secretary General for the International Civil Aviation Organization (ICAO) in March 2018.[2]

2. https://www.icao.int/secretariat/SecretaryGeneral/

Keyword Extraction

`NLTagger` makes it easy to get both part of speech and named entity tags in a unified set of results. Instead of passing the `.lexicalClass` and `.nameType` tag schemes individually, use `.nameTypeOrLexicalClass`.

This combination of tag schemes can be especially handy for doing *keyword extraction*, which is the first step in tasks like question answering or information retrieval. Given a closed domain like answering questions about the weather, a first-pass solution might involve using `NLTagger` to tag names and lexical classes and ignoring words that aren't, for example, nouns or place names.

```
import NaturalLanguage

let string = """
   What's the current temperature in Tokyo?
"""

let tagger = NLTagger(tagSchemes: [.nameTypeOrLexicalClass])
tagger.string = string

var taggedKeywords: [(NLTag, String)] = []
tagger.enumerateTags(in: string.startIndex..<string.endIndex,
                     unit: .word,
                     scheme: .nameTypeOrLexicalClass,
                     options: []]) { (tag, tagRange) in
    guard let tag = tag else { return true }
    switch tag {
    case .noun, .placeName:
        taggedKeywords.append((tag, String(string[tagRange])))
    default:
        break
    }

    return true // continue processing
}
```

Lemmatization

Lemmatization is the process of finding the *lemma*, or "root" for words in natural language text. For example, the words "flying", "flew", and "flown" are all different inflectional forms of the verb "fly".

`NLTagger` provides this functionality through its `.lemma` tag scheme:

```
import NaturalLanguage

let string = """
   Flying flights flew flyers flown.
"""

let tagger = NLTagger(tagSchemes: [.lemma])
tagger.string = string

tagger.enumerateTags(in: string.startIndex..<string.endIndex,
                     unit: .word,
                     scheme: .lemma,
                     options: []) { (tag, tagRange) in
    if let lemma = tag?.rawValue {
        print("\(string[tagRange]): \(lemma)")
    }

    return true // continue processing
}
```

Run this code, and you get the following results:

Token	Tag
Flying	fly
flights	flight
flew	fly
flyers	flyer
flown	fly

Notice how `NLTagger` is able to distinguish between the verb "fly" and the related nouns "flight" and "flyer". It also handles the capitalization in the word "Flying".

Normalizing text in this way can improve the accuracy of statistical models that use term frequency to classify documents. Without normalization, the sentence above wouldn't register as containing the term "fly"; after lemmatization, the term occurs 3 times.

We'll talk more about lemmatization later on in this chapter as part of our discussion of text classification.

Classification

The third major component of the Natural Language framework is the `NLLanguageRecognizer` class, which can — with remarkable accuracy — predict whether text is written in English or French or Japanese or any of the over 50 other supported languages.

Language recognition is a specific example of a larger problem in NLP, *classification*: assigning one or more categories to text based on its content. For example, you can use text classification to filter spam messages out of an email inbox, organize articles by topic (politics, business, sports, etc.), or determine how the percentage of positive reviews versus negative.

Let's start by looking at the language recognition functionality provided by `NLLanguageRecognizer` and then look at how the Natural Language framework can be used with Core ML for other text classification tasks.

Language Recognition

You could use `NLTagger` to determine the language and script of text by calling the `tag(at:unit:scheme)` method and specifying the `.language` tag schemes for the `.document` unit:

```
import NaturalLanguage

let string = """
   Sehr geehrte Damen und Herren,
   herzlich willkommen in Frankfurt.
"""

let tagger = NLTagger(tagSchemes: [.language])
tagger.string = string

if case let (tag?, _) = tagger.tag(at: string.startIndex,
                                   unit: .document,
                                   scheme: .language) {
    let language = tag.rawValue // "de"
}
```

However, `NLLanguageRecognizer` provides a more flexible and convenient way to do this:

```
let languageRecognizer = NLLanguageRecognizer()
languageRecognizer.processString(string)

languageRecognizer.dominantLanguage // German
```

In addition to the simpler API, `NLLanguageRecognizer` can tell you the probabilities among likely candidates. It can also be configured with weighted probabilities reflecting the *a priori* likelihood of certain languages. This can be helpful for apps specific to a locale whose primary language is similar to another language, such as Norwegian Bokmål (nb) and Danish (da). So an app that only expects input in, say, Norwegian and Swedish, could set the `languageHints` property to improve the accuracy of `NLLanguageRecognizer`:

```
import NaturalLanguage

let string = """
   God morgen mine damer og herrer.
"""

let languageRecognizer = NLLanguageRecognizer()
languageRecognizer.processString(string)

languageRecognizer.dominantLanguage?.rawValue // da

languageRecognizer.languageHints = [.norwegian: 0.5,
                                    .swedish: 0.5]

languageRecognizer.dominantLanguage?.rawValue // nb
```

Note: The Natural Language framework doesn't currently provide a comparable API to determine the dominant script of a document. Until one is provided, you can use `NLTagger` with the `.script` tag scheme in the manner described above.

Sentiment Analysis

Another example of text classification that we alluded to earlier is *sentiment analysis*, or determining the opinions expressed in a piece of text.

The most common approach to performing sentiment analysis is to go through a list of examples and manually rate each as either positive and negative. By looking at how often different words and phrases occur in each category, you can automatically predict the sentiment of a new example based on the terms it uses. This

is known as *supervised learning*. For example, you might find that positive reviews of an airline are more likely to include words like "on-time" and "comfortable", whereas negative reviews are more likely to use words like "late" and "crowded".

You can use Create ML to easily create a machine learning model for sentiment analysis from training data. For this example, we'll use the "Airline Twitter sentiment" data set[3] , which contains messages from customers to airlines on Twitter that have been categorized as either "positive", "negative", or "neutral".

In an Xcode Playground, we can load up the CSV file for our data set and specify the columns containing the text and the label. Create ML then automatically reads the data from the CSV file, tokenizes the data, and extracts the features. The resulting Core ML model (`.mlmodel`) file is then saved to disk.

```
import CreateML
import Foundation

guard let url = Bundle.main.url(forResource: "tweets",
                                withExtension: "csv")
else {
    fatalError("Missing required resource")
}

let trainingData = try MLDataTable(contentsOf: url)
let model = try MLTextClassifier(trainingData: trainingData,
                                 textColumn: "text",
                                 labelColumn:
"airline_sentiment")

let path = <#"path/to/SentimentClassifier.mlmodel"#>
try model.write(to: URL(fileURLWithPath: path))
```

Within a couple minutes, Create ML is able to generate a model with >99% accuracy that's 370 KB in size. *Not bad!*

3. CrowdFlower (2015) "Airline Twitter sentiment". https://www.figure-eight.com/data-for-everyone/, accessed October 2018.

The resulting Core ML model can be loaded by the `NLModel` class to predict the label of new text inputs:[4]

```
import NaturalLanguage

guard let url = Bundle.main.url(forResource:
"SentimentClassifier",
                                withExtension: "mlmodelc")
else {
    fatalError("Missing required file")
}

let model = try NLModel(contentsOf: url)

model.predictedLabel(for: "Nice smooth flight") // positive
model.predictedLabel(for: "Meh, it was alright") // neutral
model.predictedLabel(for: "Missed connecting flight") //
negative
```

The Natural Language framework provides a wealth of natural language processing functionality. However, it represents only a small fraction of what's possible.

The rest of this chapter will look at some additional NLP techniques that you can implement in Swift — some with only the standard library, and others in coordination with the Foundation and Natural Language frameworks.

Phonetic String Matching

A *phonetic algorithm* can be used to index words by how they sound when spoken, rather than how they're spelled.

4. Xcode automatically compiles Core ML models in iOS and macOS app bundles. To load a Core ML model in a Playground, you must compile the `.mlmodelc` file yourself by running the command `xcrun coremlc compile` from Terminal.

One of the first and most well-known of these is *Soundex*, developed nearly a century ago as a way for census workers and medical administrators to reconcile spelling variation of surnames (for example, “Smith” and “Smyth”).

Soundex represents each name with a code consisting of a letter followed by three numbers. The letter is the first letter of the name. The numbers are assigned based on the remaining consonants using the following substitutions:

Category	Letters	Number
labial plosives and fricatives	b, f, p, v	1
velars and alveolar fricatives	c, g, j, k, q, s, x, z	2
dentals	d, t	3
laterals	l	4
nasals	m, n	5
approximates	r	6

Long names are truncated, and short names are padded with zeroes. For example, the Soundex code for “Washington” is W252 and the code for “Lee” is L000.

Implementing the Soundex algorithm in code isn't particularly challenging: normalize the case, filter vowels, substitute letters for numbers, remove consecutive numbers with the same value, and right-pad with zeroes to length 4:

```
func soundex(_ string: String) -> String {
    guard case let (head?, tail) =
            (string.first, string.dropFirst())
    else {
        return string
    }

    return tail.lowercased()
               .compactMap(substitute)
               .reduce(into: "\(head)") { (result, character) in
                   if result.last != character {
                       result.append(character)
                   }
               }
               .prefix(4)
               .rpad(with: "0", to: 4)
}
```

We can now use our `soundex(_:)` method to (approximately) index surnames by pronunciation.

```
let names: [String] = [
    "Washington",
    "Lee",
    "Smith",
    "Smyth"
]

for name in names {
    print("\(name): \(soundex(name))")
}
```

Name	Soundex
Washington	W252
Lee	L000
Smith	S530
Smyth	S530

Soundex has limited usefulness beyond surnames written in English, but there are other algorithms like Double Metaphone that accommodate dictionary words as well as languages other than English.

Search engines often use phonetic algorithms in combination with conventional string metrics to find entries that are most likely related to a given input.

String Metrics

A *string metric* measures the distance between two strings.

For example, the Levenshtein distance between the word “fly” and “flew” is 2, because transforming from one to another requires a minimum of two operations: a substitution (“y” with “e”) and an insertion (“w”).

You can determine the Levenshtein distance between two sequences using the Wagner–Fischer algorithm.[5]

For example, here’s the matrix resulting from calculating the edit distance from “Saturday” to “Sunday”, where the bold entries trace the minimum path of edit operations:

		S	**a**	**t**	**u**	**r**	**d**	**a**	**y**
	0	1	2	3	4	5	6	7	8
S	1	**0**	**1**	**2**	3	4	5	6	7
u	2	1	1	2	**2**	3	4	5	6
n	3	2	2	2	3	**3**	4	5	6
d	4	3	3	3	3	4	**3**	4	5
a	5	4	3	4	4	4	4	**3**	4
y	6	5	4	4	5	5	5	4	**3**

5. R. A. Wagner and M. J. Fisher. “The string-to-string correction problem.” Journal of the Association for Computing Machinery, 21(1):168-173, January 1974.

Let's look at a Swift implementation of Levenshtein distance to understand how the algorithm works.

First, we declare a function that takes a source and target string and returns an integer distance. Those string parameters are converted into arrays of characters to allow random access by integer indices.

```
func levenshteinDistance(from sourceString: String,
                         to targetString: String) -> Int
{
    let source = Array(sourceString)
    let target = Array(targetString)
```

Intermediate results are stored in a 2-dimensional array. We define a custom `Matrix` type to abstract row and column access to a 1-dimensional array mapping. The first row and column are initialized with values equal to their position, which establishes a base edit distance. For the first row, it's the number of deletions necessary to transform into an empty string. For the first column, it's the number of insertions necessary to transform into the source prefix.

```
    var distance = Matrix(rows: source.count + 1,
                          columns: target.count + 1)

    for i in 1...source.count {
        distance[i, 0] = i
    }

    for j in 1...target.count {
        distance[0, j] = j
    }
```

The rest of the matrix is filled out by calculating the distance from the surrounding context. For each cell, the edit distance is the cost of inserting from the previous column, deleting from the previous row, or substituting from the previous row and column — whichever option requires the fewest edits. If the source and target have the same character at a given position, the additional cost for a substitution is zero.

```
    for i in 1...source.count {
        for j in 1...target.count {
            let δ = source[i - 1] == target[j - 1] ? 0 : 1
            distance[i, j] = min(
                distance[i, j - 1] + 1,      // insertions
                distance[i - 1, j] + 1,      // deletions
                distance[i - 1, j - 1] + δ   // substitutions
            )
        }
    }

    return distance[source.count, target.count]
}
```

Calling this method with the strings from our example yields the expected results.

```
levenshteinDistance(from: "Saturday", to: "Sunday") // 3
```

Spell Checking

One of the most useful applications of string metrics is *spell checking*.

Using a frequency list of English words we can suggest candidates with equal edit distance according to their overall likelihood. For this example, we'll create such a list using the Web 1T 5-Gram database, a collection of frequent 5-grams (more on n-grams in the next section).[6]

---

6. Brants, Thorsten, and Alex Franz. (2006) Web 1T 5-gram Version 1 https://catalog.ldc.upenn.edu/LDC2006T13, accessed October 2018.

A naïve implementation of a spell checker might enumerate over the entire word list and sort by edit distance to the target word. However, given a sufficiently large word list (such as the one provided by our corpus), this can take a long time.

As a first-pass optimization, we might recognize that the edit distance between two strings has a lower bound equal to the difference in length between them — at a minimum, you'll need to insert or delete that many characters. So rather than calculating the edit distance for every word, we can limit our search to only those words whose length is within a certain distance to the source word.

In our `SpellChecker` implementation, we maintain a `Set` of dictionary words as well as a dictionary of ranked words keyed by their length.

```
class SpellChecker {
    typealias RankedWord = (word: String, rank: Int)

    var metric: (String, String) -> (Int) = levenshteinDistance

    private var entries: Set<String> = []
    private var rankedWordsByLength: [Int: [RankedWord]] = [:]

    init(contentsOf file: URL) throws {
        let string = try String(contentsOf: file)

        var rank: Int = 1
        string.enumerateLines { (word, _) in
            self.entries.insert(word)

            let length = word.count
            var rankedWords = self.rankedWordsByLength[length]
?? []
            rankedWords.append((word, rank))
            self.rankedWordsByLength[length] = rankedWords

            rank += 1
        }
    }
}
```

To generate suggestions, we short-circuit evaluation if the word exists in our dictionary (meaning that it's spelled correctly). Next, we compile a list of candidates from those whose length is within ±

1 of the specified word. We then calculate the edit distance for each, segmenting those with distances of 1 or 2 into separate containers. Finally, we return the results from the the group of candidates with the least edit distance, ordered by frequency rank.

```
func suggestions(for word: String) -> [String] {
    guard entries.contains(word) else {
        return [word]
    }

    let length = word.count

    var edit1: [RankedWord] = []
    var edit2: [RankedWord] = []

    let candidates: [RankedWord] = (
        (rankedWordsByLength[length - 1] ?? []) +
        (rankedWordsByLength[length]     ?? []) +
        (rankedWordsByLength[length + 1] ?? [])
    )

    for candidate in candidates {
        switch self.metric(from: word, to: candidate.word) {
            case 1: edit1.append(candidate)
            case 2: edit2.append(candidate)
            default: continue
        }
    }

    return (!edit1.isEmpty ? edit1 : edit2)
                .sorted { $0.rank < $1.rank }
                .map { $0.word }
}
```

Trying this out for ourselves, we see it helpfully provide “spelling” as the first suggestion for the word “speling” [sic].

```
let spellChecker = try SpellChecker(contentsOf: url)

spellChecker.suggestions(for: "speling")
// ["spelling", "spewing", "sperling"]
```

Beyond the aforementioned phonetic algorithms, a more sophisticated spell checker might also take surrounding context

into consideration to find spelling errors and suggest replacements. This additional layer allows a spell checker to correct, for example, mistaken usage of there / their / they're, and the like.

More often than not, this surrounding context is provided by an n-gram.

N-Grams

An *n-gram* is a sequence of groups of *n* items that can be used to predict the next item in a sequence.

When used to analyze text, you might construct n-grams for the characters in a word or the words in a sentence. For example, an n-gram of characters in common English words could predict spelling rules like "q is always followed by u"; an n-gram of words from a corpus of French or German text could predict the gender of nouns according to their preceding definite article.

For small values of *n*, n-grams are named according to latin numerical prefixes: *unigrams* are n-grams where $n = 1$, *bigrams* for $n = 2$, and *trigrams* for $n = 3$. Beyond that, the convention changes to English cardinal numbers: four-grams, five-grams, and so on.

We can model an n-gram in Swift by defining a custom Sequence type initialized with a base sequence, making an iterator from that sequence for each element in the tuple and successively offsetting each one according to its position.[7]

```
struct BigramSequence<Element>: Sequence, IteratorProtocol {
    private var i1: AnyIterator<Element>
    private var i2: AnyIterator<Element>

    init<S: Sequence>(_ sequence: S) where S.Element == Element
{
        self.i1 = AnyIterator<Element>(sequence.makeIterator())

        self.i2 = AnyIterator<Element>(sequence.makeIterator())
        _ = self.i2.next()
    }

    mutating func next() -> (Element, Element)? {
        guard let first = self.i1.next(),
            let second = self.i2.next()
        else {
            return nil
        }

        return (first, second)
    }
}
```

Following the convention of other sequences in the standard library, we then define a top-level `bigrams(_:)` function that returns a `BigramSequence` constructed from the specified sequence.

```
func bigrams<S>(_ sequence: S) -> BigramSequence<S.Element>
        where S: Sequence
{
    return BigramSequence(sequence)
}
```

7. You could also construct a bigram from the `Zip2Sequence` of a base sequence and its `DropFirstSequence`. (that is, `zip(sequence, sequence.dropFirst())`). However, this approach doesn't scale for n > 2.

Using `NLTokenizer`, we can tokenize a sample of natural language text into words, and pass that collection into the `bigrams(_:)` function:

```
let string = """
Please direct your attention to flight attendants
as we review the safety features of this aircraft.
"""

let tokenizer = NLTokenizer(unit: .word)
tokenizer.string = string

let words = tokenizer.tokens(for:
string.startIndex..<string.endIndex)
                     .map { String(string[$0]) }

for bigram in bigrams(words) {
    print(bigram)
}
```

The resulting sequence consists of tuples of size two with the current word as the first element and the word that follows as the second element.

Please	direct
direct	your
your	attention
...	...
features	of
of	this
this	aircraft

Generated n-grams aren't particularly useful on their own, but they have useful applications when combined with other techniques, such as *stochastic*, or random, processes.

Markov Chains

N-gram models satisfy the *Markov property*, meaning that the probability of the next state depends only on the present state, not

the sequence of events that preceded it. If we define discrete start and end states, we can perform a random walk to generate a new sequence. This is called a *Markov chain*.[8]

When used to randomly generate sentences from a text corpus, Markov chains can produce surprising — and often hilarious — results.

As an example, let's use a corpus of transcribed communications from Air Traffic Control.[9] Here's a sample entry:

```
AMERICAN ONE SIXTY SIX TURN RIGHT HEADING ONE NINER ZERO
```

In this message, Air Traffic Control (tower) is telling the pilot of American Airlines flight 166, flying from San Francisco into New York City, to turn right to 190 degrees (10 degrees west of south).

This data set is a great fit for constructing Markov chains. It has a large number of samples, but manages to keep its vocabulary relatively small (case normalization helps in this regard). This increases the likelihood of a given state having multiple possible transitions (and not simply regurgitating source material verbatim).

8. A. A. Markov. (1906) "Extension of the law of large numbers to quantities that depend on each other." Proceedings of the Physics and Mathematics Society at the University of Kazan, 2nd series, vol. 15, p. 135–156.

9. Godfrey, John J. (1997) Air Traffic Control Complete. https://www.ldc.upenn.edu/Catalog/LDC94S14A.html, accessed October 2018.

We can create a `MarkovChain` type that, similar to the `Bigram Sequence` before, adopts `Sequence` and is constructed from a base sequence. Here, the base sequence is an array of sentences where each sentence is an array of words. To encode start and stop words, each sentence is padded with `nil` at the beginning and end.

```
struct MarkovChain: Sequence, IteratorProtocol {
    private let transitions: [String?: [String?]]

    init(_ sentences: [[String]]) {
        var transitions: [Element?: [Element?]] = [:]
        for sentence in sentences {
            let paddedSequence = [nil] + Array(sequence) + [nil]
            for states in bigrams(paddedSequence) {
                transitions[states.0, default: []] += [states.1]
            }
        }
        self.transitions = transitions
    }
```

The implementation of the required `next()` method couldn't be simpler: to get the next word, randomly sample the transitions from the current word. Each of the transitions from `nil` is a start word; a transition to `nil` indicates an end word.[10]

```
    private var state: String?

    mutating func next() -> String? {
        self.state = transitions[state]?.randomElement() ?? nil

        return self.state
    }
```

10. Returning `nil` from `next()` tells the sequence to stop iterating, but this doesn't necessarily indicate that the sequence is fully consumed.

To try out our implementation, load the corpus from a text file into a `String`, and use that to construct a `MarkovChain` that is iterated over in a `for-in` loop.

```
let url = Bundle.main.url(forResource: "LDC94S14A-sample",
                          withExtension: "txt")!
let text = try String(contentsOf: url)
let markovChain = MarkovChain(text)

for word in markovChain {
    print(word, terminator: " ")
}
```

Here are some examples of generated output:

```
DELTA ONE EIGHT PROCEED INBOUND TWO TWO TWO ONE
LONER CROSS LONER THREE SIX BOSTON
APPROACH SPEED TO ONE NINER POINT
ONE NINER POINT ONE SEVEN SEVENTY
TWO SEVEN SIXTY FIVE ROGER
```

Each of these is an original construction not found in the source material. But to the untrained eye, these make as much sense as any of the real transcriptions.

You may have noticed that many of the algorithms we've discussed in this chapter aren't specific to natural language text at all.

Perhaps we tipped our hand in the formulation of `Bigram`, which makes no mention of `String`, `Character`, or the like.

```
struct BigramSequence<Element>: Sequence, IteratorProtocol {
... }
```

n-grams work equally well for modeling the likelihood of the next letter as for predicting the next gene in a sequence of DNA.

Markov models likewise have a variety of applications, so there's no reason why `MarkovChain` couldn't be reformulated like so:

```
struct MarkovChain<Element>: Sequence, IteratorProtocol
        where Element: Hashable
{
    init<S>(_ sequences: S) where S: Sequence,
                                  S.Element: Sequence,
                                  S.Element.Element == Element
```

Beyond generating superficially real-looking text, including automatic speech recognition systems, statistical analysis, and algorithmic music composition.

At this point, it should be no surprise, then, that edit distance isn't specific to strings; the only requirement for determining edit distance is that two sequences have elements that can be compared to one another for equality.

By redefining `levenshteinDistance` to be generic, we could use the same function to determine the edit distance between, for example, flight segments in an itinerary:

```
func levenshteinDistance<S1: Sequence,
                         S2: Sequence>
    (from sourceSequence: S1,
     to targetSequence: S2) -> Int
    where S1.Element: Equatable, S1.Element == S2.Element { ...
}

levenshteinDistance(from: ["SFO", "LAX", "JFK", "LHR"],
                    to:   ["SFO", "DEN", "LHR"]) // 2
```

The grand irony of natural language processing in recent decades is how far it was able to progress in recent decades by treating text like any other kind of information.

This sentiment is captured perfectly by one of the pioneering researches in NLP and information theory, Fred Jelinek:

> *Every time I fire a linguist, the performance of our speech recognition system goes up.*
>
> — Fred Jelinek

Say the same word over and over again and it eventually loses meaning.

Write the same character repeatedly and the letterform starts to break down into individual strokes.

Think about how strings are encoded for too long and you risk unraveling the very fabric of your reality. Because just as words comprise arbitrary sequences of phonemes or graphemes, so too is text encoded by schemes born of a series of historical accidents, mistakes, and moments of brilliance.

Recap

- Natural language processing techniques allow developers to analyze, process, and understand text written by or for humans.
- The Natural Language framework provides sophisticated tools for tokenizing, tagging, and classifying text.
- Tokenization is the process of splitting text into paragraphs, sentences, and words.
- Tagging is the process of assigning information to tokens, such as part of speech, name type, and lemma.
- Use CreateML and the Natural Language framework to easily build classifiers based on a corpus of tagged text samples.
- Phonetic string matching algorithms like Soundex provide a simple mechanism for normalizing names and other words based on their likely pronunciation.
- String metrics like Levenshtein distance are functions that calculate the similarity between two different sequences.
- N-grams can be used to understand the co-occurrence of words; these can be fed into a Markov chain to generate new samples that resemble the original text.