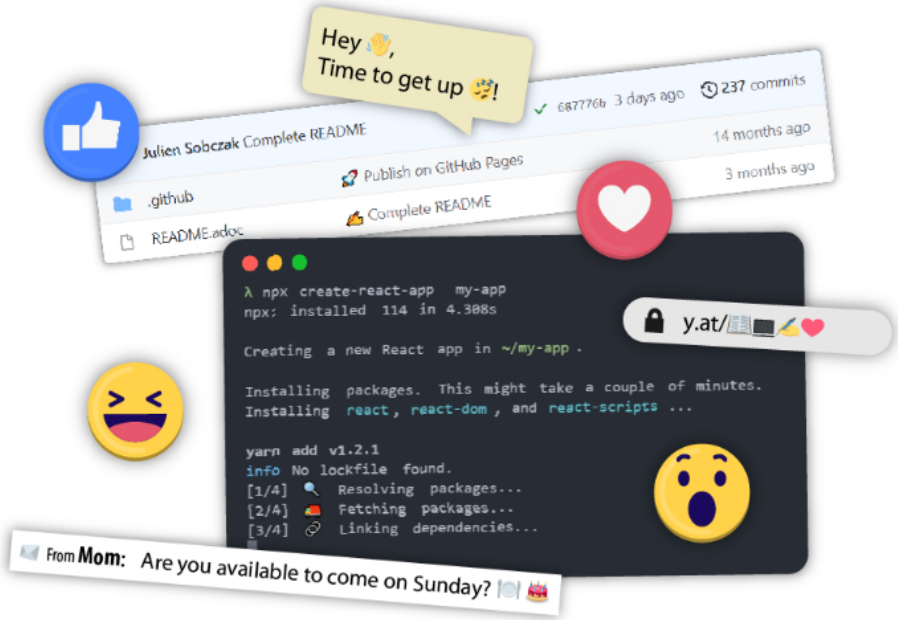
JUNE 19, 2021 EN

UNICODE FOR CURIOUS DEVELOPERS LOVING CODE 😊

THE "MINIMUM" EVERY DEVELOPER MUST KNOW ABOUT UNICODE

By Julien Sobczak

> *Our goal is to make sure that all of the text on computers for every language in the world is represented but we get a lot more attention for emojis than for the fact that you can type Chinese on your phone and have it work with another phone.*
>
> *— Unicode Consortium co-founder and president Mark Davis*



Unicode is what allows me to say hello in over 100 languages—እው ሰላም ነው (Amharic), 你好 (Chinese), γεια (Greek), שלום (Hebrew), नमस्ते (Hindi), こんにちは (Japanese), سلام (Persian), cześć (Polish), Привет (Russian), and Unicode is what allows you to understand the difference between "You are funny 🤣" and "You are funny 🖕." The web would not be the same without Unicode. But Unicode is not magic, and developers have to

understand it to create applications running in a multilingual world.

What You Will Learn

- **The Story**
  - Why Unicode makes the web possible.
  - Why emojis exist in Unicode.
  - How you can suggest a new emoji.
- **The Standard**
  - Why Unicode is not just a character set.
  - How characters are allocated in the codespace.
  - Why UTF-8 is different from Unicode, and why Unicode is different from UTF-8.
- **The Implementation**
  - Why "à" == "à" is false in most languages.
  - Why the pistol emoji is now a toy.
  - How emojis are implemented on your device.

Why Learning About Unicode?

During my career, I came across the famous Joel Spoky's blog post about Unicode and character encodings several times. The truth is, I still didn't understand what was really Unicode.

As often, if you want to have a good overview of a topic, you have to understand the details. Details create the Big Picture. The reverse is not true. Therefore, I read the Unicode Standard, browsed many articles, and inspected the code of some programming languages and libraries to understand how Unicode works. The result is this article. I hope it will help you better understand the challenges of Unicode, and why developers need to learn more about it.

TABLE OF CONTENTS

- The Story
  - The Rise of Characters
  - The Rise of Computers
  - The Rise of Unicode
  - The Rise of Emojis
- The Standard
  - The Unicode Standard
  - The Unicode Character Table

    - The Unicode Character Database
    - The Unicode Encodings
        - UTF-32
        - UTF-16
        - UTF-8
- The Implementation
    - Reading
    - Processing
    - Writing
    - Rendering
- The Future

THE STORY

THE RISE OF CHARACTERS

The story begins well after the origin of speech in the oldest known cave painting, located in France within Chauvet Cave. These paintings, dated to around 30,000 BC, were mainly symbols, and even if a picture is worth a thousand words, we cannot consider these *proto-writing* systems to be expressive enough to be considered like *writing systems*. Drawing is not writing.

We have to move forward to the beginning of the Bronze Age (around 3000 BC) to discover the earliest writing systems, in particular, the famous Egyptian hieroglyphs. They were still symbols, but once their significance was revealed, the meaning of hieroglyphs was unequivocal. **A writing system represents communication visually, using a shared understanding between writers and readers of the meaning behind the sets of characters that make up a script.**

Concerning hieroglyphs, this shared understanding was completely lost during the medieval period. The breakthrough in decipherment finally came when Napoleon's troops discovered the Rosetta Stone in 1799. The stone refuted the assumption that hieroglyphs recorded ideas and not the sounds of the language. Hieroglyphs are not drawing but true characters. This idea will serve as the main inspiration for the first alphabets, including the Latin alphabet I am currently using on my computer.

THE RISE OF COMPUTERS

Computers are not as convenient as papyrus for hieroglyphs. Computers deal with numbers composed of 0 and 1. They store letters and other characters by assigning a number to each of them, a process called *character encoding*, just like the Morse code uses signal durations to encode characters.

The earlier character encodings were limited and did not cover characters for all the world's languages. They even didn't cover all the symbols in common use in English… The fact is there are so many writing systems that classifying them is already a challenge. Chinese uses

roughly 50,000 characters representing meanings (日 for "sun"), Japanese uses approximately 100 characters representing whole syllables (た for "ta"), and Greek uses 34 characters representing sounds (α and β for "alpha" and "beta", the two first letters which serve as the etymology of the word "alphabet"). In addition, if we consider the limited space of the first computers, we can understand why **the main problem was not how to support all languages but how to save bytes**.[1]

ASCII was the first widely used encoding and used only seven bytes, enough to represent 128 unique characters ($2^7 = 128$):

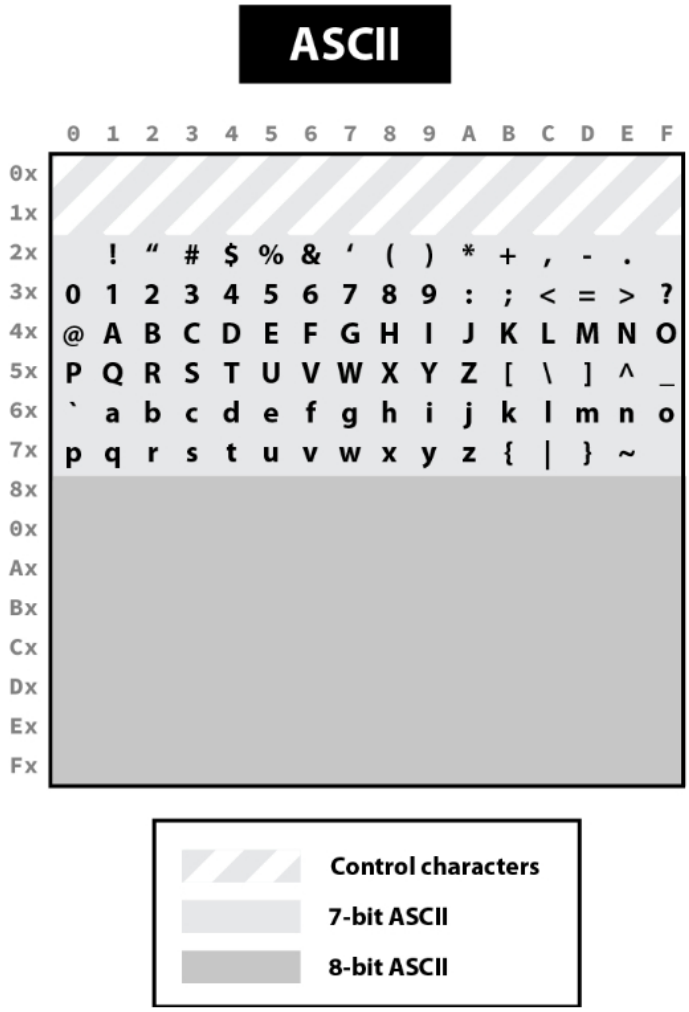


Figure 1. ASCII character set

Note that ASCII reserves the first 32 codes for non-printable control characters. For example, the character 10 represents a line feed, causing a *printer* to advance its paper to the next line and the character 8 is emitted by the *terminal* when pressing Ctrl+D to represent the "end of file."

Gradually people wanted more characters and new character encodings started to use the other 128 values available when taking all 8 bits of a byte.[2] The most famous were ISO 8859-1 and Windows 1252, the latter being a superset of the former.

ASCII 8-bit Limitation

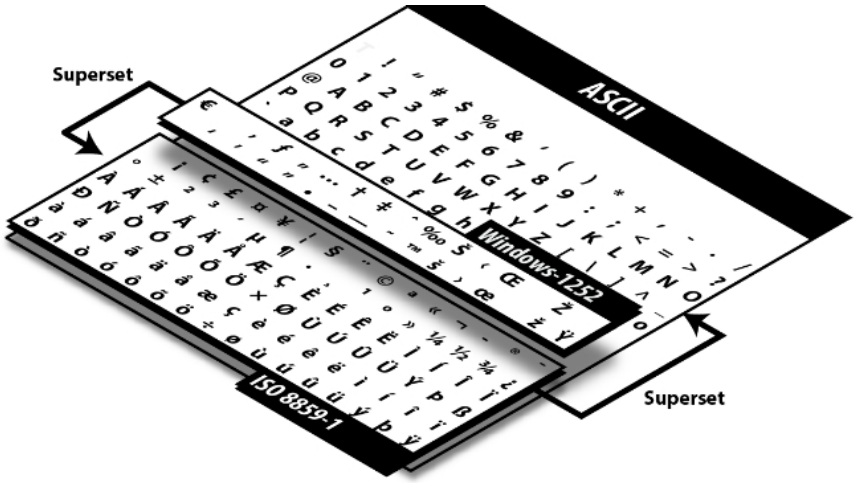


Figure 2. Common character sets extending ASCII

These encodings are just two examples among so many others.

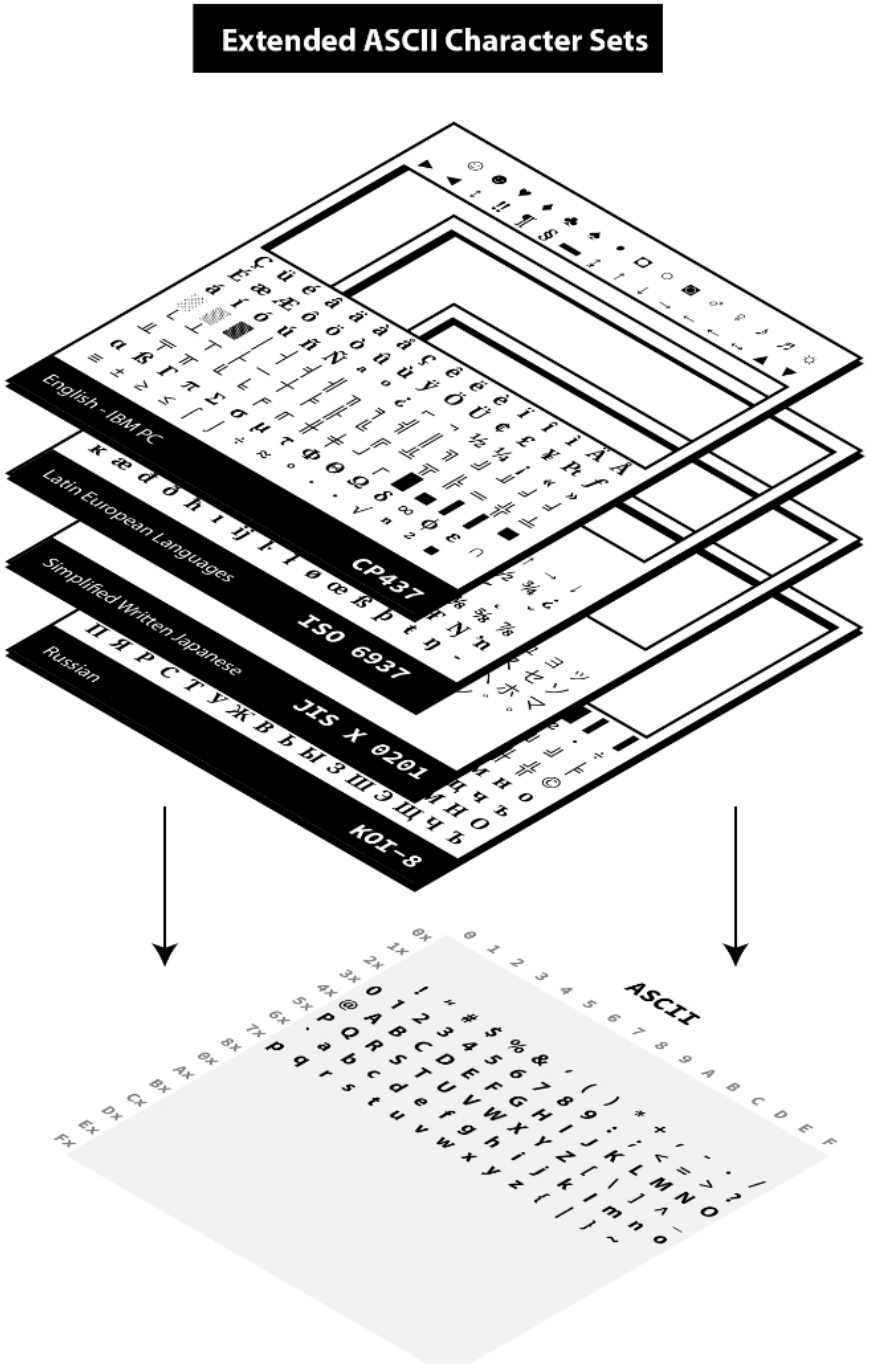


Figure 3. Examples of incompatible character sets extending ASCII

In practice, whenever textual data was exchanged between different programs or computers, the risk of corruption was high as different character encodings often use different codes for the same character. Moreover, computers did not support all character encodings, and many languages lacked character support altogether. Clearly, **256 unique codes were nowhere near enough**, especially to create multilingual documents.

THE RISE OF UNICODE

The origins of Unicode dates back to 1987, but the name Unicode first appeared the next year in the document *Unicode 88*. **The intent of Unicode was to create "a *uni*que, *uni*fied, *uni*versal enc*od*ing," so that computers would only have to implement a single encoding to support all languages**.

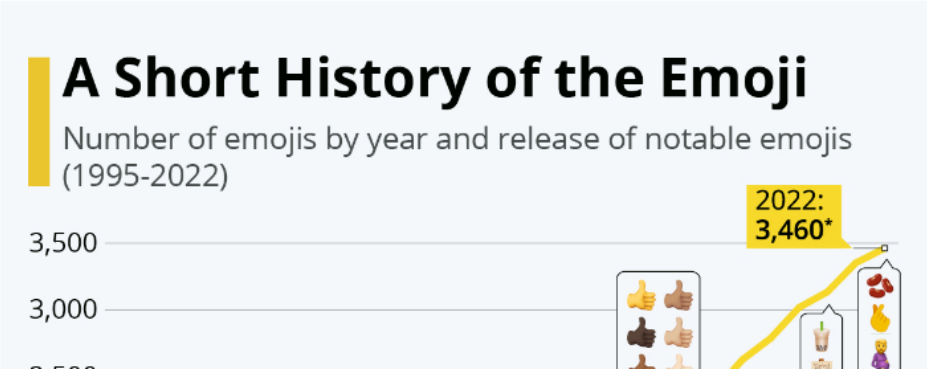
Unicode began with a 16-bit design (i.e., 65,536 codes "to encompass the characters of all the world's living languages"[3]), but was extended in 1996 to support more than a million code points. The motivation was to allow the encoding of many historic scripts (e.g., the Egyptian hieroglyphs) and thousands of rarely used or obsolete characters that had not been anticipated as needing encoding (e.g., rarely used Kanji or Chinese characters, many of which are part of personal and place names, making them rarely used, but much more essential than envisioned in the original architecture of Unicode). History is always surprising.

In the meantime, the Unicode Consortium was created in 1991. This nonprofit organization, which counts only three employees, still has the same ambitious goal of replacing all existing character encodings. This goal has almost become a reality. More than 95% of the Internet uses Unicode (it was just 50% a decade ago), and almost all electronic devices support Unicode too.

This organization is funded by membership fees (from $75 for an individual to $21,000 for a full-member company) and donations, but you can also support them by adopting a character for $100 (or $1000-$5000 for exclusivity).

THE RISE OF EMOJIS

The number of available code points exploded when Unicode dropped the 16-bit limitation. If 65,536 codes may seem a lot at that time, Unicode was now able to represent more than one million codes! Not all codes are in use. **Unicode is an evolving standard**. The current version Unicode 13.0.0 uses "only" 143,859 codes, including more than 2000 special characters that we call emojis.



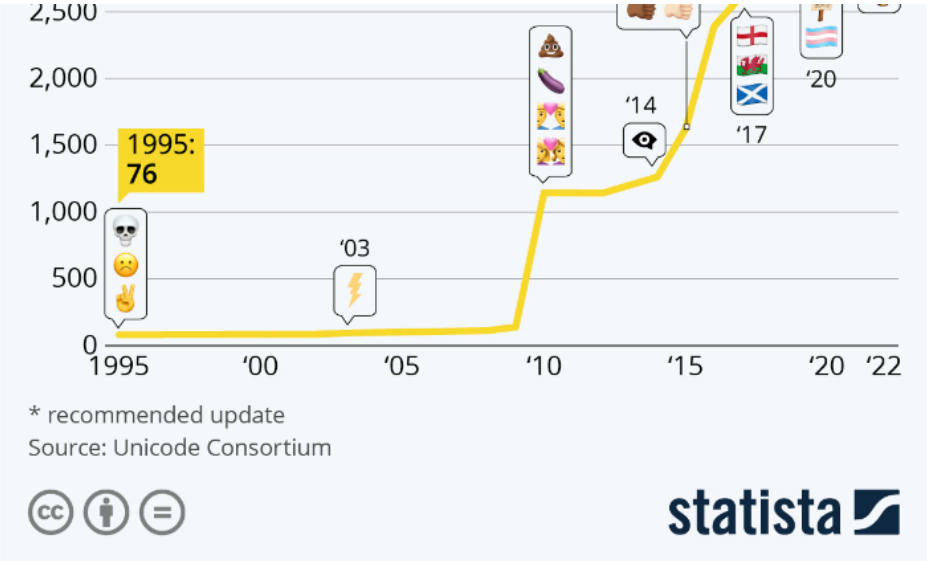


Figure 4. Emojis are probably the fatest growing language in the world (Source: Statista)

Emojis didn't appear with Unicode. They were born in the 90s in Japan (絵文字 [emodʑi] means "picture character"), and must not be confused with emoticons such as :-), which are text-based. Emojis were called *smileys* at that time. They were used mainly by mobile manufacturers and implemented using custom fonts like Wingdings:

Figure 5. Mosaic of Wingdings characters (Source: Wikipedia)

If the receiver of your message didn't have the font on his device, letters were displayed instead. For example, the national park pictogram was available in Webdings at 0x50, which corresponded to the capital letter P encoded in ASCII. To solve this problem, new character encodings were introduced to not mix characters and emojis by using different codes. Unicode was in danger.

Therefore, Google employees requested that Unicode looks into the possibility of a uniform emoji set. As a result, 722 emojis were released in 2010 as Unicode 6.0, and each new version now integrates new emojis. Unicode entered a new era.

Emoji standardization has put pressure on the Unicode Consortium, overtaking the initial focus on standardizing characters used for minority languages. But the desire for more

focus on standardizing characters used for minority languages. But the desire for more emojis has put pressure on vendors too to improve their Unicode support, leading to better support for Unicode's minority languages. A good example of a win-win situation. **Emojis contribute to preserving the writing of the past while making the writing of the future more fun**.

The primary function of emojis was to fill in emotional cues otherwise missing from typed conversations (😢🤣😊😍💁). But emojis have been extended to include a lot more (👶🤱💂🕶️🐼🍀🌞🥝⚽🚴✈️).

Now, when the Unicode Technical Committee meets quarterly to decide which new characters will be encoded, they also decide about new emojis. **Any individual or organization can suggest a new emoji by writing a proposal** (the proposal for a troll emoji is a 10-page document using Google Trends, and film references to justify its introduction). The selection process uses well-documented rules. A quorum of half of the Consortium's full members is required. There are currently ten full members, only one of which, the Ministry of Endowments and Religious Affairs of Oman, is not a tech company. The other nine are Adobe, Apple, Facebook, Google, IBM, Microsoft, Netflix, SAP, Salesforce, and a newcomer, Yat.[4]

THE STANDARD

Unicode can be used informally to refer to the *Unicode Standard*, the encoding standard for characters, and the *Unicode Consortium*, the organization that created the Unicode Standard. We have already talked about the consortium in the last part. We will now look at the Unicode Standard, and the next part will focus on its implementation in modern systems.

THE UNICODE STANDARD

The Unicode Standard is the reference document and is freely accessible online (the last edition published as a book dates back to 2006, and recent editions are available as Print-on-Demand (POD) for purchase on lulu.com). The current PDF version contains more than 1000 pages, but only the first 300 pages define Unicode and are a must-read if you want to go deeper on the subject. The remaining pages examine all supported scripts one by one.

The main contribution of the Unicode Standard is the *Unicode Character Table*. **Unicode specifies a numeric value (code point) and a name for every defined character**. In this respect, it is similar to other character encoding standards from ASCII onward. But the Unicode Standard is far more than a simple encoding of characters. **Unicode also associates a rich set of semantics with each encoded character**—properties like case and directionality required for interoperability and correct behavior in implementations. These semantics are cataloged in the *Unicode Character Database*, a collection of data files discussed at length in the last part.

The Unicode Character Table has an overall capacity for more than 1 million characters, which is more than sufficient for all the world's languages, but also for currency symbols, punctuation marks, mathematical symbols, technical symbols, geometric shapes, dingbats, and emojis. The Unicode Standard does not encode personal characters like logos but reserves 6,400 code points for private use in the Basic Multilingual Plane (BMP), the first 65,536 code points, and more than 100,000 private code points overall.

A *character* is an abstract entity, such as “latin capital letter a” or “bengali digit five,” defined by a unique code point, such as U+0041 or U+09EB. These code points are integers represented in base 16, ranging from 0 to 10FFFF (called the *codespace*). The representation of a character on screen, called a *glyph*, depends on the software implementation, based on the fonts available on your device. The Unicode Standard does not specify the precise shape, size, or orientation of on-screen characters. **Unicode characters are code points, not glyphs**. Unicode declares that U+0041 is A, U+03B1 is α, etc. Simple?

BUT.

- Unicode includes *compatibility characters*, which would not be included if they were not present in other standards.[5] The idea is to make it easy to have a one-to-one mapping between the Unicode Standard and other standards. For example, U+2163 “IV” is the roman numeral four even if we could have used two existing characters to represent it.
- Unicode includes *combining characters*, which are characters to be positioned relative to the previous base characters. Accents are implemented using combining characters, so U+0065 e + U+0302 ̂ → ê, and symbols can be composed using the same mechanism, so:

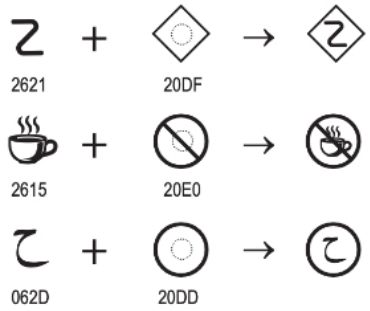


Figure 6. Combining Enclosing Marks for Symbols (From Unicode Standard, Figure 2.17)
Unicode does not restrict the number of combining characters that may follow a base character, but implementations are not obliged to support all of them.

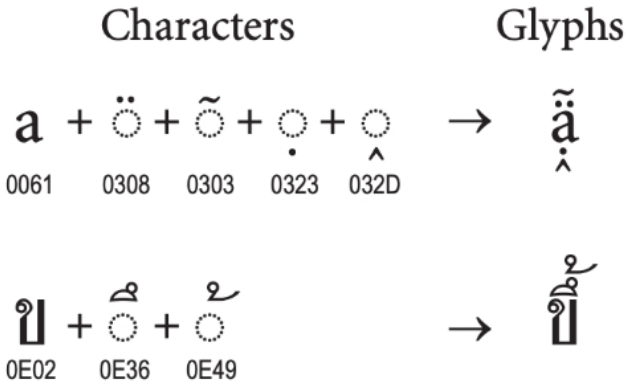
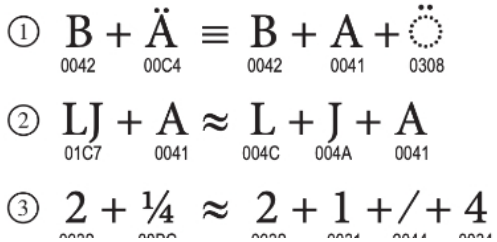


Figure 7. Stacking Sequences (From Unicode Standard, Figure 2.21)

- Unicode recognizes *equivalent sequences*, which are two sequences using different Unicode codes to represent the same character.



0032 00BC 0032 0031 2044 0034

Figure 8. Equivalent Sequences (From Unicode Standard, Figure 2.23)

- Unicode includes *special characters*. For example, UTF-16 and UTF-32 are sensible to the endianness of the hardware and often include a Byte Order Mark (BOM), which is composed of two code points—U+FEFF `zero width no-break space` and U+FFFE (a special character). Implementations can detect which byte ordering is used in a file based on the order of these two code points. Another common example is the U+FFFD `replacement character`  used to represent "unknown" characters.

We will talk more about these particularities when covering the implementation.

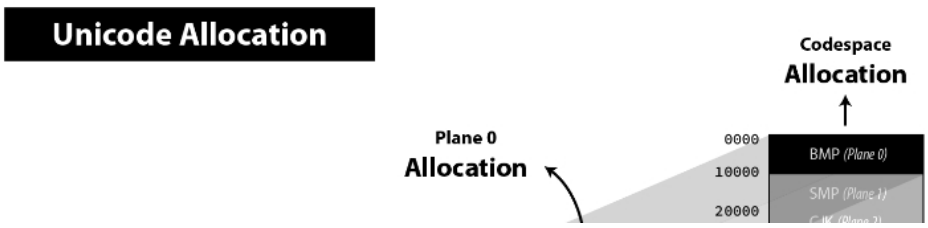
THE UNICODE CHARACTER TABLE

The Unicode Character Table has room for more than one million characters. That's a lot! Their position in this table means nothing. (There are small differences like the number of bytes to represent them in the various encodings but from a logical perspective, all characters behave similarly.) In theory, the code point of a character only needs to be unique, but in practice, its assignment follows some "conventions." Marie Kondo would probably appreciate the effort to organize the characters.

The codespace is divided up into 17 *planes* of characters—each plane consisting of 64K code points. Not all planes are currently in use.

- Plane 0: the *Basic Multilingual Plane* (BMP) is the main descendant of the first version of Unicode (with the 16-bits limitation), and the majority of frequent characters can be found here.
- Plane 1: the *Supplementary Multilingual Plane* (SMP) is the extension of the BPM for scripts or symbols with very infrequent usage. Most emojis are present in Plane 1, but as always, exceptions exist 😉.
- Plane 2: the *Supplementary Ideographic Plane* (SIP) is similar to Plane 1 but for rare CJK characters. (CJK is a collective term for the Chinese, Japanese, and Korean languages.)
- Planes 3..13 are ... empty.
- Plane 14: the *Supplementary Special-purpose Plane* (SSP) is the spillover allocation area for format control characters that do not fit into the small allocation areas for format control characters in the BMP.
- Planes 15..16: the *Private Use Planes*. These code points can be freely used for any purpose, but their use requires that the sender and receiver agree on their interpretation.

Planes are further divided into subparts called *blocks*. Character blocks generally contain characters from a single *script*, and in many cases, a script is fully represented in its block.



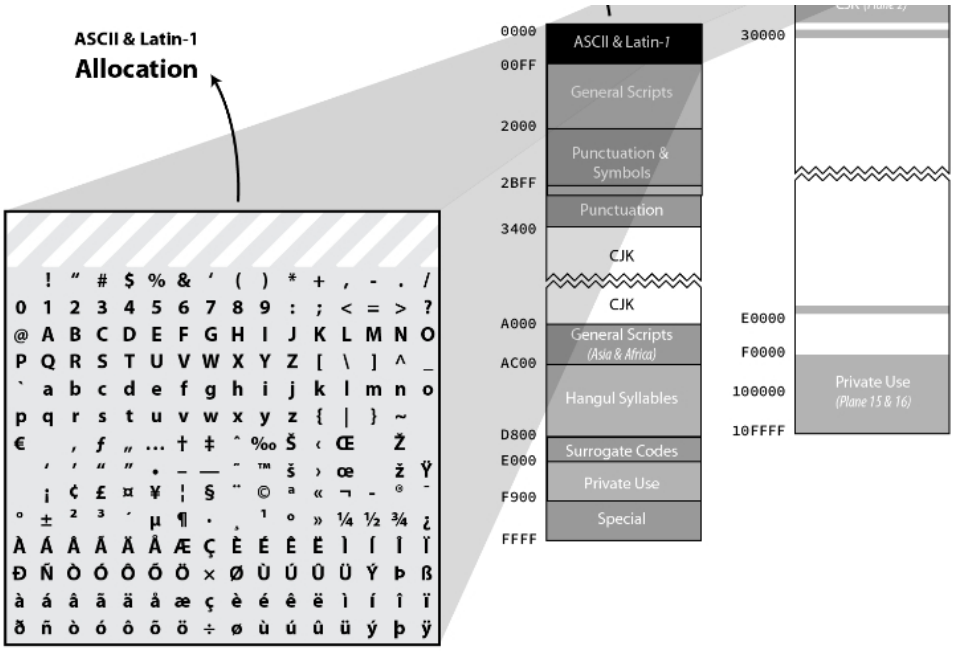


Figure 9. Unicode allocation

These blocks and scripts are also used to organize the Unicode Code Charts in the documentation so that you can quickly jump to the given script of your language. If you want the full listing instead, you can download the complete code charts in PDF (2684 pages and 110 MB! 😅).

Code Charts — Tech Site | Site Map | Searc

Unicode 13.0 Character Code Charts

SCRIPTS | SYMBOLS & PUNCTUATION | NAME INDEX

Find chart by hex code: Go Help Conventions Terms of Use

Scripts

European Scripts	African Scripts	South Asian Scripts	Indonesia & Oceania Scripts
Armenian	Adlam	Ahom	Balinese
Armenian Ligatures	Bamum	Bengali and Assamese	Batak
Carian	Bamum Supplement	Bhaiksuki	Buginese
Caucasian Albanian	Bassa Vah	Brahmi	Buhid
Cypriot Syllabary	Coptic	Chakma	Hanunoo
Cyrillic	Coptic in Greek block	Devanagari	Javanese
Cyrillic Supplement	Coptic Epact Numbers	Devanagari Extended	Makasar
Cyrillic Extended-A	Egyptian Hieroglyphs (1MB)	Dives Akuru	Rejang
Cyrillic Extended-B	Egyptian Hieroglyph Format Controls	Dogra	Sundanese
Cyrillic Extended-C	Ethiopic	Grantha	Sundanese Supplement
Elbasan	Ethiopic Supplement	Gujarati	Tagalog
Georgian	Ethiopic Extended	Gunjala Gondi	Tagbanwa
Georgian Extended	Ethiopic Extended-A	Gurmukhi	**East Asian Scripts**
Georgian Supplement	Medefaidrin	Kaithi	Bopomofo
Glagolitic	Mende Kikakui	Kannada	Bopomofo Extended
Glagolitic Supplement	Meroitic	Kharoshthi	CJK Unified Ideographs (Han) (35MB)
Gothic	Meroitic Cursive	Khojki	CJK Extension A (6MB)
Greek	Meroitic Hieroglyphs	Khudawadi	CJK Extension B (40MB)
Greek Extended	N'Ko	Lepcha	CJK Extension C (3MB)
Ancient Greek Numbers	Osmanya	Limbu	CJK Extension D
Latin	Tifinagh	Mahajani	CJK Extension E (3.5MB)
Basic Latin (ASCII)	Vai	Malayalam	CJK Extension F (4MB)
Latin-1 Supplement	**Middle Eastern Scripts**	Masaram Gondi	CJK Extension G (2MB)
Latin Extended-A	Anatolian Hieroglyphs	Meetei Mayek	(see also Unihan Database)
Latin Extended-B	Arabic	Meetei Mayek Extensions	CJK Compatibility Ideographs
Latin Extended-C	Arabic Supplement	Modi	CJK Compatibility Ideographs Supplement
Latin Extended-D	Arabic Extended-A	Mro	CJK Radicals / Kangxi Radicals
Latin Extended-E	Arabic Presentation Forms-A	Multani	CJK Radicals Supplement
Latin Extended Additional	Arabic Presentation Forms-B	Nandinagari	CJK Strokes
Latin Ligatures	Aramaic, Imperial	Newa	Ideographic Description
Fullwidth Latin Letters	Avestan	Ol Chiki	
IPA Extensions	Chorasmian	Oriya (Odia)	
Phonetic Extensions		Saurashtra	

- Phonetic Extensions Supplement
- ***Linear A***
- ***Linear B***
  - Linear B Syllabary
  - Linear B Ideograms
  - Aegean Numbers
- ***Lycian***
- ***Lydian***
- ***Ogham***
- ***Old Hungarian***
- ***Old Italic***
- ***Old Permic***
- ***Phaistos Disc***
- ***Runic***
- ***Shavian***

Modifier Letters

- ***Modifier Tone Letters***
- ***Spacing Modifier Letters***
- ***Superscripts and Subscripts***

Combining Marks

- ***Combining Diacritical Marks***
  - Combining Diacritical Marks Extended
  - Combining Diacritical Marks Supplement
- ***Combining Diacritical Marks for Symbols***
- ***Combining Half Marks***
- ***Cuneiform (1MB)***
  - Cuneiform Numbers and Punctuation
  - Early Dynastic Cuneiform
  - Old Persian
  - Ugaritic
- ***Elymaic***
- ***Hatran***
- ***Hebrew***
  - *Hebrew Presentation Forms*
- ***Mandaic***
- ***Nabataean***
- ***Old North Arabian***
- ***Old South Arabian***
- ***Pahlavi, Inscriptional***
- ***Pahlavi, Psalter***
- ***Palmyrene***
- ***Parthian, Inscriptional***
- ***Phoenician***
- ***Samaritan***
- ***Syriac***
  - *Syriac Supplement*
- ***Yezidi***

Central Asian Scripts

- ***Manichaean***
- ***Marchen***
- ***Mongolian***
  - Mongolian Supplement
- ***Old Sogdian***
- ***Old Turkic***
- ***Phags-Pa***
- ***Sogdian***
- ***Soyombo***
- ***Tibetan***
- ***Zanabazar Square***
- ***Sharada***
- ***Siddham***
- ***Sinhala***
  - Sinhala Archaic Numbers
- ***Sora Sompeng***
- ***Syloti Nagri***
- ***Takri***
- ***Tamil***
  - Tamil Supplement
- ***Telugu***
- ***Thaana***
- ***Tirhuta***
- ***Vedic Extensions***
- ***Wancho***
- ***Warang Citi***

Southeast Asian Scripts

- ***Cham***
- ***Hanifi Rohingya***
- ***Kayah Li***
- ***Khmer***
  - Khmer Symbols
- ***Lao***
- ***Myanmar***
  - Myanmar Extended-A
  - Myanmar Extended-B
- ***New Tai Lue***
- ***Nyiakeng Puachue Hmong***
- ***Pahawh Hmong***
- ***Pau Cin Hau***
- ***Tai Le***
- ***Tai Tham***
- ***Tai Viet***
- ***Thai***
- Characters
- ***Hangul Jamo***
  - Hangul Jamo Extended-A
  - Hangul Jamo Extended-B
  - Hangul Compatibility Jamo
  - *Halfwidth Jamo*
- ***Hangul Syllables***
- ***Hiragana***
- ***Kana Extended-A***
- ***Kana Supplement***
- ***Small Kana Extension***
- ***Kanbun***
- ***Katakana***
  - Katakana Phonetic Extensions
  - *Halfwidth Katakana*
- ***Khitan Small Script***
- ***Lisu***
  - Lisu Supplement
- ***Miao***
- ***Nushu***
- ***Tangut***
  - Tangut Components
  - Tangut Supplement
- ***Yi***
  - Yi Syllables
  - Yi Radicals

American Scripts

- ***Cherokee***
  - Cherokee Supplement
- ***Deseret***
- ***Osage***
- ***Unified Canadian Aboriginal Syllabics***
  - UCAS Extended

Other

- Alphabetic Presentation Forms
- *ASCII Characters*
- Halfwidth and Fullwidth Forms

Symbols and Punctuation

Notational Systems

- ***Braille Patterns***
- ***Musical Symbols***
  - Ancient Greek Musical Notation
  - Byzantine Musical Symbols
- ***Duployan***
  - Shorthand Format Controls
- ***Sutton SignWriting***

Punctuation

- ***General Punctuation***
  - *ASCII Punctuation*
  - *Latin-1 Punctuation*
  - Supplemental Punctuation
- ***CJK Symbols and Punctuation***
  - Ideographic Symbols and Punctuation
- ***CJK Compatibility Forms***
  - Halfwidth and Fullwidth Forms
  - Small Form Variants
  - Vertical Forms

Alphanumeric Symbols

- ***Letterlike Symbols***
  - *Roman Symbols*
- ***Mathematical Alphanumeric Symbols***
- ***Arabic Mathematical Alphabetic Symbols***
- ***Enclosed Alphanumerics***
  - Enclosed Alphanumeric Supplement
- ***Enclosed CJK Letters and Months***
  - Enclosed Ideographic Supplement
- ***CJK Compatibility***
  - *Additional Squared Symbols*

Technical Symbols

- *APL symbols*
- Control Pictures
- Miscellaneous Technical
- Optical Character Recognition (OCR)

Numbers & Digits

- *(see also specific scripts)*
- ***ASCII Digits***
  - *Fullwidth ASCII Digits*
- ***Common Indic Number Forms***
- ***Coptic Epact Numbers***
- ***Counting Rod Numerals***
- ***Cuneiform Numbers and Punctuation***
- ***Indic Siyaq Numbers***
- ***Mayan Numerals***
- ***Number Forms***
- ***Ottoman Siyaq Numbers***
- ***Rumi Numeral Symbols***
- ***Sinhala Archaic Numbers***
- ***Super and Subscripts***

Mathematical Symbols

- ***Arrows***
  - Supplemental Arrows-A
  - Supplemental Arrows-B
  - Supplemental Arrows-C
  - *Additional Arrows*
  - Miscellaneous Symbols and Arrows
- ***Mathematical Alphanumeric Symbols***
  - Arabic Mathematical Alphabetic Symbols
  - Letterlike Symbols
- ***Mathematical Operators***
  - *Basic operators: Plus, Factorial, Division, Multiplication*
  - Supplemental Mathematical Operators
  - Miscellaneous Mathematical Symbols-A
  - Miscellaneous Mathematical Symbols-B
  - *Floors and Ceilings*
  - *Invisible Operators*
- ***Geometric Shapes***
  - *Additional Shapes*

Emoji & Pictographs

- ***Dingbats***
  - Ornamental Dingbats
- ***Emoticons***
- ***Miscellaneous Symbols***
- ***Miscellaneous Symbols And Pictographs***
- ***Supplemental Symbols and Pictographs***
- ***Symbols and Pictographs Extended-A***
- ***Transport and Map Symbols***

Other Symbols

- ***Alchemical Symbols***
- ***Ancient Symbols***
- ***Currency Symbols***
  - (see also specific scripts)
  - *Dollar Sign, Euro Sign*
  - *Yen, Pound and Cent*
  - *Fullwidth Currency Symbols*
  - *Rial Sign*
- ***Game Symbols***
  - *Chess, Checkers/Draughts*
  - Chess Symbols
  - Domino Tiles
  - *Japanese Chess*
  - Mahjong Tiles
  - Playing Cards
  - *Card suits*
- ***Miscellaneous Symbols and Arrows***
- ***Symbols for Legacy Computing***
- ***Yijing Symbols***
  - *Yijing Mono-, Di- and Trigrams*
  - Yijing Hexagram Symbols
  - Tai Xuan Jing Symbols

Specials

- *Controls: C0, C1*
- *Layout Controls*
- *Invisible Operators*
- ***Specials***
- ***Tags***
- ***Variation Selectors***
  - *Variation Selectors Supplement*

Private Use

- ***Private Use Area***
- ***Supplementary Private Use Area-A***
- ***Supplementary Private Use Area-B***

Surrogates

- ***High Surrogates***
- ***Low Surrogates***

Noncharacters in Charts

Noncharacters in blocks

- *Range in Arabic Presentation Forms-A*
- *Range in Specials*

Noncharacters at end of ...

- *BMP, Plane 1, Plane 2, Plane 3, Plane 4, Plane 5, Plane 6, Plane 7, Plane 8, Plane 9, Plane 10, Plane 11, Plane 12, Plane 13, Plane 14, Plane 15, Plane 16*

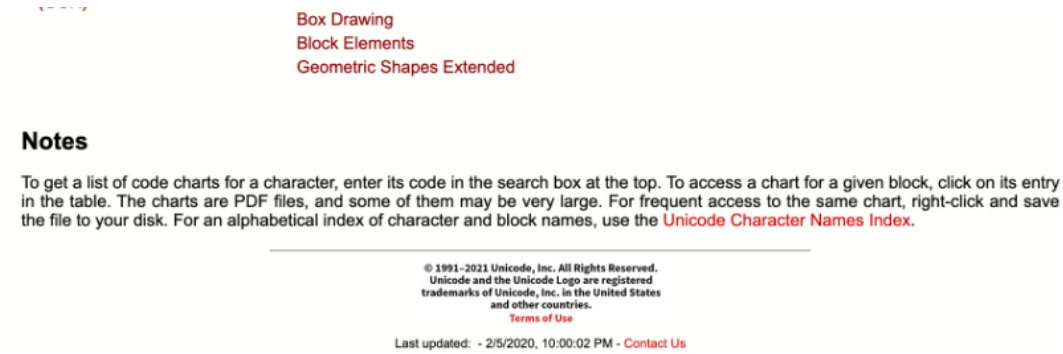


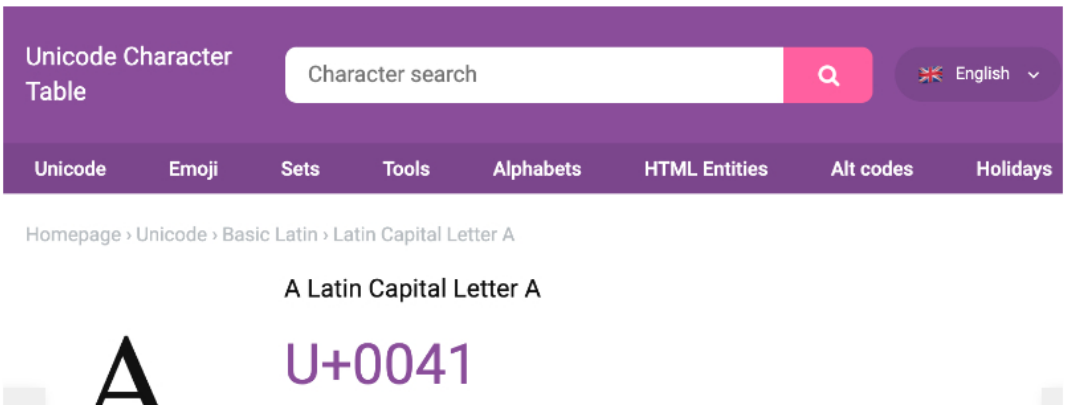
Figure 10. Unicode Character Code Charts

THE UNICODE CHARACTER DATABASE

The Unicode Character Database (UCD) is a set of documentation and data files accessible online. These files contain more than 100 character properties, including:

- A name
  → Useful to refer to a character using a unique identifier instead of a hexadecimal value, like using the name `tab` instead of U+0009.
- The general category (basic partition into letters, numbers, symbols, punctuation, and so on).
  → Useful to determine the primary use (letter, digit, punctuation, symbol) when implementing functions like `isDigit()`.
- Some general characteristics (whitespace, dash, ideographic, alphabetic, noncharacter, deprecated, and so on)
  → Useful to determine the kind of character like digits.
- Some display-related properties (bidirectional class, shaping, mirroring, width, and so on)
  → Useful when rendering the text on screen.
- The case (upper, lower, title, folding—both simple and full)
  → Useful to determine if a character is uppercase.
- The script and block a character belongs.
  → Useful to find characters commonly used together.
- and a lot more!

You can visualize these properties from many websites like unicode-table.com:



Click to copy and paste symbol

A Copy

Technical information

Name	Latin Capital Letter A
Unicode number	U+0041
HTML-code	A
CSS-code	\0041
Block	Basic Latin
Lowercase	a
Unicode version:	1.1 (1993)
Alt code:	Alt 65

Symbol meaning

Latin Capital Letter A. Basic Latin.

Latin Capital Letter A was approved as part of Unicode 1.1 in 1993.

Properties

Age	1.1
Block	Basic Latin
Bidi Paired Bracket Type	None
Composition Exclusion	No
Case Folding	0061
Simple Case Folding	0061

Encoding

Encoding	hex	dec (bytes)	dec	binary
UTF-8	41	65	65	01000001
UTF-16BE	00 41	0 65	65	00000000 01000001
UTF-16LE	41 00	65 0	16640	01000001 00000000
UTF-32BE	00 00 00 41	0 0 0 65	65	00000000 00000000 00000000 01000001
UTF-32LE	41 00 00 00	65 0 0 0	1090519040	01000001 00000000 00000000 00000000

These websites are based on the files available in the UCD. We will present the main ones in this article. These files follow a few conventions: each line consists of fields separated by semicolons, the first field represents a code point or range expressed as hexadecimal numbers. The remaining fields are properties associated with that code point. A code point may be omitted in a data file if the default value for the property in question applies.

`UnicodeData.txt` is the main file:

UnicodeData.txt

```
...
0009;<control>;Cc;0;S;;;;;N;CHARACTER TABULATION;;;;
…
0021;EXCLAMATION MARK;Po;0;ON;;;;;N;;;;;
…
0041;LATIN CAPITAL LETTER A;Lu;0;L;;;;;N;;;;0061;
...
```

Where (see full details):

- `0041` is the code point (U+0041).
- `LATIN CAPITAL LETTER A` is the property `NAME`.
- `Lu` is the abbreviation for the value `Uppercase_Letter` of the property `General_Category`.
- `L` is the abbreviation for the value `Left_To_Right` of the property `Bidi_Class` to indicate a left-to-right character. `ON` stands for `Other_Neutral` and is used by most punctuation characters.
- `0061` is the code point value of the property `Simple_Lowercase_Mapping` which means the lowercase character for U+0041 `A` is U+0061 `a`.

`emoji-data.txt` is the main file concerning emojis:

emoji/emoji-data.txt

```
...
1F600         ; Emoji                # E1.0   [1] (😀)       grinning face
1F601..1F606  ; Emoji                # E0.6   [6] (😁..😆)    beaming face with smiling eyes..grinni
1F607..1F608  ; Emoji                # E1.0   [2] (😇..😈)    smiling face with halo..smiling face w
1F609..1F60D  ; Emoji                # E0.6   [5] (😉..😍)    winking face..smiling face with heart-
1F60E         ; Emoji                # E1.0   [1] (😎)       smiling face with sunglasses
1F60F         ; Emoji                # E0.6   [1] (😏)       smirking face
1F610         ; Emoji                # E0.7   [1] (😐)       neutral face
1F611         ; Emoji                # E1.0   [1] (😑)       expressionless face
...
```

Where (see full details):

- `Emoji` is the default type. Other possible values are `Emoji_Modifier` for the skin tone modifier, `Emoji_Modifier_Base` for characters that can serve as a base for emoji modifiers. `Emoji_Component` for characters used in emoji sequences like flags.
- The comment indicates the first version that introduced the emoji(s), the count of emojis in the range, a preview of the emoji(s), and their name(s).

THE UNICODE ENCODINGS

Character encodings are necessary when exchanging or storing texts. Computers only understand 0s and 1s and therefore, Unicode code points must be converted into binary.

The Unicode Standard provides three distinct encoding forms for Unicode characters, using minimum 8-bit, 16-bit, and 32-bit units. These are named UTF-8, UTF-16, and UTF-32, respectively. **All three encoding forms can be used to represent the full range of Unicode characters and each one can be efficiently transformed into either of the other two without any loss of data**.

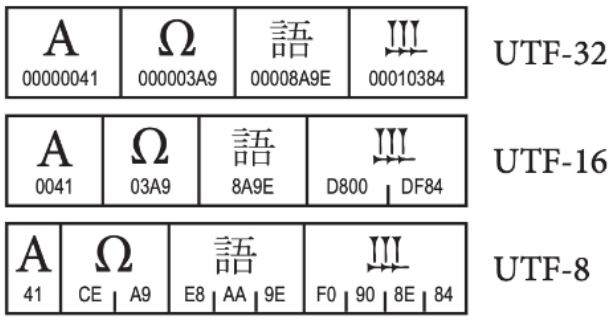


Figure 11. Unicode Encoding Forms (From Unicode Standard, Figure 2.11)

The Principle of Nonoverlapping

Unicode encodings differ from many prior encodings that also use varied-length bytes but where overlap was permitted. For example:

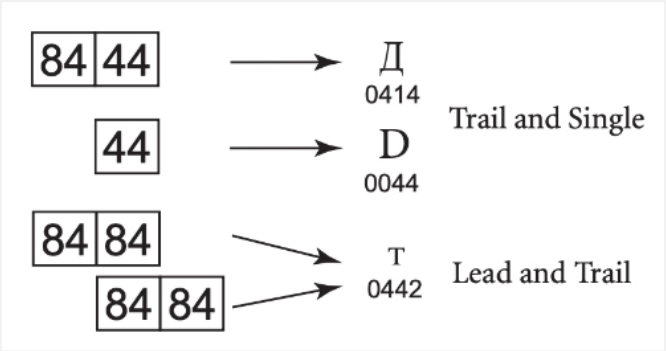


Figure 12. Overlap in Legacy Mixed-Width Encodings (From Unicode Standard, Figure 2.9)

To determine the character, these encodings depend on the first byte. If someone searches for the character "D," for example, he might find it in the trail byte of the two-byte sequence Д. The program must look backward through text to find the correct matches, but the boundaries are not always easy to interpret with overlapping:

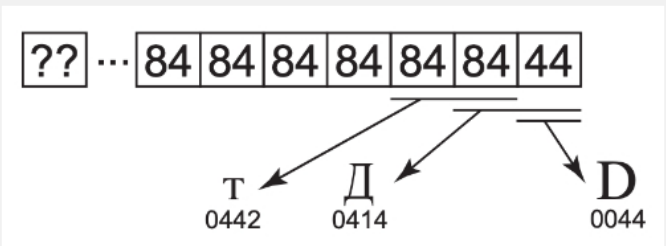


Figure 13. Boundaries and Interpretation (From Unicode Standard, Figure 2.10)

> Therefore, each of the Unicode encoding forms is designed with the principle of nonoverlapping in mind to make implementations more simple and more efficient.

UTF-32

UTF-32 is the simplest Unicode encoding form. UTF-32 is a fixed-width character encoding form. **Each Unicode code point is represented directly by a single 32-bit code unit**. Because of this, UTF-32 has a one-to-one relationship between encoded character and code unit;

Note that 32 bits have space for more than four million codes, but UTF-32 restricts the representation of code points in the standard ranges 0..10FFFF (we need to cover UTF-16 first to explain this restriction).

```go
import (
    "bytes"
    "encoding/binary"
)

func EncodeUTF32BE(codepoints []uint32) []byte {
    buf := new(bytes.Buffer)

    // BOM (optional)
    binary.Write(buf, binary.BigEndian, uint32(0xFEFF))

    for _, codepoint := range codepoints {
        // Each codepoint is written as unit32
        binary.Write(buf, binary.BigEndian, codepoint)
    }

    return buf.Bytes()
}
```

Decoding follows the inverse logic:

1. Read the next four bytes.
2. Extract the value to get the code point.
3. Repeat.

Preferred Usage: UTF-32 may be preferred where memory or disk storage space is not limited and when the simplicity of access to single code units is desired. For example, the first version of Python 3 represented strings as sequences of Unicode code points, but Python 3.3 changed the implementation to optimize the memory usage.

UTF-16

UTF-16 is the historical descendant of the earliest form of Unicode where only 16-bits were used for code points. The characters in the range U+0000..U+FFFF (the first 65,536 characters) are often called the Basic Multilingual Plane (BMP, or Plane 0), and are encoded as a single 16-bit code unit using the code point value like in UTF-32.

The remaining characters are called *surrogates* and are encoded as pairs of 16-bit code units whose values are disjunct from the code units used for the single code unit representations, thus maintaining non-overlap for all code point representations in UTF-16.

Understanding the maximum number of code points in Unicode

Not breaking already encoded texts in UCS-2 (known as Unicode at that time) was the biggest challenge to extend the initial number of Unicode characters (2^{16} = 65536 characters).

The solution (now called UTF-16) is to rely on two unused ranges 0xD800..0xDBFF and 0xDC00–0xDFFF (each one representing 1024 code points). If we concatenate these two ranges, it means we can represent 1024 * 1024 = 1,048,576 new characters, in addition to the 63488 original code points (2^{16} - 2048, the number of unique code points that fits 2 bytes minus the ranges previously unused).

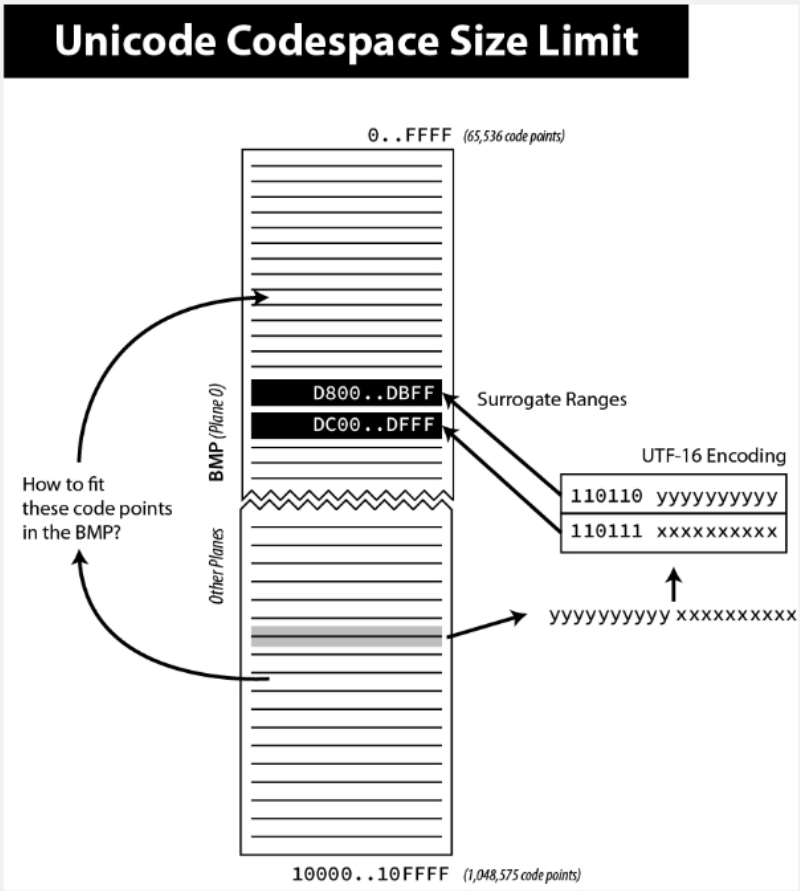


Figure 14. Unicode surrogate codes

So, in UTF-16, the representation for all initial characters does not change. But when we need to encode one of the new characters, we will not use their code points—they just cannot fit in a 16-byte word—but use instead what is called a surrogate pair, which is a pair of pointers to retrieve the original code point. If we consider the binary representation of the two previously unused ranges:

```
0xD800 = 0b1101100000000000 (110110 = high surrogate prefix)
0xDFFF = 0b1101111111111111 (110111 =  low surrogate prefix)
```

Every new code point is represented in UTF-16 by two 16-bytes words—the high surrogate

Every new code point is represented in UTF-16 by two 16-bytes words—the high surrogate followed by the low surrogate—each one uses one of the two ranges. Note that six bits are reserved for the prefix of the ranges. Therefore, if a byte starts with 110110, we know we have a high surrogate that is followed by a low surrogate starting with 110111, and inversely. It means we have 20 representative bits to represent the new characters (2^{20} = 1,048,576).

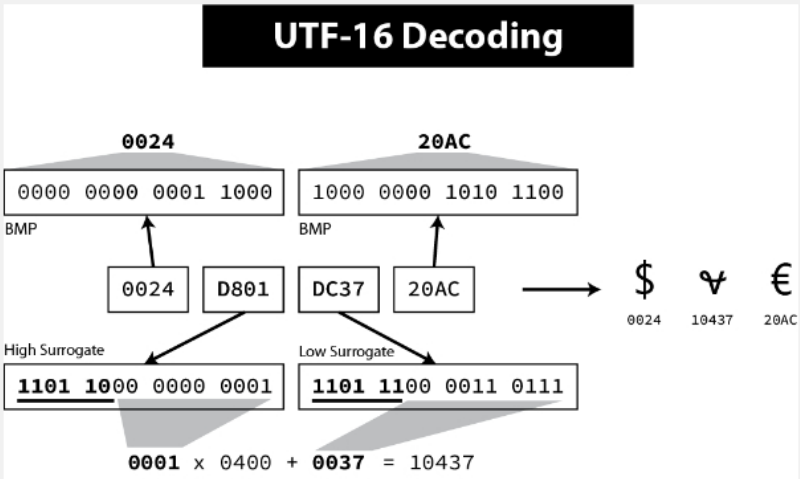


Figure 15. Decoding surrogate pairs in UTF-16

Even if other encodings like UTF-32 and UTF-8 can represent more code points, the total number of valid Unicode code points is still constrained by UTF-16 for backward compatibility reasons. The exact number is 1,111,998 possible Unicode characters, a little less than our estimation due to 2 reserved characters at the end of each plane.

```go
import (
    "bytes"
    "encoding/binary"
)

func EncodeUTF16BE(codepoints []uint32) []byte {
    // Code is inspired by Go official implementation of module unicode/utf16
    // https://github.com/golang/go/blob/go1.16/src/unicode/utf16/utf16.go

    buf := new(bytes.Buffer)

    // BOM
    binary.Write(buf, binary.BigEndian, uint16(0xFEFF))
    for _, v := range codepoints {
        switch {
        case v < 0x10000:
            // Code points in the Basic Multilingual Plane (BMP)
            // are written as such in uint16 as they can safely
            // be stored in two bytes.
            binary.Write(buf, binary.BigEndian, uint16(v))
        case 0x10000 <= v:
            // Code points in Supplementary Planes are encoded
            // as two 16-bit code units called a surrogate pair.

            // 0x10000 is subtracted from the code point,
            // leaving a 20-bit number in the hex number range 0x00000-0xFFFFF
            r := v - 0x10000

            // The high ten bits (in the range 0x000-0x3FF) are added to 0xD800
            // to give the first 16-bit code unit or high surrogate,
            // which will be in the range 0xD800-0xDBFF.
            r1 := 0xd800 + (r>>10)&0x3ff
            binary.Write(buf, binary.BigEndian, uint8(r1))
```

```
            // The low ten bits (also in the range 0x000-0x3FF) are added
            // to 0xDC00 to give the second 16-bit code unit or low surrogate,
            // which will be in the range 0xDC00-0xDFFF.
            r2 := 0xdc00 + r&0x3ff
            binary.Write(buf, binary.BigEndian, uint8(r2))
        }
    }

    return buf.Bytes()
}
```

Decoding simply needs to test for surrogates:

1. Read the next two bytes.
2. If the value is in the range U+0000..U+FFFF, this is a code point.
3. Otherwise, retrieve the value from the high surrogate (0xD800..0xDBFF) and the low surrogate (0xDC00–0xDFFF) by reading two more bytes and extract the code point using basic mathematical operations.

Preferred Usage: UTF-16 provides a balanced representation that is reasonably compact as all the common, heavily used characters fit into a single 16-bit code unit. This encoding is often used by programming languages as their internal representation of strings for that reason, but for file encoding, UTF-8 is by far the most privileged encoding.

UTF-8

UTF-8 is a variable-width like UTF-16, but offering compatibility with ASCII. That means Unicode code points U+0000..U+007F are converted to a single byte 0x00..0x7F in UTF-8 and are thus indistinguishable from ASCII itself. An ASCII document is a valid UTF-8 document (the reverse is rarely true).

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	**11**0xxxxx	10xxxxxx		
U+0800	U+FFFF	**111**0xxxx	10xxxxxx	10xxxxxx	
U+10000	U+10FFFF	**1111**0xxx	10xxxxxx	10xxxxxx	10xxxxxx

Note that when the first byte starts with 1, the number of successive 1s gives the number of bytes for this code point.

```
import (
    "bytes"
    "encoding/binary"
)
```

```
'

func EncodeUTF8(codepoints []uint32) []byte {
    // Code is inspired by Go official implementation of module unicode/utf8
    // https://github.com/golang/go/blob/go1.16/src/unicode/utf8/utf8.go

    buf := new(bytes.Buffer)

    // Note: The Unicode Standard neither requires nor recommends
    // the use of the BOM for UTF-8.

    for _, r := range codepoints {
        switch i := uint32(r); {

        // 1 byte for ASCII characters
        case int(r) <= 0x007F: // 127
            buf.WriteByte(byte(r)) // 0xxxxxxx

        // 2 bytes for most Latin scripts
        case i <= 0x07FF: // 2047
            buf.WriteByte(0b11000000 | byte(r>>6))         // 110xxxxx
            buf.WriteByte(0b10000000 | byte(r)&0b00111111) // 10xxxxxx

        // 3 bytes for the rest of the BMP
        case i <= 0xFFFF: // 65535
            buf.WriteByte(0b11100000 | byte(r>>12))           // 1110xxxx
            buf.WriteByte(0b10000000 | byte(r>>6)&0b00111111) // 10xxxxxx
            buf.WriteByte(0b10000000 | byte(r)&0b00111111)    // 10xxxxxx

        // 4 bytes for other planes and most emojis
        default:
            buf.WriteByte(0b11110000 | byte(r>>18))            // 11110xxx
            buf.WriteByte(0b10000000 | byte(r>>12)&0b00111111) // 10xxxxxx
            buf.WriteByte(0b10000000 | byte(r>>6)&0b00111111)  // 10xxxxxx
            buf.WriteByte(0b10000000 | byte(r)&0b00111111)     // 10xxxxxx
        }
    }

    return buf.Bytes()
}
```

Decoding is easy to implement.

1. Read the next byte.
2. If it starts by `0`, the character is encoded using 1 byte.
3. If it starts by `110`, the character is encoded using 2 bytes. (two leading `1`s)
4. If it starts by `1110`, the character is encoded using 3 bytes. (three leading `1`s)
5. If it starts by `11110`, the character is encoded using 4 bytes. (four leading `1`s)
6. Read the remaining bits of the first byte.
7. Read the last six bits of the other composing byte(s).
8. Reassemble using basic mathematical operations to retrieve the code unit.
9. Repeat.

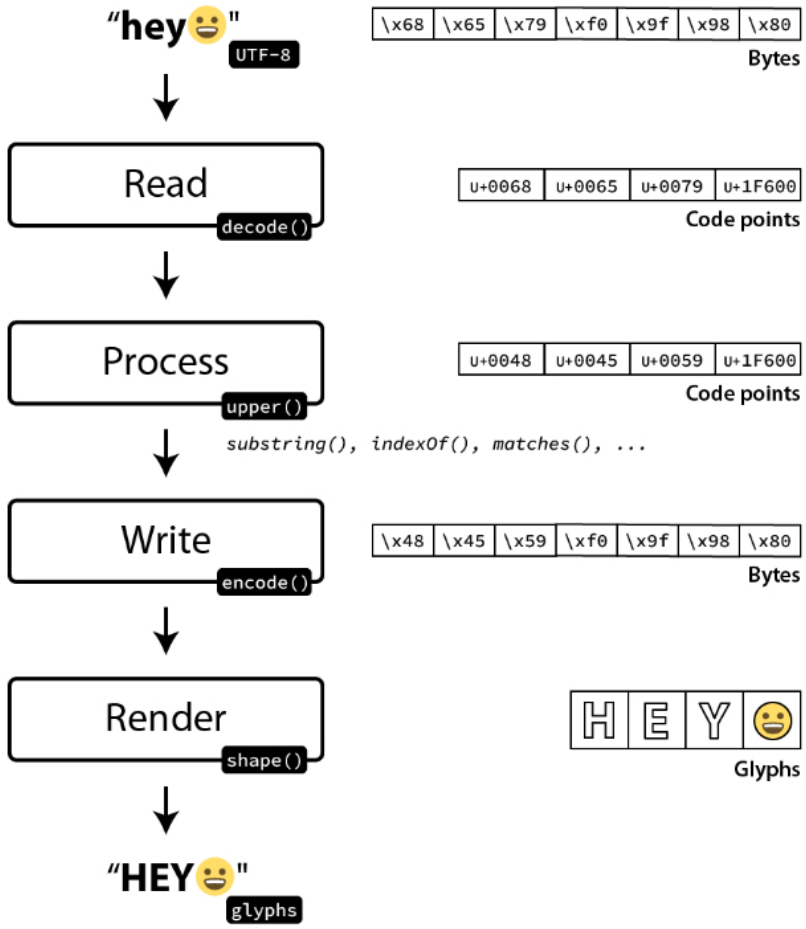
Preferred Usage: UTF-8 is particularly compact when the text contains mainly ASCII characters, which is often the case for a large percent of the population, but UTF-8 is significantly larger for Asian writing systems compared to UTF-16 as these characters require three bytes in UTF-8.

In practice, UTF-8 has become the default Unicode encoding of the Web even if all three encodings are perfectly valid.

THE IMPLEMENTATION

Now that we understand the history of Unicode and what is the scope of the Unicode Standard, we may look at the implementation. What does it mean to support Unicode? For an operating system? For a programming language? For a desktop application? For a web application?

We will split this section based on the different steps of a program. A program manipulates data that is often received and sent over networks and eventually displayed to a user. Unicode plays an essential role in every step.



READING

When a program receives or reads a text, it needs to know the encoding. Having a text without knowing its encoding is useless—just a suite of 0s and 1s. Several strategies exist to determine the encoding. For example, the source code of your program is also a text that the compiler or the interpreter must decode. The common solutions are:

- **Use the charset of the underlying operating system** (ex: Java). Unix operating system family uses UTF-8 encoding by default, which means the Java compiler expects source

files to be encoded in UTF-8 too. Otherwise, the developer needs to define the encoding explicitly from the command line (`java -Dfile.encoding=UTF-16`) or when reading a text:

```
// $LC_CTYPE returns "UTF-8"
File file = new File("/path/to/file");
BufferedReader br = new BufferedReader(new InputStreamReader(
    new FileInputStream(file), "UTF16")); // Override default encoding
...
```

- **Use a default encoding and allows the developer to override it.** (ex: Python). The Python interpreter expects UTF-8 files but supports various syntaxes to specify a different encoding.

```
#!/usr/bin/python env
# -*- coding: utf-16 -*-
```

This magic comment must be the first or second line. As the interpreter ignores the encoding when reading the file, all characters until the encoding must only be ASCII characters. This technique was already used by browsers to determine the encoding of a web page. Initially, web servers were expected to return the HTTP header `Content-Type` (ex: `text/plain; charset="UTF-8"`) so that the browser knows the encoding before reading the HTML file, but in practice, a web server can serve different files written by different persons in different languages, and thus would also need to know the encoding... The common solution was instead to include the charset directly in the document as the first meta under the `<head>` section:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
...
```

- **Force a specific encoding.** (ex: Go). Go source code can only be written in UTF-8 files.

```
$ cat > hello.go << EOF
package main

import "fmt"

func main() {
        fmt.Println("hello world")
}
EOF
$ iconv -f UTF-8 -t UTF-16 hello_UTF-16.go
$ go run hello_UTF-16.go
go: reading hello_UTF-16.go: unexpected NUL in input
```

The Byte Order Mark (BOM)

In practice, if a program only works with Unicode encodings (which is the goal of Unicode, after all), it can also check the *byte order mark* (BOM), a magic number representing the Unicode character, `U+FEFF BYTE ORDER MARK`. Depending on the starting byte sequence (`0xEF,0xBB,0xBF` for UTF-8, `0xFE 0xFF` for UTF-16 in big endian, `0xFF 0xFE` for UTF-16 in little endian, etc.), the parser can detect the encoding (and the endianness). But the BOM is optional and is often missing in UTF-8 text because the BOM character is not a valid ASCII character, and that would break the ASCII backward compatibility supported by UTF-8.

Once the encoding is known, the next step is to decode the text. We have already covered the Unicode encoding algorithms in the second part. The decoding algorithms are very similar and the code is omitted. What is more interesting is the question of how to represent Unicode text in memory. The most basic string representation for a programming language is to store a sequence of Unicode code points:

```
# A custom string implementation for illustration purposes.
class String:

    def __init__(self, codepoints=[]):
        self.codepoints = codepoints

    def __len__(self):
        return len(self.codepoints)

    def __getitem__(self, index):
        if index < len(self.codepoints):
            return self.codepoints[index]

s = String([0x0068, 0x0065, 0x0079, 0x1F600]) # "hey\N{Grinning Face Emoji}"
print(len(s)) # 4
print(s[0] == ord("h")) # True
```

In theory, this representation makes sense. A Unicode string is a sequence of code points. Encoding and decoding should only be performed when reading or sending a text to another program.

In practice, most Unicode characters land in the first plane (BMP), requiring only two bytes, and a lot of strings are composed of ASCII characters, requiring only one byte. It explains why most programming languages choose to represent strings differently. Unicode support is great but comes with a performance cost that implementations must limit.

For example, create a file `hello_UTF-8.py`:

```
print("Voila\u0300 \N{winking face}")
```

Then, convert the file in the different Unicode encodings:

```
$ iconv -f UTF-8 -t UTF-32 hello.py > hello_UTF-32.py
$ iconv -f UTF-8 -t UTF-16 hello.py > hello_UTF-16.py
```

Here are the file representations for the three Unicode encodings:

```
$ hexdump hello_UTF-8.py
0000000 70 72 69 6e 74 28 22 56 6f 69 6c 61 5c 4e 7b 63
0000010 6f 6d 62 69 6e 69 6e 67 20 61 63 63 65 6e 74 20
0000020 67 72 61 76 65 7d 20 5c 4e 7b 77 69 6e 6b 69 6e
0000030 67 20 66 61 63 65 7d 22 29 0a
000003a

$ hexdump hello_UTF-16.py
0000000 fe ff 00 70 00 72 00 69 00 6e 00 74 00 28 00 22
0000010 00 56 00 6f 00 69 00 6c 00 61 00 5c 00 4e 00 7b
0000020 00 63 00 6f 00 6d 00 62 00 69 00 6e 00 69 00 6e
0000030 00 67 00 20 00 61 00 63 00 63 00 65 00 6e 00 74
```

```
0000040 00 20 00 67 00 72 00 61 00 76 00 65 00 7d 00 20
0000050 00 5c 00 4e 00 7b 00 77 00 69 00 6e 00 6b 00 69
0000060 00 6e 00 67 00 20 00 66 00 61 00 63 00 65 00 7d
0000070 00 22 00 29 00 0a
0000076

$ hexdump hello_UTF-32.py
0000000 00 00 fe ff 00 00 00 70 00 00 00 72 00 00 00 69
0000010 00 00 00 6e 00 00 00 74 00 00 00 28 00 00 00 22
0000020 00 00 00 56 00 00 00 6f 00 00 00 69 00 00 00 6c
0000030 00 00 00 61 00 00 00 5c 00 00 00 4e 00 00 00 7b
0000040 00 00 00 63 00 00 00 6f 00 00 00 6d 00 00 00 62
0000050 00 00 00 69 00 00 00 6e 00 00 00 69 00 00 00 6e
0000060 00 00 00 67 00 00 00 20 00 00 00 61 00 00 00 63
0000070 00 00 00 63 00 00 00 65 00 00 00 6e 00 00 00 74
0000080 00 00 00 20 00 00 00 67 00 00 00 72 00 00 00 61
0000090 00 00 00 76 00 00 00 65 00 00 00 7d 00 00 00 20
00000a0 00 00 00 5c 00 00 00 4e 00 00 00 7b 00 00 00 77
00000b0 00 00 00 69 00 00 00 6e 00 00 00 6b 00 00 00 69
00000c0 00 00 00 6e 00 00 00 67 00 00 00 20 00 00 00 66
00000d0 00 00 00 61 00 00 00 63 00 00 00 65 00 00 00 7d
00000e0 00 00 00 22 00 00 00 29 00 00 00 0a
00000ec
```

We better understand why UTF-8 is preferred for writing code. The same motivation applies when designing the internal string representation.

EXAMPLE: JAVA

Before Java 9, `String` were represented internally as an array of char:

---

java/lang/String.java

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    /** The value is used for character storage. */
    private final char value[]; (1)

    public String(byte bytes[], int offset, int length, Charset charset) {
        if (charset == null)
            throw new NullPointerException("charset");
        checkBounds(bytes, offset, length);
        this.value =  StringCoding.decode(charset, bytes, offset, length); (2)
    }

    public char charAt(int index) {
        if ((index < 0) || (index >= value.length)) {
            throw new StringIndexOutOfBoundsException(index);
        }
        return value[index]; (3)
    }

    ...
}
```

---

(1) The Javadoc specifies that the `char` data type is based on the original 16-bits Unicode specification. Only characters in the BMP can be stored in a `char` and characters in other planes must use surrogate codes. In short, the `String` data type is a Unicode text encoded in UTF-16.

(2) The class `StringCoding` uses the charset to determine the decoding algorithm to convert the bytes into UTF-16.

(3) The method `charAt` retrieves a single character from its index.

Since, Java adopted compacts strings. A string is now represented in UTF-16 only if it contains at least one non-ASCII character. Otherwise, Java fallbacks to a basic implementation storing each character in a single byte.

The current `String` implementation was changed to use an array of bytes instead:

java/lang/String.java

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {

    /**
     * The value is used for character storage.
     */
    @Stable
    private final byte[] value; (1)

    /**
     * The identifier of the encoding used to encode the bytes in
     * {@code value}. The supported values in this implementation are
     *
     * LATIN1
     * UTF16
     */
    private final byte coder; (2)

    public String(byte bytes[], int offset, int length, Charset charset) {
        if (charset == null)
            throw new NullPointerException("charset");
        checkBoundsOffCount(offset, length, bytes.length);
        StringCoding.Result ret =
            StringCoding.decode(charset, bytes, offset, length); (3)
        this.value = ret.value;
        this.coder = ret.coder;
    }

    public char charAt(int index) { (4)
        if (isLatin1()) {
            return StringLatin1.charAt(value, index);
        } else {
            return StringUTF16.charAt(value, index);
        }
    }

    ...
}
```

(1) The Java `byte` type has a minimum value of -128 and a maximum value of 127 (inclusive). Depending on the content of the string, the bytes will be ASCII codes or UTF-16 bytes.

(2) The field `coder` is used by most methods in `String` to detect if the compact string optimization is used. This optimization is implemented by the new class `StringLatin1`. The former `String` implementation had been moved to `StringUTF16`.

(3) The class `StringCoding` now returns the value as bytes and the coder determined by searching for a non-ASCII character.

(4) The method `charAt` now delegates to concrete String implementations. `StringLatin1` continues to return the character at the specified index. `StringUTF16` needs to read two elements in `value` to read the two bytes representing a UTF-16 character.

The motivation for compact strings is to reduce the memory footprint when working with ASCII characters only. It can be confirmed easily using a minimalist benchmark:

```java
import java.util.ArrayList;
import java.util.List;

public class BenchmarkString {
    public static void main(String[] args) {
        List<String> results = new ArrayList<>(); // Keep strings to avoid GC
        Runtime runtime = Runtime.getRuntime();

        long startTime = System.nanoTime();
        long memoryBefore = runtime.totalMemory() - runtime.freeMemory();

        String loremIpsum = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident,
sunt in culpa qui officia deserunt mollit anim id est laborum.
        """;

        // StringLatin1
        for (int i = 0; i < 1000000; i++) {
            results.add((loremIpsum + i).toLowerCase()); ❶
        }

        long timeElapsed = System.nanoTime() - startTime;
        long memoryAfter = runtime.totalMemory() - runtime.freeMemory();
        long memoryUsed = memoryAfter - memoryBefore;
        System.out.println("(Latin1) Execution time: " + (timeElapsed / 1000000) + "ms");
        System.out.println("(Latin1) Memory usage: " +  (memoryUsed / 10000000) + "MB" );

        // StringUTF16
        for (int i = 0; i < 1000000; i++) {
            results.add((loremIpsum + "😀" + i).toLowerCase()); ❷
        }

        timeElapsed = System.nanoTime() - startTime;
        memoryAfter = runtime.totalMemory() - runtime.freeMemory();
        memoryUsed = memoryAfter - memoryBefore;
        System.out.println("(UTF-16) Execution time: " + (timeElapsed / 1000000) + "ms");
        System.out.println("(UTF-16) Memory usage: " +  (memoryUsed / 10000000) + "MB" );
    }
}
```

❶ The string contains only ASCII characters, which means Java will use compact strings.

❷ We add the emoji 😀 `GRINNING FACE` U+1F600 to force strings to be encoded in UTF-16.

The program outputs on my laptop:

```
$ javac BenchmarkString
$ java BenchmarkString
(Latin1) Execution time: 896ms
(Latin1) Memory usage: 61MB
(UTF-16) Execution time: 3162ms
(UTF-16) Memory usage: 185MB
```

If we look more closely at the UTF-16 case, we notice that the internal representation is not without consequence. Consider the following program:

```java
public class RepresentationUTF16 {
```

```
    public static void main(String[] args) {
        System.out.println("✋Hey".indexOf("H")); // Output: 1
        System.out.println("🤚Hey".indexOf("H")); // Output: 2
    }
}
```

Why does rotating the hand change the result? As discussed in the second part, UTF-16 is a variable-length character encoding. It means characters in the BMP are encoded using two bytes, whereas complementary characters are encoded using a surrogate pair (i.e., the equivalent of two codepoints). The two emojis look alike but are not stored in the same Unicode plane. ✋ `RAISED HAND` is assigned the codepoint U+270B (Plane 0) and 🤚 `RAISED BACK OF HAND` is assigned the codepoint U+1F91A (Plane 1).

Using UTF-16 for the internal representation saves bytes compared to using UTF-32, but the abstraction is leaky as the developer is not working with a sequence of Unicode code points:

```
public class RepresentationUTF16 {
    public static void main(String[] args) {
        System.out.println("✋Hey".codePointAt(1)); // U+0048 Latin Capital Letter H
        System.out.println("🤚Hey".codePointAt(1)); // U+DD1A Low Surrogate Code
        // Or
        System.out.println("✋Hey".charAt(1)); // H
        System.out.println("🤚Hey".charAt(1)); // ?
    }
}
```

The output makes sense when considering the internal representation:

```
String s1 = new String("\u270b\u0048\u0065\u0079".getBytes("UTF-16"), "UTF-16");
String s2 = new String("\uD83E\uDD1A\u0048\u0065\u0079".getBytes("UTF-16"), "UTF-16");
"✋Hey".equals(s1) // true
"🤚Hey".equals(s2) // true
```

We will continue the discussion of `String` in the next section when presenting their manipulation.

EXAMPLE: GO

Go encodes strings as a read-only slice of bytes. These bytes can be anything, even invalid Unicode code points. But as Go source code is always UTF-8, the slice of bytes for a string literal is also UTF-8 text.

For example:

```
s := "Hey 🤚!" // String literal stored in a UTF-8 file

fmt.Printf("len=%d\n", len(s))
// Print characters
for i := 0; i < len(s); i++ {
    fmt.Printf("%c ", s[i])
}
fmt.Println("")
// Print bytes
for i := 0; i < len(s); i++ {
    fmt.Printf("%v ", s[i])
}
```

```
// Output:
// len=9
// H e y   ð   !
// 72 101 121 32 240 159 164 154 33
```

Iterating over strings using this syntax does not work so well. We get bytes, not characters. We observe that these bytes correspond to the UTF-8 encoding, and we also notice that the `len` function returns the number of bytes in this encoding. This representation is not practical if we are interested by the Unicode code points.

To solve this, Go introduces the data type `rune` (a synonym of code point that is defined as a `int32`). If we convert the string to a slice of `rune`, we get a different result:

```
s := []rune("Hey 👋!") ❶

fmt.Printf("len=%d\n", len(s))
// Print the characters
for i := 0; i < len(s); i++ {
    fmt.Printf("%c ", s[i])
}
fmt.Println("")
// Print the code points
for i := 0; i < len(s); i++ {
    fmt.Printf("%#U ", s[i])
}

// Output:
// len=6
// H e y   👋 !
// U+0048 'H' U+0065 'e' U+0079 'y' U+0020 ' ' U+1F91A '👋' U+0021 '!'
```

❶ Cast the string into a slice of `rune`.

The output confirms that the string is composed of 6 Unicode code points. The same result can be obtained using a `for range` loop without having to cast the string explicitly:

```
s := "Hey 👋!"

for index, runeValue := range s {
    fmt.Printf("%#U starts at byte position %d\n", runeValue, index)
}

// Output:
// U+0048 'H' starts at byte position 0
// U+0065 'e' starts at byte position 1
// U+0079 'y' starts at byte position 2
// U+0020 ' ' starts at byte position 3
// U+1F91A '👋' starts at byte position 4
// U+0021 '!' starts at byte position 8
```

The output shows how each code point occupies a different number of bytes. For example, the emoji uses 4 bytes starting at the index 4.

Like Java, we can note that the Go internal string representation is not transparent for the developer. What about Python?

EXAMPLE: PYTHON

Python supports, since the version 3.3, multiple internal representations, depending on the character with the largest Unicode code point (1, 2, or 4 bytes). The implementation saves space in most cases and gives access to the whole "UTF-32" if needed.

```
print(len("✋")) # 1
print(len("🤚")) # 1

for c in "Hey 🤚!":
    print(c, hex(ord(c)))
    # H 0x48
    # e 0x65
    # y 0x79
    #   0x20
    # 🤚 0x1f91a
    # ! 0x21
```

The idea behind the Python implementation is similar to the Java implementation, and we will omit the code consequently. However, we observe that the internal implementation is transparent for the Python developer. Strings are sequences of Unicode code points where the length is not affected by the encoding used internally.

PROCESSING

Programming languages offers many functions operating on strings. We will cover a few of them.

LENGTH()

Determining the length of a string can be challenging with Unicode. A possible implementation is simply to return the number of code points in the string, but the result can be surprising:

```
# Python 3
print(len("à"))
# Output: 2
```

Why? The result is not as surprising as it may seem. We have mentioned that the Unicode Standard defines combining characters, like when defining accentuated letters. To understand the example, you need to remember that your editor (or browser) is rendering the text using glyphs, and thus have to manage these combining characters. The same example can be rewritten using the following syntax:

```
# Python 3
print(len("\N{LATIN SMALL LETTER A}\N{COMBINING GRAVE ACCENT}"))
# Or "\u0061\u0300"
```

Like most examples in this section, you can reproduce the same behavior in other languages like Java or Go. The fact is our Unicode text is really composed of two Unicode characters. The result may be considered correct or wrong depending on what you are looking for.

EQUALS()

Comparing if two strings are equals is not without surprise neither.

```golang
// Golang
package main

import "fmt"

func main() {
    fmt.Println("à" == "à")
    // Output: false
}
```

Why? Convertibility is one of the core design principles behind Unicode. Converting between Unicode and any other encoding must be as easy as possible, which means if a character exists in an encoding, it must also exist in Unicode to have a simple one-to-one mapping between the two encodings. So, even if Unicode favors combining characters to represent accentuated letters, prior encodings often include a single character for letters such as à. The same example can be rewritten using the following syntax:

```golang
// Golang
package main

import "fmt"

func main() {
    fmt.Println("\u00E0" == "\u0061\u0300")
    // Output: false
}
```

According to the Unicode standard, these two strings are canonically equivalent and should be treated as equal even if their sequences of bytes are different.

To solve this problem, Unicode defines different normalization algorithms to convert equivalent strings to the same normal form. These algorithms are implemented by programming languages but are rarely applied systematically (normalization isn't free). Java, Python, and Go all provide such functions.

```golang
package main

import (
    "fmt"

    "golang.org/x/text/unicode/norm"
)

func main() {
    norm1 := norm.NFD.String("\u00E0")
    norm2 := norm.NFD.String("\u0061\u0300")
    fmt.Println(norm1 == norm2)
    // Output: true
}
```

In practice, normalization is not always required, and it is crucial to understand why it exists and when to apply it, like when receiving a Unicode text from an external program.

Normalization also occurs when you are not expecting it. Consider the following program in Python:

```python
# Python 3
ℌ = "Me"
H = "Funny"
print(ℌ == H)
# Output: True
```

Why? Python accepts non-ASCII characters in identifiers but all identifiers are passed to a function to normalize their name first. In this example, the character ℌ has the same normal form as the character H:

```python
# Python 3
import unicodedata
print(unicodedata.normalize('NFKC', "ℌ"))
# Output: "H"
```

EXAMPLE: UPPER()

A function like `upper()` in Python is often used to make case-insensitive comparisons. But not all scripts use cases, and there aren't just two cases (ex: titlecase is used for book titles).

Case folding is defined by the Unicode Standard as a solution for caseless comparison of text to overcome these limitations, but Unicode also provides support to implement these classic functions using the *Unicode Character Database*—the set of data files containing character properties. In particular, we are looking for the property `Simple_Lowercase_Mapping` defined in the file `UnicodeData.txt`.

UnicodeData.txt

```
0041;LATIN CAPITAL LETTER A;Lu;0;L;;;;;N;;;;0061;
...
0061;LATIN SMALL LETTER A;Ll;0;L;;;;;N;;;0041;;0041
```

The last three columns of the file are the properties `Simple_Uppercase_Mapping`, `Simple_Lowercase_Mapping`, and `Simple_Titlecase_Mapping`.

Using this file, it is easy to implement a method to convert a string to uppercase:

String.py

```python
class String:

    UnicodeData = {}

    def __init__(self, codepoints=[]):
        self.codepoints = codepoints

    def __len__(self):
        return len(self.codepoints)

    def __getitem__(self, index):
        if index < len(self.codepoints):
            return self.codepoints[index]

    def upper(self):
        res = []
        for cl in self.codepoints:
            cu = String.UnicodeData[cl].get("Simple_Uppercase_Mapping", None)
            if cu:
                res.append(int("0x" + cu, 0))
```

```
            else:
                res.append(cl)
        return String(res)

def loadUCD():
    ucd = {}
    with open('./UnicodeData.txt') as fp:
        for line in fp:
            (codepoint, *_, upper, lower, title) = line.split(";")
            ucd[int("0x" + codepoint, 0)] = {
                "Simple_Uppercase_Mapping": upper,
                "Simple_Lowercase_Mapping": lower,
                "Simple_Titlecase_Mapping": title,
            }
    String.UnicodeData = ucd

loadUCD()

s = String([0x0068, 0x0065, 0x0079, 0x1F600]) # "hey😀"

print("".join(map(chr, s.upper()))) # Convert bytes to string
# Output: HEY😀
```

The implementation in popular programming languages follows the same logic with optimizations concerning the loading of the Unicode Character Database.

EXAMPLE: PYTHON

The string type is implemented in C in the file `unicodeobject.c`. Here is the method to test if a character is uppercase:

Objects/unicodectype.c

```
typedef struct {
    /*
       These are either deltas to the character or offsets in
       _PyUnicode_ExtendedCase.
    */
    const int upper;
    const int lower;
    const int title;
    /* Note if more flag space is needed, decimal and digit could be unified. */
    const unsigned char decimal;
    const unsigned char digit;
    const unsigned short flags;
} _PyUnicode_TypeRecord;

...

/* Returns 1 for Unicode characters having the category 'Lu', 0
   otherwise. */

int _PyUnicode_IsUppercase(Py_UCS4 ch)
{
    const _PyUnicode_TypeRecord *ctype = gettyperecord(ch);

    return (ctype->flags & UPPER_MASK) != 0;
}
```

The code relies on a global structure initialized using the Unicode Character Database. The script `Tools/unicode/makeunicodedata.py` converts Unicode database files (e.g., `UnicodeData.txt`) to `Modules/unicodedata_db.h`, `Modules/unicodename_db.h`, and `Objects/unicodetype_db.h`.

Tools/unicode/makeunicodedata.py

```
def makeunicodetype(unicode, trace): ❶
    ...

    for char in unicode.chars: ❷
        record = unicode.table[char]
        # extract database properties
        category = record.general_category
        bidirectional = record.bidi_class
        properties = record.binary_properties
        flags = 0
        if category in ["Lm", "Lt", "Lu", "Ll", "Lo"]:
            flags |= ALPHA_MASK
        if "Lowercase" in properties:
            flags |= LOWER_MASK
        if "Uppercase" in properties:
            flags |= UPPER_MASK
        ...
```

❶ The method `makeunicodetype` generates the file `Objects/unicodetype_db.h`.

❷ The variable `unicode` contains the content of `UnicodeData.txt`.

I invite you to check the generated files like `Objects/unicodetype_db.h`. These files are not a simple list of all Unicode characters but use additional optimizations. We can ignore these low-level details for the purpose of this article.

EXAMPLE: JAVA

Java implements the string data type in the class `java.lang.String`. The code is large due to several evolutions like compact strings.

Here is the code of the method `toUpperCase()`:

src/java.base/share/classes/java/lang/String.java

```
package java.lang;

public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {

    /**
     * Converts all of the characters in this {@code String} to upper
     * case using the rules of the given {@code Locale}. Case mapping is based
     * on the Unicode Standard version specified by the
     * {@link java.lang.Character Character} class.
     *
     * @param locale use the case transformation rules for this locale
     * @return the {@code String}, converted to uppercase.
     */
    public String toUpperCase(Locale locale) {
        return isLatin1() ? StringLatin1.toUpperCase(this, value, locale)
                          : StringUTF16.toUpperCase(this, value, locale);
    }

    ...
}
```

We need to check the class `java.lang.Character` to find out more about the conversion:

src/java.base/share/classes/java/lang/Character.java

```
package java.lang;

public final class Character {

    /**
     * Converts the character (Unicode code point) argument to
     * uppercase using case mapping information from the UnicodeData
     * file.
     *
     * @param   codePoint   the character (Unicode code point) to be converted.
     * @return  the uppercase equivalent of the character, if any;
     *          otherwise, the character itself.
     */
    public static int toUpperCase(int codePoint) {
        return CharacterData.of(codePoint).toUpperCase(codePoint);
    }

    ...
}
```

`java.lang.CharacterData` is an abstract class:

---

src/java.base/share/classes/java/lang/CharacterData.java

```
package java.lang;

abstract class CharacterData {

    abstract boolean isUpperCase(int ch);
    abstract int toUpperCase(int ch);
    // ...

    static final CharacterData of(int ch) {
        if (ch >>> 8 == 0) {     // fast-path (1)
            return CharacterDataLatin1.instance;
        } else {
            switch(ch >>> 16) {  //plane 00-16
            case(0):
                return CharacterData00.instance;
            case(1):
                return CharacterData01.instance;
            case(2):
                return CharacterData02.instance;
            case(3):
                return CharacterData03.instance;
            case(14):
                return CharacterData0E.instance;
            case(15):   // Private Use
            case(16):   // Private Use
                return CharacterDataPrivateUse.instance;
            default:
                return CharacterDataUndefined.instance;
            }
        }
    }
}
```

---

(1) The fast-path is an optimization for ASCII characters to avoid traversing the larger Unicode database.

The classes `CharacterDataXX` contain the properties for each plane of the Unicode Character Table and are generated by the Java build process. The definition is present in `make/modules/java.base/gensrc/GensrcCharacterData.gmk`:

---

make/modules/java.base/gensrc/GensrcCharacterData.gmk

```
#
# Rules to create $(SUPPORT_OUTPUTDIR)/gensrc/java.base/sun/lang/CharacterData*.java
#

GENSRC_CHARACTERDATA :=

CHARACTERDATA = $(TOPDIR)/make/data/characterdata
UNICODEDATA = $(TOPDIR)/make/data/unicodedata

define SetupCharacterData
  $(SUPPORT_OUTPUTDIR)/gensrc/java.base/java/lang/$1.java: \
      $(CHARACTERDATA)/$1.java.template
    $$(call LogInfo, Generating $1.java)
    $$(call MakeDir, $$(@D))
    $(TOOL_GENERATECHARACTER) $2 $(DEBUG_OPTION) \
        -template $(CHARACTERDATA)/$1.java.template \
        -spec $(UNICODEDATA)/UnicodeData.txt \ (1)
        -specialcasing $(UNICODEDATA)/SpecialCasing.txt \ (1)
        -proplist $(UNICODEDATA)/PropList.txt \ (1)
        -derivedprops $(UNICODEDATA)/DerivedCoreProperties.txt \ (1)
        -o $(SUPPORT_OUTPUTDIR)/gensrc/java.base/java/lang/$1.java \
        -usecharforbyte $3

  GENSRC_CHARACTERDATA += $(SUPPORT_OUTPUTDIR)/gensrc/java.base/java/lang/$1.java
endef

$(eval $(call SetupCharacterData,CharacterDataLatin1, , -latin1 8))
$(eval $(call SetupCharacterData,CharacterData00, -string -plane 0, 11 4 1))
$(eval $(call SetupCharacterData,CharacterData01, -string -plane 1, 11 4 1))
$(eval $(call SetupCharacterData,CharacterData02, -string -plane 2, 11 4 1))
$(eval $(call SetupCharacterData,CharacterData03, -string -plane 3, 11 4 1))
$(eval $(call SetupCharacterData,CharacterData0E, -string -plane 14, 11 4 1))

$(GENSRC_CHARACTERDATA): $(BUILD_TOOLS_JDK)

TARGETS += $(GENSRC_CHARACTERDATA)
```

---

(1) The input files correspond to the UCD files we talked about in the section about the Unicode Character Database.

Here is a preview of the resulting code:

/gensrc/java.base/java/lang/CharacterData00.java

```
package java.lang;

class CharacterData00 extends CharacterData {

    int toUpperCase(int ch) {
        int mapChar = ch;
        int val = getProperties(ch);

        if ((val & 0x00010000) != 0) {
            if ((val & 0x07FC0000) == 0x07FC0000) {
                switch(ch) {
                case 0x00B5 : mapChar = 0x039C; break;
                case 0x017F : mapChar = 0x0053; break;
                case 0x1FBE : mapChar = 0x0399; break;
                case 0x1F80 : mapChar = 0x1F88; break;
                case 0x1F81 : mapChar = 0x1F89; break;
                case 0x1F82 : mapChar = 0x1F8A; break;
                case 0x1F83 : mapChar = 0x1F8B; break;
                case 0x1F84 : mapChar = 0x1F8C; break;
                case 0x1F85 : mapChar = 0x1F8D; break;
                case 0x1F86 : mapChar = 0x1F8E; break;
                // Many more
            }
```

```
        }
        else {
            int offset = val  << 5 >> (5+18);
            mapChar =  ch - offset;
        }

        return mapChar;
    }
}
```

Like Python, we observe various optimizations but the overall idea is similar—we generate static code from the Unicode data files. In Java, accessing the properties of a character is more obvious thanks to `switch` statements using code points, whereas in Python, we have to manipulate bytes to determine the index of the code point first.

EXAMPLE: GO

The Go implementation is really close to previous languages. Go strings are implemented in Go by the file `src/strings/strings.go`, which declares the function `ToUpper()`:

---

src/strings/strings.go

```
// ToUpper returns s with all Unicode letters mapped to their upper case.
func ToUpper(s string) string {
    isASCII, hasLower := true, false
    for i := 0; i < len(s); i++ {
        c := s[i]
        if c >= utf8.RuneSelf { (1)
            isASCII = false
            break
        }
        hasLower = hasLower || ('a' <= c && c <= 'z')
    }

    if isASCII { // optimize for ASCII-only strings.
        if !hasLower {
            return s
        }
        var b Builder
        b.Grow(len(s))
        for i := 0; i < len(s); i++ {
            c := s[i]
            if 'a' <= c && c <= 'z' {
                c -= 'a' - 'A' (2)
            }
            b.WriteByte(c)
        }
        return b.String()
    }
    return Map(unicode.ToUpper, s) (3)
}
```

---

(1) `RuneSelf` is a constant with the value `0x80` (128) to determine if the code point is an ASCII-compatible character.

(2) Before Unicode, converting a string in uppercase was easily implemented by substracting the differences between the index `a` and `A` since characters were ordered in the character set.

(3) The real implementation is defined by the `unicode` package.

The Unicode Character Database (e.g., `UnicodeData.txt`) is converted to static code in the file `src/unicode/tables.go`. Go implements various optimizations using different structures. For example, instead of storing the mapping between every single uppercase and lowercase

letter, Go groups them in instances of `CaseRange`:

src/unicode/letter.go

```
// Indices into the Delta arrays inside CaseRanges for case mapping.
const (
    UpperCase = iota
    LowerCase
    TitleCase
    MaxCase
)

type d [MaxCase]rune // to make the CaseRanges text shorter

// CaseRange represents a range of Unicode code points for simple (one
// code point to one code point) case conversion.
// The range runs from Lo to Hi inclusive, with a fixed stride of 1. Deltas
// are the number to add to the code point to reach the code point for a
// different case for that character. They may be negative. If zero, it
// means the character is in the corresponding case. There is a special
// case representing sequences of alternating corresponding Upper and Lower
// pairs. It appears with a fixed Delta of
//  {UpperLower, UpperLower, UpperLower}
// The constant UpperLower has an otherwise impossible delta value.
type CaseRange struct {
    Lo    uint32
    Hi    uint32
    Delta d
}
```

For example:

src/unicode/tables.go

```
var _CaseRanges = []CaseRange{
    {0x0041, 0x005A, d{0, 32, 0}},    (1)
    {0x0061, 0x007A, d{-32, 0, -32}}, (2)
    ...
}
```

(1) For Unicode characters in the range A—Z, add 32 to the code point to get the uppercase character.

(2) For Unicode characters in the range a—z, subtract 32 to the code point to get the lowercase or titlecase character.

This variable is then used by the function `to`, which is called by higher-level functions such as `ToUpper`, `ToLower`:

src/unicode/letter.go

```
// to maps the rune using the specified case mapping.
// It additionally reports whether caseRange contained a mapping for r.
func to(_case int, r rune, caseRange []CaseRange) (mappedRune rune, foundMapping bool) { (1)
    if _case < 0 || MaxCase <= _case {
        return ReplacementChar, false // as reasonable an error as any
    }
    // binary search over ranges
    lo := 0
    hi := len(caseRange)
    for lo < hi { (2)
        m := lo + (hi-lo)/2
        cr := caseRange[m]
        if rune(cr.Lo) <= r && r <= rune(cr.Hi) {
            delta := cr.Delta[_case]
```

```
            if delta > MaxRune {
                // In an Upper-Lower sequence, which always starts with
                // an UpperCase letter, the real deltas always look like:
                //  {0, 1, 0}    UpperCase (Lower is next)
                //  {-1, 0, -1}  LowerCase (Upper, Title are previous)
                // The characters at even offsets from the beginning of the
                // sequence are upper case; the ones at odd offsets are lower.
                // The correct mapping can be done by clearing or setting the low
                // bit in the sequence offset.
                // The constants UpperCase and TitleCase are even while LowerCase
                // is odd so we take the low bit from _case.
                return rune(cr.Lo) + ((r-rune(cr.Lo))&^1 | rune(_case&1)), true
            }
            return r + delta, true (3)
        }
        if r < rune(cr.Lo) { (2)
            hi = m
        } else {
            lo = m + 1
        }
    }
    return r, false
}
```

1. The function is called with a constant `UpperCase` or `LowerCase` as the first argument and a single character to convert.
2. Go uses binary search to locate the Unicode range in O(log N).
3. Once the range is found, simply add the delta to the code point.

MATCHES()

We will close the string manipulation section with a classic example: regular expressions.

Consider the following example in Python 3:

```
# Python 3
import re

s = "100 µAh 10 mAh"
res = re.findall(r'\\d+ \\wAh', s)
print(len(res))
# Output: 1
```

Now, consider the same program with a small difference (we declare the regular expression using a `str`):

```
# Python 3
import re
s = "100 µAh 10 mAh"
res = re.findall("\\d+ \\wAh", s)
print(len(res))
# Output: 2
```

Why? The reason is specific to the Python regex engine implementation. If the regex pattern is in bytes (e.g., when using `r'\\w'`), `\w` matches any alphanumeric character (`[a-zA-Z0-9_]`). If the regex pattern is a string (e.g., when using `"\\w"`), `\w` matches all characters marked as letters in the Unicode database.

In practice, most languages are subject to this restriction:

```java
import java.text.Normalizer;
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class StringTest {

    public static void main(String[] args) {
        String s = "100 µAh 10 mAh";
        Pattern p = Pattern.compile("\\d+ \\wAh");
        Matcher m = p.matcher(s);
        System.out.println(m.results().count());
        // Output: 1

        // Using a special character class
        p = Pattern.compile("\\d+ \\p{L}Ah");
        m = p.matcher(s);
        System.out.print(m.results().count());
        // Output: 2
    }
}
```

In Java, the metacharacter `\w` also matches `[a-zA-Z_0-9]` (which is faster than checking the Unicode Character Database). Other character classes exist like `\p{L}`. (`L` matches a single code point in the category "letter", but other values are possible: `N` for any kind of numeric character in any script, etc.) This syntax is also supported by Go.

Emojis in identifiers?

Most languages require source files to be encoded in Unicode, but that does not mean these languages accept any Unicode character in variable names. Rules differs among languages but most languages like Java, Python, and Go accept only characters considered as letters or digits in the Unicode table (ex: ツ, Δ, π).

Some languages do not have these restrictions. You can write hieroglyphs in Hashell:

```
𓀀 :: (𓀁 -> 𓀂) -> [𓀁] -> [𓀂]
𓀀 𓀃 (𓀄:𓀅) = 𓀃 𓀆 : 𓀀 𓀃 𓀅
𓀀 _ _ = []
```

Or write entire programs in PHP without using any ASCII character for identifiers:

```php
<?php

class 😀 {
    public function 🍅(...$🍔) {
        $📜 = [
            '🥝' => 61,
            '🍫' => 546,
            '🍪' => 502,
            '🍔' => 515,
            '🍟' => 624,
            '🍏' => 52,
            '🥗' => 280,
        ];
```

```
        $Σ = 0;
        foreach($🥗 as $🍞) {
            $Σ += $📜[$🍞];
        }
        if ($Σ < 1000) {
            return '😀';
        } else if ($Σ < 2000) {
            return '🤢';
        } else {
            return '🤮';
        }
    }
}

$👱 = new 😃();
$👮 = new 😃();
echo $👱->🍽('🥝', '🍏', '🍓');
echo $👮->🍽('🍪', '🍔', '🍟', '🍫');
```

WRITING

A program should usually output some texts, for example, when printing messages in the console or when sending documents to another program when calling a remote API.

What this means is we have to convert Unicode texts to bytes. We need to encode them using one of the Unicode encodings.

For example, when printing a hello message in the standard output:

```
# output.py
print("Hello 👋")
```

The Python interpreter outputs:

```
$ python3 output.py | hexdump
0000000 48 65 6c 6c 6f 20 f0 9f 91 8b 0a (1)
```

(1) The BOM is not included in the resulting representation. The emoji is the single character using four bytes (`f0 9f 91 8b`), which confirms the UTF-8 encoding.

Writing to the console is no different than writing to a file. The function `print` will send bytes to the file descriptor 1 ("stdout") using a specific encoding. The same rules that we covered before apply here too.

Here is a different version of the same code showing the logic explicitly:

```
import os

with os.fdopen(1, 'wb') as stdout:
    stdout.write("Hello 👋\n".encode("utf-8"))
```

Writing texts to the console, a file, or a socket makes no difference concerning Unicode.

RENDERING

Until now, we still haven't try to render Unicode text. Sooner or later, the text will have to be displayed on the screen of a computer device.

Rendering a text means converting a Unicode sequence of code points into a sequence of glyphs. These glyphs are present in font files, which are often provided by your operating system.

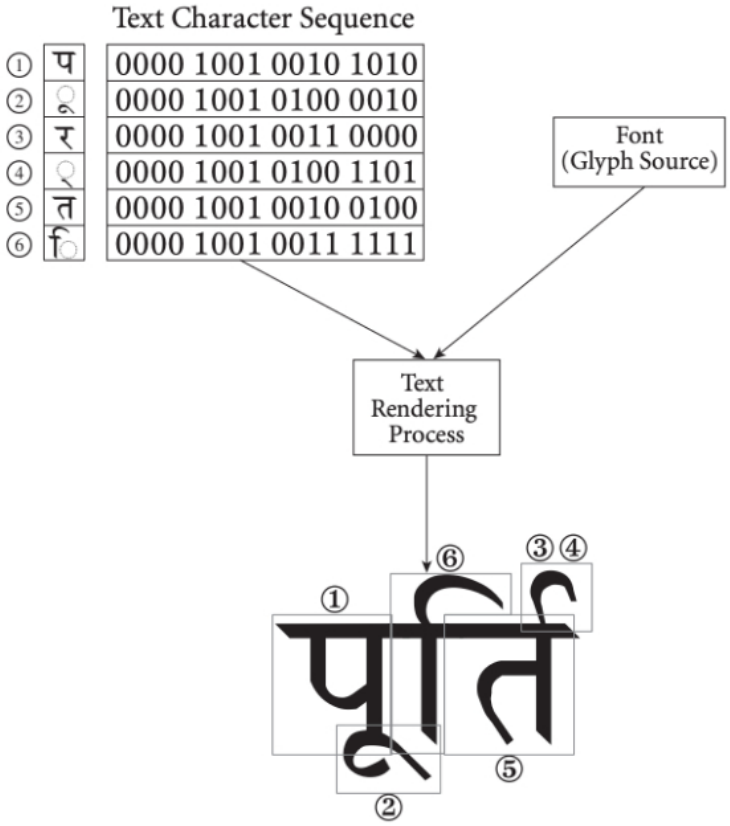


Figure 16. Unicode Character Code to Rendered Glyphs (From Unicode Standard, Figure 2.3)

Here are the fonts available under MacOS:

```
$ ls -1 /System/Library/Fonts/
Apple Braille Outline 6 Dot.ttf
Apple Braille Outline 8 Dot.ttf
Apple Braille Pinpoint 6 Dot.ttf
Apple Braille Pinpoint 8 Dot.ttf
Apple Braille.ttf
Apple Color Emoji.ttc
Apple Symbols.ttf
AppleSDGothicNeo.ttc
AquaKana.ttc
ArabicUIDisplay.ttc
ArabicUIText.ttc
ArialHB.ttc
Avenir Next Condensed.ttc
Avenir Next.ttc
Avenir.ttc
Courier.dfont
GeezaPro.ttc
Geneva.dfont
HelveLTMM
Helvetica.ttc
HelveticaNeue.ttc
HelveticaNeueDeskInterface.ttc
```

Hiragino Sans GB.ttc
Keyboard.ttf
Kohinoor.ttc
KohinoorBangla.ttc
KohinoorGujarati.ttc
KohinoorTelugu.ttc
LastResort.otf
LucidaGrande.ttc
MarkerFelt.ttc
Menlo.ttc
Monaco.dfont
MuktaMahee.ttc
NewYork.ttf
NewYorkItalic.ttf
Noteworthy.ttc
NotoNastaliq.ttc
NotoSansArmenian.ttc
NotoSansKannada.ttc
NotoSansMyanmar.ttc
NotoSansOriya.ttc
NotoSerifMyanmar.ttc
Optima.ttc
Palatino.ttc
PingFang.ttc
SFCompactDisplay.ttf
SFCompactRounded.ttf
SFCompactText.ttf
SFCompactTextItalic.ttf
SFNS.ttf
SFNSDisplayCondensed-Black.otf
SFNSDisplayCondensed-Bold.otf
SFNSDisplayCondensed-Heavy.otf
SFNSDisplayCondensed-Light.otf
SFNSDisplayCondensed-Medium.otf
SFNSDisplayCondensed-Regular.otf
SFNSDisplayCondensed-Semibold.otf
SFNSDisplayCondensed-Thin.otf
SFNSDisplayCondensed-Ultralight.otf
SFNSItalic.ttf
SFNSMono.ttf
SFNSMonoItalic.ttf
SFNSRounded.ttf
SFNSTextCondensed-Bold.otf
SFNSTextCondensed-Heavy.otf
SFNSTextCondensed-Light.otf
SFNSTextCondensed-Medium.otf
SFNSTextCondensed-Regular.otf
SFNSTextCondensed-Semibold.otf
STHeiti Light.ttc
STHeiti Medium.ttc
Supplemental
Symbol.ttf
Thonburi.ttc
Times.ttc
TimesLTMM
ZapfDingbats.ttf
ヒラギノ丸ゴ ProN W4.ttc
ヒラギノ明朝 ProN.ttc
ヒラギノ角ゴシック W0.ttc
ヒラギノ角ゴシック W1.ttc
ヒラギノ角ゴシック W2.ttc
ヒラギノ角ゴシック W3.ttc
ヒラギノ角ゴシック W4.ttc
ヒラギノ角ゴシック W5.ttc
ヒラギノ角ゴシック W6.ttc
ヒラギノ角ゴシック W7.ttc
ヒラギノ角ゴシック W8.ttc
ヒラギノ角ゴシック W9.ttc

Fonts comes in different formats, like the OpenType format (`.otf`, `.otc`, `.ttf`, `.ttc`). A font is a collection of tables of glyphs. Here is a preview of the font `NotoSansArmenian.ttf`:

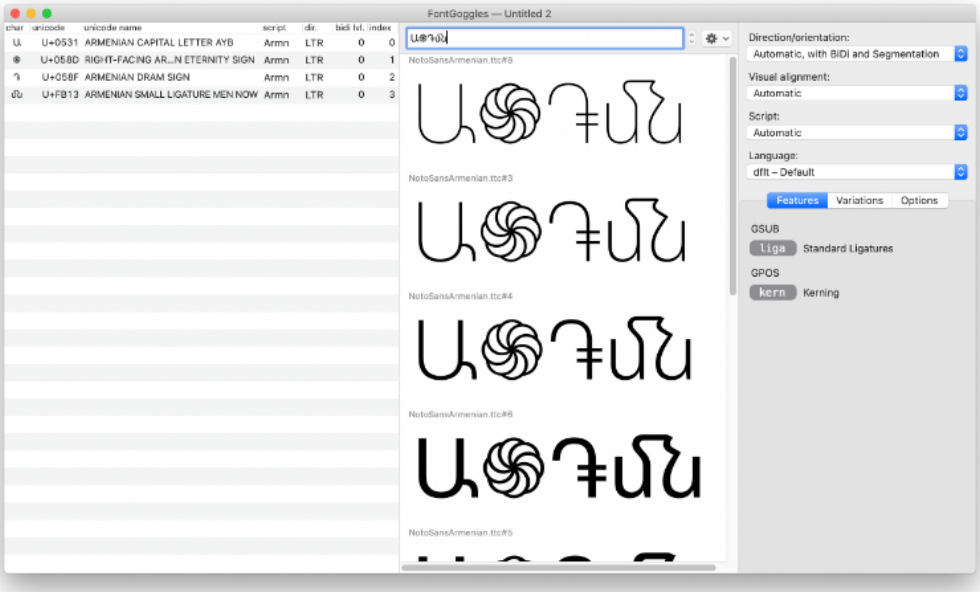


Figure 17. Examples of glyphs present in the font *Noto Sans Armenian* using FontGoggles.

Most fonts (if not all) do not contain glyphs for every Unicode character.[6] For example, trying to display "Hello" using the same font found no glyphs and shows empty squares ☐:

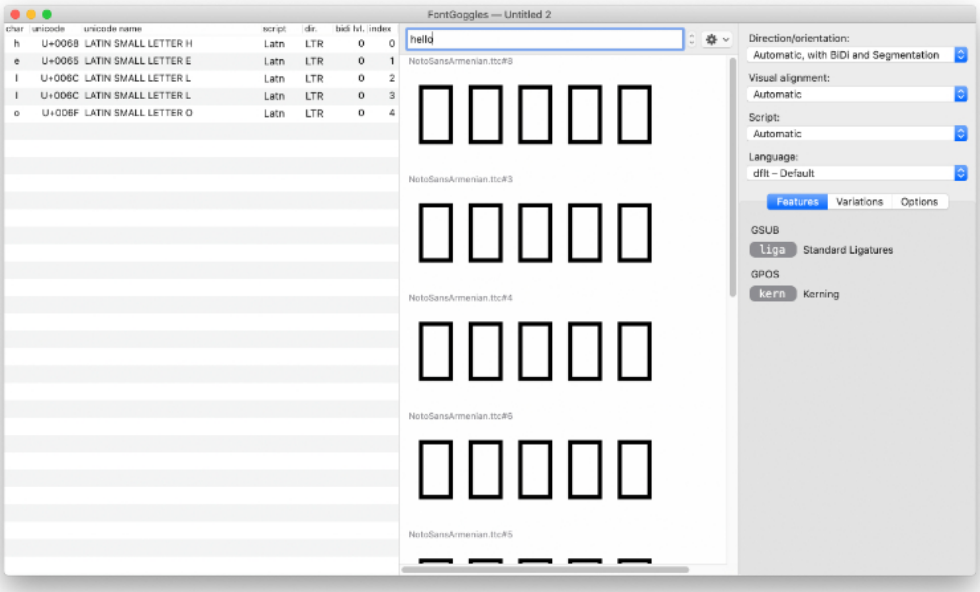


Figure 18. The same font found no glyphs to display ASCII characters.

Applications use therefore a mechanism called *font fallback* during text rendering. If some characters in a string are not supported in a given font, several fonts are tried successively until finding a matching font. If no glyph can be found, a white square ☐ will be displayed instead.

To illustrate this point, I created a basic HTML page printing a sample of characters from every script defined by Unicode. The page is rendered by Chrome like this (or try it in your

browser):

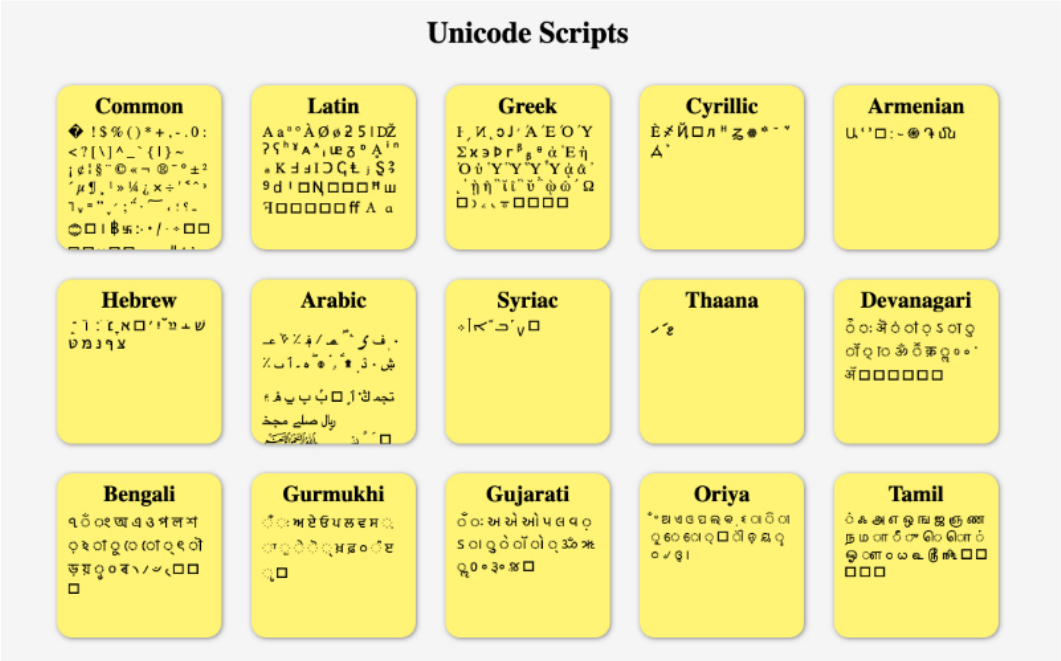


Figure 19. HTML page using characters from all Unicode scripts rendered in Chrome.

Using Chrome DevTools, we can easily find out which fonts were used to render characters on screen:

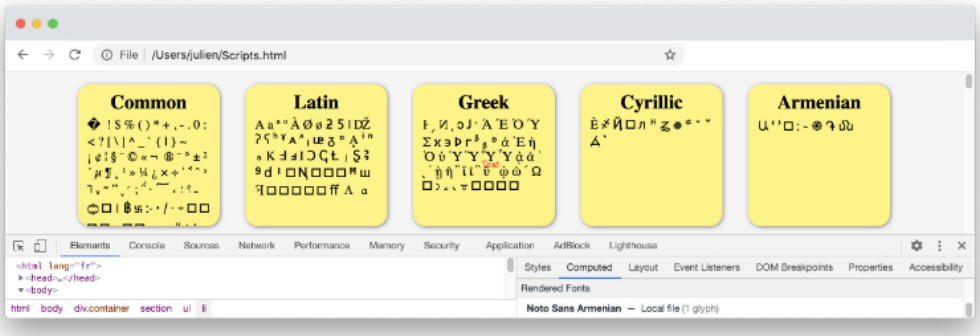


Figure 20. Demo HTML page displayed in Chrome using characters from all Unicode scripts.

If we inspect the Greek script:

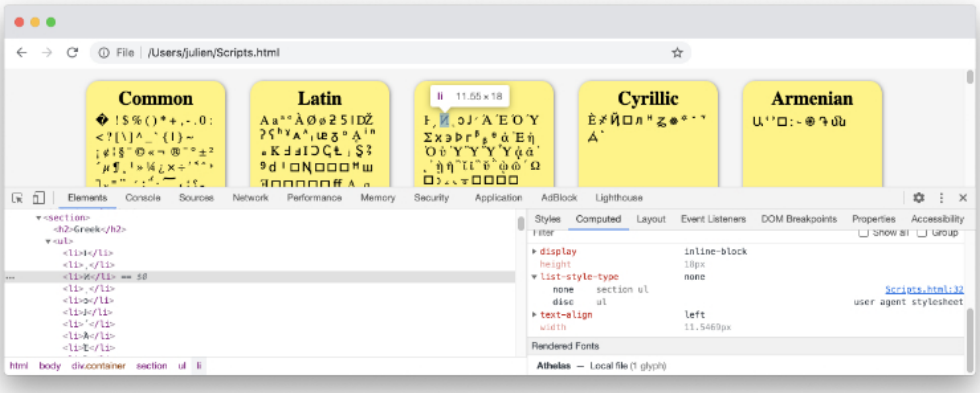


Figure 21. The Greek character `Greek Capital Letter Pamphylian Digamma` \U0376 is

displayed using the font *Athelas*.

In addition, CSS allows us to influence the order of fonts tried during the font fallback mechanism. For example, if we import the font "Oi", available from Google Fonts:

```
<head>
...
    <link rel="preconnect" href="https://fonts.gstatic.com">
    <link href="https://fonts.googleapis.com/css2?family=Oi&display=swap" rel="stylesheet">
...
</head>
```

And add the CSS declaration `font-family: 'Oi', Serif;` in our stylesheet:

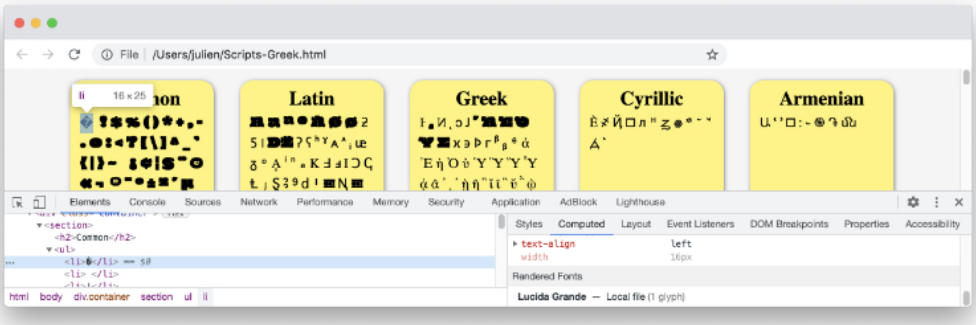


Figure 22. The Chrome rendering engine favors the font *Oi* when glyphs exist, and fallbacks to other fonts otherwise.

The challenge to render Unicode text is thus to convert a sequence of Unicode characters into a sequence of glyphs, using a list of available fonts. This process is called *text shaping*.

The logic is not simple. There is not always a 1-1 mapping between characters and glyphs. Unicode contains, for example, combining characters, emojis characters, and some emojis like flags are made by combining two abstract characters (ex: the US flag 🇺🇸 is represented by the two characters `REGIONAL INDICATOR SYMBOL LETTER U` + `REGIONAL INDICATOR SYMBOL LETTER S`). Implementing these rules is the job of the shaping engine.

Not all programs have to implement this logic from scratch. The library FreeType is used by major systems (GNU/Linux, iOS, Android, Firefox, Ghostscript, etc.). This represents billions of devices!

Under the hood, this library depends on the shaping engine HarfBuzz, which does a lot of the hard work. Both programs are written in C and are low-level code. Here is a simple program using FreeType to render a sequence of Unicode characters (the original code appears in this StackOverflow thread):

clfontpng.cc

```
// = Requirements: freetype 2.5, libpng, libicu, libz, libzip2
// = How to compile:
//  % export CXXFLAGS=`pkg-config --cflags freetype2 libpng`
//  % export LDFLAGS=`pkg-config --libs freetype2 libpng`
//  % clang++ -o clfontpng -static $(CXXFLAGS) clfontpng.cc $(LDFLAGS) \
//    -licuuc -lz -lbz2
```

```cpp
#include <cassert>
#include <cctype>
#include <iostream>
#include <memory>
#include <vector>
#include <string>

#include <stdio.h>

#include <unicode/umachine.h>
#include <unicode/utf.h>

#include <ft2build.h>
#include FT_FREETYPE_H
#include FT_TRUETYPE_TABLES_H

#define PNG_SKIP_SETJMP_CHECK
#include <png.h>

namespace {

const char* kDefaultOutputFile = "out.png";
const int kBytesPerPixel = 4; // RGBA
const int kDefaultPixelSize = 128;
const int kSpaceWidth = kDefaultPixelSize / 2;

FT_Library gFtLibrary;

// Only support horizontal direction.
class DrawContext {
 public:
  DrawContext()
      : pos_(0), width_(0), height_(0) {}
  uint8_t* Bitmap() { return &bitmap_[0]; }
  const uint32_t Width() const { return width_; }
  const uint32_t Height() const { return height_; }
  void SetSize(int width, int height) {
    width_ = width;
    height_ = height;
    int size = width * height * kBytesPerPixel;
    bitmap_.resize(size);
    bitmap_.assign(size, 0x00);
  }
  void Advance(int dx) { pos_ += dx; }
  uint8_t* GetDrawPosition(int row) {
    uint32_t index =(row * width_ + pos_) * kBytesPerPixel;
    assert(index < bitmap_.size());
    return &bitmap_[index];
  }
 private:
  DrawContext(const DrawContext&) = delete;
  DrawContext& operator=(const DrawContext&) = delete;

  uint32_t pos_;
  uint32_t width_;
  uint32_t height_;
  std::vector<uint8_t> bitmap_;
};

struct FaceOptions {
  int pixel_size;
  int load_flags;
  FT_Render_Mode render_mode;
  FaceOptions()
      : pixel_size(kDefaultPixelSize)
      , load_flags(0), render_mode(FT_RENDER_MODE_NORMAL) {}
};

class FreeTypeFace {
 public:
```

```
FreeTypeFace(const std::string& font_file)
    : font_file_(font_file)
    , options_()
    , face_(nullptr)
{
  error_ = FT_New_Face(gFtLibrary, font_file_.c_str(), 0, &face_);
  if (error_) {
    face_ = nullptr;
    return;
  }
  if (IsColorEmojiFont())
    SetupColorFont();
  else
    SetupNormalFont();
}
~FreeTypeFace() {
  if (face_)
    FT_Done_Face(face_);
}
FreeTypeFace(FreeTypeFace&& rhs)
    : font_file_(rhs.font_file_)
    , options_(rhs.options_)
    , face_(rhs.face_)
    , error_(rhs.error_)
{
  rhs.face_ = nullptr;
}
bool CalculateBox(uint32_t codepoint, uint32_t& width, uint32_t& height) {
  if (!RenderGlyph(codepoint))
    return false;
  width += (face_->glyph->advance.x >> 6);
  height = std::max(
      height, static_cast<uint32_t>(face_->glyph->metrics.height >> 6));
  return true;
}
bool DrawCodepoint(DrawContext& context, uint32_t codepoint) {
  if (!RenderGlyph(codepoint))
    return false;
  printf("U+%08X -> %s\n", codepoint, font_file_.c_str());
  return DrawBitmap(context, face_->glyph);
}
int Error() const { return error_; }

private:
FreeTypeFace(const FreeTypeFace&) = delete;
FreeTypeFace& operator=(const FreeTypeFace&) = delete;

bool RenderGlyph(uint32_t codepoint) {
  if (!face_)
    return false;
  uint32_t glyph_index = FT_Get_Char_Index(face_, codepoint);
  if (glyph_index == 0)
    return false;
  error_ = FT_Load_Glyph(face_, glyph_index, options_.load_flags);
  if (error_)
    return false;
  error_ = FT_Render_Glyph(face_->glyph, options_.render_mode);
  if (error_)
    return false;
  return true;
}
bool IsColorEmojiFont() {
  static const uint32_t tag = FT_MAKE_TAG('C', 'B', 'D', 'T');
  unsigned long length = 0;
  FT_Load_Sfnt_Table(face_, tag, 0, nullptr, &length);
  if (length) {
    std::cout << font_file_ << " is color font" << std::endl;
    return true;
  }
  return false;
}
```

```
  }
  void SetupNormalFont() {
    error_ = FT_Set_Pixel_Sizes(face_, 0, options_.pixel_size);
  }
  void SetupColorFont() {
    options_.load_flags |= FT_LOAD_COLOR;

    if (face_->num_fixed_sizes == 0)
      return;
    int best_match = 0;
    int diff = std::abs(options_.pixel_size - face_->available_sizes[0].width);
    for (int i = 1; i < face_->num_fixed_sizes; ++i) {
      int ndiff =
          std::abs(options_.pixel_size - face_->available_sizes[i].width);
      if (ndiff < diff) {
        best_match = i;
        diff = ndiff;
      }
    }
    error_ = FT_Select_Size(face_, best_match);
  }
  bool DrawBitmap(DrawContext& context, FT_GlyphSlot slot) {
    int pixel_mode = slot->bitmap.pixel_mode;
    if (pixel_mode == FT_PIXEL_MODE_BGRA)
      DrawColorBitmap(context, slot);
    else
      DrawNormalBitmap(context, slot);
    context.Advance(slot->advance.x >> 6);
    return true;
  }
  void DrawColorBitmap(DrawContext& context, FT_GlyphSlot slot) {
    uint8_t* src = slot->bitmap.buffer;
    // FIXME: Should use metrics for drawing. (e.g. calculate baseline)
    int yoffset = context.Height() - slot->bitmap.rows;
    for (int y = 0; y < slot->bitmap.rows; ++y) {
      uint8_t* dest = context.GetDrawPosition(y + yoffset);
      for (int x = 0; x < slot->bitmap.width; ++x) {
        uint8_t b = *src++, g = *src++, r = *src++, a = *src++;
        *dest++ = r; *dest++ = g; *dest++ = b; *dest++ = a;
      }
    }
  }
  void DrawNormalBitmap(DrawContext& context, FT_GlyphSlot slot) {
    uint8_t* src = slot->bitmap.buffer;
    // FIXME: Same as DrawColorBitmap()
    int yoffset = context.Height() - slot->bitmap.rows;
    for (int y = 0; y < slot->bitmap.rows; ++y) {
      uint8_t* dest = context.GetDrawPosition(y + yoffset);
      for (int x = 0; x < slot->bitmap.width; ++x) {
        *dest++ = 255 - *src;
        *dest++ = 255 - *src;
        *dest++ = 255 - *src;
        *dest++ = *src; // Alpha
        ++src;
      }
    }
  }

  std::string font_file_;
  FaceOptions options_;
  FT_Face face_;
  int error_;
};

class FontList {
  typedef std::vector<std::unique_ptr<FreeTypeFace>> FaceList;
 public:
  FontList() {}

  void AddFont(const std::string& font_file) {
    auto face = std::unique_ptr<FreeTypeFace>(new FreeTypeFace(font_file));
```

```
    face_list_.push_back(std::move(face));
  }
  void CalculateBox(uint32_t codepoint, uint32_t& width, uint32_t& height) {
    static const uint32_t kSpace = 0x20;
    if (codepoint == kSpace) {
      width += kSpaceWidth;
    } else {
      for (auto& face : face_list_) {
        if (face->CalculateBox(codepoint, width, height))
          return;
      }
    }
  }
  void DrawCodepoint(DrawContext& context, uint32_t codepoint) {
    for (auto& face : face_list_) {
      if (face->DrawCodepoint(context, codepoint))
        return;
    }
    std::cerr << "Missing glyph for codepoint: " << codepoint << std::endl;
  }

 private:
  FontList(const FontList&) = delete;
  FontList& operator=(const FontList&) = delete;
  FaceList face_list_;
};

class PngWriter {
 public:
  PngWriter(const std::string& outfile)
      : outfile_(outfile), png_(nullptr), info_(nullptr)
  {
    fp_ = fopen(outfile_.c_str(), "wb");
    if (!fp_) {
      std::cerr << "Failed to open: " << outfile_ << std::endl;
      Cleanup();
      return;
    }
    png_ = png_create_write_struct(
        PNG_LIBPNG_VER_STRING, nullptr, nullptr, nullptr);
    if (!png_) {
      std::cerr << "Failed to create PNG file" << std::endl;
      Cleanup();
      return;
    }
    info_ = png_create_info_struct(png_);
    if (!info_) {
      std::cerr << "Failed to create PNG file" << std::endl;
      Cleanup();
      return;
    }
  }
  ~PngWriter() { Cleanup(); }
  bool Write(uint8_t* rgba, int width, int height) {
    static const int kDepth = 8;
    if (!png_) {
      std::cerr << "Writer is not initialized" << std::endl;
      return false;
    }
    if (setjmp(png_jmpbuf(png_))) {
      std::cerr << "Failed to write PNG" << std::endl;
      Cleanup();
      return false;
    }
    png_set_IHDR(png_, info_, width, height, kDepth,
                 PNG_COLOR_TYPE_RGB_ALPHA, PNG_INTERLACE_NONE,
                 PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);
    png_init_io(png_, fp_);
    png_byte** row_pointers =
        static_cast<png_byte**>(png_malloc(png_, height * sizeof(png_byte*)));
```

```
    uint8_t* src = rgba;
    for (int y = 0; y < height; ++y) {
      png_byte* row =
          static_cast<png_byte*>(png_malloc(png_, width * kBytesPerPixel));
      row_pointers[y] = row;
      for (int x = 0; x < width; ++x) {
        *row++ = *src++;
        *row++ = *src++;
        *row++ = *src++;
        *row++ = *src++;
      }
      assert(row - row_pointers[y] == width * kBytesPerPixel);
    }
    assert(src - rgba == width * height * kBytesPerPixel);
    png_set_rows(png_, info_, row_pointers);
    png_write_png(png_, info_, PNG_TRANSFORM_IDENTITY, 0);
    for (int y = 0; y < height; y++)
      png_free(png_, row_pointers[y]);
    png_free(png_, row_pointers);
    Cleanup();
    return true;
  }
 private:
  PngWriter(const PngWriter&) = delete;
  PngWriter operator=(const PngWriter&) = delete;
  void Cleanup() {
    if (fp_) { fclose(fp_); }
    if (png_) png_destroy_write_struct(&png_, &info_);
    fp_ = nullptr; png_ = nullptr; info_ = nullptr;
  }

  std::string outfile_;
  FILE* fp_;
  png_structp png_;
  png_infop info_;
  char* rgba_;
  uint32_t width_;
  uint32_t height_;
};

class App {
 public:
  void AddFont(const std::string& font_file) { font_list_.AddFont(font_file); }
  bool SetText(const char* text) { return UTF8ToCodepoint(text); }
  bool Execute() {
    CalculateImageSize();
    Draw();
    return Output();
  }
 private:
  bool UTF8ToCodepoint(const char* text) {
    int32_t i = 0, length = strlen(text), c;
    while (i < length) {
      U8_NEXT(text, i, length, c);
      if (c < 0) {
        std::cerr << "Invalid input text" << std::endl;
        return false;
      }
      codepoints_.push_back(c);
    }
    return true;
  }
  void CalculateImageSize() {
    uint32_t width = 0, height = 0;
    for (auto c : codepoints_)
      font_list_.CalculateBox(c, width, height);
    printf("width: %u, height: %u\n", width, height);
    draw_context_.SetSize(width, height);
  }
  void Draw() {
```

```
    for (auto c : codepoints_)
      font_list_.DrawCodepoint(draw_context_, c);
  }
  bool Output() {
    PngWriter writer(kDefaultOutputFile);
    return writer.Write(draw_context_.Bitmap(),
                        draw_context_.Width(),
                        draw_context_.Height());
  }

  std::vector<uint32_t> codepoints_;
  FontList font_list_;
  DrawContext draw_context_;
};

bool Init() {
  int error = FT_Init_FreeType(&gFtLibrary);
  if (error) {
    std::cerr << "Failed to initialize freetype" << std::endl;
    return error;
  }
  return error == 0;
}

void Finish() {
  FT_Done_FreeType(gFtLibrary);
}

void Usage() {
  std::cout
      << "Usage: clfontpng font1.ttf [font2.ttf ...] text"
      << std::endl;
  std::exit(1);
}

bool ParseArgs(App& app, int argc, char** argv) {
  if (argc < 2)
    return false;
  for (int i = 1; i < argc - 1; ++i)
    app.AddFont(argv[i]);
  return app.SetText(argv[argc - 1]);
}

bool Start(int argc, char** argv) {
  App app;
  if (!ParseArgs(app, argc, argv))
    Usage();
  return app.Execute();
}

} // namespace

int main(int argc, char** argv) {
  if (!Init())
    std::exit(1);
  bool success = Start(argc, argv);
  Finish();
  return success ? 0 : 1;
}
```

Let's try the program on a simple Unicode emoji using the font *Noto Color Emoji*. This font, developed by Google, is available by default on Ubuntu Desktop but not on Ubuntu Server, so we need to install it first:

```
$ sudo apt install fonts-noto-color-emoji
$ sudo apt install fontconfig  # Install the command fc-list
$ fc-list  # List of the fonts on your system
```

```
/usr/share/fonts/truetype/dejavu/DejaVuSerif-Bold.ttf: DejaVu Serif:style=Bold
/usr/share/fonts/truetype/dejavu/DejaVuSansMono.ttf: DejaVu Sans Mono:style=Book
/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf: DejaVu Sans:style=Book
/usr/share/fonts/truetype/dejavu/DejaVuSans-Bold.ttf: DejaVu Sans:style=Bold
/usr/share/fonts/truetype/dejavu/DejaVuSansMono-Bold.ttf: DejaVu Sans Mono:style=Bold
/usr/share/fonts/truetype/noto/NotoColorEmoji.ttf: Noto Color Emoji:style=Regular
/usr/share/fonts/truetype/dejavu/DejaVuSerif.ttf: DejaVu Serif:style=Book
```

Then:

```
$ ./clfontpng /usr/share/fonts/truetype/noto/NotoColorEmoji.ttf 🔫
/usr/share/fonts/truetype/noto/NotoColorEmoji.ttf is color font
width: 136, height: 128
U+0001F52B -> /usr/share/fonts/truetype/noto/NotoColorEmoji.ttf
```

The emoji is correctly rendered:

This pistol has not always being a toy. If I rerun the same command using a previous version of the font:

```
root@vagrant:/vagrant# ./clfontpng NotoColorEmoji-v2017.ttf 🔫
NotoColorEmoji-v2017.ttf is color font
width: 136, height: 128
U+0001F52B -> NotoColorEmoji-v2017.ttf
```

The pistol is now a weapon:

The explanation dates back to 2016, when Apple announced that in iOS 10, the pistol emoji (U+1F52B 🔫) would be changed from a real revolver to a water pistol. At the same time, Microsoft decided the pistol emoji would be changed from a toy ray-gun to a real revolver to be more in line with industry-standard designs... Finally, in 2018, most platforms such as Google, Microsoft, Samsung, Facebook, and Twitter had transitioned their rendering of the pistol emoji to match Apple's water gun implementation, which means that during two years, the pistol could be understood as a joke or as a threat, depending on if the sender was running on Android or iOS. This is not the only example of controversial emojis.

Let's now try a font that does not support emojis:

```
$ ./clfontpng SourceSansPro-Bold.ttf 🔫
width: 0, height: 0
Missing glyph for codepoint: 128299
libpng warning: Image width is zero in IHDR
libpng warning: Image height is zero in IHDR
```

```
libpng error: Invalid IHDR data
Failed to write PNG
```

The program correctly reports the missing glyph. Now, let's try to combine several fonts to simulate the font fallback mechanism:

```
$ ./clfontpng SourceSansPro-Bold.ttf NotoColorEmoji.ttf ħéllO🎉
NotoColorEmoji.ttf is color font
width: 460, height: 128
U+00000452 -> SourceSansPro-Bold.ttf
U+00001F73 -> SourceSansPro-Bold.ttf
U+0000006C -> SourceSansPro-Bold.ttf
U+0000004C -> SourceSansPro-Bold.ttf
U+000001A0 -> SourceSansPro-Bold.ttf
U+0001F389 -> NotoColorEmoji.ttf
```

The output reveals that glyphs from different font files are used to render the final text:

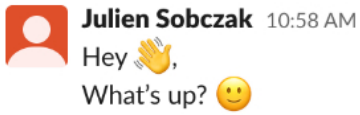
If you are curious about this text rendering process, you can inspect the Chromium rendering engine or read the documentation of the project HarfBuff.

EXAMPLE: SLACK EMOJIS

Slack and emojis are inseparable. Emojis make long messages less boring and are indispensable to communicate your mood so that others can interpret your messages correctly. Slack even support custom emojis. How does Slack support so many different emojis? It's simple. They are not Unicode emojis.

Indeed, Unicode is an evolving standard. Slack cannot wait for Unicode to approve your custom emoji during the next meeting. Slack cannot either use the private-use range of characters as it would mean regenerating new font files with your emojis and making sure your recipient receives the new font too. So, Slack is using images, even for standard Unicode characters when using the Slack web client (ex: the image 🔫 represents the `PISTOL EMOJI` 🔫 U+1F52B).

Using images instead of Unicode characters is not problematic when the same application is used on both sides. When you are sending a message on Slack Desktop, you know that your recipient will check it using one of the clients supported by Slack. Therefore, it is easy for Slack to make sure that the images for the emojis are available everywhere. The problem occurs when you try to copy/paste some text from Slack to a different application. In this case, what Slack is doing is replacing these images with textual representation like `:slightly_smiling_face:`:

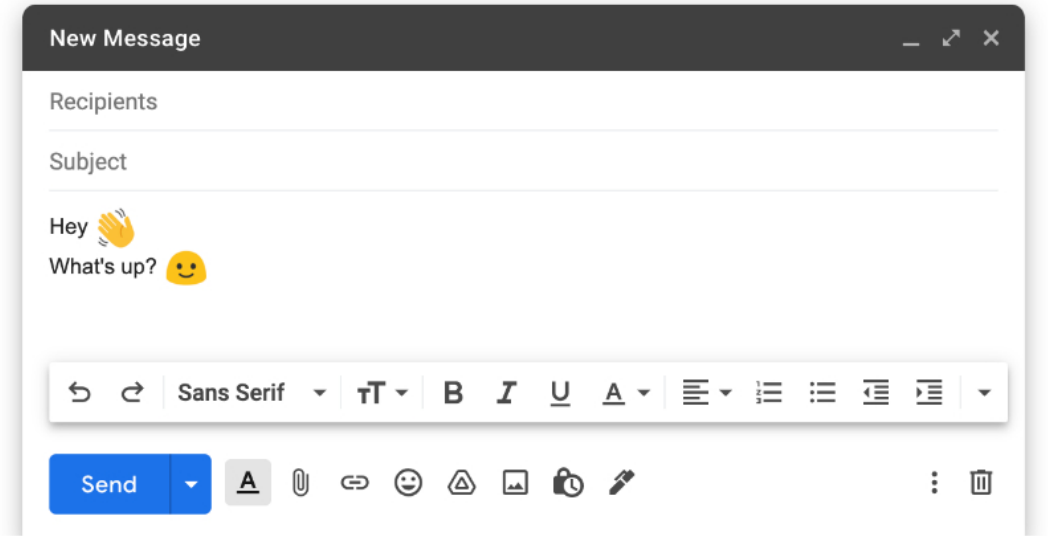


When copied into a different application:

```
Hey :wave:,
What's up? :slightly_smiling_face:
```

EXAMPLE: GMAIL

Gmail is using the same approach as Slack. As a web application, Gmail cannot be sure of the fonts available on every device, and therefore, Gmail relies on images too so that the recipient can see the email as the sender viewed it.



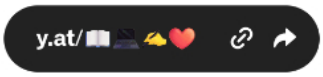
The HTLM source:

```
<div id=":hb"
     class="Am Al editable LW-avf tS-tW"
     hidefocus="true"
     aria-label="Message Body"
     g_editable="true"
     role="textbox"
     aria-multiline="true"
     contenteditable="true"
     tabindex="1"
     style="direction: ltr; min-height: 204px;"
     itacorner="6,7:1,1,0,0"
     spellcheck="false">
  Hey 
  <img src="//ssl.gstatic.com/mail/emoji/v7/png48/emoji_u1f44b.png"
       alt="@"
       goomoji="1f44b"
       data-goomoji="1f44b"
       style="vertical-align: middle; height: 24px; width: 24px;">
  <div>What's up? 
    <img src="//ssl.gstatic.com/mail/emoji/v7/png48/emoji_u1f642.png"
       alt="@"
       goomoji="1f642"
       data-goomoji="1f642"
       style="vertical-align: middle; height: 24px; width: 24px;">
    </div>
</div>
```

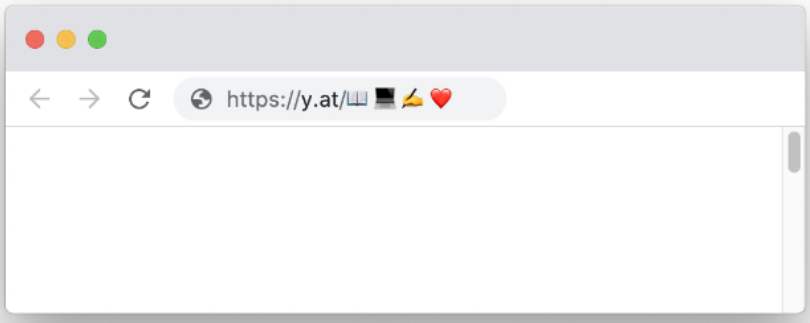
EXAMPLE: YAT

YAT allows you to generate a unique chain of emoji characters to be used as your universal identifier on the Internet. Interoperability is thus important. Your YAT name will be used on many other sites, and YAT is restricted by the emojis defined by Unicode.

After creating your YAT account, YAT offers you a unique YAT URL to redirect to your profile using your emoji chain in the path:



When copying this URL to my browser:



When pressing the Enter key, the URL becomes:

https://y.at/%F0%9F%93%96%F0%9F%92%BB%E2%9C%8D%EF%B8%8F%E2%9D%A4%EF%B8%8F

The fact is a URI must be composed of a limited set of ASCII characters consisting of digits, letters, and a few graphic symbols—some unreserved (`-`, `.`, `_`, `~`), some reserved (`:`, `/`, `?`, `#`, ...) as used as delimiters. Emojis are not directly supported. For these characters, the URI RFC defines a percent-encoding mechanism.

A percent-encoded octet is encoded as a character triplet, consisting of the percent character `%` followed by the two hexadecimal digits representing that octet's numeric value. For example, `%20` is the percent-encoding for the binary octet `00100000`. And `%F0%9F%93%96` is the encoding for the binary `11110000 10011111 10010011 10010110`, which is the UTF-8 representation (`F0 9F 95 AE`) for the book emoji U+1F56E.

If you navigate on the YAT website, you will also see a ton of beautiful emojis displayed in large definition. These emojis are not Unicode emojis but hand-crafted illustrations in PNG format that you can also browse online.

Figure 23. Shocked Face With Exploding Head Emoji U+1F92F rendered using the YAT artwork

Figure 24. Shocked Face With Exploding Head Emoji U+1F92F rendered using the HTML character

The Unicode character does not look good when displayed as a large text. The reason is most fonts like *Noto Color Emoji* or *Apple Color Emoji* are distributed with bitmap emoji glyphs. The OpenType-SVG specification adds support for vector-based glyphs using SVG. Font files will be larger as a consequence but the rendering will not be affected when scaling a glyph. The support is still partial, but Adobe is already supporting an OpenType-SVG font for Noto Color Emoji. Here is the rendering in Chrome:

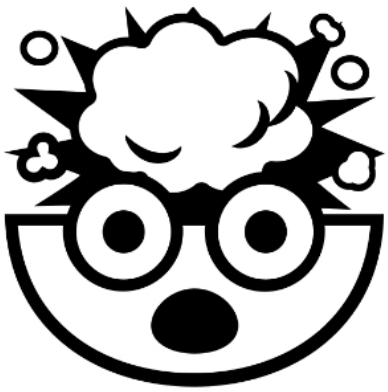


Figure 25. Shocked Face With Exploding Head Emoji U+1F92F rendered using an OpenType-SVG font

The output is still not ideal (colors are missing), but we observe that the browser enlarges the glyph like any vector illustration.

This ends the part about the implementation of Unicode.

THE FUTURE

THE FUTURE

Unicode provides a unique code for every character, in every program, on every platform. Unicode exists to preserve the world's heritage. But with approximately 6,000 languages spoken in the world today, there are still many characters, writing systems to encode, and many new emojis to make Unicode even more popular.

Unicode is widely supported. Modern operating systems, programming languages, and software applications have support for Unicode, but the devil is in the details. Unicode support does not mean you can ignore Unicode completely. You need to understand what Unicode is and how your programming language implements it to create truly multilingual applications.

To Go Further

The scope of Unicode is wider than simply assigning code points to characters. Unicode gives programmers a vast amount of data about the handling of text:

- How to divide words and break lines,
- How to sort text,
- How to format numbers, dates, times, and other elements appropriate to different locales,
- How to display text for languages whose written form flows from right to left, such as Arabic or Hebrew,
- How to deal with security concerns regarding the many look-alike characters from writing systems around the world.

---

1. The motivation for ASCII to save bytes was not really new. The problem occurred with the publication of the first books. Gutenberg used 250 characters to produce in 1455 the 42-line Bible, the first printed book. Then, character sets became smaller and smaller in successive fonts, to reduce the costs of cutting, founding, composing, and distributing type.

2. Nowadays, a byte is always considered to be 8 bits but first computers did use different sizes for a byte, which was based on the size of information it needed to hold.

3. In the document *Unicode 88*, the authors estimate the total number of characters to be less than 2^{14} = 16,384, based on the union of all newspapers and magazines printed in the world in 1988.

4. Yat was created recently with the ambitious goal to create a new censorship resistant internet identity system using a personalized string of emojis as your universal username.

5. Unicode includes more than 10,000 characters that compromise the design principles behind the Standard, like unification or dynamic composition.

6. Rob Pike and Ken Thompson discuss the benefits of using many little fonts compared to one big font during the implementation of the Plan 9 operating system. It breaks the huge Unicode codespace into manageable components

promoting sharing. For example, you can have only one font with the set of Japanese characters but dozens of fonts for Latin characters.

About the author

Julien Sobczak works as a software developer for Scaleway, a French cloud provider. He is a passionate reader who likes to see the world differently to measure the extent of his ignorance. His main areas of interest are productivity (doing less and better), human potential, and everything that contributes in being a better person (including a better dad and a better developer).

READ FULL PROFILE

TAGS

Architecture 13 | Classics 3 | Craftsmanship 11 | Computer Science 9 | Data 6 | Devops 11

Frameworks 10 | Human 11 | Languages 6 | Learning 31 | Management 28 | Productivity 16

Tools 9

LOCATION

Lille
France

AROUND THE WEB

Opinions are my own and don't reflect the views of my employer.

In addition, as anybody with an open mind, my opinions are likely to change from time to time...