Engineer

Unicode

⌨ the standard that gives every character a number of its own

What it is

Unicode is the standard that gives every letter, digit, punctuation mark and symbol of the world's writing systems a number of its own. Its maintainers call it the universal character encoding: built so that text in the languages and technical fields of the modern world can be exchanged, processed and displayed anywhere, and also able to represent older scripts and historical writing. One repertoire holds the Danish å, the Ukrainian ґ, the euro sign and the letters of scripts nobody has written for centuries, so a single document, or a single database column, can contain all of them.

Strictly speaking, a Unicode release includes several coordinated parts: its core specification, charts of characters, the annexes and a Character Database, the machine-readable files that record the properties of every character. The Unicode Technical Committee maintains it continuously, the Unicode Consortium publishes it, and version 18.0 was announced on 16 September 2026.

The number a character receives is its code point. The codespace runs from 0 to 10FFFF in hexadecimal, 1,114,112 code points in all, and a code point is written as U+ followed by at least four hex digits: å is U+00E5, ґ is U+0490 and the euro sign is U+20AC. The Consortium's stability policy promises that an encoded character keeps its place for good: it is never moved or taken out, and its name never changes, so text encoded today means the same thing in every later version.

Two distinctions carry most of the standard. The first is between a character and a glyph. Unicode encodes the character, the abstract letter, and leaves its shape to the font: the core specification states that glyph shapes are the responsibility of font vendors and not part of the standard. The same code point can look different in two typefaces, and a typeface with no shape for it cannot draw it at

all. The second is between a code point and the bytes that store it. The three Unicode encoding forms are UTF-8, UTF-16 and UTF-32, which write each code point as one or more units of 8, 16 or 32 bits, and the specification calls it a trap to try to distinguish “UTF-8 versus Unicode”: all three are equally valid ways to implement the same characters.

Use the concept in a decision

Put “What is Unicode?” to work in a real decision. We can help relate the concept to the system you are building or buying. Software that reads naturally in every market you sell in, including the dates, the currencies and the way names sort.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Built an English/Danish i18n system — 12,027 messages a locale, in English and Danish

Built an English/Danish i18n system Get in touch →

Why it matters

Before Unicode, each script had its own code pages, and a byte meant nothing until you knew which page was in use. In the Western European code page windows-1252 the byte E5 is å; in the Cyrillic windows-1251 the same byte is the letter е. Neither page could hold the other’s letters: windows-1251 has Ґ and no å, æ or ø, and windows-1252 has the reverse. A product in Danish and Ukrainian needed two encodings and a reliable record of which text used which, and a sentence that quoted both languages fit in neither encoding.

Unicode removed that bookkeeping. Every language a product publishes, and every name a customer types into a form, fits one character set, and the same number means the same character on every system that reads it. That is what lets a database, an API and a browser pass text between them without first agreeing on a code page.

The standard also settles questions that every program would otherwise settle its own way. It unifies characters across languages: the comma is one character whether it separates thousands, as in English, or decimals, as in French, so the meaning comes from the locale and not from the code point. It does not unify across scripts: the Latin o, the Cyrillic о and the Greek ο look the same in most typefaces and are three characters, because they belong to three scripts and older encodings already kept them apart. And it records which sequences are equivalent. Å can be stored as one precomposed character or as A followed by a

combining ring, and the standard defines the two as canonically equivalent, to be displayed and interpreted the same way, with normalization forms that turn either into one agreed spelling.

That generality has a price, and the programmer pays it. A character is not a byte, not always one unit of a string and not always what a reader counts as one letter. Code that treats the three as the same passes its tests on English data and fails on the names, addresses and messages of the people it was written for.

What it looks like in practice

JavaScript makes the layers visible. A string is a sequence of UTF-16 code units, and spreading it into an array splits it into code points:

```
const hex = (n) => n.toString(16).toUpperCase().padStart(4, "0");
for (const text of ["a", "å", "Ґ", "€", "𐌂"]) {
  const points = [...text].map((c) => "U+" + hex(c.codePointAt(0)));
  console.log(text, points.join(" "), text.length, [...text].length);
}
```

Run in Node.js 25, it prints each character, its code point, its length in UTF-16 units and its length in code points:

```
a U+0061 1 1
å U+00E5 1 1
Ґ U+0490 1 1
€ U+20AC 1 1
𐌂 U+10302 2 1
```

The last line is an Old Italic letter. Its code point lies beyond U+FFFF, so UTF-16 stores it as two units, a surrogate pair, and `length` reports 2 for what is one character.

Equivalence shows up the same way. Here are three strings that all display as Å:

```
const forms = ["\u00C5", "A\u030A", "\u212B"];
console.log("form length equal NFC");
for (const [i, s] of forms.entries()) {
```

```
  console.log(i + 1, s.length, s === forms[0], s.normalize("NFC") === forms[0]);
}
```

```
form length equal NFC
1 1 true true
2 2 false true
3 1 false true
```

The first is the precomposed letter, U+00C5. The second is a plain A followed by U+030A, the combining ring above. The third is U+212B, the angstrom sign, which the Unicode Character Database maps to U+00C5. Compared as they are, they are three different strings; after normalization to NFC, all three are the same one. Nothing on screen tells them apart, so a search, a unique index or a login that compares the raw strings sees three different names.

This site depends on code points in a less visible place. The note about the underline that crossed every letter describes the site's Latin and Cyrillic Ubuntu body fonts as six cuts, three weights of Latin and three of Cyrillic, so that a Ukrainian page is set in the same typeface as an English one. Each cut declares the range of characters it covers. Those ranges guide font selection for the characters on the page; an explicit preload can also request a file before normal font discovery. The Cyrillic declaration states its range in code points:

```
unicode-range: U+0400-045F, U+0490-0491;
```

The first range runs from È to џ and holds the Ukrainian Є, І and Ї along with the rest of the basic Cyrillic alphabet. The second holds only Ґ and ґ, which sit further along in the Cyrillic block. The ranges declare which face is eligible for a character, but the font must also contain its glyph. They do not guarantee that a page fetches files from only one script. English and Danish text uses the Latin faces, but the Ukrainian language label can need a Cyrillic face, and the site explicitly preloads the light Cyrillic body cut on non-Cyrillic pages for screen use.

Where it goes wrong

Length is counted in code units. JavaScript's `length` counts UTF-16 units, so 𝄞 counts as two, and an A with a combining ring counts as two where a reader sees one letter. A limit enforced in units rejects text that fits, and a cut at a fixed

position can split a character: `"A\u030A".slice(0, 1)` keeps the A and drops the ring, and cutting 𐐷 in half leaves a lone surrogate that the UTF-8 encoder replaces with U+FFFD, the replacement character. Decide what each limit counts: bytes where storage sets it, characters as a reader sees them where a person does. `Intl.Segmenter` counts the latter, and for Århus written with a combining ring it returns 5 where `length` returns 6. Cut text only at the boundaries it reports.

Equal text is compared unit by unit. The three forms of Å above are three different strings to `===`, to a hash and to a unique index, so a customer can register the same name twice, or fail to find a document by the title they typed. Normalize text to NFC where it enters the system, store it normalized, and compare normalized forms. The stability policy guarantees that a string normalized under one version stays normalized under every later one, provided the string contains only characters that version knows, so the normalizing code has to be as current as the text.

Look-alike characters pass as the same. The Latin o, the Cyrillic о and the Greek ο are different code points that most typefaces draw alike, so two account names or two product codes can look identical and differ in a single character. Unicode keeps them apart on purpose, and its standard on security mechanisms, UTS #39, sets out how to reduce that kind of spoofing. Where a name identifies a person or a payee, allow one script per name, or test it against the standard’s security data before accepting it.

Bytes are read in the wrong encoding. Text stored as UTF-8 and read as windows-1252 turns blåbær into blÃ¥bÃ¦r, because each two-byte letter is read as two one-byte ones. The word Unicode does not say how the bytes were written; only the encoding form does. Declare it wherever text crosses a boundary, in the HTTP header, in the page, on the database connection and in the file format, and treat a file that arrives without one as a question for a person rather than a guess for the code.

A missing glyph is blamed on the text. When a letter shows up as an empty box, the usual cause is a font with no shape for that code point, and the text itself is intact. Because Unicode leaves shapes to fonts, every typeface a product uses has to cover every character its languages need, Є, І, Ї and Ґ as well as A to я, or the browser draws them in whatever fallback it finds. The note above gives this as the reason the site serves Cyrillic at all: without it, a Ukrainian page would fall back to whatever the system offers. Test each language with its whole alphabet rather than a sample sentence.

Names are checked against A to Z. A pattern such as `^[A-Za-z' -]+$` accepts O'Neill and rejects Søren Ågård, Zoë and Ґалаґан, the very names a Danish or Ukrainian product exists to serve. Test for letters by their Unicode properties instead: in JavaScript, `/^[\p{L}\p{M}' -]+$/u` accepts all four. Those properties come from the copy of the Unicode Character Database inside the runtime. For example, the Node.js 25.2.1 runtime used for the examples here reports Unicode 16.0; that is a specific build, not a guarantee for every 25.x release. Letters added in later Unicode versions fail the test until the runtime's character data is upgraded. Validate names loosely, and let a person rather than a pattern decide the hard cases.

Further reading

- About the Unicode Standard — the Consortium's own description of the standard, what makes up a version, and where the latest one is published.
- The Unicode Standard, Chapter 2: General Structure — the ten design principles, the codespace, the three encoding forms and the rules for equivalent sequences.
- Unicode Character Encoding Stability Policies — what can never change once a character is encoded, from its code point and name to its normalization.
- Unicode 18.0.0 — what the current version adds, and the technical standards versioned with it, among them collation and security.
- Node.js: process.versions — the runtime's component-version report, used here to pin the Unicode-data example to one Node.js build rather than an entire release line.
- CSS Fonts: character ranges — how declared character ranges and the font's actual glyph coverage together determine which characters a face can render.



Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is Unicode?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/unicode/