

Unicode and Ü

... things a web developer **might** want to know

What is Unicode?

Defines a unique code for every character in every language

Irrelevant of platform, program, or implementation

Punctuation marks, diacritics, symbols, arrows, and emoji are also defined

Scripts include the European alphabetic scripts, Middle Eastern right-to-left scripts, and many scripts of Asia

**SUCH AS THE ENGLISH
LANGUAGE!**

ω	क	ƒ
U+0D27	U+0915	U+0284

Unicode represents a character in an abstract way

and leaves the visual rendering (size,
shape, font, or style) to other software,
such as a web browser



128,172

In all, the Unicode Standard, Version 9.0 provides codes for 128,172 characters from the world's alphabets, ideograph sets, and symbol collections.

More characters are usually added each release.

Example code block

0680	پ	خ	خ	ج	ج	خ	ج	ج	ڈ	د	د	ڈ	د	ڈ	د
0690	ڈ	ڑ	ڑ	ر	ر	ر	ر	ر	ڑ	ڑ	ہن	ہن	ص	ض	ظ
06A0	غ	ف	ف	ف	ف	پ	ق	ق	ق	ک	ک	ک	ک	ک	گ
06B0	گ	گ	گ	گ	گ	ل	ل	ل	ل	ن	ن	ن	ن	ن	ج
06C0	ف	ی	ی	ی	ی	و	و	و	و	و	و	و	و	و	و
06D0	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ
06E0	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ
06F0	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ	ہ

Arabic ▾

[Open in separate page](#) ↗

Range: **0600—06FF**

Number of characters: 256

type: alphabet

Languages: arabic, persian, kurd



Source: unicode-table.com

Example code block

25A0	■	□	◻	▣	▤	▥	▦	▧	▨	▩	▪	▫	▬	▭	▮	▯
25B0	▰	▱	▴	▵	▶	▷	▸	▹	►	▻	▼	▽	▾	▿	▿	▿
25C0	◀	◁	◂	◃	◄	◅	◆	◇	◈	◉	◊	◯	⊖	⊗	⦿	●
25D0	◐	◑	◒	◓	◔	◕	◖	◗	◘	◙	◚	◛	◜	◝	◞	◟
25E0	◠	◡	◢	◣	◤	◥	◦	◧	◨	◩	◪	◫	◬	◭	◮	◯
25F0	◰	◱	◲	◳	◴	◵	◶	◷	◸	◹	◺	◻	◼	◽	◾	◿

Geometric Shapes ▾


[Open in separate page](#) ↗

Range: [25A0](#)—[25FF](#)

Number of characters: 96



Example character



Radioactive Sign U+2622

Technical information

Unicode number	U+2622
HTML-code	☢
Block	Miscellaneous Symbols
Set	Most popular characters

ASCII & FRIENDS

A bR1ef h1st0ry in time before unicode ruled



Before the web ruined everything



Life was simple - we usually didn't care about other languages

We had ASCII and if we needed some fancy characters we'd use ISO-8859-1 or ANSI. Other characters? Other codepages!

Each character could fit nicely into 1 byte

No complex encoding, or mapping, a string is just a series of bytes which contain characters

Japanese would be Japan's problem, not ours...



ASCII

7 bits = 128 glorious characters used most frequently in US-English

We didn't need no british pound symbol

Character operations are simple. Uppercase? Just switch the 6th bit (genius)

1 byte mapping makes memory access simple

Emoji? We call them emoticons and you can combine them in ways unimaginable



ASCII

Character	Decimal	Binary	Hex
A	65	01000001	0x41
a	97	01100001	0x61
9	57	00111001	0x39

ASCII

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

ISO-8859-1 (aka Latin1, CP819, IBM819 ...)

Add 1 bit to ASCII and you get an 8-bit, single-byte encoding that provides space for 128 additional characters

The british pound symbol is finally in! Plus 95 other symbols

Operations are still simple - working with a 1 byte to 1 character mapping

It's not quite the same as Windows code page 1252 - more on that later



As of August 2016, 6.0% of all web sites claim to use ISO 8859-1

Additional characters from ISO-8859-1

NBSP	ı	ç	£	¤	¥		§	¨	©	ª	«	¬	SHY	®	¯
00A0	00A1	00A2	00A3	00A4	00A5	00A6	00A7	00A8	00A9	00AA	00AB	00AC	00AD	00AE	00AF
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
00B0	00B1	00B2	00B3	00B4	00B5	00B6	00B7	00B8	00B9	00BA	00BB	00BC	00BD	00BE	00BF
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
00C0	00C1	00C2	00C3	00C4	00C5	00C6	00C7	00C8	00C9	00CA	00CB	00CC	00CD	00CE	00CF
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
00D0	00D1	00D2	00D3	00D4	00D5	00D6	00D7	00D8	00D9	00DA	00DB	00DC	00DD	00DE	00DF
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
00E0	00E1	00E2	00E3	00E4	00E5	00E6	00E7	00E8	00E9	00EA	00EB	00EC	00ED	00EE	00EF
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ
00F0	00F1	00F2	00F3	00F4	00F5	00F6	00F7	00F8	00F9	00FA	00FB	00FC	00FD	00FE	00FF
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

ISO-8859-?

Actually there are a bunch of ISO-8859 definitions, not just ISO-8859-1

They all swap out the same range of characters (decimal 160-255)

Latin-1 (Western European languages)

Latin-2 (Non-Cyrillic Central and Eastern European languages)

Latin-3 (Southern European languages and Esperanto)

Latin-5 (Turkish)

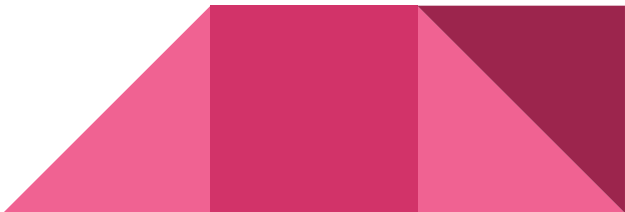
Latin-6 (Northern European and Baltic languages)

8859-5 (Cyrillic)

8859-6 (Arabic)

8859-7 (Greek)

8859-8 (Hebrew) ...





Windows Codepage 1252

Uses the unused range of ISO-8859-1 (DECIMAL 128-159) to provide additional characters

Sometimes referred to as “ANSI”

Gives you that *nice* dash that’s just a little bit longer than a plain old hyphen

1252 is basically the reason we needed to add “Paste from Word” to our web applications

 DEC: 150	 DEC: 45
---	--

Windows Codepage 1252

p 0070 112	q 0071 113	r 0072 114	s 0073 115	t 0074 116	u 0075 117	v 0076 118	w 0077 119	x 0078 120	y 0079 121	z 007A 122	{ 007B 123	 007C 124	} 007D 125	~ 007E 126	DEL 007F 127
€ 20AC 128		, 201A 130	f 0192 131	" 201E 132	... 2026 133	† 2020 134	‡ 2021 135	^ 02C6 136	‰ 2030 137	Š 0160 138	‘ 2039 139	Ⓔ 0152 140		Ž 017D 142	
	’ 2018 145	’ 2019 146	" 201C 147	" 201D 148	• 2022 149	— 2013 150	— 2014 151	~ 02DC 152	™ 2122 153	Š 0161 154	’ 203A 155	œ 0153 156		ž 017E 158	ÿ 0178 159
NBSP 00A0 160	í 00A1 161	ç 00A2 162	£ 00A3 163	¤ 00A4 164	¥ 00A5 165	 00A6 166	\$ 00A7 167	“ 00AB 168	© 00A9 169	à 00AA 170	« 00AB 171	¬ 00AC 172	SHY 00AD 173	® 00AE 174	— 00AF 175

Before Unicode

Before unicode there were 100's of different encoding systems - issues

No single encoding could contain enough characters (eg. European Union spans many encodings)

The encoding systems would also conflict with each other. Eg. Two different characters at the same address -OR- the same character defined in two different places

Unsuitable for East-Asian languages requiring 1000's codepoints

Does not cater for additional symbols



Back to Unicode

Unicode

So Unicode gives us a 21bit space **U+0000..U+10FFFF** to provide allocation for over a million codepoints

Unicode only specifies the codepoint or the “number” to reference that character, how that codepoint is represented on disk or in memory is a whole other story



The Unicode Space

Unicode allows for ~1.1million codepoints to be allocated across 17 “planes”

Each plane - 65,536 continuous characters (16 bits)

Basic Multilingual Plane (BMP) is the first plane and contains the character assignments for most of the modern languages and common symbols

There are 3 supplementary planes, plus private use blocks across multiple planes

The other planes are sometimes referred to as: *Astral Planes*



The Unicode Space

Plane 0	Basic Multilingual Plane (BMP)	U+0000 - U+FFFF
Plane 1	Supplementary Multilingual Plane (SMP)	U+10000 - U+1FFFF
Plane 2	Supplementary Ideographic Plane	U+20000 - U+2FFFF
Planes 3-13	Unassigned	U+30000 - U+DFFFF
Plane 14	Supplement-ary Special-purpose Plane (SSP)	U+E0000 – U+FFFFFF
Planes 15-16	Supplement-ary Private Use Area planes (SPUA)	U+F0000 + U+10FFFF

BMP - Basic Multilingual Plane

The first plane **plane 0** is called the Basic Multilingual Plane or BMP

It specifies the codepoint range **U+0000** -> **U+FFFF**

It contains all the most commonly used symbols, english scripts, and many modern languages

Most of the time you don't need any code points outside of the BMP especially for text documents in English. Just like any other Unicode plane, it groups about 65 thousand symbols



Encoding Formats

Unicode Transformation Format (UTF)

UTF specifies the encoding for a codepoint (eg. codepoint -> memory/storage)

UTF-8, UTF-16, and UTF-32 all provide different ways to encode a codepoint into between 1 and 4 bytes

Only these encodings are part of the Unicode Standard - but there are many more

All interchangeable - no loss when converting between each



UTF-8

Variable width encoding from 1-4 bytes

Designed to be compatible with ASCII - the first 128 characters correspond 1:1 with ASCII

Therefore ASCII is also completely valid UTF-8

UTF-8 does not require BOM although it is sometimes present

Most common on the web, HTML5 mandates its use



Bytes	Bits For code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

1-4 bytes - first byte for characters in the ASCII range

UTF-8 Binary Encoding

Working with UTF-8 at the Byte Level

You have a single byte character if the first bit is a **0 (zero)**

You know you have the first byte in a sequence if the first two bits are **11**

If you have a byte that starts with **10**, you will need to seek up to 3 bytes backwards to find the start of the sequence

With this knowledge we can randomly seek and read 1 byte anywhere in a UTF-8 encoded file and know how to proceed

UTF-8 encoding of ASCII characters

Character	Decimal	Binary	Hex
A U+0041	65	01000001	0x41
a U+0061	97	01100001	0x61
9 U+0039	57	00111001	0x39
Æ U+006C	50054	11000011 10000110	0xC3 0x86

UTF-16

Variable width encoding that uses either 16 bits or 32 bits for all codepoints

All of the characters in the Basic Multilingual Plane (BMP) are encoded as a single 16-bit unit - thus most characters of most modern languages only use one code unit

Other planes require two 16-bit units (32 bits) and we call this “surrogate pairs”

Used by Windows and Java for string/char storage



UTF-16

Characters U+0800 through U+FFFF use three bytes in UTF-8, but only two in UTF-16

As a result, text in (for example) Chinese, Japanese or Hindi will take more space in UTF-8 if there are more of these characters than there are ASCII characters

Comes in 2 forms UTF-16LE and UTF-16BE (Little Endian or Big Endian)

Not compatible with ASCII



UTF-32

32 bit (4 byte) encoding... 4 bytes for every character no matter which character

Fixed width makes codepoints directly indexable (Constant time operation)

Used by Linux and OSX for `w_char` (wide char) storage

Uses more memory than the other standards, but easier to address

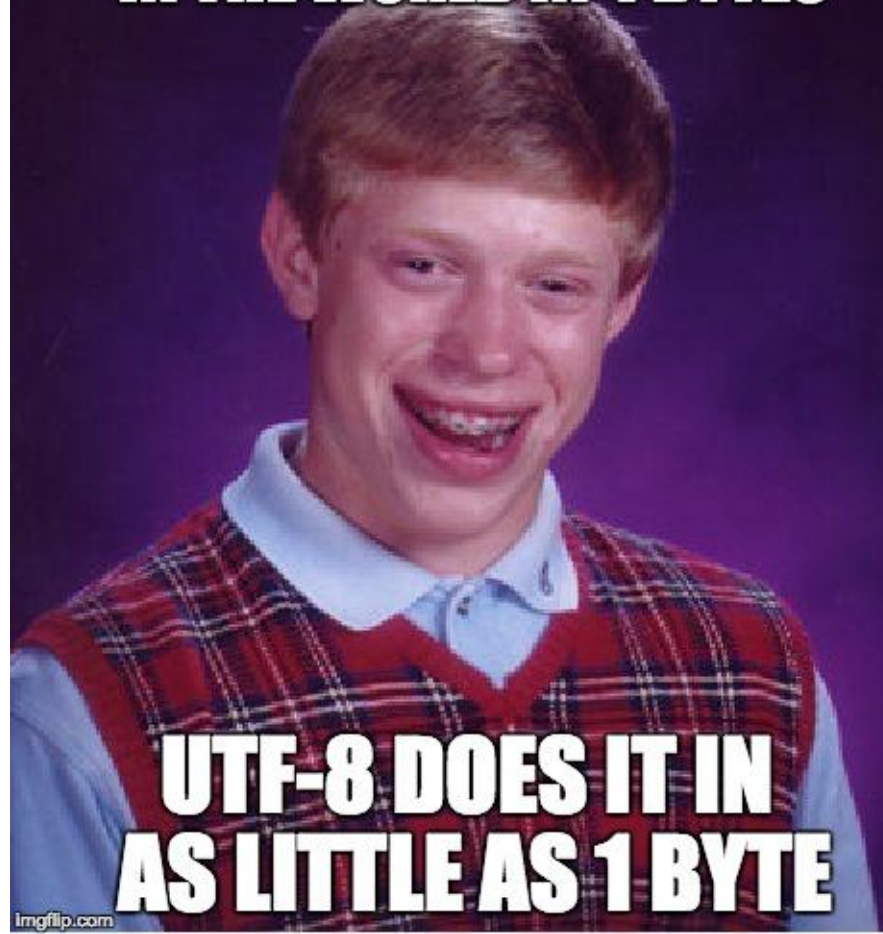
Comes in 2 forms UTF-32LE and UTF-32BE (Little Endian or Big Endian)

Not compatible with ASCII



Bad luck
UTF-32

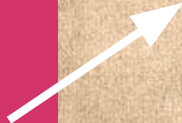
**CAN ENCODE EVERY CHARACTER
IN THE WORLD IN 4 BYTES**



**UTF-8 DOES IT IN
AS LITTLE AS 1 BYTE**

Little Endian Big Endian

NO,
THEY'RE NATIVE AMERICAN'S



Little Endian & Big Endian Variations

UTF-16 and UTF-32 may be in LE (Little Endian) or BE (Big Endian) encoding

Big Endian means most significant byte first (normal)

Little Endian means least significant byte first (reverse)

Some computers are better at one or the other

Byte order mark (BOM) “can” be used to indicate LE or BE
... or specify the encoding through some other means

You must know which order it is encoded in!



Recap


UTF-8 is most common on the web, and *potentially* uses the least storage (1 byte)

UTF-16 is either 2 or 4 bytes, used by Java and Windows, and uses less storage than UTF-8 when dealing with certain scripts. All characters in the BMP are 1 unit

UTF-8 and UTF-32 are used by Linux and various Unix systems

UTF-32 always uses 4 bytes but can be addressed more easily due to fixed width

The conversions between all of them are algorithmically based, fast and lossless



Most popular on the web			
ASCII	UTF-8	UTF-16	UTF-32
<p>1 byte</p> <p>Simple to use</p>	<p>1-4 bytes</p> <p>Flexible encoding length</p>	<p>2-4 bytes</p> <p>Better outside the BMP</p>	<p><i>Always</i> 4 bytes</p> <p>Good for internal storage</p>
Plain old text	✓ Single Encoding	✓ Supports LE & BE	✓ Supports LE & BE
✓ Constant Access	Variable Access	Variable Access	✓ Constant Access
✓ HTML Support	✓ HTML Support	✓ HTML Support	HTML Support
	✓ Recommended by W3C		
✓ Customise your extra bits			
Emoji	✓ Emoji	✓ Emoji	✓ Emoji
128 characters upgrade to iso-8859-1 available	1.1 million characters	1.1 million characters	1.1 million characters

Byte Order Mark (BOM)

Byte Order Mark (BOM) U+FEFF

An **optional** Unicode character which appears at the start of **some** text streams

Indicates that it a stream is in Unicode

Indicates the encoding (eg. UTF-8, UTF-16, UTF-32)

Indicates endianness (byte order)

If you specify your encoding through some other means then a BOM is not necessary

Sometimes causes issues



Byte Order Mark (BOM) U+FEFF

The exact bytes comprising the BOM will be whatever the Unicode character U+FEFF is converted into by that transformation format

The BOM can be used to “sniff” the format of the file or stream

If present in UTF-8 the BOM will always be the 3 bytes: EF BB BF

UTF-8 does not require a BOM and UTF-8 should always ignore the BOM

Bytes	Encoding Form
00 00 FE FF	UTF-32, big-endian
FF FE 00 00	UTF-32, little-endian
FE FF	UTF-16, big-endian
FF FE	UTF-16, little-endian
EF BB BF	UTF-8



Unicode Equivalence

And normalization and stuff ...

Unicode Equivalence

Unicode sometimes has multiple representations for the same character

Ultimately they all represent the “same” character, but with different codepoints - we call them equivalent

Sometimes these additional codepoints exist for historical reasons

Unicode provides rules around normalisation so that they can be transformed and/or treated as the same character (eg. when comparing or alphabetising)



For Example

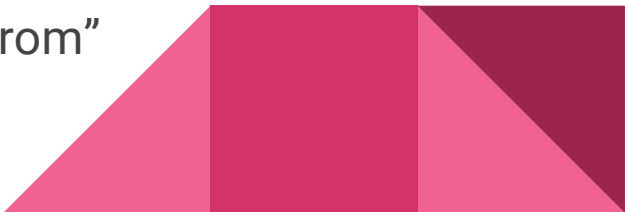
U+00C5 (Å) LATIN CAPITAL LETTER A WITH RING ABOVE

U+212B (Å) ANGSTROM SIGN

U+0041 (A) LATIN CAPITAL LETTER A + U+030A (°) COMBINING RING ABOVE

All of the above are considered ***canonically equivalent***

Each of these examples represent the same character

1. The first is the “precomposed” form
 2. The second is an alternative codepoint called “Angstrom”
 3. The last combines a character with a diacritic
- 

Normalisation

There are 4 forms of normalisation:

NFD Normalization Form Canonical Decomposition	Characters are decomposed by canonical equivalence, and multiple combining characters are arranged in a specific order.
NFC Normalization Form Canonical Composition	Characters are decomposed and then recomposed by canonical equivalence.
NFKD Normalization Form Compatibility Decomposition	Characters are decomposed by compatibility, and multiple combining characters are arranged in a specific order.
NFKC Normalization Form Compatibility Composition	Characters are decomposed by compatibility, then recomposed by canonical equivalence.

Fonts

Fonts

The Unicode standard does not specify or create the font (typeface)

A font is a collection of graphical shapes (glyphs) that may include representations of some of the unicode codepoints.

Since a single TTF or OTF font has a hard limit of 65535 there is no single font that can cover the 1.1 million unicode characters - a family of fonts needs to be used



Fonts - Unicode in the browser

If the fonts referenced in the CSS do not cover a particular Unicode character then the browser will use fallback fonts

*By the specifications, browsers should display a character if there is **any** font in the system that contains it*

Font fallback doesn't always occur nicely (looking at you IE)

Using a font that has good coverage for your application is the ideal scenario

Or use SVG or images for particular instances



Fonts - Coverage

Google Noto Font - <https://www.google.com/get/noto/>

Provides massive coverage (goal is to cover all scripts) including emoji (android style) and many scripts

DejaVu - http://dejavu-fonts.org/wiki/Main_Page

Good coverage of most common Unicode symbols and common scripts

GNU - Unifont - <http://czyborra.com/unifont/> (Pixel font)

30,000+ characters in pixel format



Fonts - Coverage

Wikipedia provide coverage tables for the various commonly installed fonts (Arial etc.) which indicates which block ranges are included:

https://en.wikipedia.org/wiki/Unicode_font#0000-077F



Emoji



U+1F4A9



U+1F40D



U+1F602

Emoji History

Originated in Japanese mobile phones in 1990's

In 2010 the Unicode consortium integrated emoji into Unicode

There are over 1700 emoji including flags, keycaps, and modifier sequences

The word “emoji”, in fact, is just as Japanese as it sounds. It’s taken from the Japanese words “e” (“picture”), and “moji” (“character”).



Emoji Definition

They are assigned code points just like other Unicode characters

Depicted as black and white pictographs within the Unicode standard












As with all Unicode characters, graphical representation is up to the software

Emoji are not the same as emoticons... but some emoticons include unicode characters ٲ_ٲ



Emoji Examples

Actual implementations differ within various software

<u>Nº</u>	<u>Code</u>	<u>Brow.</u>	<u>Chart</u>	<u>Apple</u>	<u>Goog^d</u>	<u>Twtr.</u>	<u>One</u>	<u>FBM</u>	<u>Wind.</u>	<u>Sams.</u>
1	U+1F923			—				—		
<u>Nº</u>	<u>Code</u>	<u>Brow.</u>	<u>Chart</u>	<u>Apple</u>	<u>Goog^d</u>	<u>Twtr.</u>	<u>One</u>	<u>FBM</u>	<u>Wind.</u>	<u>Sams.</u>
2	U+1F924			—				—		
<u>Nº</u>	<u>Code</u>	<u>Brow.</u>	<u>Chart</u>	<u>Apple</u>	<u>Goog^d</u>	<u>Twtr.</u>	<u>One</u>	<u>FBM</u>	<u>Wind.</u>	<u>Sams.</u>
3	U+1F920			—				—		
4	U+1F921			—				—		
5	U+1F925			—				—		
<u>Nº</u>	<u>Code</u>	<u>Brow.</u>	<u>Chart</u>	<u>Apple</u>	<u>Goog^d</u>	<u>Twtr.</u>	<u>One</u>	<u>FBM</u>	<u>Wind.</u>	<u>Sams.</u>
6	U+1F922			—				—		
7	U+1F927			—				—		

Emoji Skin Tones

In 2015 (Unicode 8.0) skin tones based on the Fitzpatrick scale were introduced











































Many emoji such as people could be assigned up to 5 different shades by combining a modifier character

When an emoji codepoint is present, if it is immediately followed by one of the following codepoints: U+1F3FB, U+1F3FC, U+1F3FD, U+1F3FE, U+1F3FF - then if the software supports it it should display the skin tone variation

Otherwise the patch colour block will be displayed individually after the emoji character



Emoji Prince - U+1F934

14	U+1F934			—				—		
15	U+1F934 U+1F3FB	 	—	—				—		
16	U+1F934 U+1F3FC	 	—	—				—		
17	U+1F934 U+1F3FD	 	—	—				—		
18	U+1F934 U+1F3FE	 	—	—				—		
19	U+1F934 U+1F3FF	 	—	—				—		

The skin tone modifiers for the Prince (U+1F934) emoji character

Emoji

As seen from the example characters their codepoint is beyond **U+FFFF**

Emoji reside in the Supplementary Multilingual Plane (SMP) **U+10000 - U+1FFFF**

Since they are not available within the BMP (Basic Multilingual Plane) they will take at least 2 bytes to encode in any of the UTF forms

Plane 1, the Supplementary Multilingual Plane (SMP), contains historic scripts such as Linear B, Egyptian hieroglyphs, and cuneiform scripts; historic and modern musical notation; mathematical alphanumeric; Emoji and other pictographic sets



THERE AINT NO SNAKES ON THIS



MOTHERFUCKIN' BASIC MULTILINGUAL PLANE

made on Imgur



SUPPLEMENTARY PLANE

PHP Unicode

PHP Unicode

Does not support unicode at a low-level, work-arounds are required

Internally PHP stores as byte strings. PHP6 was going to change all that

Provides functions for working with Unicode/UTF-8 strings

String assignment and concatenation will still work without special consideration

`strpos()` and `strlen()` etc. will count bytes, not characters, so use multibyte aware functions instead...

Programmer must be aware to avoid *Mojibake*



PHP Unicode - Set Your Charset

Set `default_charset` in `php.ini` to “UTF-8”

Specify UTF-8 in your Content-Type header:

```
header("Content-Type: text/html; charset=utf-8");
```

Set the charset on your PDO connection DSN:

```
new PDO('mysql:host=your-hostname;dbname=your-db;charset=utf8mb4', ...);
```



PHP Unicode - Set Your Encoding

Explicitly pass “UTF-8” as the encoding parameter to `htmlspecialchars()` and `htmlspecialchars()`

Use `mb_internal_encoding()` and `mb_http_output()` at the start of all PHP files to ensure that PHP considers your strings as UTF-8 and it outputs UTF-8 to the browser

Save your source files encoded as UTF-8 - without a Byte Order Mark (BOM)



PHP Unicode - Use Multibyte Functions

PHP provides mbstring (mb_*) functions for multi-byte string handling - these should be used in ALL cases when working with unicode

`mb_strlen()`, `mb_substr()`, `mb_strpos()`, `mb_send_mail()` etc.

These functions will correctly work on the character level rather than byte level

The `iconv` functions can be used for converting to/from Unicode encodings such as UTF-8, as well as detecting encodings of input



PHP Unicode - JSON

`json_encode()` may break your UTF-8...

Well, by default `json_encode()` escapes UTF-8 as unicode escape sequences by default:

```
json_encode('č') => "\u010d"
```

As of PHP5.4 you can pass an additional flag called `JSON_UNESCAPED_UNICODE`:

```
json_encode('č', JSON_UNESCAPED_UNICODE) => "č"
```

Note that `json_decode` will handle either.




PHP Unicode - PHP7

PHP 7 now allows you to specify Unicode codepoints using the `\u{XX..}` syntax which will be output as UTF-8:

```
echo "\u{aa}"; // Can also be specified with leading 0's eg. \u{0000aa}  
=> a
```

```
echo "\u{9999}";  
=> 香
```

PHP7 also includes the Intl extension which includes a lot of great functionality for working with Unicode normalization and plenty of other International good-ness.



Database

Databases and Unicode

Every modern database supports Unicode

Implementation differs slightly but most have the concept of CHARSET and COLLATION

The CHARSET defines how the data is encoded

The COLLATION defines the semantics - sorting and comparison



MySQL

MySQL still defaults to **latin1** charset when not specified and latin1_swedish_ci for the collation - as of 5.7

In MySQL the **utf8** charset refers to a 3 byte implementation of UTF-8, which is usually not what you want when working with UTF-8

Use **utfmb4** - *not* the **utf8** collation or you may have data loss

Each character set has a default collation



MySQL - Unicode charsets

utf8 - UTF-8 encoding using 1-3 bytes per character

utf8mb4 - UTF-8 encoding using 1-4 bytes per char

ucs2 - UCS-2 encoding using 16 bits per character

utf16 - UTF-16 encoding 16 bits per character (like ucs2) but with support for supplementary characters

utf32 - UTF-32 encoding using 32 bits per character



MySQL Collation

For every CHARSET there will be several collations available: xxx_general_ci, xxx_bin, xxx_unicode_ci, plus language specific collations; xxx_swedish_ci etc.

The collation used determines how strings are sorted, how strings are compared, and how indexes are built

The language specific collations such as xxx_swedish_ci



For Example

`utf8_unicode_ci` supports mappings such as expansions;

That is... when one character compares as equal to combinations of other characters.

For example, in German and some other languages

ß is equal to ss



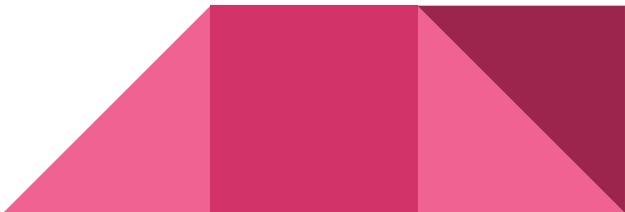
MySQL - Which collation?

Operations performed using the **xxx_general_ci** collation are **faster** than those for the **xxx_unicode_ci** collation **but slightly less correct**

xxx_bin is even faster, it works solely on code points - without normalization etc. during comparison

xxx_unicode_ci also supports contractions and ignorable characters

xxx_general_ci is a legacy collation that does not support expansions, contractions, or ignorable characters:
only one-to-one comparisons between characters.



MySQL - Declaring Charset

Connect to the database using the same encoding (so that there is no mangling between database and client)

SET NAMES utf8mb4 COLLATE utf8mb4_unicode_ci before you query/insert into the database

Ensure all your tables and columns are also in the same encoding.

Use **DEFAULT CHARSET=utf8mb4**



MySQL - Configuration

Update your MySQL configuration file (my.cnf)

```
[client]  
default-character-set = utf8mb4
```

```
[mysql]  
default-character-set = utf8mb4
```

```
[mysqld]  
character-set-client-handshake = FALSE  
character-set-server = utf8mb4  
collation-server = utf8mb4_unicode_ci
```

You can confirm these changes by issuing the following query:

```
SHOW VARIABLES LIKE 'character_set%';
```



MySQL - Configuration

Note that the **client-set-handshake=FALSE**

This means that the client handshake will be ignored, and the server will *insist* on the character set that is used during communication.



MySQL - Column storage

varchar(10) column within a utf8mb4 table will use between 1 and 40 bytes for a non-empty value

char(10) column will always use 40 bytes (pessimistic)

varbinary might show a slight improvement for columns which don't need locale side effects - eg. identifiers that don't need UTF-8 - eg. static values (such as a status column) are good candidates

MySQL's named column types (tinytext) etc. refer to byte size

Key length might be a concern



MySQL - String functions

The **LENGTH()** function returns length in **bytes** - use **CHAR_LENGTH()** to get the number of characters

```
SELECT CHAR_LENGTH('Ж'), LENGTH('Ж');  
CHAR_LENGTH = 1, LENGTH = 2
```

The other string functions behave as you would expect - they work on the character level: SUBSTR, CONCAT etc.



PostgreSQL - Installation

For new installs - initialise your database cluster as UTF8:

`initdb -E UTF8` ... On Debian or Ubuntu that's: `pg_createcluster`

The locale you use should match your system UTF-8 locale:

```
# cat /etc/default/locale
```

```
# File generated by update-locale
```

```
LANG="en_US"
```

```
LANGUAGE="en_US:"
```

```
LC_ALL=en_US.UTF-8
```



PostgreSQL - Encoding

PostgreSQL defines the encoding for a table in 3 ways:

ENCODING = How the characters are encoded eg. UTF-8

LC_COLLATE = String sort order


LC_CTYPE = Character classification (What is a letter? Its upper-case equivalent?)



PostgreSQL - Database creation

```
CREATE DATABASE "mydatabase"  
  WITH OWNER "somebody"  
  ENCODING 'UTF8'  
  LC_COLLATE = 'en_US.UTF-8'  
  LC_CTYPE = 'en_US.UTF-8'  
  TEMPLATE template0;
```

Notice that the above commands specify copying the template0 database. When copying any other database, the encoding and locale settings cannot be changed from those of the source database, because that might result in corrupt data.



PostgreSQL - Existing databases

Use `psql -l` to inspect your current database collations:

Name	Owner	Encoding	Collate	Ctype	Access privileges
iris	iris	UTF8	en_US.UTF-8	en_US.UTF-8	
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
template0	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres + postgres=CTc/postgres
template1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	=c/postgres + postgres=CTc/postgres

PostgreSQL - Client

Ensure the client communicates in UTF-8

```
SET NAMES 'UTF8';    // Same as initialise via PDO
```

To query the current client encoding:

```
SHOW client_encoding;
```

To return to the default encoding:

```
RESET client_encoding;
```



PostgreSQL - String functions

Postgres provides the convert function for converting strings between encodings:

```
convert(string using conversion_name)
```

```
convert(string text, [src_encoding name,] dest_encoding name)
```

Eg.

```
convert('PostgreSQL' using iso_8859_1_to_utf_8)
```

```
convert('text_in_unicode', 'UNICODE', 'LATIN1')
```

There are many built in conversions. See pgsql docs





HTML

HTML - History

In the early days of HTML (HTML 2.0), the document character set was specified as ISO-8859-1

HTML 4.0 was extended to support the Universal Character Set (UCS) which Unicode is basically a superset of

According to the HTML5 standard - all HTML authoring should now use UTF-8



HTML - Declaring Your Character Set

Always declare your charset as UTF-8:

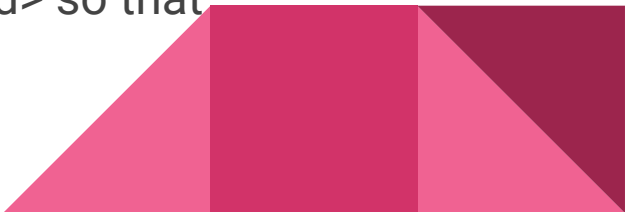
```
<meta charset="utf-8">
```

<- shorter, better

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

In HTML5 they're both the **same**, so use the shortest one!

Charset declaration should be placed right after the <head> so that browser can parse the rest of the document correctly



CSS

CSS files may contain Unicode characters (font names, content: “xx”)

The header or the document linking to the CSS will determine the perceived encoding - so what if a UTF-16 HTML file links to your CSS encoded in UTF-8

Always `@charset "UTF-8";` as the first line of your file

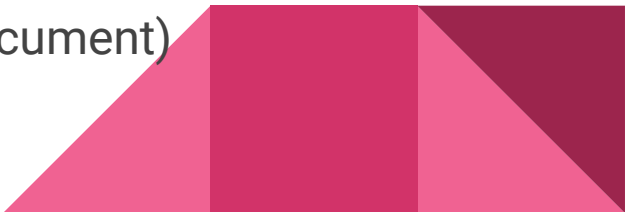


HTML - Encoding

Your HTML files should also be encoded in UTF-8 when authoring

If your HTML file is encoded in UTF-16 then the browser can't read the charset declaration to know the encoding so it must rely on header or heuristics.

From high to low priority, HTML5 uses the encoding of:

1. User override for charset (browser config)
 2. "Content-Type" header from server
 3. Document charset declaration
 4. Byte Order Mark or detection heuristics (analysis of document)
- 

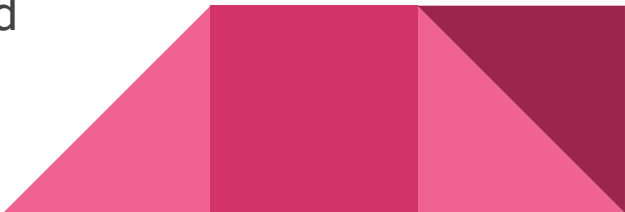
HTML - Entities

HTML entities are a plain text encoding of a character which can be passed and stored through plain text mechanisms

Entities can be used regardless of the encoding of the document

Any unicode character can be referenced by its Unicode codepoint via `—` or via the hex form `—`

HTML entities also have named versions such as `—` - all three will render the same character - there is a standard set of 252 named character entities for characters



HTML - Forms

`<form>` supports `accept-charset="utf-8"` attribute

This is only relevant if your document encoding is not already UTF-8

Within a UTF-8 document, the browser will encode user input within a `<form>` as UTF-8





Javascript

Javascript

Javascript uses unicode as storage 🐛

All strings are internally stored as UTF-16... or UCS-2 ☐

Source code (from ECMAScript 3.0) is specified as unicode too 🐛

But many of the string functions respond differently than you would expect ☐

ES6 provides additional functionality for working with Unicode 🐛



Javascript - Preparation

Make sure your JS files are saved in UTF-8 encoding

.. or only refer to codepoints using escape sequences `\uHHHH`

Send utf-8 header for your JS files

Also declare your `<html>` charset as utf-8

Charset attribute within `<script>` is only necessary if your document encoding is different or no headers were sent with the JS file:

```
<script src="/js/stuff.min.js" charset="utf-8"></script>
```



Javascript - Working with Unicode

XMLHttpRequest URL encoded form-data will always be sent as UTF-8 encoding as per the HTML5 spec.

The response encoding will be determined by the Content-Type header received (containing a charset declaration), or via a BOM, or via “sniffing” the content and/or via the current document encoding



Javascript - Working with Unicode

The way Javascript handles unicode can sometimes be surprising

JavaScript strings are represented using UTF-16 code units

Any character within the BMP (U+0000 to U+FFFF) can be represented using a single code unit, others require two units

```
'✳'.length;  
= 1           // Looks good!
```

```
'✳' == '\u2618';  
= true       // You can use unicode escapes
```



Javascript - Character storage

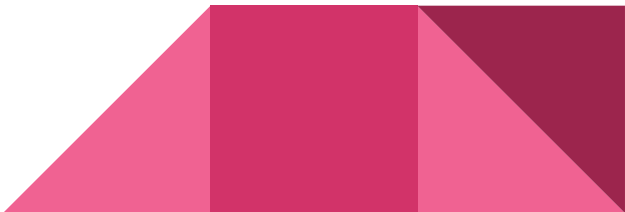
In Javascript characters such as emoji require two 16-bit code units. These continuous code units are often called surrogate pairs

And Javascript string and character functions actually work on code units rather than characters:

```
'👨'.length = 2 // Returns the number of 16-bit code units - not chars
```

The string above actually contains two code units:

```
'👨' == '\uD83D\uDCA9'
```



Javascript - Some problems

```
'🐼'.split('').reverse().join(''); // JS4EVA  
= '🐼🐼' // Reversed the code units to: '\uDCA9\uD83D'
```

```
var str = '🐼🍀';  
for (var i = 0; i < str.length; i++) {  
    console.log(str[i]);  
}
```

= 🐼

= 🐼

= 🍀

Javascript - Solutions

Javascript can handle Unicode just fine... as long as you don't touch anything

If you don't try to work on the character level, you may not encounter problems

Maybe don't use characters outside of the BMP

Definitely don't try sorting your strings...

And don't even bother comparing two de-normalized strings...

...

... ES6!



Javascript - ES6 🚀

ES6 brings several long-needed unicode improvements to Javascript

Unicode codepoint escapes `\u{1F680}` - so we work with the entire codepoint rather than the individual code units `\uD83D\uDE80`

`String.prototype.codePointAt()` and `String.fromCodePoint()` for converting between characters 🐘 and Unicode code points `\u{1F680}`

`String.prototype.normalize()` for normalising a string (ie. for comparing/sorting)

String iterator and spread operator ...



Javascript - ES6 string iterables

The string iterator in ES6 splits strings along codepoint boundaries

```
for (let ch of 'x\uD83D\uDE80y') {  
  console.log(ch.length)  
}
```

```
// ch.length = 1
```

```
// ch.length = 2
```

```
// ch.length = 1
```



Javascript - ES6 spread operator

The spread operator allows us to easily split a string into an array over its codepoints:

```
let chars = [...'abc']    // ['a', 'b', 'c']
```

We can use this to correctly count the characters in a string:

```
let chars = [...'🐾👉❤️'] // ['🐾', '👉', '❤️']  
chars.length    // 3
```



Javascript - ES6 spread operator

Your string may have been constructed using unicode characters, unicode codepoint sequence, or UTF-16 unicode sequences:

```
[...'🍀 ym elots \u{1F984} a \uD83D\uDCA9 ho'].reverse().join('')
```

```
// oh 🍷 a 🍷 stole my 🍀
```

Regardless, the spread operator splits individual codepoints into an array, and therefore can be used with **reverse()** etc.



Javascript - ES6 getCodePointAt

String.prototype.getCodePointAt(pos) returns the decimal value for a Unicode character within a string:

```
const chars = '𐀀A'    // String containing unicorn char and letter 'A'  
chars.codePointAt(0)  // 129412  -- decimal value  
'\\u{' + chars.codePointAt(0).toString(16).toUpperCase() + '}'  
    // \\u{1F984} -- The unicode escape sequence
```

BUT ...

```
chars.codePointAt(1)  // 56708  -- not 65!
```

Position is still based on 16-bit code units - use spread etc!



Javascript - ES6 fromCodePoint

Converting from a codepoint to character is easy in ES6 too:

```
String.fromCharCode(65);      // "A"      "\u0041"
String.fromCharCode(0x404);    // "Ÿ"      "\u0404"
String.fromCharCode(0x2F804);  // "𐀀"    "\uD83E\uDD84"
String.fromCharCode(129412);   // "𐀀"    "\uD83E\uDD84"

let allTheUnicorns = "𐀀"
allTheUnicorns += String.fromCharCode(129412)
allTheUnicorns += String.fromCharCode(0x2F804)
// 𐀀𐀀𐀀 - a beautiful tricorn!
```

Closing Comments

Just use UTF-8

UTF-8 all the things

In your headers, in your encodings, in your database, and through UTF-8 aware functions

Think about your input, how you're handling it, how you're storing it, how you're displaying it, and how you manipulate it

It's easy! Unicode is normally only a problem when one or more pieces of your stack or request cycle do not use the same encoding

Questions? Nah

