

Code

In communications and information processing, **code** is a system of rules to convert information—such as a letter, word, sound, image, or gesture—into another form or representation, sometimes shortened or secret, for communication through a communication channel or storage in a storage medium.

An early example is the invention of language which enabled a person, through speech, to communicate what he or she saw, heard, felt, or thought to others. But speech limits the range of communication to the distance a voice can carry, and limits the audience to those present when the speech is uttered. The invention of writing, which converted spoken language into visual symbols, extended the range of communication across space and time.

The process of **encoding** converts information from a source into symbols for communication or storage.

Decoding is the reverse process, converting code symbols back into a form that the recipient understands.

Character encodings

Probably the most widely known data communication code so far (aka character representation) in use today is **ASCII**. In one or another (somewhat compatible) version, it is used by nearly all personal computers, terminals, printers, and other communication equipment. It represents 128 characters with seven-bit binary numbers—that is, as a string of seven 1s and 0s (bits). In ASCII, a lowercase "a" is always 1100001, an uppercase "A" always 1000001, and so on. There are many other encodings which represent each character by a byte (usually referred as code pages), integer code point (Unicode) or a byte sequence (UTF-8).

Character encoding is used to represent a repertoire of characters by some kind of encoding system. Depending on the abstraction level and context, corresponding code points and the resulting code space may be regarded as bit patterns, octets, natural numbers, electrical pulses, etc. A character encoding is used in computation, data storage, and transmission of textual data. "Character set", "character map", "codeset" and "code page" are related, but not identical, terms.

Early character codes associated with the optical or electrical telegraph could only represent a subset of the characters used in written languages, sometimes restricted to upper case letters, numerals and some punctuation only. The low cost of digital representation of data in modern computer systems allows more elaborate character codes (such as Unicode) which represent most of the characters used in many written languages. Character encoding using internationally accepted standards permits worldwide interchange of text in electronic form.

Other forms

There are **codes** using colors, like traffic lights, the color code employed to mark the nominal value of the electrical resistors or that of the trashcans devoted to specific types of garbage (paper, glass, organic, etc.).

In marketing, **coupon codes** can be used for a financial discount or rebate when purchasing a product from a (usually internet) retailer.

In military environments, specific sounds with the cornet (Tr: Kornet, Trompet benzeri çalgı) are used for different uses: to mark some moments of the day, to command the infantry in the battlefield, etc.

Communication systems for sensory impairments, such as sign language for deaf people and braille for blind people, are based on **movement codes** or **tactile codes**.

Musical scores are the most common way to **encode music**.

Specific games have their own code systems to record the matches, e.g. chess notation.

ASCII

ASCII abbreviated from **American Standard Code for Information Interchange**, is a character encoding standard for electronic communication. ASCII codes represent text in computers, telecommunications equipment, and other devices. Most modern character-encoding schemes are based on ASCII, although they support many additional characters.

Originally based on the English alphabet, **ASCII encodes 128 specified characters into seven-bit integers** as shown by the ASCII chart above. It is a **7-bit code**. The original ASCII table is encoded on 7 bits therefore it has 128 characters. Ninety-five of the encoded characters are printable: these include the digits 0 to 9, lowercase letters a to z, uppercase letters A to Z, and punctuation symbols. In addition, the original ASCII specification included 33 non-printing control codes which originated with Teletype machines; most of these are now obsolete.

For example, lowercase i would be represented in the ASCII encoding by binary 1101001 = hexadecimal 69 (i is the ninth letter) = decimal 105.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

It can be inferred from the above binary representation that decimal values 0 to 127 can be represented using 7 bits leaving the 8th bit free.

This is where things started getting messy.

People came up with different ways of using the remaining eight bit which represented decimal values from 128 to 255 and collisions started to happen. For instance the decimal value 182 was used by the Vietnamese to represent the Vietnamese alphabet ð whereas the same value 182 was used by the Indians to represent the Hindi alphabet ढ. So if an email written by an Indian contains the alphabet ढ and if it is read by a person in Vietnam it would appear as ð. Clearly not the intended way to appear.

This is where **Unicode** character set came to save the day.

Extended ASCII

Extended ASCII (EASCII or high ASCII) character encodings are **8-bit** or larger encodings that include the standard seven-bit ASCII characters, plus additional characters. The use of the term is sometimes criticized, because it can be mistakenly interpreted to mean that the ASCII standard has been updated to include more than 128 characters or that the term unambiguously identifies a single encoding, neither of which is the case.

The ASCII character set is barely large enough for general use, and far too small for universal use. Many more letters and symbols are desirable, useful, or required to directly represent letters of alphabets other than English, more kinds of punctuation and spacing, more mathematical operators and symbols ($\times \div \cdot \neq \geq \approx \pi$ etc.), some unique symbols used by some programming languages, ideograms, logograms, box-drawing characters, etc.

The biggest problem for computer users around the world was other alphabets. ASCII's English alphabet almost accommodates European languages, if accented letters are replaced by non-accented letters or two-character approximations. Languages with dissimilar basic alphabets could use transliteration. Users were not comfortable with such compromises.

The ASCII code

American Standard Code for Information Interchange

www.theasciicode.com.ar

ASCII control characters			
DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de línea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift out)
15	0Fh	SI	(shift in)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowledge)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters								
DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	\$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(72	48h	H	104	68h	h
41	29h)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	
61	3Dh	=	93	5Dh]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	-	theASCIIcode.com.ar		

Extended ASCII characters																	
DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	Ł	224	E0h	Ó						
129	81h	ü	161	A1h	í	193	C1h	ł	225	E1h	ô						
130	82h	é	162	A2h	ó	194	C2h	Ł	226	E2h	Ö						
131	83h	â	163	A3h	ú	195	C3h	ł	227	E3h	Ø						
132	84h	ä	164	A4h	ñ	196	C4h	Ł	228	E4h	ō						
133	85h	à	165	A5h	Ñ	197	C5h	ł	229	E5h	Ô						
134	86h	ã	166	A6h	ª	198	C6h	Ł	230	E6h	µ						
135	87h	ç	167	A7h	º	199	C7h	ł	231	E7h	ß						
136	88h	ê	168	A8h	¿	200	C8h	Ł	232	E8h	Þ						
137	89h	ë	169	A9h	®	201	C9h	ł	233	E9h	Ú						
138	8Ah	è	170	AAh	¬	202	CAh	Ł	234	EAh	Û						
139	8Bh	ï	171	ABh	½	203	CBh	ł	235	EBh	Ü						
140	8Ch	ì	172	ACH	¼	204	CCh	Ł	236	ECh	Ý						
141	8Dh	í	173	ADh	»	205	CDh	ł	237	EDh	Ÿ						
142	8Eh	Ā	174	A Eh	«	206	CEh	Ł	238	EEh	·						
143	8Fh	Ā	175	AFh	»	207	CFh	ł	239	EFh	·						
144	90h	Ē	176	B0h	»	208	D0h	Ł	240	F0h							
145	91h	æ	177	B1h	»	209	D1h	ł	241	F1h	±						
146	92h	Æ	178	B2h	»	210	D2h	Ł	242	F2h	¼						
147	93h	ô	179	B3h	»	211	D3h	ł	243	F3h	½						
148	94h	ö	180	B4h	»	212	D4h	Ł	244	F4h	¾						
149	95h	õ	181	B5h	»	213	D5h	ł	245	F5h	¸						
150	96h	ù	182	B6h	»	214	D6h	Ł	246	F6h	÷						
151	97h	û	183	B7h	»	215	D7h	ł	247	F7h							
152	98h	ÿ	184	B8h	»	216	D8h	Ł	248	F8h	ˆ						
153	99h	Ų	185	B9h	»	217	D9h	ł	249	F9h	˜						
154	9Ah	Ų	186	BAh	»	218	DAh	Ł	250	FAh	˙						
155	9Bh	ø	187	B Bh	»	219	DBh	ł	251	FBh	˚						
156	9Ch	£	188	BCh	»	220	DCh	Ł	252	FCh	¸						
157	9Dh	Ø	189	BDh	»	221	DDh	ł	253	FDh	˚						
158	9Eh	x	190	BEh	»	222	DEh	Ł	254	FEh	˚						
159	9Fh	f	191	BFh	»	223	DFh	ł	255	FFh	˚						

Code Point

In character encoding terminology, a **code point** or **code position** is any of the numerical values that make up the code space. Many code points represent single characters but they can also have other meanings, such as for formatting.

For example, the character encoding scheme **ASCII** comprises 128 code points in the range 0hex to 7Fhex, **Extended ASCII** comprises 256 code points in the range 0hex to FFhex, and **Unicode** comprises 1,114,112 code points in the range 0hex to 10FFFFhex. The Unicode code space is divided into seventeen planes (the basic multilingual plane, and 16 supplementary planes), each with 65,536 (= 2¹⁶) code points. Thus the total size of the Unicode code space is 17 × 65,536 = 1,114,112.

Unicode

Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. The latest version contains a repertoire of 136,755 characters covering 139 modern and historic scripts, as well as multiple symbol sets. The Unicode Standard is maintained in conjunction with ISO/IEC 10646, and both are code-for-code identical.

The Unicode Standard consists of a set of code charts for visual reference, an encoding method and set of standard character encodings, a set of reference data files, and a number of related items, such as character properties, rules for normalization, decomposition, collation, rendering, and bidirectional display order (for the correct display of text containing both right-to-left scripts, such as Arabic and Hebrew, and left-to-right scripts). As of June 2017, the most recent version is **Unicode 10.0**. The standard is maintained by the **Unicode Consortium**.

Unicode's success at unifying character sets has led to its widespread and predominant use in the internationalization and localization of computer software. The standard has been implemented in many recent technologies, including modern operating systems, XML, Java (and other programming languages), and the .NET Framework.

Unicode can be implemented by different character encodings. The Unicode standard defines **UTF-8**, **UTF-16**, and **UTF-32**, and several other encodings are in use. The most commonly used encodings are UTF-8, UTF-16 and UCS-2, a precursor of UTF-16. UTF-8, dominantly used by websites (over 90%), uses one byte for the first 128 code points, and up to 4 bytes for other characters.

- The first **128 Unicode code points** are the **ASCII characters**, which means that any ASCII text is also a UTF-8 text.

UTF : Unicode Transformation Format

Some people are under the misconception that Unicode is simply a 16-bit code where each character takes 16 bits and therefore there are 65,536 possible characters. **This is NOT, actually, correct.**

Every platonic letter in every alphabet is assigned a magic number by the Unicode consortium which is written like this: **U+0639**. This magic number is called a code point. The **U+** means “Unicode” and the numbers are hexadecimal. **U+0639** is the Arabic letter **Ain**. The English letter **A** would be **U+0041**. You can find them all by visiting the Unicode web site.

There is no real limit on the number of letters that Unicode can define and in fact they have gone beyond 65,536 so not every unicode letter can really be squeezed into two bytes, but that was a myth anyway.

OK, so say we have a string:

Hello which, in Unicode, corresponds to these five code points:

U+0048 U+0065 U+006C U+006C U+006F.

Just a bunch of code points. Numbers, really. We haven't yet said anything about how to store this in memory or represent it in an email message.

Encodings

That's where **encodings** come in. The earliest idea for Unicode encoding, which led to the myth about the two bytes, was, hey, let's just store those numbers in two bytes each.

So Hello becomes 00 48 00 65 00 6C 00 6C 00 6F Right?

Not so fast! Couldn't it also be: 48 00 65 00 6C 00 6C 00 6F 00 ?

Well, technically, yes, I do believe it could, and, in fact, early implementors wanted to be able to store their Unicode code points in **high-endian** or **low-endian** mode, whichever their particular CPU was fastest at, and lo, it was evening and it was morning and there were already two ways to store Unicode.

So the people were forced to come up with the bizarre convention of storing a **FE FF** at the beginning of every Unicode string; this is called a Unicode Byte Order Mark and if you are swapping your high and low bytes it will look like a **FF FE** and the person reading your string will know that they have to swap every other byte. Phew. Not every Unicode string in the wild has a byte order mark at the beginning.

For a while it seemed like that might be good enough, but programmers were complaining. “**Look at all those zeros!**” they said, since they were Americans and they were looking at English text which rarely used code points above **U+00FF**.

Thus was invented the brilliant concept of **UTF-8**. UTF-8 was another system for storing your string of Unicode code points, those magic U+ numbers, in memory using **8 bit bytes**. In UTF-8, every code point from **0-127** is stored in a single byte. Only code points 128 and above are stored using 2, 3, in fact, up to 4 bytes.

- Unicode code point başındaki 0'ları atıyoruz. Bize gereken sadece Unicode hex değeri, Unicode'da U+0048 yazılan code noktasının hex değeri 48, UTF-8 e sadece 48 yazıyoruz.

Number of 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

Backwards Compatibility : This has the neat side effect that English text looks exactly the same in UTF-8 as it did in ASCII, so Americans don't even notice anything wrong. Only the rest of the world has to jump through hoops.

Specifically, Hello, which was U+0048 U+0065 U+006C U+006C U+006F, will be stored as 48 65 6C 6C 6F, which, behold! is the same as it was stored in ASCII, and ANSI, and every OEM character set on the planet. Now, if you are so bold as to use accented letters or Greek letters or Klingon letters, you'll have to use several bytes to store a single code point, but the Americans will never notice.

(UTF-8 also has the nice property that ignorant old string-processing code that wants to use a single 0 byte as the null-terminator will not truncate strings).

And in fact now that you're thinking of things in terms of platonic ideal letters which are represented by Unicode code points, those unicode code points can be encoded in any old-school encoding scheme, too! For example, you could encode the Unicode string for Hello (U+0048 U+0065 U+006C U+006C U+006F) in ASCII, or the old OEM Greek Encoding, or the Hebrew ANSI Encoding, or any of several hundred encodings that have been invented so far, with one catch: some of the letters might not show up! If there's no equivalent for the Unicode code point you're trying to represent in the encoding you're trying to represent it in, you usually get a little question mark: ? or, if you're really good, a box. Which did you get? -> 📦

- Kısaca burda demek istenen: Hala eski bir encoding sistemi kullanan birisine Unicode gönderdiğin zaman, o senin gönderdiğin codu kendi sistemine çevirebilir , bazı karakterler kendi sisteminde olmadığı için, bu simgeyi görebiliriz 📦

The Single Most Important Fact About Encodings

If you completely forget everything I just explained, please remember one extremely important fact.

- **It does not make sense to have a string without knowing what encoding it uses.** You can no longer stick your head in the sand and pretend that “plain” text is ASCII. **There Ain’t No Such Thing As Plain Text. If you have a string, in memory, in a file, or in an email message, you have to know what encoding it is in or you cannot interpret it or display it to users correctly.**

If you have a string, in memory, in a file, or in an email message, you have to know what encoding it is in or you cannot interpret it or display it to users correctly.

Almost every stupid “my website looks like gibberish” or “she can’t read my emails when I use accents” problem comes down to one naive programmer who didn’t understand the simple fact that if you don’t tell me whether a particular string is encoded using UTF-8 or ASCII or ISO 8859-1 (Latin 1) or Windows 1252 (Western European), you simply cannot display it correctly or even figure out where it ends. There are over a hundred encodings and above code point 127, all bets are off.

A Different Article About Encoding

There are 95 human readable characters specified in the ASCII table, including the letters A through Z both in upper and lower case, the numbers 0 through 9, a handful of punctuation marks and characters like the dollar symbol, the ampersand and a few others. It also includes 33 values for things like space, line feed, tab, backspace and so on. These are not printable per se, but still visible in some form and useful to humans directly.

A number of values are only useful to a computer, [like codes to signify the start or end of a text](#).

Encode: convert into a coded form

code: a system of words, letters, figures, or other symbols substituted for other words, letters, etc.

To **encode** something in ASCII, follow the table from right to left, substituting letters for bits. To **decode** a string of bits into human readable characters, follow the table from left to right, substituting bits for letters.

To **encode** means to use something to represent something else. An **encoding** is the set of rules with which to convert something from one representation to another.

Character set, charset: The set of characters that can be encoded. "The ASCII encoding encompasses a character set of 128 characters." Essentially synonymous to "encoding".

Code Page: A "page" of codes that map a character to a number or bit sequence. A.k.a. "the table". Essentially synonymous to "encoding".

String: A string is a bunch of items strung together. A bit string is a bunch of bits, like 01010011. A character string is a bunch of characters, like this. Synonymous to "sequence".

- So, how many bits does Unicode use to encode all these characters? **None. Because Unicode is not an encoding.**

Confused? Many people seem to be. Unicode first and foremost defines a table of code points for characters. That's a fancy way of saying "65 stands for A, 66 stands for B and 9,731 stands for ☺" (seriously, it does). How these code points are actually encoded into bits is a different topic. To represent 1,114,112 different values, two bytes aren't enough. Three bytes are, but three bytes are often awkward to work with, so four bytes would be the comfortable minimum. But, unless you're actually using Chinese or some of the other characters with big numbers that take a lot of bits to encode, you're never going to use a huge chunk of those four bytes. If the letter "A" was always encoded to 00000000 00000000 01000001, "B" always to 00000000 00000000 00000000 01000010 and so on, any document would bloat to four times the necessary size.

To optimize this, there are several ways to encode Unicode code points into bits:

UTF-32 is such an encoding that encodes all Unicode code points using 32 bits. That is, four bytes per character. It's very simple, but often wastes a lot of space.

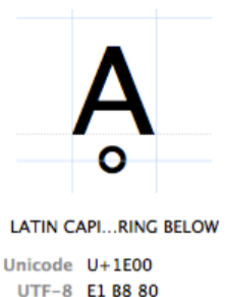
UTF-16 and **UTF-8** are **variable-length encodings**. If a character can be represented using a single byte (because its code point is a very small number), **UTF-8** will encode it with a single byte. If it requires two bytes, it will use two bytes and so on. It has elaborate ways to use the highest bits in a byte to signal how many bytes a character consists of. This can save space, but may also waste space if these signal bits need to be used often. UTF-16 is in the middle, using at least two bytes, growing to up to four bytes as necessary.

character	encoding	bits
A	UTF-8	01000001
A	UTF-16	00000000 01000001
A	UTF-32	00000000 00000000 00000000 01000001
あ	UTF-8	11100011 10000001 10000010
あ	UTF-16	00110000 01000010
あ	UTF-32	00000000 00000000 00110000 01000010

And that's all there is to it. Unicode is a large table mapping characters to numbers and the different UTF encodings specify how these numbers are encoded as bits. Overall, Unicode is yet another encoding scheme. There's nothing special about it, it's just trying to cover everything while still being efficient. And that's A Good Thing

Code Points

Characters are referred to by their "Unicode code point". Unicode code points are written in hexadecimal (to keep the numbers shorter), preceded by a "U+" (that's just what they do, it has no other meaning than "this is a Unicode code point"). The character Å has the Unicode code point U+1E00. In other (decimal) words, it is the 7680th character of the Unicode table. It is officially called "LATIN CAPITAL LETTER A WITH RING BELOW".



Different encodings: Any character can be encoded in many different bit sequences and any particular bit sequence can represent many different characters, depending on which encoding is used to read or write them. The reason is simply because different encodings use different numbers of bits per characters and different values to represent different characters.

	bits	encoding	characters
	11000100 01000010	Windows Latin 1	ÄB
	11000100 01000010	Mac Roman	/B
	11000100 01000010	GB18030	腩

characters	encoding	bits
Föö	Windows Latin 1	01000110 11111000 11110110
Föö	Mac Roman	01000110 10111111 10011010
Föö	UTF-8	01000110 11000011 10111000 11000011 10110110

Misconceptions, Confusions And Problems

Having said all that, we come to the actual problems experienced by many users and programmers every day, how those problems relate to all of the above and what their solution is. The biggest problem of all is:

Why In God's Name Are My Characters Garbled?!

ÉĞÉİÉRÅ[ÉfÉBÉİÉOÇÖiÔÇμÇ#Ç»ÇÇ

If you open a document and it looks like this, there's one and only one reason for it: Your text editor, browser, word processor or whatever else that's trying to read the document is assuming the wrong encoding. That's all. The document is not broken (well, unless it is, see below), there's no magic you need to perform, you simply need to select the right encoding to display the document. The hypothetical document above contains this sequence of bits:

```
10000011 01000111 10000011 10010011 10000011 01010010 10000001 01011011
10000011 01100110 10000011 01000010 10000011 10010011 10000011 01001111
10000010 11001101 10010011 11101111 10000010 10110101 10000010 10101101
10000010 11001000 10000010 10100010
```

Now, quick, what encoding is that? If you just shrugged, you'd be correct. Who knows, right?

Well, let's try to interpret this as ASCII. Hmm, most of these bytes start with a 1 bit. If you remember correctly, ASCII doesn't use that bit. So it's not ASCII. What about UTF-8? Hmm, no, most of these sequences are not valid UTF-8. So UTF-8 is out, too. Let's try "Mac Roman" (yet another encoding scheme for them Europeans). Hey, all those bytes are valid in Mac Roman. 10000011 maps to "É", 01000111 to "G" and so on. If you read this bit sequence using the Mac Roman encoding, the result is "ÉĞÉİÉRÅ[ÉfÉBÉİÉOÇÖiÔÇμÇ#Ç»ÇÇ". That looks like a valid string, no? Yes? Maybe? Well, how's the computer to know? Maybe somebody meant to write "ÉĞÉİÉRÅ[ÉfÉBÉİÉOÇÖiÔÇμÇ#Ç»ÇÇ". For all I know that could be a DNA sequence.⁵ Unless you have a better suggestion, let's declare this to be a DNA sequence, say this document was encoded in Mac Roman and call it a day.

Of course, that unfortunately is complete nonsense. The correct answer is that this text is encoded in the **Japanese Shift-JIS encoding** and was supposed to read "エンコーディングは難しくない". Well, who'd've thunk?

The primary cause of garbled text is: Somebody is trying to read a byte sequence using the wrong encoding. The computer always needs to be told what encoding some text is in. Otherwise it can't know. There are different ways how different kinds of documents can specify what encoding they're in and these ways should be used. A raw bit sequence is always a mystery box and could mean anything.


Most browsers allow the selection of a different encoding in the View menu under the menu option "Text Encoding", which causes the browser to reinterpret the current page using the selected encoding. Other programs may offer something like "Reopen using encoding..." in the File menu, or possibly an "Import..." option which allows the user to manually select an encoding.

My Document Doesn't Make Sense In Any Encoding!

If a sequence of bits doesn't make sense (to a human) in any encoding, the document has mostly likely been converted incorrectly at some point. Say we took the above text "ÉĞÉİÉRÅ[ÉfÉBÉİÉOÇÖiÔÇμÇ#Ç»ÇÇ" because we didn't know any better and saved it as UTF-8. The text editor assumed it correctly read a Mac Roman encoded text and you now want to save this text in a different encoding. All of these characters are valid Unicode characters after all. That is to say, there's a code point in Unicode that can represent "É", one that can represent "G" and so on. So we can happily save this text as UTF-8:

```
11000011 10001001 01000111 11000011 10001001 11000011 10101100 11000011
10001001 01010010 11000011 10001001 01011011 11000011 10001001 01100110
11000011 10001001 01000010 11000011 10001001 11000011 10101100 11000011
10001001 01001111 11000011 10000111 11000011 10010101 11000011 10101100
11000011 10010100 11000011 10000111 11000010 10110101 11000011 10000111
11100010 10001001 10100000 11000011 10000111 11000010 10111011 11000011
10000111 11000010 10100010
```


This is now the UTF-8 bit sequence representing the text "ÉÉÉÉÉÉ[ÉÉÉÉÉÉÖÏÔÇµÇ#Ç»ÇÇ". This bit sequence has absolutely nothing to do with our original document. Whatever encoding we try to open it in, we won't ever get the text "エンコーディングは難しくない" from it. It is completely lost. It would be possible to recover the original text from it if we knew that a Shift-JIS document was misinterpreted as Mac Roman and then accidentally saved as UTF-8 and reversed this chain of missteps. But that would be a lucky fluke.

Many times certain bit sequences are invalid in a particular encoding. If we tried to open the original document using ASCII, some bytes would be valid in ASCII and map to a real character and others wouldn't. The program you're opening it with may decide to silently discard any bytes that aren't valid in the chosen encoding, or possibly replace them with ?. There's also the "Unicode replacement character"  (U+FFFD) which a program may decide to insert for any character it couldn't decode correctly when trying to handle Unicode. If a document is saved with some characters gone or replaced, then those characters are really gone for good with no way to reverse-engineer them.

If a document has been misinterpreted and converted to a different encoding, it's broken. Trying to "repair" it may or may not be successful, usually it isn't. Any manual bit-shifting or other encoding voodoo is mostly that, voodoo. It's trying to fix the symptoms after the patient has already died.

So How To Handle Encodings Correctly?

It's really simple: Know what encoding a certain piece of text, that is, a certain byte sequence, is in, then interpret it with that encoding. That's all you need to do. If you're writing an app that allows the user to input some text, specify what encoding you accept from the user. For any sort of text field, the programmer can usually decide its encoding. For any sort of file a user may upload or import into a program, there needs to be a specification what encoding that file should be in. Alternatively, the user needs some way to tell the program what encoding the file is in. This information may be part of the file format itself, or it may be a selection the user has make (not that most users would usually know, unless they have read this article).

If you need to convert from one encoding to another, do so cleanly using tools that are specialized for that. Converting between encodings is the tedious task of comparing two code pages and deciding that character 152 in encoding A is the same as character 4122 in encoding B, then changing the bits accordingly. This particular wheel does not need reinventing and any mainstream programming language includes some way of converting text from one encoding to another without needing to think about code points, pages or bits at all.

Say, your app must accept files uploaded in GB18030, but internally you are handling all data in UTF-32. A tool like `iconv` can cleanly convert the uploaded file with a one-liner like `iconv('GB18030', 'UTF-32', $string)`. That is, it will preserve the characters while changing the underlying bits:

character	GB18030 encoding	UTF-32 encoding
繚	10111111 01101100	00000000 00000000 01111110 00100111

That's all there is to it. The content of the string, that is, the human readable characters, didn't change, but it's now a valid UTF-32 string. If you keep treating it as UTF-32, there's no problem with garbled characters. As discussed at the very beginning though, **not all encoding schemes can represent all characters. It's not possible to encode the character "縹" in any encoding scheme designed for European languages.** Something Bad™ would happen if you tried to.

Flukes

I have this website talking to a database. My app handles everything as UTF-8 and stores it as such in the database and everything works fine, but when I look at my database admin interface my text is garbled.

-- Anonymous code monkey

There are situations where encodings are handled incorrectly but things still work. An often-encountered situation is a database that's set to **latin-1** and an app that works with **UTF-8** (or any other encoding). Pretty much any combination of 1s and 0s is valid in the single-byte **latin-1** encoding scheme. If the database receives text from an application that looks like 11100111 10111000 10100111, it'll happily store it, thinking the app meant to store the three latin characters "ç, \$". After all, why not? It then later returns this bit sequence back to the app, which will happily accept it as the UTF-8 sequence for "縹", which it originally stored. The database admin interface automatically figures out that the database is set to **latin-1** though and interprets any text as **latin-1**, so all values look garbled only in the admin interface.

That's a case of fool's luck where things happen to work when they actually aren't. Any sort of operation on the text in the database may or may not work as intended, since the database is not interpreting the text correctly. In a worst case scenario, the database inadvertently destroys all text during some random operation two years after the system went into production because it was operating on text assuming the wrong encoding.

Native-Schmative

So what does it mean for a language to natively support or not support Unicode? **It basically refers to whether the language assumes that one character equals one byte or not.** For example, PHP allows direct access to the characters of a string using array notation:

```
echo $string[0];
```

If that `$string` was in a single-byte encoding, this would give us the first character. But only because "character" coincides with "byte" in a single-byte encoding. PHP simply gives us the first byte without thinking about "characters". Strings are byte sequences to PHP, nothing more, nothing less. All this "readable character" stuff is a human thing and PHP doesn't care about it.

The same goes for many standard functions such as `substr`, `strpos`, `trim` and so on. The non-support arises if there's a discrepancy between the length of a byte and a character.

```
11100110 10111100 10100010 11100101 10101101 10010111
漢                                字
```

Using `$string[0]` on the above string will, again, give us the first byte, which is 11100110. In other words, a third of the three-byte character "漢". 11100110 is, by itself, an invalid UTF-8 sequence, so the string is now broken. If you felt like it, you could try to interpret that in some other encoding where 11100110 represents a valid character, which will result in some random character. Have fun, but don't use it in production.

And that's actually all there is to it. "PHP doesn't natively support Unicode" simply means that most PHP functions assume one byte = one character, which may lead to it chopping multi-byte characters in half or calculating the length of strings incorrectly if you're naively using non-multi-byte-aware functions on multi-byte strings. It does not mean that you can't use Unicode in PHP or that every Unicode string needs to be blessed by `utf8_encode` or other such nonsense.

Luckily, there's the [Multibyte String extension](#), which replicates all important string functions in a multi-byte aware fashion. Using `mb_substr($string, 0, 1, 'UTF-8')` on the above string correctly returns 11100110 10111100 10100010, which is the whole "漢" character. Because the `mb_` functions now have to actually think about what they're doing, they need to know what encoding they're working on. Therefore every `mb_` function accepts an `$encoding` parameter as well. Alternatively, this can be set globally for all `mb_` functions using `mb_internal_encoding`.

Encoding-Aware Languages

What does it mean for a language to support Unicode then? **Javascript** for example supports Unicode. In fact, any string in Javascript is UTF-16 encoded. In fact, it's the only thing Javascript deals with. You cannot have a string in Javascript that is not UTF-16 encoded. Javascript worships Unicode to the extent that there's no facility to deal with any other encoding in the core language. Since Javascript is most often run in a browser that's not a problem, since the browser can handle the mundane logistics of encoding and decoding input and output.

Other languages are simply encoding-aware. Internally they store strings in a particular encoding, often UTF-16. In turn they need to be told or try to detect the encoding of everything that has to do with text. They need to know

- what encoding the source code is saved in,
- what encoding a file they're supposed to read is in,
- what encoding you want to output text in;

and they convert encodings on the fly as needed with some manifestation of Unicode as the middleman. They're doing the same thing you can/should/need to do in PHP semi-automatically behind the scenes. That's neither better nor worse than PHP, just different. The nice thing about it is that standard language functions that deal with strings Just Work™, while in PHP one needs to spare some attention to whether a string may contain multi-byte characters or not and choose string manipulation functions accordingly.

Final TL;DR

- Text is always a sequence of bits which needs to be translated into human readable text using lookup tables. If the wrong lookup table is used, the wrong character is used.
- You're never actually directly dealing with "characters" or "text", you're always dealing with bits as seen through several layers of abstractions. Incorrect results are a sign of one of the abstraction layers failing.
- If two systems are talking to each other, they always need to specify what encoding they want to talk to each other in. The simplest example of this is this website telling your browser that it's encoded in UTF-8.
- In this day and age, the standard encoding is UTF-8 since it can encode virtually any character of interest, is backwards compatible with the de-facto baseline ASCII and is relatively space efficient for the majority of use cases nonetheless.
 - Other encodings still occasionally have their uses, but you should have a concrete reason for wanting to deal with the headaches associated with character sets that can only encode a subset of Unicode.
- The days of one byte = one character are over and both programmers and programs need to catch up on this.

Percent-encoding (URL encoding)

Percent-encoding, also known as **URL encoding**, is a mechanism for encoding information in a Uniform Resource Identifier (URI) under certain circumstances. Although it is known as URL encoding it is, in fact, used more generally within the main Uniform Resource Identifier (URI) set, which includes both Uniform Resource Locator (URL) and Uniform Resource Name (URN).

Percent-encoding in a URI

Types of URI characters

The characters allowed in a URI are either **reserved** or **unreserved** (or a percent character as part of a percent-encoding).

Reserved characters are those characters that sometimes have special meaning. For example, forward slash / characters are used to separate different parts of a URL (or more generally, a URI).

Unreserved characters have no such meanings.

Using percent-encoding, reserved characters are represented using special character sequences. The sets of reserved and unreserved characters and the circumstances under which certain reserved characters have special meaning have changed slightly with each revision of specifications that govern URIs and URI schemes.

[RFC 3986](#) section 2.2 *Reserved Characters* (January 2005)

!	*	'	()	;	:	@	&	=	+	\$,	/	?	#	[]
---	---	---	---	---	---	---	---	---	---	---	----	---	---	---	---	---	---

[RFC 3986](#) section 2.3 *Unreserved Characters* (January 2005)

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

Percent-encoding reserved characters

When a character from the reserved set (a "reserved character") has special meaning (a "reserved purpose") in a certain context, and a URI scheme says that it is necessary to use that character for some other purpose, then the character must be percent-encoded.

Percent-encoding a reserved character involves:

1. converting the character to its corresponding byte value in ASCII
2. and then representing that value as a pair of hexadecimal digits. The digits, preceded by a percent sign (%) which is used as an **escape character**, are then used in the URI in place of the reserved character.

(For a **non-ASCII character**, it is typically converted to its byte sequence in UTF-8, and then each byte value is represented as above.)

In this example we are **NOT** escaping the chars, 'ü' is **NOT** a reserved character, we are just representing it in a different way (in Unicode) so every browser and server will be able to decode this character.

non-ASCII example:

U+00FC ü 11000011 10111100 → 11000011 10111100

C3

BC

Plain:
Encoded:

The reserved character `/`, for example, if used in the "path" component of a URI, has the special meaning of being a delimiter between path segments. If, according to a given URI scheme, `/` needs to be in a path segment, then the three characters `%2F` or `%2f` must be used in the segment instead of a raw `/`.

Reserved characters after percent-encoding																	
!	#	\$	&	'	()	*	+	,	/	:	;	=	?	@	[]
%21	%23	%24	%26	%27	%28	%29	%2A	%2B	%2C	%2F	%3A	%3B	%3D	%3F	%40	%5B	%5D

Reserved characters that have no reserved purpose in a particular context may also be percent-encoded but are not semantically different from those that are not.

In the "query" component of a URI (the part after a `?` character), for example, `/` is still considered a reserved character but it normally has no reserved purpose, unless a particular URI scheme says otherwise. The character does not need to be percent-encoded when it has no reserved purpose.

URIs that differ only by whether a reserved character is percent-encoded or appears literally are normally considered not equivalent (denoting the same resource) unless it can be determined that the reserved characters in question have no reserved purpose. This determination is dependent upon the rules established for reserved characters by individual URI schemes.

Percent-encoding the percent character

Because the percent character (`%`) serves as the indicator for percent-encoded octets, it must be percent-encoded as `%25` for that octet to be used as data within a URI.

Different Definition

URL encoding is the practice of translating unprintable characters or characters with special meaning within URLs to a representation that is unambiguous and universally accepted by web browsers and servers. These characters include –

It is a way to tell browsers and servers which character you send and how they should interpret it.

For example, we can tell the server/browser that we are using the forward slash `'` character NOT to separate files but just as an input. This is the whole point of URL-Encoding.

- **ASCII control characters** – Unprintable characters typically used for output control. Character ranges 00-1F hex (0-31 decimal) and 7F (127 decimal). A complete encoding table is given below.
- **Non-ASCII control characters** – These are characters beyond the ASCII character set of 128 characters. This range is part of the ISO-Latin character set and includes the entire "top half" of the ISO-Latin set 80-FF hex (128-255 decimal). A complete encoding table is given below.
- **Reserved characters** – These are special characters such as the dollar sign, ampersand, plus, common, forward slash, colon, semi-colon, equals sign, question mark, and "at" symbol. All of these can have different meanings inside a URL so need to be encoded. A complete encoding table is given below.
- **Unsafe characters** – These are space, quotation marks, less than symbol, greater than symbol, pound character, percent character, Left Curly Brace, Right Curly Brace, Pipe, Backslash, Caret, Tilde, Left Square Bracket, Right Square Bracket, Grave Accent. These character present the possibility of being misunderstood within URLs for various reasons. These characters should also always be encoded. A complete encoding table is given below.

The encoding notation replaces the desired character with three characters: a percent sign and two hexadecimal digits that correspond to the position of the character in the ASCII character set.

Common character encodings

- [ISO 646](#)
 - [ASCII](#)
- [EBCDIC](#)
 - [CP37](#)
 - [CP930](#)
 - [CP1047](#)
- [ISO 8859](#):
 - [ISO 8859-1](#) Western Europe
 - [ISO 8859-2](#) Western and Central Europe
 - [ISO 8859-3](#) Western Europe and South European (Turkish, Maltese plus Esperanto)
 - [ISO 8859-4](#) Western Europe and Baltic countries (Lithuania, Estonia, Latvia and Lapp)
 - [ISO 8859-5](#) Cyrillic alphabet
 - [ISO 8859-6](#) Arabic
 - [ISO 8859-7](#) Greek
 - [ISO 8859-8](#) Hebrew
 - [ISO 8859-9](#) Western Europe with amended Turkish character set
 - [ISO 8859-10](#) Western Europe with rationalised character set for Nordic languages, including complete Icelandic set
 - [ISO 8859-11](#) Thai
 - [ISO 8859-13](#) Baltic languages plus Polish
 - [ISO 8859-14](#) Celtic languages (Irish Gaelic, Scottish, Welsh)
 - [ISO 8859-15](#) Added the Euro sign and other rationalisations to [ISO 8859-1](#)
 - [ISO 8859-16](#) Central, Eastern and Southern European languages (Albanian, Bosnian, Croatian, Hungarian, Polish, Romanian, Serbian and Slovenian, but also French, German, Italian and Irish Gaelic)
- [CP437](#), [CP720](#), [CP737](#), [CP850](#), [CP852](#), [CP855](#), [CP857](#), [CP858](#), [CP860](#), [CP861](#), [CP862](#), [CP863](#), [CP865](#), [CP866](#), [CP869](#), [CP872](#)
- [MS-Windows character sets](#):
 - [Windows-1250](#) for Central European languages that use Latin script, (Polish, Czech, Slovak, Hungarian, Slovene, Serbian, Croatian, Bosnian, Romanian and Albanian)
 - [Windows-1251](#) for Cyrillic alphabets
 - [Windows-1252](#) for Western languages
 - [Windows-1253](#) for Greek
 - [Windows-1254](#) for Turkish
 - [Windows-1255](#) for Hebrew
 - [Windows-1256](#) for Arabic
 - [Windows-1257](#) for Baltic languages
 - [Windows-1258](#) for Vietnamese
- [Mac OS Roman](#)
- [KOI8-R](#), [KOI8-U](#), [KOI7](#)
- [MIK](#)
- [ISCII](#)
- [TSCII](#)
- [VISCII](#)
- [JIS X 0208](#) is a widely deployed standard for Japanese character encoding that has several encoding forms.
 - [Shift JIS](#) ([Microsoft Code page 932](#) is a dialect of [Shift_JIS](#))
 - [EUC-JP](#)
 - [ISO-2022-JP](#)
- [JIS X 0213](#) is an extended version of [JIS X 0208](#).
 - [Shift_JIS-2004](#)
 - [EUC-JIS-2004](#)
 - [ISO-2022-JP-2004](#)
- Chinese [Guobiao](#)
 - [GB 2312](#)
 - [GBK](#) ([Microsoft Code page 936](#))
 - [GB 18030](#)
- Taiwan [Big5](#) (a more famous variant is [Microsoft Code page 950](#))
 - Hong Kong [HKSCS](#)
- Korean
 - [KS X 1001](#) is a Korean double-byte character encoding standard
 - [EUC-KR](#)
 - [ISO-2022-KR](#)
- [Unicode](#) (and subsets thereof, such as the 16-bit 'Basic Multilingual Plane')
 - [UTF-8](#)
 - [UTF-16](#)
 - [UTF-32](#)
- [ANSEL](#) or [ISO/IEC 6937](#)

Questions

How can I display a Unicode Character above U+FFFF using a single char in Java?

Answer : We can't

- **char**: The char data type is a single **16-bit Unicode character**. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).
- The set of characters from U+0000 to U+FFFF is sometimes referred to as the **Basic Multilingual Plane (BMP)**.
- Characters whose code points are greater than U+FFFF are called **supplementary characters**.
- A char value, therefore, represents Basic Multilingual Plane (BMP) code points, including the surrogate code points, or code units of the UTF-16 encoding. An int value represents all Unicode code points, including supplementary code points.

Java uses **UTF-16**. A single Java `char` can only represent characters from the **basic multilingual plane**. Other characters have to be represented by a *surrogate pair* of two `char` s. This is reflected by API methods such as `String.codePointAt()`.

And yes, this means that a lot of Java code will break in one way or another when used with characters outside the basic multilingual plane.

Is there any reason why Java char primitive data type is 2 bytes unlike C which is 1 byte?

When Java was originally designed, it was anticipated that any Unicode character would fit in 2 bytes (16 bits), so `char` and `Character` were designed accordingly. In fact, a Unicode character can now require up to 4 bytes. Thus, UTF-16, the internal Java encoding, requires supplementary characters use 2 code units. Characters in the Basic Multilingual Plane (the most common ones) still use 1. A Java `char` is used for each code unit. This [Sun article](#) explains it well.

Comparing character encoding in C, C#, Java, Python and Ruby

Language	Type	Width (bits)	Implicit Encoding
C	char	8	implementation specific
	wchar_t	implementation specific (8+)	implementation specific
C#	char	16	UTF-16
	string	16 (char sequence)	UTF-16
Java	char	16	UTF-16
	String	16 (char sequence)	UTF-16
Python	str	8 (octet sequence)	ASCII (can be changed)
	unicode	16 or 32 (code unit sequence)	UCS2 (16) or UCS4 (32)
Ruby	String	8 (octet sequence)	none

Don't assume that the character handling conventions you've learnt in one language/platform will automatically apply in others. I've selected a cross-section of popular languages to contrast the different ways character encoding is handled.

What is the difference between a “line feed” and a “carriage return”?

A **line feed** means moving one line forward. The code is `\n`.

A **carriage return** means moving the cursor to the beginning of the line. The code is `\r`.

Windows editors often still use the combination of both as `\r\n` in text files. Unix uses mostly only the `\n`.

The separation comes from typewriter times, when you turned the wheel to move the paper to change the line and moved the carriage to restart typing on the beginning of a line. This was two steps.

What is the binary representation of "end of line" in UTF-8.

There are [a bunch](#):

- **LF** : Line Feed, [U+000A](#) (UTF-8 in hex: 0A)
- **VT** : Vertical Tab, [U+000B](#) (UTF-8 in hex: 0B)
- **FF** : Form Feed, [U+000C](#) (UTF-8 in hex: 0C)
- **CR** : Carriage Return, [U+000D](#) (UTF-8 in hex: 0D)
- **CR+LF** : CR ([U+000D](#)) followed by LF ([U+000A](#)) (UTF-8 in hex: 0D0A)
- **NEL** : Next Line, [U+0085](#) (UTF-8 in hex: C285)
- **LS** : Line Separator, [U+2028](#) (UTF-8 in hex: E280A8)
- **PS** : Paragraph Separator, [U+2029](#) (UTF-8 in hex: E280A9)

...and probably many more.

The most commonly used ones are `LF` (*nix), `CR+LF` (Windows and DOS), and `CR` (old pre-OSX Mac systems, mostly).

End-of-Text character

The End-of-Text character (ETX) (hex value of **0x03**, often displayed as ^C) is an **ASCII** control character used to inform the receiving computer that the end of the data stream has been reached. This may or may not be an indication that all of the data has been received.

It is often used as a "**break**" character (Control-C) to interrupt a program or process. In TOPS-20, it was used to gain the system's attention before logging in.

Unicode Character 'END OF TEXT' (U+0003).