

APPENDIX A



Character Encodings

A character is the basic unit of a writing system, for example, a letter of the English alphabet and an ideograph of an ideographic writing system such as Chinese and Japanese ideographs. In the written form, a character is identified by its shape, also known as *glyph*. The identification of a character with its shape is not precise. It depends on many factors, for example, a hyphen is identified as a minus sign in a mathematical expression and some Greek and Latin letters have the same shapes, but they are considered different characters in two written scripts. Computers understand only numbers, more precisely, only bits 0 and 1. Therefore, it was necessary to convert, with the advent of computers, the characters into codes (or bit combinations) inside the computer's memory, so that the text (sequence of characters) could be stored and reproduced. However, different computers may represent different characters with the same bit combinations, which may lead to misinterpretation of text stored by one computer system and reproduced by another. Therefore, for correct exchange of information between two computer systems, it is necessary that one computer system understand unambiguously the coded form of the characters represented in bit combinations produced by another computer system and vice versa. Before we begin our discussion of some widely used character encodings, it is necessary to understand some commonly used terms:

- An *abstract character* is a unit of textual information, for example, the Latin capital letter A ('A').
- A *character repertoire* is defined as the set of characters to be encoded. A character repertoire can be fixed or open. In a fixed character repertoire, once the set of characters to be encoded is decided, it is never changed. ASCII and POSIX portable character repertoires are examples of fixed character repertoires. In an open character repertoire, a new character may be added any time. Unicode and Windows Western European repertoires are examples of open character repertoires. The Euro currency sign and Indian Rupee sign were added to Unicode because it is an open repertoire.
- A *coded character set* is defined as a mapping from a set of non-negative integers (also known as code positions, code points, code values, character numbers, and code space) to a set of abstract characters. The integer that maps to a character is called the *code point* for that character, and the character is called an *encoded character*. A coded character set is also called a character encoding, coded character repertoire, character set definition, or code page. Figure A-1 depicts two different coded character sets; both of them have the same character repertoire, which is the set of three characters (A, B, and C) and the same code points, which is the set of three non-negative integers (1, 2, and 3).

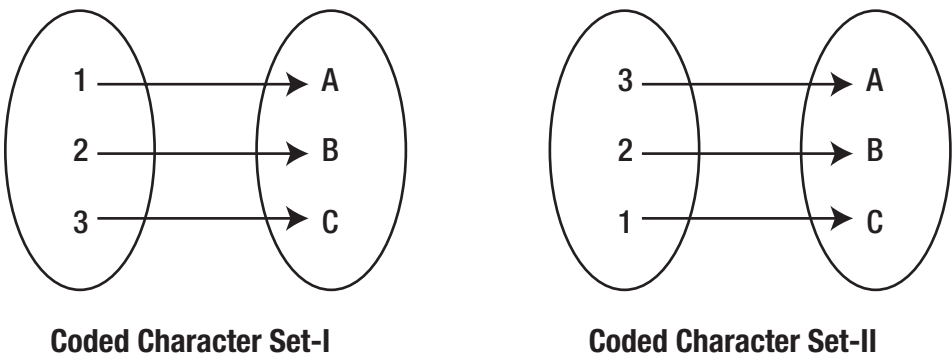


Figure A-1. Two coded character sets having the same character repertoire and code points

To define a coded character set, you need to specify three things:

- A set of code points
- A set of characters
- A mapping between the set of code points and the set of characters

The number of bits used to represent a character determines how many distinct characters can be represented in a coded character set. Some widely used coded character sets are outlined in the following sections.

ASCII

ASCII, the American Standard Code for Information Interchange, is a 7-bit coded character set. ASCII has 2^7 (=128) code points, and so it represents 128 distinct characters whose numeric values range from 0 (binary 0000000) to 127 (binary 1111111). The characters NUL and DELETE are represented by code points 0000000 and 1111111, respectively. There are historical reasons to assign these code points to NUL and DELETE characters. It was common to use punched paper tapes to store data for processing by the time ASCII was developed. A 1 bit was used to represent a hole on the paper tape, whereas a 0 bit represented the absence of a hole. Since a row of seven 0 bits would be indistinguishable from blank tape, the coding 0000000 would have to represent a NUL character, that is, the absence of any effect. Since holes, once punched, could not be erased but an erroneous character could always be converted into 111111, this bit pattern was adopted as the DELETE character.

ASCII uses the first 32 bit combinations (or code points) to represent control characters. This range includes the NUL character, but not the DELETE character. Therefore, it leaves 95 bit combinations for printing characters:

$$128(\text{All characters}) - 32(\text{Control Characters}) - 1(\text{DELETE}) = 95(\text{Printing characters})$$

All printing characters are arranged in the order that could be used for sorting purposes. The SPACE character is normally sorted before any other printing character. Therefore, the SPACE character is allocated the first position among the printing characters. The code point for the SPACE character in ASCII is 32, or 1100000. The code point range of 48–57 represents 0–9 digits, range 65–90 represents 26 uppercase letters A–Z, and range 97–122 represents 26 lowercase letters a–z. Modern computers use an 8-bit combination, also known as a *byte*, as the smallest unit for storage. Therefore, on modern computers, a 7-bit ASCII character uses 8 bits (or 1 byte) of memory, of which the most significant bit is always set to 0; for example, SPACE is stored as 01100000, and DELETE is stored as 01111111. Table A-1 contains the list of characters in the ASCII character set.

Table A-1. ASCII Character Set

Decimal	Hexadecimal	Binary	Character	Official Name
0	0	0	NUL	NULL
1	1	1	SOH	Start of heading
2	2	10	STX	Start of text
3	3	11	ETX	End of text
4	4	100	EOT	End of transmission
5	5	101	ENQ	Enquiry
6	6	110	ACK	Acknowledge
7	7	111	BEL	Bell
8	8	1000	BS	Backspace
9	9	1001	TAB	Horizontal tab
10	0A	1010	LF	Line feed (new line)
11	0B	1011	VT	Vertical tab
12	0C	1100	FF	Form feed (new page)
13	0D	1101	CR	Carriage return
14	0E	1110	SO	Shift out
15	0F	1111	SI	Shift in
16	10	10000	DLE	Data link escape
17	11	10001	DC1	Device control 1
18	12	10010	DC2	Device control 2
19	13	10011	DC3	Device control 3
20	14	10100	DC4	Device control 4
21	15	10101	NAK	Negative acknowledge
22	16	10110	SYN	Synchronous idle
23	17	10111	ETB	End of transmission block
24	18	11000	CAN	Cancel
25	19	11001	EM	End of medium
26	1A	11010	SUB	Substitute
27	1B	11011	ESC	Escape
28	1C	11100	FS	File separator
29	1D	11101	GS	Group separator
30	1E	11110	RS	Record separator

(continued)

Table A-1. *(continued)*

Decimal	Hexadecimal	Binary	Character	Official Name
31	1F	11111	US	Unit separator
32	20	100000	SP	Space
33	21	100001	!	Exclamation mark
34	22	100010	"	Quotation mark
35	23	100011	#	Number sign
36	24	100100	\$	Dollar sign
37	25	100101	%	Percent sign
38	26	100110	&	Ampersand
39	27	100111	'	Apostrophe
40	28	101000	(Left parenthesis
41	29	101001)	Right parenthesis
42	2A	101010	*	Asterisk
43	2B	101011	+	Plus sign
44	2C	101100	,	Comma
45	2D	101101	-	Hyphen/minus
46	2E	101110	.	Full stop/period
47	2F	101111	/	Solidus/slash
48	30	110000	0	Digit zero
49	31	110001	1	Digit one
50	32	110010	2	Digit two
51	33	110011	3	Digit three
52	34	110100	4	Digit four
53	35	110101	5	Digit five
54	36	110110	6	Digit six
55	37	110111	7	Digit seven
56	38	111000	8	Digit eight
57	39	111001	9	Digit nine
58	3A	111010	:	Colon
59	3B	111011	;	Semicolon
60	3C	111100	<	Less-than sign
61	3D	111101	=	Equals sign

(continued)

Table A-1. *(continued)*

Decimal	Hexadecimal	Binary	Character	Official Name
62	3E	111110	>	Greater-than sign
63	3F	111111	?	Question mark
64	40	1000000	@	Commercial at
65	41	1000001	A	Latin capital letter A
66	42	1000010	B	Latin capital letter B
67	43	1000011	C	Latin capital letter C
68	44	1000100	D	Latin capital letter D
69	45	1000101	E	Latin capital letter E
70	46	1000110	F	Latin capital letter F
71	47	1000111	G	Latin capital letter G
72	48	1001000	H	Latin capital letter H
73	49	1001001	I	Latin capital letter I
74	4A	1001010	J	Latin capital letter J
75	4B	1001011	K	Latin capital letter K
76	4C	1001100	L	Latin capital letter L
77	4D	1001101	M	Latin capital letter M
78	4E	1001110	N	Latin capital letter N
79	4F	1001111	O	Latin capital letter O
80	50	1010000	P	Latin capital letter P
81	51	1010001	Q	Latin capital letter Q
82	52	1010010	R	Latin capital letter R
83	53	1010011	S	Latin capital letter S
84	54	1010100	T	Latin capital letter T
85	55	1010101	U	Latin capital letter U
86	56	1010110	V	Latin capital letter V
87	57	1010111	W	Latin capital letter W
88	58	1011000	X	Latin capital letter X
89	59	1011001	Y	Latin capital letter Y
90	5A	1011010	Z	Latin capital letter Z
91	5B	1011011	[Left square bracket/opening square bracket
92	5C	1011100	\	Reverse solidus/backslash

(continued)

Table A-1. *(continued)*

Decimal	Hexadecimal	Binary	Character	Official Name
93	5D	1011101]	Right square bracket/closing square bracket
94	5E	1011110	^	Circumflex accent
95	5F	1011111	_	Low line/spacing underscore
96	60	1100000	`	Grave accent
97	61	1100001	A	Latin small letter A
98	62	1100010	B	Latin small letter B
99	63	1100011	C	Latin small letter C
100	64	1100100	D	Latin small letter D
101	65	1100101	E	Latin small letter E
102	66	1100110	F	Latin small letter F
103	67	1100111	G	Latin small letter G
104	68	1101000	H	Latin small letter H
105	69	1101001	I	Latin small letter I
106	6A	1101010	J	Latin small letter J
107	6B	1101011	K	Latin small letter K
108	6C	1101100	L	Latin small letter L
109	6D	1101101	M	Latin small letter M
110	6E	1101110	N	Latin small letter N
111	6F	1101111	O	Latin small letter O
112	70	1110000	P	Latin small letter P
113	71	1110001	Q	Latin small letter Q
114	72	1110010	R	Latin small letter R
115	73	1110011	S	Latin small letter S
116	74	1110100	T	Latin small letter T
117	75	1110101	U	Latin small letter U
118	76	1110110	V	Latin small letter V
119	77	1110111	W	Latin small letter W
120	78	1111000	X	Latin small letter X
121	79	1111001	Y	Latin small letter Y
122	7A	1111010	Z	Latin small letter Z
123	7B	1111011	{	Left curly bracket/opening curly bracket

(continued)

Table A-1. (continued)

Decimal	Hexadecimal	Binary	Character	Official Name
124	7C	1111100		Vertical line/vertical bar
125	7D	1111101	}	Right curly bracket/closing curly bracket
126	7E	1111110	~	Tilde
127	7F	1111111	DEL	DELETE

8-Bit Character Sets

The ASCII character set worked fine for the English language. Representing the alphabets from other languages, for example, French and German, led to the development of an 8-bit character set. An 8-bit character set defines 2^8 (or 256) character positions whose numeric values range from 0 to 255. The bit combination for an 8-bit character set ranges from 00000000 to 11111111. The 8-bit character set is divided into two parts. The first part represents characters, which are the same as in the ASCII character set. The second part introduces 128 new characters. The first 32 positions in the second part are reserved for control characters. Therefore, there are two control character areas in an 8-bit character set: 0–31 and 128–159. Since the SPACE and DELETE characters are already defined in the first part, an 8-bit character set can accommodate 192 printing characters ($95 + 97$), including SPACE. ISO Latin-1 is one example of an 8-bit character set.

Even an 8-bit character set is not large enough to accommodate most of the alphabets of all languages in the world. This led to the development of a bigger (may be the biggest) character set, which is known as the Universal Character Set (UCS).

Universal Multiple-Octet Coded Character Set (UCS)

The Universal Multiple-Octet Coded Character Set, simply known as UCS, is intended to provide a single coded character set for the encoding of written forms of all the languages of the world and of a wide range of additional symbols that may be used in conjunction with such languages. It is intended not only to cover languages in current use but also languages of the past and such additions as may be required in the future. The UCS uses a four-octet (one octet is 8 bits) structure to represent a character. However, the most significant bit of the most significant octet is constrained to be 0, which permits its use for private internal purposes in a data processing system. The remaining 31 bits allow us to represent more than two billion characters. The four octets are named as follows:

- The Group-Octet, or G
- The Plane-Octet, or P
- The Row-Octet, or R
- The Cell-Octet, or C

G is the most significant octet, and C is the least significant octet. So the whole code range for UCS is viewed as a four-dimensional structure composed of

- 128 groups
- 256 planes in each group
- 256 rows in each plane
- 256 cells in each row

Two hexadecimal digits (0–9, A–F) specify the values of any octet. The values of G are restricted to the range 00–7F. The plane with G=00 and P=00 is known as the Basic Multilingual Plane (BMP). The row of BMP with R=00 represents the same set of characters as 8-bit ISO Latin-I. Therefore, the first 128 characters of ASCII, ISO Latin-1, and BMP with R=00 match. Characters 129th to 256th of ISO Latin-I and those of BMP with R=00 match. This makes UCS compatible with the existing 7-bit ASCII and 8-bit ISO Latin-I. Further, BMP has been divided into five zones:

- *A-zone*: It is used for alphabetic and symbolic scripts together with various symbols. The code position available for A-zone ranges from 0000 to 4DFF. The code positions 0000–001F and 0080–009F are reserved for control characters. The code position 007F is reserved for the DELETE character. Thus, it has 19903 code positions available for graphics characters.
- *I-zone*: It is used for Chinese/Japanese/Korean (CJK) unified ideographs. Its range is 4E00–9FFF, so 20992 code positions are available in this zone.
- *O-zone*: It is used for Korean Hangul syllabic scripts and for other scripts. Its range is A000–D7FF, so 14336 code positions are available in this zone.
- *S-zone*: It is reserved for use with transformation format UTF-16. The transformation format UTF-16 is described shortly. Its range is D800–DFFF, so 2048 code positions are available in this zone.
- *R-zone*: It is known as the restricted zone. It can be used only in special circumstances. One of the uses of this zone is for specific user-defined characters. However, in this case an agreement is necessary between the sender and the recipient to communicate successfully. Its range is E000–FFFF, so 8190 code positions are available in this zone.

UCS is closely related to another popular character set called Unicode, which has been prepared by the Unicode Consortium. Unicode uses a two-octet (16 bits) coding structure, and hence it can accommodate 2^{16} (= 65536) distinct characters. Unicode can be considered as the 16-bit coding of the BMP of UCS. These two character sets, Unicode and UCS, were developed and are maintained by two different organizations. However, they cooperate to keep Unicode and UCS compatible. If a computer system uses the Unicode character set to store some text, each character in the text has to be allocated 16 bits even if all characters in the text are from the ASCII character set. Note that the first 128 characters of Unicode match with those of ASCII, and a character in ASCII can be represented only in 8 bits. So to use 16 bits to represent all characters in Unicode is wasteful of computer memory. An alternative would be to use 8 bits for all characters from ASCII and 16 bits for characters outside the range of ASCII. However, this method of using different bits to represent different characters from Unicode has to be consistent and uniform, resulting in no ambiguity when data is stored or interchanged between different computer systems. This issue led to the development of the character encoding methods. Currently, there are four character encoding methods specified in ISO/IEC 10646-1:

- UCS-2
- UCS-4
- UTF-16
- UTF-8

UCS-2

This is a two-octet BMP form of encoding, which allows the use of two octets to represent a character from the BMP. This is a fixed-length encoding method. That is, each character from BMP is represented by exactly two octets.

UCS-4

This encoding method is also called the four-octet canonical form of encoding, which uses four octets for every character in UCS. This is also a fixed-length encoding method.

UTF-16 (UCS Transformation Format 16)

Once characters outside the BMP are used, the UCS-2 encoding method cannot be applied to represent them. In this case, the encoding must switch over to use UCS-4, which will just double the use of resources, such as memory, network bandwidth, etc. The transformation format UTF-16 has been designed to avoid such a waste of memory and other resources, which would have resulted in using the UCS-4 encoding method. The UTF-16 is a variable-length encoding method. In the UTF-16 encoding method, UCS-2 is used for all characters within BMP, and UCS-4 is used for encoding the characters outside BMP.

UTF-8 (UCS Transformation Format 8)

This is a variable-length encoding method, which may use one to six octets to represent a character from UCS. All ASCII characters are encoded using one octet. In the UTF-8 format of character encoding, characters are represented using one or more octets, as shown in Table A-2.

Table A-2. List of Legal UTF-8 Sequences

Number of Octets	Bit Patterns Used	UCS Code
1	Octet 1: 0xxxxxxx	00000000–0000007F
2	Octet 1: 110xxxxx Octet 2: 10xxxxxx	00000080–000007FF
3	Octet 1: 1110xxxx Octet 2: 10xxxxxx Octet 3: 10xxxxxx	00000800–0000FFFF
4	Octet 1: 11110xxx Octet 2: 10xxxxxx Octet 3: 10xxxxxx Octet 4: 10xxxxxx	00010000–001FFFFF
5	Octet 1: 111110xx Octet 2: 10xxxxxx Octet 3: 10xxxxxx Octet 4: 10xxxxxx Octet 5: 10xxxxxx	00200000–03FFFFFF
6	Octet 1: 1111110x Octet 2: 10xxxxxx Octet 3: 10xxxxxx Octet 4: 10xxxxxx Octet 5: 10xxxxxx Octet 6: 10xxxxxx	04000000–7FFFFFFF

The “x” in the table indicates either a 0 or a 1. Note that, in UTF-8 format, an octet that starts with a 0 bit indicates that it is representing an ASCII character. An octet starting with 110 bit combinations indicates that it is the first octet of the two-octet representation of a character. And so on. Also note that, when an octet is part of a multi-octet character representation, the octet other than the first one starts with a 10-bit pattern. Security checks can be easily implemented for UTF-8 encoded data. UTF-8 octet sequences, which do not conform to the octet sequences shown in the table, are considered invalid.

Java and Character Encodings

Java stores and manipulates all characters and strings as Unicode characters. In serialization and bytecodes, Java uses the UTF-8 encoding of the Unicode character set. All implementations of the Java virtual machine are required to support the character encoding methods, as shown in Table A-3.

Table A-3. *List of the Supported Character Encodings by a JVM*

Character Encoding	Description
ASCII	7-bit ASCII (also known as ISO-646-US, the basic Latin block of the Unicode character set).
ISO-8859-1	ISO Latin Alphabet No. 1 (also known as ISO Latin-1).
UTF-8	8-bit Unicode Transformation Format.
UTF-16BE	16-bit Unicode Transformation Format, big-endian byte order. Big-endian is discussed in Chapter 3.
UTF-16LE	16-bit Unicode Transformation Format, little-endian byte order. Little-endian is discussed in Chapter 3.
UTF-16	16-bit Unicode Transformation Format, byte order specified by a mandatory initial byte order mark (either order accepted on input, big-endian used on output).

Java supports UTF-8 format with the following two significant modifications:

- Java uses 16 bits to represent a NUL character in a class file, whereas standard UTF-8 uses only 8 bits. This compromise has been made to make it easier for other languages to parse a Java class file where a NUL character is not allowed within a string. However, in some cases, Java uses standard UTF-8 format to represent the NUL character.
- Java recognizes only one-octet, two-octet, and three-octet UTF-8 formats, whereas standard UTF-8 format may use one-octet, two-octet, three-octet, four-octet, five-octet, and six-octet sequences. This is because Java supports the Unicode character set, and all characters from Unicode can be represented in one-, two-, or three-octet formats of UTF-8.

When you compile the Java source code, by default, the Java compiler assumes that the source code file has been written using the platform’s default encoding (also known as local code page or native encoding). The platform’s default character encoding is Latin-1 on Windows and Solaris and MacRoman on Mac. Note that Windows does not use true Latin-1 character encoding. It uses a variation of Latin-1 that includes fewer control characters and more printing characters. You can specify a file encoding name (or code page name) to control how the compiler interprets characters beyond the ASCII character set. At the time of compiling

your Java source code, you can pass the character encoding name used in your source code file to the Java compiler. The following command tells the Java compiler (`javac`) that the Java source code `Test.java` has been written using a traditional Chinese encoding named Big5. Now, the Java compiler will convert all characters encoded in Big5 to Unicode:

```
javac -encoding Big5 Test.java
```

The JDK since version 9 supports UTF-8-based properties resource bundles. There is a rare issue that could arise where an ISO-8859-1 properties file could be recognized as a valid UTF-8 file. To accommodate for this, the JDK provides a way to designate the encoding of resource bundles either “ISO-8859-1” or “UTF-8”, by setting the system property `java.util.PropertyResourceBundle.encoding` to either value.