Codifica dei caratteri

Codifica numerica dei caratteri

Per la elaborazione automatica oltre alle informazioni numeriche è necessario codificare anche informazioni non numeriche come ad esempio i caratteri tipografici.

La rappresentazione interna delle informazioni in un sistema di elaborazione deve comunque essere numerica e binaria; si quindi deve definire una corrispondenza tra ogni simbolo tipografico ed una particolare sequenza di bit in modo che in programmi possano riconoscere ed elaborare il carattere.

In base al numero di caratteri che si vogliono rappresentare sarà necessario individuare il numero minimo di bit che ne permetta la rappresentazione.

Se per esempio fosse sufficiente rappresentare solo le lettere dell'alfabeto maiuscole basterebbero 5 bit:

$\mathbf{2^5=32}$ combinazioni **>** **26** caratteri dell'alfabeto internazionale.

In realtà i simboli da codificare sono molti di più perché ogni lettera ha due rappresentazioni (maiuscola e minuscola) e si devono aggiungere molti simboli di interpunzione e altri simboli ausiliari come ad esempio le 10 cifre numeriche da '0' a '9'.

Il campo della codifica numerica dei caratteri è ulteriormente ampliato dalla esistenza dei caratteri nazionali specifici di ogni lingua (come ad esempio i caratteri accentati dell'italiano).

Infine è necessario codificare anche le lingue orientali che usano simboli diversi da quelli dell'alfabeto internazionale.

Le codifiche devono essere definite da enti internazionali di standardizzazione in modo che tutti i sistemi di elaborazione indipendentemente dall'hardware dal software che utilizza tali codifiche.

Codifica ASCII

Il primo standard che è stato codificato dalla "**American Standard Code for Information Interchange**" (ASCII pronuncia *æski*) risale al 1960 ma è tuttora in uso perché, per garantire la compatibilità le sue codifiche sono state recepite in tutti gli standard successivi.

Lo standard codifica solo i caratteri internazionali ed alcuni caratteri di interpunzione usando una codifica a 7 bit e consentendo quindi la rappresentazione di 128 caratteri.

La seguente tabella mostra le associazioni fissate dallo standard ASCII:

Bits b_4	b_3	b_2	b_1	Row ↓ / Column →	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FC	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

La tabella a doppia entrata mostra sulle righe i quattro bit meno significativi e sulle colonne i tre bit più significativi del pacchetto di sette bit che codifica ogni carattere; incrociando i valori delle due parti si trova il simbolo rappresentato.

Attenzione!!! Diversamente dalle abituali rappresentazioni informatica il pedice di posizionamento dei bit parte da 1 e va a 7 mentre normalmente il pedice viene indicato con partenza da 0.

Caratteri stampabili e caratteri di controllo

In alcuni casi il simbolo è un normale simbolo tipografico mentre in altri casi è una sigla. Non tutte le codifiche corrispondono ad effettivi caratteri tipografici; alcuni caratteri vengono definiti "caratteri di controllo" mentre i normali caratteri tipografici vengono definiti "caratteri stampabili".

La distinzione mette in evidenza il differente comportamento che questi codici hanno sui dispositivi di I/O (tastiera, monitor, stampante, modem ecc).

I codici corrispondenti ai caratteri stampabili quando vengono inviati su un dispositivo di uscita producono la presentazione del simbolo codificato.

Invece i codici corrispondenti ai caratteri di controllo quando vengono inviati su un dispositivo di uscita producono un'azione di controllo del dispositivo. Ad esempio l'invio del carattere di controllo LF (0001010) su un monitor o su una stampante produce l'avanzamento alla riga successiva (in ambiente windows è necessaria la coppia di caratteri CR+LF. L'invio del carattere dicontrollo SP (0100000) produce la generazione di uno spazio vuoto.

Molti dei codici di controllo non servono per la comunicazione con i dispositivi locali ma per la comunicazione con dispositivi remoti.

Ordinamento lessicografico

Il posizionamento dei simboli nella codifica non è casuale ma segue un ordine numerico che facilita la realizzazione degli algoritmi di ricerca ed ordinamento.

Ad esempio i caratteri alfabetici maiuscoli sono posizionati a partire dal codice:

'A' = 1000001 (65)
'B' = 1000010 (66)
…
'Z' = 1011010 (90)

Questo ordinamento numerico consente di dire che 'A' precede 'B', … 'Z'.

I caratteri alfabetici minuscoli sono invece posizionati a partire dal codice:

'a' = 1100001 (97)
'b' = 1100010 (98)
…
'z' = 1111010 (122)

Anche in questo caso 'a' precede 'b' ...'z' e confrontando con le maiuscole si vede che tutte le maiuscole precedono tutte le minuscole.

Esiste però una relazione numerica tra un carattere maiuscolo ed il corrispondente carattere minuscolo. Questa relazione non si vede bene considerando il loro codice in decimale ma risulta evidente in binario: le due versione di uno stesso carattere hanno la stessa codifica numerica ad eccezione del sesto bit che per le maiuscole è sempre 0 mentre per le minuscole è sempre 1.

Si può quindi pensare ad un algoritmo che traduca le maiuscole in minuscole e viceversa:

conversione da maiuscola a minuscola:
    minuscola = maiuscola or 0100000
conversione da minuscola a maiuscola:
    maiuscola = minuscola and 1011111

Ordinamento delle cifre

Anche il posizionamento dei simboli delle cifre non è casuale ma segue un ordine numerico che facilita gli algoritmi di conversione da stringa numerica a numero e viceversa. Si ricorda infatti che in "numeri" inseriti attraverso il canale di ingresso e mostrati sul canale di uscita non sono realmente numeri ma "stringe numeriche" formate di simboli tipografici numerici mentre i dati interni di elaborazione sono numeri con un formato che dipende dal tipo di dato (int, float, double ecc)

I caratteri numerici sono posizionati a partire dal codice:

'0' = 0110000 (48)
'1' = 0110001 (49)
…
'9' = 0111001 (57)

Questo ordinamento numerico consente di dire che '0' è più piccolo di '1' ... '9'.

L'ordinamento consente anche una rapida conversione tra simbolo e corrispondente numero e viceversa. Anche in questo caso la relazione non si vede bene considerando il loro codice in decimale ma risulta evidente in binario: la parte bassa della codifica (quattro bit meno significativi) corrisponde al numero naturale che rappresenta mentre la parte alta è fissa per tutte le cifre (110)

Si può quindi pensare ad un algoritmo di conversione da carattere a numero e viceversa:

conversione da carattere a numero
    numero = carattere and 0001111
conversione da numero a carattere
    carattere = numero or   0110000

Rappresentazione esadecimale delle codifiche binarie

Come si vede dalle precedenti trasformazioni le relazioni tra le informazioni sono molto più evidenti nella rappresentazione binaria rispetto a quella decimale. Il problema deriva dal fatto che la base 10 del sistema decimale non è multiplo della base 2 del sistema binario.

La rappresentazione binaria ha però il problema di essere poco leggibile perché a causa della base di piccolo valore le codifiche sono formate da lunghe sequenze di simboli difficili da ricordare ed analizzare.

Per rendere più agevole l'analisi e l'interpretazione dei codici gli sviluppatori informatici usano spesso basi "di lavoro" che sono più grandi per avere rappresentazioni più compatte e facilmente leggibili ma che sono multiple di 2 in modo da conservare una stretta relazione con il sistema binario. Le due basi normalmente usate sono la base 16 (esadecimale) e la base 8 (ottale).

Nel caso della base 16 si hanno 16 simboli numerici che sono costituiti dai dieci simboli della base 10 (0...9) e dalle prime sei lettere dell'alfabeto (A...F)

Si ottiene quindi la seguente corrispondenza:

Codifica binaria	*Simbolo esadecimale*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Si crea quindi una corrispondenza tra pacchetti di quattro bit e singole cifre esadecimali rendendo rapida la conversione da un formato all'altro a patto di ricordarsi le prime 16 sequenze binarie.
Ad esempio la sequenza binaria:
001111101000
rappresenta il decimale 1000 ma la conversione non è immdiata.
E' invece immediato il passaggio alla rappresentazione esadecimale:
0011 - 1110 - 1000
3 E 8 H
Per mettere in evidenza che si tratta di un numero esadecimale si termina il numero con il carattere H ma in molti linguaggi di programmazione, compreso Java, l'inserimento di un numero

esadecimale nel programma avviene con la notazione "0x..." quindi in questo caso il numero sarebbe: 0x3E8.
Ad esempio:

```
int miavar=0x3E8;
```

Applicando questa regola ai codici alfabetici e numerici si ottiene:
Per le maiuscole:
'A' = 1000001 (41H oppure 0x41)
'B' = 1000010 (42H)
…
'Z' = 1011010 (5AH)
Per le minuscole:
'a' = 1100001 (61H)
'b' = 1100010 (62H)
…
'z' = 1111010 (7AH)
Per le cifre:
'0' = 0110000 (30H)
'1' = 0110001 (31H)
…
'9' = 0111001 (39H)

Questa rappresentazione rende più evidente il significato dell'ordinamento e delle relative conversioni.

Codifica Extended ASCII (ANSI)

La rappresentazione a ASCII a 7 bit occupa in un sistema informatico comunque un byte che è la unità minima di memorizzazione lasciando l'ottavo bit inutilizzato (sempre a 0).

Dalla codifica ASCII sono state ricavate diverse codifiche "Extended ASCII" anche chiamate ANSI dall'ente American National Standard Institute che per primo le ha definite.

Queste codifiche utilizzano l'ottavo bit non usato dalla codifica ASCII per rappresentare caratteri nazionali aggiuntivi.

Con 8 bit si possono rappresentare 256 caratteri.

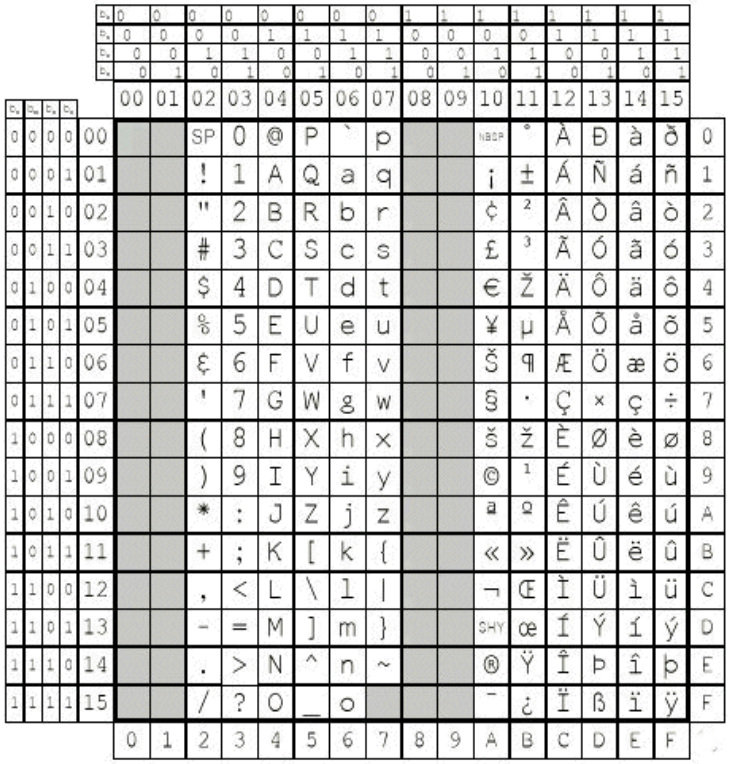
I primi 128 caratteri di una Extended ASCII coincidono sempre con i caratteri ASCII mentre i rimanenti 128 possono cambiare in base alle norme locali.

Le varie codifiche Extended ASCII vengono anche chiamate "codepage" perché i sistemi operativi, che nel loro nucleo devono essere indipendenti dalla codifica possono essere "localizzati" cambiando la "codepage" usata per rappresentare i caratteri.

Lo standard è stato successivamente recepito dalla ISO (International Standardization Organization) che ha definito le varie codifiche locali.

Per l'Europa occidentale è stato definito lo standard ISO-8859-1 che consente la rappresentazione completa della maggior parte delle lingue occidentali basate sull'alfabeto latino.

In alternativa può essere usato lo standard ISO-8859-15 che introduce poche modifiche rispetto al precedente (la più siginficativa è la presenza del simbolo euro €)



	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0000			SP	0	@	P	`	p			NBSP	°	À	Ð	à	ð	0
0001			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0010			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0011			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0100			$	4	D	T	d	t			€	Ž	Ä	Ô	ä	ô	4
0101			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0110			&	6	F	V	f	v			Š	¶	Æ	Ö	æ	ö	6
0111			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1000			(	8	H	X	h	x			š	ž	È	Ø	è	ø	8
1001			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1010			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1011			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1100			,	<	L	\	l	|			¬	Œ	Ì	Ü	ì	ü	C
1101			-	=	M	]	m	}			SHY	œ	Í	Ý	í	ý	D
1110			.	>	N	^	n	~			®	Ÿ	Î	Þ	î	þ	E
1111			/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Esistono anche codifiche "proprietarie" che in genere sono estensioni degli standard. Ad esempio sui sistemi operativi Windows al posto di ISO-8859-x si trova una codifica ANSI di nome "Windows-1252".

Codifica unicode

La rappresentazione ad 8 bit dei caratteri costringe ad adottare le "codepage" e quindi non è possibile avere contemporaneamente la codifica in set di caratteri diversi perché i codici si sovrappongono.

Per superare questo ostacolo sono state sviluppate negli anni '80 codifiche che superano il limite degli 8 bit e queste codifiche sono state raccolte nello standard UNICODE.

Questo standard si pone l'obiettivo di poter rappresentare i set di caratteri di tutte le lingue contemporaneamente e al tempo stesso di ridurre al mimino possibile l'occupazione di memoria e la complessità di elaborazione.

La impostazione di base prevede la possibilità di rappresentare un simbolo attraverso una codifica a 32 bit. Ogni simbolo rappresentato a 32 bit viene chiamato "codepoint" e viene descritto con la notazione "U-numero esadecimale" quindi un codepoint va da U-0x00000000 a U-0xFFFFFFFF. L'insieme dei caratteri così ottenuti si chiama UCS (Universal Character Set).

Per ottimizzare la memorizzazione, la comunicazione e la interpretazione però lo standard si divide in tre sotto-codifiche UTF (UCS Transformation Format):

- UTF-8: è una codifica a lunghezza variabile in cui i caratteri del set ASCII sono rappresentati da un solo byte, quelli delle codepage più comuni da due byte ed i rimanenti caratteri meno comuni da tre byte a quattro byte. Questa codifica è particolarmente usata nelle comunicazione come ad esempio la produzione di pagine web.
- UTF-16: è una codifica a lunghezza variabile in cui i caratteri delle lingue più comuni (BMP = Basic Multilingual Plane) sono rappresentati da due byte ed i rimanenti caratteri sono rappresentati da quattro byte. Questa codifica è spesso usata nelle elaborazioni applicative; ad esempio questa è la codifica interna di Java.

- UTF-32: è una codifica a lunghezza fissa di 4 byte che consente la contemporanea rappresentazione dei codici di tutte le lingue. E' di uso non molto frequente perché porta ad un notevole impiego di memoria.

Codifica utf-8

UTF-8 è il charset preferenziale nelle comunicazione ed è di norma usato nella generazione delle pagine web. Questa scelta consente di produrre pagine con un set di caratteri internazionali indipendenti dalla localizzazione. L'inserimento nella pagina web della meta-tag <meta charset="UTF-8" /> consente alla maggior parte dei browser di selezionare questo charset per la presentazione.

UTF-8 codifica i codepoint del set di caratteri Unicode utilizzando da 2 a 4 byte.

I codepoint coni più bassi valori numerici che corrispondono a codici di charset pre-esistenti ad Unicode che in pratica si verificano più spesso sono codificati utilizzando un minor numero di byte.

I primi 128 caratteri di Unicode, che corrispondono uno-a-uno con i caratteri ASCII, sono codificati utilizzando un singolo byte o con il valore binario stesso ASCII, mantenendo quindi la validità del codice ASCII nella codifica UTF-8.

Lo schema di codifica UTF-8 è indicato nella seguente tabella:

Bits	Last codepoint	Byte 1	Byte 2	Byte 3	Byte 4
7	U+007F	0xxxxxxx			
11	U+07FF	110xxxxx	10xxxxxx		
16	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
21	U+1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Le caratteristiche salienti del sistema sono i seguenti:

- I codici ad 1 byte sono utilizzati solo per i valori ASCII da 0 a 127. In questo caso i codici UTF-8 hanno lo stesso valore dei codici ASCII. Il bit più significativo di questi codici è sempre 0.

- I codepoint maggiori di 127 sono rappresentati da sequenze multi-byte, composte da un byte iniziale e da uno o più bytes di continuazione.
- Nel caso di sequenze a due byte il primo byte inizia sempre con un 1 e questo consente di distinguerlo dai caratteri a singolo byte che cominciano sempre per 0. Il secondo bit vale sempre 1 e questo consente di distinguerlo dai byte di continuazione che hanno il secondo byte sempre a 0. Il terzo bit vale sempre 0 e questo consente di distinguerlo dai multibyte tripli e quadrupli che hanno il bit ad 1. In conclusione il primo byte di una sequenza a due byte deve iniziare per 110 lasciando 5 bit per la codifica del carattere e continua con un secondo byte che deve cominciare per 10 lasciando altri 6 bit per la codifica del carattere per un totale di 11 bit. Se il codice ha meno di 11 bit significativi viene allineato a destra riempendo di 0 a sinistra
- Nel caso di sequenze a tre byte la sequenza iniziale è 1110 mentre nel caso delle sequenze a quattro byte è 11110. Si ottengono rispettivamente 16 e 21 bit per la codifica sempre con allineamento a destra e riempimento di 0.
- Il numero di 1 del primo byte consente di individuare automaticamente il numero dei byte che seguono; la presenza della sequenza 10 nei byte di continuazione consente di fare un controllo della correttezza della codifica (ridondanza)

Il principale vantaggio di utf-8 rispetto ad utf-16 è di minimizzare la quantità di bit utilizzati in codifica a parità di quantità di informazione e questo costituisce un vantaggio soprattutto nelle comunicazioni; infatti utf-8 è usato soprattutto per internazionalizzare le pagine web.
Il principale svantaggio consiste in una complicazione dell'algoritmo generazione e di interpretazione a causa della elevata probabilità di avere lunghezze variabili.

Esempi:

Consideriamo il carattere 'A'.
Nella codifica ASCII ha il valore:
01000001 = 41H
Nella codifica Unicode secondo la rappresentazione UCS avrebbe il codice:
U+0x00000041
poiché tutti i bit sono a 0 fino al bit 7 viene codificato in UTF-8 con un singolo byte coincidente con il valore ASCII.
01000001 = 41H

Consideriamo il carattere 'é'
Nella codifica ASCII non esiste
Nella codifica Extended ASCII (ANSI) sia nella versione ISO-8859-1 che nella versione ISO-8869-15 ha codifica:
11101001 = E9H
Nella codifica Unicode secondo la rappresentazione UCS avrebbe il codice:
U+0x000000E9
poiché il bit 7 vale 1 deve essere scomposto in due byte. Il primo byte quindi deve iniziare per 110 mentre il byte di continuazione deve iniziare per 10; gli 8 bit che formano la codifica del numero sono suddivisi tra gli 11 bit disponibili allineando a destra e riempendo di 0 a sinistra.
110*00011* - **10***101001* = C3H - A9H

Consideriamo il carattere '€'
Nella codifica ASCII non esiste
Nella codifica Extended ASCII (ANSI) la situazione dipende dalla codepage:
nella versione ISO-8859-1 non esiste
nella versione ISO-8859-15 ha codifica:
11101001 = A4H
nella versione Winsow-1252 ha codifica:
10000000 = 80H
Nella codifica Unicode secondo la rappresentazione UCS avrebbe il codice:

U+0x000020AC
(0000-0000-0000-0000-00**10-0000-1010-1100**)
la parte significativa del codice occupa 14 bit quindi richiede una codifica a 3 byte.
1110*0010* - **10***000010* - **10***101100* = E2H - 82H - ACH
il primo byte inizia con 1110 e può contenere 4 bit di cui due sono riempiti di 0 perchè il codice ha solo 14 bit mentre la codifica ne può contenere 16; il secondo e terzo byte inziano per 10 e contengono i rimanenti 12 bit.

Codifica utf-16

UTF-16 è un charset comunemente utilizzato nei sistemi operativi e nelle applicazioni in sostituzione delle codepage ANSI.

UTF-16 codifica i codepoint del set di caratteri Unicode utilizzando da 2 o 4 byte.

Lo spazio di codice Unicode è diviso in diciassette "piani" di 2^{16} (65.536) codepoint ciascuno. I codepoint di ogni piano hanno valori i esadecimali da xx0000 a xxFFFF, dove xx è un valore esadecimale 00-10 che indica il piano di appartenenza mentre la parte rimanente si ripete in piani diversi.

I codepoint di uso più frequente sono codificati nel "*Basic Multilingual Plane*" (BMP) che occupa i codici Unicode nei campi:

da U-0x0000 a U-0xD7FF e da U-0xE000 a U0xFFFF

Questi codepoint sono rappresentati in UTF16 con due byte ed hanno una rappresentazione coincidente con i due byte meno significativi della rappresentazione UCS con la rimozione dei 16 bit non significativi a 0.

Lo spazio di codifica da U-0xD800 a U-0xDFFF del BMP è riservato e non utilizzabile per rappresentare caratteri perché è usato per realizzare la parte alta della codifica dei rimanenti 16 piani di codifica (Supplementary planes).

L'algoritmo di generazione dei codici supplementari che vengono chiamati "coppie surrogate":

- Dal codepoint UCS si sottrae 0x10000 ottenendo un numero a 20 bit che va da 0x00000 a 0xFFFFF

- i 10 bit più significativi (da 0x000 a 0x3FF) si sommano a 0xD800 ottenendo un numero che va da 0xD800 a 0xDBF che rappresenta la parte alta del codice supplementare.
- i 10 bit meno significativi (da 0x000 a 0x3FF) si sommano a 0xDC00 ottenendo un numero da 0xDC00 a 0xDFFF che rappresenta la parte bassa del codice supplementare.

Poiché ogni elemento di informazione di UTF-16 è formato da almeno due byte è anche necessario definire in che ordine compaiono i due byte. Sono quindi possibili due codifiche distinte:

- Little Endian (UTF-16LE): ogni elemento da 16 bit è rappresentato prima dalla parte meno significativa seguita dalla parte più significativa. Ad esempio il carattere 'A' è scritto come 41-00
- Big Endian (UTF-16BE): ogni elemento da 16 bit è rappresentato prima dalla parte più significativa seguita dalla parte meno significativa. Ad esempio il carattere 'A' è scritto come 00-41

E' quindi necessario inserire prima di un testo UTF-16 una sequenza che consenta di riconoscere quale disposizione è stata adottata.
Nel caso di LE il testo deve iniziare con il codepoint FF-FE mentre nel caso BE deve iniziare con il codepoint FE-FF.

Il principale svantaggio è di avere rispetto ad utf-8 una quantità di bit di codifica a parità di quantità di informazione.

Riprendendo gli esempi precedenti si vede che tutti i caratteri considerati appartengono al BMP e quindi sono rappresentati con due byte:

Consideriamo il carattere 'A'.
Nella codifica Unicode secondo la rappresentazione UCS avrebbe il codice:
U+0x00000041
(0000-0000-0000-0000-0000-0000-**0100-0001**)
poiché i 16 bit più significativi sono tutti a 0 ricade nel BMP e può essere codificato in utf-16 come:
0x0041 (0000-0000-**0100-0001**)

Consideriamo il carattere 'é'
Nella codifica Unicode secondo la rappresentazione UCS avrebbe il codice:
U+0x000000E9
(0000-0000-0000-0000-0000-0000-**1110-1001**)
poiché i 16 bit più significativi sono tutti a 0 ricade nel BMP e può essere codificato in utf-16 come:
0x00E9 (0000-0000-**1110-1001**)

Consideriamo il carattere '€'
Nella codifica Unicode secondo la rappresentazione UCS avrebbe il codice:
U+0x000020AC
(0000-0000-0000-0000-00**10-0000-1010-1100**)
poiché i 16 bit più significativi sono tutti a 0 ricade nel BMP e può essere codificato in utf-16 come:
0x20AC (**0010-0000-1110-1001**)

Tutti i caratteri considerati fin'ora si trovano nel BMP e quindi sono codificati a 16 bit.
La maggior parte dei caratteri usati comunemente risiede nel BMP. Si deve tenere presente che il BMP dispone di 65536 - 2048 = 63488 caratteri. Con meno di 300 caratteri si codificano quasi tutti i caratteri della maggior parte delle lingue "occidentali" (quelle che usano l'alfabeto latino); rimangono quindi oltre 63000 caratteri per rappresentare le lingue orientali.
Consideriamo ora un carattere la cui codifica eccede dai 16 bit e quindi si trova in un supplementary plane.
Si tratta di un ideogramma facente parte del sistema CJK (Chinese-Japanese-Korean) che non fa parte del BMP ma ha un glifo che lo rappresenta nel nostro set di caratteri:

U-0002A6D6 = 

Si sottrae a 0x0002A6D6 il valore 0x00010000 ottenendo il valore: 0x0001A6D6

Si converte in binario limitando la sequenza a 20 bit
0x00**01A6D6** = 0000-0000-0000-**0001-1010-0110-1101-0110**
Si sommano a 0xD800 i 10 bit più significativi
0xD800 = 1101-1000-0000-0000 +
0000-0000-0110-1001 =
1101-1000-0110-1001 = 0xD869
ottenendo la parte alta della coppia surrogata
Si sommano a 0xDC00 i 10 bit meno significativi
0xDC00 = 1101-1100-0000-0000 +
0000-0010-1101-0110 =
1101-1110-1101-0110 = 0xDED6
ottenendo la parte bassa della coppia surrogata
Si ottengono così i due codici UTF-16:
UTF-16LE = 69-D8-D6-DE
UTF-16BE = D8-69-DE-D6

Codifica utf-32

Le due codifiche precedenti utilizzano la codifica a lunghezza variabile per ottimizzare le dimensioni dei codici.
Nel caso dell'UTF-32 invece viene usata una codifica a lunghezza fissa di 32 bit in cui i codici dei caratteri corrispondono ai codici UCS.
Anche nel caso di UTF-32 sono possibili le due codifiche Little Endian e Big Endian.
La codifica UTF-32 è poco usata a causa della considerevole occupazione di memoria.
Il vantaggio è la totale eliminazione degli algoritmi di interpretazione perché tutti i caratteri UCS sono direttamente rappresentati.
Non viene mai usata nelle comunicazioni Internet.
Alcuni casi di impiego sono nelle strutture interne di sistema operativo e nella memorizzazione dei dati.

Caratteri e stringhe in Java

In ambiente Java esistono il tipo primitivo char la classi Character e String e StringBuffer.

Il tipo primitivo char

Il tipo primitivo char consente di rappresentare singoli caratteri nella codifica UTF-16.
E' quindi realizzato con un'area di memoria di due byte (16 bit) in grado di contenere un singolo codice UTF-16.

Dichiarazione

char <nomevariabile>;

Esempio di assegnazione:

```
char c;          //dichiarazione di un carattere
```

Assegnazione

Si può assegnare ad una variabile di tipo char un qualsiasi valore numerico a 16 bit da 0 a 65536. Questo valore assume il significato di un codice UTF-16.

Caratteri stampabili

per i caratteri "stampabili" si può usare la notazione in singola quota contenente il corrispondente carattere tipografico:

```
c='A';        //assegna il carattere tipografico A
```

in alternativa si può usare una qualsiasi rappresentazione numerica del simbolo come numero intero in varie basi:

```
c=65;         //assegna il numero decimale 65 ('A')
c=0x41;       //assegna il numero esadecimale 41 ('A')
c=0101;       //assegna il numero ottale 101 ('A')
c=0b1000001   //assegna il numero binario 1000001 ('A')
```

Le regole di riconoscimento delle basi sono:

- decimale nessun modificatore (senza zeri non significativi)
- esadecimale 0x
- ottale 0
- binario 0b

Il numero di cifre significative può essere qualsiasi purché non si superi la precisione consentita dai 16 bit.

Caratteri di controllo

Per i caratteri di controllo non esiste una rappresentazione tipografica. La rappresentazione in singola quota si può realizzare

usando il carattere di escape "backslash" (\) che viene anche definito "carattere di escape".
Il carattere di escape introduce una sequenza di due caratteri che sono interpretati dalla JVM come un unico carattere di controllo:
Esempi:

```
c='\n'; //carattere caporiga
        //in ambiente linux produce LF
        //in ambiente windows produce CR+LF
c='\r'  //carattere CR (ritorno carrello)
c='\f'  //carattere FF (cambio pagina)
c='\b'  //carattere BS (spazio indietro)
c='\t'  //carattere TAB (sposta di una tabulazione)
c='\\'  //carattere backslah
c='\''  //carattere singola quota
c='\"'  //carattere doppia quota
```

Caratteri non disponibili sulla tastiera

Il carattere di escape può essere usato per introdurre sequenze numeriche decimali o esadecimali per rappresentare caratteri qualsiasi anche se non disponibili sulla tastiera.

```
c='\x20AC' //carattere € espresso in esadecimale
c='\8364'  //carattere € espresso in decimale
c='\020254'//carattere € espresso in ottale
c='\u20AC' //carattere € espresso come sequenza UTF-16
```

Caratteri dei piani supplementari

Una variabile char può contenere solo un singolo codice a 16 bit quindi per rappresentare un carattere UTF-16 di un piano supplementare si devono utilizzare due caratteri.

```
ch='\uD869' //parte più significativa
cl='\uDED6' //parte meno significativa
```

Cosa si può fare e cosa non si fare con i char

I caratteri sono a tutti gli effetti dei numeri interi quindi è possibile effettuare con i caratteri qualsiasi operazione algebrica e logica. Il risultato sarà ancora un intero che se assegnato ad un char assume il significato di un nuovo carattere:

```
char c1='A'+'B';
char c2='B';
char c3=(char)c1+c2;
```

Le operazioni sono tutte lecite ma non producono una concatenazione di stringhe ma un vero e proprio calcolo numerico generando un nuovo carattere. L'operazione di casting è necessaria perché la somma viene fatta a 32 bit.

Essendo un intero può essere usato come argomento di uno switch:

```
Scanner lettore=new Scanner(System.in);
char c=lettore.nextChar();
switch(c) {
  case 'A':
  case 'a':
       ...
       break;
  case 'B':
  case 'b':
       ...
       break;
  default:
       ...
}
```

Ogni variabile di tipo carattere può contenere un solo carattere quindi non è lecito inserire più caratteri nella stessa variabile:

```
char c1='AB'; //le signole quote accettano un solo valore
char c1="AB"; //le doppie quote definiscono un oggetto String
char c1="A";  //anche se contiene un solo elemento
```

Array di caratteri

E' possibile raccogliere più caratteri in un unico array:

```
char data[]={'a','b','c'};
```

In Java, diversamente da altri linguaggi come C/C++ un array di caratteri NON è una stringa di caratteri.

La classe Character

La classe Character rappresenta la controparte "Boxed" del tipo primitivo char.

La classe incapsula un carattere in rappresentazione UTF-16 ed offre molti metodi di conversione e di estrazione di informazioni sul contenuto del carattere sia statici che di istanza.

La classe String

La classe String rappresenta una sequenza di caratteri di lunghezza arbitraria.

Gli oggetti di classe String in Java sono delle costanti (oggetti immutabili). Questo significa che quando un oggetto di tipo String viene costruito con con una stringa letterare:

```
String s="abc";
```

oppure con uno dei molti costruttori di stringa:

```
String s=new String(tipo parametro);
```
l'oggetto creato è immutabile e non può più essere modificato ma solo distrutto quando non ha più reference.
Quindi l'operazione:
```
String s=s+"altra stringa";
```
in realtà fa la concatenazione del contenuto attuale di s con la nuova stringa, generando una nuova stringa il cui reference viene assegnato ad s; la vecchia stringa riferita da s, ormai senza reference verrà poi cancellata dal garbage collector.
Questo modo di lavorare è comodo per le stringhe che subiscono poche modifiche al tempo di esecuzione mentre è poco efficiente se la stringa subisce frequenti trasformazioni e in questo caso conviene usare oggetti della classe StringBuffer che sono mutabili ma un poì più complessi da gestire.

Accesso ai caratteri di una String
Incapsulati all'interno di una stringa si trova una quantità indeterminata di caratteri a partire da 0 caratteri (stringa vuota) fino ad un numero arbitrario.
I singoli caratteri non sono direttamente accessibili (una String non è un array) ma sono disponibili vari metodi per accedere indirettamente ai caratteri di una stringa
Metodo int length()
Restituisce il numero di caratteri presenti nella stringa
Metodo char charAt(int index)
Restituisce il carattere che si trova alla posizione index.
Coerentemente con il fatto che l'oggetto è immutabile non esiste un metodo che consenta la modifica dei singoli caratteri.

Ordinamento lessicografico
Non è possibile applicare alle stringhe gli operatori relazionali dei tipi primitivi (<,>,>=,<=,!=).
E' definito però per le stringhe l'ordinamento lessicografico in base ai caratteri che contengono è quindi possibile confrontare due stringhe per determinare in quale ordine lessicografico si trovano.

Metodo int compareTo(String anotherString)
Si tratta di un metodo di istanza che applicato ad un oggetto String consente di confrontarlo con un altro oggetto String passato come parametro e di determinare se l'oggetto a cui si applica il metodo precede, segue o coincide con l'oggetto passato come parametro.
Se l'oggetto precede il parametro il metodo restituisce un valore negativo, se i due oggetti coincidono restituisce 0, se l'oggetto segue il parametro restituisce un valore positivo.
Esempio:

```
String s1="AA";
String s2="AB";
int ris=s1.compareTo(s2);
if(ris<0) {
   System.out.println("s1 precede s2");
}
else if(ris>0) {
   System.out.println("s2 precede s1");
}
else {
   System.out.println("s1 uguale s2");
}
```

Nell'esempio "AA" precede "AB" perchè i caratteri alfabetici sono ordinati in ordine progressivo:
Altri esempi di ordinamento:
"AB" precede "ABA"
"AAA" precede "AB"
"Aa" precede "aa"
"AB" precede "aa"
Metodo int compareToIgnoreCase(String anotherString)
Non distingue tra maiuscole e minuscole quindi con questo metodo:
"Aa" coincide con "AA"
"aa" precede "AB"

Metodi di analisi della stringa
La classe String è dotata di molti metodi di analisi della stringa che consento di ottenere informazioni sul suo contenuto.
I metodi usati più di frequente sono:

- **boolean contains(CharSequence s)** //determina se una sequenza di caratteri è contenuta nella stringa

- **boolean endsWith(String s)** //determina se la stringa termina con la sottostringa s
- **boolean startsWith(String s)** //determina se la stringa inizia con la sottostringa s
- **int indexOf(char c)** //restituisce la posizione della prima occorrenza del carattere c nella stringa
- **int indexOf(char c, offset i)** //restituisce la posizione della prima occorrenza del carattere c nella stringa a partire dalla posizione i
- **int lastIndexOf(char c)** //restituisce la posizione dell'ultima occorrenza del carattere c nella stringa
- **String substring(int start)** //restituisce una sotttostringa a partire da start compreso fino alla fine della strigna
- **String substring(int start, int end)** //restituisce una sottostringa a partire da start compreso ad end escluso
- **String[] split(String s)** //restituisce un array di sottostringhe usando s come delimitatore

Metodi di conversione
La classe String dispone di molti metodi statici di conversione che consentono di trasformare variabili di molti tipi in stringhe

Metodo static String valueOf(<tipo> parametro)
Restituisce una rappresentazione testuale del parametro passato come argomento.
Il <tipo> può essere un qualsiasi tipo primitivo (int, long, float, double, boolean, char, char[]).

Metodi delle classi boxed
La precedente operazione equivale a quella effettuata dai metodi di conversione delle rispettive classi "boxed" sia nella versione statica che nella versione di istanza.
Per la classe Integer:

```
String toString();
static String toString(int i);
```

Per la classe Double:

```
String toString();
```

```
static String toString(double d);
```
…

Conversione inversa.

Le classi boxed offrono anche metodi per le conversioni inverse da Stringa al tipo rappresentato dalla classe boxed :

Per la classe Integer:

```
static int parseInt(String s);
static Integer valueOf(String s);
```

Per la classe Double:

```
static double parseDouble(String s);
static Double valueOf(String s);
```

…

Formattazione di una stringa

La classe String dispone del metodo statico:

String Format(String formatter, Object … args)

Questo metodo consente di definire nella stringa formatter le regole di formattazione da applicare alla conversione degli oggetti che seguono.

Per le regole di formattazione consultare la classe Formatter.

Ad esempio se si dispone di una variabile double dotata di una quantità indeterminata di cifre dopo la virgola e si vuole limitare la presentazione a due cifre si può generare la stringa nel seguente modo:

```
double v=1.0/3.0;     //genera una approssimazione del numero
                      //periodico con 16 cifre decimali
System.out.println(v) //stampa 0.3333333333333333
String s=String.format("%10.2f",v2); //genera da v
                     //una stringa in forma decimale che
                      //contiene fino a 10 interi e fino 2
                      //decimali
System.out.println(s) //stampa 0,33
```

N.B

- i tre puntini nel secondo parametro indicano che il numero dei parametri è variabile da 0 ad un valore indeterminato ma deve corrispondere in quantità e posizione ai % presenti nella stringa di formattazione; questo impedisce al compilatore un effettivo controllo sintattico della correttezza della chiamata
- ogni % introduce una regola di formattazione a cui corrisponde un valore da formattare nella lista successiva

- il formattatore in questo caso è f che vuole dire che la rappresentazione deve essere decimale; diversamente dalla stampa immediata di un reale che produce il simbolo di punto frazione il formattatore è localizzato quindi con la localizzazione IT produce la virgola frazione
- Il formattatore può essere preceduto dall'indicazione della precisione richiesta per parte intera e parte frazionaria.

La classe StringBuffer

A differenza della classe String che genera oggetti immutabili la classe StringBuffer genera oggetti modificabili.
Quando si modifica la sequenza di caratteri di un oggetto StringBuffer l'oggetto, diversamente da un oggetto String, non viene distrutto e ricostruito, viene invece modificata internamente la sequenza.
La classe StringBuffer non dispone del costruttore implicito con una stringa di costanti e può essere costruita solo con i seguenti costruttori:

- **StringBuffer()** //costruisce una stringa vuota
- **StringBuffer(int n)** //costruisce una stringa con n caratteri non inizializzati
- **StringBuffer(String s)** //costruisce una stringa a partire da una stringa immutabile

Oltre ai metodi getter posseduti anche dalla classe String la classe StringBuffer dispone di metodi setter che consentono di modificare la struttura della sequenza di caratteri contenuta nella stringa.

- **StringBuffer append(<tipo> param)** //aggiunge caratteri in fondo alla stringa
- **StringBuffer insert(int offset,<tipo> param)** //inserisce caratteri a partire dalla posizione offset
- **StringBuffer delete(int start,int end)** //cancella caratteri a partire da start compreso fino a end escluso
- **StringBuffer replace(int start,int end, String s)** //sostituisce i caratteri a partire da start compreso fino ad end escluso con i caratteri della stringa. La lunghezza della

stringa risultante viene adattata per accogliere la sottostringa s.

I metodi append() ed insert() sono sovraccaricati (overloaded) in modo da poter ricevere tutti i tipi primitivi consentendo quindi una conversione implicita da un qualsiasi tipo primitivo a stringa.

Confronto tra String e StringBuffer

La classe String consente di creare delle stringhe immutabili a partire da un qualsiasi tipo di origine e rende disponibili molti metodi per la estrazione di informazioni dalla stringa.

E' anche possibile generare a partire da oggetti String nuove stringhe componendole a partire da stringhe di origine ma questo implica ogni volta la creazione implicita di una nuova stringa.

Al contrario la classe StringBuffer è meno dotata di metodi di estrazione delle informazioni ma consente la modifica del contenuto della stringa senza ricrearla completamente ogni volta.

L'uso della classe String è quindi vantaggioso quando dopo aver costruito una stringa non la si deve ulteriormente modifica ma invece la si deve analizzare mentre l'uso della classe StringBuffer è vantaggio quando non si deve analizzare il contenuto della stringa ma la si deve modificare molto di frequente

Esempio:

Supponiamo di concatenare due stringhe:

Usando la classe String si genera:

```
String s="Salve ";
s=s+"mondo!";
```

Usando la classe StringBuffer invece si genera:

```
StringBuffer s=new StringBuffer("Salve ");
s.append("mondo!);
```

Dal punto di vista sintattico la prima versione è più semplice e compatta ma dal punto di vista esecutivo la seconda è più efficiente perchè mentre nel primo caso quando ad s viene assegnato il risultato viene creata implicitamente una nuova stringa e la vecchia, perdendo il reference viene distrutta mentre nel secondo caso si continua ad usare lo stesso oggetto.