Lección 6
Codificación de caracteres

Desde el ASCII al UTF-8

Introducción

- ¿Qué es un carácter?
  - ¿Son caracteres distintos ‘A’ y ‘a’?
  - ¿Son caracteres distintos ‘A’, ‘*A*’, ‘**A**’, ‘*A*’?
- Debe decidirse qué serán caracteres y qué no. Esto nos dará el *repertorio*.
- A cada carácter del repertorio se le asignará un *código numérico*.
- Los computadores manejan el código numérico (en ficheros y transmisiones de datos por red).
- El hardware o software ocupado de mostrar esa información convierte el código numérico en un *glifo,* que depende de la fuente elegida o instalada.

ASCII [1963]

- Cuando dos computadores intercambian información, deben asegurarse de estar usando el mismo código numérico para cada carácter.
- El primer estándar para cubrir este aspecto fue el ASCII.

	·0	·1	·2	·3	·4	·5	·6	·7	·8	·9	·A	·B	·C	·D	·E	·F
0·	U+0000	U+0001	U+0002	U+0003	U+0004	U+0005	U+0006	U+0007	U+0008	U+0009	U+000A	U+000B	U+000C	U+000D	U+000E	U+000F
1·	U+0010	U+0011	U+0012	U+0013	U+0014	U+0015	U+0016	U+0017	U+0018	U+0019	U+001A	U+001B	U+001C	U+001D	U+001E	U+001F
2·	U+0020	! U+0021	" U+0022	# U+0023	$ U+0024	% U+0025	& U+0026	' U+0027	(U+0028	) U+0029	* U+002A	+ U+002B	, U+002C	- U+002D	. U+002E	/ U+002F
3·	0 U+0030	1 U+0031	2 U+0032	3 U+0033	4 U+0034	5 U+0035	6 U+0036	7 U+0037	8 U+0038	9 U+0039	: U+003A	; U+003B	< U+003C	= U+003D	> U+003E	? U+003F
4·	@ U+0040	A U+0041	B U+0042	C U+0043	D U+0044	E U+0045	F U+0046	G U+0047	H U+0048	I U+0049	J U+004A	K U+004B	L U+004C	M U+004D	N U+004E	O U+004F
5·	P U+0050	Q U+0051	R U+0052	S U+0053	T U+0054	U U+0055	V U+0056	W U+0057	X U+0058	Y U+0059	Z U+005A	[U+005B	\ U+005C	] U+005D	^ U+005E	_ U+005F
6·	` U+0060	a U+0061	b U+0062	c U+0063	d U+0064	e U+0065	f U+0066	g U+0067	h U+0068	i U+0069	j U+006A	k U+006B	l U+006C	m U+006D	n U+006E	o U+006F
7·	p U+0070	q U+0071	r U+0072	s U+0073	t U+0074	u U+0075	v U+0076	w U+0077	x U+0078	y U+0079	z U+007A	{ U+007B	| U+007C	} U+007D	~ U+007E	U+007F

Limitaciones del ASCII

- Al usar sólo 7 bits, sólo permite 128 caracteres.
- 32 de ellos están reservados para códigos de control (no caracteres)
- Tan solo contiene letras del alfabeto inglés (***A****merican Standard Code*...)
- No obstante presenta algunas propiedades interesantes:
  - Orden alfabético
  - Mayúsculas/minúsculas se diferencian en 1 bit
  - Código de los dígitos.

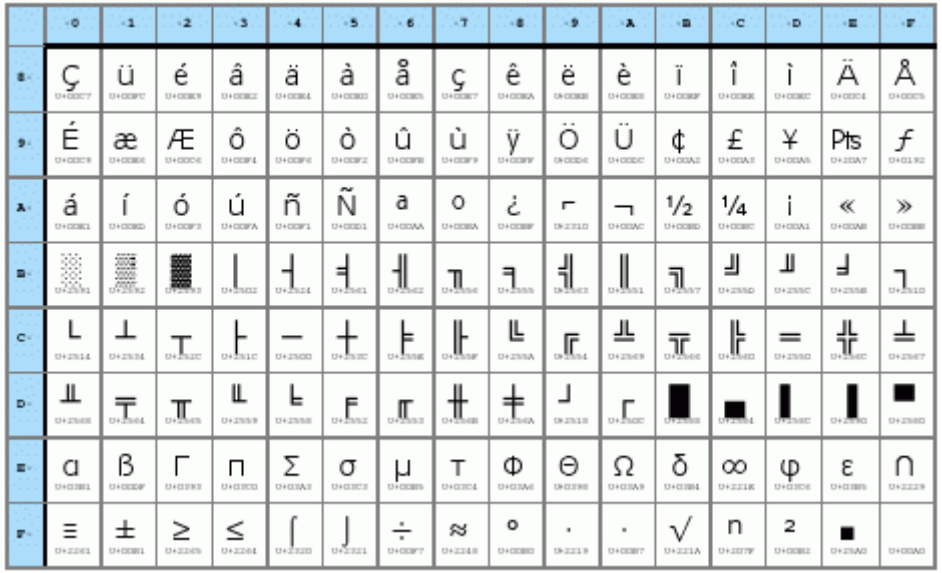
Usando 8 bits

- Usando el byte completo se tendrían 256 posibles caracteres.
- Para no romper la compatibilidad con ASCII, se hace que el primer bit signifique:
  - 0: Los 7 bits inferiores siguen la tabla ASCII
  - 1: Los 7 bits inferiores siguen otra tabla
- Problemas:
  - Estandarizar el uso de la "otra tabla".
  - ¿Qué pasa con los lenguajes que requieran más de 128 códigos? (Ej: chino)
  - ¿Qué pasa si queremos usar varios lenguajes?

"Otras tablas" famosas

- IBM usaba el término "codepages" para referirse a las tablas.
- Tenían varias diferentes, según el país donde se fuera a distribuir el computador
  - La *codepage 437* fue la original del IBM-PC
  - Entonces conocida como "ASCII extendido" [1981]
  - Posteriormente sustituida por la *codepage 850*, para incluir letras mayúsculas con acentos.
  - Todavía usadas en el *interfaz de comandos* de Windows.

CP437 y CP850



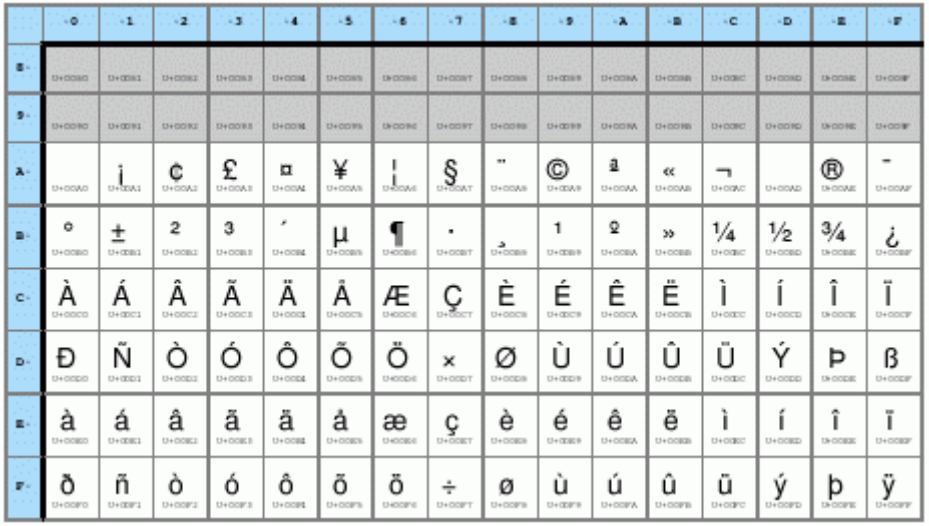
Codepage 437 “MS-DOS *default*”

Codepage 850 “MS-DOS latin”

Estándares ISO 8859

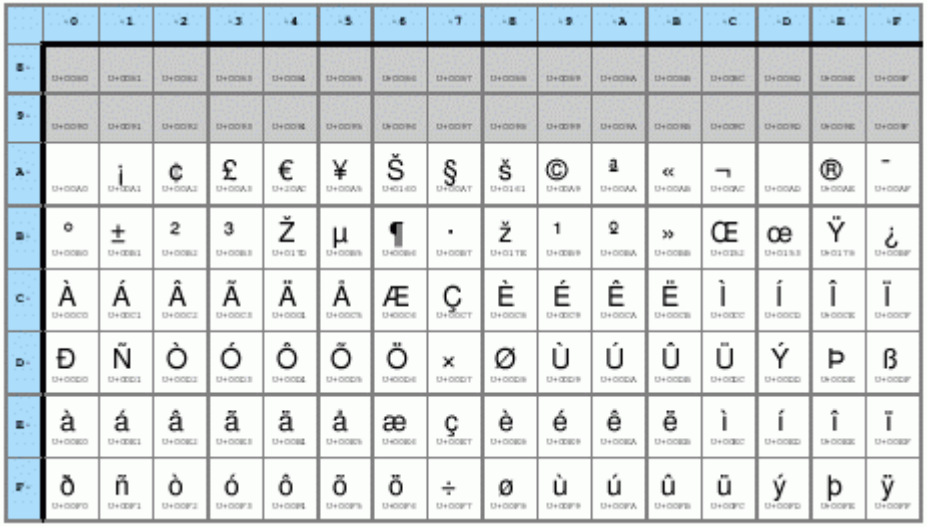
- ISO crea una serie de 15 estándares.
- Denominados ISO-8859-1 hasta ISO-8859-16
- Cada uno cubre necesidades de un alfabeto diferente (latino, europa del este, hebreo, cirílico, etc.)
- Los más usados en occidente:
  - ISO-8859-1 (“Latin1”) [1998]
  - ISO-8859-15 (“Latin9”) Variante del anterior para incluir el euro [1999]

Latin1 y Latin9



ISO-8859-1 (obsoleto)

ISO-8859-15



Codificación en Windows

- Microsoft Windows usa sus propias variantes de los estándares ISO.
- En particular, para países latinos, se trata de la *codepage 1252*
- Es "compatible" con ISO-8859-1 excepto en los primeros 32 códigos:
  - Estos códigos son de control en ISO-8859-1
  - Pero Microsoft los ha usado para incluir caracteres que faltaban en ISO-8859-1 (el euro entre ellos)
  - Esto causa incompatibilidad con ISO-8859-15

Windows-1252

	·0	·1	·2	·3	·4	·5	·6	·7	·8	·9	·A	·B	·C	·D	·E	·F
8·	€ U+20AC	...	‚ U+201A	ƒ U+0192	„ U+201E	… U+2026	† U+2020	‡ U+2021	ˆ U+02C6	‰ U+2030	Š U+0160	‹ U+2039	Œ U+0152	...	Ž U+017D	...
9·	...	‘ U+2018	’ U+2019	“ U+201C	” U+201D	• U+2022	– U+2013	— U+2014	˜ U+02DC	™ U+2122	š U+0161	› U+203A	œ U+0153	...	ž U+017E	Ÿ U+0178
A·	U+00A0	¡ U+00A1	¢ U+00A2	£ U+00A3	¤ U+00A4	¥ U+00A5	¦ U+00A6	§ U+00A7	¨ U+00A8	© U+00A9	ª U+00AA	« U+00AB	¬ U+00AC	U+00AD	® U+00AE	¯ U+00AF
B·	° U+00B0	± U+00B1	² U+00B2	³ U+00B3	´ U+00B4	µ U+00B5	¶ U+00B6	· U+00B7	¸ U+00B8	¹ U+00B9	º U+00BA	» U+00BB	¼ U+00BC	½ U+00BD	¾ U+00BE	¿ U+00BF
C·	À U+00C0	Á U+00C1	Â U+00C2	Ã U+00C3	Ä U+00C4	Å U+00C5	Æ U+00C6	Ç U+00C7	È U+00C8	É U+00C9	Ê U+00CA	Ë U+00CB	Ì U+00CC	Í U+00CD	Î U+00CE	Ï U+00CF
D·	Ð U+00D0	Ñ U+00D1	Ò U+00D2	Ó U+00D3	Ô U+00D4	Õ U+00D5	Ö U+00D6	× U+00D7	Ø U+00D8	Ù U+00D9	Ú U+00DA	Û U+00DB	Ü U+00DC	Ý U+00DD	Þ U+00DE	ß U+00DF
E·	à U+00E0	á U+00E1	â U+00E2	ã U+00E3	ä U+00E4	å U+00E5	æ U+00E6	ç U+00E7	è U+00E8	é U+00E9	ê U+00EA	ë U+00EB	ì U+00EC	í U+00ED	î U+00EE	ï U+00EF
F·	ð U+00F0	ñ U+00F1	ò U+00F2	ó U+00F3	ô U+00F4	õ U+00F5	ö U+00F6	÷ U+00F7	ø U+00F8	ù U+00F9	ú U+00FA	û U+00FB	ü U+00FC	ý U+00FD	þ U+00FE	ÿ U+00FF

Unicode [1991-2005]

- Objetivos:
  - Crear un repertorio de todos los caracteres en todos los lenguajes del mundo (y más)
  - Asignar un código numérico único a cada carácter (abandonando la idea de usar 8 bits)
  - Mantener en lo posible la compatibilidad con ASCII e ISO
- Organización de los códigos:
  - Se organizan en "planos"
  - Cada "plano" tiene espacio para 2^{16} caracteres
  - Se limita el número de planos a 17.
- Por tanto:
  - Hay espacio para $17*2^{16}$ caracteres=$2^{20}+2^{16}$
  - Necesarios 21 bits como máximo

Unicode: uso de los planos

- De los 17 planos previstos:
  - El plano 0 está completamente definido.
    - Se denomina “Basic Multilingual Plane” (BMP)
    - Contiene todos los alfabetos “vivos”
  - El plano 1 está en proceso de ser definido.
    - Contiene alfabetos “muertos” o exóticos.
    - Más simbolos matemáticos y musicales
  - Los restantes planos aún no han sido definidos
- Notación:
  - Un código Unicode suele escribirse como “U-” más 8 cifras hexadecimales (realmente, bastaría con 6)
  - Las cuatro primeras cifras dan el número de plano. Las cuatro inferiores la localización dentro del plano
  - Si el plano es 0, suele omitirse y escribirse “U+” más 4 cifras hexadecimales.
  - Ej: U-00013EF2, U-000032B0 (= U+32B0)

Unicode: compatibilidad ASCII

- La “compatibilidad” se logra en la forma siguiente:
  - Los códigos desde U+0000 hasta U+007F siguen la tabla ASCII
  - Los códigos desde U+0080 hasta U+00FF siguen la tabla ISO-8859-1
- Por tanto la conversión entre “Latin1” (que incluye ASCII) y Unicode es trivial

Ejemplos del repertorio Unicode

Código Unicode	Carácter	Nombre
U+0041	A	Letra A Latina Mayúscula
U+00D1	Ñ	Letra Ñ Latina Mayúscula
U+03B4	δ	Letra Delta Griega Minúscula
U+20AC	€	Símbolo del Euro
U+0259	ə	Letra "Schwa" del IPA
U+13A3	Ꭳ	Letra O Cherokee
U+263A	☺	Smiley
U+6F22	漢	Carácter chino "Han"
U+3042	あ	Letra A del alfabeto hiragana (japonés)
U+FEB6	ﺷ	Letra Sheen en forma final (árabe)
U+2815	⠕	Letra O en Código Braille
U-00010330	𐌰	Letra A en Gótico
U-0001D4DB	𝓛	Letra L caligráfica matemática (transformada de Laplace)
U-0001D571	𝕱	Letra F gótica matemática (transformada de Fourier)
U-0001D160	♪	Nota Musical Corchea

Codificación de los códigos

- Unicode sólo especifica qué valor numérico tiene cada carácter, pero no cómo almacenarlo (codificarlo) en un computador.
- A priori, parece que se requerirían 21 bits para cada carácter, pero:
  - No es potencia de 2. => Usemos 32 bits
  - Demasiado desperdicio de memoria, el 99,999% de las ocasiones sólo usaremos caracteres del plano 0, (caben en 16 bits) => Usemos 16 bits
  - Pero entonces no se pueden representar los demás planos... ¿?
  - Muchas aplicaciones sólo requieren ASCII. Incluso 16 bits sería desperdicio para ellas.
  - ¿?

Codificaciones triviales: UCS

- UCS-4 [2000]
  - La más directa: usar 32 bits por código
  - También llamada UTF-32
  - No se usa apenas.
- UCS-2 [2000]
  - Usar 16 bits por código
  - Recortar el repertorio al plano 0 (BMP)
  - Los caracteres en los demás planos no son representables.
  - Utilizada por la máquina virtual Java
  - [No confundir con UTF-16]

Codificaciones más elaboradas: UTF

- UTF: ***U****nicode* ***T****ransformation* ***F****ormat*
  - Un carácter puede ser representado por un solo dato, o por una secuencia de datos.
- UTF-16[2000]: el dato "mínimo" es de 16 bits
  - Los caracteres del plano 0 requieren un solo dato
  - Los de los restantes planos requieren dos datos (32 bits)
- UTF-8[2003]: el dato "mínimo" es de 8 bits
  - Los caracteres ASCII requieren un solo dato
  - Los ISO-8859-1 requieren dos datos
  - A medida que crece el valor del código Unicode, la representación requiere más datos
  - El mayor carácter Unicode posible requerirá cuatro datos. (32 bits)

UTF-16: Algoritmo

- Si el carácter está en el rango U+0000 hasta U+FFFF, se representa con **un solo dato**
- Si el carácter está por encima de U+FFFF, se aplica la siguiente transformación:
  - Restar 00010000h al carácter. Se obtiene un número que cabe en 20 bits
  - Si llamamos xxxxxxxxxxyyyyyyyyyy a estos 20 bits, la codificación consistirá en los **DOS** siguientes datos, en secuencia:
    - 110110xxxxxxxxxx
    - 110111yyyyyyyyyy
- Ejemplo:
  - Carácter: nota corchea. Código Unicode: U-0001D160
  - ¿Codificación UTF-16? Respuesta: D834, DD60

UTF-16: Observaciones

- Cuando un carácter requiere dos datos, estos dos datos estarán ambos en el rango D800-DFFF
- Unicode no define ningún carácter en ese rango. Esto es U+D800 hasta U+DFFF no son caracteres válidos (precisamente para hacer posible UTF-16)
- Para UTF-16, la secuencia D834, DD60 representa una corchea, pero para para UCS-2 representa sólo dos caracteres inválidos.
- Cualquier caracter del plano 0 (el más común) requiere sólo 16 bits.
- La detección de errores es sencilla:
  - Si aparece un dato en el rango D800-DBFF, el siguiente debe estar en el rango DC00-DFFF, si no es un error
  - Si aparece un dato en el rango DC00-DFFF pero el anterior no estaba en D800-DBFF, es un error

UTF-16 y la *endianity*

- UTF-16 maneja datos de 16 bits. Pero a nivel de socket y de ficheros se transfieren bytes.
  - Cada dato UTF-16 debe ser almacenado en 2 bytes => Problema de la *endianity*.
- Soluciones (tres variantes):
  - UTF-16LE: Orden *Little Endian*
  - UTF-16BE: Orden *Big Endian*
  - UTF-16: Orden no especificado. El primer carácter de la secuencia debe ser U+FEFF
    - Este carácter no es imprimible (espacio de ancho cero)
    - Al recibirlo, se verá si llega FE, FF ó FF, FE, y se detecta la *endianity* del emisor.
    - El carácter U+FFFE no existe, para evitar confusión.

UTF-16: Ejemplo

- Usamos una arquitectura Little Endian (ej: un PC)
- Y un editor que almacena en UTF-16 (ej: Wordpad)
- Escribimos el texto: $\mathfrak{F}$ourier
- Almacenamos en un fichero. ¿Qué bytes quedan almacenados?
- Respuesta:

```
FF FE 35 D8 71 DD 6F 00 75 00
72 00 69 00 65 00 72 00
```

UTF-8: Algoritmo

- Rango U+0000 hasta U+007F
  - Se requieren 7 bits
  - Se usa un byte:
    - 0xxxxxxx
- Rango U+0080 hasta U+07FF
  - Se requieren 11 bits
  - Se usan dos bytes:
    - 110xxxxx 10xxxxxx
- Rango U+0800 hasta U+FFFF
  - Se requieren 16 bits
  - Se usan 3 bytes:
    - 1110xxxx 10xxxxxx 10xxxxxx
- Rango U-00010000 hasta máximo
  - Se requieren 21 bits
  - Se usan 4 bytes
    - 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

UTF-8: Decodificación y detección de errores

- Se mira el primer bit del dato
  - Si es 0, es un ASCII. Los 7 inferiores nos dan el código del carácter
  - Si es 1:
    - Si los dos primeros son 10... es un error.
    - El número de unos seguidos al principio del dato nos dice cuántos bytes ocupa la codificación. Ej: 110... indica 2 bytes, 1110... indica 3 bytes
    - Los restantes bytes de la codificación deben comenzar por 10..., si no es un error.
- Ejemplo:
  - Detectar si hay error en la secuencia:

    `41 6F E7 84 A6 70 C6 71 20 84 61`

UTF-8: Propiedades interesantes

- Muy eficiente en términos de espacio si el texto es mayormente ASCII.
- Los caracteres “Latin1”, en cambio, requieren dos bytes. En todo caso, lo mismo que con UTF-16 ó UCS-2
- Los caracteres por encima del plano 0 requieren 4 bytes, pero es lo mismo que en UTF-16 ó UCS-4
- Sólo en algunos caracteres del plano 0 (U+0800 a U+FFFF) es ineficiente pues requiere 3 bytes (alfabetos asiáticos).
- Nunca puede aparecer un byte de valor 00 en una secuencia UTF-8
  - Por tanto es compatible con los *strings* del C
- Muy utilizado, y el estándar para XML.

UTF-8: Ejemplo

- ¿Cuál es la codificación UTF-8 del carácter hebreo Aleph, si su código Unicode es el U+FB21?
  - Caso entre U+0800 y U+FFFF, requiere 16 bits: 1111101100100001
  - La codificación requiere 3 bytes. 1110xxxx, 10xxxxxx, 10xxxxxx
  - Los 16 bits se han de dividir en grupos de 4, 6 y 6, y resultan: 1111 101100 100001
  - Al primer grupo se le pone delante 1110, y a los restantes se les pone 10, lo que resulta en los tres bytes: 11101111, 10101100, 10100001
- Resultado: