

To the BMP and beyond!

Eric Muller
Adobe Systems

Welcome!

The goal of this class is to introduce you to the Unicode standard, which deals with the representation and manipulation of characters in computers.

Unicode is a fairly complex standard. This is mostly due to the complexity of the writing systems of the world. Don't be too afraid, but be prepared for some work!

We have placed a particular emphasis on using the correct terminology. Using the appropriate words from the start will make your journey with Unicode much more enjoyable.

These notes have two purposes: first, to clarify the necessarily simplified text of the slides; second, to provide additional details that cannot be covered during the class itself.

Content

1. Why Unicode
2. Character model
3. Principles of the Abstract Character Set
4. The characters in 5.0
5. Development of the standard
6. Processing
7. Unicode and other standards
8. Resources

Here is the plan of the class.

Section 1 explains the goals of Unicode in general terms.

Section 2 presents the character model that underlies Unicode.

Section 3 describes the principles that guide the choice of characters in the Unicode standard.

Section 4 gives an overview of the characters and scripts covered by version 4.0 of the standard,

Section 5 describes the process by which the standard evolves.

Section 6 introduces the major algorithms defined by Unicode.

Section 7 explains how other character standards can be implemented using Unicode.

Finally, section 8 points to additional resources.

Part I
Why Unicode

ASCII

- 128 characters
 A = 41
- supports meaningful exchange of text data
- very limited: not even adequate for English:
 Adobe®
 he said "Hi!"
 résumé†
 cañon

Let's start with the familiar case of the ASCII character standard. The designers of ASCII selected 128 characters, and for each one defined its representation in a computer, in the form a byte. For example, the character **A** is represented by the byte 0x41.

Having such a standard is a critical step to support the meaningful exchange of text between applications. Consider something as simple as copy/paste: the source and destination must have some agreement about the meaning of what is exchanged.

ASCII is very simple, but too much so. Even for English, more than 128 characters are needed: besides the letters and digits, one needs a fair complement of typographic symbols (quotes, dashes, bullets, etc), accented characters for loan words, etc. And of course, other scripts and languages need even larger collections.

Many other standards

- national or regional standards
 - ISO-Latin-1/9: targets Western Europe
 - JIS: targets Japan
- platform standards
 - Microsoft code pages
 - Apple: MacRoman, etc.
 - Adobe: PDFDocEncoding, etc.
- but none for many writing systems!

The limitations of ASCII led to many other character standards. Some are national or regional standards (such as the ISO-Latin family, or the JIS family). Some are defined by software vendors, the most notable examples being the Microsoft code pages and the Apple encodings.

While each standard solves some problems, their multiplicity makes the life of the software developer and of the users more complicated. In particular, meaningful exchange of text data now requires a fairly complex infrastructure to deal with those multiple standards. One needs to interchange not only the characters themselves, but an indication of the standard used to represent them (e.g. the charset attribute of the Content-Type header in HTTP).

Furthermore, many writing systems are not covered by those standards, thus limiting computer usage in some regions of the world.

Unicode

- enables world-wide *interchange* of data
- contains all the *major living* scripts
- *simple enough* to be implemented everywhere
- supports *legacy data* and implementation
- allows a *single* implementation of a product
- supports *multilingual* users and organizations
- conforms to *international standards*
- can serve as *the foundation* for other standards

Unicode's goals are to improve dramatically this state of affair.

The approach is to have a single character standard that covers all the uses, thus enabling the world-wide interchange of data. To do so, Unicode must also cover all the major living scripts of the world.

To succeed, Unicode must be simple enough (but no more so!) to be implemented everywhere and must offer a viable transition from the legacy standards.

Software developers should be able to write a single implementation of their product to cover the whole world. Multi-lingual data producers should be supported.

The last two goals are related to standardization; this is important since text is used in so many contexts which are themselves standardized, such as programming languages, database, and XML.

Part II
Character model

Four layers

- abstract character set
smallest components of written language
- coded character set
adds name and code point
- character encoding forms
representation in computer
- character encoding schemes
byte serialization

Designing a character representation touches on many problems. To better tackle those, Unicode defines a model that separates them in distinct layers, much like the OSI networking model.

The first layer is that of the abstract character set and it tackles the question: what is a character?

The second layer is fairly mechanical: the goal is mostly to name and enumerate the abstract characters, resulting in a coded character set.

The third layer deals with the representation of coded characters in computers.

The last layer provides an unambiguous serialization into bytes of those representations.

Abstract character set

- character:
the smallest component of written language that has semantic value
- wide variation across scripts
alphabetic, syllabary, abugidas, abjad, logographic
- even within scripts, e.g. “ch”:
two components in English
one component in Spanish?
- abstract character:
a unit of information used for the organization, control, or representation of textual data.

The word “character” is used in many contexts, with different meanings. Human cultures have radically differing writing systems, leading to radically differing concepts of a character. Such wide variation in end user experience can, and often does, result in misunderstanding. This variation is sometimes mistakenly seen as the consequence of imperfect technology. Instead, it derives from the great flexibility and creativity of the human mind and the long tradition of writing as an important part of the human cultural heritage.

A starting definition of character is “the smallest component of written language that has semantic value”. Clearly, there is no sense encoding the bowl and the stem of a “p” as separate characters: these things do not have semantic value on their own, and in this case, it is pretty clear that “p” itself has semantic value.

The scripts of the world show a lot of variation in their organization:

- *alphabets*, such as Latin, Greek and Cyrillic, represent consonant and vowel sounds by independent letters.
- *syllabaries*, such as Hiragana and Katakana, represent the combination of a consonant and a vowel sound, and those combinations are atomic.
- *abugidas*, such as the Indic scripts, represent consonants with an inherent vowel sound, which can be combined with an independent vowel to replace the inherent vowel.
- in *abjads*, such as Arabic, only consonants and long vowels are usually written. In the case of Arabic, short vowels are written only in teaching materials and in the Quran.
- *logographic* scripts, such a Chinese, are another ball game all together.

Even within a script, the notion of character is not as clear as it may seem. For example, traditional Spanish mostly treats “ch” as single character.

We sidestep a little bit this question by defining a character as a unit of information. We will see later that Unicode follows general principles in selecting the abstract characters.

Coded character set

- give a name and a code point to each abstract character
- name: LATIN CAPITAL LETTER A
- code point: pure number, no computer connection
 - legal values: U+0000 - U+10FFFF
 - space for over a million characters
- characters specific to a script mostly grouped

The next step is to give a unique name (such as LATIN CAPITAL LETTER A) and code point to each abstract character that is encoded.

The name clearly intends to describe the character it names. For practical reasons, Unicode decided that names would never be changed, even if a more descriptive one could be found later.

The code point is an integer in the ranges 0 - 10FFFF. Note the convention when writing a code point: it is prefixed by “U+”, and is made of four to six hexadecimal digits (0-padded). This number provides a very compact way of identifying a character.

Characters from the same script (e.g. Greek) or same functionality (e.g. Arrows) are mostly grouped, i.e. have code points in the same range.

It is worth noting that in some cases, a single abstract character generates multiple coded characters, mostly to support round-tripping with other standards.

17 Planes

- 17 planes of 64k code points each
- plane 0: Basic Multilingual Plane (BMP, 1.0)
frequent characters
- plane 1: Supplementary Multilingual Plane (SMP, 3.1)
infrequent, non-ideographic characters
- plane 2: Supplementary Ideographic Plane (SIP, 3.1)
infrequent, ideographic characters
- plane 14: Supplementary Special-purpose Plane (SSP, 3.1)
- planes 15 and 16: Private use planes (2.0)

The full set of code points is organized in 17 planes of 64k characters each.

Plane 0 (U+0000 - U+FFFF) is called the *Basic Multilingual Plane* (BMP) and it contains the most frequent characters. It was populated starting in Unicode 1.0.

Plane 1 (U+10000 - U+1FFFF) is called the *Supplementary Multilingual plane* (SMP) and it contains infrequently used scripts, such as Deseret. It was populated starting in Unicode 3.1.

Plane 2 (U+20000 - U+2FFFF) is called the *Supplementary Ideographic Plane* (SIP) and it contains ideographic characters, most of which are infrequent. It was populated starting in Unicode 3.1.

Plane 14 (U+E0000 - U+EFFFE) is called the *Supplementary Special-purpose Plane* (SSP); don't worry about it. It was populated starting in Unicode 3.1.

Planes 15 and 16 (U+F0000 - U+10FFFF) are Private Use planes. Those planes were introduced in Unicode 2.0.

The remaining planes have no encoded characters at this time.

Private Use Area

- for your own characters; will never be assigned
- *must* agree on the meaning of those code points
- Unicode does not provide a mechanism to do so
- very delicate to use
 - avoid it if possible
- distribution:
 - U+E000 - U+F8FF: 6,400 in the BMP
 - U+F0000 - U+FFFFF: 64k in plane 15
 - U+100000 - U+10FFFF: 64k in plane 16

The Private Use Area is a bunch of code points set aside so that users can extend Unicode. Those code points will never be assigned a specific character by the Unicode standard.

To use the PUA, consenting parties *must* agree on the meaning of the PUA code points. By construction, this agreement is outside of Unicode itself; furthermore, Unicode provides no mechanism to establish such an agreement.

Using the PUA is a delicate proposition, because ensuring that all the parties understand and obey the agreement can be difficult. This is certainly true as soon as the documents that use the PUA are propagated more or less freely.

Surrogate code points and scalar values

- Unicode was originally defined as a “16 bit character set”
- in 1996 (Unicode 2.0), realized that this was not enough
- code points set aside: *surrogates code points*
 - U+D800 - U+DBFF: 1,024 high surrogates
 - U+DC00 - U+DFFF: 1,024 low surrogates
- remaining code points: *scalar values*
 - U+0000 - U+D7FF
 - U+E000 - U+10FFFF
- surrogates code points must never appear in data

Originally, Unicode was defined as a 16 bit character set. It was thought that 64k characters was enough, but most notably because of the approach eventually used to encode Han ideographs, this turned out not to be the case.

To extend the original scheme, two blocks of 1,024 (then) unused code points were set aside. Those two blocks are known as the *surrogate code points*.

The remaining code points are known as the *scalar values*.

The benefit of setting aside the surrogate code points will become apparent when we look at the encoding forms, and most specifically at UTF-16.

Character encoding forms: UTFs

- the representation of *scalar values* in computers
- each scalar value represented by a sequence of *code units*
- three forms, defined by:
 - size of the underlying code unit (8, 16, 32 bits)
 - method to convert a scalar value to code units
- all three forms can represent all scalar values and only the scalar values
- no escapes, self-synchronizing

The next layer of the character model deals with the representation of scalar values in computers. In general, each scalar value is represented by a sequence of *code units*. Unicode defines three methods to do so, collectively named UTF (Unicode Transformation Format), which differ in the size of the code units, and the conversion method.

All three UTFs have two important characteristics. The first is that each UTF can represent all the scalar values; thus, there is no reason to choose one over another based on the set of characters to represent.

The second characteristic is that UTFs are efficient to use. There are no escapes (i.e. no state), so one can simply juxtapose two strings to concatenate them. Sequences of code units are self-synchronizing: discovering the boundaries of characters is a trivial operation that does not require to scan the whole string.

UTF-8

- 8 bit code units, 1 to 4 units

USV	Unit 1	Unit 2	Unit 3	Unit 4
0000-007F	0xxxxxxx			
0080-07FF	110xxxxx	10xxxxxx		
0800-D7FF	1110xxxx	10xxxxxx	10xxxxxx	
E000-FFFF				
10000-10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

- e.g. $F03F_{16} = 1111\ 0000\ 0011\ 1111_2$
 $\rightarrow 11101111\ 10000000\ 10111111_2 = EF\ 80\ BF_{16}$
- use this table strictly
- coincides with ASCII
- mostly found in protocols and files

The first UTF is UTF-8, which uses 8 bit code units (hence its name). Each character is represented by 1 to 4 code units.

The table above shows how a given scalar value is represented in UTF-8. For example, to represent the scalar value $U+F03F$, one needs to use the fourth row. The number $0xF03F$ is written $1111\ 0000\ 0011\ 1111$ in binary. The top 4 bits are combined with “1110” to form the first code unit, i.e. $1110\ 1111$ or $0xEF$. The next 6 bits are combined with “10” to form the second code unit, i.e. $1000\ 0000$ or $0x80$. The bottom 6 bits are combined with “10” to form the third and final code unit, i.e. $1011\ 1111$ or $0xBF$. Thus the UTF-8 representation of $U+F03F$ is $0xEF\ 0x80\ 0xBF$.

It is important to use this table strictly, i.e. to use the proper row. For example, the scalar value $U+0001$ *must* be represented using the first row, i.e. by $0000\ 0001$ or $0x01$. It is illegal to represent it using the second row, i.e. by $11000000\ 10000001$ or $0xA0\ 0x81$.

Similarly, the surrogate code points (D800 through DFFF) cannot be represented in UTF-8 (or any other UTF). For example, $11101101\ 10100000\ 10000000$ does not represent D800. This is not a limitation since the surrogate code points must not occur in data.

UTF-8 coincides with ASCII; if a string contains only ASCII characters, then its representations in ASCII and in UTF-8 are identical. Thus, when extending to Unicode a protocol or document format that is based on ASCII, it is convenient to use UTF-8: all existing documents are still legal and retain the same interpretation.

Given a code unit in an UTF-8 sequence, finding the beginning of the character it represents is easy: while the top two bits are “10”, move backwards; in at most three steps, you are at the beginning of the character, without having to look at the previous characters. The top bits of the first code unit of a character indicate the number of code units for that character: “0” for one code unit, “110” for two code units, “1110” for three code units and “11110” for four code units.

UTF-16

- 16 bit code units, 1 or 2 units

USV	Unit 1	Unit 2
0000-D7FF	xxxxxxxxxxxxxxxx	
E000-FFFF		
10000-10FFFF (10000 bias)	<i>110110</i> xxxxxxxx	<i>110111</i> xxxxxxxx

- takes advantage of gap in scalar values
 - 110110xxxxxxxx = D800 - DBFF
 - 110111xxxxxxxx = DC00 - DFFF
- because of frequency of BMP, efficient
- appropriate for applications
- UCS-2 is ISO term for UTF-16 restricted to BMP

UTF-16 uses 16 bit code units. Scalar values in the BMP are represented using one code unit, and scalar values in supplemental planes are represented using two code units.

The table above shows how a given scalar value is represented in UTF-16. For all BMP scalar values, simply use the (bottom) 16 bits of the scalar value to form a single code unit. Otherwise, subtract 0x10000 from the scalar value, which is then expressed in at most 20 bits; prefix the top ten bits by “110110” to form the first code unit, and the bottom ten bits by “110111” to form the second code unit.

UTF-16 works because the surrogates have been set aside. Since surrogate code points do not appear in data, their numeric values can and are used as code units to represent supplementary code points.

Because the vast majority of the frequent characters are in the BMP, UTF-16 is fairly efficient: most frequently, a single code unit entirely represents a character, and the path to handle two code units is rarely taken.

UTF-16 is the expected representation for the Windows, Java, ICU and MacOS APIs. This consideration alone makes UTF-16 the most convenient internal representation for applications.

Given a code unit in an UTF-16 sequence, finding the beginning of the character it represents is easy: if the top bits are “110111”, move back one unit. The top bits of the first code unit of a character indicate the number of code units for that character: “110110” for two code units, otherwise one code units.

The ISO 10646 standard defines the UCS-2 encoding form. For all practical purposes, it is UTF-16 restricted to the BMP, i.e. to those scalar values in the range U+0000 - U+FFFF.

UTF-32

- 32 bit units, 1 units

USV	Unit 1
0000-D7FF	<i>0000000000</i> xxxxxxxxxxxxxxxxxxxxxxxxxxxx
E000-10FFFF	

- convenient, but expensive
- rarely used
- UCS-4 is the ISO term for UTF-32

The final UTF is fairly straightforward: each scalar value is represented by exactly one 32 bit code unit, which is the scalar value itself.

While this is an extremely convenient representation, it is also fairly expensive. The top 11 bits are always zero, and most frequently, the top 16 bits are all zeros. Therefore it is rarely used.

The ISO 10646 standard defines the UCS-4 encoding form. For all practical purposes, it is UTF-32.

Character encoding schemes

- mapping of code units to bytes
- UTF-8: obvious
- UTF-16LE
 - little endian
 - initial FF FE (if present) is a character
- UTF-16BE
 - big endian
 - initial FE FF (if present) is a character
- UTF-16
 - either endianness
 - may have a BOM: FF FE or FE FF, not part of text
 - if no BOM, then must be BE
- UTF-32: similarly, UTF-32LE, UTF-32BE and UTF-32

The final layer of the character model deals with the serialization in bytes of the code units.

For UTF-8, where the code units are already bytes, this step is trivial, and there is a single encoding scheme.

For UTF-16, there are three encoding schemes:

- in UTF-16LE, the least significant byte of each code unit comes first. If the string starts with the bytes FF FE, those two bytes should be interpreted as the FEFF code unit, i.e. as the character U+FEFF ZERO WIDTH NO-BREAK SPACE.
- in UTF-16BE, the most significant byte of each code unit comes first. If the string starts with the bytes FE FF, those two bytes should be interpreted as the FEFF code unit, i.e. as the character U+FEFF ZERO WIDTH NO-BREAK SPACE.
- in UTF-16, either endianness is possible. The endianness may be indicated by starting the byte stream with FF FE (little endian) or FE FF (big endian), and those bytes are not part of the string. If no endianness is specified, then the byte order must be big endian.

UTF-32 also has three encoding schemes, defined in a similar way.

Character Latin A

- abstract character:
the letter **A** of the Latin script
- coded character:
name: LATIN CAPITAL LETTER A
code point: U+0041
- encoding forms:
UTF-8: 41
UTF-16: 0041
UTF-32: 00000041

Let's look at a few characters in this model:

The abstract character **A** of the Latin script is represented by the coded character named LATIN CAPITAL LETTER A, and its code point is U+0041.

The representation of that scalar value in UTF-8 is a single 8 bit code unit, with value 0x41. The UTF-16 representation is a single 16 bit code unit, with the value 0x0041. The UTF-32 representation is a single 32 bit code unit, with the value 0x00000041.

Character Hiragana MA

- abstract character:
the letter ま of the Hiragana script
- coded character:
name: HIRAGANA LETTER MA
code point: U+307E
- encoding forms:
UTF-8: E3 81 BE
UTF-16: 307E
UTF-32: 0000307E

The abstract character ま of the Hiragana script is represented by the coded character named HIRAGANA LETTER MA (there is no case in that script), and its code point is U+307E.

The representation of that scalar value in UTF-8 is three 8 bit code units, with values 0xE3, 0x81, 0xBE. The UTF-16 representation is a single 16 bit code unit, with the value 0x307E. The UTF-32 representation is a single 32 bit code unit, with the value 0x0000307E.

Character Deseret AY

- abstract character:
the letter **ᐃ** of the Deseret script
- coded character:
name: DESERET CAPITAL LETTER AY
code point: U+1040C
- encoding forms:
UTF-8: F0 90 90 8C
UTF-16: D801 DC0C
UTF-32: 0001040C

Finally, the character **ᐃ** of the Deseret script is represented by the coded character named DESERET CAPITAL LETTER AY, and its code point is U+1040C. This code point is in the Supplemental Multilingual Plane.

The representation of that scalar value in UTF-8 is four 8 bit code units, with values 0xF0, 0x90, 0x90, 0x8C. The UTF-16 representation is two 16 bit code units, with the value 0xD801, 0xDC0C. The UTF-32 representation is a single 32 bit code unit, with the value 0x0001040C.

Deseret is a phonemic alphabet devised to write the English language. It was originally developed in the 1850s at the University of Deseret, now the University of Utah. It was promoted by The Church of Jesus Christ of Latter-day Saints, also known as the “Mormon” or LDS Church, under Church President Brigham Young (1801-1877). The name Deseret is taken from a word in the Book of Mormon defined to mean “honeybee” and reflects the LDS use of the beehive as a symbol of cooperative industry.

The Church commissioned two typefaces and published four books using the Deseret Alphabet. The Church-owned Deseret News also published passages of scripture using the alphabet on occasion. In addition, some historical records, diaries, and other materials were handwritten using this script, and it had limited use on coins and signs. There is also one tombstone in Cedar City, Utah, written in the Deseret Alphabet. However, the script failed to gain wide acceptance and was not actively promoted after 1869. Today, the Deseret Alphabet remains of interest primarily to historians and hobbyists.

The script consists of thirty-eight letters. The alphabet is bicameral; capital and small letters differ only in size and not in shape. The order of the letters is phonetic: letters for similar classes of sound are grouped together. In particular, most consonants come in unvoiced/voiced pairs.

Terminology

Basic Type	Character status	Code point status
Graphic	Assigned to abstract character	Designated (assigned) code point
Format		
Control		
PUA		
Surrogate	Not assigned to abstract character	Undesignated (unassigned) code point
Noncharacter		
Reserved		

code points = scalar values + surrogates

We have seen a few names which have a precise meaning in the Unicode model: code point, scalar value, Private Use Area, surrogate. Here is the complete terminology.

Let's look first at left column of the table, which shows how the code points classified. As we have just seen, the code points are made of the *surrogates* (D800 - DFFF) and the *scalar values* (0000 - D7FF, E000 - 10FFFF). Scalar values are further decomposed into:

- *graphic* characters, that is letters, marks, numbers, punctuations, symbols and spaces.
- *format* characters invisible but they affect neighboring characters (e.g., line and paragraph separators)
- *controls* (U+0000 - U+001F, U+007E, and U+0080 - U+009F), inherited and extended from the ASCII control characters
- *PUA* or Private Use Area characters
- *noncharacter* scalar values. Those will never be assigned abstract characters, they can be used by applications as sentinels, and they can never be exchanged.
- *reserved* scalar values are the remaining ones, i.e. those that have not been given any function yet.

There are two additional ways to group the code points, as indicated by the other columns of the table.

One way is to ask: “has this code point been assigned to an abstract character?”, and the answer is “yes” for the graphic, format, control and PUA code points, “no” for the others.

The other way is to ask: “has this code point been designated for some use?”, and the answer is “no” for the reserved code points, “yes” for the others.

Part III

Principles of the Abstract Character Set

Principles

- characters, not glyphs
- plain text only
- unification, within each script, across languages
- well-defined semantics for characters
- dynamic composition of marked forms
- equivalence for precomposed forms
- characters are stored in logical order
- round-tripping with some other standards

From the early days, Unicode adopted a set of ten principles and all but one still apply today. The first principle was to have a uniform representation of characters by 16 bit numbers, but as we have seen, this proved too limiting. The second is about efficient processing of Unicode, and it mostly concerns the coded character set (characters of the same script are mostly grouped, frequent characters are in the BMP) and the encoding forms (which have no escape and are self-synchronizing).

The remaining eight principles deal with the abstract character set, that is the selection of characters to encode, and we will examine them now.

Characters, not glyphs

- the character U+0041 can equally well be displayed as *A, A, A, ...*
- sometimes different glyphs are required: U+0647: *o a e e*
- going from characters to glyphs: *shaping*

प/ूर्/र/्त/ि



Unicode decided to encode characters rather than glyphs. This means that U+0041 for example, represents the Latin capital A character, rather than one of the many possible shapes which can be used to display it, such as upright or italic, with or without serif, etc. Any of these many shapes is a legitimate representation of the character A, and none is privileged by the Unicode standard.

In some scripts, such as Arabic, the same letter takes different shapes depending on its context. For example, the letter HEH takes different shapes when isolated *o*, connecting to the right *a*, connecting to both sides *e* or connecting to the left *e*. However, all these shapes represent the same letter, so there is a single abstract character for it.

As a consequence of this principle, the process of going from characters to glyphs, known as *shaping* or *rendering* can be fairly complex. A well-known example is the shaping of Indic scripts. Here we see how a sequence of six Devanagari characters is typically rendered.

Plain text only

- *plain text must contain enough information to permit the text to be rendered legibly, and nothing more*
- e.g. small capitals are not encoded for English
- different requirements for English and IPA
 - U+0262 LATIN LETTER SMALL CAPITAL **G**
 - voiced uvular stop in IPA
 - not used in English

This principle is closely related to the previous one.

Unicode decided to deal with plain text only, defined as *enough information to permit the text to be rendered legibly, and nothing more*.

For example, small capitals are not necessary to write plain text English, and are therefore not encoded. Similarly, the representation of superscripts and subscripts is not necessary in plain text, and those forms are not encoded.

However, some small capitals are necessary to write in the IPA (International Phonetic Alphabet). IPA uses a small capital G to write a voiced uvular stop. For this specific use, Unicode has an abstract character, encoded at U+0262. This character is meant to write IPA only, not English.

Unification, within each script, across languages

- no distinction between English **A** and French **A**
U+0041 LATIN CAPITAL LETTER A
- single **,** regardless of its usage
- no confusion between Latin **A**, Greek **A** and Cyrillic **A**
U+0041 **A** LATIN CAPITAL LETTER A
U+0391 **A** GREEK CAPITAL LETTER ALPHA
U+0410 **A** CYRILLIC CAPITAL LETTER A
- fairly specific rules for Han unification
Chinese hanzi
Japanese kanji
Korean hanja
Vietnamese Chữ hán

Unicode decided not to distinguish a **A** used in English from an **A** used in French. Both letters have the same history, and are therefore unified, i.e. represented by the single character U+0041 LATIN CAPITAL LETTER A.

A similar argument was invoked to encode a single version of many punctuation marks. For example, there is a single “,” shared by most scripts.

On the other hand, consider the Latin A, the Greek Alpha and the Cyrillic A; they clearly have a common ancestry, and can even share the same glyph in many typefaces. Yet, they have different functions and so have not been unified.

The unification of the East-Asian ideographs (Chinese hanzi, Japanese kanji, Korean hanja, Vietnamese Chữ hán), collectively known as the Han ideographs, follows its own set of specific rules (see the Unicode standard for the details).

Well-defined semantics for characters

- the intended use of a character is unambiguous
- the behavior of a character is unambiguous
 - properties
 - algorithms

To support the interchange of text data, it is critical that the intended use of a character be as unambiguous as possible.

Futhermore, the behaviour of a character during processing must be well known. Unicode achieves this by specifying properties of characters and describing algorithms using those properties. We will look at some of those later.

Dynamic composition

- marked forms are a productive mechanism in writing systems
 - accents in Latin
 - negation in Math
 - vowels in Hebrew and Arabic
 - nukta in Indic scripts
 - etc.
- built from components:
 - é : U+0065 e U+0301 ◌̇
 - ∄ : U+2208 ∈ U+0338 ∄
 - ॢ : U+0921 ङ U+093C ◌̣

Marked forms are a productive mechanism of writing systems. They are common as accents in Latin, negation signs in mathematics, vowels in Arabic and Hebrew, nuktas in Indic scripts, etc.

Because there are many combinations, and because new combinations are routinely created when new orthographies are developed, it is not practical to encode the set of marked forms. Instead, Unicode provides for the dynamic composition of such forms, and encodes the marks as *combining characters*. Thus, a marked form such as é is encoded by the base form U+0065 e followed by the combining character U+0301 ◌̇.

Note the convention of displaying combining characters using a dotted circle, which shows how the mark gets positioned with respect to the base form it modifies. Of course, this dotted circled is used only when explaining the Unicode standard.

Dynamic composition (2)

- can have multiple marks
- from base character outwards

u ö õ → ū

u õ ö → ū

a ̇ ̈ → ä

a ̈ ̇ → ä

Multiple marks can be attached to a single base form. They are stacked outwards from the base letter, i.e. the first one is closest to the base form, and the last one is farthest. The relative order of marks that do not interact (e.g. ̇ and ̈) does not matter.

Equivalence for precomposed forms

- some precomposed characters are encoded
U+00E9 é
- canonical equivalence
U+00E9 é ≡ U+0065 e U+0301 ˆ
- similarly for Hangul syllables
U+D4DB 풀 ≡ U+1111 ㅍ U+1171 ㅏ U+11B6 ㅎ
- *A process shall not assume that the interpretation of two canonically equivalent character sequences are distinct*

For compatibility with existing standards, a number of precomposed characters are directly encoded in Unicode. For example, é is encoded as a single character as U+00E9 (more on that later).

This means that the same result can be expressed in two different ways: either as U+00E9, or as U+0065 U+0301. But it is important to understand that both forms are equivalent, so Unicode formally defines the notion of canonical equivalence to record that state of affair.

Canonical equivalence is also used heavily with Hangul syllables: Unicode encodes both the *jamo*, which are individual components representing consonant and vowel sounds, as well as the commonly used combinations of those into syllables.

When presented with two canonically equivalent sequences, a process should not assume that they have a different meaning. As much as possible, it should treat these as entirely equivalent.

Characters are stored in logical order

- logical order ~ pronunciation order ~ typing order

storage	display
ASDF	ASDF
גבנש	שנבג

- combining marks (when separate) follow their base character

When considering the scripts of the world, the order in which characters should be stored is not as trivial as it seems when thinking about English. Unicode chose to use the logical order, which is more or less the pronunciation order, and is often the typing order. (There is one notable exception with Thai.)

We will see later how the gap between storage order and display order is bridged.

Combining marks are stored after the base character they modify.

Round-tripping with some other standards

- the price for acceptance
- often at odds with other principles
- extra characters:
U+00E9 **é**, U+FB00 **ff**, U+F900 **豈**
- canonical decomposition
U+00E9 **é** ≡ U+0065 **e** U+0301 **◌́**
U+F900 **豈** ≡ U+8C48 **豈**
- compatibility decomposition
U+FB00 **ff** ≈ U+0066 **f** U+0066 **f**

A great concern of the Unicode designers was to produce a practical solution. They decided that exact round-tripping with the then common character standards was necessary, so that an application could use Unicode for its internal representation of characters, yet consume and produce character streams using the character standards imposed by other systems.

This principle is often at odds with the other principles of Unicode. For example, the inclusion of precomposed characters is the result of the round-tripping principle, yet it is somewhat in conflict with the dynamic composition principle. As we have seen, this was resolved by the canonical equivalence mechanism.

There are other examples of such conflicts. In some East Asian standard, the character **豈** is encoded twice, to capture different pronunciations. Unicode needs to have two coded characters for this character, to support round-tripping. At the same time, Unicode neutralizes this difference by establishing the two coded characters as canonically equivalent.

In some cases such as **ff**, the characters that are encoded for compatibility purpose capture a glyph difference which Unicode does not wish to make. Yet, such a difference cannot be entirely ignored because of round tripping, so Unicode provides a *compatibility* decomposition to support meaningful processing.

Part IV

The characters in 5.0

How many?

- 99,024 assigned graphic and format characters
 - 71,226 Han ideographs
 - 11,172 Hangul syllables
 - 16,486 alpha/symbols
 - 140 format characters
- 875,441 reserved, total should last a while!

Here is the complete count of code points, organized by type:

Code Point	Range	Count
	0000-10FFFF	1,114,112
Graphic and Format		99,024
Alpha/Symbols		16,486
Han ideographs		71,226
Hangul syllables		11,172
Format		140
Control	00-1F, 7F, 80-9F	65
PUA	E000-F8FF, planes 15 & 16	137,468
Surrogate	D800-DFFF	2,048
Noncharacter	FDD0-FDEF, xxFFFE, xxFFFF	66
Reserved		875,441

The interesting numbers are:

- the number of graphic and format code points, i.e. those to pick from when representing some text using Unicode. The bulk of those are the Han ideographs. There is also a fair chunk for the Hangul syllables. Nevertheless, that are still 16k characters or so to choose from for all the other scripts!
- the total number of reserved code points. Given the rate at which characters are added, this should last quite a while!

The scripts

Latin	IPA	Greek and Coptic
Cyrillic	Armenian	Hebrew
Arabic	Syriac	Thaana
Bengali	Gurmukhi	Gujarati
Oriya	Tamil	Telugu
Kannada	Malayalam	Sinhala
Thai	Lao	Tibetan
Myanmar	Georgian	Hangul
Ethiopic	Cherokee	Canadian Aboriginal
Ogham	Runic	Tagalog

All these characters cover quite a number of writing systems!

The scripts (2)

Hanunoo	Buhid	Tagbanwa
Khmer	Mongolian	Hiragana
Katakana	Bopomofo	Kanbun
CJK Ideographs	Yi	Old Italic
Gothic	Deseret	Musical Symbols
Arrows	Math Operators	Math Symbols
Misc Technical	Control Pictures	OCR
Enclosed Alpha.	Box Drawing	Block Elements
Geometric Shapes	Misc Symbols	Dingbats
Braille Patterns		

and some more...

The scripts (3)

Limbu	Tai Le	UPA
Linear B	Aegean numbers	Ugaritic
Shavian	Osmanya	Cypriot syllabary
Hexagrams	Tetragrams	New Tai Lue
Buginese	Glagolitic	Coptic
Tifinagh	Syloti Nagri	Old Persian
Kharoshthi	Balinese	N'Ko
Phags-pa	Phoenician	Sumero-Akkadian Cuneiform

and some more! The last five are the new scripts in 5.0.

Block descriptions

Greek: U+0370–U+03FF

The Greek script is used for writing the Greek language and (in an extended variant) the Coptic language. The Greek script had a strong influence in the development of the Latin and Cyrillic scripts.

The Greek script is written in linear sequence from left to right with the occasional use of nonspacing marks. Greek letters come in upper- and lowercase pairs.

Standards. The Unicode encoding of Greek is based on ISO 8859-7, which is equivalent to the Greek national standard ELOT 928. The Unicode Standard encodes Greek characters in the same relative positions as in ISO 8859-7. A number of variant and archaic characters are taken from the bibliographic standard ISO 5428.

Polytonic Greek. Polytonic Greek, used for ancient Greek (classical and Byzantine), may be encoded using either combining character sequences or precomposed base plus diacritic combinations. For the latter, see the following subsection, "Greek Extended: U+1F00–U+1FFF."

Nonspacing Marks. Several nonspacing marks commonly used with the Greek script are found in the Combining Diacritical Marks range (see *Table 7-1*).

Table 7-1. Nonspacing Marks Used with Greek

Code	Name	Alternative Names
U+0300	COMBINING GRAVE ACCENT	<i>varia</i>
U+0301	COMBINING BREVE ACCENT	

As can be expected, the bulk of the Unicode standard describes the characters which are encoded and how to use them.

Here we see the beginning of the description of the Greek block. Generally, those descriptions provide some historical background on the script, cover the principles of the encoding, highlight the use of some characters, and point out some particularities of processing the script.

Code charts

Greek and Coptic

	037	038	039	03A	03B	03C	03D	03E	03F
0			ι 0390	Π 03A0	ϖ 03B0	π 03C0	β 03D0	ϑ 03E0	χ 03F0
1			Α 0391	Ρ 03A1	α 03B1	ρ 03C1	θ 03D1	□ 03E1	ϱ 03F1
2			Β 0392		β 03B2	ς 03C2	Υ 03D2	Ψ 03E2	Ϸ 03F2
3			Γ 0393	Σ 03A3	γ 03B3	σ 03C3	Ϛ 03D3	ϣ 03E3	ϣ 03F3

The code charts give details for each character. Here is the beginning of the code chart for Greek and Coptic. You can see the code point as well as a representative glyph for that character.

Remembering the “characters, not glyphs” principle, it is important to realize that the glyphs shown in the charts are in no way normative. Their only purpose is to illustrate the identity of the characters.

Code charts (2)

Based on ISO 8859-7

0374	'	GREEK NUMERAL SIGN		0389	H	GREEK
		= dexia keraia				≡ 0397 H
		• indicates numeric use of letters		038A	I	GREEK
		→ 02CA ' modifier letter acute accent				≡ 0399 I
		≡ 02B9 ' modifier letter prime		038B	<reserved>	
0375	,	GREEK LOWER NUMERAL SIGN		038C	Ο	GREEK
		= aristeri keraia				TONOS
		• indicates numeric use of letters				≡ 039F O
		→ 02CF , modifier letter low acute accent		038D	<reserved>	
0376		<reserved>		038E	Υ	GREEK
0377		<reserved>				TONOS
0378		<reserved>				≡ 03A5 Y
0379		<reserved>		038F	Ω	GREEK
037A		GREEK YPOGEGRAMMENT				TONOS
						≡ 03A9 Ω

The characters are also described by their names. In this example, we see some of the other annotations, introduced by specific marks:

- = for an alternate, but non-normative, name
- • for an annotation that clarifies the usage of the character
- → for related but distinct characters
- ≡ for canonical decompositions
- ≈ for compatibility decompositions

Part V

Development of the standard

Unicode Inc.

- the Unicode standard grew from work at Xerox and Apple
- the Unicode Consortium was incorporated in 1991
- six levels of membership
- ~120 members: companies, governments, individuals and organizations; ~20 voting members

A little bit of history:

The Unicode standard grew from work at Xerox and Apple in the late 80s.

The Unicode Consortium was incorporated in 1991, and has about 120 members today; about 20 are voting members.

Technical committees

- UTC: defines The Unicode Standard and associated standards and technical reports
- CLDR-TC: manages the Common Locale Data Repository and associated standards and technical reports

The technical work is performed by two committees, composed of representatives of the members.

Standards

- The Unicode Standard
- A Standard Compression Scheme for Unicode
- Unicode Collation Algorithm
- Unicode Regular Expression Guidelines
- Character Mapping Markup Language
- Local Data Markup Language (LDML)
- Ideographic Variation Database

The Unicode Consortium currently publishes seven standards, the most important of which is of course The Unicode Standard.

CLDR and LDML

- CLDR: Common Locale Data Repository
 - collects and organizes locale data for the world
 - highly cooperative effort
 - formatting (and parsing) of numbers, dates, times, currency values, ...
 - display names for language, script, region, currency, time-zones, ...
 - collation order (used in sorting, searching, and matching text)
 - identifying usage of measurement systems, weekend conventions, currencies, ...
- LDML: Locale Data Markup Language
 - the XML markup in which the CLDR is represented

The Common Locale Data Repository is a highly cooperative effort which attempts to solve the next layer of internationalization.

The Unicode Standard

- three levels of versions:
 - major (4.0): a new book is published
 - minor (4.1): no new book, but new characters
 - dot (4.0.1): no new characters
- stability guarantees
 - to ensure that data is perennial
- standard comprises:
 - a book
 - annexes (will be part of the 5.0 book, separate before)
 - the UCD
 - a release description, for non-major releases

The Unicode Standard is versioned. There are three different levels of versions, to capture whether the standard is published in book form, and whether new characters are added.

The Consortium offers stability guarantees, that is properties relating successive versions; the goal primarily to ensure that data which is created with one version of the standard remains valid and keeps the same meaning in future versions.

The manifestation of a given version of the standard comprises:

- the book of the corresponding major version, as a base text
- may be new versions of the standard annexes, which are an integral part of the standard. The UAXes are for those areas of the standard which see a faster evolution than the publication as a book permits
- the Unicode Character Database (UCD)
- a description of the character additions and/or changes for that particular version

ISO/IEC 10646

- defined by JTC1/SC2/WG2
- aligned with Unicode, via cooperation
 - same repertoire
 - same character names
 - same code points
- does not define properties or processing
- current: ISO/IEC 10646:2003 + Amendments 1 and 2 + 5 characters corresponds to Unicode 5.0

The ISO standard 10646 is aligned with the Unicode standard. Both standards define the same repertoire of characters, using the same character names, and the same code points.

The main difference is that Unicode also handles the processing of characters (ISO 10646 simply enumerates them). On the other hand, the structure of ISO (one vote per participating country) is more acceptable in some regions of the world.

Unicode 5.0 includes 5 characters which are not present in the corresponding version of 10646, but have already been approved for the next version of 10646.

Part VI
Processing

Properties and algorithms

- each character has a number of properties in the Unicode Character Database (UCD)
- algorithms based on properties easy to upgrade to a new repertoire in the standard or in technical reports

We indicated earlier that Unicode is not simply a repertoire of characters, but also intends to help proper processing of characters.

Each character has a number of properties, recorded in the Unicode Character Database.

Unicode also specifies a number of algorithms, in the standard itself and in technical reports, to perform common operations. These algorithms are driven by the property values.

Properties and algorithms are separated, such that when new characters are added to Unicode, it is most often enough to replace the data files that drive the algorithms, and the code itself does not have to be modified.

In the rest of this section, we will look at some of the most important algorithms. The purpose is not so much to equip you to implement them (you should use some kind of library), but rather to give you a feeling for processing of characters.

You are encouraged to look at the complete list of technical reports.

The Bidirectional Algorithm

- text is stored in logical order: גבנש \$10.
- bidi computes the display order: . \$10ג ש
- characters have directionality:

chars	directionality
A, B, C	L - Left to Right (strong)
ש, ג, ב, א	R - Right To Left (strong)
1, 0	EN - European Number (weak)
\$	ET - European Number Terminator (weak)
.	CS - Common Number Separator (weak)

Remember that text is stored in logical order. The function of the bidirectional algorithm is to compute the display or visual order.

The *directionality* property captures how a given character behaves in bidirectional text. For example, the Latin letters have a strong Left to Right directionality, while the Hebrew letters have a strong Right to Left directionality. Many other characters have a weak directionality, which means that it needs to be adapted to the context in which those characters appear.

The Bidirectional Algorithm (2)

- bidi resolves the directionality of weak characters
 - stored גבנש \$10.
 - resolved גבנש \$10.
 - displayed . \$10 שנבג
- context matters; e.g. adding a 5
 - stored גבנש \$10.5
 - resolved גבנש \$10.5
 - displayed \$10.5 שנבג
- format characters to handle special cases
 - stored <RLO>abc<PDF>def
 - resolved abc def
 - displayed def cba

To compute the visual order, bidi resolves the directionality of weak characters. In the first example, the final period is determined to function as an end of sentence marker and is resolved to be Right to Left.

In the second example, where a 5 was added at the end, the period is determined to function as a decimal separator and is resolved to be Left to Right.

Unicode also include a few format characters to control the bidi algorithm in special circumstances. In the last example, the RLO/PDF pair of characters forces the directionality of “abc” to Right to Left (RLO stands for Right to Left Override, and PDF for Pop Directional Formatting). In general, format characters do not display, they only affect the display of the characters around them.

The Bidirectional Algorithm (3)

- shape of character can depend on directionality
U+0028 LEFT PARENTHESIS
function is opening parenthesis
displays as (in ltr
displays as) in rtl
- captured in the mirrored and mirror glyph properties
can be overridden by higher level protocols

A couple of details:

Some characters are rendered by different shapes depending on the context in which they appear. For example, the function of U+0028 LEFT PARENTHESIS is to open a parenthesis, and it should display as (in a left to right context, and as) in a right to left context.

(Yes, the name OPENING PARENTHESIS would be better; but remember that Unicode decided it was less convenient to change the character names than to document them. Here is one example).

Unicode Normalization Forms

- multiple representations of the “same” text
é vs. e ◌
- a normalization form selects one of those representations
e.g. allows binary comparisons
- two basic forms
 - NFC: prefers *composed* characters
 - NFD: prefers *decomposed* characters
- guarantee of stability
e.g. for databases

As we have seen earlier, there are cases where the same text can be represented in more than one way. A normalization form is the selection of one of the representation among all the equivalent representations.

Using normalized text makes some operations simpler. For example, string equality can be logically implemented as normalization followed by binary comparison.

There are two basic forms of normalization: in NFC, the precomposed form of the characters is preferred; in NFD, the decomposed form is preferred.

Both forms have a guarantee of stability, i.e, given a string of characters, its NFC and NFD normal forms will never change. This is important for use in databases, where performance is achieved by binary comparisons; normalization occurs when data is inserted in the database, and normalization also occurs (on the query) when data is queried.

Canonical Decomposition Property

- canonical decomposition is a property
- maps one character to one or more characters
- includes:
 - combining sequences: é ≡ e ◌́
 - Hangul syllables: 풀링 ≡ ㅍ ㄹ ㅇ
 - singletons: Ω ≡ Ω (ohm sign and omega)

Before we see how NFC and NFD forms are computed, we need to understand the canonical decomposition of characters.

Each character can have the canonical decomposition property. The value of this property is one or more characters, into which the original character decomposes. Not all characters have a decomposition; in fact most of them (such as **A**) do not have one.

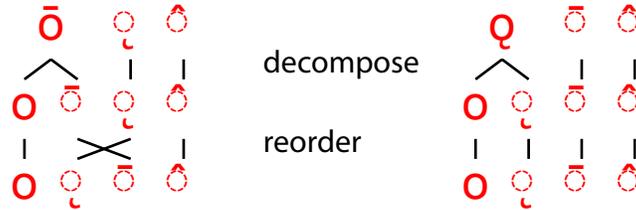
Precomposed characters have the expected decomposition into a base character followed combining mark character(s).

Hangul syllables decompose in their jamo sequence.

There are also a few singleton decompositions, where one character decomposes into a single character. A typical example is U+2126 OHM SIGN which decomposes into U+03A9 GREEK CAPITAL LETTER OMEGA.

NFD

- apply repeatedly the canonical decompositions
- reorder combining marks by increasing combining class



Normal Form D (for *decomposed* or NFD) prefers the decomposed form of characters.

The first step is to apply repeatedly the canonical decompositions until none can be applied.

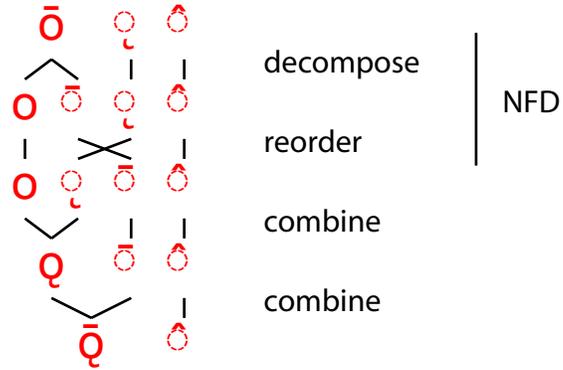
The second step takes care of non-interacting combining marks. In the example on the left, the base character, **o**, carries three combining marks. Two are above: the macron **̄** and the circumflex **ˆ** and they interact, in the sense that the circumflex is above the macron, not the other way around. One mark is below: the ogonek **˛**. The ogonek does not interact with the macron and circumflex; in the character stream, it could appear just as well before the macron or after the circumflex. The reordering select one particular order (ogonek first, macron and circumflex second).

The example on the right shows a canonically equivalent sequence, and how it leads to the same NFD form.

Two character sequences which have the same NFD representation are said to be canonically equivalent. Canonical decompositions are carefully built such that a character and its canonical decomposition are canonically equivalent.

NFC

- transform to NFD
- recombine combining sequences



Normal Form C (for *composed*, or NFC) prefers the composed form of characters.

The first step is to transform the string to NFD.

The second step is to recombine combining sequences as much as possible. In our example, $\text{O} \text{̄}$ are recomposed into U+01EB $\text{O} \text{̄}$, which is further recomposed with $\text{O} \text{̃}$ into U+01ED $\text{O} \text{̄} \text{̃}$.

This description of the NFC computation is logical. In practice, there are useful optimizations which avoid unnecessary intermediate steps.

NFKD, NFKC

- also use compatibility mappings when decomposing
- compatibility mappings are a mixed bag
- therefore, NFKD and NFKC are difficult to use

There are two additional normal forms, derived from NFD and NFC. They differ in that they also apply the compatibility decompositions in the first step (the K stands for *compatibility*).

Because the compatibility decompositions are a mixed bag, NFKD and NFKC are difficult to use and are best avoided.

Case mappings

- simple mappings:
 - one to one
 - context independent
 - avoid: insufficient for e.g. ligatures, German
- complex mappings:
 - one to many: $\beta \rightarrow \text{SS}$, or $\text{fi} \rightarrow \text{FI}$
 - contextual: $\Sigma \rightarrow \zeta$ in final position, not σ
 - local-sensitive: $\text{i} \rightarrow \text{İ}$ in Turkish, not I
- case folding for caseless matches

Another set of algorithms defined by Unicode concerns case mapping, that is the replacement of uppercase by lowercase and vice-versa.

Unicode defines simple mappings, where one character is replaced by exactly one character, and those replacements are independent of the context. While such mappings are necessary in some cases, their limitations cannot correctly handle a number of cases.

Unicode also defines complex mappings which more accurately reflect reality. Those mappings may replace a single character by multiple characters (as they do for β or fi). They may be contextual, to cover cases such as a Greek sigma for which there are two forms, one to be used in final position. They are also local-sensitive, to cover cases such as the Turkish dotted i which uppercases to the uppercase I with a dot.

Unicode Collation Algorithm

- many different sorting orders
 - English: *péché* < *pêche*; French: *pêche* < *péché*
 - Spanish (trad): *ch* single letter, between *c* and *d*
 - German (trad): *ö* equivalent to *oe*
- dictionary, phonebook, etc.
- sorting algorithm that:
 - is efficient
 - accounts for canonical and compatibility equivalences
 - can be tailored to implement most orders
 - by default, provides a reasonable sorting

Sorting strings is another important operation in text processing.

Like other aspects of writing systems, sorting conventions vary greatly across cultures. Yet, it is critical to sort according to users expectations: a phonebook or dictionary is not very useful if the sorting is unexpected.

Here are some examples of variation: In English, when two words differ only on the accents, it is the leftmost accent difference that counts; in French, it is the rightmost difference that counts. In traditional Spanish sorting, the two characters *ch* are considered a single letter, which sorts between *c* and *d*. In traditional German sorting, the character *ö* is considered as equivalent to the two letters *oe*, and therefore *öt* sorts after *od* and before *of*, interleaved with *oe*.

Even within one culture, there may be different conventions in different contexts. For example, the dictionary order and the phonebook order may be different.

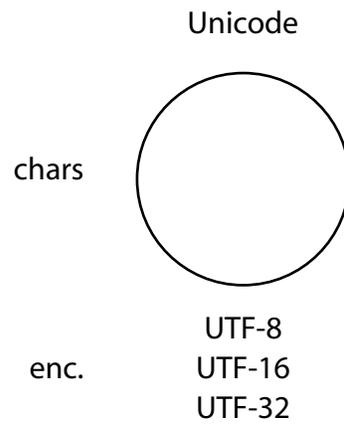
Getting in the details of the Unicode Collation Algorithm is beyond the scope of this presentation, so we will just mention a few characteristics here:

- it is an efficient algorithm
- it accounts for canonical and compatibility equivalence
- it can be tailored to implement most sorting orders found in the real world
- even without tailoring, it provides a reasonable sorting order

Part VII

Unicode and other standards

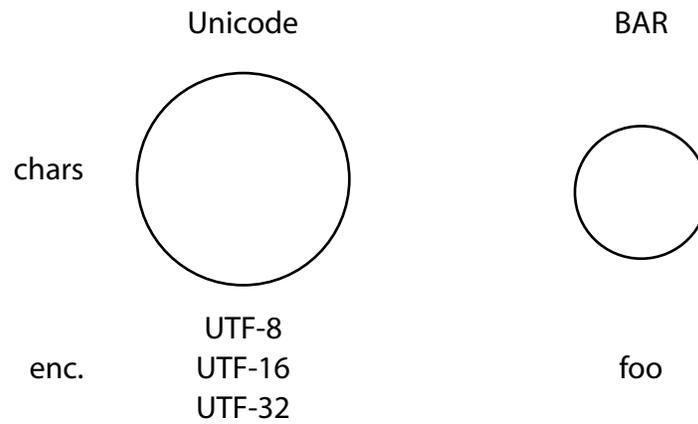
Transcoding



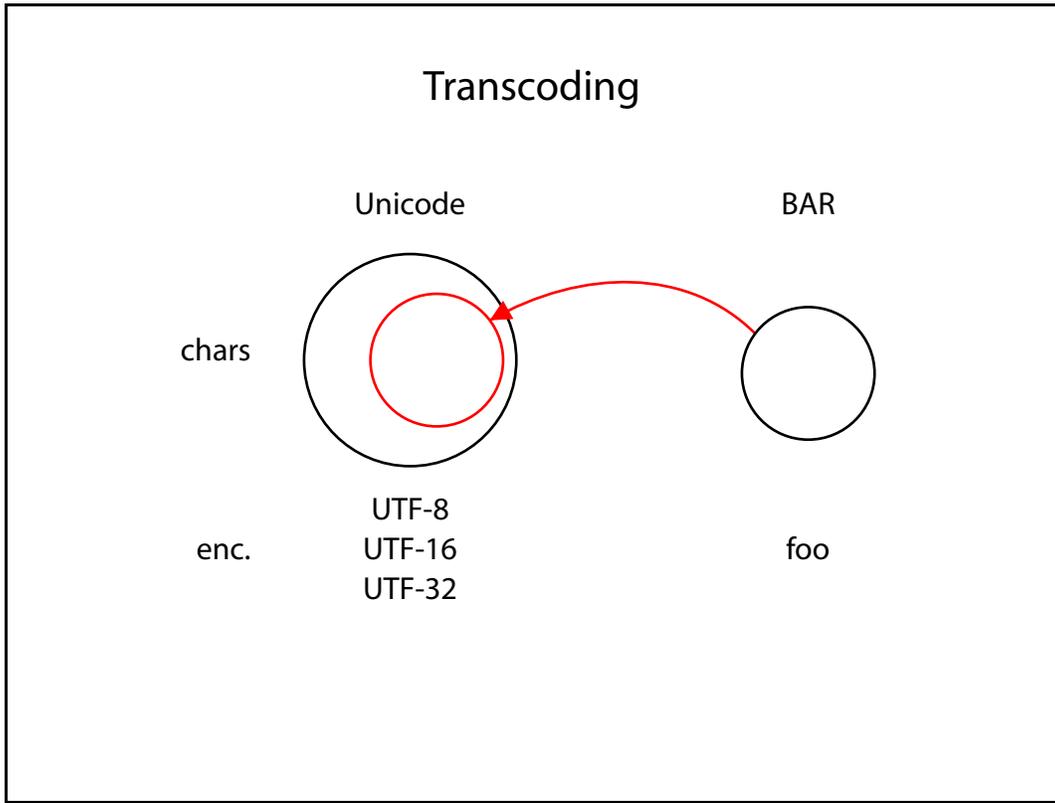
One of the important principles of Unicode is to work relatively well with legacy characters standards. This section explains how this works in practice, and characterizes the level of Unicode implementation needed to deal with some common East Asian standards.

Remember our character model. For this discussion, we are going to focus on the coded character set layer and on the encoding forms layer. (More precisely, we are going to merge encoding forms and encoding schemes; there is no loss of generality in doing so.)

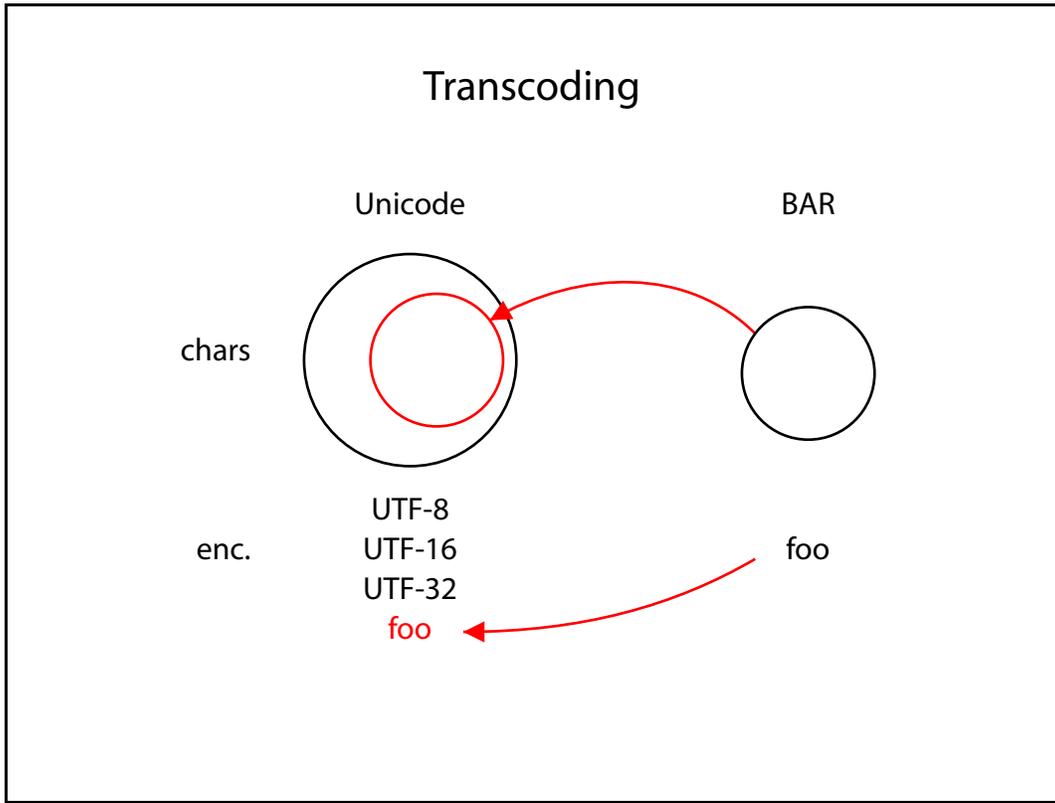
Transcoding



Our character model is not specific to Unicode. In fact, we can apply it to most other character standards. Here, we will take the example of the BAR standard, and assume the existence of an encoding form for it, *foo*.

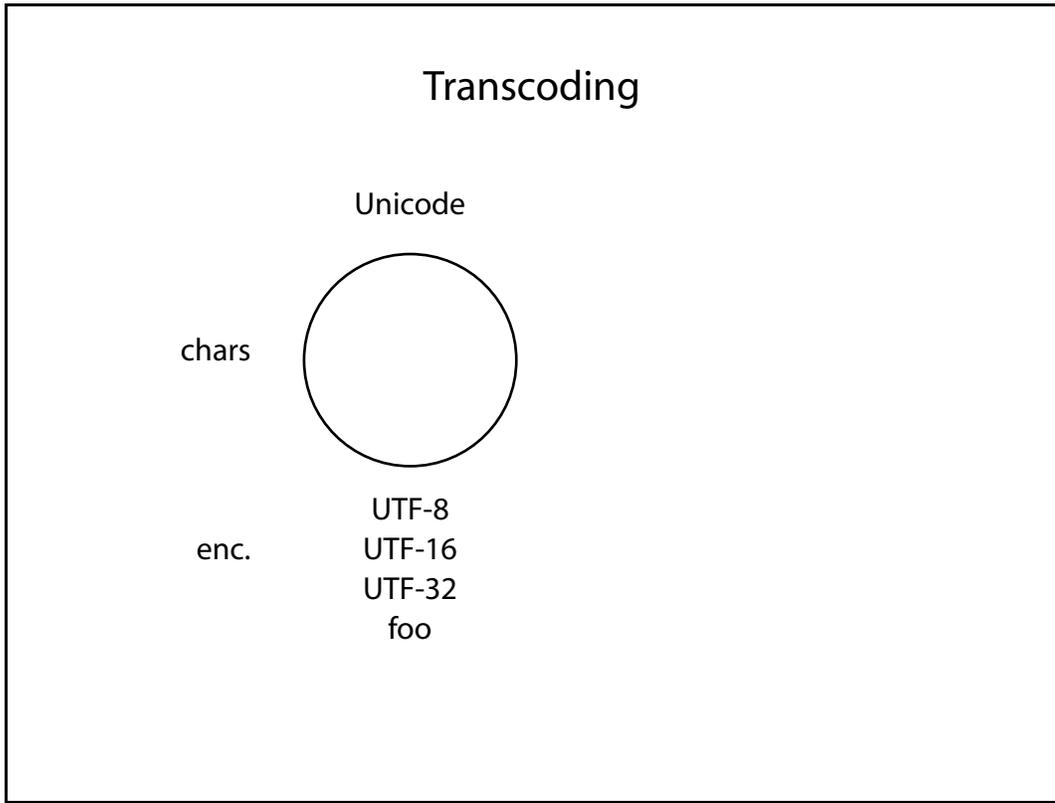


The first piece of magic is to identify the BAR character set with a subset of Unicode. This is possible because Unicode carefully designed its character set for that purpose (remember the inclusion of “duplicate” characters, together with decomposition mechanisms).



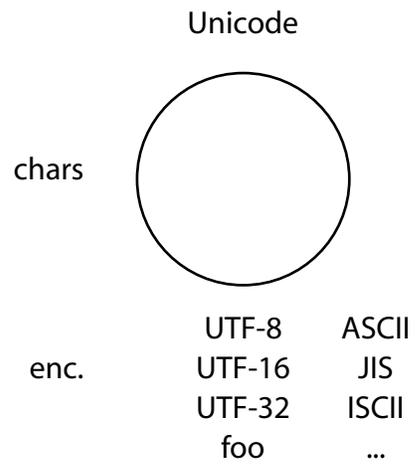
The second piece of magic is to consider *foo* not just as an encoding for the BAR standard, but also an encoding for Unicode.

This encoding may not be as nice as the UTFs. For one thing, it can be used only with the subset of Unicode that is mapped one-for-one from the BAR collection. Also, it may not be self-synchronizing. Nevertheless, it can be viewed as a Unicode encoding.



Having done that, we may as well forget BAR as a character standard on its own, and just view it as no more than an encoding of Unicode. Just like with the UTFs, we need to convert imported text to the internal representation we have selected for our application, and we need to convert exported text from our internal representation.

Transcoding



Of course, this is not limited to BAR: practically every other character standard we care to deal with can be handled in this way.

Character Hiragana MA (revisited)

- abstract character:
the letter ま of the Hiragana script
- coded character:
name: HIRAGANA LETTER MA
code point: U+307E
- encoding forms:
UTF-8: E3 81 BE
UTF-16: 307E
UTF-32: 0000307E
ASCII: n/a
JIS 0208: 245E
KSX 1001: 2A5E

Let's revisit our character Hiragana MA. The abstract character, name, code point and UTF encodings are all the same as before. But we can now add a few more encodings:

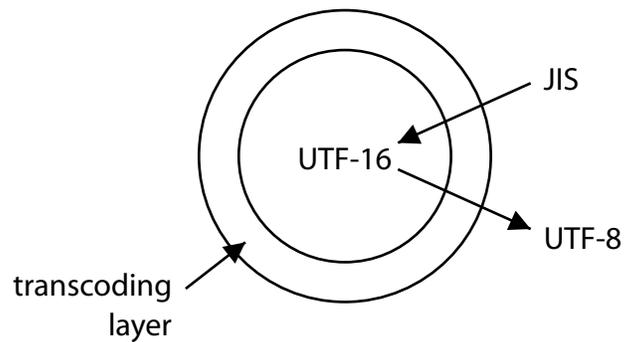
- the ASCII encoding does not support this character
- the JIS 0208 encoding is 245E
- the KS X 1001 encoding is 2A5E

XML

- XML does just that!
- all characters are Unicode characters
- any encoding form (including non-UTFs) is acceptable

It is interesting to note that this is exactly the path taken by XML. In an XML document, all characters are Unicode characters; the XML declaration at the beginning of the document can indicate which encoding form is used, and is not limited to the UTFs.

Anatomy of an implementation

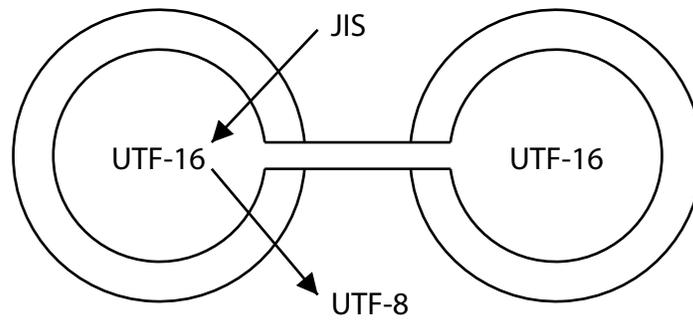


This transcoding capability of Unicode, and in particular its perfect round-tripping with important standards, suggests the following organization for a software component:

The internal representation of text is Unicode, using one of the UTFs. Whenever communication with the outside world happens, the text is transcoded between the encoding form used outside and the encoding form used inside. The outside world does not know any better, but this allows the software to work unchanged anywhere in the world (assuming the appropriate transcoding is in place). It further permits the software to *simultaneously* manipulate text presented to it in different encoding forms, without having to tag every string with the encoding form it uses.

The internal representation could actually be any encoding form, including ASCII. However, the full benefits of the approach are realized only if the internal encoding form supports all the characters that the software could ever encounter, so a UTF is an obvious choice.

Anatomy of an implementation (2)



Applications are typically built out of components. Each component will have its own internal representation and its own transcoding layer. As soon as two components share the same internal representation, it is possible to bypass the transcoding layers, thereby making the implementation more efficient.

A particular case is the platform on which the software runs, such as Windows or MacOS. Selecting the same internal representation as those components makes perfect sense. Since those use UTF-16, that is often a logical choice.

JIS X 0208:1997 and JIS X 0213:2000

- standard:
 - Japan
 - two complementary standards
 - market will probably demand both
- characters:
 - JIS X 0208: 7k
 - JIS X 0213: 4k
 - ~300 in plane 2
- mappings:
 - mapping to Unicode 3.2 does not use PUA
- encoding:
 - 2 bytes

Let's look at a few important East Asian character standard in that light.

First, we have the pair of complementary standards JIS X 0208:1997 and JIS X 0213:2000. Those standards are defined by the Japanese standardization body, and it seems that the Japanese market will demand support for them.

The combined character set is about 11k characters. Considered as Unicode characters, all these characters are in the Unicode 3.2 repertoire, with about 300 in plane 2, the Supplementary Ideographic Plane.

In the “native” encoding (S-JIS), all these characters are represented in one 2 byte unit. Note that in UTF-16, about 300 will need two 16 bit units, and the others will need just one 16 bit unit.

GB 18030-2000

- standard:
 - People's Republic of China
 - mandatory for products sold there
- characters:
 - 28k
- mappings:
 - mapping to Unicode 3.2, BMP only, uses PUA for 25 characters
 - mapping to Unicode 4.1: no PUA
- encoding:
 - 1, 2, or 4 bytes

The GB 18030-2000 standard has been defined by the People's Republic of China (PRC) and its support is mandatory for software sold in the PRC.

This standard is a little bit weird, because it can be interpreted in three different ways. The more conventional way says that it contains about 28k characters. The standard defines a mapping to Unicode 3.2, BMP only, which uses the PUA for 25 characters. It is also possible to construct a mapping to Unicode 4.1 which does not use the PUA at all.

The “native” encoding represents those characters in one, two or four 8 bit code units.

HKSCS

- standard:
 - Hong Kong SAR
 - supplements BIG5
 - mandatory for selling to the government
- characters:
 - 4,818 characters
 - 1,651 in plane 2
- mappings:
 - mapping to Unicode 3.0 uses PUA for 1,686 characters
 - mapping to Unicode 3.1-4.0 uses PUA for 35 characters
 - mapping to Unicode 4.1 does not use the PUA
- encoding:
 - 2 bytes

The HKSCS (Hong Kong Supplementary Character Set) standard is defined by the Hong Kong Special Administrative Region, and its support is mandatory to sell software to the Hong Kong government.

This standard has about 5k characters (to add to the ~13k characters of Big5), a third of those in plane 2.

HKSCS defines a mapping to Unicode 3.0, which uses the PUA for for plane 2 characters (those did not exist in Unicode 3.0) and for a few others. It also defines a mapping to Unicode 3.1, which takes advantage of the plane 2 characters, but still uses the PUA for 25 characters. Finally, it is possible to define a mapping to Unicode 4.1 which does not use the PUA at all.

The “native” encoding represents those characters in one 16 bit code unit.

The bad news

- there is not always an “official” mapping
different vendors do different things
- PUA conflicts:
HKSCS 9571 (U+2721B) ↔ U+E78D
GB18030 A6D9 (,) ↔ U+E78D
- PUA differentiation:
HKSCS 8BFA (U+20087) ↔ U+F572
GB18030 FE51 (U+20087) ↔ U+E816
- no need for PUA with Unicode 4.1

Handling other characters standards via Unicode, in particular the East Asian ones, has a few pitfalls.

First, there is not always an official mapping between another standard and Unicode, and in practice, different vendors have implemented different mappings.

Second, uses of the PUA by mappings of different standards are not necessarily compatible. Sometimes there are conflicts, sometimes there is differentiation.

Consider the case of HKSCS 9571, which is clearly U+2721B: under the mapping to Unicode 3.0, this character is mapped in the PUA to U+E78D. At the same time, the GB 18030 character A6D9, which is clearly not U+2721B, is also mapped in the PUA to U+E78D. In other words, when encountering Unicode text that contains U+E78D, one needs to know if it follows the HKSCS usage of the PUA, or the GB 18030 usage of the PUA (or yet some other usage).

Consider the case of HKSCS 8BFA, which is clearly U+20087: under the mapping to Unicode 3.0, this character is mapped in the PUA to U+F572. At the same time, this same character U+20087 is also present in GB18030 as FE51, and it is also mapped to the PUA, but at U+E816. This time, we have the symmetric situation, where the same character is mapped to different PUA code points.

The good news is that with Unicode 4.1, it is possible to fully support GB 18030 and HKSCS without using the PUA at all.

Part VIII
Resources

Unicode Inc. Resources

- Unicode Consortium: <http://www.unicode.org>
 - UAXes
 - technical reports
 - UCD
 - unibook: application to explore the UCD
 - online Unihan database
- The Unicode Standard, Version 4.0
 - ISBN 0-321-18578-1
 - also at www.unicode.org
- Unicode Guide
 - 6 pages laminated quick study guide
 - ISBN 9781423201809

The Unicode Consortium maintains a web site where you will find the standard itself, the technical reports, the Unicode Character Database, and many other things. Among them is unibook, a very useful program to examine the UCD (the code charts in the published standard are built by this application). There is also the online Unihan database, which permits a more interactive exploration of the Unihan database, which contains all sorts of metadata about the Han ideographs.

The published book can be found in all good bookstores...

There is also a 6 page quick study guide, published by BarCharts.

Internationalization and Unicode Conference

- Internationalization and Unicode Conference
once or twice a year
IUC 30, Washington DC, November 2006
(<http://www.unicode.org/conference>)

The International Unicode Conference is a good place to stay in touch with the Unicode community.

Other Resources

- IBM's site for Unicode: <http://www.ibm.com/developer/unicode>
- International Components for Unicode:
<http://oss.software.ibm.com/developerworks/opensource/icu/project/>
- Mark Davis' site: <http://www.macchiato.com>
- Michael Everson's site: <http://www.evertype.com>
- *Unicode Demystified* by Richard Gillam (ISBN 0201700522), August 2002
- *Unicode Explained* by Jukka Korpela (ISBN 0-596-10121-X), June 2006

There are of course many other web pages about Unicode. Here are some we have found useful.

There are a few books on Unicode besides the Unicode Inc. publications.