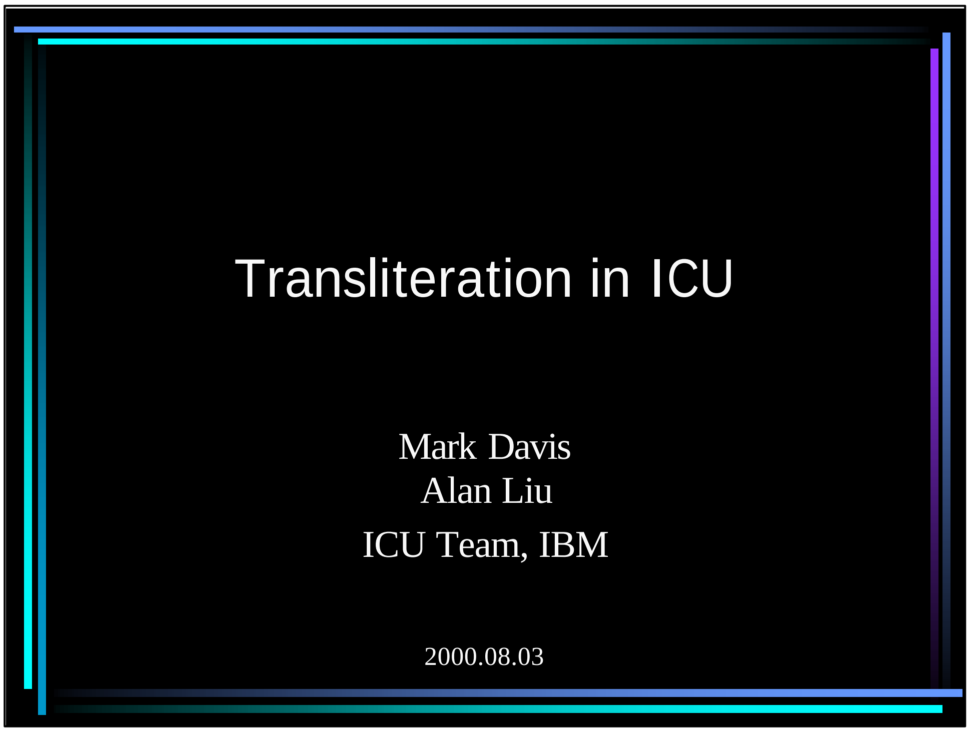
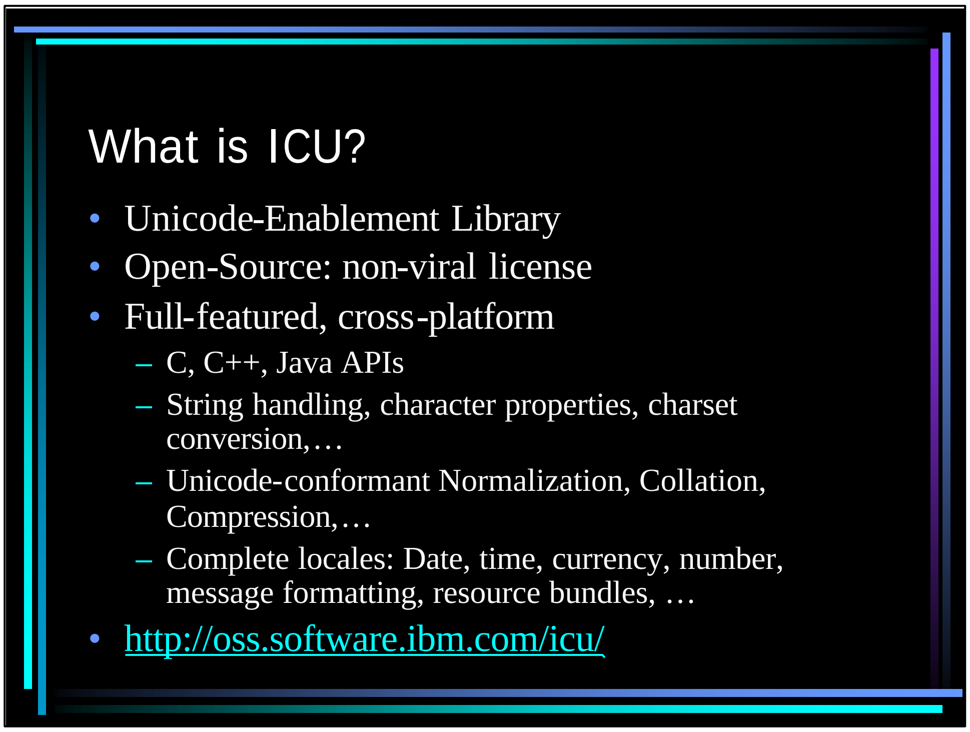
SC22/WG20 N915



Transliteration in ICU

Mark Davis
Alan Liu
ICU Team, IBM

2000.08.03



What is ICU?

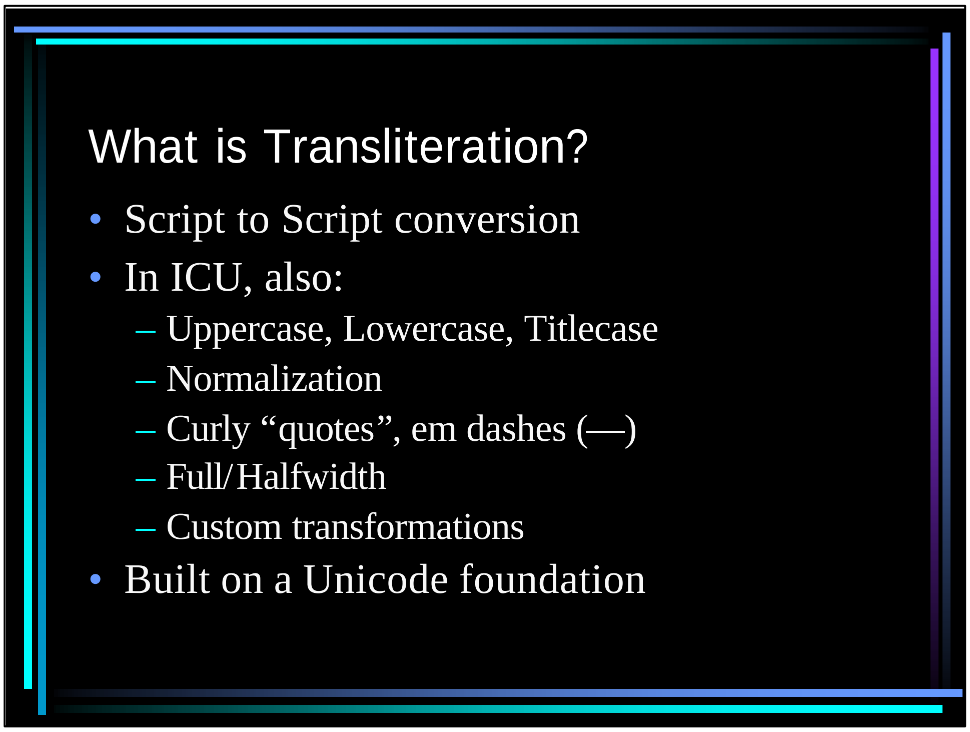
- Unicode-Enablement Library
- Open-Source: non-viral license
- Full-featured, cross-platform
  - C, C++, Java APIs
  - String handling, character properties, charset conversion,…
  - Unicode-conformant Normalization, Collation, Compression,…
  - Complete locales: Date, time, currency, number, message formatting, resource bundles, …
- http://oss.software.ibm.com/icu/

The International Components for Unicode(ICU) is a C and C++ library that provides robust and full-featured Unicode support on a wide variety of platforms. The library provides:

•Calendar support
Character set conversions
Collation (language-sensitive)
Date & time formatting
Locales (170+ supported)
Resource Bundles

•Message formatting
Normalization
Number & currency formatting
Time zones
Transliteration
Word, line & sentence breaks

The ICU project is licensed under the X License, which is compatible with GPL but non-viral.



What is Transliteration?

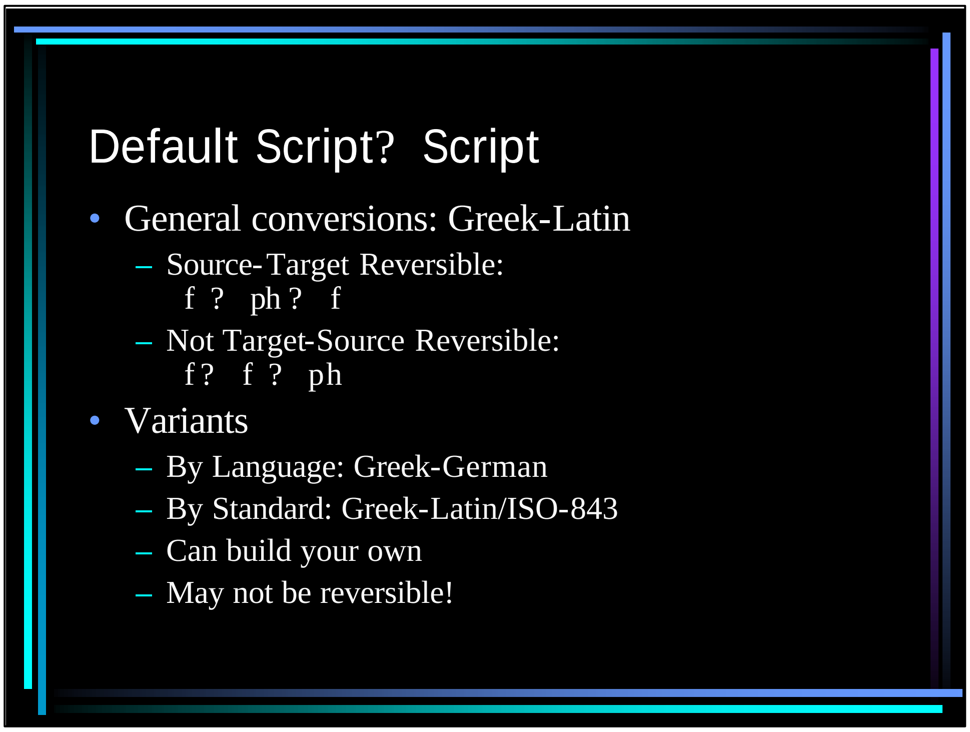
- Script to Script conversion
- In ICU, also:
  - Uppercase, Lowercase, Titlecase
  - Normalization
  - Curly “quotes”, em dashes (—)
  - Full/Halfwidth
  - Custom transformations
- Built on a Unicode foundation

Transliteration is the general process of converting characters from one particular script to another one. This provides a way for people to see personal names or place names in a much more recognizable format.

ICU provides a general mechanism for performing transliterations. It includes a set of standard transliterators, such as Greek or Katakana to Latin. Most of these transliterators also have inverse mappings, which convert in the other direction. Filters can also be specified, so that a transliterator only applies to specific characters. Additional transliterators can be easily built from a series of textual rules (at runtime).

Broadly speaking, ICU transliteration can also include manipulations of characters within a single script, such as upper- and lowercasing, or producing special symbols. For example, this includes converting typewriter 'straight quotes' and fake dashes (--) to curly quotes and long dashes. It can also be used to convert unfamiliar letters within the same script, such as converting Icelandic THORN (þ) to th. It is very important to note that this transliteration is *not* translation. It is converting the letters from one script to another, not translating the underlying words.

Of course, all of this is built, like all of ICU, on a foundation of Unicode. Without Unicode it would be almost impossible to construct efficient software that covers the same range of languages and scripts.

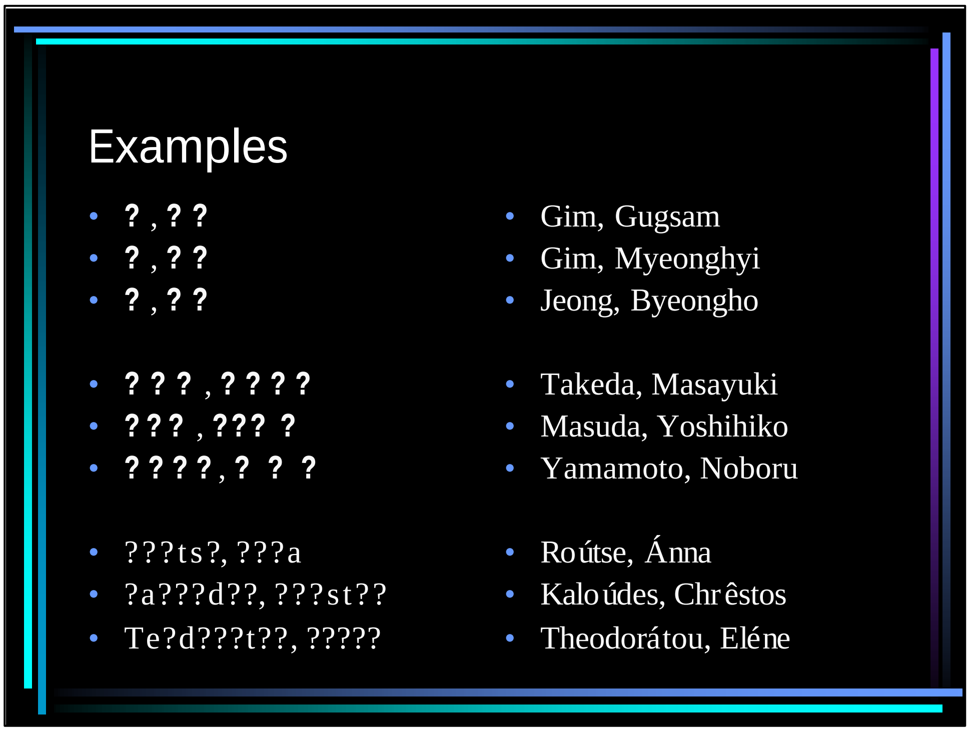


Default Script? Script

- General conversions: Greek-Latin
  - Source-Target Reversible:
    f ? ph ? f
  - Not Target-Source Reversible:
    f ? f ? ph
- Variants
  - By Language: Greek-German
  - By Standard: Greek-Latin/ISO-843
  - Can build your own
  - May not be reversible!

ICU provides a number of default script-to-script transliterations, including transformations to Latin for scripts of the locales supported by ICU, and certain other transformations such as Hiragana-Katakana, and conversions between the Indic scripts. The transformations to the Latin script are source-target reversible.

ICU also provides a number of variant transliterations, and you can build your own. These transliterations may not be reversible, unless you make the extra efforts to make them so!

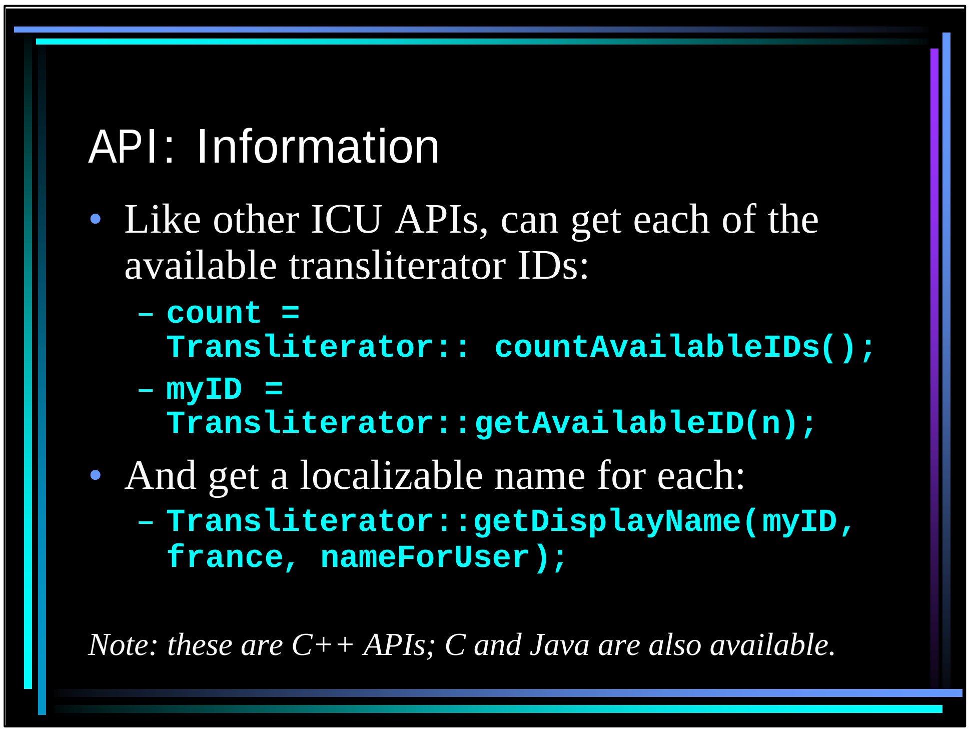


Examples

? , ? ?	Gim, Gugsam
? , ? ?	Gim, Myeonghyi
? , ? ?	Jeong, Byeongho
? ? ? , ? ? ? ?	Takeda, Masayuki
? ? ? , ??? ?	Masuda, Yoshihiko
? ? ? ? , ? ? ?	Yamamoto, Noboru
???ts?, ???a	Roútse, Ánna
?a???d??, ???st??	Kaloúdes, Chrêstos
Te?d???t??, ?????	Theodorátou, Eléne

Here are some examples of script-script transliterations. On the left are customer names from a database. On the right are transliterations; text that will be read far more easily by the average English-speaking database support engineer.

Of course, an Arabic-speaking support engineer might choose to have the names all transliterated into Arabic, instead!

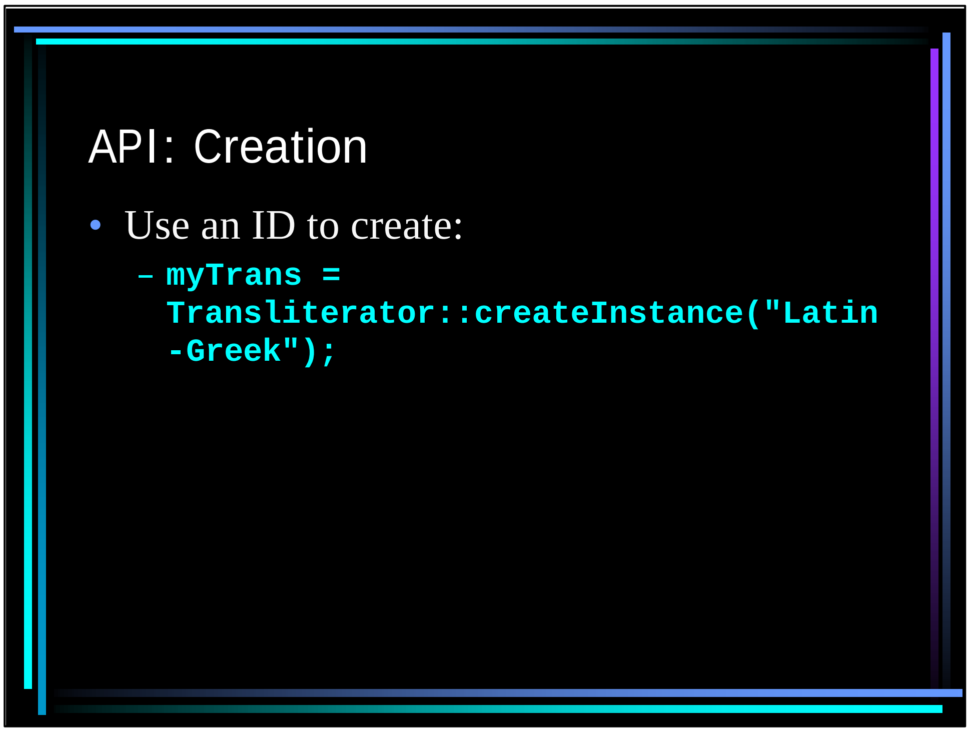


API: Information

- Like other ICU APIs, can get each of the available transliterator IDs:
  - `count = Transliterator:: countAvailableIDs();`
  - `myID = Transliterator::getAvailableID(n);`
- And get a localizable name for each:
  - `Transliterator::getDisplayName(myID, france, nameForUser);`

Note: these are C++ APIs; C and Java are also available.

The API is pretty simple. ICU allows you to get a list of all the available transliterator IDs, and localizable names for them.



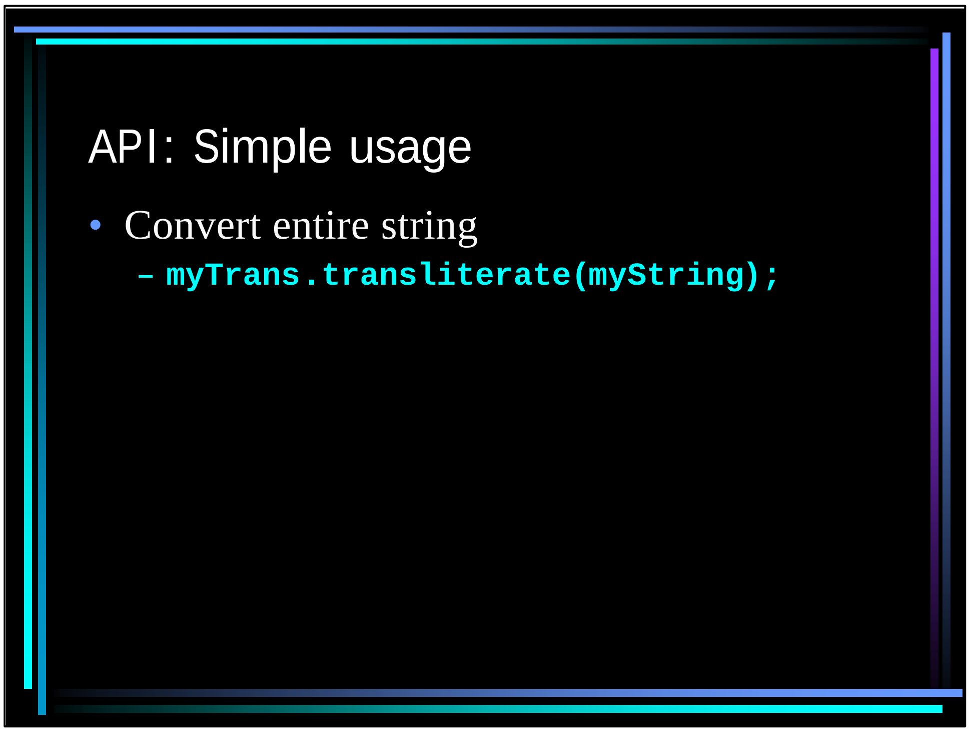
API: Creation

- Use an ID to create:

  -
    ```
    myTrans =
    Transliterator::createInstance("Latin
    -Greek");
    ```

Once you have an ID, you can create an instance of a transliterator.

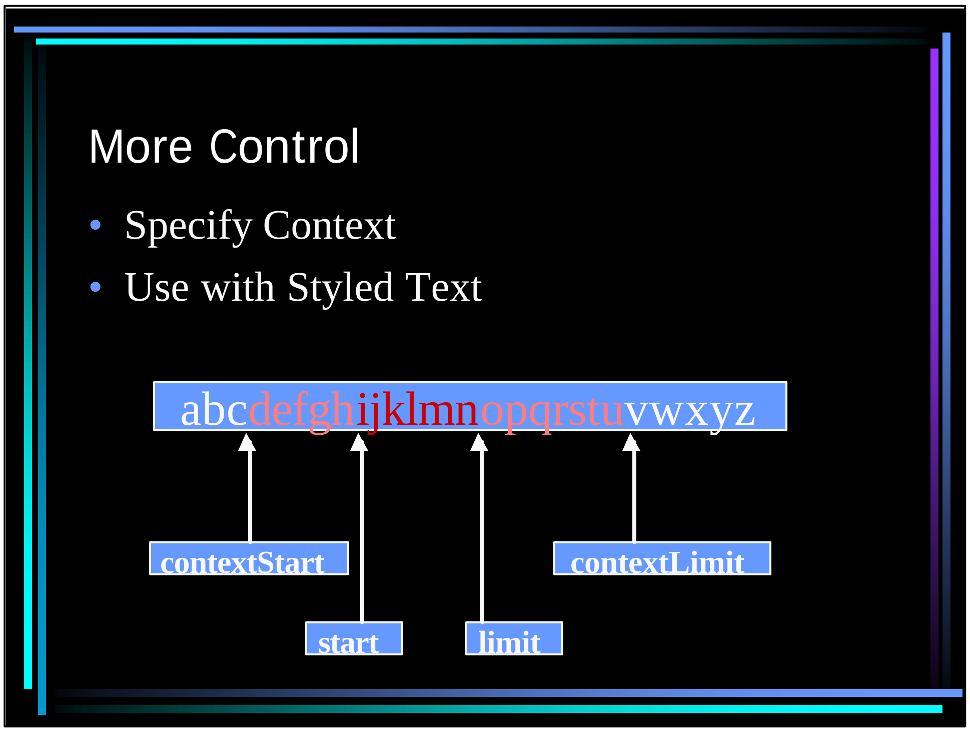
In the C API, this is an “open” call.



API: Simple usage

- Convert entire string
  - `myTrans.transliterate(myString);`

The simplest usage is to just convert an entire string.



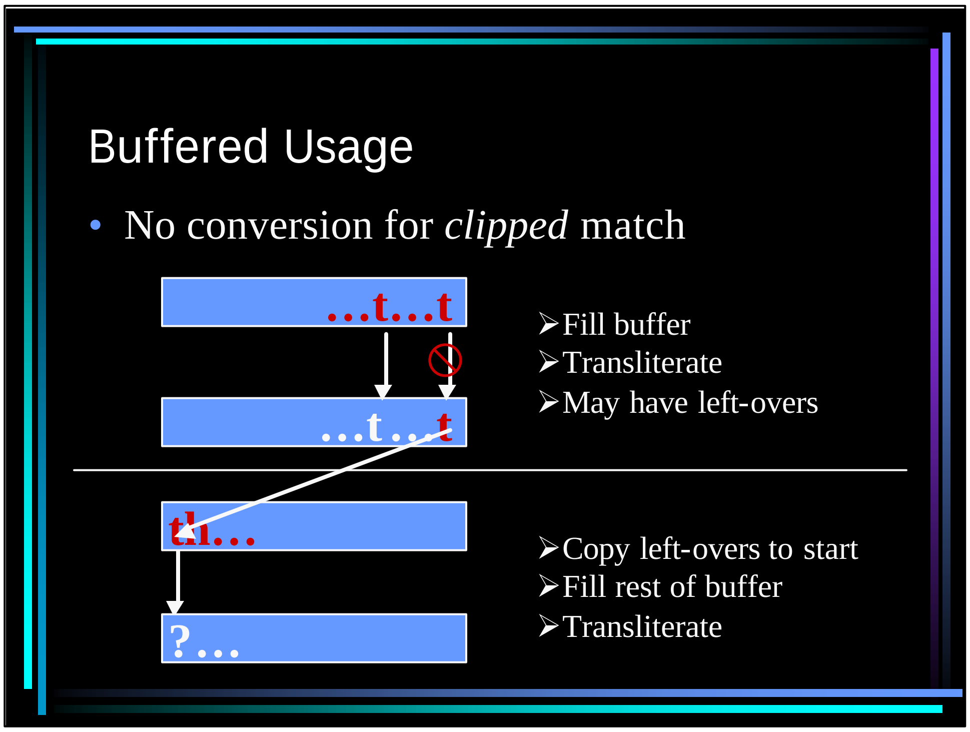
More Control

- Specify Context
- Use with Styled Text

abcdefghijklmnopqrstuvwxyz
contextStart
start
limit
contextLimit

More sophisticated APIs allow more complex text to be used with Transliterators, such as styled text. With these APIs, the input parameters supply information about the range of text that should be transliterated, plus the possibly larger range of text that can serve as context. The transliterator can take account of that context in performing its transformations.

Transliterators don't just work on plain text; they can also work on styled text. This is done through the Replaceable interface. This is an interface (or abstract class in C++) to text that handles a very few operations: essentially access to characters, plus replacement of a substring by another. By using this interface, replacement text can take on the same style as the text it is replacing, so that style information is not lost. With a replaceable interface to HTML or XML, even higher level structure can be preserved.



Buffered Usage

- No conversion for *clipped* match

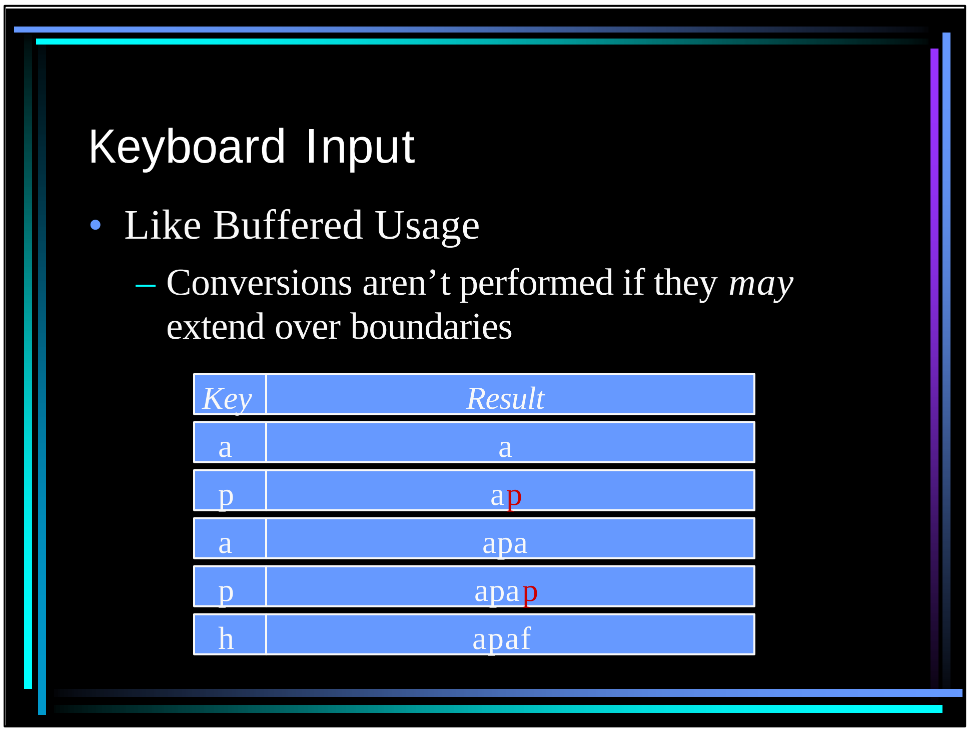
➢Fill buffer
➢Transliterate
➢May have left-overs

➢Copy left-overs to start
➢Fill rest of buffer
➢Transliterate

Transliterator objects are *stateless*; they retain no information between calls to **transliterate**(). This might seem to limit the complexity of the transliteration operation. In practice, subclasses perform complex transliterations by delaying the replacement of text until it is known that no other replacements are possible. In other words, although the Transliterator objects are stateless, the source text itself embodies all the needed information, and delayed operation allows arbitrary complexity.

The `complete` parameter indicates whether or not you are to consider the text up to limit to be complete or not. For example, for keyboard input this should normally be false. Only when the conversion is complete is that parameter set to true. For example, suppose that a transliterator converts "sh" to X, and "s" in other cases to Y. If the *complete* parameter is true, then a dangling "s" converts to Y; when the complete parameter is false, then the dangling "s" should not be converted, since there is more text to come.

In other words, a rule will have a *clipped* match if the rule matches up to the end of the buffer, but would require more characters past the end of the buffer for it to really match. When the *complete* parameter is not set, then any *clipped* match will stop the processing, because more characters could be coming.



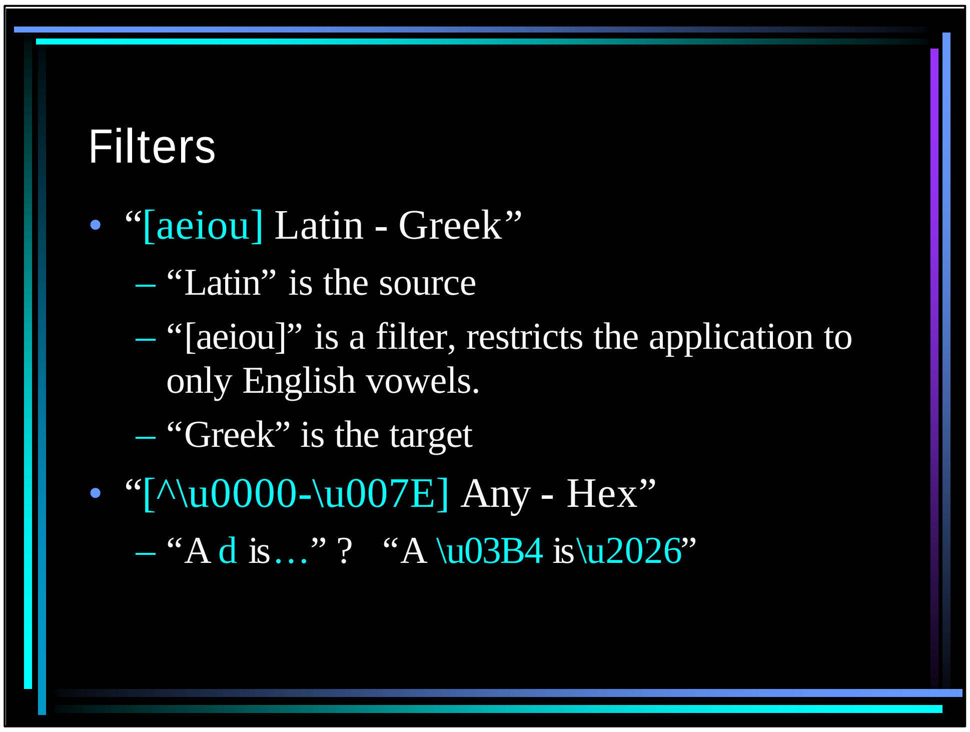
Keyboard Input

- Like Buffered Usage
  - Conversions aren't performed if they *may* extend over boundaries

Key	*Result*
a	a
p	ap
a	apa
p	apap
h	apaf

The buffering support is also used for keyboard input.

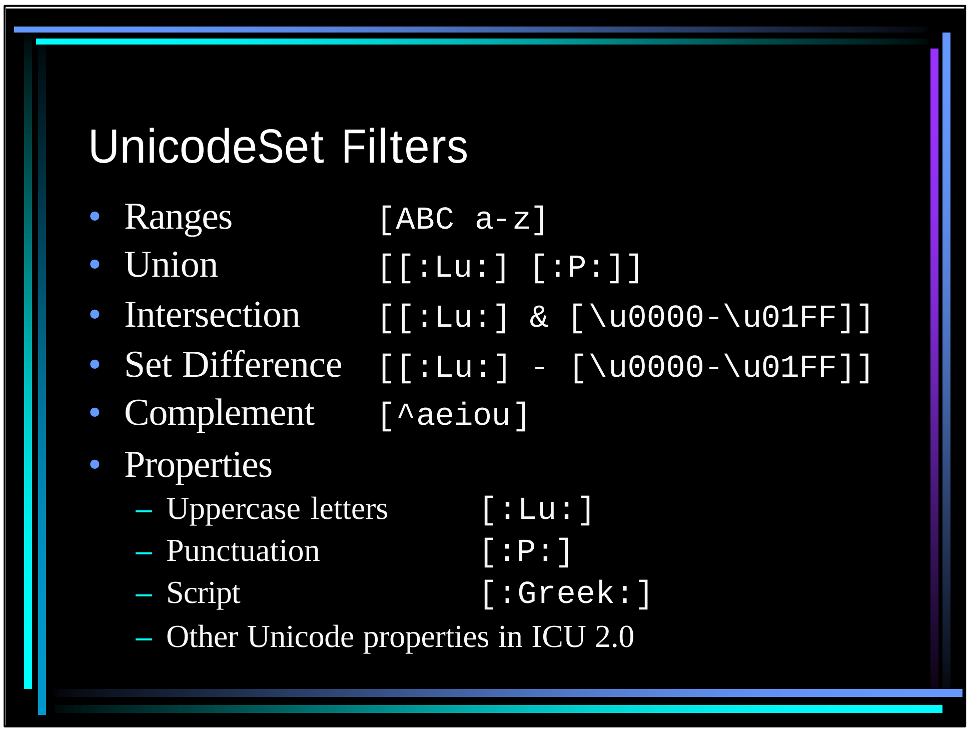
This shows what appears on the screen as the user types. The left column is the key typed by the user, while the right column shows what appears on the screen. Notice that the "p" does not get converted until the next letter is typed.



Filters

- “[aeiou] Latin - Greek”
  - “Latin” is the source
  - “[aeiou]” is a filter, restricts the application to only English vowels.
  - “Greek” is the target
- “[^\u0000-\u007E] Any - Hex”
  - “A d is…” ? “A \u03B4 is\u2026”

Filters can be used to restrict the application of a transliterator to a subset of Unicode characters. For example, we may only want to convert certain characters from Latin to Greek, or normalize letters from Devanagari, or transform non-ASCII characters into an escaped-hex representation.

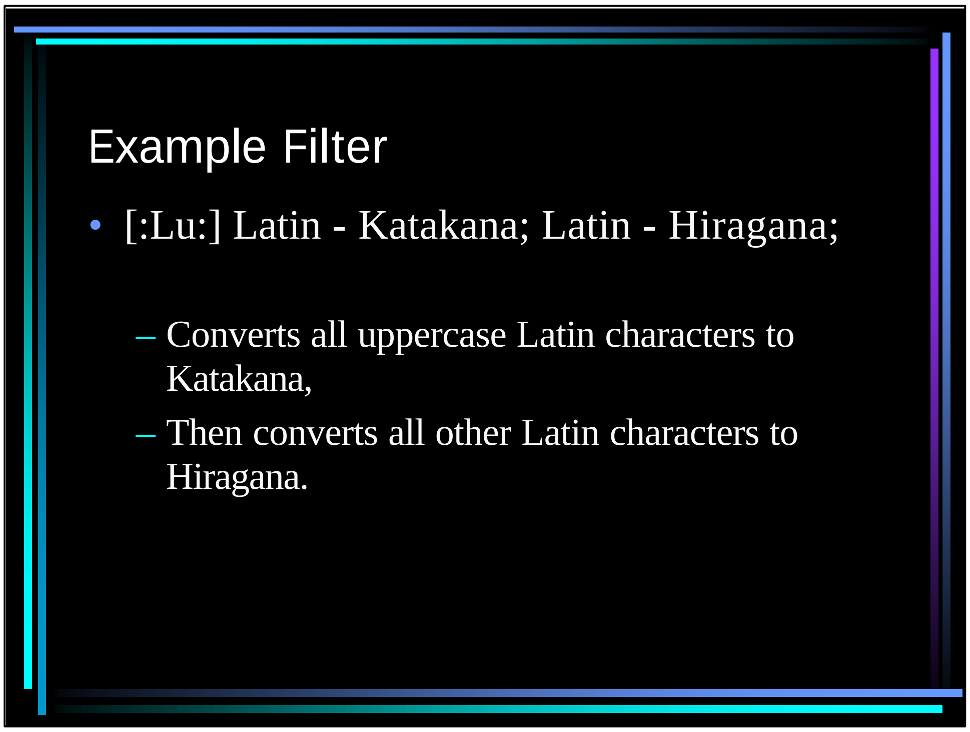


UnicodeSet Filters

• Ranges	`[ABC a-z]`
• Union	`[[:Lu:] [:P:]]`
• Intersection	`[[:Lu:] & [\u0000-\u01FF]]`
• Set Difference	`[[:Lu:] - [\u0000-\u01FF]]`
• Complement	`[^aeiou]`
• Properties	
– Uppercase letters	`[:Lu:]`
– Punctuation	`[:P:]`
– Script	`[:Greek:]`
– Other Unicode properties in ICU 2.0	

The filters use something called a UnicodeSet. This is similar to regular expression ranges. It allows ranges, boolean operations, and Unicode properties. These will be enhanced considerably in ICU 2.0.

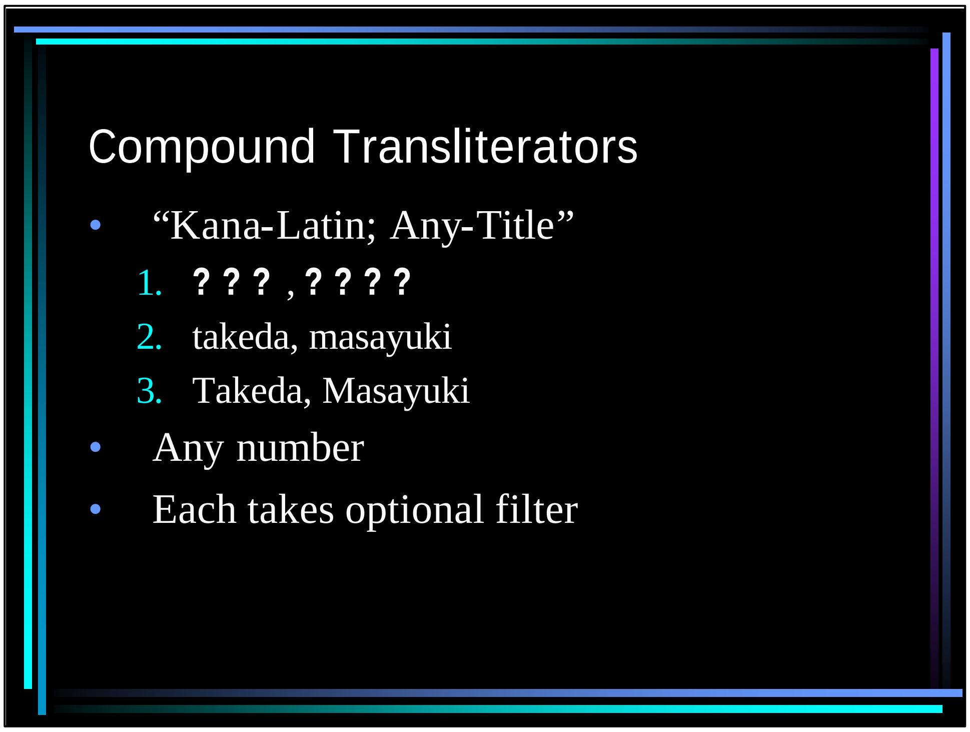
UnicodeSet is also used within rules, in building Transliterators. We will see that a bit later.



Example Filter

- [:Lu:] Latin - Katakana; Latin - Hiragana;
  - Converts all uppercase Latin characters to Katakana,
  - Then converts all other Latin characters to Hiragana.

A common transliteration for input is uppercase to katakana, lowercase to hiragana. You would do that with the example here.

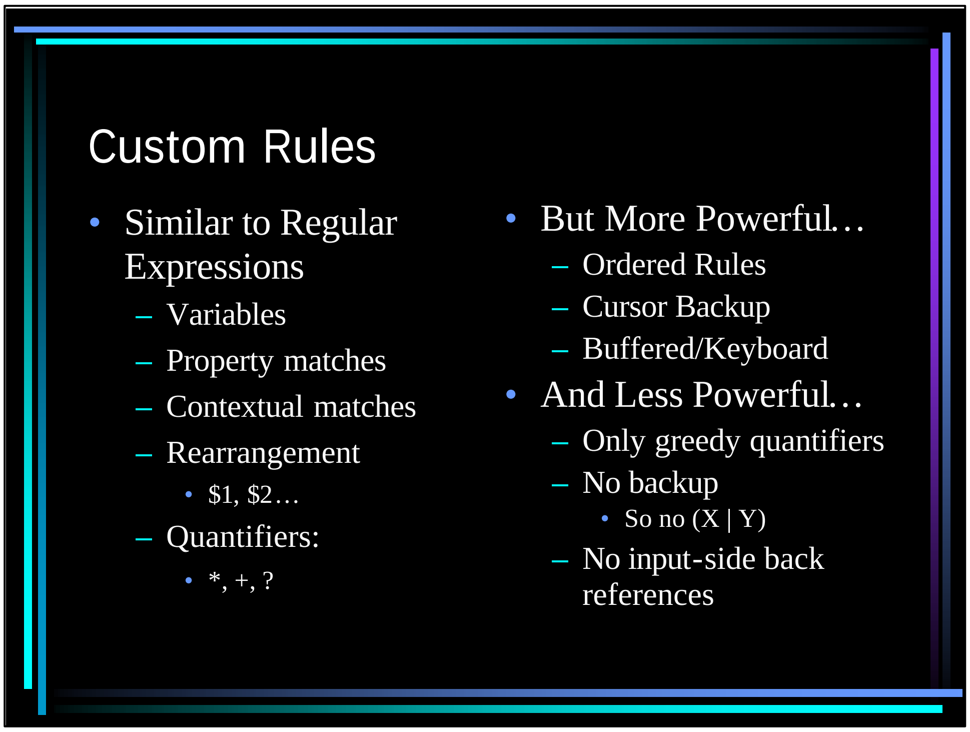


Compound Transliterators

- "Kana-Latin; Any-Title"
  1. ? ? ? , ? ? ? ?
  2. takeda, masayuki
  3. Takeda, Masayuki
- Any number
- Each takes optional filter

Transliterators can also be chained together. This is done by using a semicolon as a separator, and listing a number of Transliterator IDs. The text is processed as if each transliterator is invoked on it, one after another.

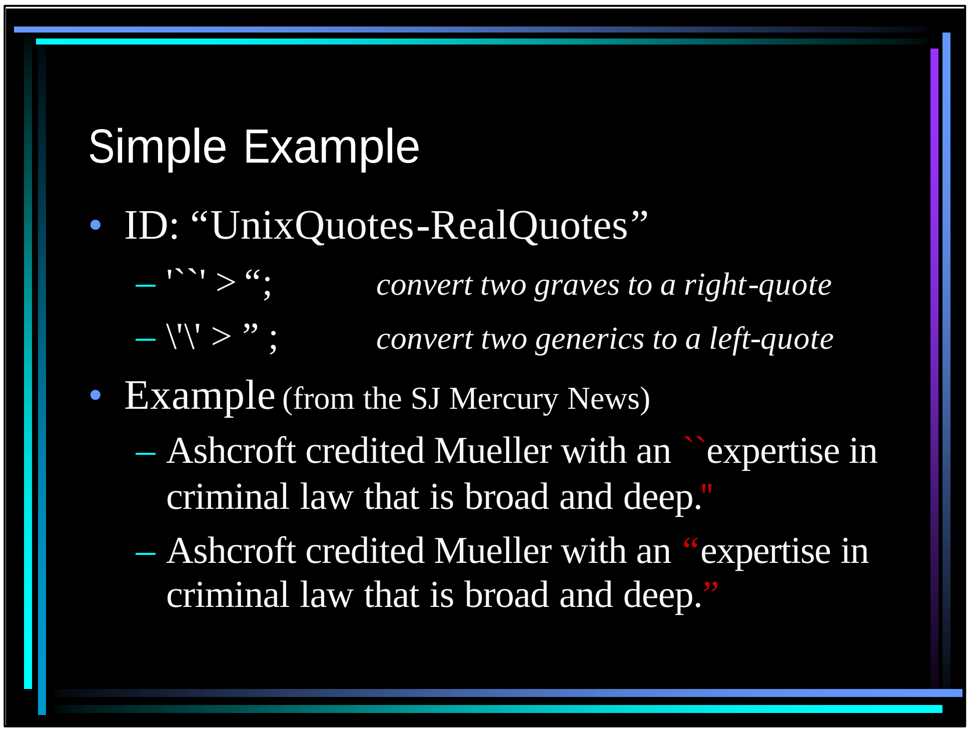
Any number of IDs can be used, and each ID can include a filter. In addition, a filter can be applied to the entire compound.



Custom Rules

- Similar to Regular Expressions
  - Variables
  - Property matches
  - Contextual matches
  - Rearrangement
    - $1, $2…
  - Quantifiers:
    - *, +, ?
- But More Powerful…
  - Ordered Rules
  - Cursor Backup
  - Buffered/Keyboard
- And Less Powerful…
  - Only greedy quantifiers
  - No backup
    - So no (X | Y)
  - No input-side back references

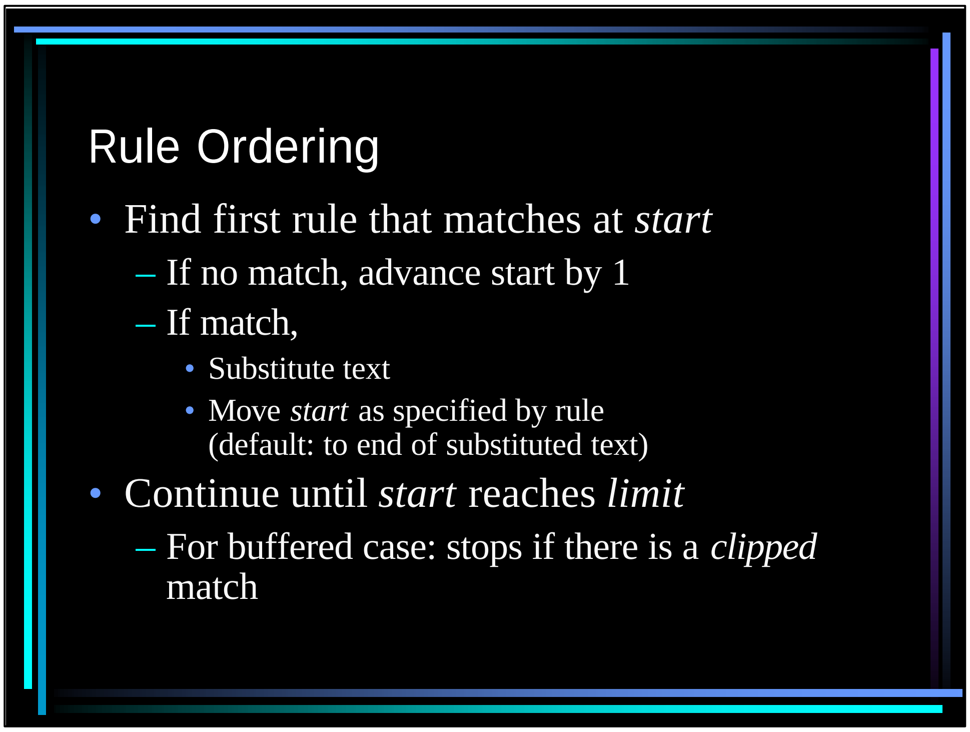
For the more adventurous, Transliterators can be constructed using rules. These rules are similar to regular expression matches, but not identical. They are not as powerful as full regular expressions in general, since their primary focus is different; on the other hand, some of their capabilities exceed what is found in normal regular expression engines.



Simple Example

- ID: “UnixQuotes-RealQuotes”
  - '``' > “; *convert two graves to a right-quote*
  - \'\' > ” ; *convert two generics to a left-quote*
- Example (from the SJ Mercury News)
  - Ashcroft credited Mueller with an ``expertise in criminal law that is broad and deep."
  - Ashcroft credited Mueller with an “expertise in criminal law that is broad and deep.”

Here is a simple example. The two rules you see here are used to convert Unix quote marks into real quote marks. Below is a sample of text from an article in the San Jose Mercury News online, with this transliterator applied.

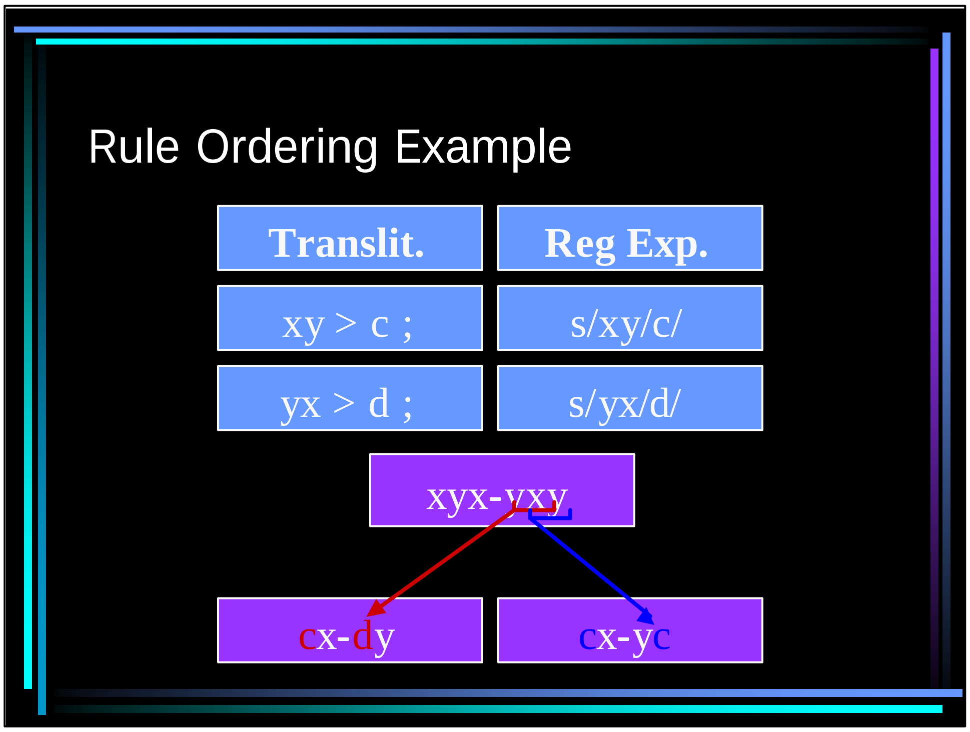


Rule Ordering

- Find first rule that matches at *start*
  - If no match, advance start by 1
  - If match,
    - Substitute text
    - Move *start* as specified by rule (default: to end of substituted text)
- Continue until *start* reaches *limit*
  - For buffered case: stops if there is a *clipped* match

Rules are applied in order, as described here. We will see the consequences on that on the next slide.

Note that in buffered input, where more characters could be coming, then the processing stops if a clipped match is reached. That is where a rule matches up to the end of the buffer, but needs more characters to complete the match.

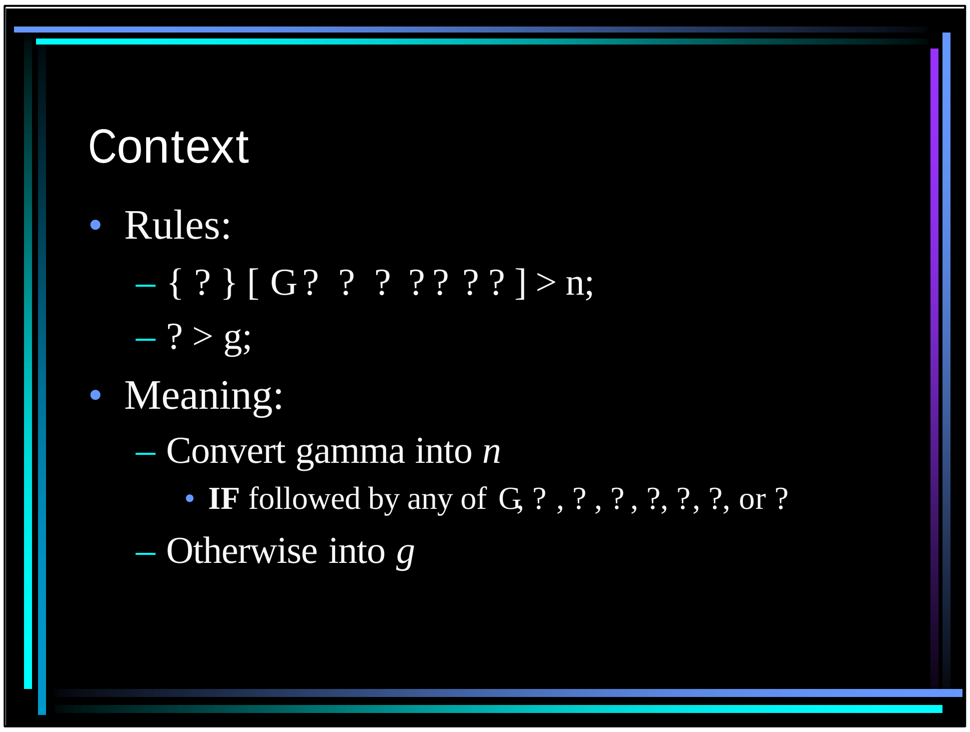


Rule Ordering Example

Translit.	Reg Exp.
xy > c ;	s/xy/c/
yx > d ;	s/yx/d/

Because of the ordering, the results of a transliteration is **not** the same as what you would get by simply applying the first rule to the whole text, then the next rule to the whole text, etc. Instead, the text at each point is converted according to the rule precedence.

The example on this slide illustrates this. The result of transliteration differs significantly from what would happen by simply applying multiple regular expressions. Since the transliterator processes each of its rules at each point, it catches the yx before the xy in the second case. Since each of the regular expressions is evaluated over the whole string, that isn't possible. Simply using multiple regular expressions can't account for the interaction and ordering of characters and rules.

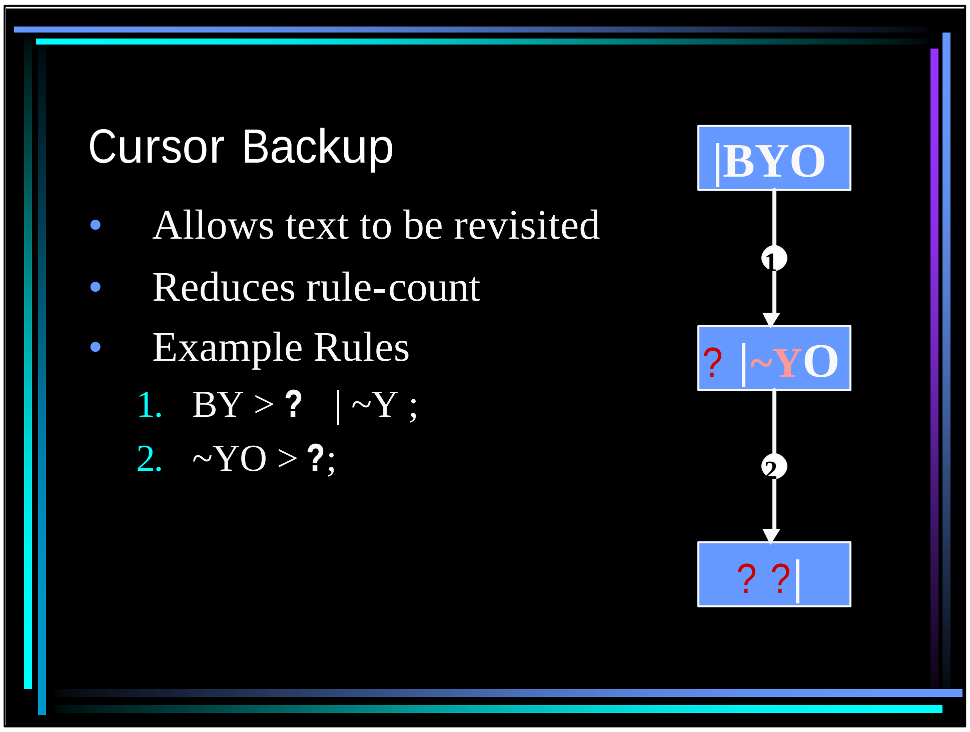


Context

- Rules:
  - { ? } [G ? ? ? ? ? ? ?] > n;
  - ? > g;
- Meaning:
  - Convert gamma into *n*
    - **IF** followed by any of G, ? , ? , ? , ?, ?, ?, or ?
  - Otherwise into *g*

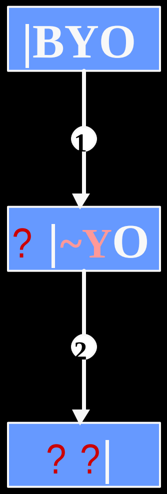
Context can be used to have the results of a transformation be different depending on the characters before or after.

Note: The context itself (e.g. G, …, ?) is unaffected by the replacement; *only* the text between the curly braces is changed.



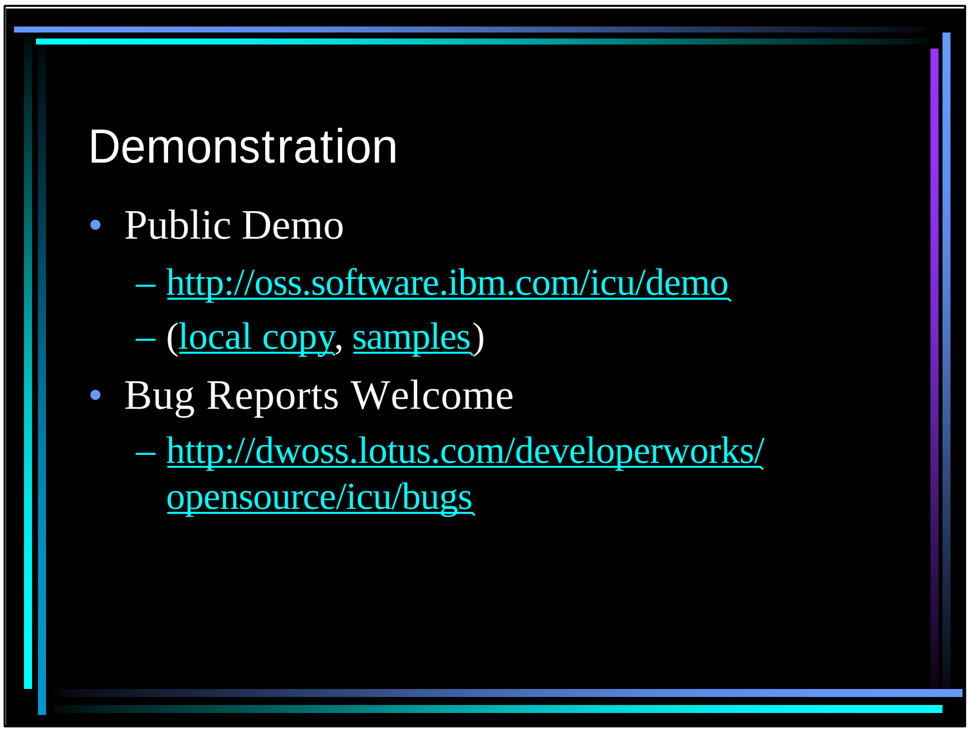
Cursor Backup

- Allows text to be revisited
- Reduces rule-count
- Example Rules
  1. BY > **?** | ~Y ;
  2. ~YO > **?**;



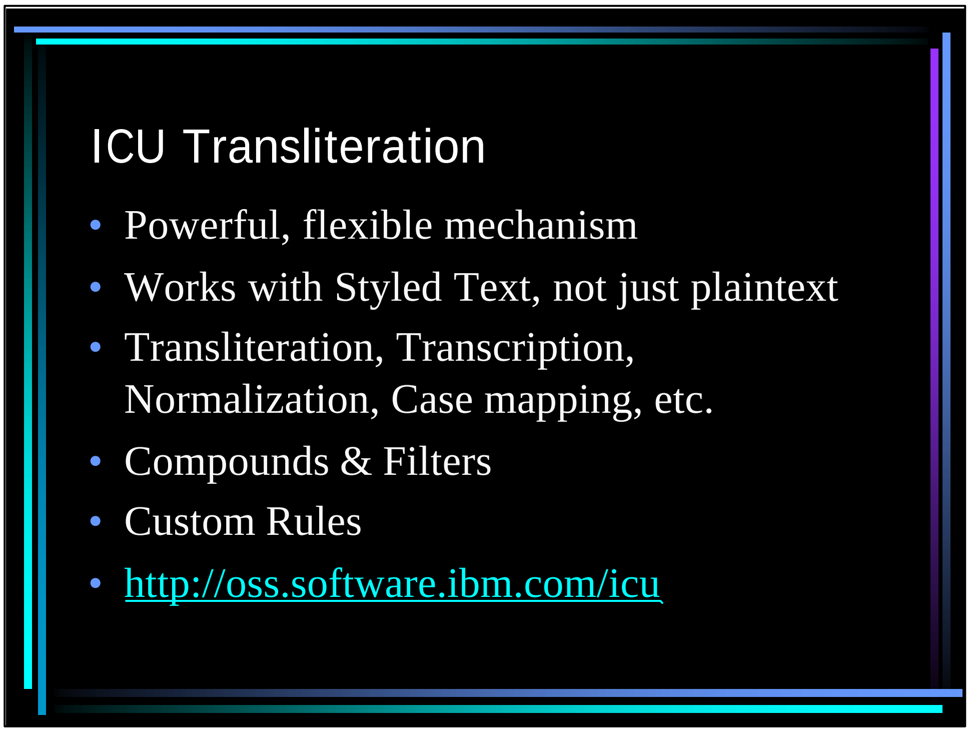
Cursor backup is a very powerful ability. It allows rules to be structured so that they do a replacement, then backup to allow other rules to be invoked on the transformed text.

It can drastically reduce rule count. In this example, instead of having rules for the combinations of *consonant* + *Y* + *vowel* (for the relevant vowels and consonants), you can just have rules for *consonant* + *Y*, and *Y* + *vowel*. This reduces the number of rules from *consonant_count* ***times*** *vowel_count* to *consonant_count* ***plus*** *vowel_count*.



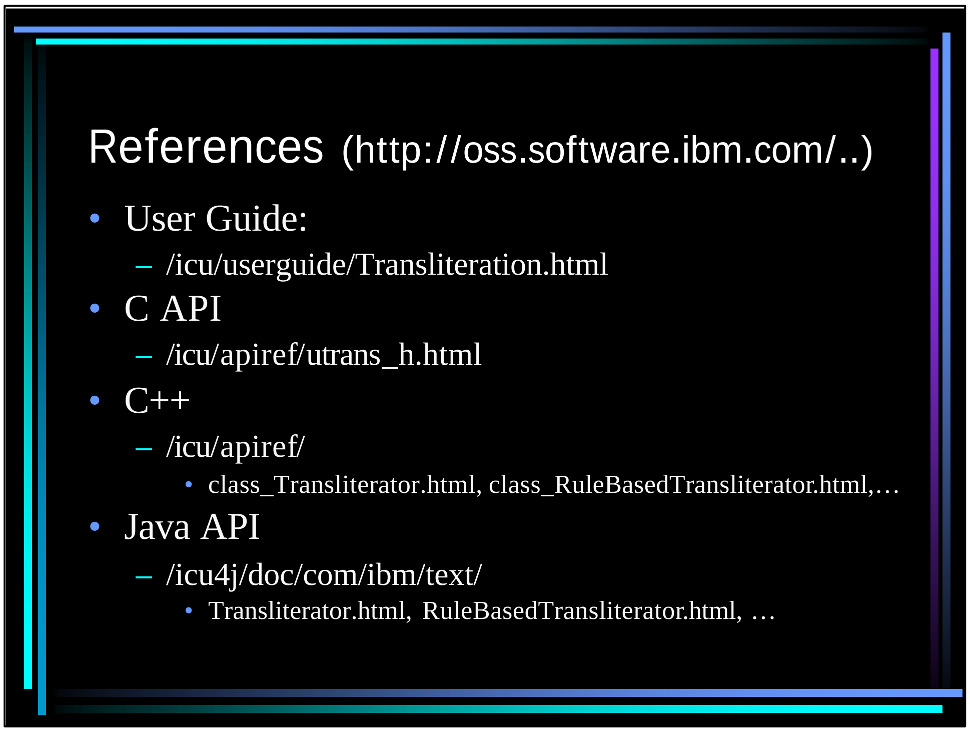
Demonstration

- Public Demo
  - http://oss.software.ibm.com/icu/demo
  - (local copy, samples)
- Bug Reports Welcome
  - http://dwoss.lotus.com/developerworks/opensource/icu/bugs



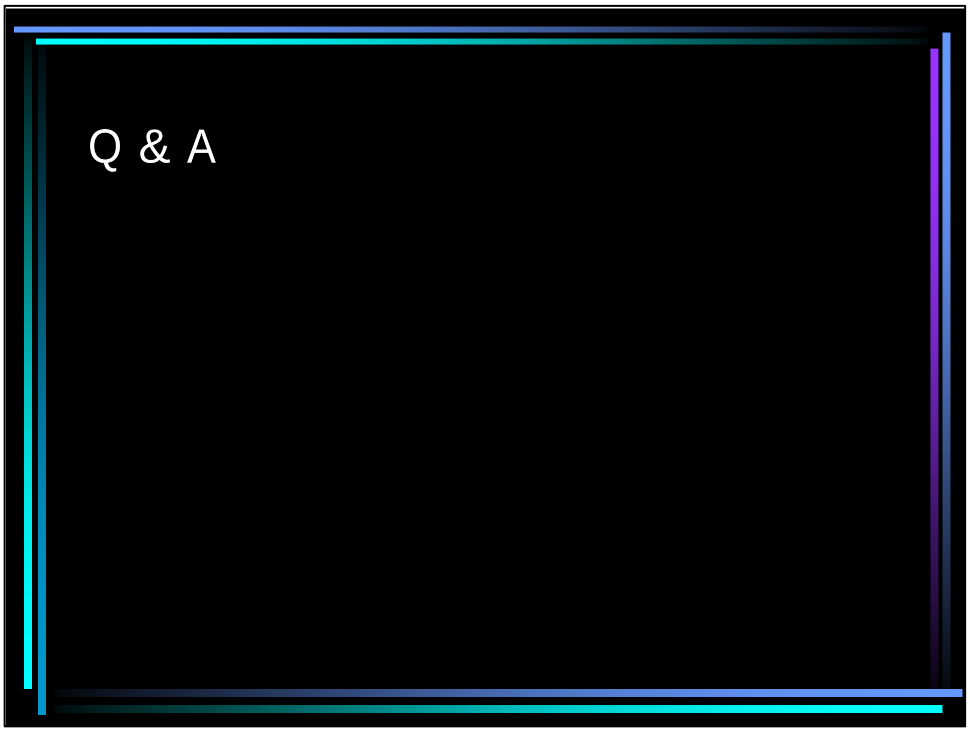
ICU Transliteration

- Powerful, flexible mechanism
- Works with Styled Text, not just plaintext
- Transliteration, Transcription, Normalization, Case mapping, etc.
- Compounds & Filters
- Custom Rules
- http://oss.software.ibm.com/icu

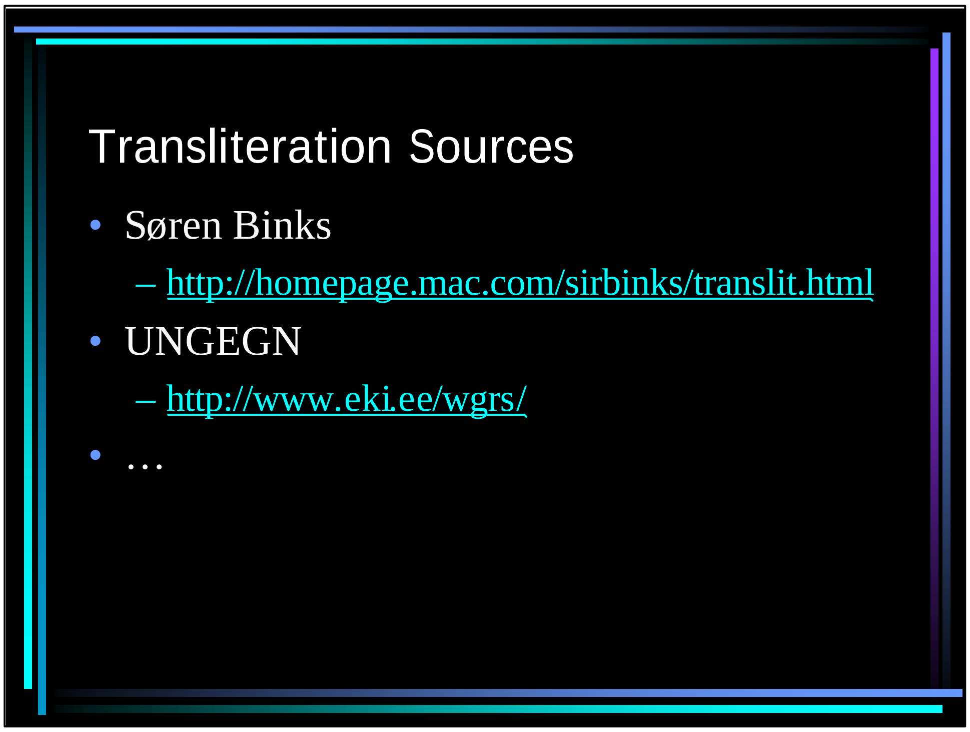


References (http://oss.software.ibm.com/..)

- User Guide:
  - /icu/userguide/Transliteration.html
- C API
  - /icu/apiref/utrans_h.html
- C++
  - /icu/apiref/
    - class_Transliterator.html, class_RuleBasedTransliterator.html,…
- Java API
  - /icu4j/doc/com/ibm/text/
    - Transliterator.html, RuleBasedTransliterator.html, …



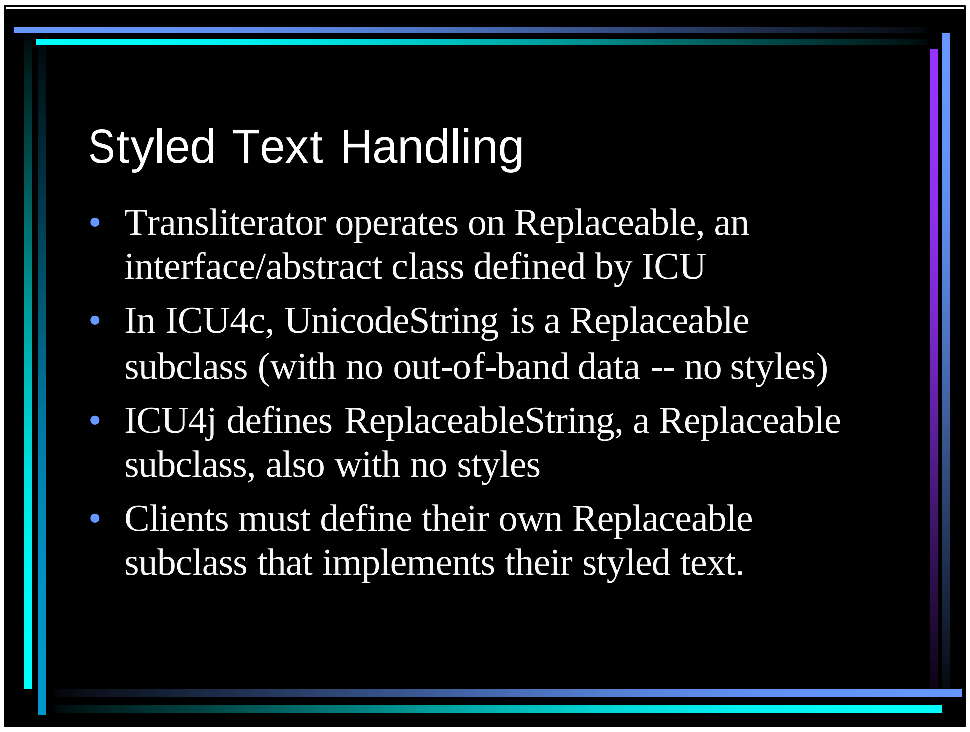
Q & A



Transliteration Sources

- Søren Binks
  - http://homepage.mac.com/sirbinks/translit.html
- UNGEGN
  - http://www.eki.ee/wgrs/
- …

Backup Slides



Styled Text Handling

- Transliterator operates on Replaceable, an interface/abstract class defined by ICU
- In ICU4c, UnicodeString is a Replaceable subclass (with no out-of-band data -- no styles)
- ICU4j defines ReplaceableString, a Replaceable subclass, also with no styles
- Clients must define their own Replaceable subclass that implements their styled text.