Multi-Language Character Sets

Bob Balaban, President
Looseleaf Software, Inc.
http://www.looseleaf.net

What They Are, How to Use Them
Agenda

- Speaker introduction
- Background: terminology
- History of electronic character sets
- The need for multi-language character sets
- LMBCS - Lotus MultiByte Character Set
  - How it works
  - Why no one else uses it
Agenda - 2

- Unicode
  - How it works
  - Why everyone uses it
- Software for multi-language character sets
  - LMBCS
  - Unicode
  - Java, C, LotusScript
- Web applications and MLCS
Speaker Introduction

- Master's degree in Chinese History/Anthropology
- Professional software developer since 1978
- Engineer at Lotus Development Corp. 1987-97
- Reviewer of original LMBCS specification (1988)
Speaker - 2

- Team leader for adding LMBCS/Kanji support to 123/G (os/2)
- Developer on Notes/Domino 1993-97
  - Author of LotusScript "back-end classes"
  - Author of Java APIs for back-end classes
- Founded Looseleaf Software, Inc., 1997
  - Custom development, training, architecture/design, consulting
  - Notes/Domino, J2EE, Groove
Terminology

- Textual characters on a computer are really numbers
  - Like everything else
- The numbers, when representing text, are mapped to symbols on a screen
  - or on paper
- The number of symbols you can display depends on the number of bits assigned to the "character set", or "code space"
Terminology - 2

- One "character" is a "code point"
  - Just a number
- The screen symbol is the "glyph"
- The "font" is the style in which the glyphs are displayed
  - serif, sans serif, etc.
- The mapping of numbers to code points defines a "character set", or "code page"
What is a "native character set"?
The default character set used by the operating system
Many OS's can handle multiple character sets (Ascii, CP850, etc.)
Many OS versions depend on a "locale" specification
▶ E.g., CP932 in Japan, CP850 in NAmerica
Terminology - 4

- All mappings are essentially arbitrary
- But some have been agreed upon as standards

A = 1
B = 2
...
Z = 26

A possible mapping

A = 17
B = 18
....
K = 99
....
Z = 102

Another possible mapping
What are "control codes"?
- Generally, non-printing
  - No visible representation, no glyph
- Invented to manage early printer/terminal devices, and communication protocols
  - Ack/Nak
  - Carriage return, newline, tab, formfeed
  - Bell
Background: History of electronic character sets

- Original character set was Binary Coded Decimal (BCD)
  - IBM
  - 64 characters, 6 bits
  - Known later as "SIXBIT ASCII"
- Developed for IBM punch cards
- Upper case letters, digits, punctuation
When IBM created the System 360, they extended BCD

- 1965
- EBCDIC
- Extended Binary Coded Decimal Interchange Code
- 256 code points
- 64 control codes
- Upper and lower case!
- Some slots were empty, customers complained about wasted memory!
IBM EBCDIC, ca. 1965
EBCDIC and Beyond

- The most popular character set through the 1970s
  - Until the PC was born
- IBM used 8 bits because the word size on S360 was 32 bits
- IBM machines (other than PCs) still use EBCDIC
  - There are a few variants, some code points may be different
Beyond - 2

- Note the strange ordering, where the alphabetic sequences are interrupted
- Makes for lots of fun when writing string comparison/sorting code!
  - Or converting between upper and lower case
- Note, no accented characters
  - Or non-Latin glyphs
The Rise of ASCII

- American Standard Code for Information Interchange
  - Or something like that
- Network bandwidth was very expensive, people wanted to save money and utilize 7-bit channels
- And wanted to conserve on memory, eliminate all that "wasted" space in EBCDIC
ASCII - 2

- Adopted by PC OSs (DOS), and by Unix systems
- 32 control codes, punctuation, digits, upper and lower alphabetics (0 is special)
  - This time letters were in a rational sequence (caps first)!
  - You could add/subtract 32 to go from upper to lower and back
- Still no accents or "foreign" glyphs
### ASCII

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Chrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>NUL (null)</td>
<td>32</td>
<td>20</td>
<td>040</td>
<td>#32; Space</td>
<td>64</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>SOH (start of heading)</td>
<td>33</td>
<td>21</td>
<td>041</td>
<td>#33;</td>
<td>65</td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td>STX (start of text)</td>
<td>34</td>
<td>22</td>
<td>042</td>
<td>#34; &quot;</td>
<td>66</td>
</tr>
<tr>
<td>3</td>
<td>003</td>
<td>ETX (end of text)</td>
<td>35</td>
<td>23</td>
<td>043</td>
<td>#35;</td>
<td>67</td>
</tr>
<tr>
<td>4</td>
<td>004</td>
<td>EOT (end of transmission)</td>
<td>36</td>
<td>24</td>
<td>044</td>
<td>#36;</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>ENQ (enquiry)</td>
<td>37</td>
<td>25</td>
<td>045</td>
<td>#37;</td>
<td>69</td>
</tr>
<tr>
<td>6</td>
<td>006</td>
<td>ACK (acknowledge)</td>
<td>38</td>
<td>26</td>
<td>046</td>
<td>#38;</td>
<td>70</td>
</tr>
<tr>
<td>7</td>
<td>007</td>
<td>BEL (bell)</td>
<td>39</td>
<td>27</td>
<td>047</td>
<td>#39;</td>
<td>71</td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>BS (backspace)</td>
<td>40</td>
<td>28</td>
<td>050</td>
<td>#40; (</td>
<td>72</td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td>HT (horizontal tab)</td>
<td>41</td>
<td>29</td>
<td>051</td>
<td>#41; )</td>
<td>73</td>
</tr>
<tr>
<td>10</td>
<td>A12</td>
<td>LF (NL line feed, new line)</td>
<td>42</td>
<td>2A</td>
<td>052</td>
<td>#42; *</td>
<td>74</td>
</tr>
<tr>
<td>11</td>
<td>B13</td>
<td>VT (vertical tab)</td>
<td>43</td>
<td>2B</td>
<td>053</td>
<td>#43; +</td>
<td>75</td>
</tr>
<tr>
<td>12</td>
<td>C14</td>
<td>FF (NP form feed, new page)</td>
<td>44</td>
<td>2C</td>
<td>054</td>
<td>#44; ,</td>
<td>76</td>
</tr>
<tr>
<td>13</td>
<td>D15</td>
<td>CR (carriage return)</td>
<td>45</td>
<td>2D</td>
<td>055</td>
<td>#45; -</td>
<td>77</td>
</tr>
<tr>
<td>14</td>
<td>E16</td>
<td>SO (shift out)</td>
<td>46</td>
<td>2E</td>
<td>056</td>
<td>#46; .</td>
<td>78</td>
</tr>
<tr>
<td>15</td>
<td>F17</td>
<td>SI (shift in)</td>
<td>47</td>
<td>2F</td>
<td>057</td>
<td>#47; /</td>
<td>79</td>
</tr>
<tr>
<td>16</td>
<td>020</td>
<td>DLE (data link escape)</td>
<td>48</td>
<td>30</td>
<td>060</td>
<td>#48; 0</td>
<td>80</td>
</tr>
<tr>
<td>17</td>
<td>021</td>
<td>DC1 (device control 1)</td>
<td>49</td>
<td>31</td>
<td>061</td>
<td>#49; 1</td>
<td>81</td>
</tr>
<tr>
<td>18</td>
<td>022</td>
<td>DC2 (device control 2)</td>
<td>50</td>
<td>32</td>
<td>062</td>
<td>#50; 2</td>
<td>82</td>
</tr>
<tr>
<td>19</td>
<td>023</td>
<td>DC3 (device control 3)</td>
<td>51</td>
<td>33</td>
<td>063</td>
<td>#51; 3</td>
<td>83</td>
</tr>
<tr>
<td>20</td>
<td>024</td>
<td>DC4 (device control 4)</td>
<td>52</td>
<td>34</td>
<td>064</td>
<td>#52; 4</td>
<td>84</td>
</tr>
<tr>
<td>21</td>
<td>025</td>
<td>NAK (negative acknowledge)</td>
<td>53</td>
<td>35</td>
<td>065</td>
<td>#53; 5</td>
<td>85</td>
</tr>
<tr>
<td>22</td>
<td>026</td>
<td>SYN (synchronous idle)</td>
<td>54</td>
<td>36</td>
<td>066</td>
<td>#54; 6</td>
<td>86</td>
</tr>
<tr>
<td>23</td>
<td>027</td>
<td>ETB (end of transmission block)</td>
<td>55</td>
<td>37</td>
<td>067</td>
<td>#55; 7</td>
<td>87</td>
</tr>
<tr>
<td>24</td>
<td>030</td>
<td>CAN (cancel)</td>
<td>56</td>
<td>38</td>
<td>070</td>
<td>#56; 8</td>
<td>88</td>
</tr>
<tr>
<td>25</td>
<td>031</td>
<td>EM (end of medium)</td>
<td>57</td>
<td>39</td>
<td>071</td>
<td>#57; 9</td>
<td>89</td>
</tr>
<tr>
<td>26</td>
<td>032</td>
<td>SUB (substitute)</td>
<td>58</td>
<td>3A</td>
<td>072</td>
<td>#58; :</td>
<td>90</td>
</tr>
<tr>
<td>27</td>
<td>033</td>
<td>ESC (escape)</td>
<td>59</td>
<td>3B</td>
<td>073</td>
<td>#59; ;</td>
<td>91</td>
</tr>
<tr>
<td>28</td>
<td>034</td>
<td>FS (file separator)</td>
<td>60</td>
<td>3C</td>
<td>074</td>
<td>#60; &lt;</td>
<td>92</td>
</tr>
<tr>
<td>29</td>
<td>035</td>
<td>GS (group separator)</td>
<td>61</td>
<td>3D</td>
<td>075</td>
<td>#61; =</td>
<td>93</td>
</tr>
<tr>
<td>30</td>
<td>036</td>
<td>RS (record separator)</td>
<td>62</td>
<td>3E</td>
<td>076</td>
<td>#62; &gt;</td>
<td>94</td>
</tr>
<tr>
<td>31</td>
<td>037</td>
<td>US (unit separator)</td>
<td>63</td>
<td>3F</td>
<td>077</td>
<td>#63; ?</td>
<td>95</td>
</tr>
</tbody>
</table>
The need for multi-language character sets

- Eventually, manufacturers realized that they could maybe sell more software and hardware outside NAmerica if they supported languages other than English.
- But how to deal with those "foreign" symbols?
  - While still allowing for interoperability with Latin text?
Multi-language - 2

- Easy! Go back to 8-bit "codepages"!
  - Use the "upper half" of the 8-bit space (another 128 characters beyond ASCII)
- Each computer sold in a "foreign" country would carry a different code point mapping (a "Codepage")
- Again, this system was invented by IBM
  - And done very systematically
  - Each codepage got a numeric designation
  - And a glyph mapping chart in a book
**Codepages - continued**

- ASCII was always the "lower half" of the codepage
  - So everyone in the world could benefit from using English
- Windows ASCII (again, 8 bits) is cp437
- IBM "international English" is cp850
- Japanese is cp932 (more on this later)
- PRC Simplified Chinese: cp936
- Korean: 949
- Cyrillic: cp1251
- Latin1: cp1252
- etc.
### Codepage 850

#### 850 MS-DOS Latin 1

<table>
<thead>
<tr>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
<th>D0</th>
<th>E0</th>
<th>F0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>@</td>
<td>P</td>
<td>`</td>
<td>p</td>
<td>Ç</td>
<td>É</td>
<td>à</td>
<td>å</td>
<td>L</td>
<td>ò</td>
<td>Ó</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
<td>ü</td>
<td>æ</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&quot;</td>
<td>2</td>
<td>BR</td>
<td>b</td>
<td>r</td>
<td>é</td>
<td>Æ</td>
<td>Ó</td>
<td>T</td>
<td>Ë</td>
<td>Ø</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
<td>á</td>
<td>ò</td>
<td>Õ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>t</td>
<td>õ</td>
<td>Ñ</td>
<td>Í</td>
<td>Ð</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>u</td>
<td>à</td>
<td>Ò</td>
<td>å</td>
<td>Ó</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>v</td>
<td>á</td>
<td>å</td>
<td>Í</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>'</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
<td>ç</td>
<td>û</td>
<td>À</td>
<td>Í</td>
<td>Ñ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>x</td>
<td>e</td>
<td>y</td>
<td>i</td>
<td>ç</td>
<td>Í</td>
<td>Ñ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>y</td>
<td>é</td>
<td>Ò</td>
<td>ï</td>
<td>Ó</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>*</td>
<td>J</td>
<td>Z</td>
<td>i</td>
<td>z</td>
<td>e</td>
<td>Ù</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>+</td>
<td>K</td>
<td>{</td>
<td>i</td>
<td>ø</td>
<td>½</td>
<td>œ</td>
<td>ÿ</td>
<td>Ù</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>,</td>
<td>L</td>
<td>l</td>
<td>i</td>
<td>E</td>
<td>ñ</td>
<td>ñ</td>
<td>ñ</td>
<td>ñ</td>
<td>ñ</td>
<td>ñ</td>
<td>ñ</td>
<td>ñ</td>
</tr>
<tr>
<td>D</td>
<td>-</td>
<td>M</td>
<td>}</td>
<td>i</td>
<td>Ø</td>
<td>ç</td>
<td>ç</td>
<td>ç</td>
<td>ç</td>
<td>ç</td>
<td>ç</td>
<td>ç</td>
<td>ç</td>
</tr>
<tr>
<td>E</td>
<td>.</td>
<td>N</td>
<td>n</td>
<td>Å</td>
<td>×</td>
<td>¥</td>
<td>û</td>
<td>ÿ</td>
<td>ÿ</td>
<td>ÿ</td>
<td>ÿ</td>
<td>ÿ</td>
<td>ÿ</td>
</tr>
<tr>
<td>F</td>
<td>/</td>
<td>O</td>
<td>o</td>
<td>Å</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright 2001, Looseleaf Software, Inc.
All rights reserved
But.... Japanese?

- What if you can't fit a culture's "character set" into 256 slots?
- Asian languages, for example:
  - Thai: 40+ alphabetic/phonetic symbols
  - Chinese: 5000 common ideographs, 60,000+ total, TWO different systems!
  - Japanese: same (almost) ideographs as Chinese (trad.), PLUS 1 alphabetic series (Katakana), PLUS 1 phonetic (Hiragana), PLUS double-wide latin letters (so they line up with Kanji)!
Japanese - 2

- But a codepage only has an 8-bit namespace!!?
- So, reserve a sequence (or 2, or 3) in the upper half of the table as "pointers" to another 8-bit space
- As in CP932 (Japanese)
## Codepage 932

### Extract for Lead Byte E9-EA

<table>
<thead>
<tr>
<th>X9</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>F9</td>
<td>E9</td>
<td>D9</td>
<td>C9</td>
<td>B9</td>
<td>A9</td>
<td>99</td>
<td>89</td>
<td>79</td>
<td>69</td>
<td>59</td>
<td>49</td>
<td>39</td>
<td>29</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>F8</td>
<td>E8</td>
<td>D8</td>
<td>C8</td>
<td>B8</td>
<td>A8</td>
<td>98</td>
<td>88</td>
<td>78</td>
<td>68</td>
<td>58</td>
<td>48</td>
<td>38</td>
<td>28</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>F7</td>
<td>E7</td>
<td>D7</td>
<td>C7</td>
<td>B7</td>
<td>A7</td>
<td>97</td>
<td>87</td>
<td>77</td>
<td>67</td>
<td>57</td>
<td>47</td>
<td>37</td>
<td>27</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>F6</td>
<td>E6</td>
<td>D6</td>
<td>C6</td>
<td>B6</td>
<td>A6</td>
<td>96</td>
<td>86</td>
<td>76</td>
<td>66</td>
<td>56</td>
<td>46</td>
<td>36</td>
<td>26</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>F5</td>
<td>E5</td>
<td>D5</td>
<td>C5</td>
<td>B5</td>
<td>A5</td>
<td>95</td>
<td>85</td>
<td>75</td>
<td>65</td>
<td>55</td>
<td>45</td>
<td>35</td>
<td>25</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>F4</td>
<td>E4</td>
<td>D4</td>
<td>C4</td>
<td>B4</td>
<td>A4</td>
<td>94</td>
<td>84</td>
<td>74</td>
<td>64</td>
<td>54</td>
<td>44</td>
<td>34</td>
<td>24</td>
<td>13</td>
</tr>
</tbody>
</table>

**Copyright 2001, Looseleaf Software, Inc. All rights reserved**
This gives us an effective 16-bit space
  ▶ Almost, sort of
  ▶ It's really a partially-chained space
But requires that we allow for 2-byte (and later, even 3-byte) sequences to describe one "character"
You'd better know which CodePage you're dealing with for every string
BUT: still no way to combine multi-CP values in one string

Unless you start inventing special "opcodes" to switch codepages in the middle of a byte stream

No one thought that was a good idea
  ▶ So that particular problem was ignored for a while
LMBCS - Lotus MultiByte Character Set

- Lotus invented a new scheme as part of the work done for 123 Release 3
  - The first Lotus spreadsheet coded in C
  - Ran on both DOS and OS/2 (NOT Presentation Manager, which didn't exist yet)
  - Shipped in March, 1989
LMBCS, continued

- Essentially also a "lead-byte" system, taking advantage of the fact that the ASCII codepoints 01 through 1F are non-printing characters
  - 00 is still special, particularly for C strings
- So, let's assign every glyph to a "code group", every code group has a unique identifier, a value between 1 and 1F
- Each code group corresponds (approximately) to a CodePage
LMBCS, continued

- Code group 1 is Latin1
  - Very close to IBM CP850
- Each code group is either single byte, or multi byte
- Each code group's "lower half" is ASCII
- Code group 16 (Japanese) is multi byte, with designated "lead bytes", very similar to CP932
- And so on...
LMBCS, continued

- The big LMBCS innovation:
  - Every character in "canonical" LMBCS is preceded by its code group identifier

"ABC" in hex format:
01-41-01-42-01-43-0
LMBCS, continued

- Characters in multi-byte code groups could therefore be 2, 3, or 4 bytes long
- 0 still used (in C) as a string terminator
- Guaranteed to be no embedded 0s in a string

Example of 2-byte Japanese character:
10-<lead byte>--<2nd byte>--0
LMBCS, continued

- But this is obviously flawed!
- All strings double in size!
- So we apply an "optimization"
- Each application file comes with a "default code group" designation
- Any character where the code group value is omitted is assumed to be in the default code group
- We detect code group identifiers because they are non-printing characters in a specified range
- Any character in a non-default code group must have its code group value specified
LMBCS, continued

- This is known as "compressed", or "optimized" LMBCS

"ABC" in default code group 01:
41-42-43-0

"ABC" in default code group 16:
01-41-01-42-01-43-0

Hiragana small o in code group 16:
82-A7-0

Hiragana small o in code group 1:
10-82-A7-0
Special Topics

- What about diacriticals?
- What about BIDI (bi-directional) character sets?
- What about "ligatures"?
- How to handle the huge Chinese/Japanese/Korean character namespaces?
- These are issues that all multi-language systems must handle properly
- (There are even more issues, but these are the common ones)
Diacriticals

- We have a choice for dealing with accent marks
  - Create a code point and glyph for every combination

a á à

é è
Diacriticals - 2

- Or,
  - Invent the "non-spacing character"
- The "base" character has its own code point, each accent has its own
- Everyone knows not to move the cursor for the non-spacing character

\[
\begin{align*}
a + \` &= \text{à} \\
e + \' &= \text{é}
\end{align*}
\]
Diacriticals - 3

- You might want it either way
- Using non-spacing characters might make sorting easier (accent insensitive sorting)
- Or, might want each code point to be separate (accent sensitive sorting)
- LMBCS does not use non-spacing characters
Some character sets are written right-to-left
  ▶ Hebrew
  ▶ Arabic

But numbers are still left-to-right

Some character sets are traditionally top-to-bottom, right-to-left
  ▶ Japanese
  ▶ Chinese
BIDI - 2

- Mostly it's an issue for the rendering software
  - And for the input software
- But, how are the code points represented in memory?
  - Which order?
- The convention is that strings are represented in memory in "logical" order
  - I.E., in the order that you input them
BIDI - 3

- So, if you have a RTL sequence of 3 characters
  - Let's represent as: "ZYX"
- Followed by 3 digits ("123")
- You would see those code points in memory as:
  - XYZ123
- The rendering software has to know to reverse the letters, not the digits
Ligatures

- Historically, physical typefaces have combined certain individual letter combinations into a single glyph
  - ff, ffl, fl, fi, ffi
  - Because the tip of the 'f' overhangs the next character
- Most computer fonts ignore this
- Some don't, they want to render as historically accurate
  - You need to know the character following the 'f' before you can render
Ligatures - 2

- High-end publishing systems want to do this
- So room was made in some code pages for special glyphs representing the ligatures
- Makes sorting very complicated!
Chinese, Japanese, Korean

- Present a special problem, because to fully represent the writing system would consume over 150,000 code points
- Even fully 16-bit systems only give you 65,000 code points
- Compromise: Consolidate
  - A large percentage of the Traditional Chinese, Japanese Kanji and Traditional Korean character sets are held in common
  - So we can represent the set of common glyphs only once
- Referred to as CJK Consolidation
Limitations of LMBCS

- It's a very robust technique for representing multi-lingual text
  - Including within a single string!
  - In a reasonably compressed format (no embedded 0s)

- But...
  - The rest of the world adopted Unicode

- LMBCS remains a "proprietary" Lotus technology
Where is LMBCS?

- LMBCS is used uniformly throughout Lotus products
  - SmartSuite
  - Notes
  - Domino
- And nowhere else
  - Major investment in supporting software tools
  - More on this later
What Does Everyone Use Now?

- Unicode
  - Lotus also supports Unicode, as we'll see
- Unicode is an open standard
  - See http://www.unicode.org
What is Unicode?

- Earlier character encoding schemes are often referred to as SBCS or DBCS or MBCS
  - Single Byte Character Set (e.g., Ascii)
  - Double Byte Character Set (e.g., CP932)
  - Multi-Byte Character Set
- Unicode is none of these
What is Unicode?

- Unicode is not byte oriented, so calling it even MBCS is misleading
  - Though most of the time this is a true statement
  - There are times when you can treat it as an SBCS too
- My working definition:
  - A character encoding scheme that assigns a unique numeric value to every character
  - All written characters for all human languages (and many special characters as well) are accommodated
What is Unicode?

- **Note:** this definition says nothing about how big the values might be
  - Or how many "bytes" might be needed to represent any given character
  - Because that's not the important issue

- **The key point** is that the "name space" for characters in Unicode is flat
  - 0 is still a special value
  - Upper value is essentially unbounded
  - And therefore very extensible
What is Unicode?

- The current standard defines an upper limit of 0x10FFFFF (using 3 8-bit bytes)
  - More than one million code points
  - Of which about 5% is allocated
  - 13% is reserved for private use
  - 2% reserved
  - 5% planning underway
  - Plenty of room for growth
- Of course, the devil is in the details
Unicode Mappings

- The very large Unicode character space is laid out in regions
- Designed for interoperability with Ascii and ISO Latin-1
  - For ease of translation
- Ascii values (up to 0xff) are maintained unchanged
- Standards committee determines the rest of the values
Unicode Encodings

- Question on computer representations:
  - \(0x10ffff\) is 3 bytes, 24 bits
  - Most computers do not align on 3-byte boundaries
  - So we would "round up" to 4 bytes, 32 bits
- So, do we need to allocate 32 bits per character in all our strings?
  - And, if we do, won't there be embedded 0s?
Encodings - 2

- Yes, there would
- But we don't necessarily need all 32 bits for each character.
- There are alternate Unicode "encodings":
  - UTF-8: each char is 1, 2, 3, or 4 units
  - UTF-16: each char is 1 or 2 units
  - UTF-32: each char is 1 unit
- UTF-16 is the "default" encoding
A char's real value does not change with the encoding
  ▶ Only the number of bytes used to represent that value

"A" in UTF-8: 0x41 (byte)
in UTF-16: 0x0041 (short)
in UTF-32: 0x00000041 (DWORD)
Enodings - 4

- Implication:
  - In UTF-8 and (to a lesser extent) in UTF-16
  - You may STILL need multiple units to represent a given code point
  - Therefore, Unicode can still be "multi-byte", in a sense
  - Meaning, "multiple positions"
- Zero still used by convention to terminate strings
  - Though this can vary by programming language
  - And the number of bytes must conform to the encoding (UTF-8, 16, 32)
A Point on Diacriticals

- Unicode also gives you a choice on representation
  - One glyph for a "pre-composed" character
    - U-umlaut is 0x00fc
  - Base char + non-spacing char ("composed")
    - U + umlaut is 0x0075 + 0x0308
- So even in UTF-32 you may need 2 positions
Referencing Characters in Unicode

- All characters have a unique name
  - Based on ISO 10646
  - E.g., "Bengali Digit 5"
- The convention for representing a character's code point value in Unicode
  - U+xxxx
  - Always assumes Hex representation
Software for multi-language character sets

- The "standard" C library string functions are no longer usable
  - They all assume 1 byte == 1 char
  - strlen() tells you the length of a string in bytes, but not how many chars are in it
  - strcmp() is only valid for some SBCS, would never work with (e.g.) EBCDIC
    - Cannot add 0x20 to go from uc to lc
- You must use a software library that supports the character set in use
Software for multi-language character sets - 2

- What kinds of things do you need to do?
  - Length of string (chars, bytes)
  - Search for char in string
  - Search for substring in string
  - Move cursor within string (by char):
    - First, Last
    - Next, Prev
  - Truncate at a certain byte position
Software for multi-language character sets - 3

- Tasks, continued:
  - Test type of char (digit, alpha, upper, lower, accented, not accented...)
  - Modify casing (upper, lower)
  - Modify accenting (with, without)
  - Sort (compare 2 strings lexically)
  - Translate to/from native character set
A Word on Sorting

- All lexical sorting requires is the ability to compare any 2 characters and say one is < the other
- BUT: it's a complex topic
- Easy sorting simply compares raw code point values
  - 'A' < 'a'
  - 'a' < 'b'
  - '1' < '9'
- But where are numbers relative to letters?
  - In Ascii, all digits are < all letters
  - But not universally true!
Sorting - 2

- Need to handle comparison of characters using more than one byte
- Need to handle composed characters, too
- Real-world products need to offer multiple compare/sort options:
  - Case sensitive/insensitive
  - Accent sensitive/insensitive
  - Width sensitive/insensitive (for Asian double-wide characters)
  - Number first/numbers last
Software for multi-language character sets - 4

- Topics:
  - Unicode and Java
  - MLCS and Fonts: Rendering text
  - LMBCS and Unicode in Notes
Unicode and Java

- Java is all Unicode internally
  - Generally UTF-16 encoding
  - "char" is a 16-bit quantity in most cases
Editing Java Code

- Be careful using non-Ascii characters as literals in Java code
  - 'a' is fine
  - Special characters ('\t', '\n') are ok
  - 'KK' (some Kanji character) is dangerous
- This is editor-dependent
  - If your text editor supports Unicode, then it's ok
  - Otherwise, your code may not be compiled on all systems
Editing Java Code - 2

- Not an issue for Java code entered directly in Domino Designer
  - It's a LMBCS editor
  - Source code is stored in UTF-16

- This is not an issue for LotusScript or @Function code
  - Because you are always using a LMBCS editor
Unicode and Java - 2

- Java lets you specify the code point value for a single char:
  - '\uXXXX'
  - Must supply 4 digits, implying UTF-16
- byte is an 8-bit quantity in Java
  - Be careful! Smaller than a char!
- int is a 32-bit quantity in Java
  - So cannot always freely convert
  - c = (char) i; works most of the time
  - i = (int) c; always works
Character manipulation is built-in to String/StringBuffer classes
- `char c = string.charAt(i)`
- `toUpperCase/LowerCase()`
- `toCharArray()`
- `search, replace, etc., etc.`
- `append()`
Java Type Conversion

- Java will convert between String and char[] transparently
- Java will convert between String and byte[] with a specified encoding

```java
byte[] arrOfBytes = null;
String st = "Some string"
arrOfBytes = st.getBytes("UTF-8");

String st2 = new String(arrOfBytes, "UTF-8");
```
Java Type Conversion - 2

- Java will convert between String and char[] transparently
- Java will convert between String and byte[] with a specified encoding

```java
byte[] arrOfBytes = null;
String st = "Some string";
arrOfBytes = st.getBytes("UTF-8");

String st2 = new String(arrOfBytes, "UTF-8");
```

NOTE: length of st2 (chars) may not = length of arr (bytes)!
Java Type Conversion - 3

- Useful for converting byte-oriented strings (e.g., from C programs) to "real" ML strings
- Example: Notes converts all strings to UTF-8 when passing them to Java
  - But what if it's Kanji?

```java
String utf8 = document.getItemValueString("abc");
byte[] bytes = utf8.getBytes("UTF-16");
String kanji = new String(bytes, "UTF-16");
```
Java Supports Localization

- `java.util.Locale`
- Represents:
  - A "language code", e.g., "en", "de"
    - ISO 639
  - A "country code", e.g., "US", "CA"
    - ISO 3166
  - (Optional) A "variant", e.g., "posix", "MAC"
    - vendor specific
Localization - 2

- A Locale instance implies something about the usage/formatting of:
  - dates (9/10/01 vs. 10/09/01 vs. 10.09.01)
  - currency symbols
  - accented characters
- So, these are all different:
  - en/US, en/UK, en/IR
  - fr/CA, fr/FR, fr/BE
But: Locale is independent of display (font)
- Can query Java for the current, "default" locale
  - Dependent on system configuration
- Locale instances can be used to auto-format strings for display
If the file system is Unicode (e.g., Windows NT), then there are no issues

Otherwise, you have 2 choices:
  ▶ Write binary data in Unicode format
     • Not readable by other system tools
  ▶ Convert going in and out

LotusScript does Choice 2 automatically
Java and File I/O

- With Java, you can control how you do it
- Package java.io.*
  - Base layer consists of InputStream and OutputStream
  - Byte oriented, you figure it out
  - Other layers support richer data types
Java and File I/O

What's wrong with this code?
  ▶ Works, sometimes...

```java
FileInputStream fis = new FileInputStream("d:\temp\abc.txt");
char c; int i;
while ((i = fis.read()) >= 0)
{
    c = (char) i;
    // etc....
}
fis.close();
```
Java and File I/O

- It works when:
  - File contents are Ascii
  - File contents are Unicode
- Otherwise, you can't assume that raw bytes are cast-able to Unicode
  - Correctly, that is
  - You are likely to get garbage
Java and File I/O

- For "native" file i/o, use Reader/Writer classes
  - char oriented, not byte oriented

FileReader fr = new FileReader("d:\temp\abc.txt");
char c; int i;
while ((i = fis.read()) >= 0)
{
    c = (char) i;
    // etc....
}
fr.close();
MLCS and Fonts: Rendering Text

- Rendering means:
  - Displaying a stream of code point values on a screen, or on paper
- Requires mapping a code point value to a particular glyph
  - Or a sequence of code point values to a single glyph
- Requires a displayable glyph for the code point value
  - With attributes too (bold, italic, superscript, etc.)
Rendering - 2

- A "Font" (in computer terms) is a set of glyphs
  - Generally a table of displayable glyphs
  - Upper and lower case letters, digits, etc.
  - Kanji characters
- Can have multiple Fonts for any language character set
  - Just a different way of rendering each glyph
  - Times New Roman vs. Helvetica
Rendering - 3

- Not all fonts have glyphs for the entire Unicode space
  - Most don't
- "Missing" glyphs default to a "fallback" character
  - The character data is still all there!
  - It's only the rendering software that can't handle the "missing" glyph
The character set that a font maps to is vendor-dependent

- Some are ASCII only
- Some are Asian only
  - Though most Asian fonts also include ASCII
- Others handle (some subset of) Unicode
- Rendering software is responsible for directionality
  - BIDI
  - Top-down
Rendering - 5

- The amount of space a given string uses on the screen (or on paper) depends on:
  - Font characteristics (monospaced, proportional)
  - Attributes (bold, plain)
  - Point size (10, 12, 48...)
  - Pixel resolution of the display (DPI)
  - Actual chars used (esp. if proportional)

- Can be very complex to compute!
Java & LMBCS in Notes

- All system display software requires "native" character set
- All Lotus product code bases use LMBCS as the character set
  - Internal manipulation (sorting, searching, etc.)
  - Persistent storage (on disk)
- With one exception:
  - LotusScript stores all strings as Unicode (UTF-16)
  - Must convert between:
    - LMBCS (for Notes)
    - Native (for display)
Calling External Code From Java & LotusScript

- LotusScript calls externally using "Declare" statements
  - Describes library where code resides, name of entry point, and arguments
  - For String types, can declare the arg as:
    - "ByVal stringarg As LMBCS String"
    - Does conversion automatically
    - MUST be "ByVal"
- Otherwise, String is Unicode
Calling External Code From Java & LotusScript - 2

- In Java, you can't invoke C entry points directly
  - Must use Java Native Interface (JNI)
  - C entry point receives a pointer to a Java services vector, plus arguments
    - Args are scalars or Java object handles
- You use the JVM services vector to manipulate the arguments
  - E.g., convert String objects to UTF-8 string buffers
UI Support for Composed Characters

- In the Notes Client you can use Alt-F1 to compose chars
  - Alt-F1 + a + ` = à
  - Alt-F1 + c + , = ç
  - and so on
Notes - 2

- Most conversions are transparent to the LS developer
  - Java Strings converted to Notes LMBCS
  - Notes LMBCS strings converted to Unicode (UTF-8)
  - LS strings converted to Notes LMBCS
  - Notes LMBCS converted to Unicode (UTF-16)
Notes - 3

- When using Notes C API from LotusScript, need to worry about string type
  - Declare .... (...., st byval As LMBCS string)
  - Converts automatically to LMBCS

- When using custom C DLLs called from LotusScript
  - Either do same Declare trick
  - Or receive strings as UTF-16 Unicode
For Java, when calling C:
- String objects received as object handles
- Use Java Native Interface (JNI) services to extract string value the way you want it
  - UTF-8, UTF-16

Notes has many entry points in the C API for dealing with multi-language strings
Multi-language APIs

- Categories:
  - Translation (OSTranslate, NLS_Translate)
  - Parsing, searching (NLS_XXX)
- All calls are documented in the Notes C API toolkit
- NOTE: All "char *" decls in the C API mean "unsigned char"
  - LMBCS
OSTranslate

- The key is the "TranslateMode" options
  - Note: result-length may vary from input-length!

```c
WORD LNPUBLIC OSTranslate(
    WORD TranslateMode,
    char far *In,
    WORD InLength,
    char far *Out,
    WORD OutLength);
```
OSTranslate - 2

- Assumes that either source or destination string (or both) is LMBCS Group 1
  - Notes assumes Group 1 strings are optimized
    - Leading group indicator-bytes compressed out
  - All others non-optimized
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS_TRANSLATE_NATIVE_TO_LMBCS</td>
<td>Convert input string from machine's native character set to LMBCS</td>
</tr>
<tr>
<td>OS_TRANSLATE_LMBCS_TO_NATIVE</td>
<td>Convert input string from LMBCS to machine’s native character set.</td>
</tr>
<tr>
<td>OS_TRANSLATE_LOWER_TO_UPPER</td>
<td>Current international case table.</td>
</tr>
<tr>
<td>OS_TRANSLATE_UPPER_TO_LOWER</td>
<td>Current international case table.</td>
</tr>
<tr>
<td>OS_TRANSLATE_UNACCENT</td>
<td>International unaccenting table. NO REVERSAL!!</td>
</tr>
<tr>
<td>OS_TRANSLATE_OSNATIVE_TO_LMBCS</td>
<td>Same as NATIVE (not documented!)</td>
</tr>
<tr>
<td>OS_TRANSLATE_LMBCS_TO_OSNATIVE</td>
<td>Same as TO_NATIVE (not documented!)</td>
</tr>
<tr>
<td>OS_TRANSLATE_LMBCS_TO_ASCII</td>
<td>Convert the input string from LMBCS to character text.</td>
</tr>
<tr>
<td>OS_TRANSLATE_LMBCS_TO_UNICODE</td>
<td>Convert the input string from LMBCS to UNICODE.</td>
</tr>
<tr>
<td>OS_TRANSLATE_LMBCS_TO_UTF8</td>
<td>Convert the input string from LMBCS to UTF8.</td>
</tr>
<tr>
<td>OS_TRANSLATE_UNICODE_TO_LMBCS</td>
<td>Convert the input string from UNICODE to LMBCS.</td>
</tr>
<tr>
<td>OS_TRANSLATE_UTF8_TO_LMBCS</td>
<td>Convert the input string from UTF8 to LMBCS.</td>
</tr>
</tbody>
</table>
OSTranslate - 4

- Note that multiple translations are "lossless"
  - If you go one way, then come back, all data is preserved
- EXCEPT:
  - Accented to unaccented
  - LMBCS to Ascii (maybe)
- Complicates display input and editing
National Language System (NLS) APIs

- Supports low-level character based parsing, searching and translation
- Any character set, not just LMBCS/1 or current native
NLS_PINFO

- Use NLS_load_charset to read a charset descriptor table from disk
- Returns a NLS_PINFO *
  - Opaque data structure, but you will need it
- List of character set IDs found in NLS.H
- Used in NLS_translate
NLS-translate

- More functionality than in OSTranslate
  - Arbitrary translation to/from any character set
  - NLS_translate

```c
NLS_STATUS LNPUBLIC NLS_translate(
    BYTE far *pString,
    WORD    Len,
    BYTE far *pStringTarget,
    WORD    far *pSize,
    WORD    far *pControlFlags,
    NLS_PINFO  pInfo);
```
# NLS Translation Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLS_NONNULLTERMINATE</td>
<td>Does not add a NULL to the end of the translated result.</td>
</tr>
<tr>
<td>NLS_NULLTERMINATE</td>
<td>Adds a NULL to the end of the translated result.</td>
</tr>
<tr>
<td>NLS_STRIPUNKNOWN</td>
<td>Strips unknown characters.</td>
</tr>
<tr>
<td>NLS_TARGETISLMBCS</td>
<td>Converts target string to LMBCS.</td>
</tr>
<tr>
<td>NLS_SOURCEISLMBCS</td>
<td>Converts source string from LMBCS.</td>
</tr>
<tr>
<td>NLS_TARGETISUNICODE</td>
<td>Converts target string to UNICODE.</td>
</tr>
<tr>
<td>NLS_SOURCEISUNICODE</td>
<td>Converts source string from UNICODE.</td>
</tr>
<tr>
<td>NLS_TARGETISPLATFORM</td>
<td>Converts target string to the encoding used by the host OS.</td>
</tr>
<tr>
<td>NLS_SOURCEISPLATFORM</td>
<td>Converts source string from the encoding used by the host OS.</td>
</tr>
</tbody>
</table>
Other NLS API Calls

- NLS_string_chars
  - Number of chars in a string
- NLS_string_bytes
  - Number of bytes in a string (same as strlen)
- Test a character for type:
  - isdigit, isupper, islower, isspace, ispunct, iscntrl, isalnum, isalpha, isarith,
Other NLS API Calls - 2

- isleadbyte
  - If True, then the next 1 (or 2, or 3) bytes are part of the same character
- Enumerate through a string
  - NLS_get
- Parse strings
  - goto_next_break
  - goto_next/prev_word_end/start
Other NLS API Calls - 3

- Search
  - find, find_substr
String Truncation

- String truncation: don't ignore this!
  - Failure to do this properly can cause crashes

- Situation:
  - You have an N-byte long MLCS string
  - You must truncate to M bytes
  - M < N
  - For display, to fit in a fixed buffer, whatever
String Truncation - 2

- Can't just chop it off at M bytes!
  - Might be the middle of a multi-byte sequence
  - Will result (at least) in garbage on the display

- Correct technique:
  - Get pointer to max slot in string
  - Call NLS_goto_prev
    - Pass in ** for current position, * for start of string
  - Moves your pointer to the start of the previous char
  - Can truncate there, or at start of next char
MLCS APIs - Summary

- All the old C techniques are now useless
  - Worse: wrong!
- You will need to accustom yourself to a new set of assumptions and functions
- But once you get comfortable with it, it's not difficult
  - Make it habitual!
- NLS calls somewhat limited:
  - Only a few of the string-compare and search equivalents for MLCS are exposed!
  - Testers for full/half pitch not exposed!
Web applications and MLCS

- What happens Notes data served to the Web?
- HTTP allows specification of the page charset
  - Client can specify desired charsets on request
  - Server specifies charset on result page
Web Applications - 2

- Notes defaults to ISO Latin-1 for output
  - Automatically converts from LMBCS
- But, how do you know what the Client's real charset is?
  - Look at requested charsets
  - Look at country/language code
- Can you tell Domino what charset to use on output?
Web Applications - 3

- Yes!

- Domino examines Web agent output for HTTP headers
  - Must occur BEFORE any data
  - Domino will notice them and pass them through
  - If you specify a Content-Type header, Domino will try to translate
Web Applications - 4

- Just print out the header (in the correct format) as the first thing:

```
print "Content-Type: text/html; charset=ISO-8859-4\nprint ""\n' null to force a newline: REQUIRED\nprint "Yo, world!"
```
Web Applications - 5

- Unfortunately, only works from agents
- No way (that I know of) to build this into a form or page
Web applications and MLCS

- Another trick for using in browsers
- HTTP specifies that Unicode values can be substituted for any char
  - 'Space' == U+0020
  - Can encode for HTTP as
    - %0020
    - Or %20
- So you can encode arbitrary Unicode chars in your Notes form or page!
  - The browser will interpret it correctly
  - Very useful when encoding pass-thru URLs
Summary

- If everyone spoke (and wrote!) only one language, it would be easy
- Remember: character set encodings are independent of rendering
  - Though typographical legacies complicate encodings (e.g., ligatures)
- Unicode is an important technological advance
  - Everything defined in a single "space"
But the tools are not yet sophisticated enough

- Developers still need some knowledge of how character sets work
- And Unicode implementations are not always uniform
  - UTF-8? 16? 32?
  - Java UTF-16 still has multi-char code points
Summary - 3

- It will keep getting better
  - A good thing
- It will keep changing
  - Not always so good
Q & A