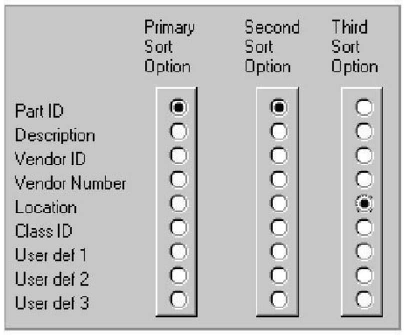
Lecture 22: Internationalization

UI Hall of Fame or Shame?



Source: UI Hall of Shame

Our Hall of Fame or Shame candidate for the day is this interface for choosing how a list of database records should be sorted. Let's discuss its advantages and disadvantages, and contemplate alternative designs.

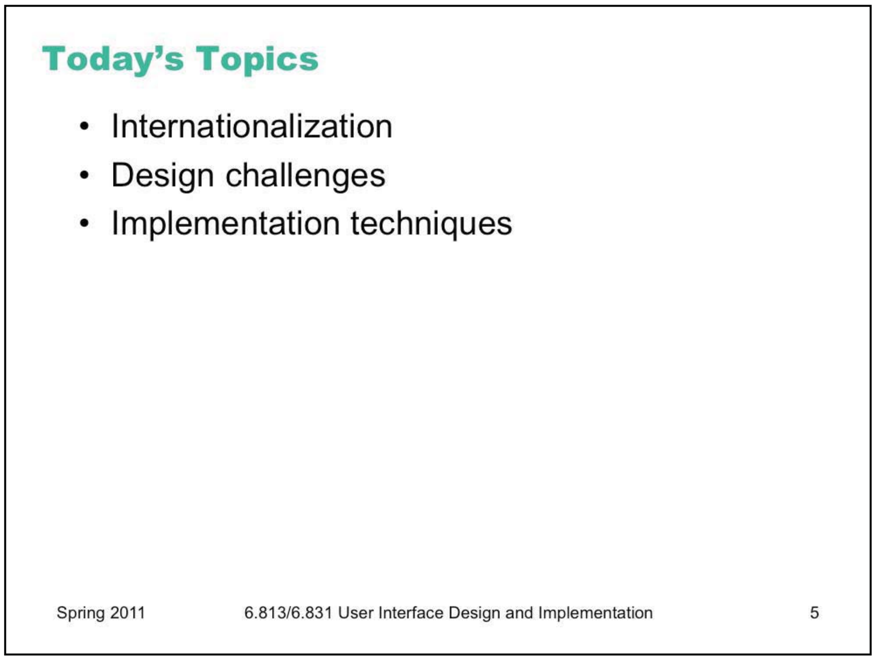
Like other sorting interfaces, this one allows the user to select more than one field to sort on, so that if two records are equal on the primary sort key, they are next compared by the secondary sort key, and so on.

On the plus side, this interface communicates one aspect of its model very well. Each column is a set of radio buttons, clearly grouped together (both by **gestalt proximity** and by an explicit raised border). Radio buttons have the property that only one can be selected at a time. So the interface has a clear **affordance** for picking only one field for each sort key.

But, on the down side, the radio buttons don't afford making NO choice. What if I want to sort by only one key? I have to resort to a trick, like setting all three sort keys to the same field. The interface model clearly doesn't map correctly to the task it's intended to perform. In fact, unlike typical model mismatch problems, both the user and the system have to adjust to this silly interface model – the user by selecting the same field more than once, and the system by detecting redundant selections in order to avoid doing unnecessary sorts.

The interface also fails on **minimalist** design grounds. It wastes a huge amount of screen real estate on a two-dimensional matrix, which doesn't convey enough information to merit the cost. The column labels are similarly redundant; "sort option" could be factored out to a higher level heading.

The term "Primary" is not really consistent with "Second" and "Third"; "First" would be simpler.



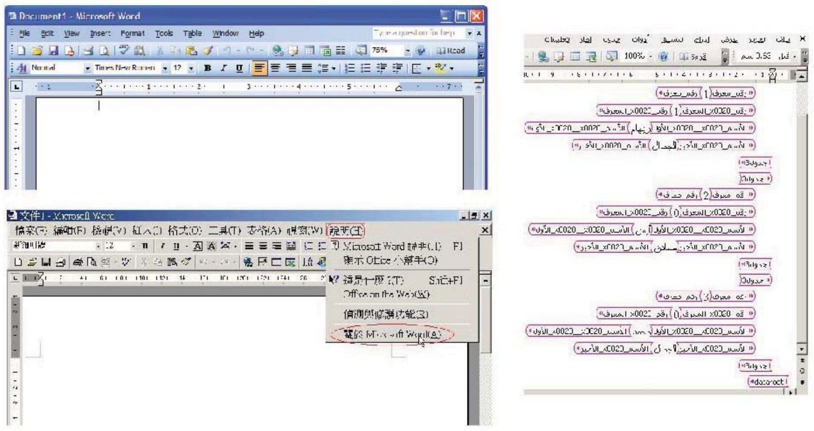
Today's Topics

- Internationalization
- Design challenges
- Implementation techniques

Today's lecture concerns **internationalization**: supporting users who speak different languages and have different cultural conventions. We'll talk about some of the reasons why internationalization can be hard, and discuss some of the support that exists in GUI toolkits for making it easier.

A good source of information about this problem is *Java Internationalization*, by Andy Deitsch and David Czarnecki (O'Reilly, 2001). There's also a trail in the Java Tutorial about Java's internationalization features (http://java.sun.com/docs/books/tutorial/i18n/index.html).

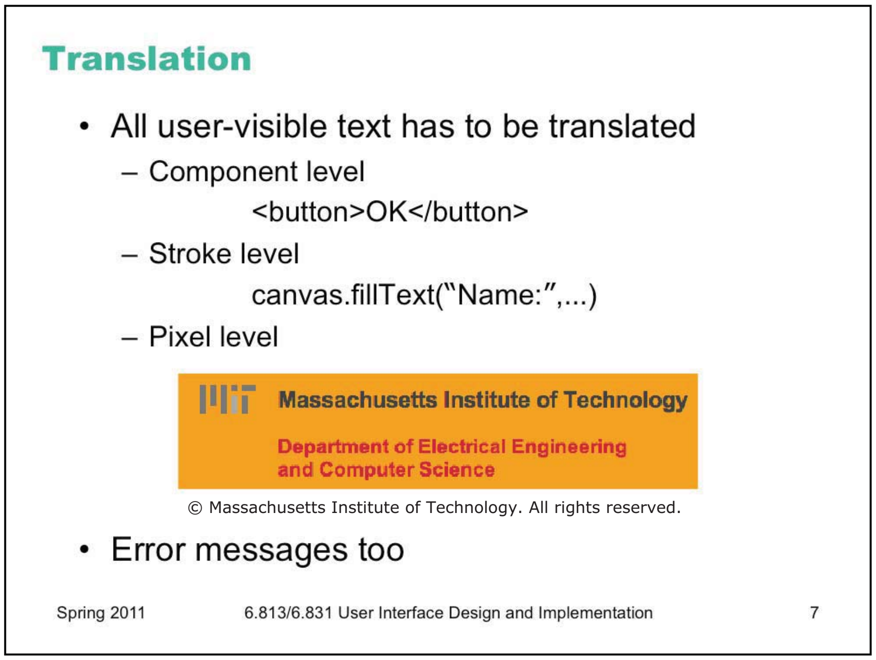
Internationalization and Localization


Document1 - Microsoft Word
File Edit View Insert Format Tools Table Window Help
Normal
Times New Roman
12
75%

Interfaces with international user populations – such as Microsoft Word, shown here – have to be carefully designed to make them easy to adapt to other languages and cultures. The process of making a user interface *ready* for translation is called **internationalization** (often called **i18n** for short – "18" because it replaces 18 characters in the middle of "internationalization"). Essentially, internationalization separates the language-specific parts of the interface from the rest of the code, so that those parts can be easily replaced. The translation is usually done by nonprogrammers, so their job is easier if the textual messages are separate from the code. Actually doing this translation for a particular language and culture is called **localization**.

One way to understand the difference between these two technical terms is by analogy to portability across operating system platforms. If you write your program carefully so that it doesn't depend on specific features of an operating system or processor, you've made it portable. Making a program portable is analogous to internationalizing it. Actually *porting* it to another particular platform, e.g. by recompiling it, is analogous to localizing it.

Unfortunately localization is much harder than merely knowing what words to substitute (and online translators like Babelfish and Google Translate can only barely do that, so don't rely on them!) You can't necessarily rely on bilingual members of your design team, either. They may be reasonably fluent in the other language, but not sufficiently immersed in the **culture** or **national standards** to notice all the places where the application needs to change. **You are not the user** is especially true in internationalization.



Translation

- All user-visible text has to be translated
  - Component level

    `<button>OK</button>`
  - Stroke level

    `canvas.fillText("Name:",...)`
  - Pixel level

    Massachusetts Institute of Technology
    Department of Electrical Engineering and Computer Science

    © Massachusetts Institute of Technology. All rights reserved.
- Error messages too

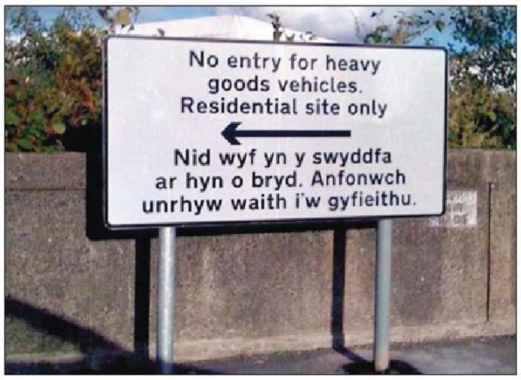
Spring 2011 6.813/6.831 User Interface Design and Implementation 7

Here are some of the reasons why internationalization is hard.

First, every piece of text that might be shown to the user is a potential candidate for translation. That means not just properties of components (like menu items and button labels), but also text drawn with stroke drawing calls, and text embedded in pixel images (like this one taken from the MIT EECS web page). Translation can easily change the size or aspect ratio of the text; German labels tend to be much longer than English ones, for example.

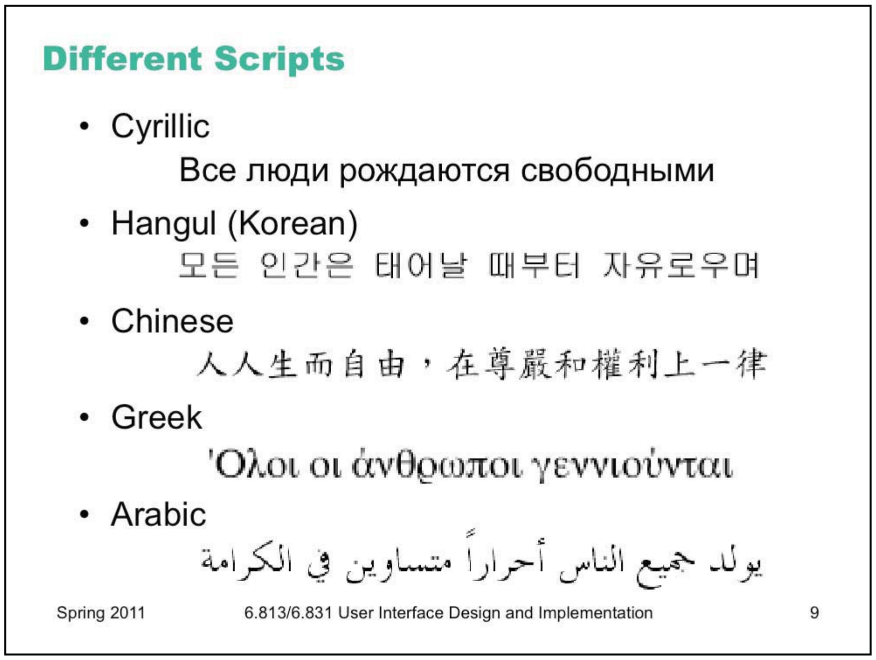
Error messages also need to be translated, of course – which is another reason not to expose internal system names in error messages. An English-reading user might be able to figure out what FileNotFoundException means, but it won't internationalize well.

Risks of Translation


No entry for heavy goods vehicles. Residential site only
Nid wyf yn y swyddfa ar hyn o bryd. Anfonwch unrhyw waith i'w gyfieithu.

© BBC News. All rights reserved.

Here's a sign from Wales, where official signs are required to be bilingual (English and Welsh). The English is clear enough to English-speaking lorry drivers - but the Welsh actually reads "I am not in the office at the moment. Send any work to be translated." The translation was outsourced by email, you see... (http://news.bbc.co.uk/2/hi/uk_news/wales/7702913.stm)
There's a larger lesson here that translation without sufficient context can lead to errors. The BBC article cited just above has some amusing examples of other English/Welsh signs that are mistranslated ("staff" => "wooden stick") because the translator wasn't fully aware of the context.



Different Scripts

- Cyrillic
  Все люди рождаются свободными
- Hangul (Korean)
  모든 인간은 태어날 때부터 자유로우며
- Chinese
  人人生而自由，在尊嚴和權利上一律
- Greek
  'Ολοι οι άνθρωποι γεννιούνται
- Arabic
  يولد جميع الناس أحراراً متساوين في الكرامة

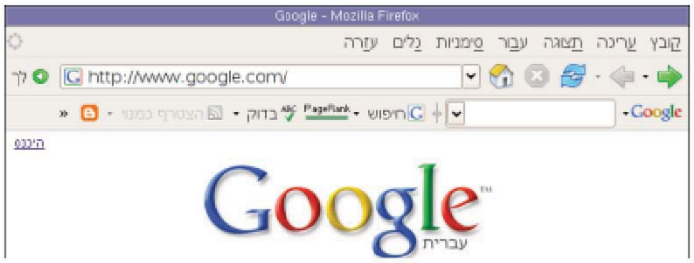
Spring 2011 6.813/6.831 User Interface Design and Implementation 9

Different languages obviously use scripts other than the Latin alphabet. Here are some of the scripts that Windows, Mac, and web browsers all support.

It's important to distinguish between **script** (or alphabet) and **language**. Western languages like English, French, German, and Italian are different languages that all use the Latin alphabet (basically). Russian, Ukrainian, and Bulgarian (among others) use the Cyrillic alphabet.

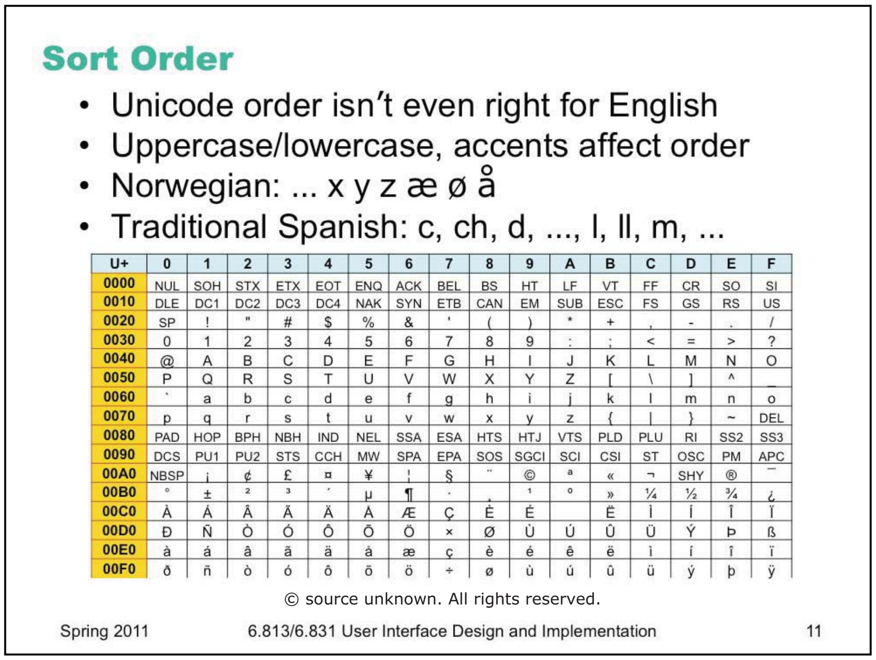
Text Direction

- Some scripts don't read left-to-right
  - Arabic, Hebrew are right-to-left
  - Affects drawing, screen layout, even icons



© Google, Inc. All rights reserved.

Many scripts are not even written left-to-right; Arabic and Hebrew are the most common languages with scripts written right-to-left. CJK (Chinese, Japanese, Korean) characters are usually written left-to-right, but can also appear vertically (top-to-bottom) and occasionally even right-to-left. Reversing the direction of reading requires reversing the entire layout of your screen, since the user is now accustomed to starting from the right edge when they scan. It might even affect the "natural" direction of arrow icons. The picture above shows the Hebrew version of Firefox. Notice that the menu bar is reversed (it starts from the right, and the rightmost menu is the File menu), the toolbar is reversed, and the Back button (which is now the rightmost button) is now pointing to the right! The URL box isn't reversed, however, because it uses the Latin alphabet, not Hebrew. This is another common wrinkle in right-to-left languages: when they embed foreign words, or Arabic numbers, the embedded words go in left-to-right order. So the text might be constantly switching direction.



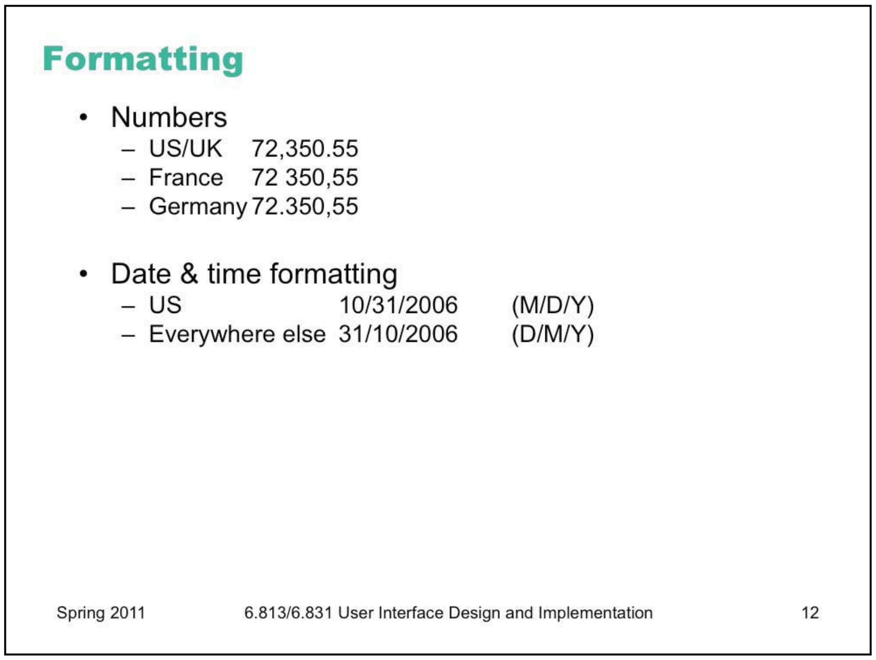
Sort Order

- Unicode order isn't even right for English
- Uppercase/lowercase, accents affect order
- Norwegian: ... x y z æ ø å
- Traditional Spanish: c, ch, d, ..., l, ll, m, ...

U+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0010	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0020	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0030	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
0080	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
0090	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
00A0	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
00B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
00C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É		Ë	Ì	Í	Î	Ï
00D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
00E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
00F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Sorting, or collation, is another way that languages differ. In software, each character is represented by a number. This mapping is the **character encoding**. Java, for example, uses Unicode, representing each character by a 16-bit number. But the ordering of these numbers doesn't necessarily match the conventional ordering of the characters in the language, so sorting text with < or String.compareTo() is almost certainly wrong. It's even wrong for English! Unicode groups the uppercase and lowercase letters separately, so that the sort order by < would be ABC...XYZ...abc... xyz...

Similarly, in most European languages, accented characters are sorted with the plain version of the character, even though the Unicode characters may be nowhere near each other in numerical order. And that general rule is not true in Norwegian, where å actually appears at the *end* of the alphabet, after z.



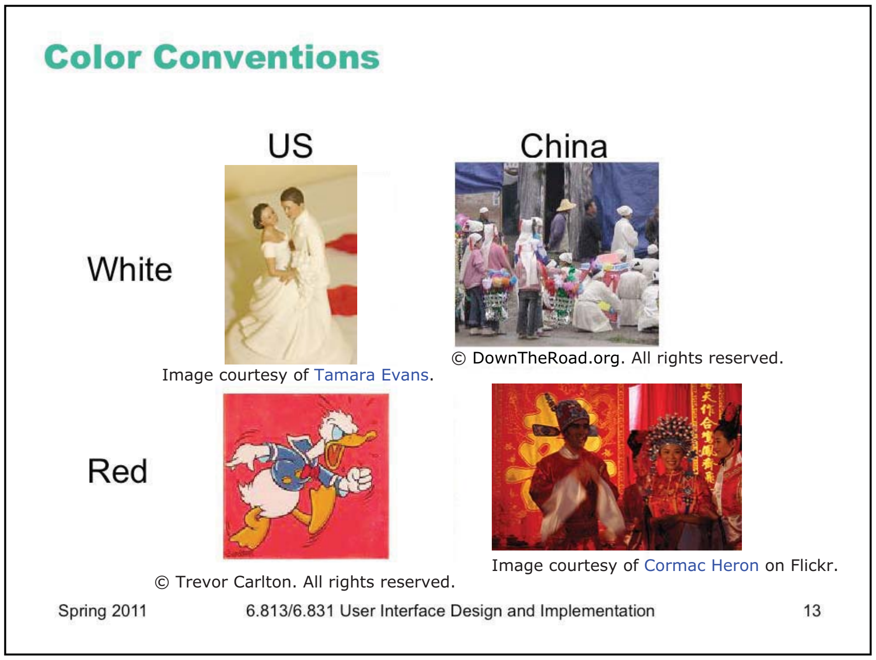
Formatting

- Numbers
  - US/UK 72,350.55
  - France 72 350,55
  - Germany 72.350,55
- Date & time formatting
  - US 10/31/2006 (M/D/Y)
  - Everywhere else 31/10/2006 (D/M/Y)

Spring 2011 6.813/6.831 User Interface Design and Implementation 12

Number formats and date formats also vary – not just by language, but by country. In the US, commas are used for millions and thousands, and a period for the decimal point, as in "72,350.55". But the convention in Germany is precisely the opposite: "72.350,55". Even countries that share the same language may differ on conventional formats. Americans tend to write dates as MM/DD/YY, but British write DD/MM/YY (as does most of the rest of the world).

The target for localization therefore needs to be specified by a language/country pair, also called a **locale**, such as US English, UK English, or Canadian French.



Color Conventions

US China

White

Image courtesy of Tamara Evans.

© DownTheRoad.org. All rights reserved.

Red

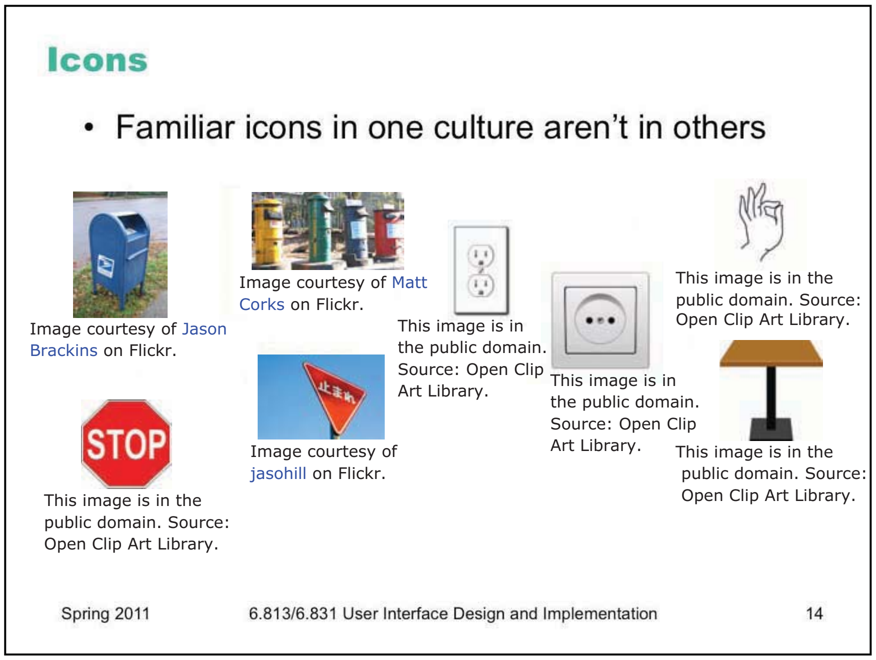
© Trevor Carlton. All rights reserved.

Image courtesy of Cormac Heron on Flickr.

Spring 2011 6.813/6.831 User Interface Design and Implementation 13

Localizing a user interface requires knowing about the cultural associations attached to symbols or colors, and making sure you don't send the wrong message.

For example, **colors** have different meanings in different cultures. In East Asia, particularly China, white is associated with death, and is used as a color theme for funerals. In the West, on the other hand, white is a symbol of purity, and brides wear white at their weddings. Traditional Chinese weddings involve a lot of red, because it symbolizes luck. Western cultures don't have the same association for red.



Icons

- Familiar icons in one culture aren't in others

Image courtesy of Jason Brackins on Flickr.

Image courtesy of Matt Corks on Flickr.

This image is in the public domain. Source: Open Clip Art Library.

This image is in the public domain. Source: Open Clip Art Library.

This image is in the public domain. Source: Open Clip Art Library.

This image is in the public domain. Source: Open Clip Art Library.

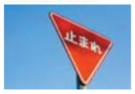
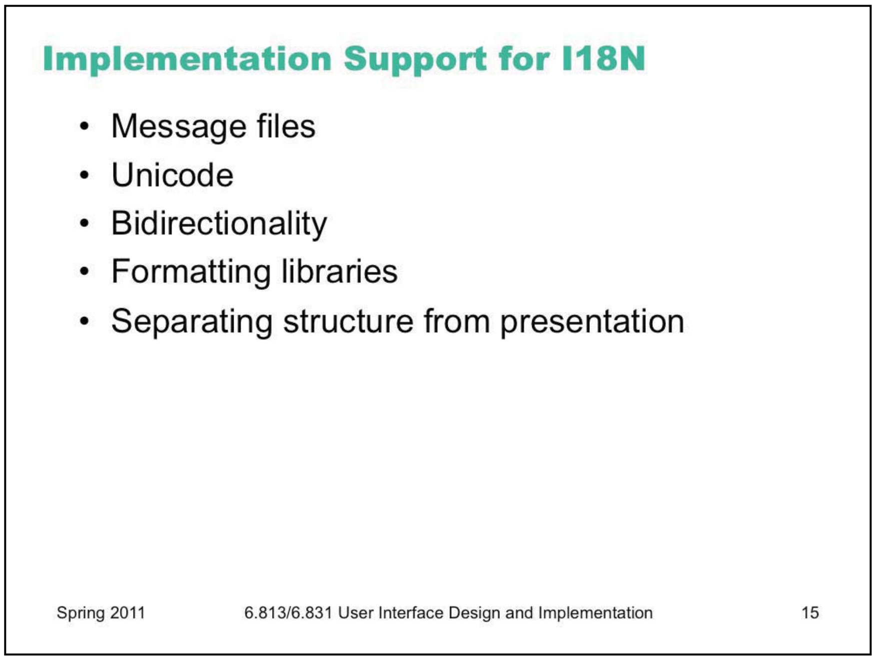


Image courtesy of jasohill on Flickr.

This image is in the public domain. Source: Open Clip Art Library.

Spring 2011 6.813/6.831 User Interface Design and Implementation 14

Icons must also be carefully chosen, or replaced when the interface is localized. Metaphorical icons that refer to everyday objects like mailboxes and stop signs aren't necessarily recognizable, because the objects may look different in different countries. (Stop signs are actually pretty universal, however – I had to look hard to find a stop sign that wasn't a red octagon, like this Japanese inverted triangle.) Hand gestures pictured as icons may actually be offensive in some countries. And **visual puns** are always a bad idea – an English-speaking designer might think it's cute to use a picture of a table (the furniture) to represent table (the 2D grid), because the words are the same in English. But the words in German are *tisch* (furniture) and *tabelle* (grid), so a German may find the joke incomprehensible.

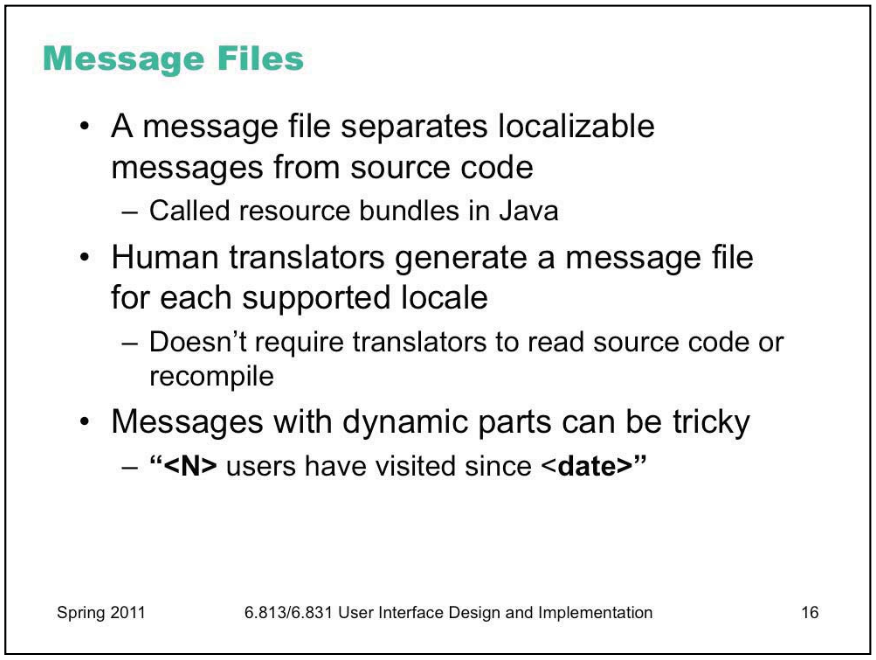


Implementation Support for I18N

- Message files
- Unicode
- Bidirectionality
- Formatting libraries
- Separating structure from presentation

Spring 2011 6.813/6.831 User Interface Design and Implementation 15

Now that we've surveyed the challenges, let's talk about some solutions. Modern UI toolkits provide support that make it easier to implement internationalized interfaces.



Message Files

- A message file separates localizable messages from source code
  - Called resource bundles in Java
- Human translators generate a message file for each supported locale
  - Doesn't require translators to read source code or recompile
- Messages with dynamic parts can be tricky
  - **"<N>** users have visited since **<date>"**

Spring 2011 6.813/6.831 User Interface Design and Implementation 16

Java, for example, has a framework called **resource bundles** that allow textual messages to be stored separately from the code, but as loadable resources in JAR files, so that an application can be localized simply by replacing those text messages. The messages are referred to by names, such as bundle.getString("file-menu-label").

This is an example of the general strategy for internationalization. First, use abstraction to isolate the parts of your system that need to change from one locale to another, separating it from the rest of your program. (This is an application of a familiar software engineering rule – if you know something will change, isolate it.) Second, as much as possible, design these locale-specific parts so that they don't require reading source code or recompiling the program, so that localization can be done by nonprogrammers.

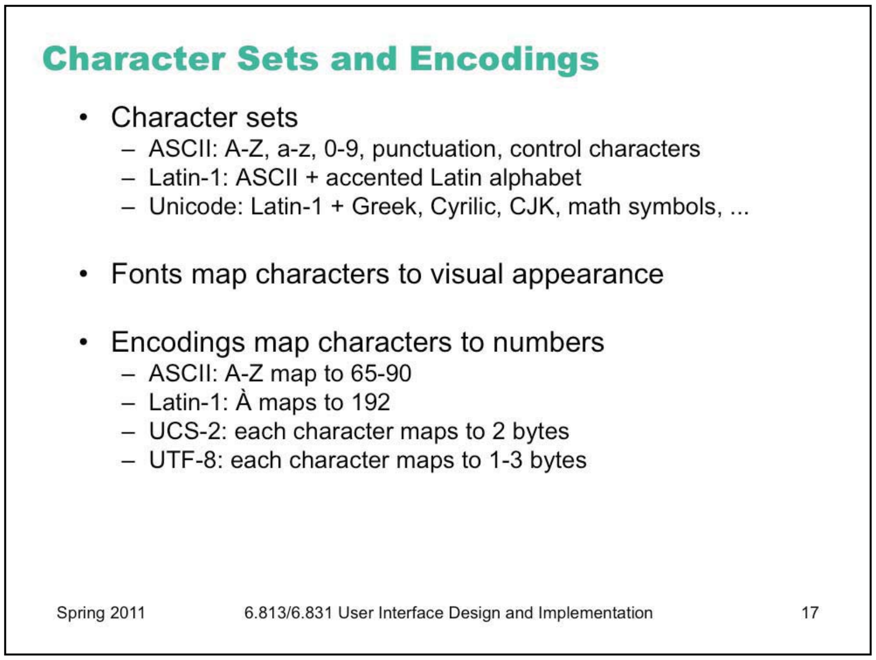
Internationalization gets a little tricky when a message has dynamic parts, like "25 users have visited since January 1". In an uninternationalized program, you might simply concatenate in your source code: num + "users have visited since "+ date. For internationalization, you need to give the translator flexibility to put the dynamic parts anywhere, using a format like "%1 users have visited since %2", so that it could be rewritten as "Since %2, %1 users have visited" if the language demands it. And you also need to think about plurals, usually by having different versions of the entire message that depend on the value of num:

num == 0 => "%1 users have visited since %2"

num == 1 => "%1 user has visited since %2"

num > 1 => "%1 users have visited since %2"

Java has a class ChoiceFormatter that makes this task somewhat easier. But be careful – Arabic has a different plural form when num == 2 than when num > 2. (http://en.wikipedia.org/wiki/Grammatical_number).



Character Sets and Encodings

- Character sets
  - ASCII: A-Z, a-z, 0-9, punctuation, control characters
  - Latin-1: ASCII + accented Latin alphabet
  - Unicode: Latin-1 + Greek, Cyrilic, CJK, math symbols, ...
- Fonts map characters to visual appearance
- Encodings map characters to numbers
  - ASCII: A-Z map to 65-90
  - Latin-1: À maps to 192
  - UCS-2: each character maps to 2 bytes
  - UTF-8: each character maps to 1-3 bytes

Spring 2011 6.813/6.831 User Interface Design and Implementation 17

Supporting other languages means you're not just playing with ASCII anymore. The old 7-bit ASCII character set only supports English, in fact. The 8-bit Latin-1 adds accented characters for Western European languages among the upper 128 characters, but for real internationalization, you need 16-bit Unicode, which includes characters from many other alphabets (e.g. Arabic, Hebrew, Cyrillic) and thousands of CJK (Chinese Japanese Korean) ideograms. Java uses Unicode, so chars and Strings in Java can represent text in all these languages.

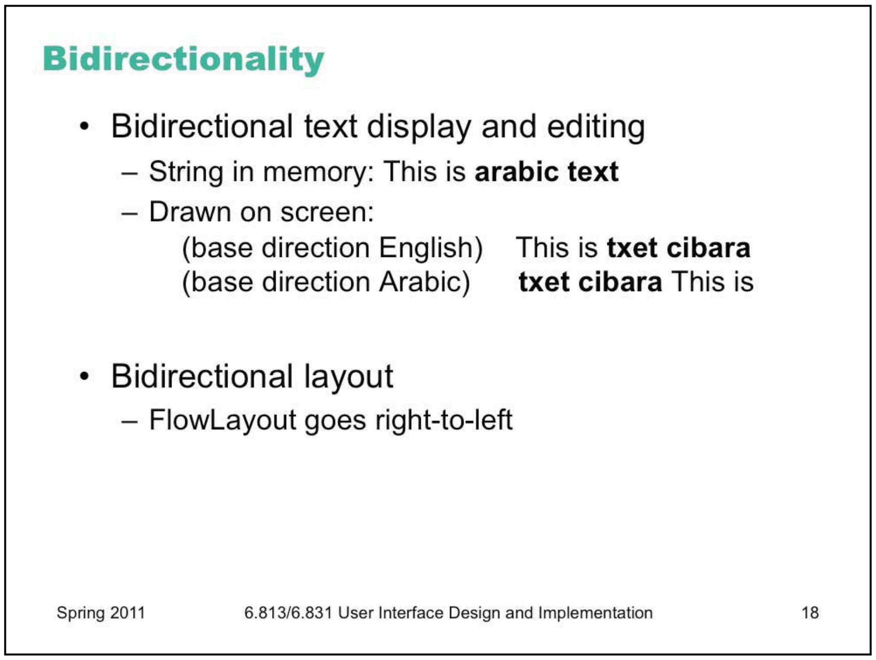
Note the difference between **character sets** and **fonts**. The Unicode character A' doesn't actually say how to *draw* A on the screen; a font does that. So even though you can represent many different alphabets in a single Unicode string, the *font* you're drawing the string with doesn't necessarily know how to draw all those characters. The appearance of a particular character in a font is called a glyph. Many fonts only have glyphs for a small subset of Unicode. For characters that aren't supported by the font, you'll see an error glyph, which might look like a little empty square or a question mark.

Note also the difference between character sets and **encodings**. A character set is an abstract set of possible characters. ASCII had 128 characters; Latin-1 had 256 characters, and Unicode has thousands of characters. An encoding maps each character in a character set to a number (or a small sequence of numbers). Internally, Java uses a 16-bit encoding for Unicode characters, representing each character by two bytes in memory. But the most common encoding for Unicode text in files and web pages is UTF-8, which does *not* use two bytes per character. Instead, UTF-8 uses 1, 2, or 3 bytes to represent each character. Single bytes are used to represent all the 7-bit ASCII characters, so UTF-8 can be thought of as an extension to ASCII.

There are other encodings as well. ASCII maps its characters to the numbers 0-127, which are stored in bytes. Latin-1 (also called ISO 8859-1 after its ISO standard) maps its characters to 0-255 (compatibly).

In general, **you cannot correctly interpret a text file or web page without knowing its encoding**. If your code ignores encodings and assumes everything is ASCII, you will find that it mostly works as long as you only use English, because encodings generally strive for backwards compatibility with ASCII. In other words, an English text would probably look identical in ASCII, UTF-8, and Latin-1. But it may break horribly on text in other languages. Even English text has problems when the author uses punctuation that isn't available in the basic ASCII character set. For example, ASCII only had one kind of double-quote mark (a vertical one), but many word processors now use left and right double quotes that are available in Unicode and other character sets, which often turn into garbage characters when you load the text into encoding-ignorant programs.

For more about encodings, Joel Spolsky has a good article (http://www.joelonsoftware.com/articles/Unicode.html).



Bidirectionality

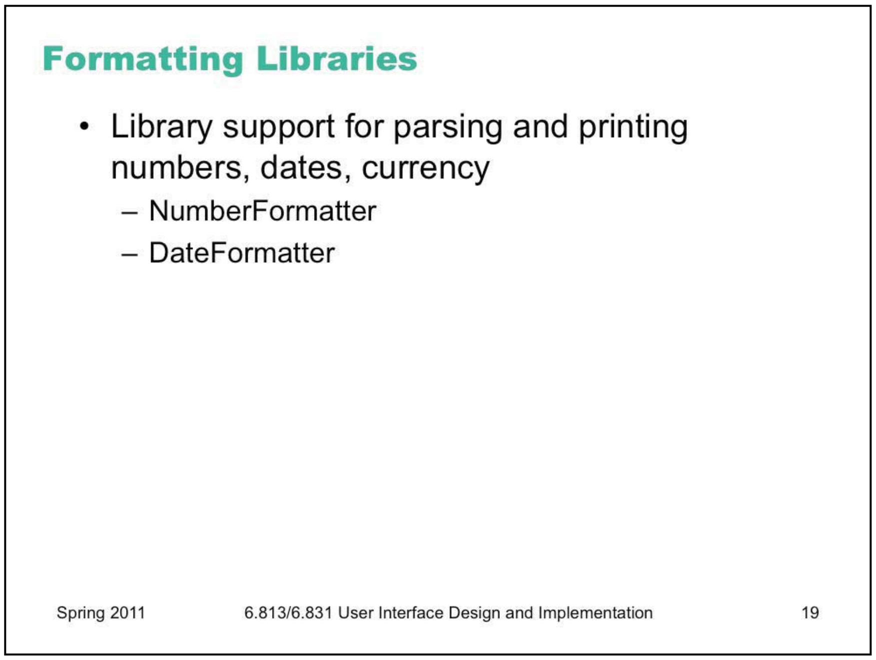
- Bidirectional text display and editing
  - String in memory: This is **arabic text**
  - Drawn on screen:
    (base direction English) This is **txet cibara**
    (base direction Arabic) **txet cibara** This is
- Bidirectional layout
  - FlowLayout goes right-to-left

Spring 2011 6.813/6.831 User Interface Design and Implementation 18

To handle languages that read right-to-left, UI toolkits like Java provide support for **bidirectional text** (sometimes called BiDi or BIDI for short). The trickiest part here is that Unicode strings may (and often do!) mix characters from multiple scripts: Arabic and English, for example. A good UI toolkit will ensure that when you draw such a string to the screen, it draws the appropriate characters in the appropriate order. There must be a *base direction* that determines whether the whole string starts at the left or the right; if the interface is primarily English, for example, then the base direction should be left to right, but if it's primarily Arabic, the base direction should be right to left. To avoid messing up bidirectionality, don't try to draw a sentence in little pieces; instead, put together a string first, and draw it all at once, letting the toolkit figure it out. (If you're using message files properly, of course, this will happen anyway.)

International toolkits must also support bidirectional text editing, making (for example) arrow keys and selection work in the correct direction for the script.

Automatic layout managers can also support bidirectionality. In Java, for example, FlowLayout can lay out elements either left-to-right or right-to-left, depending on the current global language setting. Similarly, GridLayout's grid coordinates can count either from the left or from the right.

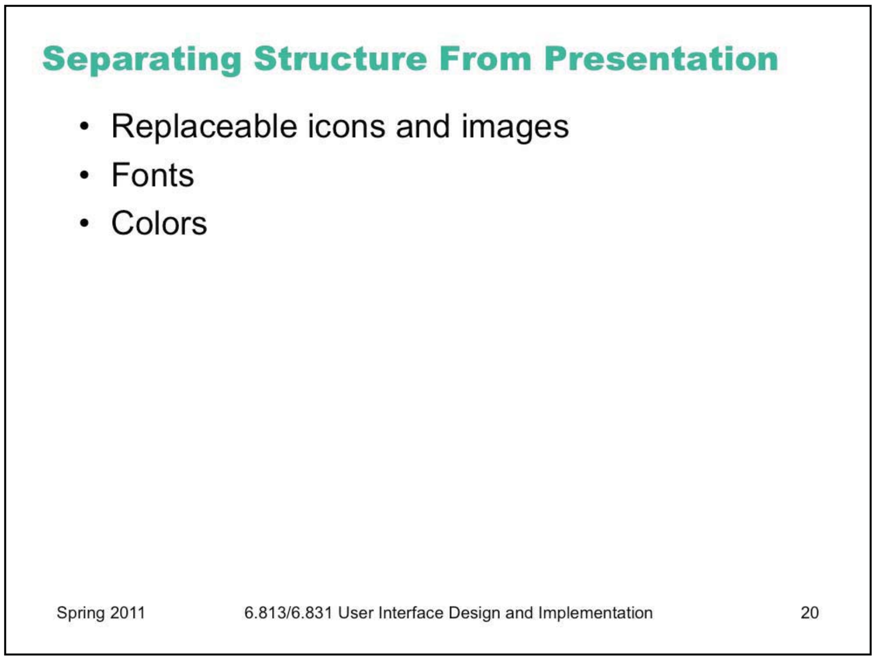


Formatting Libraries

- Library support for parsing and printing numbers, dates, currency
  - NumberFormatter
  - DateFormatter

Spring 2011 6.813/6.831 User Interface Design and Implementation 19

For handing variation in number, date, and currency formats, you should rely on libraries, rather than rolling your own parsers and formatters. Java has good library support for this, for example.

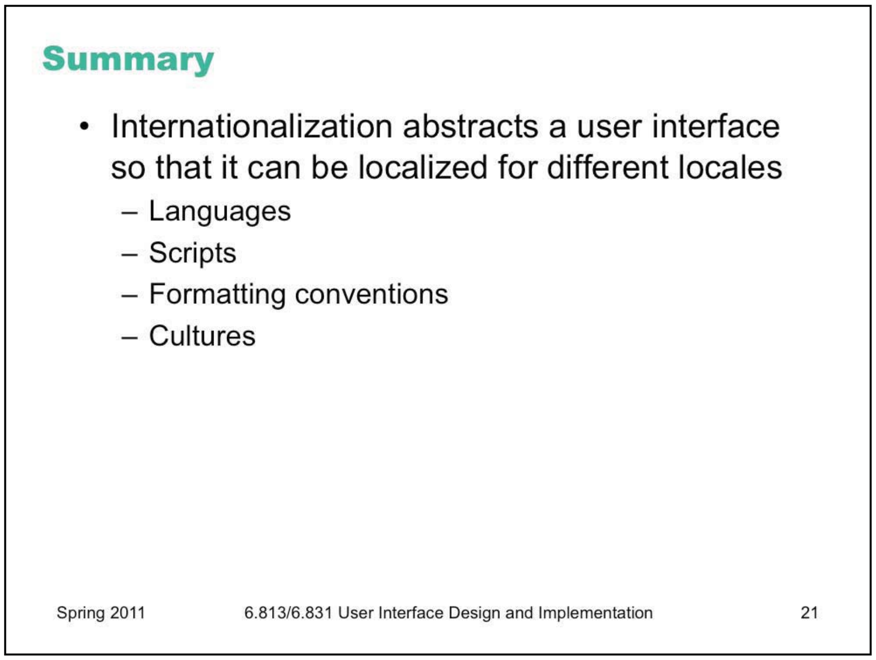


Separating Structure From Presentation

- Replaceable icons and images
- Fonts
- Colors

Finally, to handle other changes that localization might impose, it helps to isolate details of the presentation. Images and icons might need language translation (if they contain text) or cultural translation (if they use unfamiliar symbols). Fonts might need to change to handle different scripts, since fonts rarely have glyphs for every script in Unicode. And colors might need to change if they have cultural problems.

For web programming, CSS makes this kind of separation easier. In Java, resource bundles can be used.



Summary

- Internationalization abstracts a user interface so that it can be localized for different locales
  - Languages
  - Scripts
  - Formatting conventions
  - Cultures