10

Internationalization

Most of the people in the world are not North Americans or even European. Though English is the dominant language of trade, most people speak another language. Because human-computer interaction is designed to serve people, the creation of culture and language-specific user interfaces is very important. In this chapter we will not cover all of the possible ways that a culture or language can impact software design, but we will cover the major ones.

Internationalization itself is a misnomer. What we want is a *globalized* implementation for our user interface. A globalized interface is one that is prepared to be easily *localized* to a particular language and culture. A major issue in localization is adapting to the language. Not only do different languages have different words but they vary widely in how words are assembled into phrases and concepts. This can pose a challenge.

Language also impacts screen layout. For cultures that read European languages the most important information should be placed in the top and left of the screen. This is because European languages are read left to right and top to bottom. Because of the reading order this is the way European users naturally scan a screen, whether reading or not. In languages such as Arabic or Hebrew the order is right to left. Chinese is frequently written vertically. Language-imposed reading order trains literate users to scan a page in a particular way. Attention to this scan order can impact the usability of the interface. One of the advantages of variable intrinsic sized layouts is that logically organized rather than spatially organized layouts can be rapidly modified to produce new language-adapted layouts.

Language, however, is not the only issue. There is wide diversity in the expression of common concepts such as money, dates, times and numbers. The meaning of color is very culture dependent. The pervasiveness of traffic signals has defined the meaning of green (go), yellow (caution), and red(stop). One cannot necessarily extend these concepts into green being good or red being a warning. For those with cultural antecedents in Rome, purple is the imperial

color while in China it is yellow. Symbols and icons are very different in many cultures.

Globalization of an application involves *externalizing* any information that is culture or language-dependent. Externalizing information moves it out of the code and into resources. Once the information is moved into resources, culture specialists can localize the interface by modifying those resources rather than modifying the code. The architecture imposed by interface design tools (Chapter 9) greatly facilitates our ability to globalize and localize our user interfaces.

Figure 10.1 shows a list of culturally dependent items that must be externalized. Most of these can be moved into resources and then localized quite easily. The most complex problems involve language and this will be the topic for most of the chapter.

Messages and alerts	Labels on user interface components
Help text	Sounds
Colors	Icons and symbols
Dates	Times
Numbers	Currencies
Measurements	Phone numbers
Honorifics and personal titles	Postal addresses
Page layout	

Figure 10.1 – Items requiring localization

There are many complexities and subtleties involved in localizing a user interface to a particular culture. The best that can be done in this chapter is to highlight the issues, leaving the reader to work out the details for a particular application implemented in a particular graphical toolkit. The issues discussed are:

- locales and resources
- character encodings and input
- numbers, currency and measurements
- dates and times
- formatting compound strings
- sorting

Locales

Many modern programming languages come with standard libraries for supporting internationalization. Most of these revolve around the notion of a *locale*. A locale specification consists of a language and a country. Optionally a

locale can have a variant code if the language and country are not sufficiently specific. In C# locales are handled by the CultureInfo class.

Figure 10.2 shows an excerpt of the ISO 639 standard for language abbreviations and figure 10.3 shows an excerpt of the ISO 3166 standard for country abbreviations. These standard abbreviations are rarely violated although the API of most locale systems does not require that these abbreviations be used. Note also that the abbreviations are derived from that language's own term for itself, not the English word for that language. The country codes have a two letter, three letter and numeric variants. In locales the alphabetic forms are generally used.

az Azerbaijani	ba Bashkir	be Byelorussian
bg Bulgarian	bh Bihari	bi Bislama
bn Bengali; Bangla	bo Tibetan	br Breton
ca Catalan	co Corsican	cs Czech
cy Welsh	da Danish	de German

Figure 10.2 – Excerpt from ISO 639 standard language codes

Country	A2	A3	Number
AFGHANISTAN	AF	AFG	004
ALBANIA	AL	ALB	008
ALGERIA	DZ	DZA	012
AMERICAN SAMOA	AS	ASM	016
ANDORRA	AD	AND	020
ANGOLA	AO	AGO	024
ANGUILLA	AI	AIA	660

Figure 10.3 – Excerpt from ISO 3166 country codes

A locale is a grouping of resources tuned to a particular language and culture. In systems such as Java or .Net the locale also provides many of the services for handling money, numbers, dates and times. These come with a large number of predefined locales for these concepts. Figure 10.4 shows the Java code for obtaining a list of currently installed locales. This is useful if the application can dynamically change locale. The user can be presented with a list of locales and select one.

```
static public void main(String[] args)
{
      Locale list[] = DateFormat.getAvailableLocales();
      for (int i = 0; i < list.length; i++)
      {
         System.out.println(list[i].toString());
      }
}
```

Figure 10.4 – Java code to list available locales

Most systems implement locales by bundling resources and associating them with locale names. Some of the resources are the properties set by the IDT when creating the user interface. There are also other kinds of resources for messages, help text, conversion factors and string formats. Resources are grouped together in *resource bundles*. A resource bundle is a group of conceptually related resources. For example there are several resources associated with the tool bar in figure 10.5. These resources would be the icons, colors (background and highlight), tool tip text and tool tip font. We could collect them into a resource bundle called ToolBar.

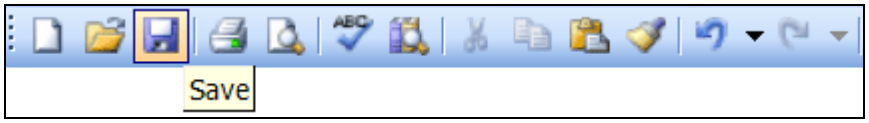

Figure 10.5 – an example tool bar

Resources are simply bindings between string names and some object. In many cases the objects themselves are text strings. For icons we could have the names of files where the icon images are stored, for colors the RGB values, and font names can also be given as text. If all of the resources are text strings a simple file like that shown in figure 10.6 would suffice to represent the resources.

```
# this is a comment for the file that provides resources to the ToolBar resource bundle
Background = 200,200,255
HighlightColor = 255, 255, 200
NewFileIcon = NewFileIcon.gif
NewFileDisabledIcon = DisabledNewFile.gif
NewFileTip = “Create New File”
. . . .
```

Figure 10.6 – Resource bundle file fragment

We could place this text into a file called ToolBar.properties (using the Java convention). To localize this user interface into French we could create a new file called ToolBar_fr.properties. In this new file we would translate all of the tooltips into French. If some of the tool tips contained words used primarily in Quebec we could create a Toolbar_fr_CA.properties file with those terms unique to French-speaking Canadians. In a Chinese or Arabic resource bundle we would use a different icon for the one containing ABC.

To obtain a bundle of resources the programmer provides the name of the resource bundle (“ToolBar”) and a locale. Most systems also provide for a current locale setting as the default. If the current locale is fr (French) and CA (Canada) then we could retrieve the ToolBar bundle and it would look for ToolBar_fr_CA.properties. Most such systems provide an inheritance mechanism so that common resources need not be repeated. When asking a bundle for the resource “NewFileIcon” it would first look in the following files in order.

- ToolBar_fr_CA.properties
- ToolBar_fr.properties
- ToolBar.properties.

For most locales the icons will remain the same, but the tool tip text will change with the various languages. Therefore the ToolBar_fr bundle would only contain definitions for tool tip text and not for icons. The ToolBar_fr_CA bundle would only contain colloquialisms found in Quebec and inherit the rest of the text from standard French.

In some cases there are resources that are not easily represented as text. Good examples are the conversions between various calendar systems. These must use algorithms to do the conversions. We would like to associate a Calendar object with each locale. Java handles this using reflection. A class can be defined for a resource bundle that extends ListResourceBundle. This class must provide the getContents() method that returns a mapping between resource names and object. When searching for a resource in the bundle ToolBar_fr_CA, Java will first look for a class (using reflection) named ToolBar_fr_CA that extends ListResourceBundle. If it does not find the class or does not find the named resource in that class it will then look in ToolBar_fr_CA.properties. The full search path for a resource then is:

- ToolBar_fr_CA.class
- ToolBar_fr_CA.properties
- ToolBar_fr.class
- ToolBar_fr.properties
- ToolBar.class
- ToolBar.properties.

Note that a resource can be anything. If it is text then placing it in a resource file will simplify the localization process. However, the reflection mechanism makes any kind of resource possible. The externalization process is one of factoring all culture-specific concepts out of the code and into resources where localization teams can define new values for particular locales.

Unicode

In the beginning of computing, everything was based on American English and character encodings defined by American computer manufacturers. The standard for text for many years has been ASCII (American Standard Code for Information Interchange). By its very name it is obviously not very international. The reason for its wide usage has been the dominance of the US computing industry and the fact that historically programming languages are ASCII-only. This has dictated that ASCII be the language of software. However, ASCII can never successfully be the language of commerce. There are too many cultures and peoples involved.

There have been a variety of encodings for international character sets. Most of them have been dominated by the desire to conserve disk and memory space. With the advent of cheap RAM and disk space, the size of text files and strings has become irrelevant. Fundamentally the question revolves around how many bits can be devoted to a single character. ASCII and its Latin extension (to include European accents, umlauts and other marks) fit a character into 1 byte, which is very convenient. The Unicode[1] standard broke that barrier by allocating 16 and then 32 bits per character. The Unicode Standard, Version 4.0 contains definitions for 96,447 characters including all European, Middle Eastern and many Asian scripts. Virtually all of the commonly used scripts fit in the first 64K code points and thus fit in 2 bytes. The Unicode Standard refers to this as the Basic Multilingual Plane (BMP). Conveniently the ASCII code occupies the first 128 code points and therefore the ASCII value and the Unicode value for a character are identical.

In Unicode each writing system is allocated a range of values for their characters. Figure 10.7 shows part of the definition for Arabic.

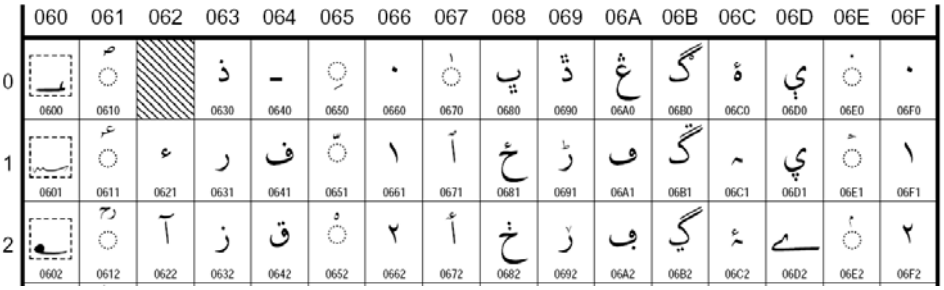

Figure 10.7 – Excerpt of Unicode for Arabic

There are several ways for encoding Unicode in strings and in files. The UTF-8 encoding focuses on the dominance of ASCII and preserving file size. Because ASCII occupies only 128 code points (7 bits) the high order bit is available. With ASCII characters the high order bit is always zero. In UTF-8 a high order bit of 1 indicates that some other character ranges are being specified. The UTF-8 encoding uses from 1 to 4 bytes to encode any of the Unicode characters. The advantage of UTF-8 is that ASCII files automatically conform without modification. The disadvantage is the variable length of the character codes. HTML generally uses UTF-8.

The Unicode BMP occupies only 16 bits. Based on this the UTF-16 encoding allocates virtually all of the BMP into 16 bits with special codes to indicate when an extra 16 bit word is required to encode the remainder of the 95K+ characters in Unicode. Java supports only the first two bytes of the UTF-16 encoding. This guarantees that every character is exactly 16 bits long. It does not encode the entire Unicode standard but does handle all of the BMP. The 16 bit restriction allows for a wide degree of internationalization without incurring the computational costs of variable length character sets.

There is also a UTF-32 encoding that requires exactly 4 bytes per character. This encodes the entire Unicode standard in a fixed character size, but uses four times as much space as ASCII.

Working with Unicode Characters

There are many basic operations on characters that we regularly use in ASCII. For example, in ASCII, white space is any character less than or equal to a space. This is only true in European languages. The concepts of white space, word breaks, sentences and punctuation vary among writing systems. The Java `Character` class and the C# `Char` class provide methods to perform a variety of Unicode compliant tests, including:

- isDigit()
- isLetter()
- isLetterOrDigit()
- isLowerCase()
- isUpperCase()
- isSpace().

Using these methods will produce reliable international results that are way too cumbersome to maintain on your own.

A major problem when interacting with characters is their direction of writing. European languages are left to right, top to bottom. Many languages, such as Arabic and Hebrew are right to left, top to bottom. Some Asian languages read top to bottom, right to left. These issues make international text editing somewhat problematic. Even more complicating are bidirectional languages. Arabic and Hebrew both write their numbers left to right rather than right to left. Such mixed directionality makes it quite complicated. Apple has attempted to address this problem with their Text Services Manager that encodes all of the editing of text so as to provide a common implementation of international editing and display. There is not room here to do more than note the problem.

Fonts

The Unicode standard only associates numbers with characters. A font associates character codes with *glyphs*. A glyph is a picture of a character to be drawn on the screen. Though Java, C# and most web browsers have Unicode support built into them, most users do not have fonts to display all possible Unicode characters. Without an appropriate font the character can be stored, but not seen appropriately.

Character Input

Interaction, of course, involves not only presentation but input. In phonetic languages such as English, French or Russian there are a small number of characters and a keyboard is easily designed. At the system level a keyboard produces a key-code that associates a unique number with each key. The key-codes are translated into characters in coordination with various shift keys. To move from one language to another the keyboard key caps are changed to show the new letters and the translation from key-code to letter is changed.

The input problem is more serious when using ideographic writing systems. Kanji, for example, can contain up to 50,000 symbols. Obviously one key per character is not going to work. There are a variety of typing techniques for such writing systems. The Tsangchi system for Chinese characters uses a keyboard of 24 symbols that can be composed together to form a single ideographic character. In Japanese there are two other writing phonetic systems, Hiragana and Katakana, that each have 46 characters. Some text entry systems have users type using the phonetic system and then translate into the ideographic characters.

Numbers, Currency and Measurements

Fortunately most of the world's commerce represents numbers using the basic ten digits. These are referred to as "Arabic numbers" because they were designed by Arab scholars. However, they are not the symbols used in the Arabic language. Virtually every language has alternate words for the numbers and most writing systems have their own symbols for the ten digits. However, in the world of commerce, the familiar ten is almost universally used.

The punctuation of numbers, however, varies among cultures. In the U.S. we write "95,241.23" whereas in Europe the same number would be written "95.241,23". The number system is the same, but the punctuation is different. Most software libraries that support internationalization provide a class for formatting numbers. The locale object will yield a number formatter object that has methods appropriate for that locale. Many systems offer special custom formatting of numbers.

The uniformity of base-10 numbers throughout the world of commerce simplifies our internationalization problem. This uniformity breaks down when we look at speech interfaces. People rarely speak in digits. We generally have language forms for numbers such as "three thousand two hundred fifty four point seven five." These language forms for numbers vary widely with language and must be accounted for in spoken language interfaces.

Currency

Currency is a common number form that is very important in international computing. Again the pressures of international commerce have already assisted us in localizing our interfaces. The international monetary system requires that money be quoted as a decimal number for some unit of currency. In the U.S. this is dollars, in Japan it is yen, in Europe it is euros and in the U.K. it is pounds. The international monetary system does not recognize nickels, shillings, guineas, pence, or pieces of eight. Though a particular culture may use various locally significant pieces of money, when currency is quoted on virtually all computers it is a decimal number quoted in a particular unit of currency.

Many cultures have a specific symbol for their currency such as $ (dollars), £ (pounds), € (euros), or ¥ (yen). These symbols can become rather confusing when there are many currencies involved. This is also a problem when multiple countries have their own monetary system yet share the same name for their currency. To resolve this there are standard three character symbols for particular

currencies. For example $ could be replaced by USD (U.S. dollars), CAD (Canadian dollars), or AUD (Australian dollars).

Conversion among currencies is simply a matter of multiplying by a conversion factor. These conversion factors are not constant, due to fluctuations in international markets. The conversion factor is also slightly different depending on the direction the money is flowing. This is due to markets moving money in and out of currencies and the handling charges of those providing the exchange. Thus currency conversion is generally handled by special-purpose software. User interfaces only format the amount correctly and let other software manage the conversion process.

Measurements

Measurements are another specialized form of number that varies among cultures. Many numbers such as American and European clothing sizes are not numerically convertible. For our user interface, these are not a problem because they are directly associated with what is being sold. We need only quote them as text.

Other measurements are convertible and those conversions are constant. Thus our user interface can readily support a variety of such measurements. As with currency, most measurement systems can be represented as a single unit. The metric system for length uses the meter as its basic unit. The English system (not used by the English) uses feet. There is a constant conversion factor between feet and meters of approximately 0.3048. The simple multiplier conversion works well with three exceptions. The first is for measurement systems for which no standard zero point was known as in temperature. Until absolute zero was discovered, the Fahrenheit and Celsius systems arbitrarily picked a zero point. For such conversions a multiply and a constant offset will handle the conversion. There are some measurement systems, such as the Richter scale for earthquake strength, that are logarithmic rather than linear.

An additional complication is colloquial fractional units. Prime examples are feet and inches for length and pounds and ounces for weight. In many such systems the sub-units are not decimal multiples but use other units such as 12 inches to the foot, 3 feet to the yard, 16 ounces to the pound and 2,000 pounds to the ton. Fortunately most of the world has adopted the metric system for most measures. This eliminates the need for conversions and greatly simplifies everything for computer applications. However, the world’s largest economy (U.S.) is still mired in a colloquial system already abandoned by its creators (the English). Fortunately most colloquial systems have simple numeric conversions

among their subunits that are not difficult to program. The lack of difficulty to program them has not lead to standard libraries for their conversion because there is no simple straightforward conversion system that unifies them all. Heavy sigh.

Date and Time

Dates and times are very important units because people in technically developed societies organize their lives around them. Dates and times are simply different units for the same measure of time since some starting point. We commonly think about dates and times as being different, but this is only when we consider our day time activities. An activity may start at 11 PM today and end at 1 AM tomorrow. We cannot exclude dates from the specification of these times.

The standard international unit of time is the second, with smaller units such as milliseconds, microseconds and nanoseconds being decimal fractions of a second. Larger units unfortunately use 60 seconds to the minute, 60 minutes to the hour and 24 hours to the day (approximately). We also need to differentiate between times and durations. Seconds are actually a unit of duration or the amount of time that has passed between two events. When we ask "what time is it" the answer is quoted in the duration since some fixed point. When we quote a time of 10:30 PM we mean 22 hours and 30 minutes past midnight today. There then are the questions of midnight where and it is political midnight or astronomical midnight?

For computing use, time and dates are generally stored as the number of milliseconds since some epoch. There are variations on when the epoch starts. Many computers use 1960 since no good time-keeping computers existed before then. Others use 1900 because it naturally subsumes the 20th century and beyond. The Macintosh used 1904 as its epoch base. Whatever the epoch used, the conversion among them is trivial and not part of the user interface. There is also the question of where on the earth that epoch started. Because of England's domination of the sea during the great years of exploration, the base location is the Royal Observatory at Greenwich, England. Originally this was called Greenwich Mean Time (GMT). To break its cultural association it is now called Universal Time Coordinated (UTC). Because the solar day is not absolutely 24x60x60 seconds long, systems of leap seconds have been created to keep time synchronized with the sun.

Time zones are represented as offsets from UTC. Most time zones are defined as constant hour offsets, but there are other offsets. Newfoundland is

offset on a 30 minute boundary and Nepal is on a 45 minute boundary. The simple answer is that UTC time can be converted to or from any other time zone by adding a constant number of minutes.

More recently the Swiss watch company, Swatch, has created Internet time[2]. This divides the solar day into 1000 "beats" and uses Biel, Switzerland (the headquarters of Swatch) as the base meridian. This defines a universal time all over the world that is absolutely synchronized with the sun.

The textual representation of the time of day varies from culture to culture. The two common variants are AM/PM and the 24 hour clock. There are also variations in punctuation. A locale object can provide the appropriate formatting for a culture.

Calendars

Calendars are very complicated things. Most calendars were created to represent the order of astronomical events. Years are based on revolutions around the sun, months on revolutions of the moon and days on rotations of the earth. Unfortunately for computer software none of these revolutions are integer multiples of each other. Calendars based on days get out of synch with years, thus the introduction of leap years which add days to the year periodically. Lunar months are radically out of synch with years. This leads to the modern European calendar with months of varying lengths or traditional Hebrew calendars that periodically add a 13^{th} month to resynchronize the months with the year.

Calendars are very important culturally. Because observance of the heavens is a part of so many religions, their religious observances are strongly related to astronomical periods and events. This religious connection leads to strong cultural connections to calendars. Various cultures also define quite different zero points for their calendars. The Christian, Hebrew, Arabic, Chinese and Japanese years are quite different because of the event on which they base the start of their counting.

In many cultures work is organized around weeks. The combination of Jewish, Christian and Islamic cultures have established the seven day week. However, the French Republican Calendar established the 10 day week. This was discarded by Napoleon because nobody wanted to work 9 days without a day off.

Internationally the Gregorian calendar (decreed by Pope Gregory XIII in 1582) is the common form for communication. The Gregorian calendar is a correction of the Julian, established by Julius Caesar in 45 BC (note the Christian reference BC to a Roman calendar). The Gregorian calendar, like the metric

system, provides a common basis for international trade. The Gregorian calendar's strong religious and cultural basis does not make it the common calendar of everyday life for much of the world's populations. Even where the Gregorian calendar is the cultural norm, each language has its own words for the days of the week and months of the year.

In dealing with dates in localized software it is appropriate to use the Gregorian calendar in most situations provided the locale is used to translate the names to the appropriate language. Some systems provide a Calendar class for modeling other calendars, handling date conversions to and from those calendars and formatting date representations appropriately. There does not yet seem to be any standard for such facilities.

Compound String Formatting

The most difficult problem when localizing a piece of software is the compound string. Suppose for example there were a small data base with the names of people and the number of apples that they each bought. We might write a program that generates the message "Mary bought 4 apples", where Mary and 4 are values drawn from the database. Many web-based systems do exactly this when translating data into web pages. There is first the obvious problem that "bought" and "apples" need to be changed when moving to a new language. Suppose, however, that the data base has the value 1 for George. The appropriate string would be "George bought 1 apple". Notice that "apples" has changed to "apple" to create a correct sentence in English.

A naïve approach to the compound string problem is to create a "bought" resource and an "apples" resource in a resource bundle. The code might then read.

```
message=name+getResource("bought")+number+getResource("apples");
```

The resources would handle the translation. Our example for George showed that even in English the output is awkward. In many languages there are more complex number and gender associations between words. There are also wide variations in word order. The problem is to dynamically generate a phrase that is semantically consistent with the target language.

One technique that is used to resolve this problem is the macro resource. We might create a resource "AppleSales" that contains the string

"<name> bought <number> apples"

The localization library would provide a mechanism for generating a full string by substituting the name and number into the resource string. The advantage of this approach is that when localizing the message the entire phrase can be translated rather than just words at a time. Localization libraries also provide mechanisms for selecting different translations depending on the numbers involved. Even with these facilities it can be hard to get correct message generation that works in all languages. The message macro resource technique does provide a way to generate understandable messages in most languages even if the syntax is slightly awkward in some cases.

Sorting

Sorting is a very important component of a user interface because it is the primary mechanism for people to search through long lists. Most computer science students learned that sorting is done by comparing characters and putting them in numerical order based on their ASCII code. This is great for teaching a first programming course, but it fails even for English because of upper and lower case letters. The additional characters from the ISO Latin encoding for European characters are completely out of their sort order and Unicode is only mildly based on an appropriate sort order. The complexity of sort order is much worse in ideographic languages such as Chinese.

For user interface development the answer is the Collator class. This can be obtained from a locale object and provides a method for comparing two Unicode strings to establish their sort order. Trying to encode sort order on your own is an exercise in pain. In those cases where there is no available locale specification most Collator classes provide a mechanism for adding explicit rules for character precedence. This can adapt your sort order to a new language.

Summary

Internationalizing an interface involves designing it so that it can be easily localized to a particular language and culture. The primary mechanism for doing this is to externalize information into resources and resource bundles. These pull culture-specific information out of the code and into resources where they can be easily edited by language/culture experts. Careful attention must be paid to:

- Character sets
- Icons
- Colors
- Messages both simple and compound
- Numbers
- Dates and times

There are other subtle complexities that may arise in particular cultures. It is always a tradeoff between localization effort and expected return from more careful attention to a particular culture.

Exercises

1. What is the difference between internationalization, globalization and localization?
2. Take a small user interface and do the following:
    a. Identify a set of resource bundles in which the parts of the UI needing globalization can be grouped.
    b. Identify all of the resources that would go into each bundle.
    c. Describe how you would efficiently organize the locales for USA, France, English Canada and Quebec so that duplication of resources is minimized.
3. Describe how you would organize your code so that the user can select the locale that they want to use.
4. Why is sorting Unicode characters a problem?
5. Why are simple resource files not sufficient for taking care of date issues?
6. Why is it so hard to internationalize compound strings?

---

[1] www.unicode.org

[2] http://www.timeanddate.com/time/internettime.html