INTERNATIONALIZATION BEST PRACTICES FOR NUMERIC FORMATTING

Often times when people think of internationalization (i18n), they limit their scope of thought to linguistic aspects such as string externalization and concatenation, searching, and sorting. These aspects may represent the bulk of the i18n work; however, dealing with numeric formatting issues can also be very important and sometimes cause errors that silently go undetected.

One best practice for handling numeric data is also a general purpose best practice across all software development: keep data in its native format as long as possible and only convert to other types when necessary, at the outer boundaries of the application. For example, use numeric data types such as double, int, Decimal, Float, Long, etc., to hold numeric data and use them throughout your application processing. Only convert these values to character strings when performing operations such as displaying values in a user interface, accepting input from a user, or passing data to a system interface that requires string values. Never pass locale-formatted data across a subsystem boundary unless it is really needed!

Native numeric data types such as double, int, Decimal, etc., store their data in a locale neutral format, so every consumer of that data knows exactly what the value is. Numeric formatting varies internationally, so the logical value -1234.56 could be represented as a number of different strings depending on the locale of the end user. Some examples of formatted strings are:

- -1,234.56
- -1.234,56
- (1,234.56)
- -1 234.56
- -1'234.56

Many external system libraries or internal libraries provide support that help format and parse strings in these localized formats. However, due to the complexity and variation of formats, some are overly lenient in their attempt to parse and recognize the actual value represented. For example, some tools simply look for grouping (thousands) separators and decimal separators without trying to validate positioning. If an application is executing with a locale that uses a comma as a grouping separator and a user from a different locale accidentally inputs a value string of 1,23 (representing one and 23 hundredths in most European locales), the application may assume this is the value 1230 (one thousand two hundred and thirty) and silently proceed without raising an exception or generating an error.

This is particularly important in applications that support multiple locales, such as localized client applications or web pages communicating with the same server code. If an application carries numeric data values around in string variables, then it not only greatly increases the chance that a value is misinterpreted, but it also must retain the locale the numeric value was meant for throughout the application, so the value can be interpreted correctly. Whereas, converting the data to a numeric data type as soon as the value is entered allows the rest of the application to understand the value without question and without caring what locale the end user was using.

In some cases, especially service or system interfaces, numeric data is explicitly represented as a string. This allows two communicating processes to interact with each other without having to know the architecture and internal representation of numeric data types of the other process. In these cases, the best practice is to always use a standardized locale-neutral numeric format for the strings being passed. For example, the W3C definition of the XML decimal data type uses the same sign characters (+ or -) which are always in the beginning of the string when specified, uses no grouping separator, and always uses the same decimal character (period) without regard to locale. So, the data value can be understood in any locale.

This does require some extra work to convert between locale neutral and locale specific format each time the service boundary is exercised; however, it eliminates ambiguity, reduces the potential for misinterpretation, and allows each side of the interface to execute independently without regard to the locale of the other side of the interface.

Using native numeric data types, as discussed previously, complements the use of standardized locale-neutral strings for service and system interfaces since it is very easy to convert a native numeric data type such as double to and from a standardized locale-neutral string. These types of conversion functions are included in many development tools.

In conclusion, a best practice end-to-end application uses locale specific strings at the outer, end-user facing edge of the application, converts to native numeric data types as soon as possible, uses the native numeric data types for business logic and as many service and system interfaces as possible (such as a database interface), and converts native numeric data types to a standardized locale-neutral string format for other service and system interfaces that require this format.

Corporate Headquarters

Lionbridge
1050 Winter Street
Waltham, MA 02451
USA
www.lionbridge.com