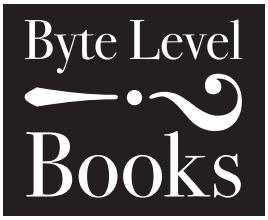
Going Global with JavaScript & Globalize.js

Jukka K. Korpela

Going Global with JavaScript and Globalize.js

Jukka K. Korpela



www.ByteLevelBooks.com

Going Global with JavaScript and Globalize.js
Jukka K. Korpela

Publisher
Byte Level Books
www.bytelevelbooks.com

Copyright © 2011 Jukka K. Korpela
All rights reserved.

ISBN 978-1-61822-000-4

All trademarks, screen shots, and logos are the property of their respective owners. No part of this publication may be reproduced in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise) without the prior written permission of Byte Level Books.

Contents

Introduction **5**
Common globalization challenges in JavaScript 7
Who this book is for 8
What you need to know 8
How to use this book 8
Code downloads 9
Conventions used in this book 9
To keep up-to-date 9
Acknowledgments 9
Localization is about people 10

Part I: Globalization, Localization, and JavaScript **11**

Globalization and JavaScript **13**
JavaScript as part of the problem 14
JavaScript as part of a solution 16
Globalization challenges in JavaScript 17

Hands-on: JavaScript and Dates **19**
The not-so-simple approach to dates 19
Globalize to the rescue 21

HTML and Emerging Opportunities (and Challenges) **24**

Localization and Libraries **28**
Localization scope: Which features should we localize? 28
Localization scale: From specific to universal 31
Localization on the server-side vs. client-side 38
Using libraries 40
Identifying locales 42
The Common Locale Data Repository (CLDR) 44

Part II: Going Global with Globalize **49**

Using the Globalize Framework **51**
What is Globalize? 51
Installing Globalize 51

The Globalize approach 52
Selecting a locale 53

Hands-on with Globalize: Formatting Dates and Times 56
Formatting codes recognized by Globalize.format 56
Notes on formatting codes 58
12 h vs. 24 h time 59
Localizing for time zone 60
The neutral ISO 8601 notation 62
Date and time in four formats 64
The Internet date format 65
Cultural relativity in dates 65
Notations and calendars 66
Cultures with multiple calendars 67
Date format as a personal preference 70
Summary of culture objects in Globalize 71
Generating “Last modified” information 74
Reading dates 75
Coding it yourself 77

Hands-on with Globalize: Numbers 78
The problems with formatting numbers 78
Using Globalize 79
Exponential notation 82
Using a real minus 84
Ordinal numbers 85
Special values and errors 85
Localized number systems 87

Advanced Use of Globalize 88
Parsing data with Globalize 88
String localization in Globalize 90
Modifying and adding locales 91
Setting up a generic localizer function 97
Example: A localized input form 99
The new <input> elements in HTML5 102

Part III: Managing Languages, Cultures, and Characters .. 105

Making It Globalization Ready 107

Translating Strings Successfully 109
Identifying text for translation 109
Exporting strings for translation 110

Identifying Languages 112
Language negotiation and its limitations 115

Managing Languages 120
Deducing the language of a web page 120
Deducing the language of an element 121
Passing language parameters to web pages 122
Changing the language of an element 124
Size requirements 125
Hyphenation and word division 125
Directionality 127

Punctuation Marks 131
Quotations 132

Programmatic Generation of Text 134
The problem with plurals 134
From localization by pattern to localized patterns 135

Embedding Machine Translation 138
A translator gadget for dynamic text 138
A translator gadget for user input 140

Typography 144
Fonts 144
Ligatures 144

Adapting to Local and Cultural Conventions 147
Monetary values 147
Physical quantities 148
Telephone numbers 150
Other identifying numbers 151
Personal names 153
Postal addresses 154
Graphic symbols 155
Colors 155
List notations 157

User-driven Adaptations 159
Geolocation 159
Accessibility 160
Personal preferences 161

Managing Characters and Strings 162
Legacy encodings 162
Unicode 165
Character escapes 167
Character input 168
Supporting native characters vs. falling back to ASCII 169
Rejecting what you cannot handle 170
Regular expressions: The ASCII legacy 172

Using Unicode character properties 173
Understanding the string type 174
BMP, the “16-bit subrange,” and beyond 175
Private use characters 178
Sanity checks 179
String comparisons and sorting 180
Case conversions 180
Normalization 182

Error Handling and Testing 184
Minimizing error messages 184
Always check return values 185
Using try and catch 185
Testing localization 187

Appendix: Localizing Identifiers 190
About the author 195
About Byte Level Books 197

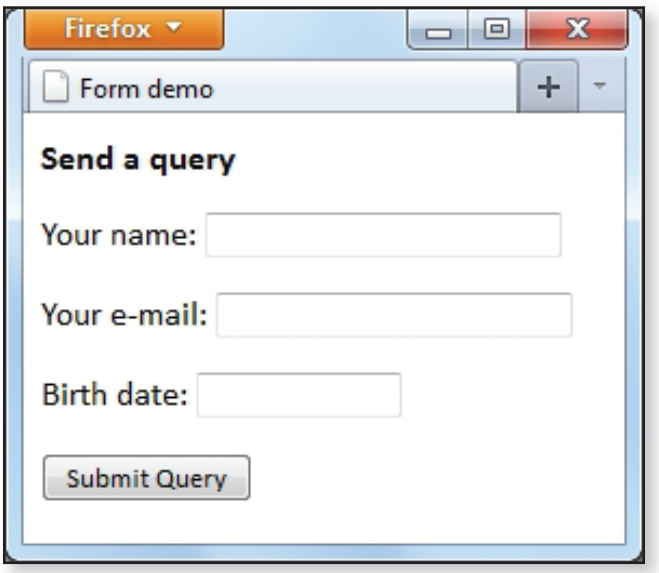
Introduction

Once viewed as a simple scripting language, JavaScript has evolved over the years into a robust programming language. Today, JavaScript powers many of the world's most popular and interactive websites and applications. From Google to Twitter to Amazon, it's hard to find a major website today that does not make use of JavaScript in some shape or form.

Yet despite the ubiquity of JavaScript, the language is far from perfect. JavaScript wasn't written with the world's many human languages and cultural standards in mind. And although a number of libraries have been developed to extend JavaScript to people around the world, there is a dearth of documentation on how to make best use of these libraries.

That's where this book fits in.

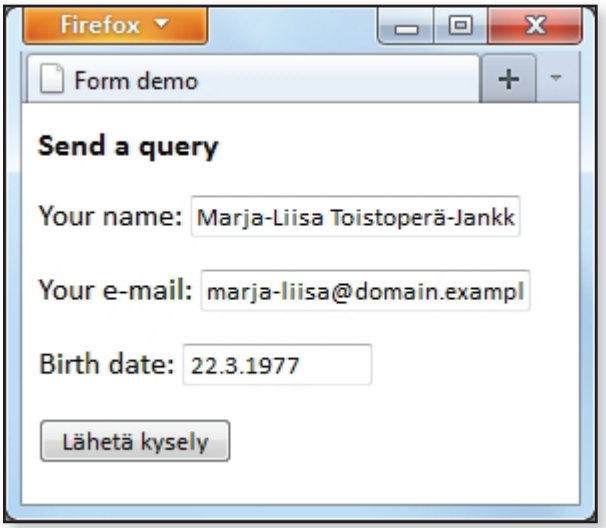
To give you a brief idea of the problems addressed in this book, consider the following input form:



A basic user input form.

If you live in the United States and you speak English, you would likely have little trouble completing this form.

But let's assume you do not live in the United States and you do not speak English. As you can see below, a Finnish web user might find this form to be unnecessarily frustrating:



Each input field presents globalization challenges.

Here are the global challenges with this form:

- The visible width of the field for the *personal name* is too short for the user's name. The length has been defaulted (there is no size attribute in the input element). The common browser default is 20, which is too short for the names of people in many countries.
- The *email* field is too narrow for similar reasons.
- There is no information about the expected *date format*, so a user is likely to enter the date in a format that is most natural to her—and this might result in an error message, or in the wrong interpretation of the data.
- In the submit button, the "Submit Query" text aligns with the default language of the user's browser, in this case Finnish. There is no hard-coded text as this is a feature native to the browser. While someone viewing, say, an English-language web page on a Finnish version of a browser might be expected to understand both languages, this button still looks *odd*—perhaps even suspicious. Applications and web pages that contain an illogical mix of languages are often the result of poorly globalized projects.

This example addresses the challenges of adapting an input form for just one addi-

Globalization and JavaScript

This section introduces you to key globalization concepts and the role that JavaScript plays in enabling support for multilingual applications.

But before we go any further, it's important to understand how software globalization works and, more important, how global developers talk. Here are the most important terms you should know:

- **Globalization** refers—in this book at least—to localization via internationalization. That is, you first must internationalize your code before you localize it. Globalization does not necessarily mean you are out to cover the globe with your software. In fact, you can "globalize" software without targeting users outside of a given country. The practice of globalization is just as relevant if all you are doing is developing software to support English and Spanish speakers *within* the US.

- **Internationalization** means making things as *neutral and stable* as possible with respect to languages and cultural conventions. During the internationalization stage, you remove language and cultural dependencies from your code. You also identify all user-facing text strings that will require translation and optimize the strings so they can be efficiently and accurately translated. And you internationalize the data to the fullest extent possible. For example, a date format like 2011-04-28 is internationalized (and specifically follows the international standard ISO 8601); though this date format may mean little to users, that's okay because there is still a localization stage ahead. The goal of internationalization is to create code that is "world ready."

- **Localization** means making things as *specific* with respect to languages and cultural conventions as possible, trying to make things look natural and familiar. This means using *different* presentations for different people. Examples of localized data formats: "April 28, 2011," "28.4.2011," "28 авриля 2011 г." Translation also occurs during the localization stage.

- **Locale** refers to a collection of settings or preferences to be used in localization. A locale is often described in terms of a language and country pair, such

as “English US” or “German Austria,” and denoted by language and country codes: “en-US” and “de-AT”.

- **Culture**, in the context of the Globalize library, is the same thing as locale. Microsoft favors “culture” over “locale,” and because Globalize was originally developed by Microsoft (following .NET standards), you’ll see “culture” used frequently.

The line between internationalization and localization is a blurry one. In a small company, a developer might manage both tasks, while in a larger company there will be engineers focused purely on internationalization (creating “world-ready” code) while others are devoted purely to creating local products.

Finally, as you spend more time in this field you should expect to see some rather odd-looking abbreviations in which a number is used to indicate the number of letters between the first and last letter:

- **i18n** stands for internationalization.
- **l10n** stands for localization.
- **g11n** stands for globalization.

Technically, these words are known as numeronyms, and it’s a bit of a mystery why they remain so popular within the globalization industry.

JavaScript as part of the problem

JavaScript was designed to improve the user experience, not make it worse. Yet developers must be keenly aware of JavaScript’s limitations when working with different languages and cultures. These limitations can unintentionally result in negative user experiences.

Let us consider the basic task of validating a user name, say, on an input form before that name is sent as a query to a database on the server. If we assume that a person’s name is typically something like “Jane Doe” or “Joe Q. Public,” we might write a regular expression that matches two or three space-separated strings of letters, where the second string may have a period after it. This would mean something like the following:

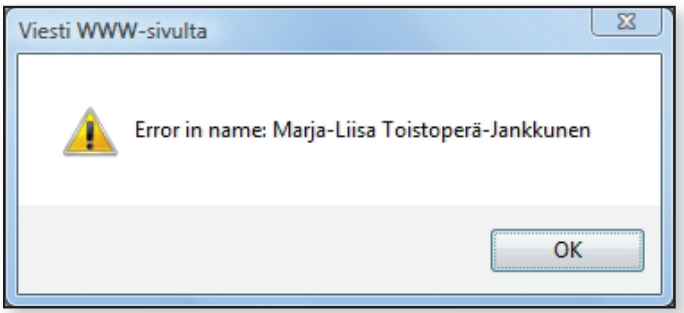
```
<script>
function nameOK(elem) {
  var ok = elem.value.match(/^\w+\s(\w+(\.?)\s)?\w+$/);
  if(!ok) {
    alert('Error in name: ' + elem.value);
```

```
    }
    return ok;
}
</script>
<input name=name onblur="return nameOK(this)">
```

However, this code would reject the name: *Marja-Liisa Toistoperä-Jankkunen*.

This is because the notation `\w` in a JavaScript regular expression matches a "word" character in a limited sense only: letters A to Z and a to z, not the letter "ä" nor the hyphen (-).

Similarly, the check would reject names like *Åberg*, *Buñuel*, and *Strauß*, not to mention names in Greek or Chinese characters. A personal name may contain a diverse mix of characters, or at least more than many developers anticipate.



A name is rejected for no apparent reason and with no explanation.

Now let's focus on the error message above. Notice how the error message is in English while the window title is in Finnish; the text "Viesti WWW-sivulta" translates to "Message from a WWW page." In general, texts that are part of the web browser's user interface remain outside the control of a JavaScript developer, which, in this case, results in a potentially confusing mix of languages.

Even if personal names could be restricted to names used in one country, it is generally impossible to check their syntax in JavaScript. People's names can be in different languages and may thus have characters other than those of the majority language.

If you're not confident that you can get client-side analysis of a text string to function correctly, don't do it at all. Getting it right means no false negatives (no rejection of valid data). False positives are also something to avoid, though they are typically less serious. For example, if you are afraid of allowing an apostrophe in a name because of possible cross-site scripting (XSS) attacks, you should consider what it means to reject a fairly normal name like O'Connor. As a real name, it should be accepted, of course; as a user ID, it's up to a policy decision. What's even worse is the possibility that the person's name exists within the server database, but because of the client-

side error, the user is left thinking she made a mistake. In this example, JavaScript becomes part of the problem because the developer didn't consider the wide range of characters that would need to be validated as part of a person's name.

This problem isn't unique to user names. Even a string analysis that might be regarded as largely technical may have hidden dependencies on a particular language or culture. For instances, there is still JavaScript code around that checks email addresses for formal correctness, blocking any addresses that have a top-level domain in excess of three ASCII characters. Such code would reject valid addresses that end with ".museum" or ".aero" as well as non-Latin internationalized domain names (IDNs), such as рф (Russia) and 中国 (China). Although IDNs are still only just emerging around the world, JavaScript code will need to be modified in the months and years ahead to permit these domains as valid.

JavaScript as part of a solution

Now that you have a taste for how JavaScript can unintentionally result in a negative user experience, let's focus now on how JavaScript can intentionally be used to avoid these problems in the first place.

Proactively, JavaScript may be used to:

- Localize the length and validation of input fields based on the user's language.
- Parse date and time notations based on the user's locale.
- Modify the UI elements, such as the input button, so it displays messages in the same language as the language of the web page.

These fixes are best accomplished when generating the web page. This typically means relying on a technology other than JavaScript—PHP, ASP, CGI, or whatever you use server-side. If you use server-side JavaScript, then the solutions presented in this book, with mainly client-side JavaScript in mind, should mostly be usable with obvious modifications.

If server-side localization is impossible, then you might use client-side JavaScript as a partial solution. For example, if the server-side form handler requires American date notation like 5/22/1988, then you can at least use JavaScript to enforce on the client-side that such a notation is used. In addition, the code can notify the user of this format limitation and perhaps even apologize for the inconvenience. Effectively localized software applications aren't always perfectly localized in the eyes of the user, but they do an excellent job of managing user expectations.

JavaScript is essential in handling errors and checking input data on the client-side.

It is also needed for the functionality of stand-alone pages like HTML documents containing applications to be used without any server.

You may also use JavaScript to implement customizability so that users can locally change the notation of dates to match their personal preferences. Not everyone prefers the "official" notation for their given language or country. It is preferable to make such changes without any connection to a server, though server-side code might be used as backup solution.

The key here is to be proactive with JavaScript to better understand user preferences and also to let users be more proactive in letting you know what language and other settings they prefer.

Globalization challenges in JavaScript

When it comes to supporting the world's languages and locales, JavaScript offers a highly uneven degree of support. In some cases, such as its support for Unicode, JavaScript does an excellent job of supporting websites and software across different languages. But in other instances, JavaScript is still lacking. For example:

- JavaScript has a fairly large character repertoire. You can use almost any kind of character in literals and operate on it. However, the built-in functions for working with characters are very limited. For example, there is no built-in way to determine whether a character is a letter.
- Regular expressions are strongly oriented toward dealing with a very limited character repertoire. For example, `\w` matches a "word" character in a primitive sense, effectively limited to English texts.
- The `toString()` methods are non-localized and produce string presentations of data in a computerish way, such as number formats like 1.23e9. Similarly, functions for parsing numbers and data are non-localized.
- Automatic conversions may generate non-localized presentations. Due to their implicit nature, the problems of statements like `document.write(myDate)` are not that easy to notice, until you get some experience in recognizing localization problems.
- There are some specifically locale-dependent functions, such as `Date.toLocaleString()` and `String.localeCompare()`. However, they can be a problem rather than part of a solution, as they relate to the locale of the implementation (such as a web browser), and there is no way to make them work according to an author-specified locale, such as the language of a web page.

- Case mappings are partly globalization ready; `String.toUpperCase` and `String.toLowerCase` manipulate letters in general, not just English letters, but implementations do not handle all aspects properly.
- System messages appear in a fixed language, though the language can be different in different implementations. There is no direct way to customize the messages, but you can catch exceptions and have your own code that overrides the system handlers and messages.
- There is no built-in support for things like making strings vary by language. However, the object concept in JavaScript is quite helpful in setting up such support.

Thus, there is little in JavaScript you can directly use in localization, but there are useful building blocks. This implies the need for general-purpose libraries (or modules, or frameworks, or whatever you prefer to call them).

Until recently, there was an extension to jQuery known as jQuery Global, which offered an extensive library of locale-specific data and formats. This library has since been delinked from jQuery and is now a stand-alone library known as Globalize, which we'll study in great detail in Part II.

To get a feel for Globalize and the inherent challenges of JavaScript in regards to globalization, let's take a look at how one would manage dates.

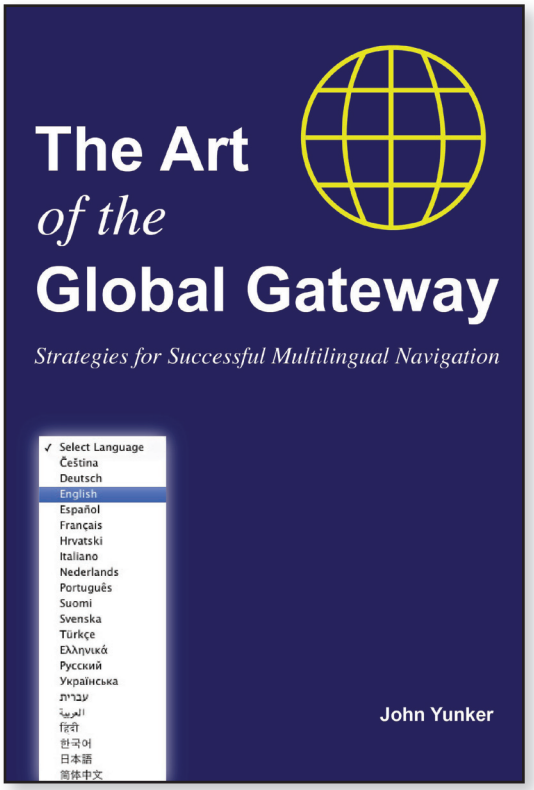
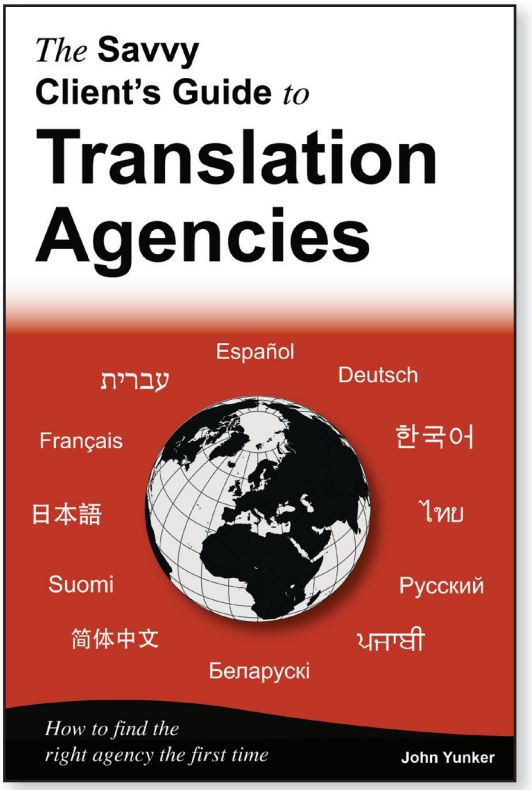
About the author

Jukka K. Korpela is a consultant who specializes in character codes, localization, orthography, usability, and accessibility, and has been actively promoting web technologies since the web was born. With vast experience in programming languages and compilers as well as in human languages, he knows how to combine information technology with the linguistic and cultural considerations needed in building user interfaces. A regular contributor to standardization efforts as well as online discussion forums, he recognizes the gaps between standards and practices and knows how to bridge them.

After more than 25 years in different positions at the Helsinki University of Technology, working with IT services, user support, software development, and teaching, Jukka has worked as a consultant and author, as well as at the Finnish Information Society Development Center (IT standardization) and at the Blue1 airline company (multilingual online sales and website). He has authored or co-authored several books on programming languages, web design and technologies, typography, etc., in Finnish. He is also author of *Unicode Explained* (O'Reilly, 2006).

About Byte Level Books

Byte Level Books is an independent publisher of books with a world view. From translation guides to global design best practices, our mission is to publish a range of resources that help individuals and companies go global.

To learn more, visit www.bytelevelbooks.com, and follow us at @bytelevelbooks.

Also available from Byte Level Books: