PHP and Unicode: A Love at Fifth Sight

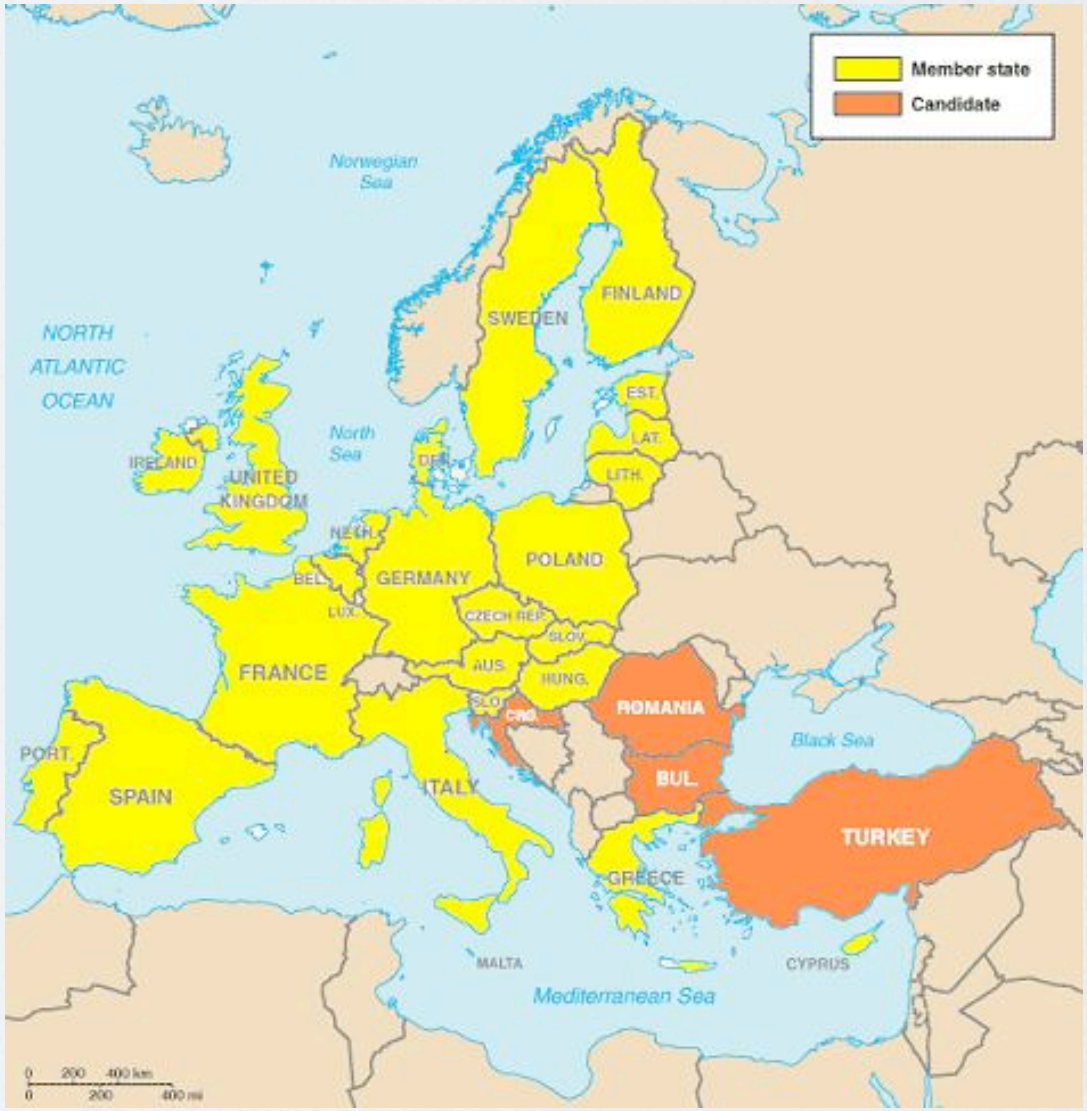
Andrei Zmievski
Yahoo! Inc.

Agenda

- Quick look at today's challenges
- What is Unicode and why is it?
- Unicode and PHP

European Union

25 states and growing



Official Languages of EU

Czech
Danish
Dutch
English
Estonian
Finnish
French
German
Greek (Demotic)
Hungarian
Italian
Latvian
Lithuanian
Maltese
Polish
Portuguese
Slovak
Slovene
Spanish
Swedish

Character Encodings of EU

- ISO 8859-1/ISO 8859-15
  - Danish
  - Dutch
  - English
  - Finnish
  - French
  - German
  - Italian
  - Portuguese
  - Spanish
  - Swedish
- ISO 8859-7 Greek
- ISO 8859-3 Maltese
- ISO 8859-2
  - Czech
  - Latvian
  - Lithuanian
  - Polish
  - Slovak
  - Slovene
  - Estonian
  - Hungarian

Today's Business Challenges

- Supporting languages needed for business
  - Used in the EU, officially or regionally
  - Used by export markets: Japan, China, etc.
- Adding characters as needed, easily
  - e.g. Euro
  - without rewriting each application

Today's Technical Challenges

- Differences in character encodings
  - Require different algorithms
  - Imply different code in each market
  - Lack of integration and interoperability
  - High error rate and poor quality

One Encoding, Many Languages

- One distributable package
- Less maintenance, fewer errors
- Integration! Interoperability! Interchange!
- Extensible! Add characters as needed
- Multilingual: mix languages in documents
- Dynamically change languages

Unicode Character Standard

- One encoding for worldwide use
- International Standard – ISO 10646
- Precisely defined
  - Includes character properties/attributes
  - Algorithms define exact behavior
- Widely supported by standards & industry
  - HTML, XML, Java, Oracle, IBM, MySQL

Unicode Character Standard

- Required by Web & modern applications
  - e,g, International Domain Names
  - DOM
- Increasingly, users are not satisfied with incorrect spellings, or restrictions to write their names, addresses in ASCII or incompatible encodings

Unicode Overview

- Developed by the Unicode Consortium
- Covers all major living scripts
- Version 4.0 has 96,000+ characters
- Capacity for 1 million+ characters
- Unicode Character Set = ISO 10646

Unicode Character Set

- Code Points 0 to 10FFFF, (Maximum 21 Bits)
  - Unicode notation for code point is U+hhhh
  - 17 Planes of 64K (FFFF) code points
- Basic Multilingual Plane (BMP) U+0000-U+FFFF
  - Commonly used characters in living scripts
- 1st Supplementary Plane (U+10000-U+1FFFF)
  - archaic, fictional characters
- 2nd Supplementary Plane (U+20000-U+2FFFF)
  - Ideographs

Unicode Character Set

The primary scripts currently supported by Unicode 4.0 are:

- Arabic
- Armenian
- Bengali
- Bopomofo
- Buhid
- Canadian Syllabics
- Cherokee
- Cypriot
- Cyrillic
- Deseret
- Devanagari
- Ethiopic
- Georgian
- Gothic
- Greek
- Gujarati
- Gurmukhi
- Han
- Hangul
- Hanunóo
- Hebrew
- Hiragana
- Kannada
- Katakana
- Khmer
- Lao
- Latin
- Limbu
- Linear B
- Malayalam
- Mongolian
- Myanmar
- Ogham
- Old Italic (Etruscan)
- Osmanya
- Oriya
- Runic
- Shavian
- Sinhala
- Syriac
- Tagalog
- Tagbanwa
- Tai Le
- Tamil
- Telugu
- Thaana
- Thai
- Tibetan
- Ugaritic
- Yi

Organized by scripts into blocks

Example Unicode Characters

ASCII	ABCDEFGHIJKLMNOP
Latin-1	ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏ
Latin-2	āĂăĄąĆćĈĉĊċČčĎďĐ
Greek	ΪABΓΔEZHΘIKΛMNΞO
Cyrillic	рстуфхцчшщъыьэюя
Thai	ภมยรฤลฦวศษสหฬอฮฯ
CJK	北両丢乖两严並丧
Korean	감갑값갓갔강갖갘

Unicode is Generative

- Composition can create “new” characters
- Base + non-spacing (combining) character(s)

A + ° = Å

U+0041 + U+030A = U+00C5

a + ˆ + . = ậ

U+0061 + U+0302 + U+0323 = U+1EAD

a + . + ˆ = ậ

U+0061 + U+0323 + U+0302 = U+1EAD

Unicode Characteristics

- Multilingual
  - All scripts/languages, one character set
- Character Properties
  - Case, digit, alpha/letter/ideogram, directional class, mirroring, combining class, etc. provided by Unicode
- Logical order for bidirectional languages
- Round Trip Conversion To Legacy Encodings
  - Byte Order Mark (BOM)
  - Big vs. Little endian and encoding identifier

Unicode Characteristics

- 16 bit design originally
- Now has 3 equivalent forms
  - UTF-8: 8-bit variable width, multi-byte (max. 4)
  - UTF-16: 16-bit, variable width, surrogates (max 2)
  - UTF-32: 32-bit, fixed width (max 1)
- Design avoids multi-byte performance problems
- Algorithm specifications provide interoperability
- Allows one program image to be used worldwide
- Developers do not need to be linguists to implement

Definitions

- Abstract Character
  - Unit of information used to organize, control, or represent textual data.
- Code Point = Unicode Scalar Value
  - The unique number assigned to each Unicode abstract character
- Code Unit
  - The code point mapped into basic architectural units (i.e. a form size).

Abstract Character

- A unit of information used to organize, control, or represent textual data.
  - Has no concrete form, not to be confused with a glyph.
  - May not correspond to users' ideas of a "character". Do not confuse with grapheme.
  - Abstract characters not directly encoded by the Unicode Standard are often represented with combining character sequences.

Abstract Character Example

- The character Å can be represented by two abstract characters
  - Latin Capital Letter A U+0041
  - Combining Ring Above ° U+0301

Example Code Units

- This **abstract character** is U+233B4 
- It's **code point** is 233B4, independent of UTF encoding
- In UTF-32 the **code unit** is 233B4
- In UTF-16 it has two 16-bit **code units**:
  - D84C, DFB4 (high and low surrogates)
- In UTF-8, it has four 8-bit **code units**:
  - F0, A3, 8E, B4

Summary

- Unicode is well-supported, ubiquitous, and often required in integrated environments.
- Unicode simplifies working with different languages
- But the large character set requires some additional considerations
- Unicode requires removing assumptions that 1 character is 1 byte or word

Unicode != I18N

- Unicode simplifies development
  - Single source code
  - Enables multilingual processing
  - Properties reduce research for each language
- Unicode does not fix all internationalization
  - E.g. Date, time, number and other formats
  - Linguistic processing can require additional algorithms, data (e.g. word breaking)
  - Still must identify, support cultural requirements
  - Conversion to native encodings for interface to legacy software, systems can impose limitations

Definitions

Internationalization

I18n

To design and develop an application:

- ✓ without built-in cultural assumptions
- ✓ that is **efficient** to localize

Localization

L10n

To tailor an application to meet the needs of a particular region, market, or culture

Date Formats

- U.S.A.: 2/16/05
- France: 16.2.05 or 16-2-05
- Japan, China: 2005年2月16日

Calendars

- Gregorian 2005
- Thailand: 2548 (Buddhist Year)
- Taiwan: 94 (1911-based)
- Hebrew 5765
- Also Hijri (Islamic), Lunar (Asia) and many others

Time Formats

- U.S.A.: 4:00 P.M.
- France: 16.00
- Japan: 1600
- Don't forget to identify the time zone

Number Formats

- England: 12,345.67
- Germany: 12.345,67
- Switzerland: 12’345,67
- Swiss money: 12’345.67
- France: 12 345,67
- India: 12,34,567.89

Curency

- Symbol placement
- Symbol length (1-15)
- Number width
- Number precision
  - Spain, Japan 0
  - Mexico, Brazil 2
  - Egypt, Iraq 3

Currency examples

US $12.34

12.345,67 €

12$34€

¥123

Capitalization

- Greece: Σ → σ (in the middle of a word)
- Greece: Σ → ς (at the end of a word)
- Turkey: i → İ, ı → I
- Germany: ß → SS (lower[SS]=ss)

Sorting

English: **ABC...RSTUVWXYZ**

German: **AÄB...NOÖ...SßTUÜV…YZ**

Swedish/Finnish: **ABC...RSTUVWXYZÅÄÖ**

- Languages may sort more than one way
  - German dictionary vs. phone book
  - Japanese stroke-radical vs. radical-stroke
  - Traditional vs. modern Spanish

Collation Examples

- Swedish: z < ö
- German: ö < z
- Dictionary: öf < of
- Phonebook: of < öf
- Upper-first: A < a
- Lower-First: a < A
- Contractions: H < Z, but CH > CZ
- Expansions: OE < Œ < OF

Normalization

- Equivalent strings are put into a single standardized form
- Benefits
  - Fast binary comparison
  - Accurate digital signatures
  - Clarifies “equivalence”

Motivation

- Increasingly globalized world
- Sites driven by PHP are seen by eyeballs from many different countries
- Yet PHP has no intrinsic awareness of and support for multilingual processing and i18n

Unicode and Competition

Python

separate Unicode type

module for basic string manipulation

basic regexp support

Unicode and Competition

Python

upgrades strings to Unicode when needed

IO layer support

Perl

regexp support

higher level services are available through CPAN

Unicode and Competition

Python

Perl

most complete Unicode support

many i18n features

Java

State of I18N in PHP

- Locale support based on POSIX
- mbstring extension attempts to solve certain issues
- Inherent problems remain

Unicode Support

- Native Unicode string type
- Unicode string literals
- Updated language semantics
- Relevant functions understand Unicode

ICU

- IBM Components for Unicode
- Why not our own solution?
  - Lots of know-how is required
  - Reinventing the wheel
  - In the spirit of PHP: borrow when possible, invent when needed, but solve the problem

Why ICU?

- It exists
- Full-featured
- Robust
- Fast
- Proven
- Portable
- Extensible
- Open Source
- Supported and maintained

ICU Features

- ✓ Unicode Character Properties
- ✓ Unicode String Class & text processing
- ✓ Text transformations (normalization, upper/lowercase, etc)
- ✓ Text Boundary Analysis (Character/ Word/Sentence Break Iterators)
- ✓ Encoding Conversions for 500+ legacy encodings
- ✓ Language-sensitive collation (sorting) and searching
- ✓ Unicode regular expressions
- ✓ Thread-safe
- ✓ Formatting: Date/Time/Numbers/ Currency
- ✓ Cultural Calendars & Time Zones
- ✓ (230+) Locale handling
- ✓ Resource Bundles
- ✓ Transliterations (50+ script pairs)
- ✓ Complex Text Layout for Arabic, Hebrew, Indic & Thai
- ✓ International Domain Names and Web addresses
- ✓ Java model for locale-hierarchical resource bundles. Multiple locales can be used at a time

Goals

- Backwards compatibility!
- Making simple things easy and complex things possible
- Concentrating on functionality first
- Achieving parity with Java's Unicode support

Caveats

- Very preliminary look at the state of Unicode support
- Many issues are still being discussed
- No public access to the code for now

Getting There

- Retrofitting the engine to support Unicode
- Making existing extensions Unicode-aware
- Exposing ICU API

Let There Be Unicode!

- Unicode should not be imposed on everyone
- Changing certain language semantics would make development easier, however
- Solution: a control switch we'll call `unicode`
- Per-request INI setting or `declare()` pragma
- No changes to program behavior unless enabled
- Does not imply no Unicode at all when disabled!

String Types

- Unicode: textual data (UTF-16 internally)
- Binary: binary data and strings meant to be processed on the byte level
- Codepage: for backwards compatibility and representing strings in a certain encoding

Creating Strings

- With `unicode=off`, string literals are old-fashioned 8-bit strings
- 1 character = 1 byte

```
$str = "hello world"; // ASCII string
echo strlen($str); // result is 11


$jp = "検索オプション"; // UTF-8 string
echo strlen($str); // result is 21
```

Creating Strings

- With `unicode=on`, string literals are Unicode
- 1 character may be > 1 byte

```
// unicode = on
$str = "hello world"; // Unicode
echo strlen($str); // result is 11


$jp = "検索オプション"; // Unicode
echo strlen($str); // result is 7
```

- To obtain length in bytes one would use a separate function

Creating Strings

- Binary string literals require new syntax

```
$str = b'abcd';
$str = b"abc\xa0cd";
$str = b<<<EOD
   ab\x20cd
EOD;
```

Escape Sequences

- In Unicode strings `\uXXXX` and `\UXXXXXXX` escape sequences may be used to specify Unicode code points explicitly

```
// these are equivalent
$str = "Hebrew letter alef: א";
$str = "Hebrew letter alef: \u05D0";

// so are these
$str = 'ideograph: 丈';
$str = 'ideograph: \U02000B';
```

Encodings

- Encodings are essential in specifying the format for various Unicode-related operations
- PHP will have at least:
  - runtime encoding
  - script encoding
  - output encoding
  - http input encoding
  - fallback encoding

Runtime Encoding

- Specifies what encoding to attach to codepage strings generated at runtime

```
// runtime_encoding = iso-8859-1

$uni = "Cinéma";  // Unicode
$str = (string)$str; // ISO-8859-1 string
$uni = (unicode)$uni; // back to Unicode
```

- Also used when interfacing with functions that do not work with Unicode yet

```
$str = long2ip(20747599); // $str is iso-8859-1
```

Script Encoding

- Currently, scripts are written in a variety of encodings: ISO-8859-1, Shift-JIS, UTF-8, etc.
- Engine needs to know the encoding of a script in order to parse it
- Encoding can be specified as INI setting or with `declare()` pragma
- Affects how identifiers and string literals are interpreted

Script Encoding

- Whatever the encoding of the script, the internal type of the string is Unicode

```
// script_encoding = iso-8859-1
$uni = "øl"; // bytes are F8 6C

// script_encoding = utf-8
$uni = "øl";  // bytes are C3 B8 6C
```

Script Encoding

- Encoding can be also changed with a pragma
- Pragma does not propagate to included files

```
// script_encoding = utf-8

declare(encoding="iso-8859-1");
$uni = "øl"; // bytes are F8 6C

// the contents of file are read as UTF-8
include "myfile.php";
```

Output Encoding

- Specifies the encoding for the standard output stream
- The script output is transcoded on the fly
- Does not affect binary strings

```
// output_encoding = utf-8
// script_encoding = iso-8859-1

$uni = "øl"; // input bytes are F8 6C
echo $uni; // output bytes are C3 B8 6C

echo b"øl"; // output bytes are F8 6C
```

Fallback Encoding

- The encoding is used when other encodings are not specified explicitly
- Easy, one-stop configuration
- Defaults to UTF-8 if not set

Fallback Encoding

- If you work only with ISO-8859-2 data:

```
fallback_encoding = iso-8859-2
```

- With this setting script, runtime, and output encodings are also ISO-8859-2 unless you override them explicitly

Conversion Semantics

- Binary type cannot be converted to other string types, via casting or implicitly
- Codepage strings can be freely converted to Unicode, but no implicit conversions happen from Unicode to codepage strings
- Both codepage strings and Unicode ones can be cast to binary

Conversion Issues

- Not all characters can be converted between Unicode and legacy encodings
- PHP will attempt to convert as much of the data as possible and issue an error
- The conversion error behavior may be customizable

Operator Support

- Concatenating a codepage string with Unicode one requires upconverting it to Unicode

```
$str = foo();        // foo() returns a codepage string
$uni = "def";        // Unicode string
$res = $str . $uni; // result is Unicode
```

- Binary type cannot be concatenated with other types

```
$res = b"abc" . "新着情報";          // runtime error!
$res = b"abc" . b"新着情報";         // OK
$res = b"abc" . (binary)"新着情報"; // OK, but different result
```

Operator Support

- String offset operator works on code points, not bytes!

```
$str = "大学";   // bytes are e5 a4 a7 e5 ad a6
echo $str{1}; // result is 学
$str{0} = 'サ'; // string is now サ
                // bytes are e3 82 b5 e5 ad a6
```

- No need to change existing code if working only with single-byte encodings, like ASCII or ISO-8859-1

Arrays

- Unicode and binary strings can be used as keys
- Unicode switch affects how lookup is done
  - With `unicode=on`, codepage `"abc"` and Unicode `"abc"` are equivalent for hash lookup purposes
  - With `unicode=off`, they are distinct

Inline HTML

- HTML blocks are already in the output encoding
- Hence, they are treated as binary strings and passed through

Functions

- Ideally, all functions should be upgraded to detect the type of the string passed to them and do the right thing
- This is a continuous process that will require involvement from extension authors

Functions

- In order to ease the transition, some stop-gap measures will be implemented
- If a Unicode string is passed to a function expecting a legacy string, the engine will attempt to convert Unicode to the runtime encoding
- The inverse happens for functions requiring Unicode that are passed a legacy string

Functions

- Many Unicode operations may require additional context
- Upgraded functions will use the most common mode of operation, and leave the edge cases to ICU API
- Consider `strcasecmp()`

Functions

- By default `strcasecmp()` would use default locale and collation

```
if (strcasecmp($a, $b) == 0) {
  ...
}
```

- If you needed to compare strings using adjusted collation parameters, you would use ICU API

```
$collator = ucol_open("fr_FR", ...);
ucol_setAttribute($collator, ...);
if (ucol_strcoll($collator, $a, $b) == 0) {
 ...
}
```

Stream IO

- When unicode switch is on, we cannot make any assumptions about the type or encoding of the data coming in from the streams
- Streams will be in binary mode by default

Stream IO

- Applications can manage Unicode conversion itself themselves

```
$data = file_get_contents('mydata.txt');
$unidata = unicode_decode($data, 'EUC-JP');
```

- Or apply a conversion filter to the stream

```
$fp = fopen($file, 'rw');
stream_filter_append($fp, 'convert', 'EUC-JP');
// reads EUC-JP data and converts to Unicode
$data = fread($fp, 1024);
// converts Unicode to EUC-JP and writes it
fwrite($fp, $data);
```

Unicode Identifiers

- PHP will allow Unicode characters in identifiers
- Can have ideographic in addition to accented characters

```
class コンポーネント {
    function コミット {  .. }
}

$プロバイダ = array();
$プロバイダ[‘רַעְיוּלוּחַ שָׁנָה’] = new コンポーネント();
```

HTTP Input

- Incoming content may or may not specify the encoding it is in
- If not specified, PHP can use the `http_input_encoding` setting to decode the data
- If the encoding is passed as an application request variable, the application can call a function to repopulate the request arrays based on this encoding

Looking Forward

- Finalize base design and API
- Merge code into public CVS
- Help extension authors upgrade their functions
- Expose ICU services
- Optimize performance
- Document and educate

Thank You!

have a nice day