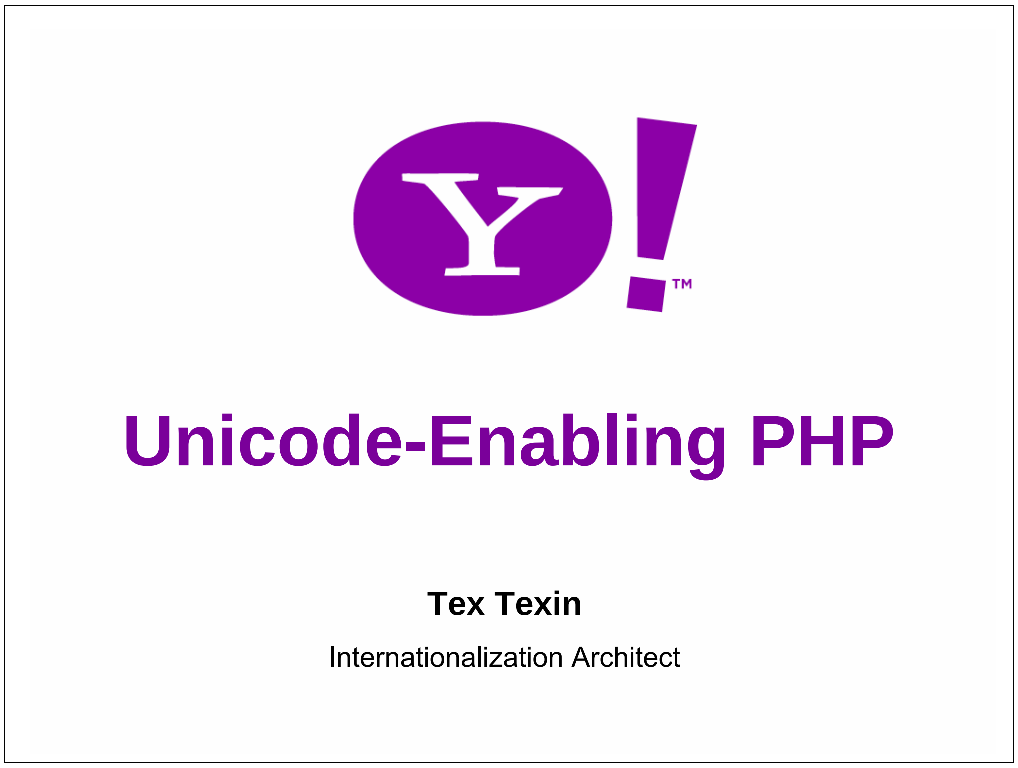
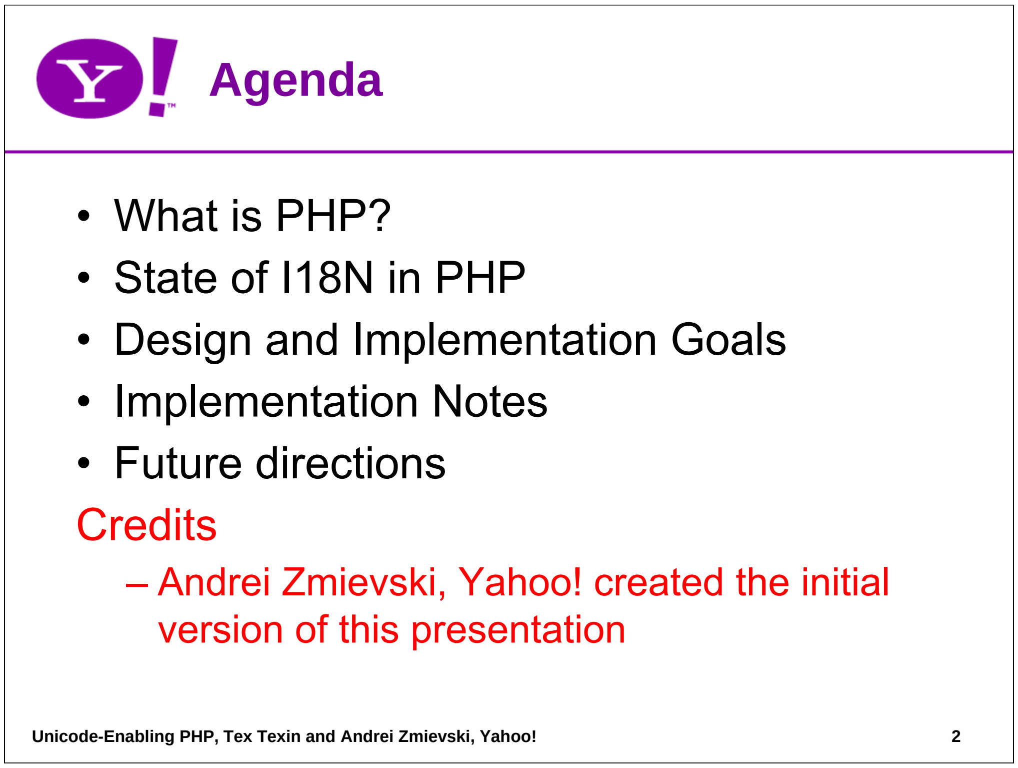


Unicode-Enabling PHP

Tex Texin

Internationalization Architect



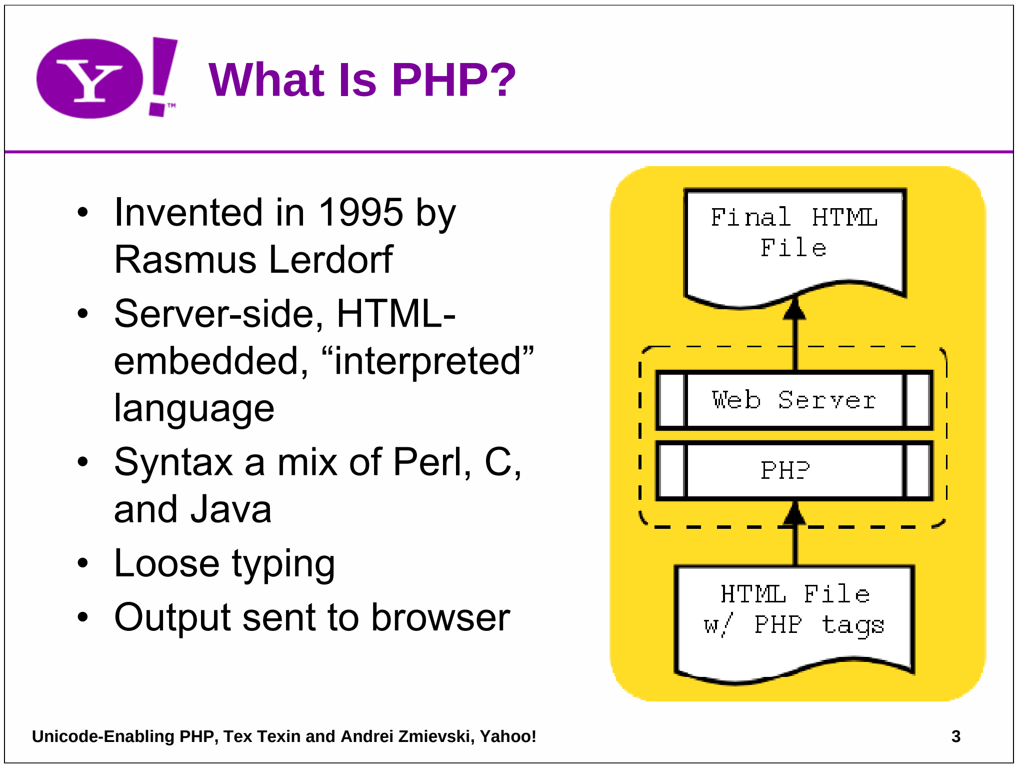
Agenda

- What is PHP?
- State of I18N in PHP
- Design and Implementation Goals
- Implementation Notes
- Future directions

Credits

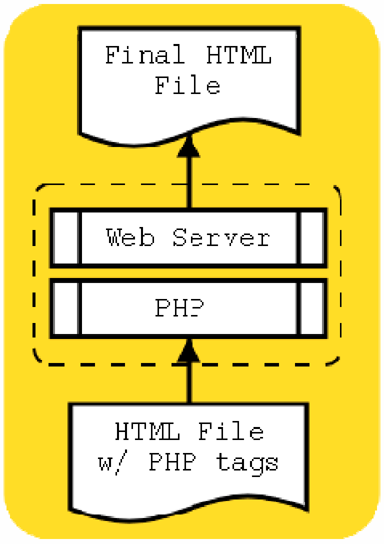
– Andrei Zmievski, Yahoo! created the initial version of this presentation

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 2



What Is PHP?

- Invented in 1995 by Rasmus Lerdorf
- Server-side, HTML-embedded, “interpreted” language
- Syntax a mix of Perl, C, and Java
- Loose typing
- Output sent to browser

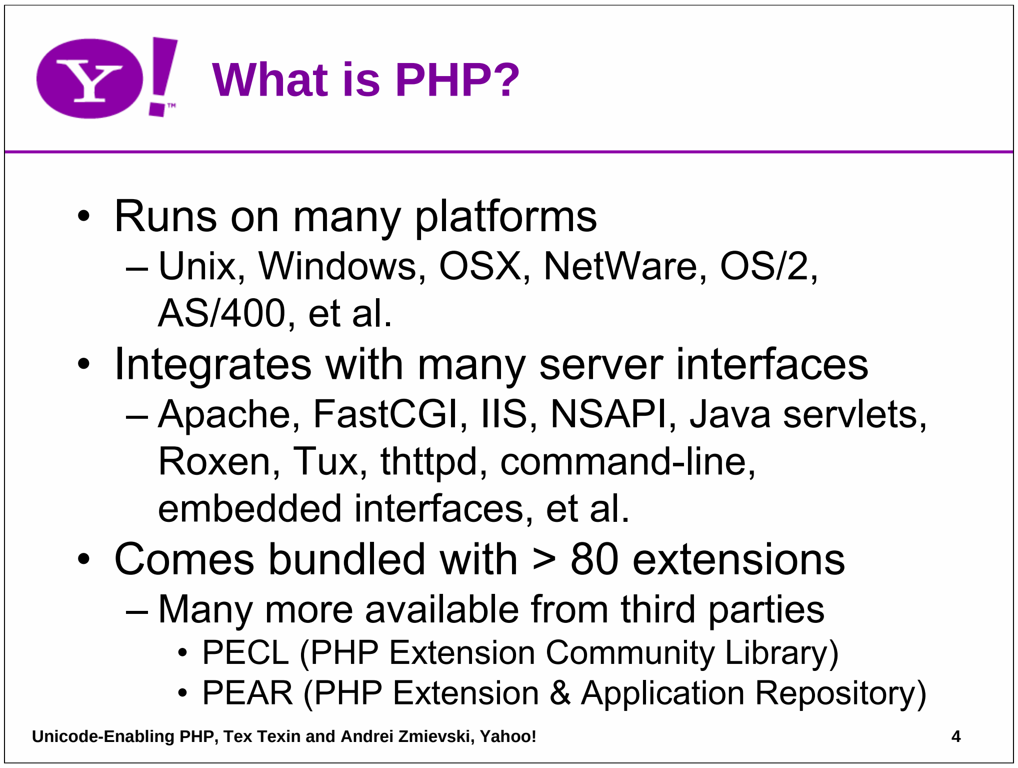


Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 3

PHP has been used to develop Web applications from day one. Rasmus Lerdorf originally created it to track visits to his online resume and shortly thereafter released it for public consumption. The language has grown from a set of simple tags to a full-fledged scripting framework.

It is a server-side language, meaning that execution of the scripts is done by the Web server, rather than the browser. PHP script fragments are embedded inside HTML (or textual) files and their output is sent to the browser. It is “interpreted” in the sense that it is not compiled to machine code, but rather to a set of opcodes that are interpreted inside a virtual machine.

Its syntax and semantics have been borrowed from Perl, C, and Java, so it is easy for someone proficient in one of these languages to learn PHP. One of the hallmarks of PHP is the loose data typing: the type of the variable is decided at runtime according to the context in which the variable is used, and the variables do not need to be syntactically marked as containing a specific type.



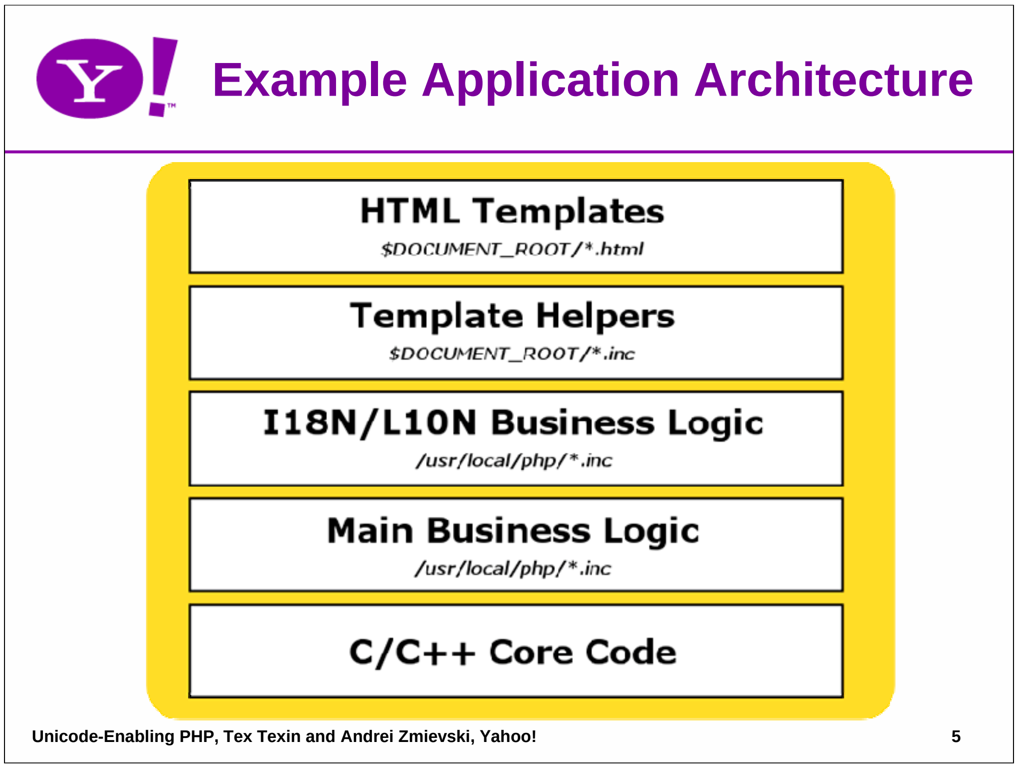
What is PHP?

- Runs on many platforms
  - Unix, Windows, OSX, NetWare, OS/2, AS/400, et al.
- Integrates with many server interfaces
  - Apache, FastCGI, IIS, NSAPI, Java servlets, Roxen, Tux, thttpd, command-line, embedded interfaces, et al.
- Comes bundled with > 80 extensions
  - Many more available from third parties
    - PECL (PHP Extension Community Library)
    - PEAR (PHP Extension & Application Repository)

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 4

PHP is platform and server agnostic. It runs on all varieties of Unix, Windows, OSX, NetWare, OS/2, AS/400, and many others. Its Server API is flexible enough to work under Apache, FastCGI, IIS, NSAPI, Java servlets, Roxen, Tux, thttpd, command-line, embedded interfaces, and pretty much anything else that comes along.

Everyday Web programming needs are supported by the 80 or so bundled modules. Additionally, PECL (PHP Extension Community Library) and PEAR (PHP Extension and Application Repository) provide diverse collections of third-party code that can be easily deployed.

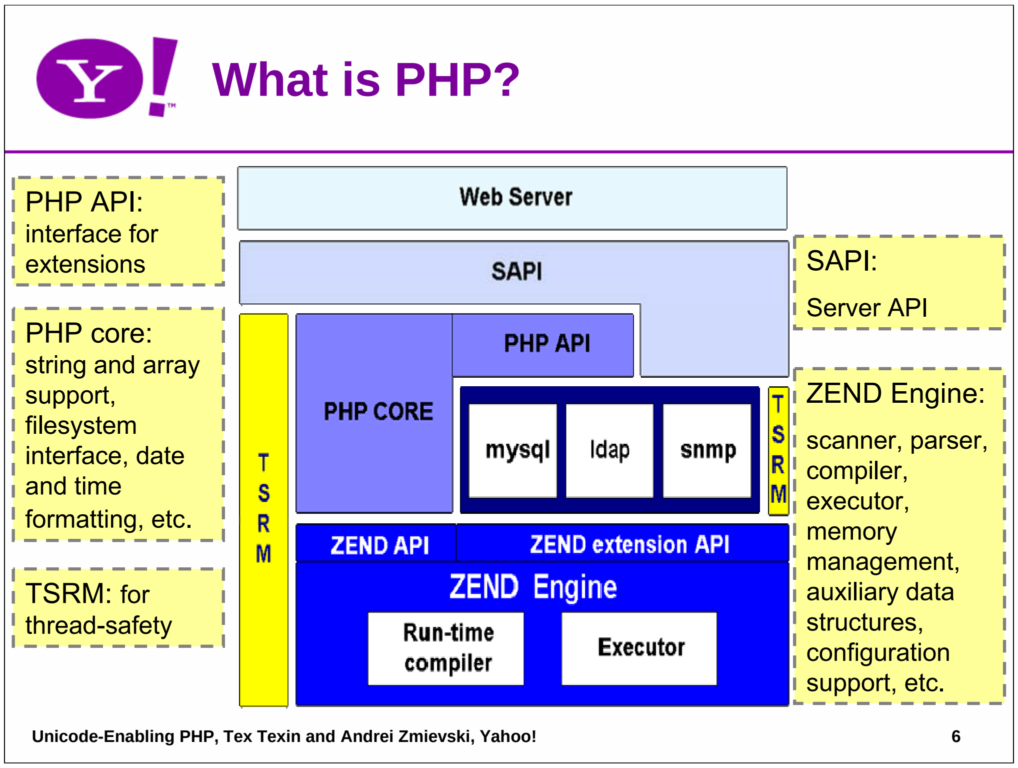


Example Application Architecture

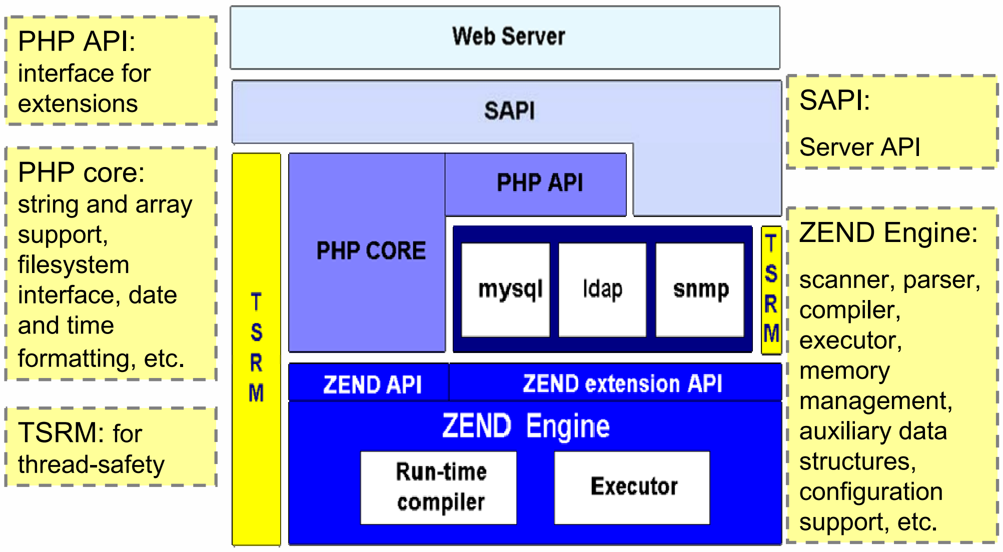
Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 5

A suggested architecture for a PHP application is shown above. The template layer should have as little business logic as possible. As you go down the ladder, you have less presentation and more business logic.

The HTML Templates layer consists of HTML files with a few output statements and calls into PHP code to produce dynamic content. The Template Helpers encapsulate code that generates dynamic HTML code, such as form elements for date entry. Logic related to I18N/L10N is contained in the next layer. The main business logic contains the bulk of data processing, data model interface, and communication with external sources. Finally, the low-level C/C++ code can be used to improve performance or provide capabilities not available in PHP itself.

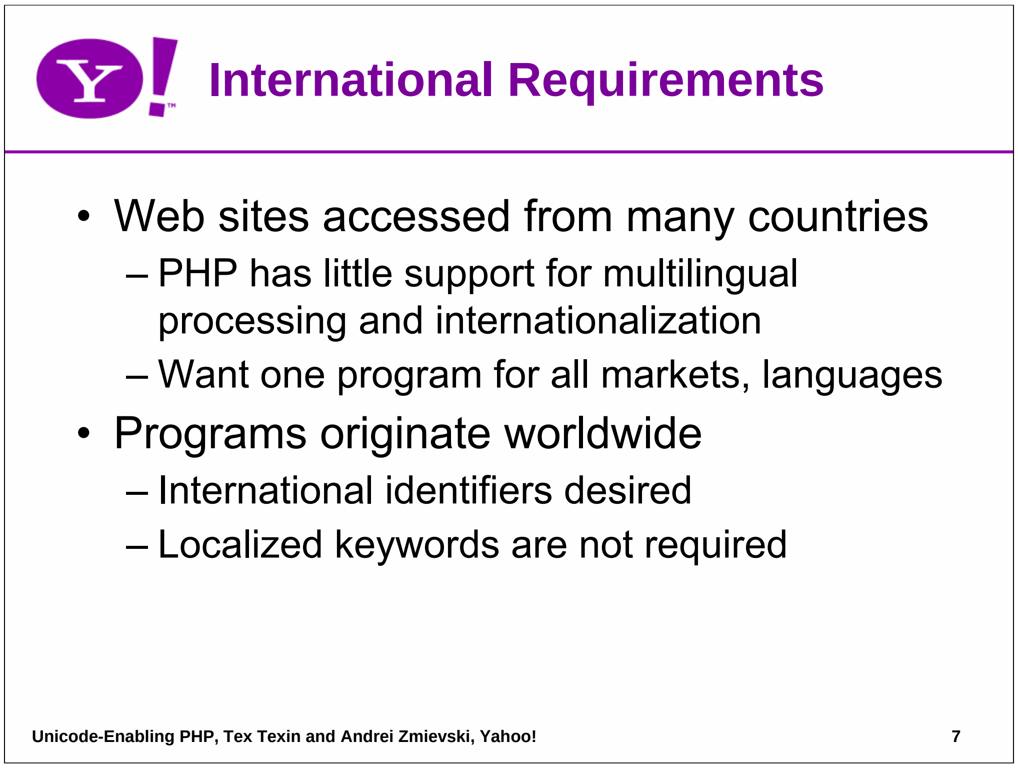


What is PHP?



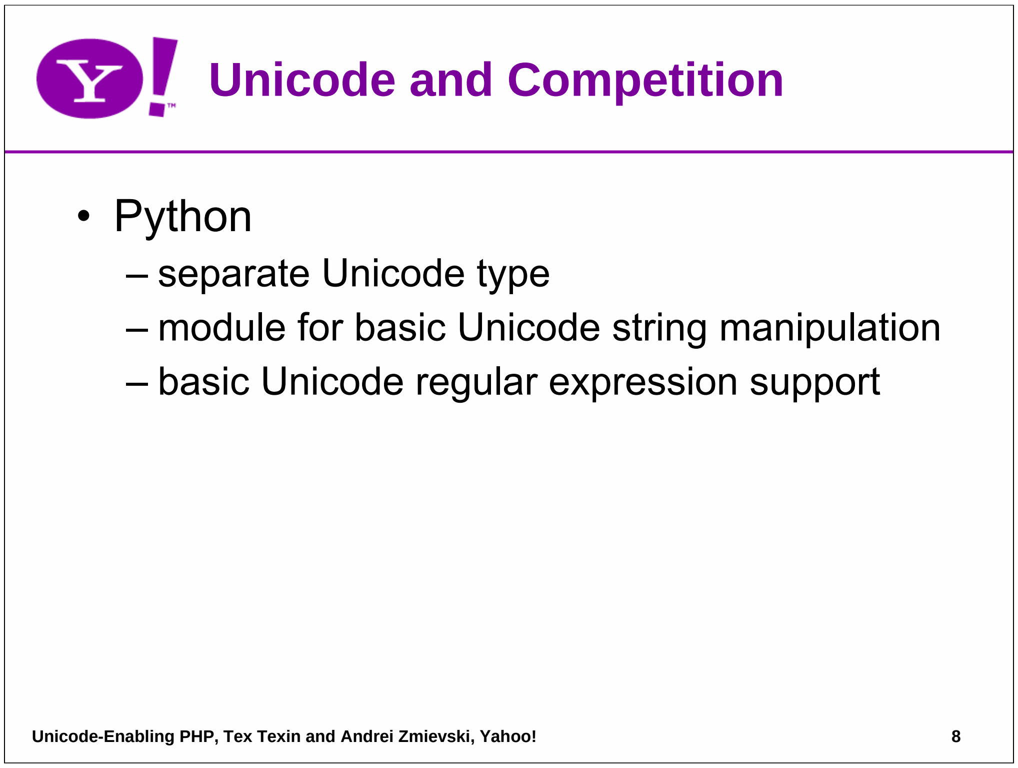
Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 6

The low-level functionality of PHP is based on the Zend engine. It provides scanner & parser, compiler & executor, memory handling routines, auxiliary data structures, configuration files support, and a few other low-level things. By itself, it contains very few functions. Instead, it provides an API interface that is used to develop all the extensions that make PHP so powerful. PHP Core is an extension called ‘standard’ that contains a few hundred functions that deal with string and array manipulation, filesystem interface, date and time formatting, and so on. The rest of extensions such as mysql, ldap, etc make use of Zend API and exposes certain APIs of their own. The TSRM module is used to make PHP and Zend thread-safe across various platforms. Finally, SAPI (Server API) component is what enables PHP to hook into Apache, IIS, and other components that drive PHP.



International Requirements

- Web sites accessed from many countries
  - PHP has little support for multilingual processing and internationalization
  - Want one program for all markets, languages
- Programs originate worldwide
  - International identifiers desired
  - Localized keywords are not required

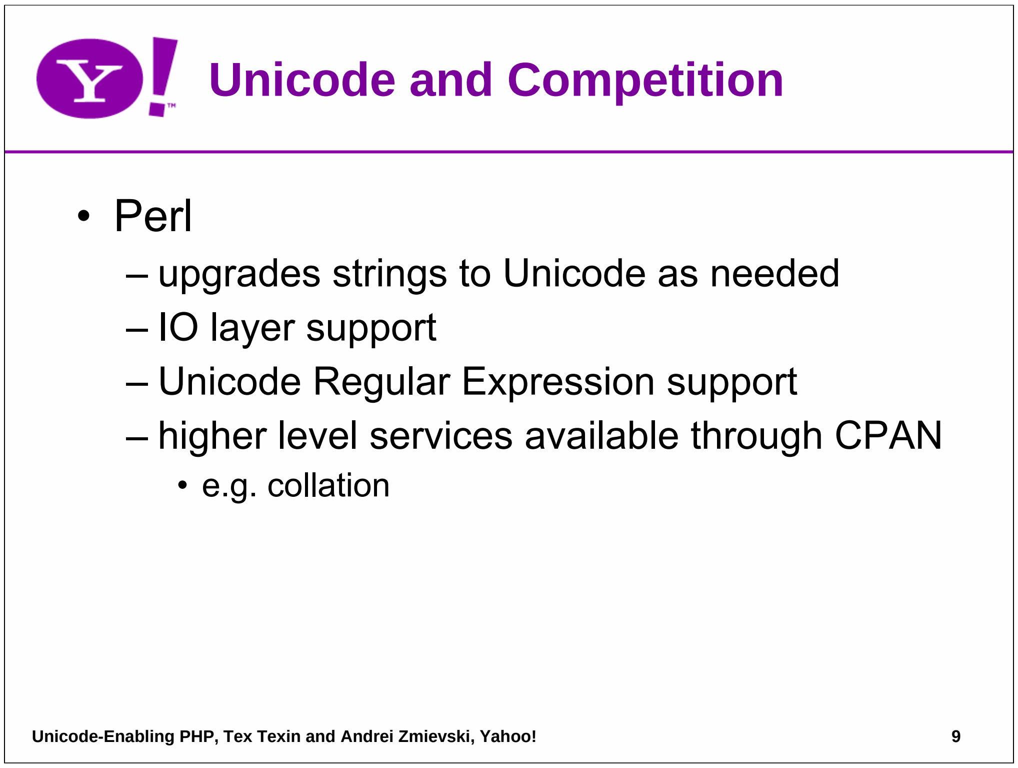


Unicode and Competition

- Python
  - separate Unicode type
  - module for basic Unicode string manipulation
  - basic Unicode regular expression support

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 8

Python provides a separate Unicode string type as well as Unicode module that has functions for manipulating Unicode strings. Regular expressions can search Unicode strings. Most of the standard library works smoothly with Unicode strings. Some modules still are not fully Unicode-friendly, but the most important pieces are in place.

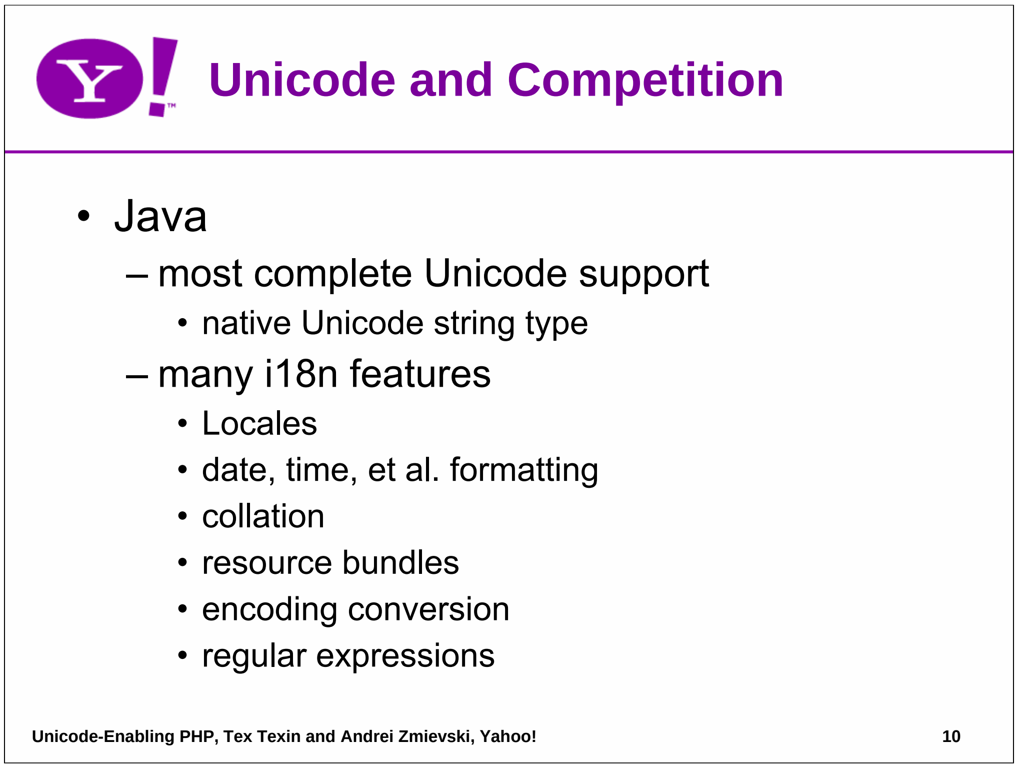


Unicode and Competition

- Perl
  - upgrades strings to Unicode as needed
  - IO layer support
  - Unicode Regular Expression support
  - higher level services available through CPAN
    - e.g. collation

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 9

Perl supports both pre-5.6 strings of eight-bit native bytes, and strings of Unicode characters. The principle is that Perl tries to keep its data as eight-bit bytes for as long as possible, but as soon as Unicodeness cannot be avoided, the data is transparently upgraded to Unicode. It has support for Unicode in the PerlIO layer and its regular expressions. Most of its string operators support UTF-8, higher level functionality like collation is provided by CPAN modules.

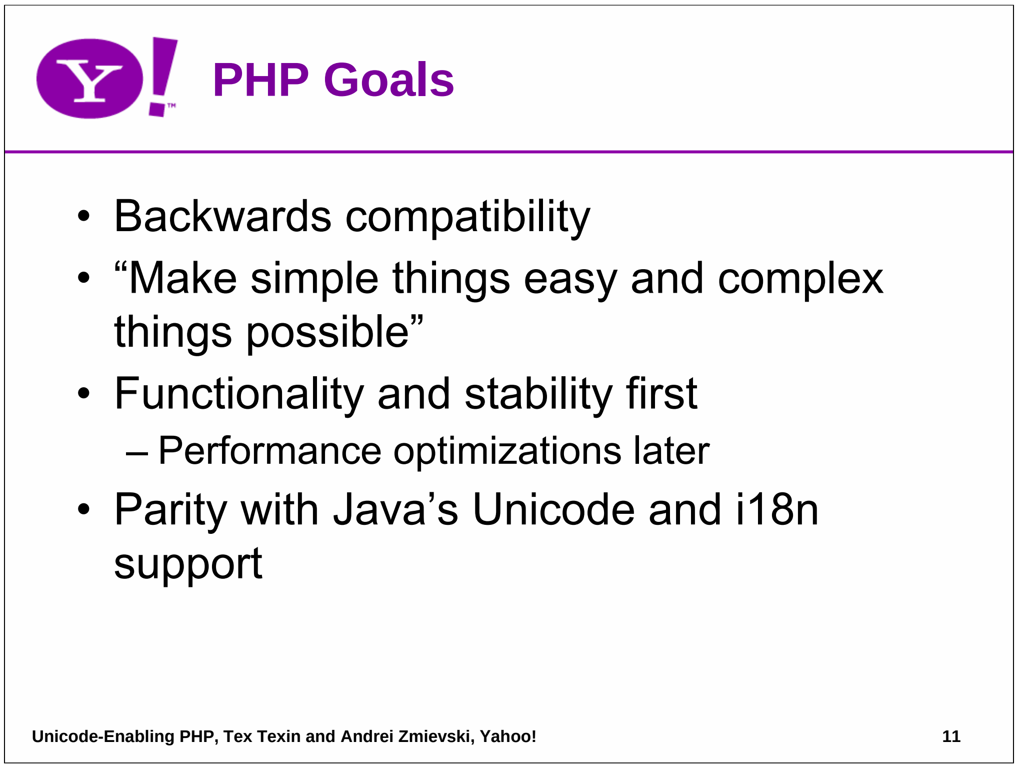


Unicode and Competition

- Java
  - most complete Unicode support
    - native Unicode string type
  - many i18n features
    - Locales
    - date, time, et al. formatting
    - collation
    - resource bundles
    - encoding conversion
    - regular expressions

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 10

Java has perhaps the most complete Unicode support of the languages reviewed here. It has native Unicode string type, Locale identification and localization, date and time handling, collation, text processing, resource bundles, character encoding conversion, regular expressions, and much more.



PHP Goals

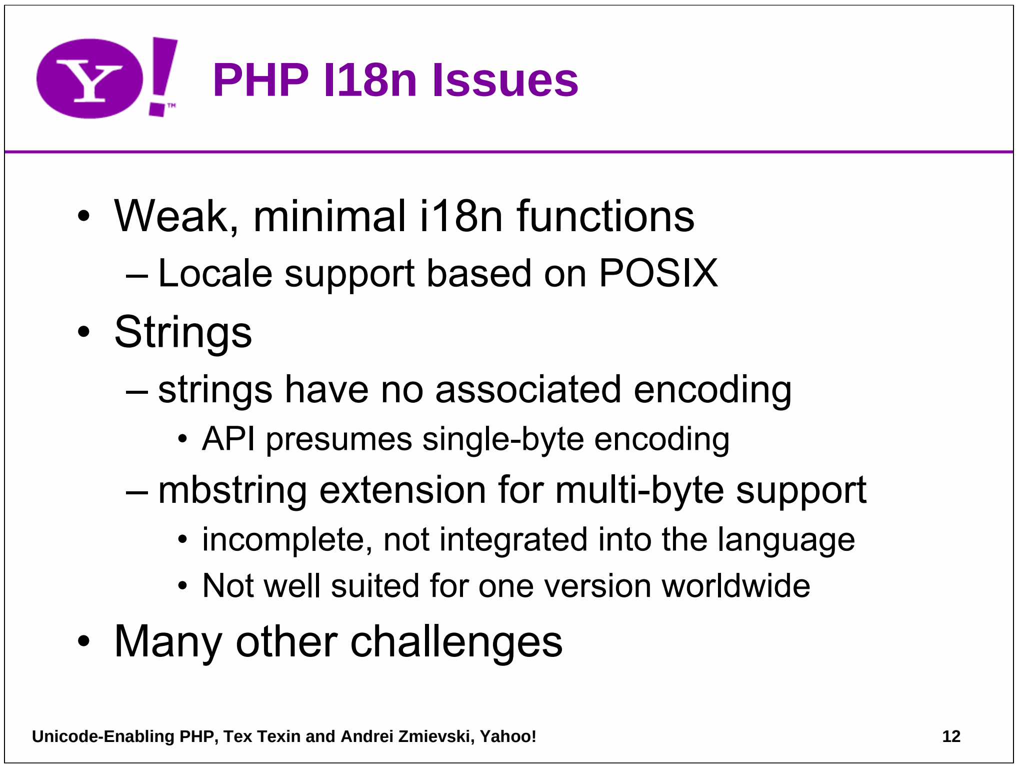
- Backwards compatibility
- “Make simple things easy and complex things possible”
- Functionality and stability first
  - Performance optimizations later
- Parity with Java’s Unicode and i18n support

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 11

Throughout the years we have tried to make sure that PHP remains easy to use, that it is easy to upgrade to new versions, and that there are no surprises when developers migrate their apps to the latest and greatest release. Thus, the number one priority during this project has been backwards compatibility.

We also wanted to keep the learning curve as shallow as possible, keeping with PHP’s motto of making simple things easy and complex things possible. While performance has been a major consideration, functionality and stability are taking the front seat, and the optimizations will be left until we are sure that the new version passes all our tests and has no backwards compatibility problems.

During development, we frequently referred to Java as the paragon of languages with good Unicode and i18n support, and we strived to make PHP’s Unicode features equally powerful.



PHP I18n Issues

- Weak, minimal i18n functions
  - Locale support based on POSIX
- Strings
  - strings have no associated encoding
    - API presumes single-byte encoding
  - mbstring extension for multi-byte support
    - incomplete, not integrated into the language
    - Not well suited for one version worldwide
- Many other challenges

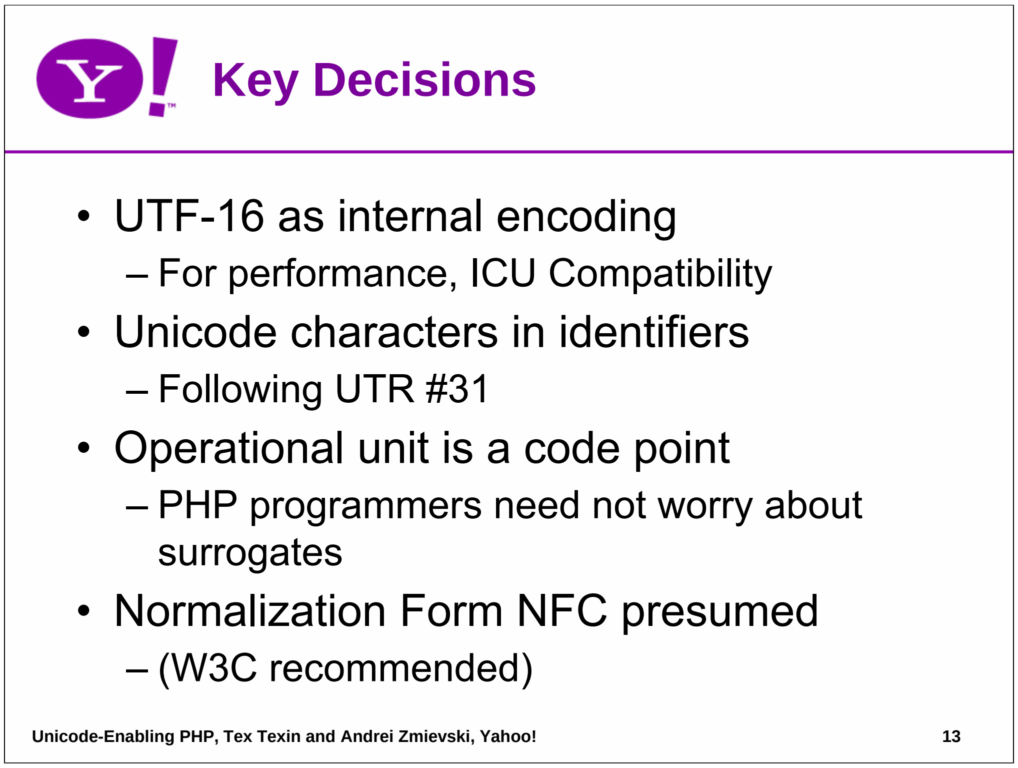
Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 12

PHP strings are considered to contain 8-bit binary data, without any associated encoding. Most functions operate on strings as single-bytes, which is a problem for multi-byte text. PHP is unaware of encodings, forcing users to manually convert data using **iconv()** API.

The **mbstring** extension solves some problems, but not all. It is tailored for CJK market, supports ~20 common encodings, and has 10+ primitive string functions. But it is not integrated into the language runtime and lacks collation, search, boundary analysis features, etc.

Locale support is limited to the underlying C POSIX library. It relies on the locale data installed on the system, which may be out of date or missing altogether. Locale fallback is also not available.

PHP has a few functions for formatting date and time, numbers, monetary units, etc, but the API is not consistent and has overlaps and gaps. e.g. Both **date()** and **strftime()** format dates.



Key Decisions

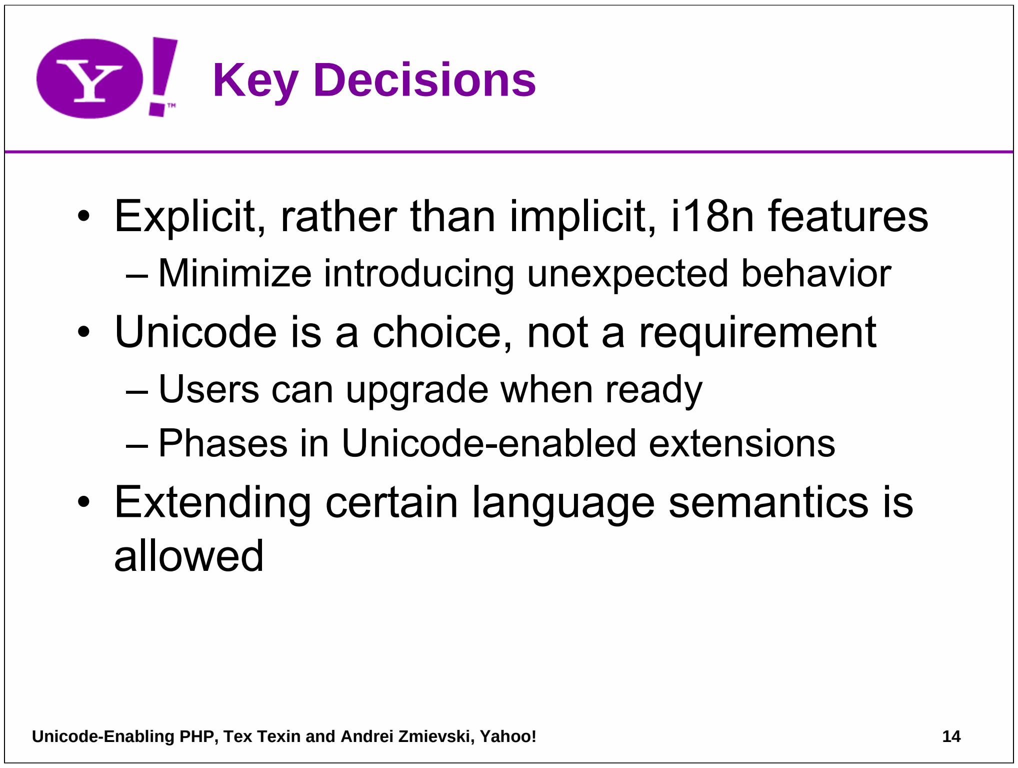
- UTF-16 as internal encoding
  - For performance, ICU Compatibility
- Unicode characters in identifiers
  - Following UTR #31
- Operational unit is a code point
  - PHP programmers need not worry about surrogates
- Normalization Form NFC presumed
  - (W3C recommended)

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 13

The only real choices for internal encoding during the design phase were UTF-8 and UTF-16. While, UTF-8 would have allowed a more transparent interface to external libraries and some memory savings for ASCII text, we felt that UTF-16 provided the best space/time tradeoff for majority of languages and was also the encoding that ICU library uses internally.

PHP identifiers would include any Unicode character that is valid according to UTR #31. We chose to use the character properties in order to allow valid characters, rather than allowing any except certain invalid ones.

The most logical unit of operation in our opinion was code point, not code unit or grapheme. It simplifies migration, since in the majority of legacy encodings the text is precomposed and we can expect that most of text on the Web is in the NFC form anyway. It also frees the user from having to worry about handling surrogate pairs.



Key Decisions

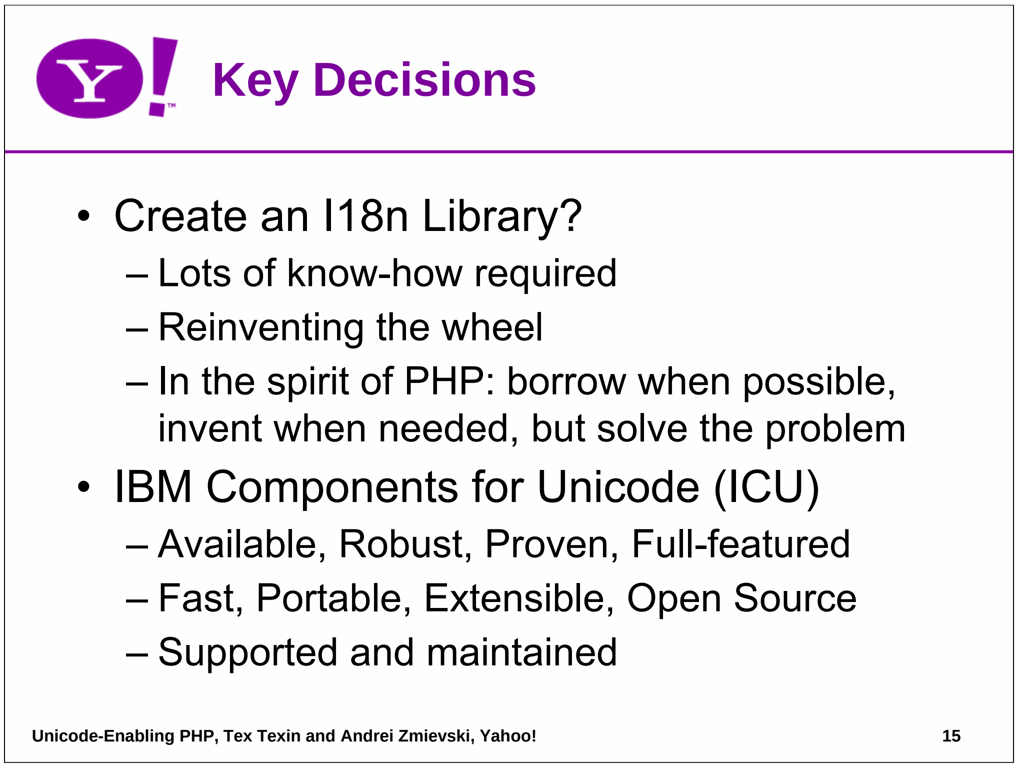
- Explicit, rather than implicit, i18n features
  - Minimize introducing unexpected behavior
- Unicode is a choice, not a requirement
  - Users can upgrade when ready
  - Phases in Unicode-enabled extensions
- Extending certain language semantics is allowed

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 14

It was tempting to modify certain language features and functions to do have implicit i18n support, such as respecting Thai numbers when casting strings to integers or using collation in string comparison operators. In the end, we decided that i18n support should be explicit so that the user run into unexpected results during migration, however proper they might be.

The majority of text on the Web is in the NFC form. Expecting strings to conform to this form by default would simplify implementation and would cover 90% of cases out there.

We did not want to impose Unicode onto everyone all at once. We decided that at least in the next major version of PHP, using Unicode should be a conscious choice, rather than a requirement. We decided to have a switch that would change certain language semantics that would make working with Unicode easier and would place the choice in the hands of the user.



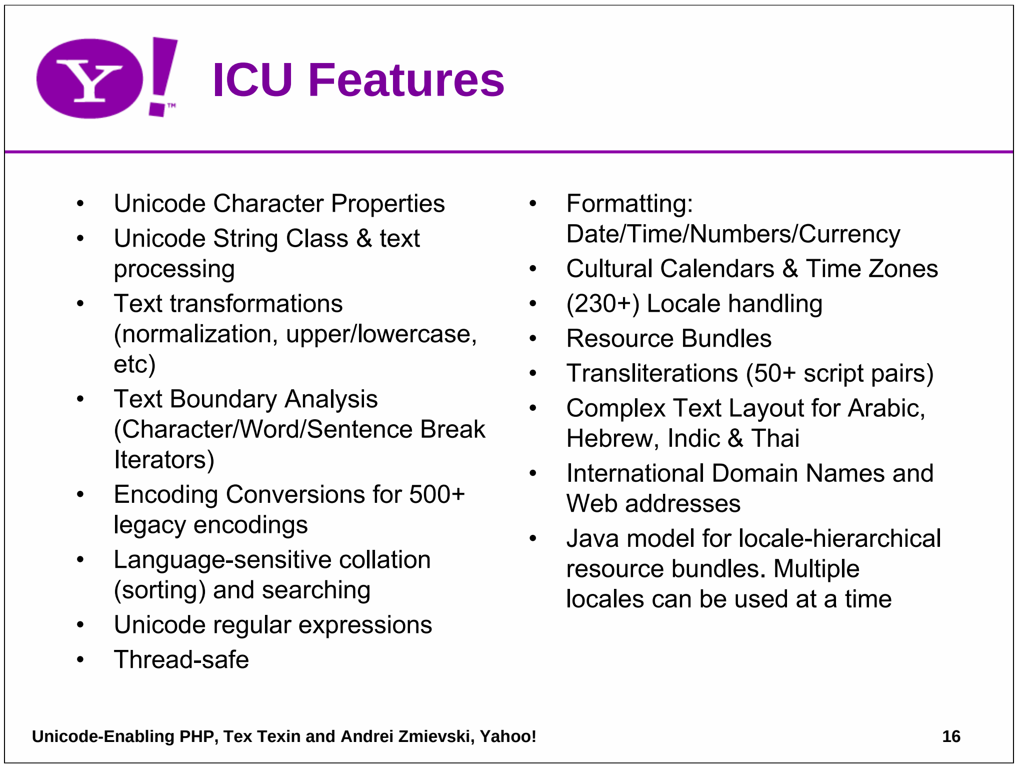
Key Decisions

- Create an I18n Library?
  - Lots of know-how required
  - Reinventing the wheel
  - In the spirit of PHP: borrow when possible, invent when needed, but solve the problem
- IBM Components for Unicode (ICU)
  - Available, Robust, Proven, Full-featured
  - Fast, Portable, Extensible, Open Source
  - Supported and maintained

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 15

There were basically two approaches to obtaining low-level Unicode character support and algorithms for use in PHP: write them ourselves or use an existing library. Developing our own solution was quickly discarded as inefficient, since learning all the intricacies of Unicode character database and algorithms, and implementing them would take a long time and would probably duplicate existing efforts.

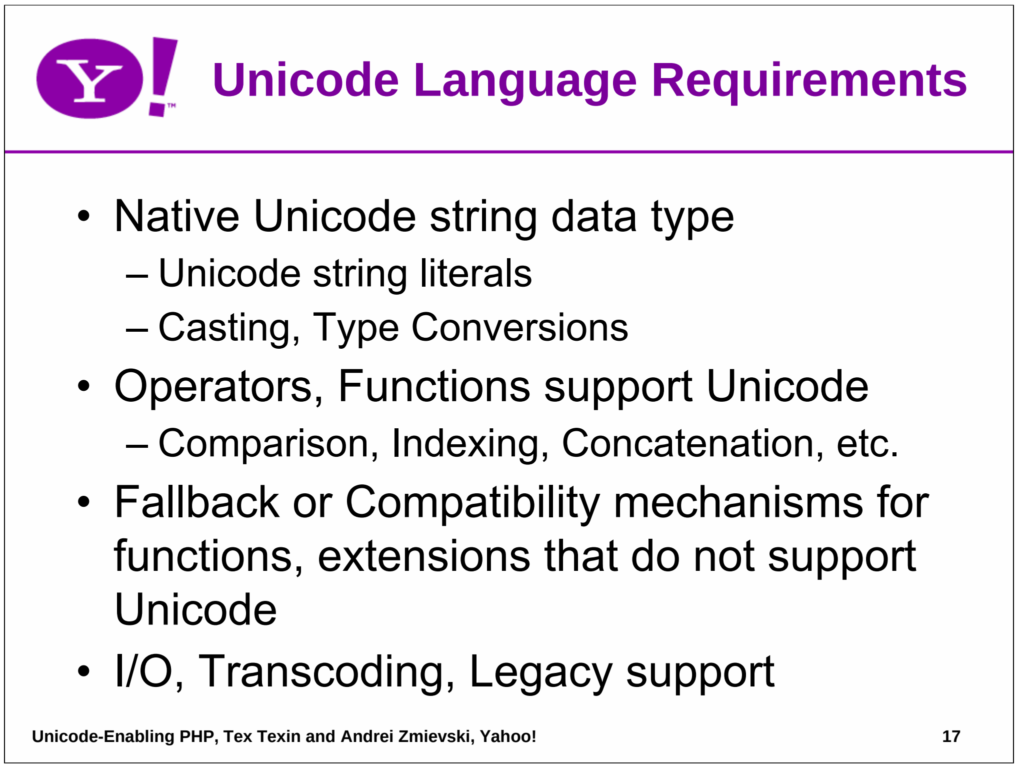
The only serious choice among the libraries that allegedly provide Unicode support was ICU.



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

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 16

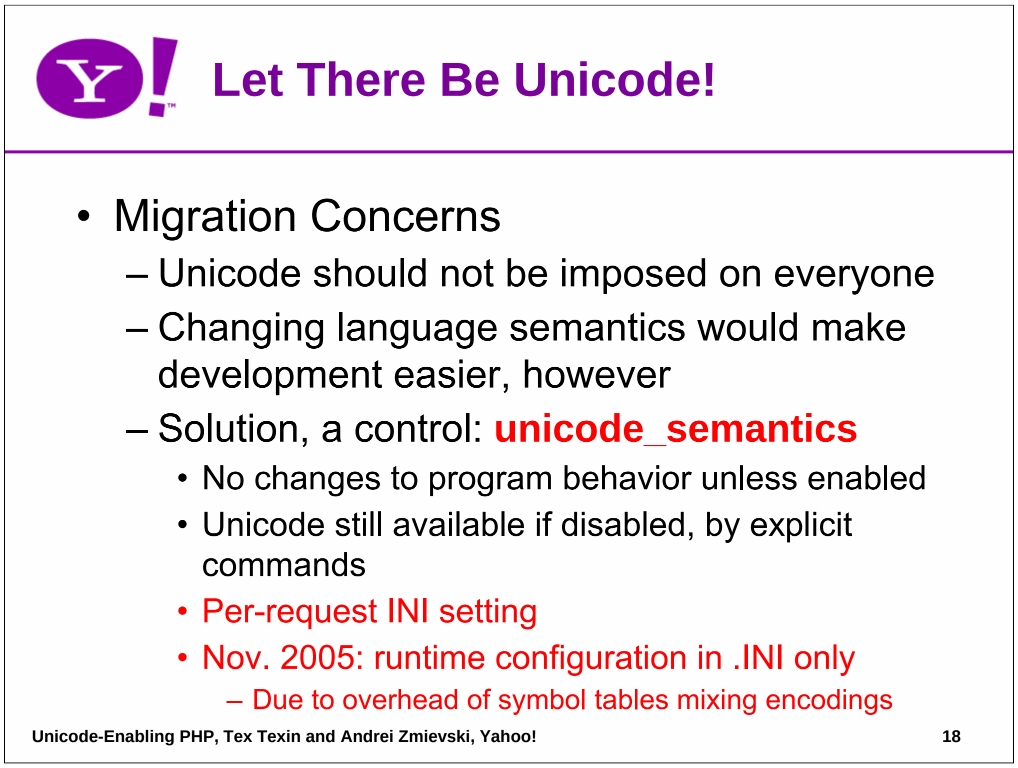


Unicode Language Requirements

- Native Unicode string data type
  - Unicode string literals
  - Casting, Type Conversions
- Operators, Functions support Unicode
  - Comparison, Indexing, Concatenation, etc.
- Fallback or Compatibility mechanisms for functions, extensions that do not support Unicode
- I/O, Transcoding, Legacy support

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 17

Native Unicode string type All string literals are Unicode unless specified otherwise Upgrading functions to understand Unicode and do the Right Thing Providing a fallback mechanism for those functions that are not upgraded yet Language features supporting Unicode as well: comparison, indexing, concatenation, etc.



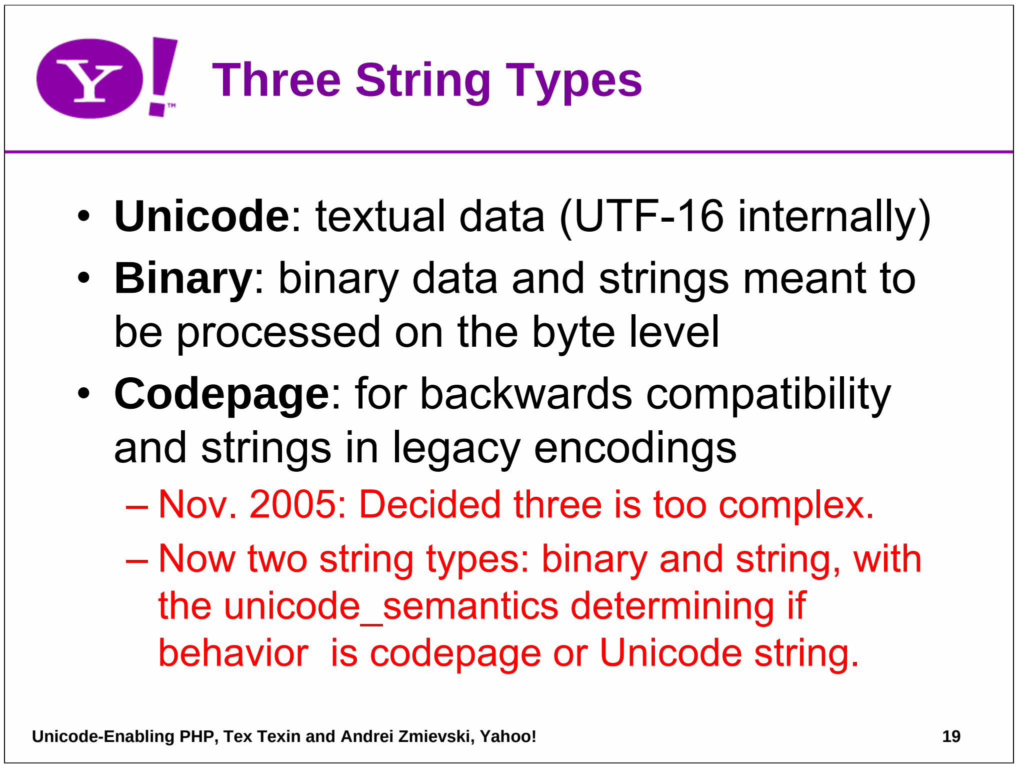
Let There Be Unicode!

- Migration Concerns
  - Unicode should not be imposed on everyone
  - Changing language semantics would make development easier, however
  - Solution, a control: **unicode_semantics**
    - No changes to program behavior unless enabled
    - Unicode still available if disabled, by explicit commands
    - Per-request INI setting
    - Nov. 2005: runtime configuration in .INI only
      - Due to overhead of symbol tables mixing encodings

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! **18**

Places the choice in the hands of the developer. Mainly affects interpretation of string literals. Can be specified as .ini setting. Unicode=off does not mean that there will be no Unicode at all. Unicode string can still be created programmatically.

Nov. 2005 decision allows default configuration of unicode_semantics off, but it can be easily upgraded with a .INI configuration change.

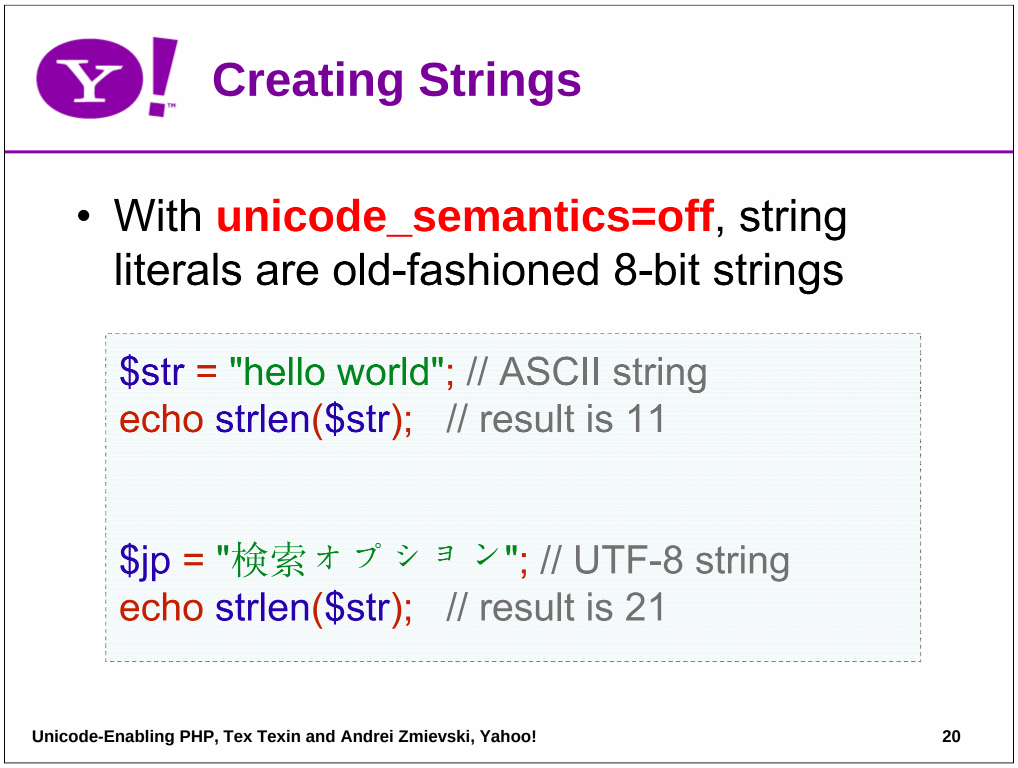


Three String Types

- **Unicode**: textual data (UTF-16 internally)
- **Binary**: binary data and strings meant to be processed on the byte level
- **Codepage**: for backwards compatibility and strings in legacy encodings
  - Nov. 2005: Decided three is too complex.
  - Now two string types: binary and string, with the unicode_semantics determining if behavior is codepage or Unicode string.

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 19

The current overloaded string type is now cleanly separated. Unicode for text, binary for byte-level data with no encoding attached to it, and codepage strings for exchange with the outside world. Binary data are things like images, PDFs, etc, but can also be used for low-level string manipulation.



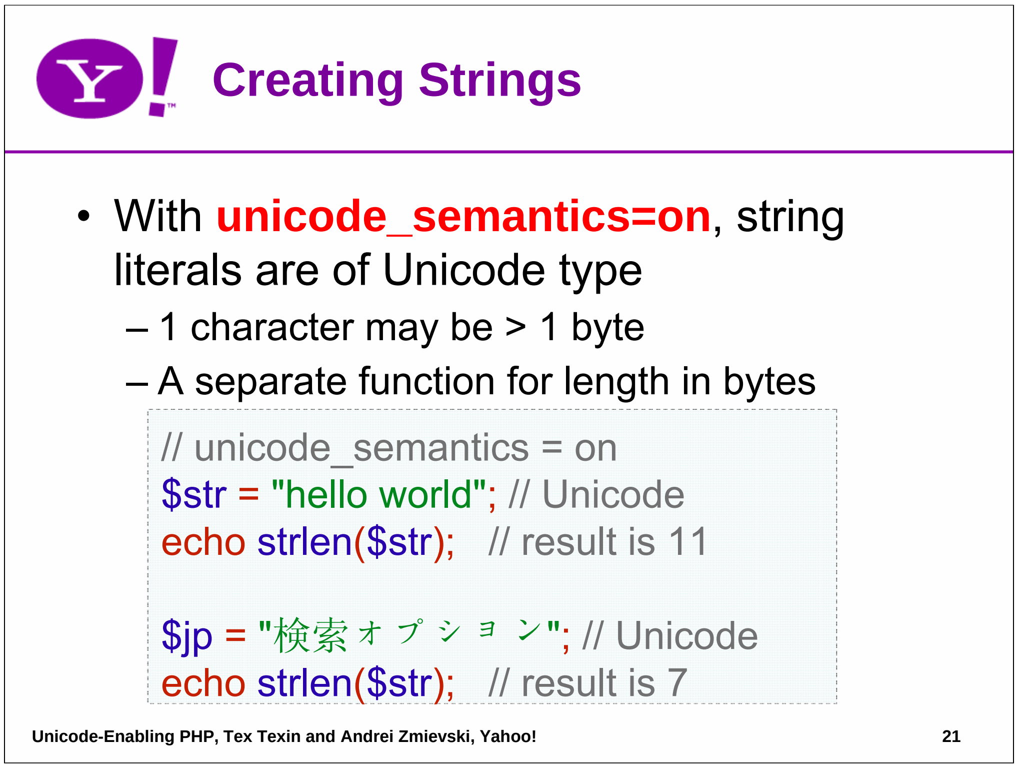
Creating Strings

- With **unicode_semantics=off**, string literals are old-fashioned 8-bit strings

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


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

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 20

This example illustrates that with **unicode_semantics** switch turned off, string literals remain 8-bit binary chunks of data, and the string functions treat them as such. The actual byte sequences depend on the encoding that the script is in, of course, but PHP does not care what it is.



Creating Strings

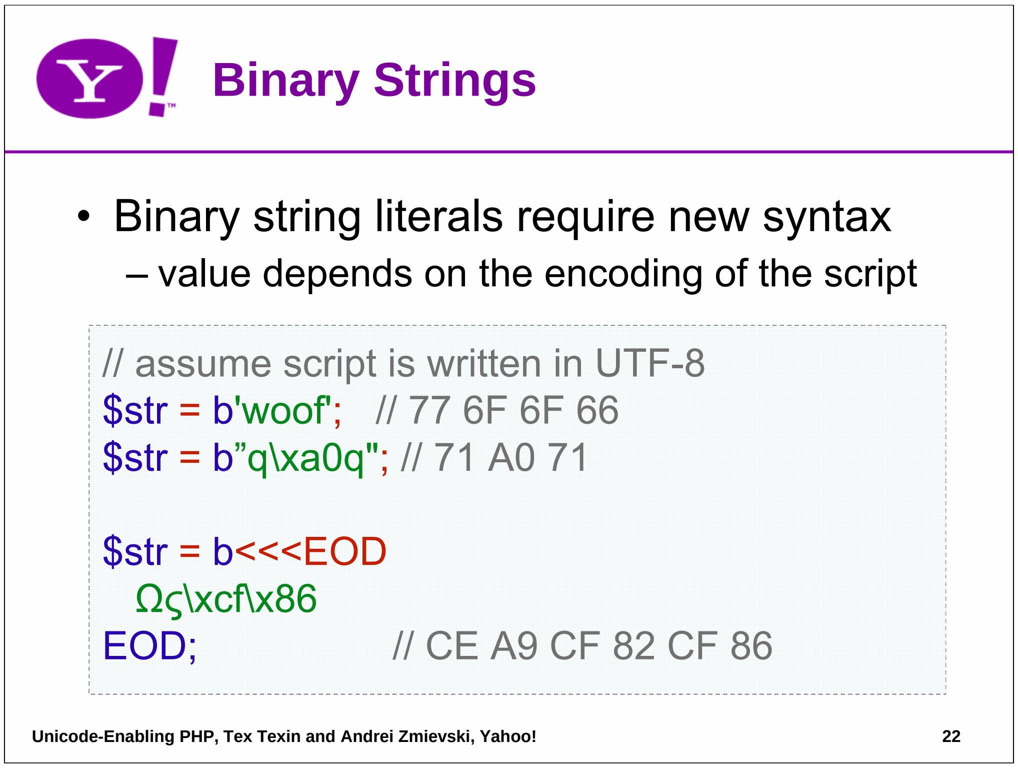
- With **unicode_semantics=on**, string literals are of Unicode type
  - 1 character may be > 1 byte
  - A separate function for length in bytes

```
// unicode_semantics = 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
```

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 21

When unicode_semantics switch is enabled, the string literals in the script are converted to Unicode during parsing stage. The conversion uses the script encoding setting specified externally or in the script itself.

Note that the **strlen()** function now returns the number of code points in the string, instead of the number of bytes. If the user needs to obtain the latter, they can use a separate function.



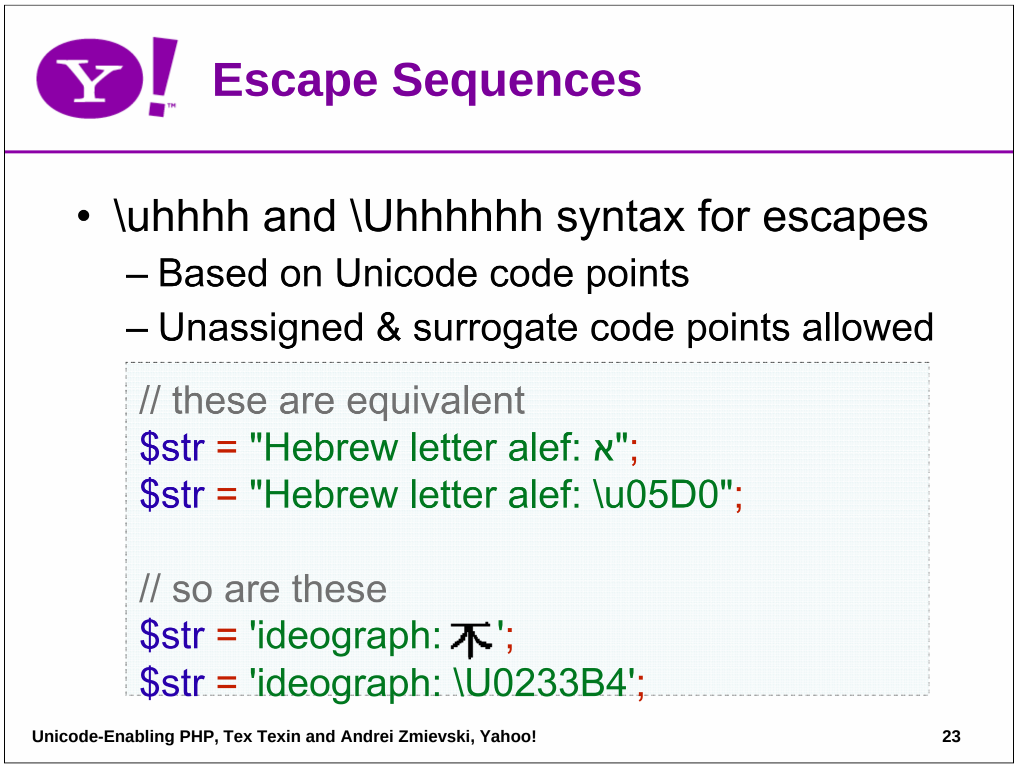
Binary Strings

- Binary string literals require new syntax
  - value depends on the encoding of the script

```
// assume script is written in UTF-8
$str = b'woof';   // 77 6F 6F 66
$str = b”q\xa0q"; // 71 A0 71

$str = b<<<EOD
   Ως\xcf\x86
EOD;                  // CE A9 CF 82 CF 86
```

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 22

Creation of binary strings requires introduction of new syntax. Binary string literals have to be prefixed with **b**. During the parsing stage, these literals are simply scanned as-is, without any conversions, so the resulting contents are a direct representation of the byte sequences in the script. Single and double-quoted strings, and heredoc syntax are all supported.



Escape Sequences

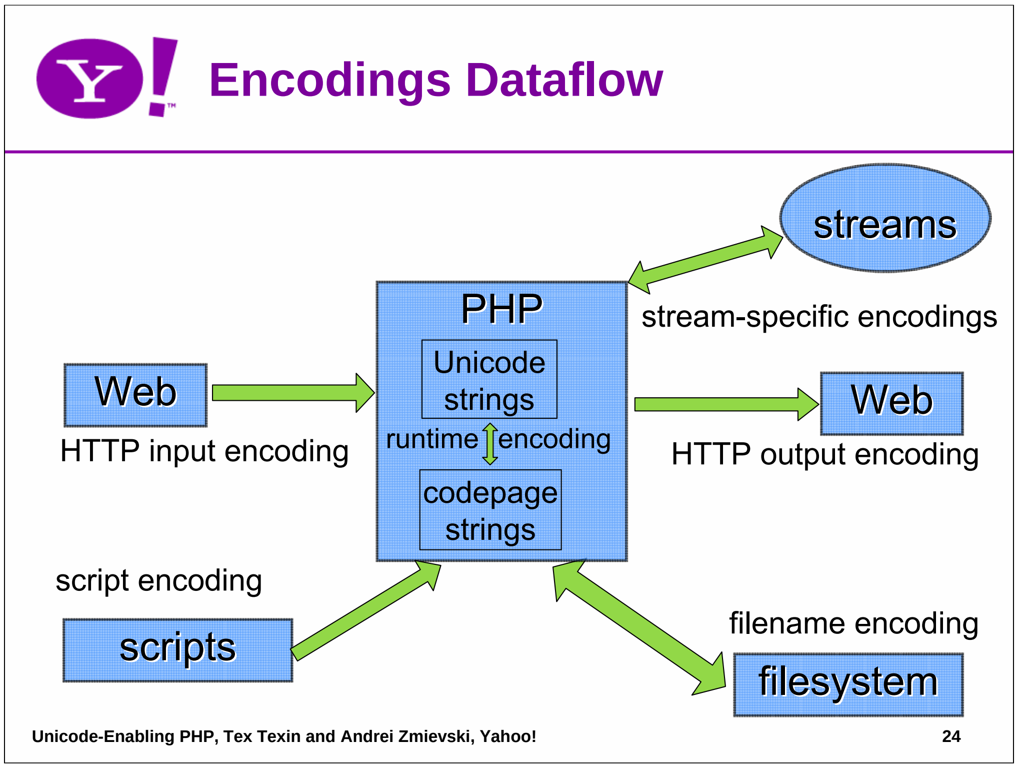
- \uhhhh and \Uhhhhhh syntax for escapes
  - Based on Unicode code points
  - Unassigned & surrogate code points allowed

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

// so are these
$str = 'ideograph: 𣎴';
$str = 'ideograph: \U0233B4';
```

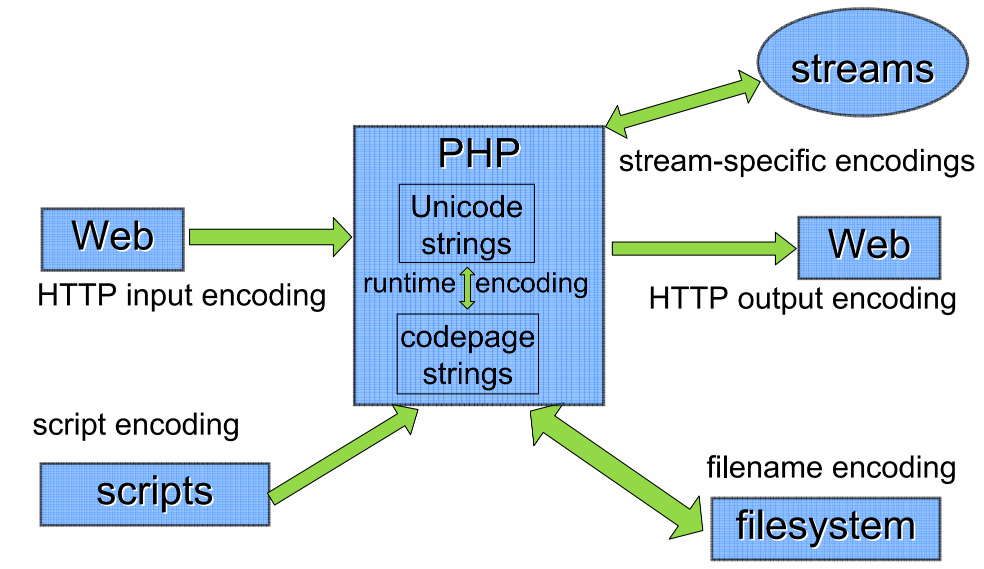
Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 23

In order to make it easy to specify Unicode characters, we introduced two new escape sequences: **\u** and **\U**, which have to be followed by 4 or 6 hexadecimal digits respectively. The digits are interpreted as a Unicode codepoint value, such that **\u05D0** represents **U+05D0**, and **\U02000B** represents **U+2000B**.

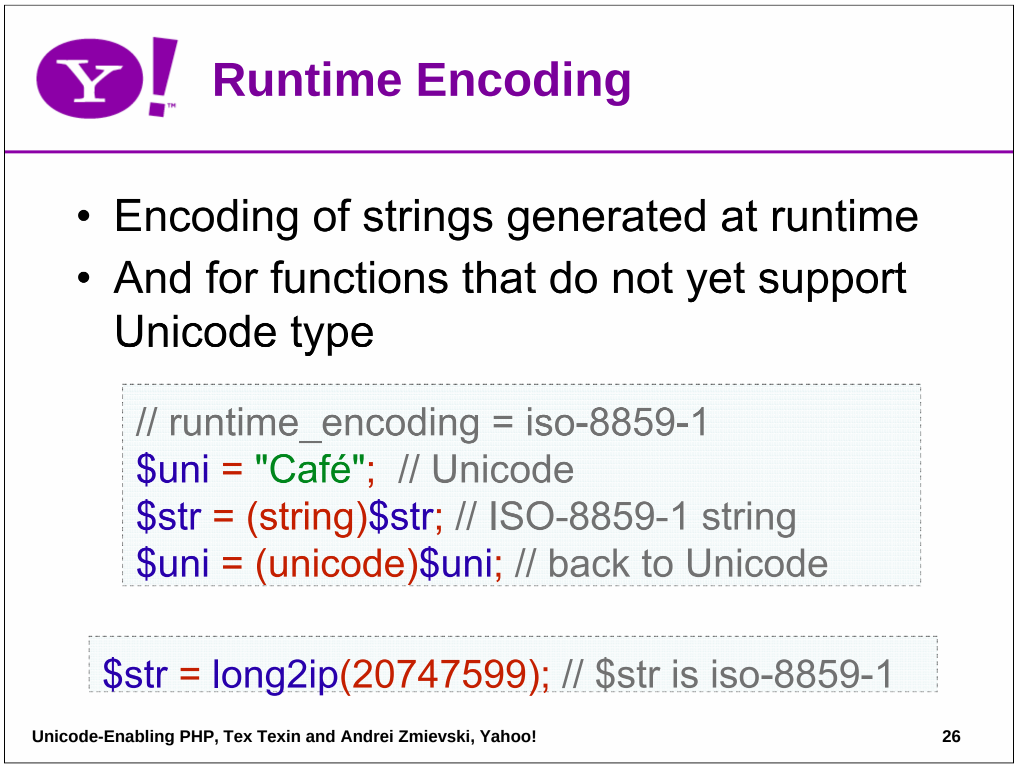
We decided to allow direct representation of surrogate pairs as well as unassigned codepoints. Our reasoning was that validation of such things is best left to the intermediate layers, rather than the parser or compiler.



Encodings Dataflow



Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 24



Runtime Encoding

- Encoding of strings generated at runtime
- And for functions that do not yet support Unicode type

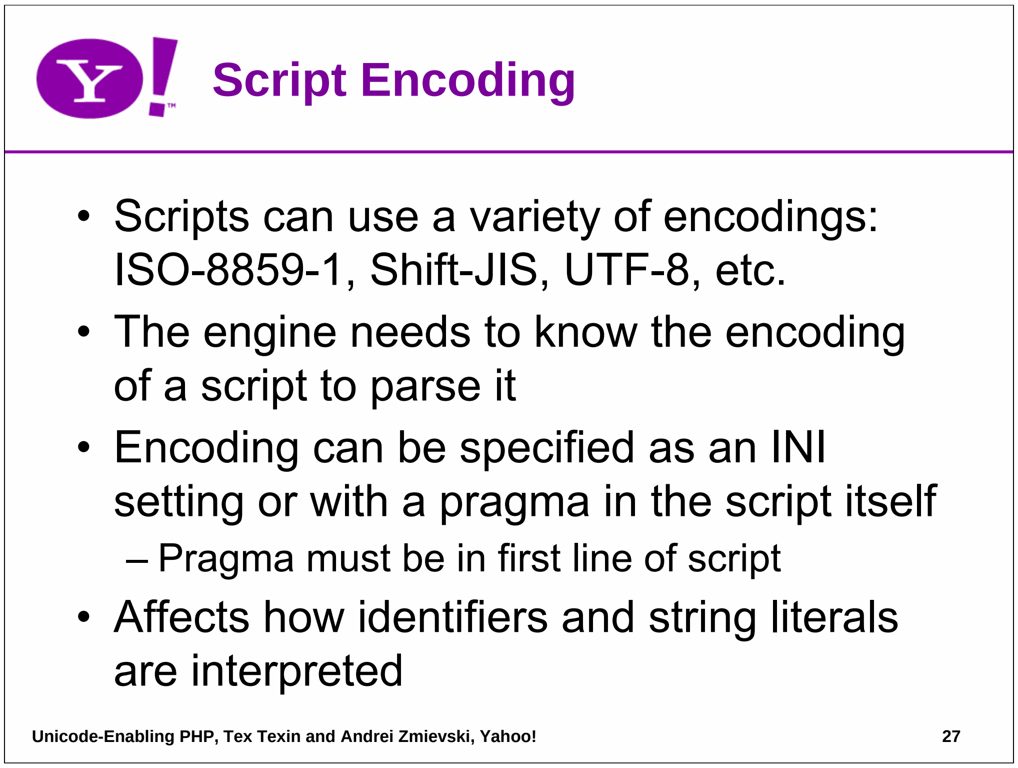
```
// runtime_encoding = iso-8859-1
$uni = "Café";  // Unicode
$str = (string)$str; // ISO-8859-1 string
$uni = (unicode)$uni; // back to Unicode
```

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

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 26

Not all the strings in PHP will be of Unicode type, of course. There are many places where codepage strings may occur and we need to know what their encoding is in order to convert them to and from UTF-16 properly. Some of the places where this might happen are casting between Unicode and codepage strings, concatenation of strings of dissimilar types, and supporting legacy functions that have not yet been upgraded to understand Unicode type.

Why can't we make these legacy functions return Unicode type strings by default? Because we cannot assume that the semantics are the same everywhere. These may have a completely valid reason for returning codepage strings - one example being **iconv()** function.



Script Encoding

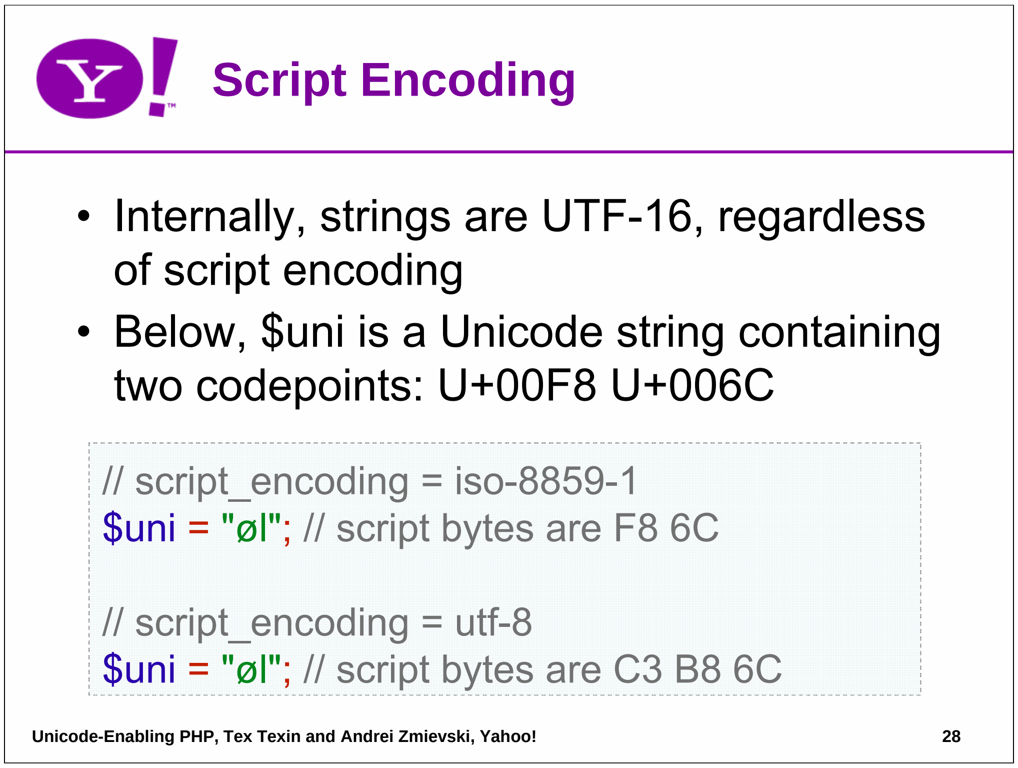
- Scripts can use a variety of encodings: ISO-8859-1, Shift-JIS, UTF-8, etc.
- The engine needs to know the encoding of a script to parse it
- Encoding can be specified as an INI setting or with a pragma in the script itself
  - Pragma must be in first line of script
- Affects how identifiers and string literals are interpreted

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 27

We felt that making programmers use a single encoding for their scripts, such as UTF-8, would not be in the spirit of PHP. Instead, we allow a couple of ways to tell PHP what encoding the scripts are written in. If all of the scripts in a certain installation are in a common encoding, then the best way to specify it is an INI setting (a value in a configuration file read by PHP on startup and each request).

Alternatively, the script encoding can be specified by a pragma in the script itself. The pragma is similar to the XML header, in that it has to be the very first line of the script and must be written in ASCII-compatible encoding.

The script encoding is used to convert identifiers and string literals encountered by parser into Unicode form.



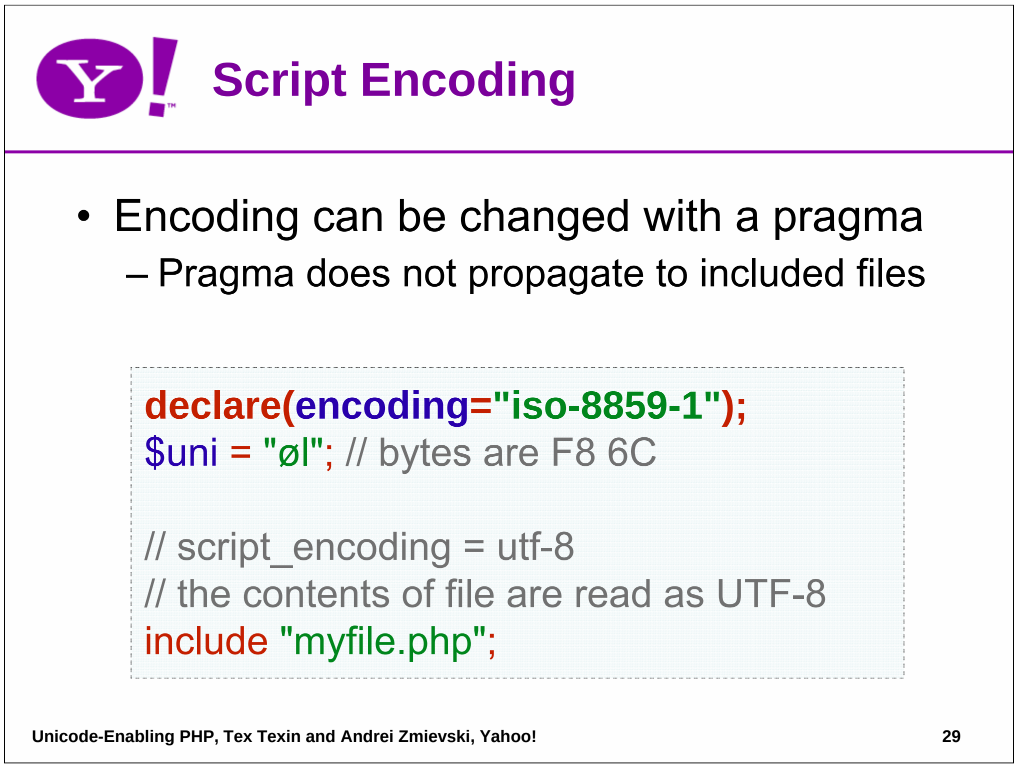
Script Encoding

- Internally, strings are UTF-16, regardless of script encoding
- Below, $uni is a Unicode string containing two codepoints: U+00F8 U+006C

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

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

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 28

In this example, we see a short script that visually appears the same, but is written in different encodings. Script encoding here is specified in an external configuration file. In both cases, the parser uses this script encoding to convert string literals to Unicode form internally.



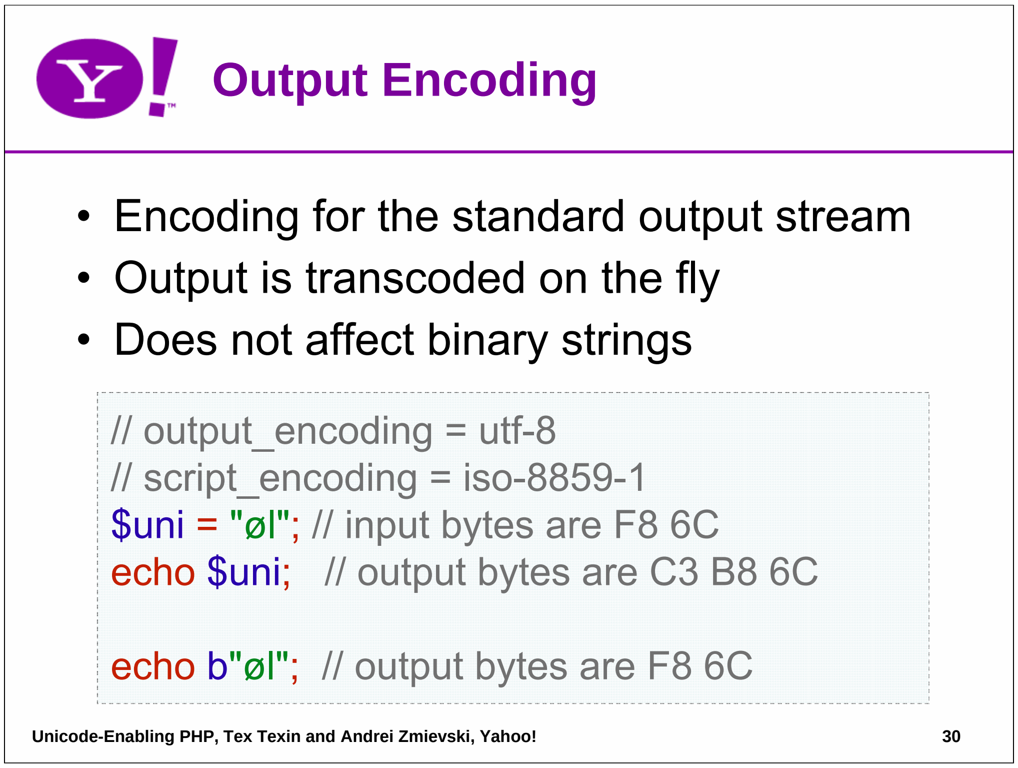
Script Encoding

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

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

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

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 29

This example illustrates the usage of script encoding pragma. The external configuration file sets **script_encoding** to UTF-8, but the pragma in the file overrides it. The string literal is parsed as ISO-8859-1 byte sequence. Note that the pragma does not propagate to the scripts included from this one. The reason for this is because we cannot be sure of the encoding of third-party libraries that the script may include.



Output Encoding

- Encoding for the standard output stream
- Output is transcoded on the fly
- Does not affect binary strings

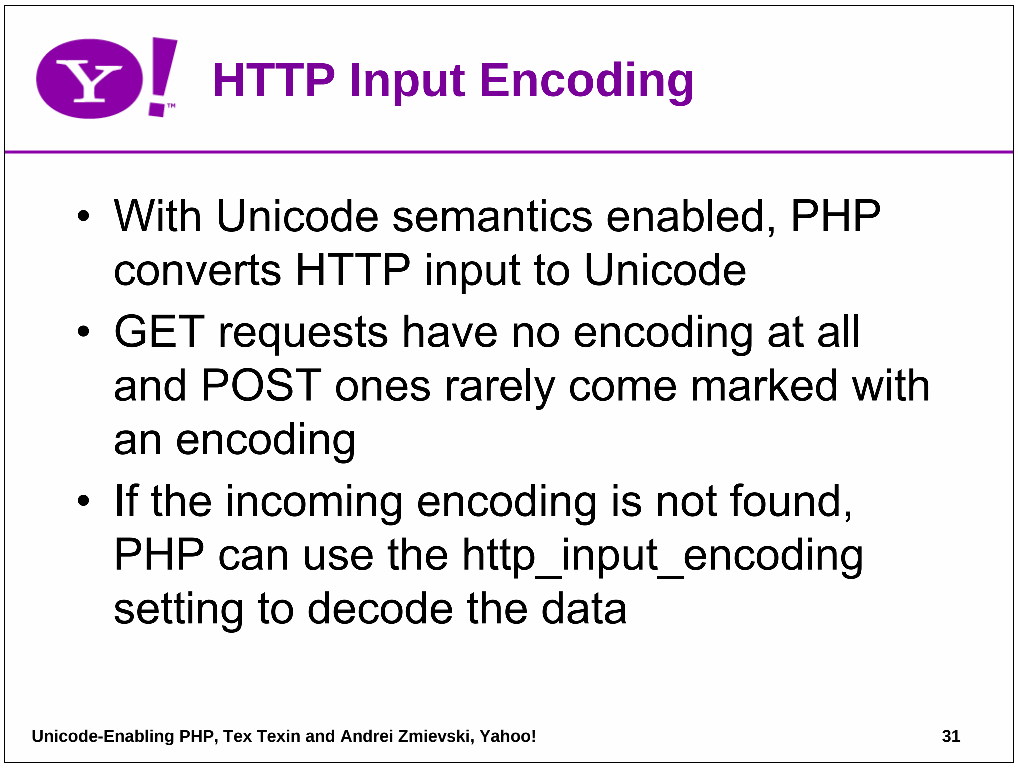
```
// output_encoding = utf-8
// script_encoding = iso-8859-1
$uni = "øl"; // input bytes are F8 6C
echo $uni;   // output bytes are C3 B8 6C

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

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 30

The majority of the scripts simply use **echo** or **print** or related functions to send the output to the browser. All of it is supposed to be in the same encoding, so it makes sense to have a setting that PHP can use to automatically transcode the output of the script. This setting also changes the charset parameter in the HTTP headers, i.e. if **output_encoding** is UTF-8, then the default HTTP header sent out by PHP will be:

```
Content-Type: text/html; charset=utf-8
```

Note that binary strings are not transcoded and are simply passed through as is. We assume that the user knows what they are doing when working with binary strings and we do not attempt to second guess them.

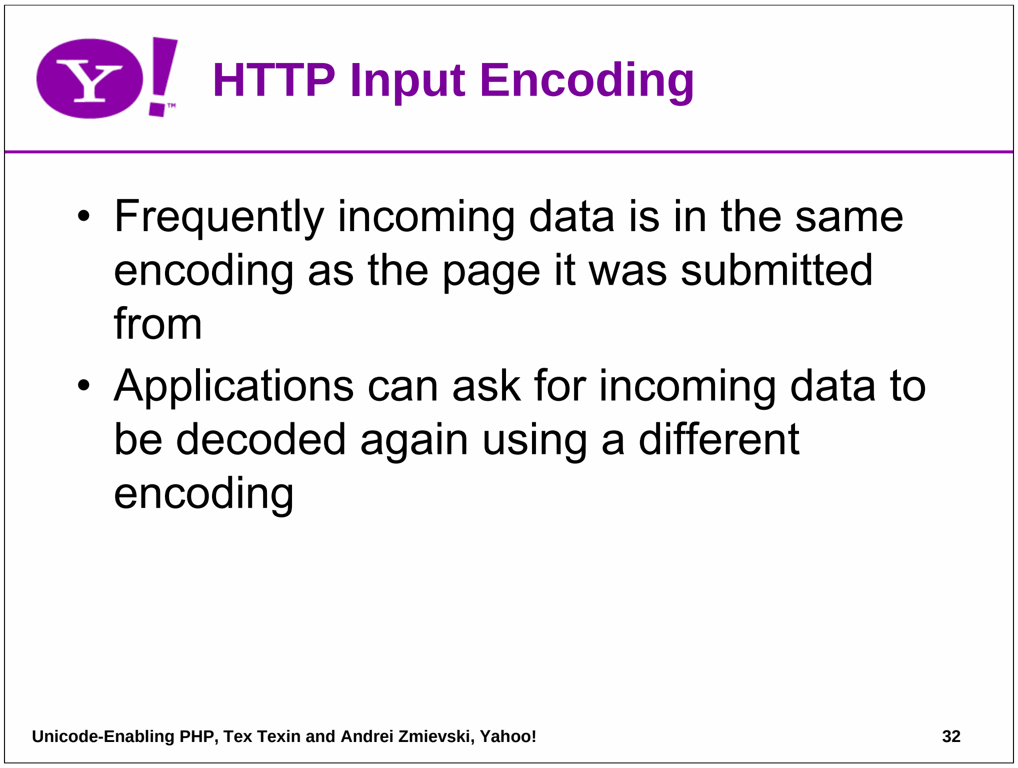


HTTP Input Encoding

- With Unicode semantics enabled, PHP converts HTTP input to Unicode
- GET requests have no encoding at all and POST ones rarely come marked with an encoding
- If the incoming encoding is not found, PHP can use the http_input_encoding setting to decode the data

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 31

Correctly decoding HTTP input is somewhat of an unsolved problem. Using automatic encoding detection is not very reliable, due to small amounts of data in most of the requests. The best we can do is prepare the data for user based on the previously specified setting, and then provide a way to easily re-decode the data in case we were wrong.



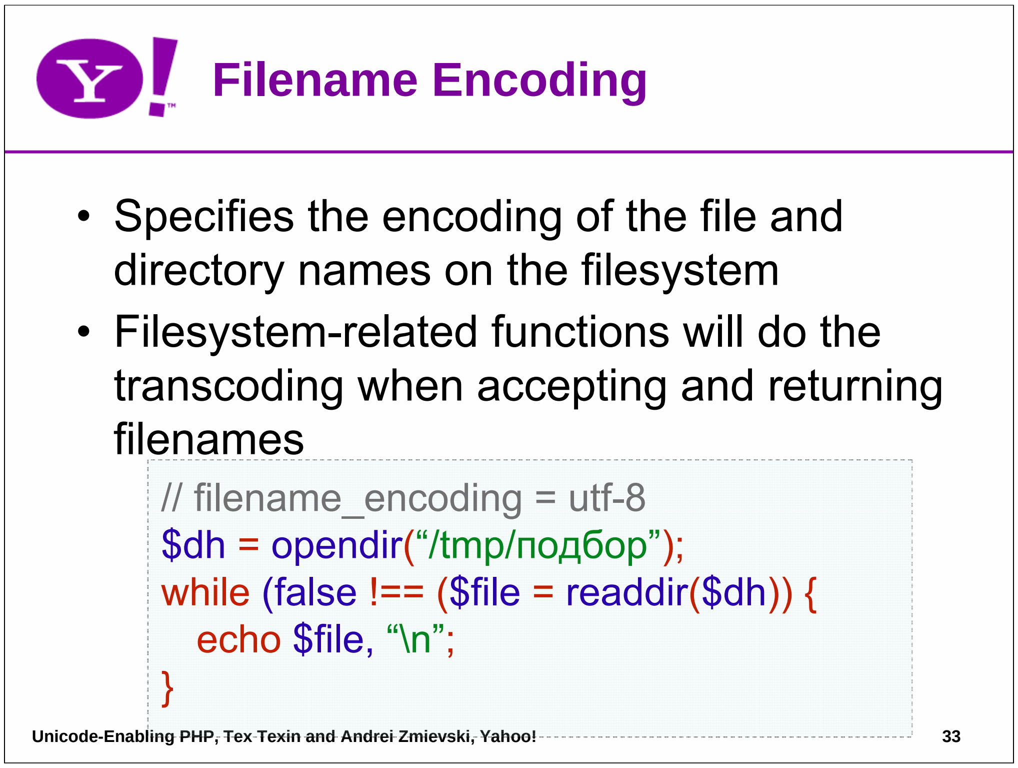
HTTP Input Encoding

- Frequently incoming data is in the same encoding as the page it was submitted from
- Applications can ask for incoming data to be decoded again using a different encoding

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 32

One heuristic that we can use is that most modern browsers will send back the data in the same encoding as they received it in, i.e. if the **output_encoding** was UTF-8, then the incoming data is very likely to be in UTF-8 as well.

Also, if the encoding is passed as an application request variable, the application can ask PHP to re-decode the data based on this explicitly specified encoding.



Filename Encoding

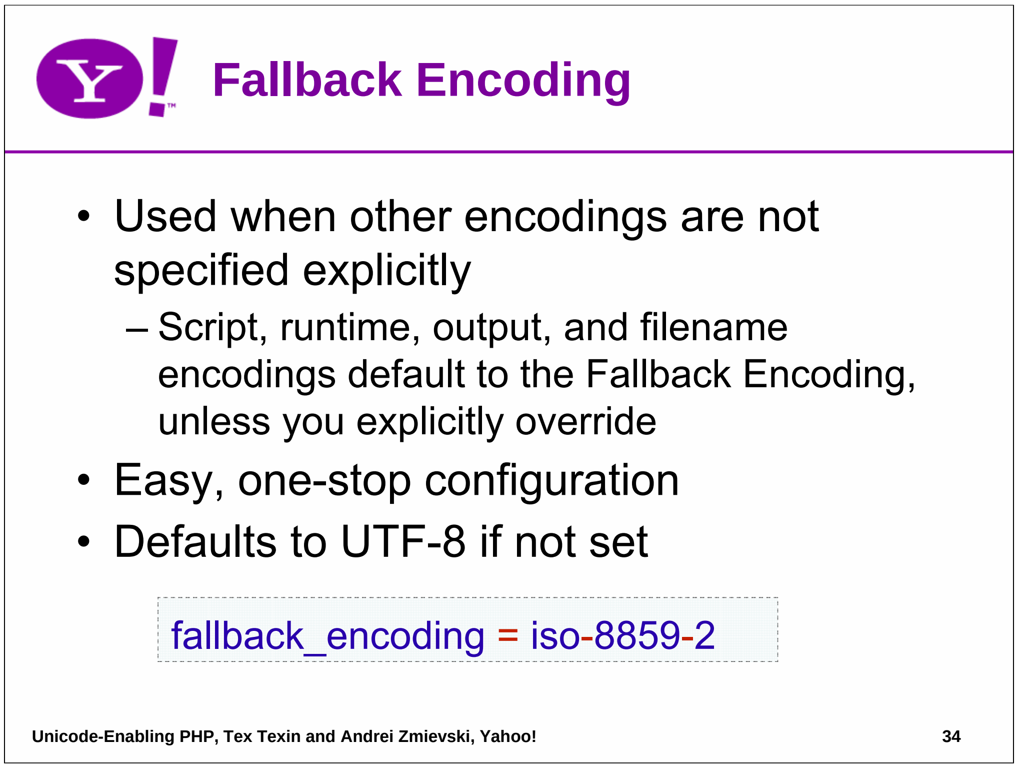
- Specifies the encoding of the file and directory names on the filesystem
- Filesystem-related functions will do the transcoding when accepting and returning filenames

```
// filename_encoding = utf-8
$dh = opendir("/tmp/подбор");
while (false !== ($file = readdir($dh)) {
   echo $file, "\n";
}
```

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 33

Modern file systems should be able to support one of the Unicode encodings, such as UTF-16 or UTF-8. PHP, however, runs on such a variety of platforms that it is impossible to predict what particular encoding the platform filesystem uses. Thus, we placed the choice in the hands of the user via the **filename_encoding** setting.

The transcoding of the filenames between Unicode and filesystem encoding is not automatic. All the functions that deal with filenames need to be upgraded to do the conversion.



Fallback Encoding

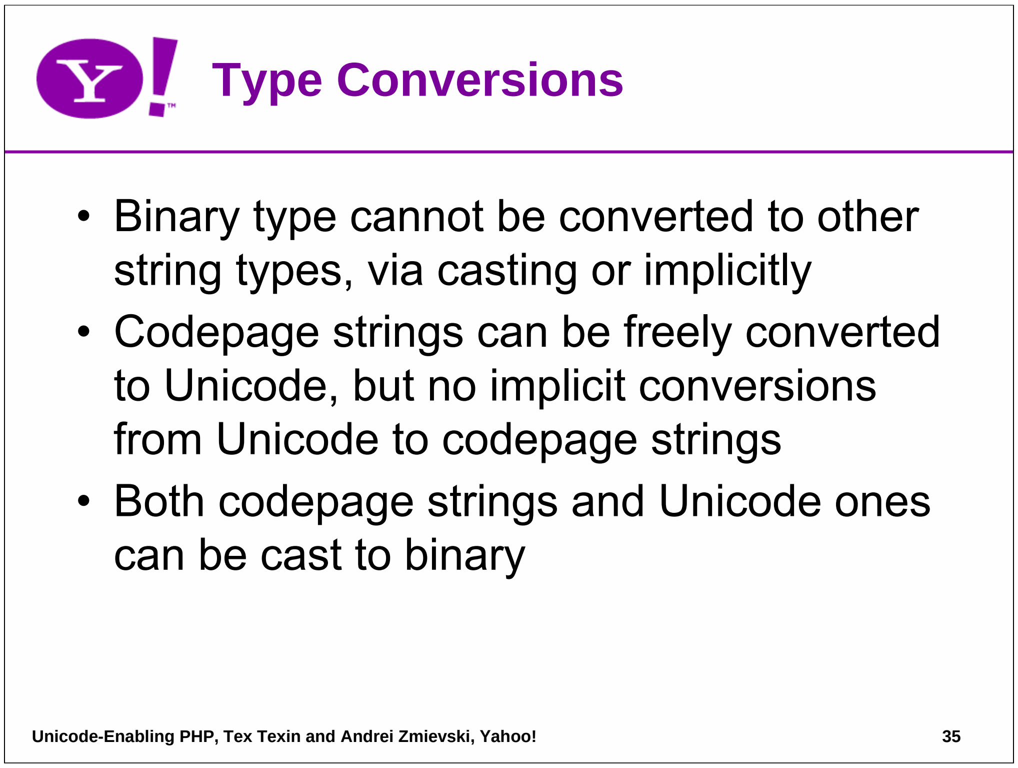
- Used when other encodings are not specified explicitly
  - Script, runtime, output, and filename encodings default to the Fallback Encoding, unless you explicitly override
- Easy, one-stop configuration
- Defaults to UTF-8 if not set

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

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 34

We thought it was prudent to have a fallback encoding setting that can be used when the other encodings are not specified explicitly, and that can serve as a one-stop configuration value if the application mainly works in a single encoding (such as many legacy apps). The fallback encoding itself would default to UTF-8.

If fallback encoding is set to ISO-8859-2, for example, the script, runtime, output, and filename encodings are also ISO-8859-2 unless they are overridden.



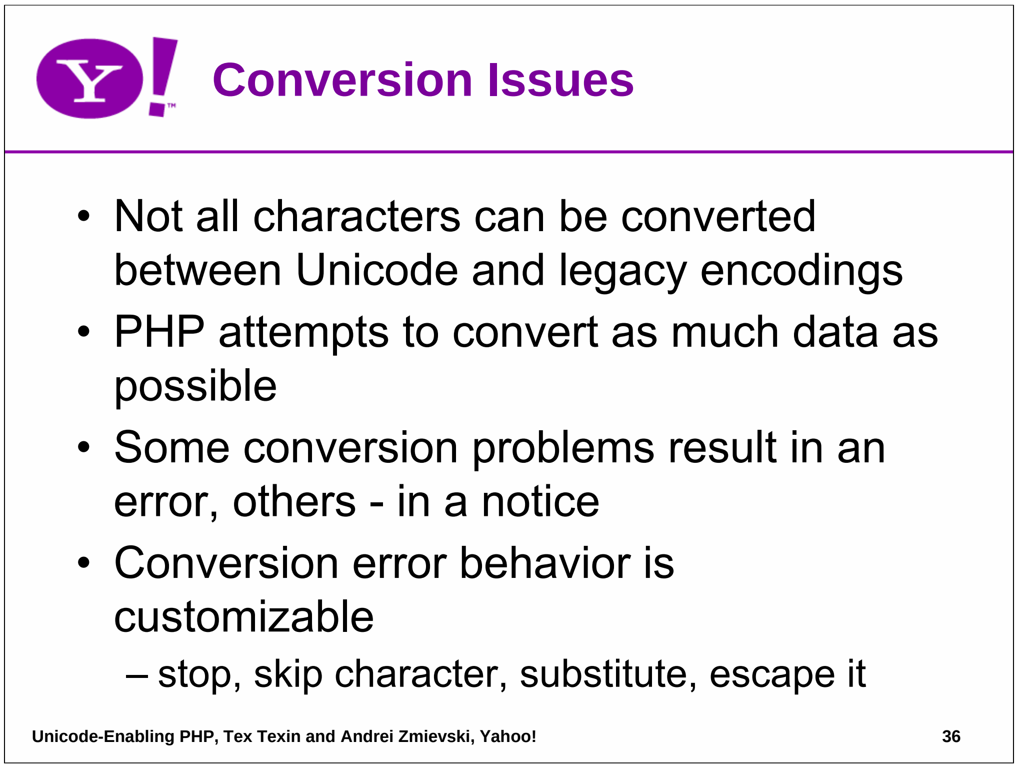
Type Conversions

- 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 from Unicode to codepage strings
- Both codepage strings and Unicode ones can be cast to binary

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 35

Conversions between Unicode, binary, and codepage strings may happen in a variety of places during the execution of a script, explicitly or implicitly. An example of an explicit conversion is casting, and an implicit one happens when strings of dissimilar types are concatenated together.

We cannot convert binary type to any other string type because it has no encoding associated with it, but inverse can be done without a problem. No implicit conversion from Unicode type to codepage one is to maintain data integrity, since Unicode is a more precise character set than any codepage.



Conversion Issues

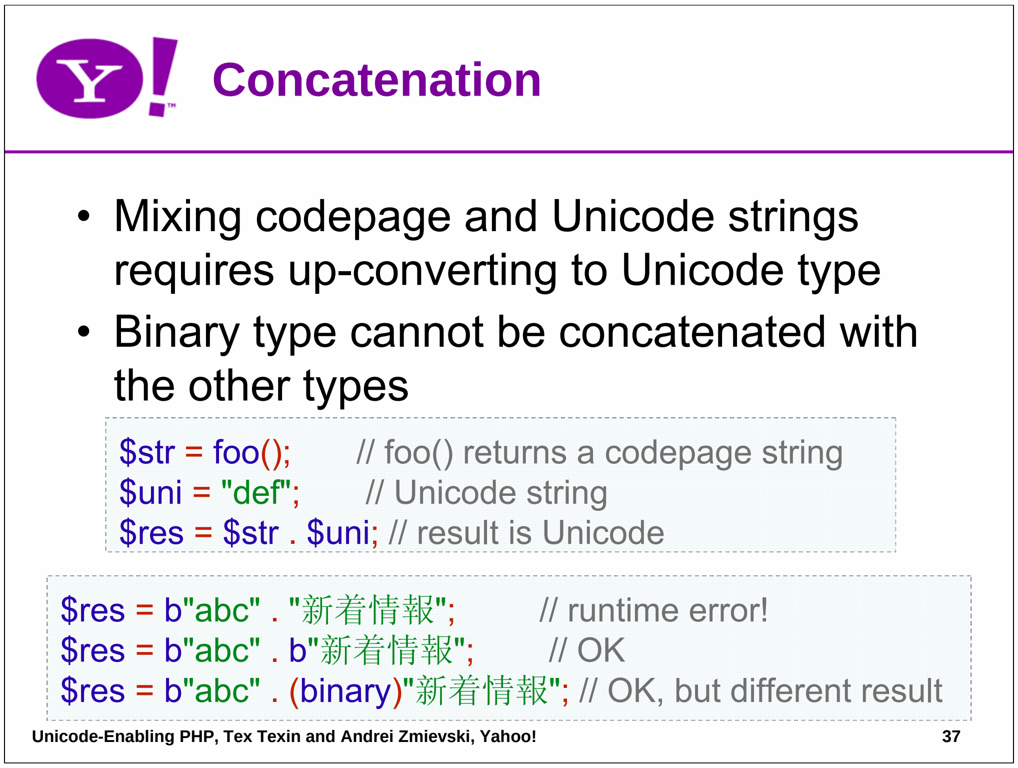
- Not all characters can be converted between Unicode and legacy encodings
- PHP attempts to convert as much data as possible
- Some conversion problems result in an error, others - in a notice
- Conversion error behavior is customizable
  - stop, skip character, substitute, escape it

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 36

Several issues may come up during the conversion procedure, such as a Unicode character that cannot be represented in a particular codepage, or an invalid or truncated sequence found in the source encoding. The behavior of PHP in case of such an error can be customized via a couple of settings.

The main setting controls what PHP should do when it encounters an invalid or unassigned character: stop conversion, skip the character, substitute it with another one, or escape it in one of several possible ways. Another setting specifies the substitution character itself, as a Unicode code point.

If an illegal byte sequence is encountered, PHP will always stop the conversion process and issue an error.



Concatenation

- Mixing codepage and Unicode strings requires up-converting to Unicode type
- Binary type cannot be concatenated with the other types

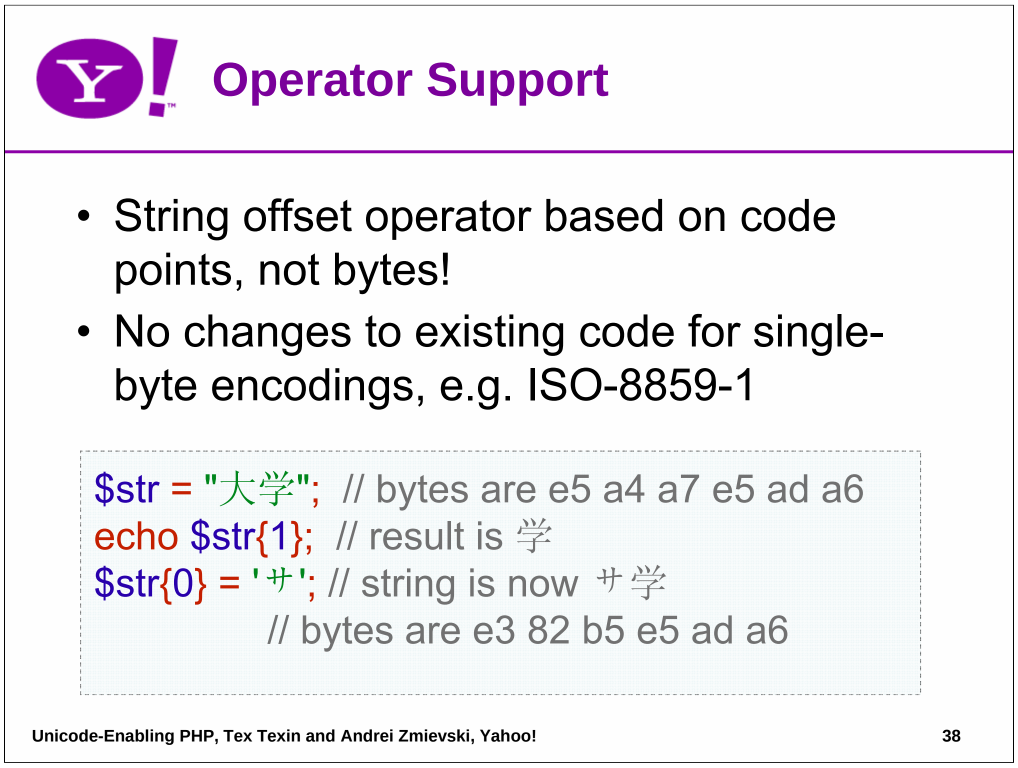
```
$str = foo();      // foo() returns a codepage string
$uni = "def";      // Unicode string
$res = $str . $uni; // result is Unicode
```

```
$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
```

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 37

The concatenation operator follows the outlined conversion semantics. Applying it to strings of the same type simply joins the strings together. If one of the operands is a codepage string and the other is a Unicode one, the codepage string will be converted to Unicode using the runtime encoding setting.

No such conversion is performed when one of the operands is a binary string, because we want to make sure that the semantics of the conversion are handled by the programmer. One may want to convert a Unicode string to another encoding before concatenating it, for example.



Operator Support

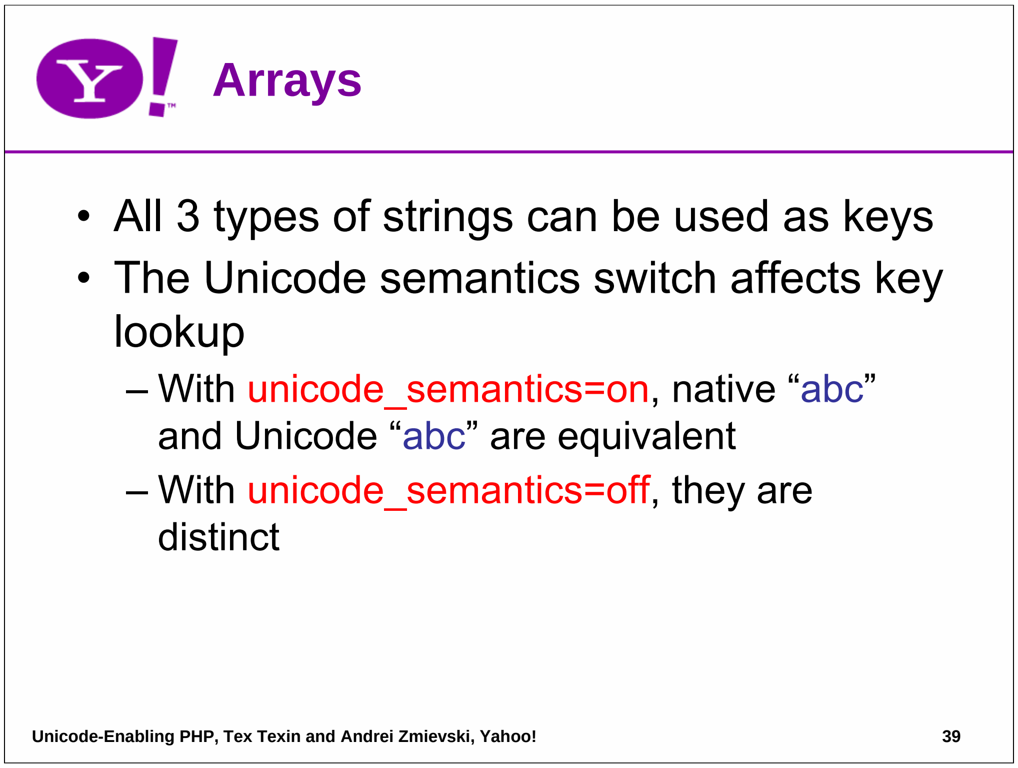
- String offset operator based on code points, not bytes!
- No changes to existing code for single-byte encodings, e.g. ISO-8859-1

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

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 38

Following the general rules, the string offset indexing operator works on the code points, not on bytes or code units. While this does affect performance since it is not possible to extract the required codepoint with simple indexing, we felt it was better than making the user worry about handling surrogate pairs in their code. Scripts that used to work with only single-byte encodings, such as ASCII or ISO-8859-1, would not be affected by this change, and those working with multi-byte encodings most likely do not use this operator at all.

If byte-level indexing is desired, the proper approach would be to convert the string to codepage string using UTF-16 encoding or cast it to binary type.



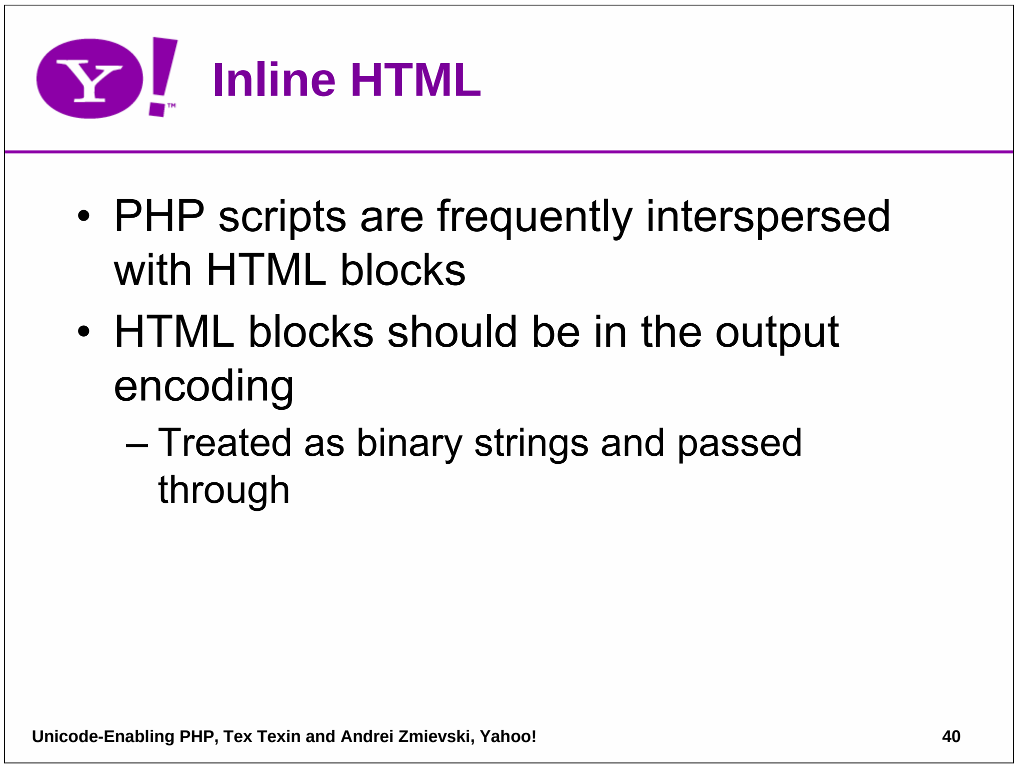
Arrays

- All 3 types of strings can be used as keys
- The Unicode semantics switch affects key lookup
  - With unicode_semantics=on, native “abc” and Unicode “abc” are equivalent
  - With unicode_semantics=off, they are distinct

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 39

PHP’s arrays are actually ordered maps, which associate a key with a value. The keys can be either integers or strings. This type is optimized in several ways, so that you can use it as a real array, vector, hashtable, dictionary, queue, etc.

We allow all string types to be used as keys. However, when Unicode semantics are enabled, it is desirable to have Unicode **“abc”** and codepage **“abc”** strings mapping to the same value. This helps with migrating scripts to support Unicode, since old legacy strings floating around can be used alongside Unicode strings and resolve equivalently. No such folding is done when Unicode semantics are turned off, for performance and compatibility reasons.



Inline HTML

- PHP scripts are frequently interspersed with HTML blocks
- HTML blocks should be in the output encoding
  - Treated as binary strings and passed through

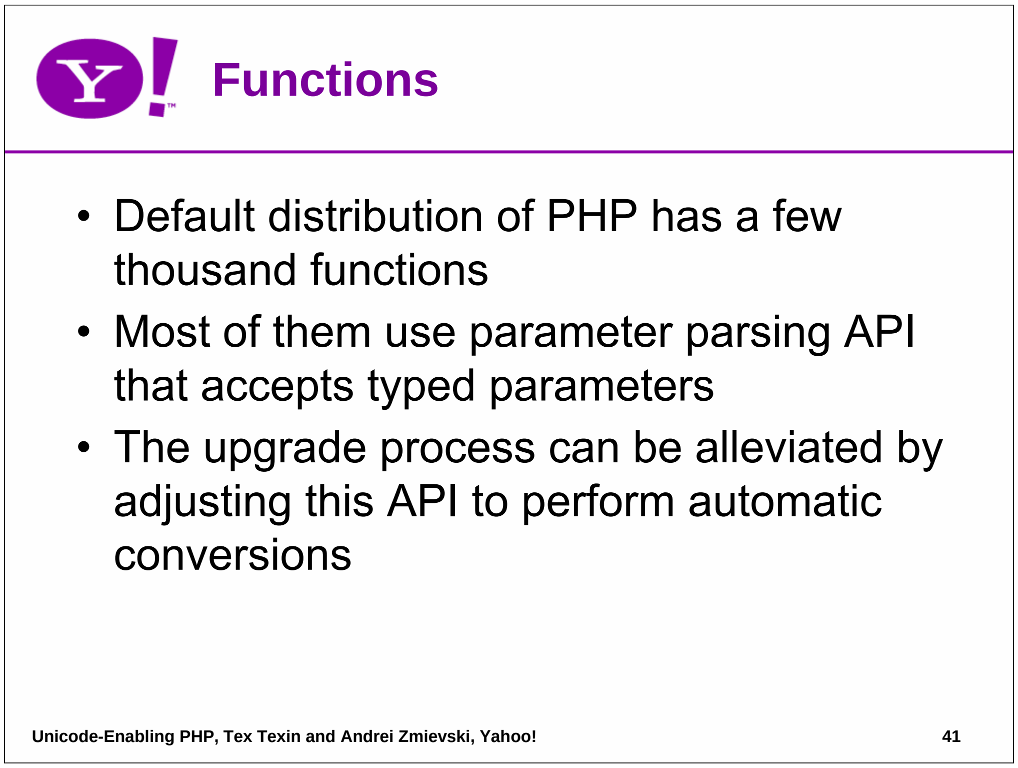
Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 40

At the top 2 or 3 layers of the previously mentioned architecture, PHP blocks are embedded inside HTML pages, e.g.

```
<h1>Hello, <?php echo get_user_name(); ?></h1>
```

The HTML content is actually processed by the PHP engine, resulting in an implicit **print** statement internally. The encoding of the HTML content should already be the same as the output encoding specified by the user - anything else would not make sense, e.g. having HTML content in ISO-8859-1 but specifying **output_encoding** as UTF-8.

Relying on this assumption, we can simply treat the HTML blocks as binary strings and hand them off to PHP's output layer, which passes them through without any transcoding, as discussed before.



Functions

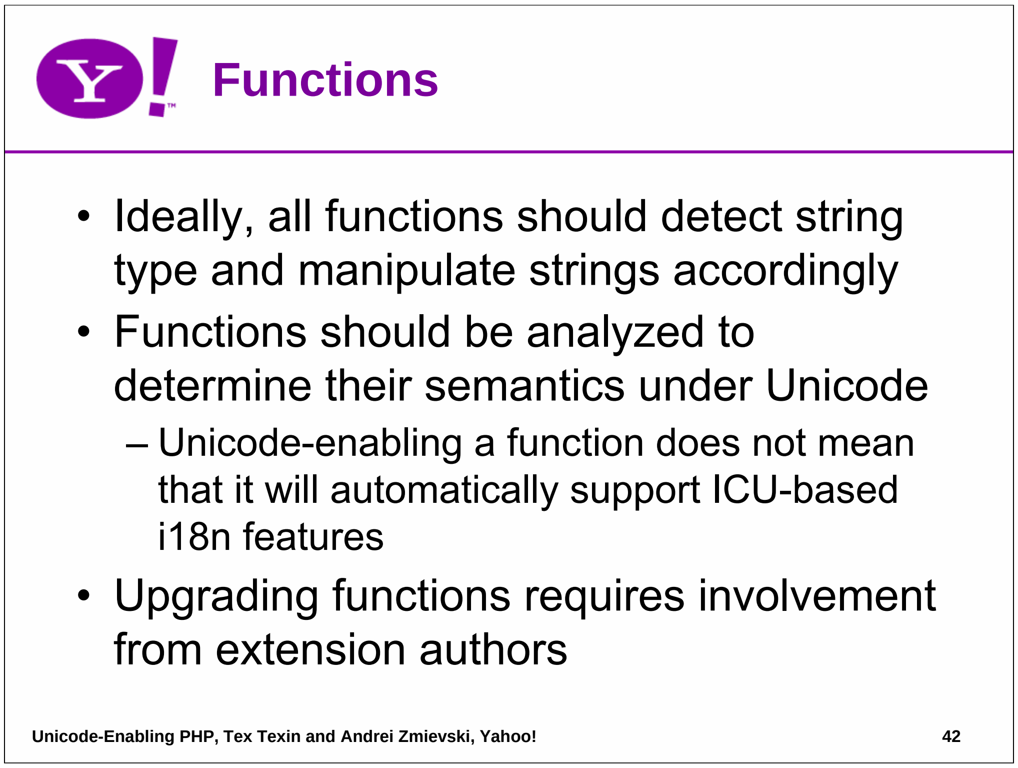
- Default distribution of PHP has a few thousand functions
- Most of them use parameter parsing API that accepts typed parameters
- The upgrade process can be alleviated by adjusting this API to perform automatic conversions

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 41

As mentioned previously, PHP comes with around 80 extensions which together offer a few thousand functions. Those functions that accept or return string values should be upgraded to work properly with new Unicode and binary string types. Most of them use parameter parsing API that uses a format string to specify what parameter types the function expects.

We can simplify the upgrade process by making the parsing API perform on-the-fly parameter conversion. For example, if a Unicode string is passed to a function expecting a codepage string, the engine will attempt to convert the Unicode string using the runtime encoding. The inverse happens for functions that expect a Unicode string, and are passed a codepage one.

As a result, all of the functions will be able to accept Unicode parameters using this fallback mode until their behavior is upgraded to handle Unicode, codepage, and binary types explicitly.

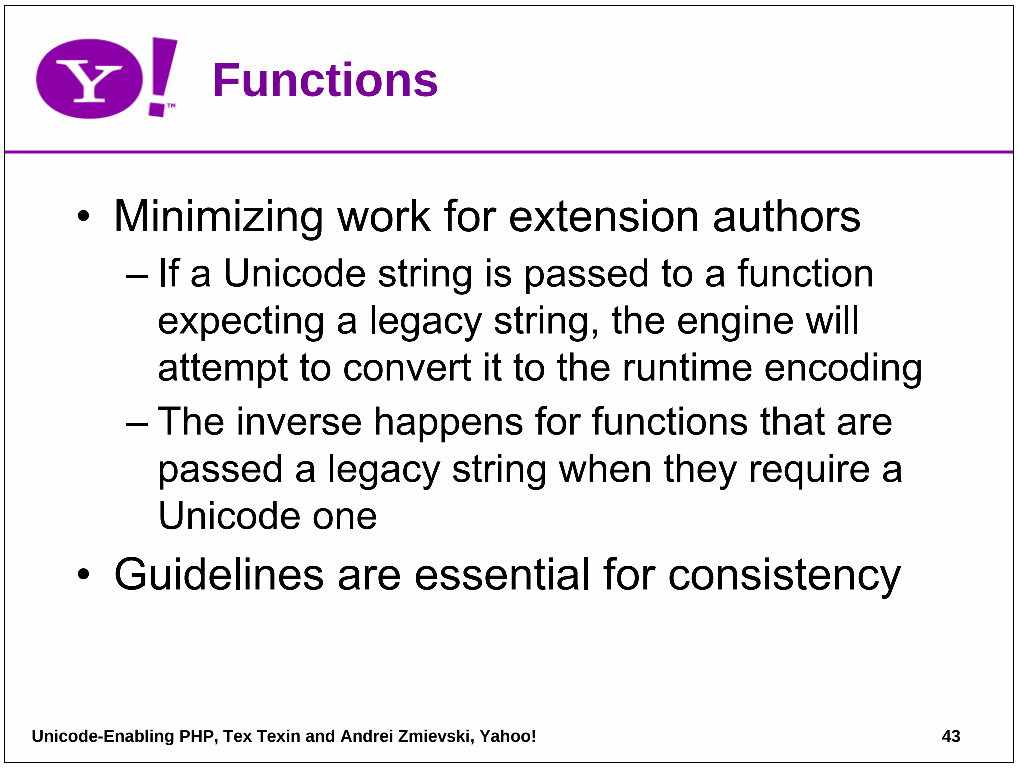


Functions

- Ideally, all functions should detect string type and manipulate strings accordingly
- Functions should be analyzed to determine their semantics under Unicode
  - Unicode-enabling a function does not mean that it will automatically support ICU-based i18n features
- Upgrading functions requires involvement from extension authors

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 42

All the core functions should be carefully analyzed to determine their semantics when working with Unicode strings. Making a function Unicode-enabled does not mean that it will automatically support ICU-based i18n features. For example, strftime() relies on POSIX locale information and while we can make it accept Unicode format strings, you should use ICU-specific date formatting mechanisms for most cases.



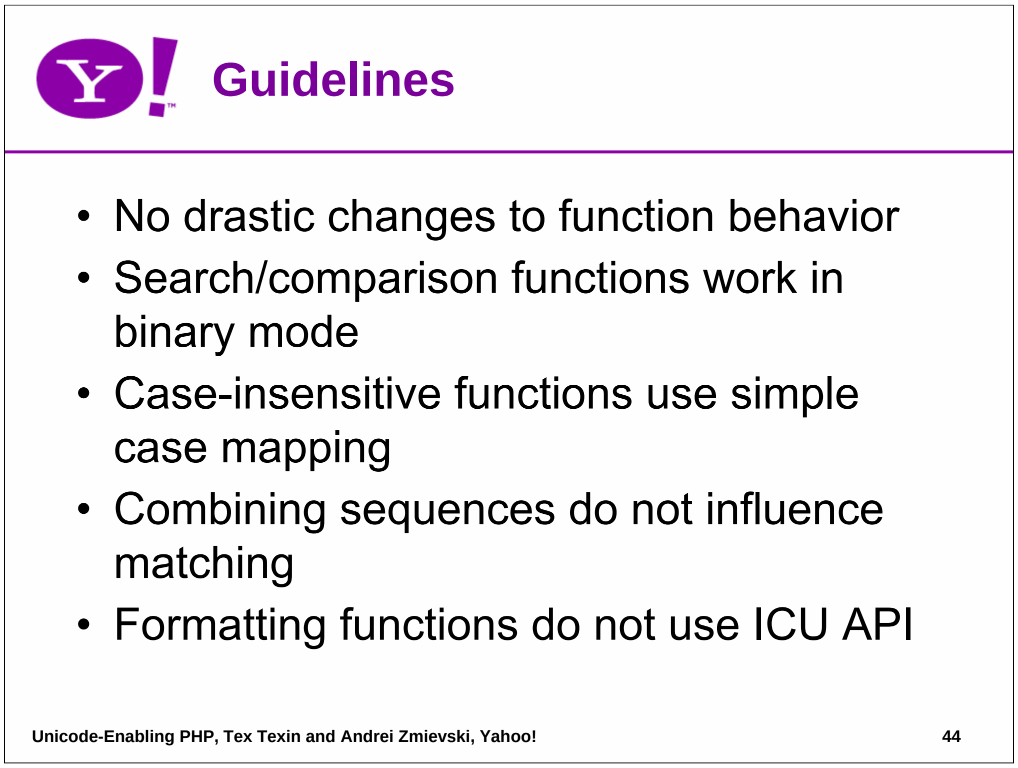
Functions

- Minimizing work for extension authors
  - If a Unicode string is passed to a function expecting a legacy string, the engine will attempt to convert it to the runtime encoding
  - The inverse happens for functions that are passed a legacy string when they require a Unicode one
- Guidelines are essential for consistency

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 43

It is impossible to upgrade all of the functions at once. Each one has to be analyzed to determine what its behavior should be when it is passed a Unicode or a binary string. For example, **trim()** function currently removes a certain set of whitespace characters from the beginning and end of a string. A question comes up, should this set be extended to cover additional Unicode whitespace characters when **trim()** is passed a Unicode string?

To avoid haphazard approach to the migration, we developed a set of guidelines that the extension authors should adhere to as closely as possible.



Guidelines

- No drastic changes to function behavior
- Search/comparison functions work in binary mode
- Case-insensitive functions use simple case mapping
- Combining sequences do not influence matching
- Formatting functions do not use ICU API

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 44

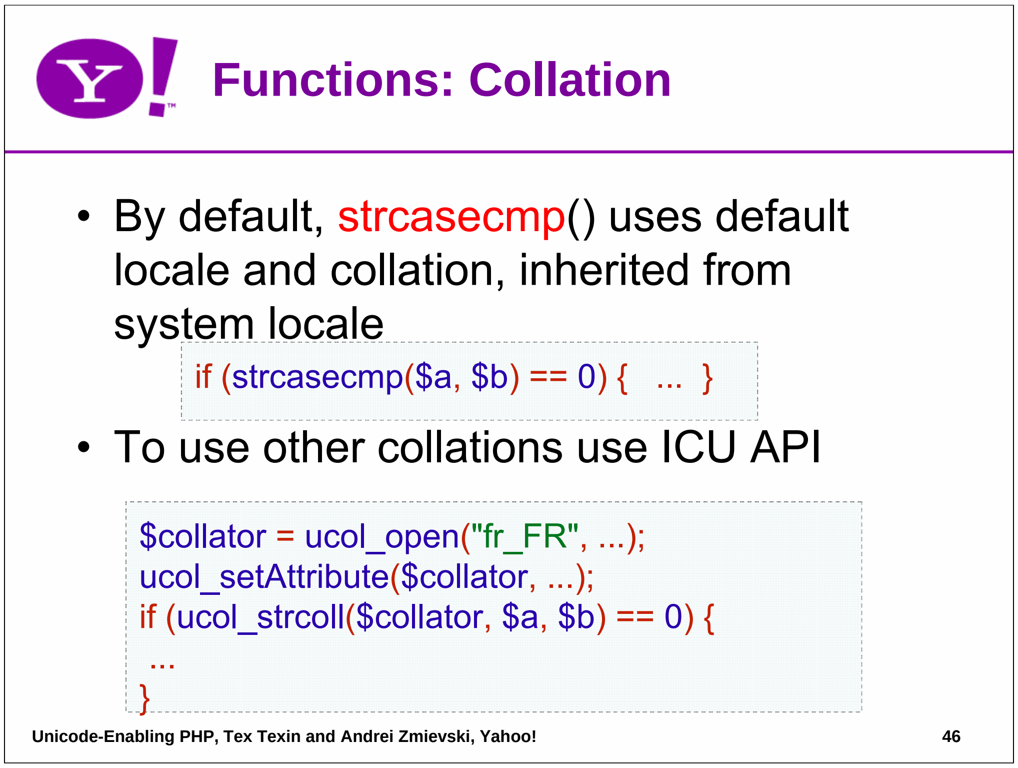
For backwards compatibility, functions should not change drastically, even to achieve more correct Unicode support. E.g. **trim()** should not start removing Unicode characters with the General Category property of whitespace.

Collation and string search API are not used when upgrading functions for 2 reasons. 1) We did not want to add parameters to function prototypes. 2) New collation changes results and inhibits uptake. PHP will continue defaulting to binary compares. Users other functions offering ICU collation and search API.

Supporting full case mapping would break legacy applications. E.g. finding a match where the search string is longer than the located substring, e.g. **stristr('eßen', 'essen')**.

String search matches strings followed by combining characters. This may be revised in the future, but since Normalization form NFC is expected, it should not be a frequent problem.

Formatting functions, e.g. **strftime(), printf(), number_format()**, etc should accept Unicode strings as parameters, but should continue to work with POSIX API and locales. Supporting both POSIX and ICU formats would be too complicated and confusing. The recomendation is to switch scripts to use ICU-based API exclusively.



Functions: Collation

- By default, strcasecmp() uses default locale and collation, inherited from system locale

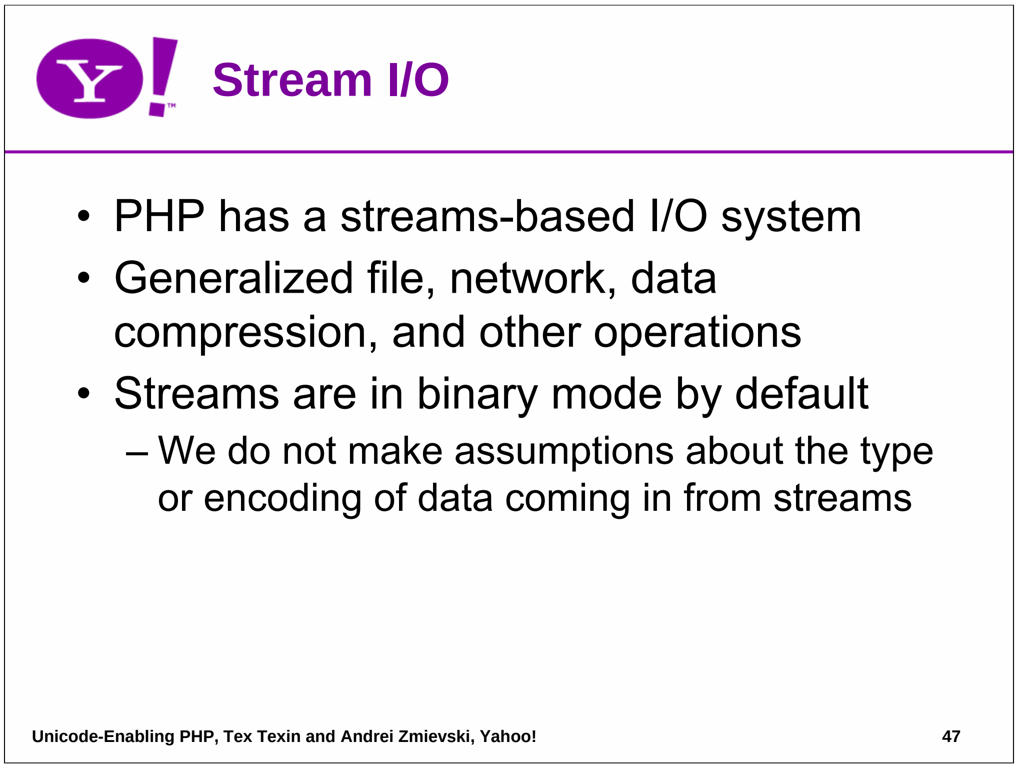
```
if (strcasecmp($a, $b) == 0) {   ...  }
```

- To use other collations use ICU API

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

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 46

Default locale is inherited from the system locale. It can be overridden by the application if necessary, but to process multi-locale data it's best to manage locales dynamically.



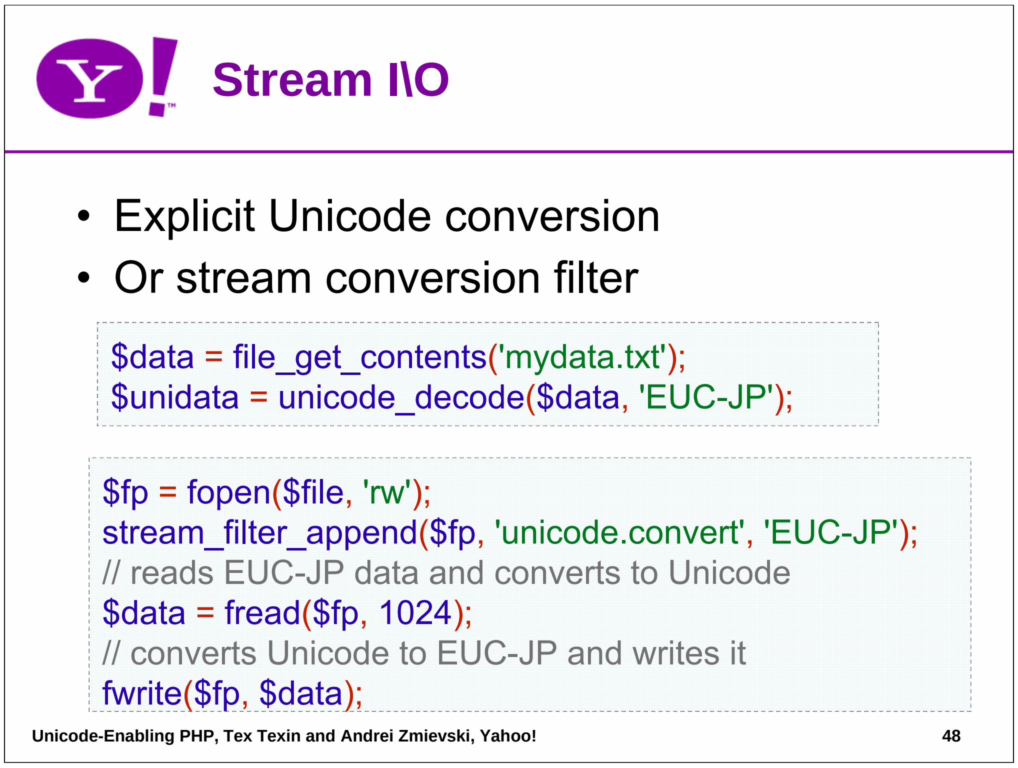
Stream I/O

- PHP has a streams-based I/O system
- Generalized file, network, data compression, and other operations
- Streams are in binary mode by default
  - We do not make assumptions about the type or encoding of data coming in from streams

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 47

In the simplest definition, a stream is a resource object that exhibits streamable behavior, i.e. it can be read from or written to in a linear fashion. Streams can accept one or more filters that perform operations on data as it is being transmitted within the stream. Most of PHP's I/O behavior is implemented on top of the streams.

When **unicode_semantics** switch is on, we cannot make any assumptions about the type or encoding of the data coming in from the streams. Consequently, functions reading from a stream will return strings of binary type, and functions writing to a stream will simply pass through the data as is.



Stream I\O

- Explicit Unicode conversion
- Or stream conversion filter

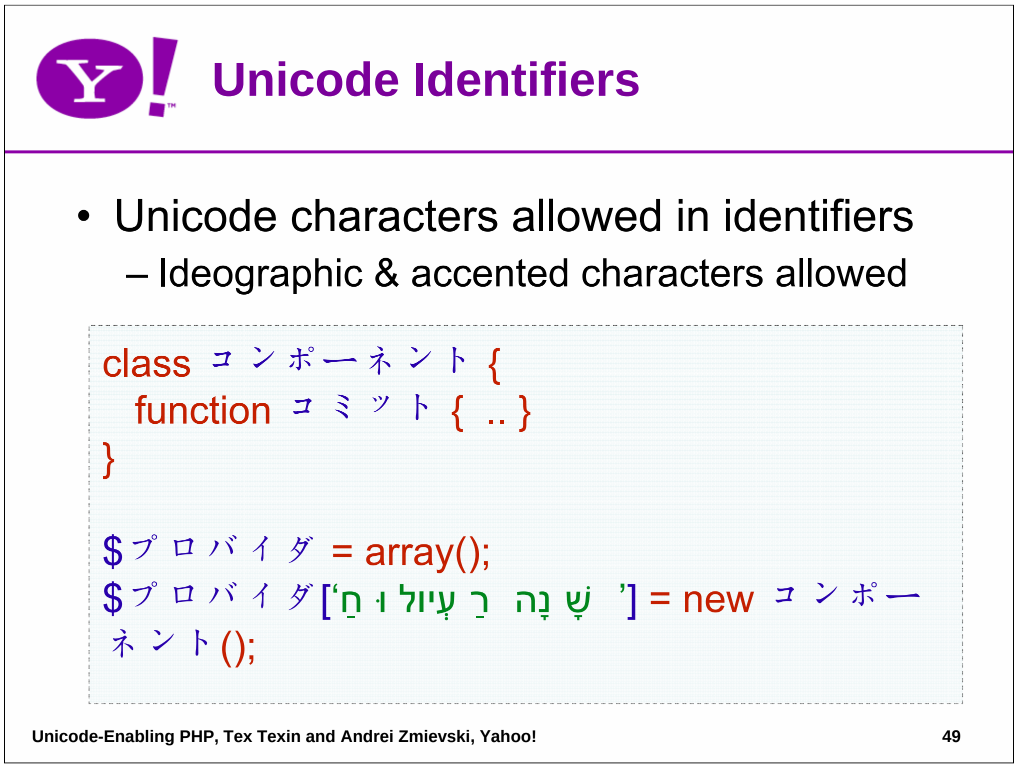
```
$data = file_get_contents('mydata.txt');
$unidata = unicode_decode($data, 'EUC-JP');
```

```
$fp = fopen($file, 'rw');
stream_filter_append($fp, 'unicode.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-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 48

Applications have a couple of ways to convert the binary data into the expected encoding. One would be to manually decode or encode the data using the provided API. Another one would rely on the filters mechanism, where a filter for a specific encoding is applied to the stream and manages the conversion to and from Unicode.

We can do a few other tricks, such as having a default filter that gets applied to every stream, or checking the stream mode in **fopen()** and returning binary or text data in the default encoding. The exact degree of automatization remains to be determined.



Unicode Identifiers

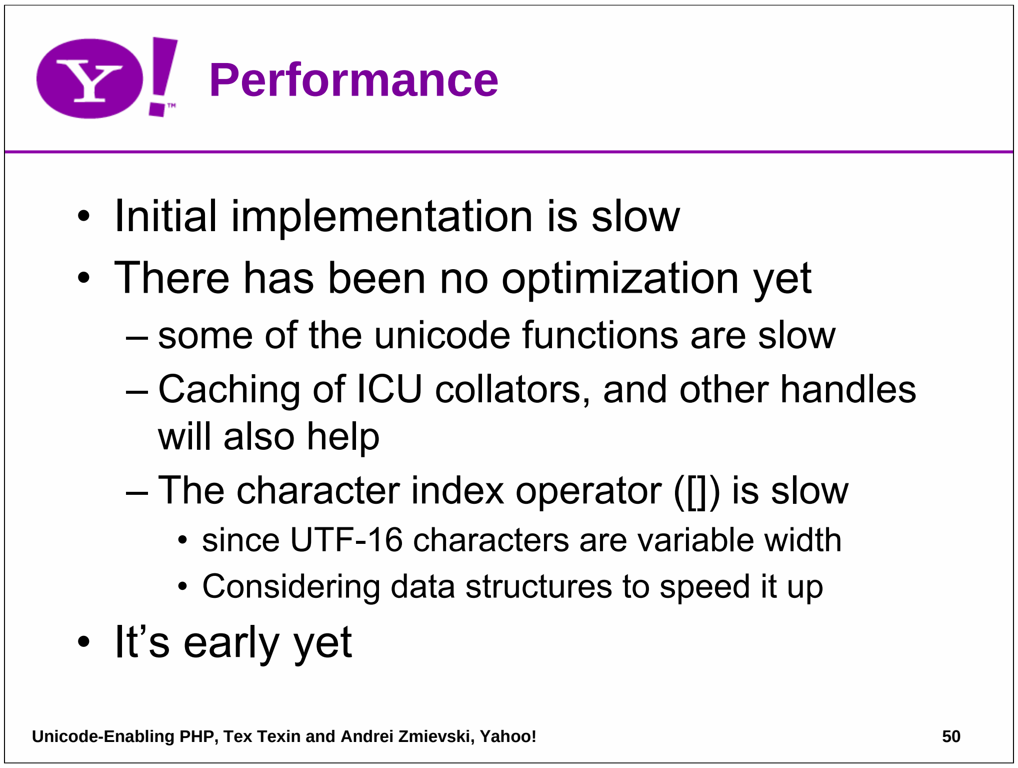
- Unicode characters allowed in identifiers
  - Ideographic & accented characters allowed

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

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

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 49

In previous versions PHP has accepted accented characters in identifiers. We decided to extend the definition of an identifier to include any symbols determined as valid according to Unicode Technical Report 31, Identifier and Pattern Syntax.

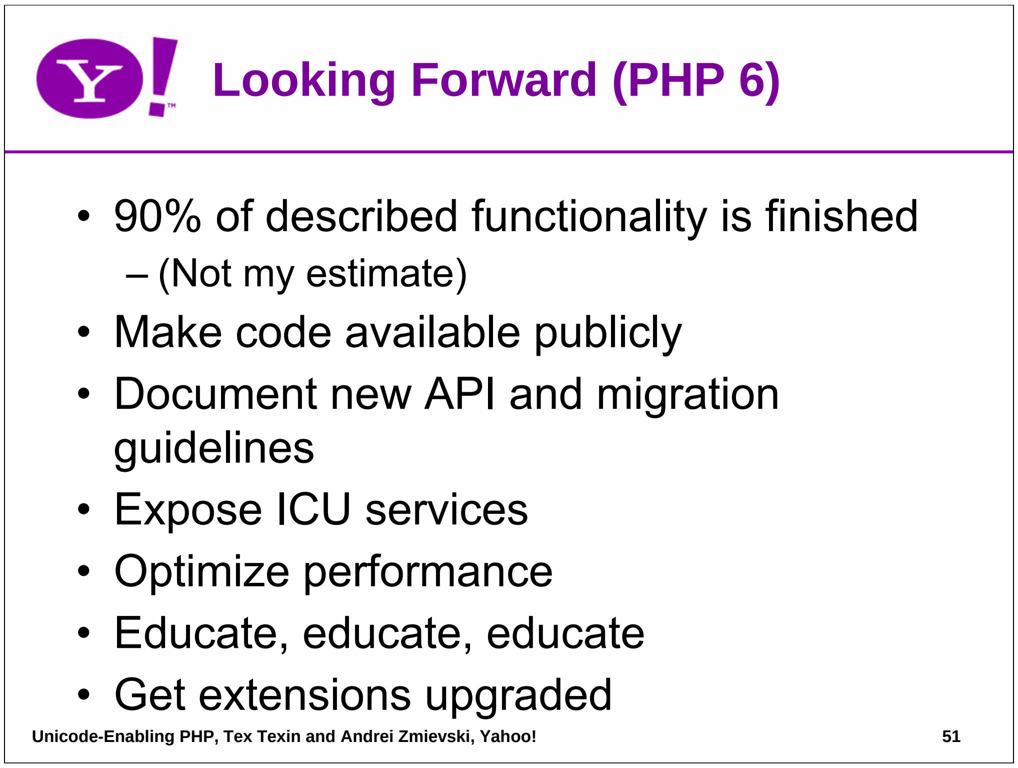
This example presupposes that **script_encoding=utf-8**, since we are using both Hebrew and Japanese character sets.



Performance

- Initial implementation is slow
- There has been no optimization yet
  - some of the unicode functions are slow
  - Caching of ICU collators, and other handles will also help
  - The character index operator ([]) is slow
    - since UTF-16 characters are variable width
    - Considering data structures to speed it up
- It's early yet

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 50



Looking Forward (PHP 6)

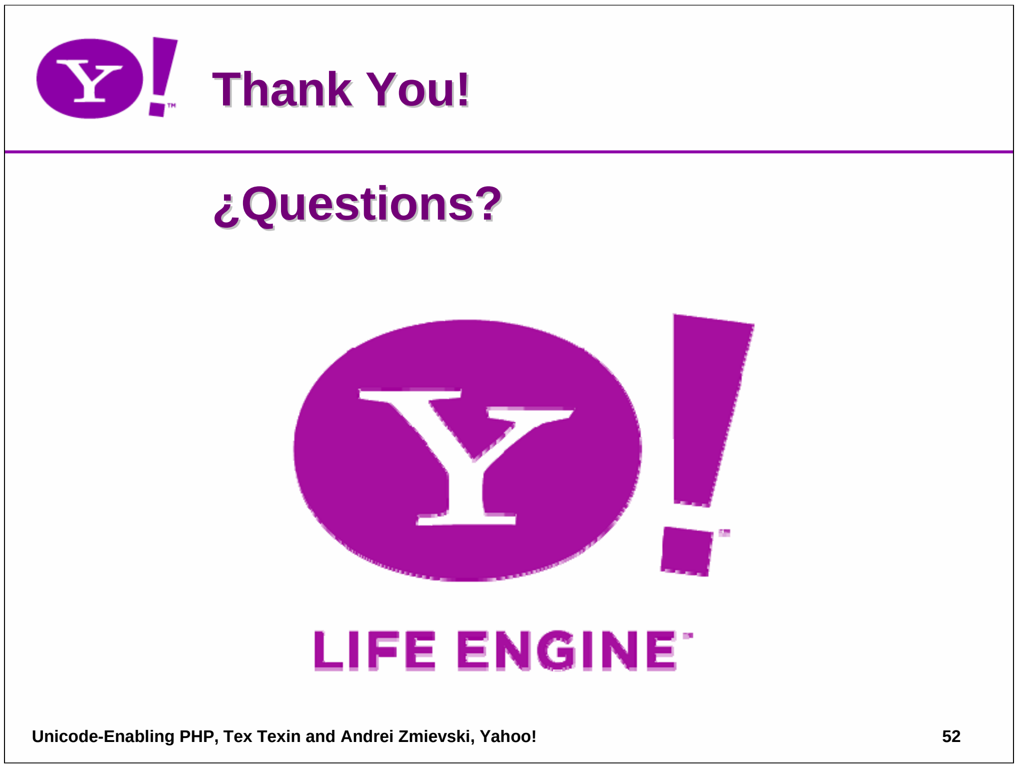
- 90% of described functionality is finished
  - (Not my estimate)
- Make code available publicly
- Document new API and migration guidelines
- Expose ICU services
- Optimize performance
- Educate, educate, educate
- Get extensions upgraded

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 51

The project has been progressing, and approximately 90% of the functionality has been implemented. The process of upgrading existing extensions will probably take from three to six months. Good API documentation and migration guidelines are crucial in making it go faster.

We also need to develop an extension that exposes ICU services, such as collation, boundary analysis, transformations, and so on.Once the desired functionality is achieved, we can concentrate on optimizing performance and hopefully bringing it up to existing level.

We have an ongoing need to educate both PHP developers and PHP users to understand Unicode character model, encodings, algorithms, internationalization and how the new version of PHP supports migrating and developing applications.



Thank You!

¿Questions?

LIFE ENGINE™

Unicode-Enabling PHP, Tex Texin and Andrei Zmievski, Yahoo! 52