String Functions

Disclaimer: вы смотрите просто запись лекции,
это **НЕ** специально подготовленный видеокурс!

Intro

In PHP we deal with strings on regular basis. So, a lot of operations with strings are already automated with a lot of inbuilt functions.

Full list: https://www.php.net/manual/en/ref.strings.php

Warning! When dealing with strings, mind the encoding! If you confuse single-byte encoding with multi-byte encoding, you'll get wrong result for sure!

Just a quick recap: important facts about strings in PHP

No native UTF support.

At least for now.

There’s difference between “ ” and ‘ ’ strings.

Refer to “General Syntax and Data Types” chapter.

No particular restrictions on a string length.

Since PHP 7.0.0 x64.

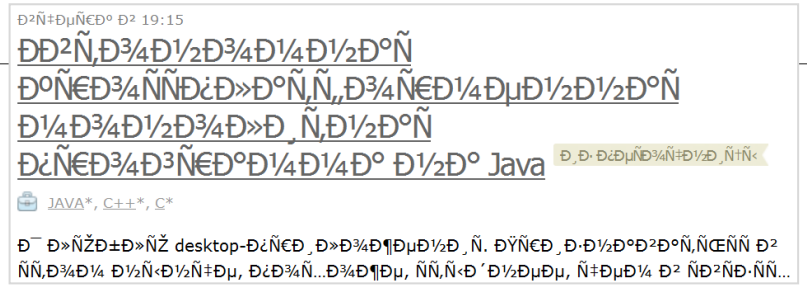
A string may be analyzed (and modified) as an array of bytes.

A lot of binary data is represented as strings in PHP.

E.g., a file contents, etc.

Mind the encoding!!!

Once again: if you misinterpret the encoding of a string, you'll get a wrong result.



So, encodings has to be the same in:

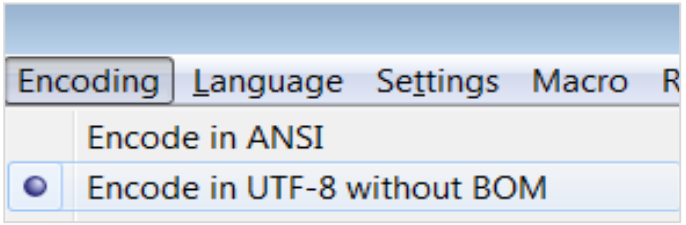
- Source files.
- Templates.
- Configuration files.
- HTML pages and HTML metadata.
- In DBMS: connection, DB (tables, fields, etc.)
- **EVERYWHERE!**

Mind the encoding!!!

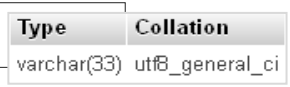
And again. This is really important as it may affect any application in rather unexpected way. And it is one of the most beginner mistake… ☹

So, encodings has to be the same in:

- Source files.
- Templates.
- Configuration files.
- HTML pages and HTML metadata.
- In DBMS: connection, DB (tables, fields, etc.)
- **EVERYWHERE!**




```
<meta charset='utf-8'>
<meta http-equiv='Content-Type' content='text/html; charset=utf-8' />
$db->query("SET CHARACTER SET UTF8");
$db->query("SET CHARSET UTF8");
$db->query("SET NAMES UTF8");
```



Mind the encoding!!!

Finally! ☺

If you need to process a multi-byte string, try searching a function similar to the one you use for single-byte strings, but with “mb_” at the beginning of the function name, i.e.:

strlen → mb_strlen
strpos → mb_strpos
strtolower → mb_strtolower

Mind the encoding!!!

And again. The most obvious case of problems with encoding interpretation is the attempt to read/modify a byte in a multi-byte string:

```php
<?php

$singleByteEncoding = 'Test';
$multiByteEncoding = 'Тест';

echo $singleByteEncoding[2]; // s
echo $multiByteEncoding[2]; // ?

$singleByteEncoding[2] = 'z';
$multiByteEncoding[2] = 'z';

echo $singleByteEncoding; // Tezt
echo $multiByteEncoding; // Tz?ст
```

Quoting and unquoting strings

In many cases a string should be pre-processed in order to quote/unquote some symbols or replace them with another symbols:

```php
<?php

// All HTML entities processing:
$someString = "<img src='1.png'>";
echo htmlentities($someString) . "\n"; // &lt;img src=&#039;1.png&#039;&gt;
$someString = "&lt;img src=&#039;1.png&#039;&gt;";
echo html_entity_decode($someString) . "\n"; // <img src='1.png'>

// Some HTML entities processing (& " ' < >):
$someString = "<img src='1.png'>";
echo htmlspecialchars($someString) . "\n"; // &lt;img src=&#039;1.png&#039;&gt;
$someString = "&lt;img src=&#039;1.png&#039;&gt;";
echo htmlspecialchars_decode($someString) . "\n"; // <img src='1.png'>

// Regexp string quoting:
$someRegexpInputString = ". \ + * ? [ ^ ] $ ( ) { } = ! < > | : -";
echo preg_quote($someRegexpInputString) . "\n";
// \. \\ \+ \* \? \[ \^ \] \$ \( \) \{ \} \= \! \< \> \| \: \-
```

Quoting and unquoting strings

Why is it so important?! Because without such pre-processing you may get catastrophic application failure. The simples visual example is:

Without string pre-processing:


```
value="Hotel "Minsk""
```

With string pre-processing:


```
value="Hotel &quot;Minsk&quot;"
```

Quoting and unquoting strings at DBMS interaction

The next (even more dangerous) situation becomes actual when interacting with a DBMS via SQL queries with some user-defined data:

```php
<?php

$mysqli = new mysqli("127.0.0.1", "user", "password", "db");
$city = "'s-Hertogenbosch";

// This query with escaped $city will work
$query = sprintf("SELECT CountryCode FROM City WHERE name='%s'", $mysqli->real_escape_string($city));
$result = $mysqli->query($query);

// This query will fail, because we didn't escape $city
$query = sprintf("SELECT CountryCode FROM City WHERE name='%s'", $city);
$result = $mysqli->query($query);
```

```php
<?php

$pdo = new PDO('sqlite:/home/user/db.sql3');
$city = "'s-Hertogenbosch";

// This query with escaped $city will work
$query = sprintf("SELECT CountryCode FROM City WHERE name='%s'", $pdo->quote($city));
$result = $pdo->query($query);

// This query will fail, because we didn't escape $city
$query = sprintf("SELECT CountryCode FROM City WHERE name='%s'", $city);
$result = $pdo->query($query);
```

Converting symbols to byte values and vice versa

Some algorithms (e.g. in cryptography) require byte representation of symbols (followed by backward byte-to-symbol conversion).

If you are not familiar with ASCII-table, see here:
http://en.wikipedia.org/wiki/ASCII

```php
<?php

echo ord('A') . "\n"; // 65
echo chr(65) . "\n";  // A
```

Exploding and imploding strings

There are many tasks that are solved easily with strings explosion and implosion. PHP provides a lot of functions for that:

```php
<?php

$somePathOne = 'c:/dir1/dir2/file.ext';
$partsArray = explode('/', $somePathOne);
print_r($partsArray);
// Array ( [0] => c: [1] => dir1 [2] => dir2 [3] => file.ext )

echo $somePathTwo = implode('\\\\', $partsArray) . "\n";
// c:\\dir1\\dir2\\file.ext

$someCsvLine = 'coumn 1,column 2,column 3';
$columnsArray = str_getcsv($someCsvLine);
print_r($columnsArray);
// Array ( [0] => coumn 1 [1] => column 2 [2] => column 3 )
```

```php
<?php

$someLineWithGetParameters = 'a=99&b=55&c=1';
parse_str($someLineWithGetParameters, $parsedGetParameters);
print_r($parsedGetParameters);
// Array ( [a] => 99 [b] => 55 [c] => 1 )

$someLongString = 'This is a long string';
echo wordwrap($someLongString, 5, '<br>', false) . "\n";
// This<br>is a<br>long<br>string
echo wordwrap($someLongString, 5, '<br>', true) . "\n";
// This<br>is a<br>long<br>strin<br>g
```

Converting string case

If you need a string case to be converted to a give one, there are convenient pre-defined functions:

```php
<?php

echo strtolower('just a TEST string') . "\n"; // just a test string
echo strtoupper('just a TEST string') . "\n"; // JUST A TEST STRING

echo lcfirst('JUST a TEST string') . "\n"; // jUST a TEST string
echo ucfirst('just a TEST string') . "\n"; // Just a TEST string

echo ucwords('just a TEST string') . "\n"; // Just A TEST String
```

String length detection, words or characters counting

Obviously, there are a lot of functions to detect a string length or to count words or characters in a string:

```php
<?php

echo strlen('Test') . "\n"; // 4
echo mb_strlen('Test', 'UTF8') . "\n"; // 4
echo strlen('Тест') . "\n"; // 8
echo mb_strlen('Тест', 'UTF8') . "\n"; // 4

$stringCharacters = count_chars('test', 1);
print_r($stringCharacters); // Array ( [101] => 1 [115] => 1 [116] => 2 )

echo str_word_count('Just a test'); // 3
```

Trimming strings

There are a lot of cases when leading or trailing spaces may affect application behavior on a bad way, so, it’s a good idea to trim them:

```php
<?php

echo '[' . trim('   Just a test     ') . ']' . "\n";   // [Just a test]
echo '[' . ltrim('   Just a test     ') . ']'  . "\n"; // [Just a test     ]
echo '[' . rtrim('   Just a test     ') . ']' . "\n";  // [   Just a test]

// You may even define your own set of characters to be trimmed
echo '[' . trim('!!!   Just a test  !!!  ', "! \n\r\t\v\x00") . ']' . "\n";  // [Just a test]
```

Formatting strings

Obviously, there are a lot of situations when a string should be formatted according to some rule, and there are a lot of functions to do that:

```php
<?php

echo number_format(123456.78, 3) . "\n"; // 123,456.780
echo number_format(123456.78, 3, '.', '\'') . "\n"; // 123'456.780

echo str_pad('Test', 10, '*', STR_PAD_RIGHT) . "\n"; // Test******
echo str_pad('Test', 10, '*', STR_PAD_LEFT) . "\n";  // ******Test
echo str_pad('Test', 10, '*', STR_PAD_BOTH) . "\n";  // ***Test***

echo str_repeat('+-', 5) . "\n"; // +-+-+-+-+-

echo str_shuffle('ABCDEF') . "\n"; // ABEFCD

echo strrev('ABCDEF') . "\n"; // FEDCBA
```

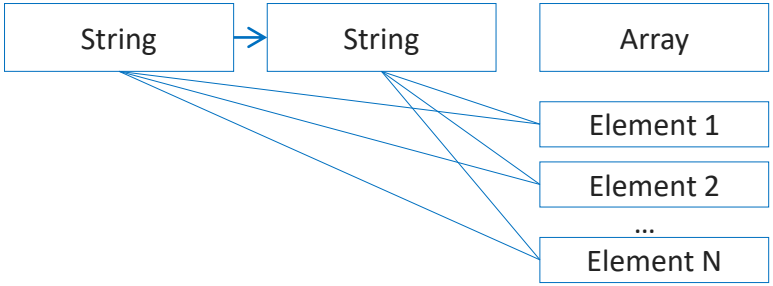
Searching and replacing substrings

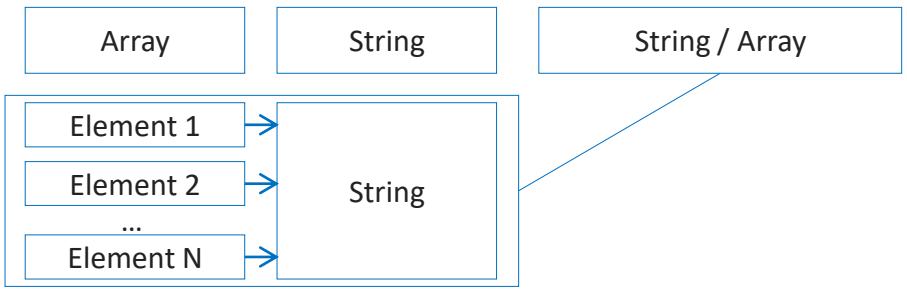
One of the most common operation with strings is the search/replace of a substring withing a string, and there are lot of functions to do that:

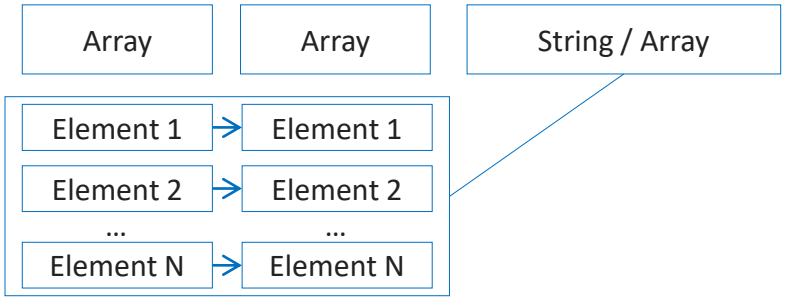
```php
<?php

echo str_replace('BC', '****', 'ABCDEF') . "\n"; // A****DEF
echo str_ireplace('bc', '****', 'ABCDEF') . "\n"; // A****DEF

echo strpos('ABCDEF', 'BC') . "\n";  // 1
echo stripos('ABCDEF', 'bc') . "\n"; // 1

echo strrpos('ABCDEF', 'E') . "\n"; // 4
echo strripos('ABCDEF', 'e') . "\n"; // 4

// Since PHP 8 instead of
if (strpos('string with lots of words', 'words') !== false) { /* ... */ }
// you may use
if (str_contains('string with lots of words', 'words')) { /* ... */ }

// And two more useful functions are available since PHP 8
if (str_starts_with('haystack', 'hay')) { /* true */ }
if (str_ends_with('haystack', 'stack')) { /* true */ }

echo substr_count('this is just a test', 'is') . "\n"; // 2

echo substr('this is just a test', 0, 4) . "\n"; // this
```

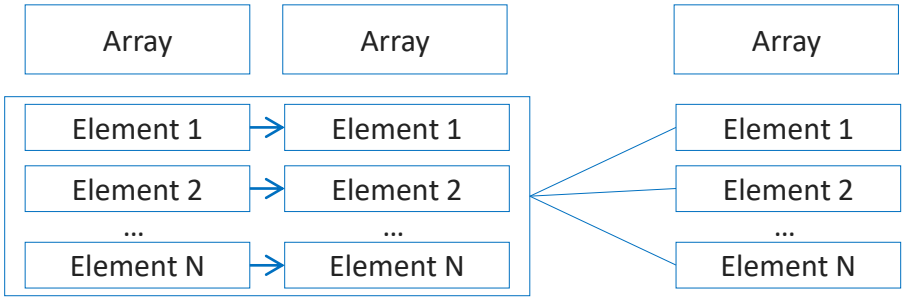
Searching and replacing substrings: power of str_replace (str_ireplace)

These two functions support a lot of parameter variations, and that allows quick and easy processing of huge amounts of data:









Hashing strings

A hash function is a function that can be used to map data of some variable size to fixed-size values. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes.

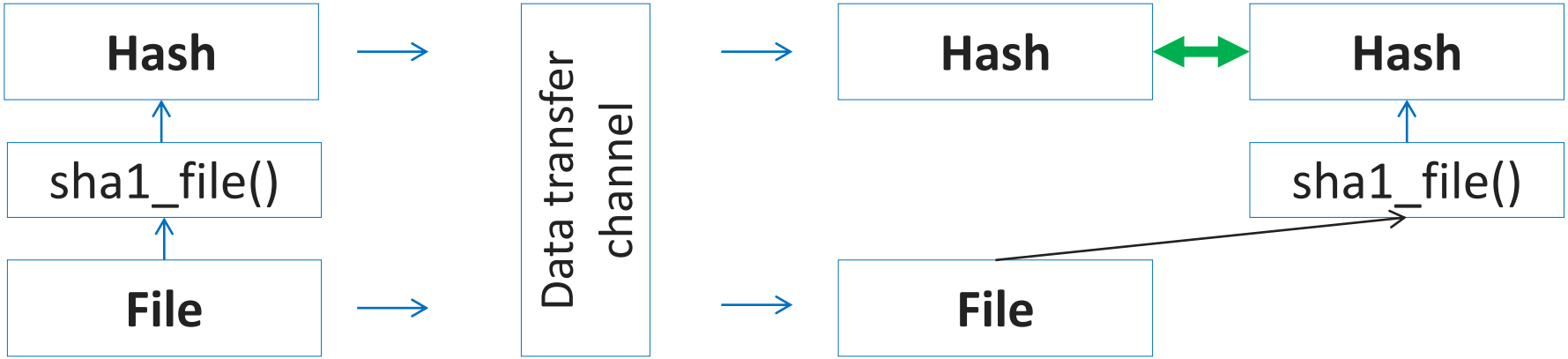
Hashes are used for:

- checksums;
- passwords protection;
- non-secure values to secure values transformation;
- etc.

Hashing strings: checksums

Checksums are widely used to control data consistency while storing or transferring files and their parts (e.g., with torrents ☺):

```php
<?php

echo sha1_file($argv[0]);
// 7a44895d0653e048d592d9b00df58215de4046e1
```



Hashing strings: passwords protection

Passwords should NEVER be stored in the original text representations. Always use hash-functions to increase the security:

```php
<?php
echo hash('sha512', 'SomePassword', false);
// 255b593ddf734ddcb33515f01222ffb944e40d3c6f771696f61e14d38bf7d11952ac253f
a23e91588f70e62e8224dbd934cb9208e9d34892ab7e2bcfce2fd261
```

Hashing strings: non-secure values to secure values transformation

In many cases some user-defined values may be dangerous, but that danger can be easily eliminated with hashes:

```php
<?php

$userFilename = "/../../etc/passwd";
echo $safeFilename = sha1($userFilename . microtime(true) . mt_rand(1000000, 9999999));
// 139596efa322fdaeea970ad153b4d2a0a5fbb455
```

With this approach we may easily avoid attacks on server file system, because we use hashes as file names:

id	user_filename	stored_filename
987	/../../etc/passwd	139596efa322fdaeea970ad153b4d2a0a5fbb455

String Functions

Disclaimer: вы смотрите просто запись лекции,
это **НЕ** специально подготовленный видеокурс!