Синтаксис PHP 5.3, 5.4, 5.5, 5.6, 7 — что нового?

В этой статье я рассмотрю интересные моменты в развитии синтаксиса PHP - получится своего рода список изменений PHP. Что появилось в версиях PHP 5.3 и выше. Основная цель этой статьи создать некую карту изменений синтаксиса, что можно было быстро освежить в памяти «фишки» синтаксиса PHP, которые упрощают коддинг. У меня в последнее время частенько возникает необходимость убедиться, что можно в PHP 5.3, а чего нельзя.

При написании плагинов или тем WordPress можно использовать только возможности PHP 5.3, выше пожалуй не стоит... Все что можно в PHP 5.4 и выше, нужно решать через создание вспомогательных переменных. Впрочем, даже 5.3 не всегда работает, есть еще серверы с PHP 5.2, но хорошо что это редкость. К слову, сам WordPress еще поддерживает PHP 5.2.

Итак, к новинкам...

*Оглавление ▴*

* [Заметки до PHP 5.3](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#zametki-do-php-5.3)
* [*$str1{0}* — синтаксис получения символа строки](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#$str1-0-sintaksis-polucheniya-simvola-stroki)
* [PHP 5.3](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#php-5.3)
* [*?:* — сокращение тернарного оператора](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#sokrashhenie-ternarnogo-operatora)
* [*$func = function(){};* — анонимные (лямбда) функции](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#$func-=-function-anonimnye-lyambda-funktsii)
* [*method()->var* — получение объекта из метода/функции](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#method---var-poluchenie-obekta-iz-metoda-funktsii)
* [*<<<'DOC'* — поддержка NOWDOC](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#lt-lt-lt-'doc'-podderzhka-nowdoc)
* [*namespace* — поддержка пространств имен](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#namespace-podderzhka-prostranstv-imen)
* [*$class::$foo* — динамичное указание класса](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#$class-$foo-dinamichnoe-ukazanie-klassa)
* [*const* — ключевое слово для создания констант вне классов](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#const-klyuchevoe-slovo-dlya-sozdaniya-konstant-vne-klassov)
* [*static::method()* — статическое связывание](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#static-method-staticheskoe-svyazyvanie)
* [*goto hell;* — оператор goto](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#goto-hell-operator-goto)
* [*\_\_callStatic()*, *\_\_invoke()* — магические методы](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#__callstatic-__invoke-magicheskie-metody)
* [PHP 5.4](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#php-5.4)
* [*<?=* — короткая запись вывода на экран работает всегда](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#lt-=-korotkaya-zapis-vyvoda-na-ekran-rabotaet-vsegda)
* [*[1,2]* — запись массива, без слова array](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#1-2-zapis-massiva-bez-slova-array)
* [*trait Class {}* — примеси (трейты)](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#trait-class-primesi-trejty)
* [*foo()[0]* — быстрое получение элемента массива](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#foo-0-bystroe-poluchenie-elementa-massiva)
* [*(new Foo)->method()* — доступ к элементу объекта при его создании](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#new-foo---method-dostup-k-elementu-obekta-pri-ego-sozdanii)
* [*Class::{'foo'}()* — динамичное указание метода](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#class-'foo'-dinamichnoe-ukazanie-metoda)
* [*callable* — новый тип для аргументов функции/метода](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#callable-novyj-tip-dlya-argumentov-funktsii-metoda)
* [*@* — улучшена производительность](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#uluchshena-proizvoditelnost)
* [PHP 5.5](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#php-5.5)
* [*[1,3,4][2]*, *"foobar"{2}* — разыменования только-созданных массивов и строк](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#1-3-4-2-foobar-2-razymenovaniya-tolko-sozdannyh-massivov-i-strok)
* [*empty()* — можно применять к результатам функций и выражений](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#empty-mozhno-primenyat-k-rezultatam-funktsij-i-vyrazhenij)
* [*list() в foreach*](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#list-v-foreach)
* [*finally* — в конструкции try/catch](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#finally-v-konstruktsii-try-catch)
* [*Class::class* — для получение имени класса в пространствах](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#class-class-dlya-poluchenie-imeni-klassa-v-prostranstvah)
* [*yield* — создание генераторов](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#yield-sozdanie-generatorov)
* [API для хэширования паролей](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#api-dlya-heshirovaniya-parolej)
* [PHP 5.6](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#php-5.6)
* [*const PLUS = 1 + 2;* — скалярные выражения в константах/свойствах/аргументах функции](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#const-plus-=-1-+-2-skalyarnye-vyrazheniya-v-konstantah-svojstvah-argumentah-funktsii)
* [*const ARR = ['a', 'b'];* — константа может хранить массив](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#const-arr-=-'a'-'b'-konstanta-mozhet-hranit-massiv)
* [*func( ...$args )* или *func( ...[2, 3] )* — неизвестное число аргументов функции или распаковка массива с помощью '...' (splat оператор)](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#func-...$args-ili-func-...-2-3-neizvestnoe-chislo-argumentov-funktsii-ili-raspakovka-massiva-s-pomoshhyu-'...'-splat-operator)
* [*\*\** — оператор возведения в степень](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#**-operator-vozvedeniya-v-stepen)
* [*use function* и *use const* — импорт функций и констант в пространство](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#use-function-i-use-const-import-funktsij-i-konstant-v-prostranstvo)
* [Куда делся PHP 6?](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kuda-delsya-php-6)
* [PHP 7](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#php-7)
* [*$a ?? ''* — одновременная проверка isset и получение значения](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#$a-''-odnovremennaya-proverka-isset-i-poluchenie-znacheniya)
* [*$a <=> $b* — одновременное выполнение трех сравнений: больше, равно или меньше.](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#$a-lt-=-$b-odnovremennoe-vypolnenie-treh-sravnenij-bolshe-ravno-ili-menshe.)
* [*define('FOO', [1,2]);* — передача массива константе через define()](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#define-'foo'-1-2-peredacha-massiva-konstante-cherez-define)
* [*use name\space\{A, B, C as c};* — группировка импорта при помощи use](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#use-name-space-a-b-c-as-c-gruppirovka-importa-pri-pomoshhi-use)
* [*int, float, bool* — новые типы для аргументов функции/метода](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#int-float-bool-novye-tipy-dlya-argumentov-funktsii-metoda)
* [*int, float, bool, array* — указание возвращаемых типов для функции/метода](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#int-float-bool-array-ukazanie-vozvrashhaemyh-tipov-dlya-funktsii-metoda)
* [*foo()(), $a::$b::$c, $$foo->bar* — единый синтаксис переменных: СЛЕВА НАПРАВО](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#foo-$a-$b-$c-$$foo--bar-edinyj-sintaksis-peremennyh-sleva-napravo)
* [foreach — изменена логика работы](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#foreach-izmenena-logika-raboty)
* [*$class = new class{}* — анонимные классы](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#$class-=-new-class-anonimnye-klassy)
* [*yield ... return 99;* — возврат (return) выражений в генераторах](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#yield-...-return-99-vozvrat-return-vyrazhenij-v-generatorah)
* [*yield from gen()* — делегирование генераторов](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#yield-from-gen-delegirovanie-generatorov)
* [Еще новинки PHP 7.0](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#eshhe-novinki-php-7.0)
* [PHP 7.1](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#php-7.1)
* [*void* — возвращаемый тип](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#void-vozvrashhaemyj-tip)
* [*iterable* — новый псевдо-тип](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#iterable-novyj-psevdo-tip)
* [*null* — тип передаваемых/возвращаемых значений](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#null-tip-peredavaemyh-vozvrashhaemyh-znachenij)
* [*{-1}* — отрицательное значение смещения в строках](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#1-otritsatelnoe-znachenie-smeshheniya-v-strokah)
* [*['key'=>$a] = ['key'=>'Значение']* — поддержка ключей и новый синтаксис list()](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#'key'=-$a-=-'key'=-'znachenie'-podderzhka-klyuchej-i-novyj-sintaksis-list)
* [Область видимости констант в классах](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#oblast-vidimosti-konstant-v-klassah)
* [Заметки по PHP 7.1](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#zametki-po-php-7.1)

Заметки до PHP 5.3

*$str1{0}* — синтаксис получения символа строки

Новый синтаксис обращения к символам строки:

// получение символа

$str = 'abc';

**echo** $str{0}; //> a

**echo** $str{1}; //> b

**echo** 'abc'{1}; //> b

// установка символа

$str1 = $str2 = 'ff';

$str1{0} = 'a';

**echo** $str1; //> af

$str2[0] = 'a'; // этот синтаксис не рекомендуется

**echo** $str2; //> af

$str2[0] не рекомендуется, хотя и работает точно также. Рекомендация использовать фигурные скобки *{}* связана с тем, чтобы при прочтении кода было сразу понятно, что обрабатывается строка, а не элемент массива. Как мы знаем квадратными скобками в PHP принято обозначать массивы, а не строки.

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

PHP 5.3

В PHP 5.3, как и во всей пятой ветке PHP, включена новая машина-интерпретатор скриптов [Zend Engine 2.0](https://ru.wikipedia.org/wiki/Zend_Engine). Благодаря этому PHP стал работать быстрее примерно на 15-20%.

Новые возможности в PHP 5.3 ([ссылка на офф. сайт](http://php.net/manual/ru/migration53.new-features.php)):

*?:* — сокращение тернарного оператора

С PHP 5.3 стало возможным не писать среднюю часть тернарного оператора. Выражение *expr1 ?: expr3* возвращает expr1 если expr1 не пустой, и expr3 в противном случае.

Тернарный — состоящий из трёх частей, компонентов.

$a = $expr1 ?: $expr3;

// равносильно записи:

$a = $expr1 ? $expr1 : $expr3;

Пример тернарного оператора:

// полная запись

**if** ( $a > 100 )

 $result = "Больше";

**else**

 $result = "Меньше";

// краткая запись

$result = $a > 100 ? "Больше" : "Меньше";

В короткой записи есть еще момент производительности, например:

// полная запись

**if** ( get\_post\_meta(25, 'meta\_key', 1) )

 **echo** esc\_html( get\_post\_meta(25, 'meta\_key', 1) );

**else**

 **echo** 'Мета поля нет';

// краткая запись

**echo** esc\_html( get\_post\_meta(25, 'meta\_key', 1) ?: 'Мета поля нет' );

В полной записи функция get\_post\_meta() вызывается 2 раза. В короткой один раз, и если она что-то вернула, второму аргументу сразу передается полученное значение: не нужны дополнительные переменные...

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*$func = function(){};* — анонимные (лямбда) функции

Лямбда-функции еще называют «анонимными функциями», потому что для них не указывается название.

Лямбда-функции представляют собой замыкание (англ. closure) — это особый вид функции, которая определена в теле другой функции и создаётся каждый раз во время её выполнения. Синтаксически это выглядит как функция, находящаяся целиком в теле другой функции. Насколько я понял, любая функция — это замыкание текущего контекста, т.е. контекст не будет очищен пока работает функция. Но если в функции есть лямбда-функция, то она становится замыканием, и если в неё передаются переменные из «верхней» функции, то они не будут очищены, до тех пор пока работает вложенная-функция, даже если «верхняя» функция работу закончила...

В ранних версиях, анонимные функции создавались с помощью функции create\_function().

Пример создания анонимной функции для сортировки usort():

$arr = **array**(3, 2, 5, 6, 1);

usort( $arr, **function**($a, $b) {

 **if** ( $a == $b )

 **return** 0;

 **return** ( $a > $b ) ? -1 : 1;

});

Еще одна фишка лямбда-функций — это использование переменных из текущей области видимости, с помощью оператора *use*:

$var = 'Превед, Медвед!';

$func = **function**() **use** ( $var ) { echo $var; };

$func(); //> Превед, Медвед!

Переменные передаются как значение, но можно передать и ссылку на переменную, указав *&*:

$var = 'Превед, Медвед!';

$func = **function**() **use** ( & $var ) { $var = $var .' Мы в гости!'; };

$func(); // вызовем

**echo** $var; //> Превед, Медвед! Мы в гости!

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*method()->var* — получение объекта из метода/функции

Это удобно:

$object->method()->method()->method();

$object->method()->method()->member = 5;

В PHP ниже 5.3 писали как-то так:

$tmp = & $object->method();

$tmp = & $tmp->method();

$tmp->method();

*<<<'DOC'* — поддержка NOWDOC

В php 5.3 можно использовать аналог [HEREDOC](http://ru2.php.net/manual/en/language.types.string.php#language.types.string.syntax.heredoc), который называется [NOWDOC](http://php.net/nowdoc). Особенность его в том, что внутри него переменные остаются простым текстом, как если бы мы указали её в строке с одинарными кавычками: *'текст $foo'*:

$foo = 'Лето';

// HEREDOC был в 5.2

$str = <<<DOC

 Текст с переменной '**$foo**'

DOC;

**echo** $str; // Текст с переменной 'Лето'

// NOWDOC появился в 5.3

$str = <<<'DOC'

 Текст с переменной '. **$foo** .'

DOC;

**echo** $str; // Текст с переменной '. $foo .'

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*namespace* — поддержка пространств имен

Пространства имен нужны, чтобы избежать конфликтов при совпадении названий функций/классов/переменных/констант. Если коротко: одинаковые называния в разных пространствах — это разные названия.

Пример ниже должен объяснить почти все, что возможно в пространствах имен. За подробностями [идем в официальную документацию](http://php.net/namespaces).

<?php

#

# Объявлять пространство нужно в самом начале файла содержащего пространство имен,

# т.е. до любого кода, кроме зарезервированного declare(encoding='...');.

# Также ничего не должно выводиться на экран до объявления пространства

# Одно и тоже пространство имен можно определять в разных файлах. Так эти файлы будут относиться к одному пространству

# Объявляем пространство my\name

**namespace** my\name;

## Динамичное получения названия пространства --------------

$s = **\_\_NAMESPACE\_\_**; //> my\name

$s = **\_\_NAMESPACE\_\_** . '\HELLO'; //> my\name\HELLO

// namespace: есть еще специальное слово namespace, которое используется для динамичного

// получения названия текущего пространства при вызове функций/методов/констант (см. ниже)

## ГЛОБАЛЬНЫЕ функции/классы/константы в нашем пространстве --------------

$s = strlen('hi'); // вызовет my\name\strlen() - если функция есть в нашем пространстве, иначе глобальную функцию strlen()

define('HELLO', 'HI всем'); // добавит константу в глобальное пространство "\HELLO"

# Доступ к глобальным классам/функциям/константам из пространства имен

$a = \strlen('hi'); // вызывает глобальную функцию strlen()

$b = \ABSPATH; // получает доступ к глобальной константе ABSPATH

$c = **new** \WP\_Query; // создает экземпляр глобального класса WP\_Query

## ФУНКЦИЯ в нашем пространстве --------------

**function** my\_func(){ **return** 'Моя функция'; }

// Вызов

my\_func(); //> "Моя функция"

**namespace**\my\_func(); //> "Моя функция"

\my\name\my\_func(); //> "Моя функция"

// my\name\my\_func(); //> ошибка: будет вызвана функция my\name\my\name\my\_func()

 // такой синтаксис можно использовать для доступа к под-пространствам нашего пространства

## ФУНКЦИЯ в нашем пространстве, которая существует в глобальном --------------

**function** trim( $str ){

 **return** \trim( $str, '-' ); # если вызвать trim( $str, '-' ), то функция вызовет сама себя...

}

// Вызов

$s = trim('-foo'); // вызов trim() из текущего пространства. Выведет: foo

$s = \my\name\trim('-foo'); // тоже что в строке выше

$s = **namespace**\trim('-foo'); // тоже что в строке выше

$s = \trim('-foo'); // вызов trim() из глобального пространства. Выведет: -foo

## КОНСТАНТЫ в нашем пространстве --------------

**const** HELLO = 'HI'; // добавим константу в текущее пространство

define('my\name\HELLO', 'HI'); // тоже что в строке выше

define(**\_\_NAMESPACE\_\_** . '\HELLO', 'HI'); // тоже что в строке выше

// Вызов

$s = HELLO; //> HI - если константа есть в текущем пространстве, или значение глобальной константы

$s = \my\name\HELLO; //> HI

$s = **namespace**\HELLO; //> HI

$s = \HELLO; //> HI всем - глобальная константа HELLO

## КЛАСС в нашем пространстве --------------

**class** MyClass {

 **function** method(){ **return** 'метод MyClass'; }

 **static** **function** static\_method(){ **return** 'статический метод MyClass'; }

}

// Вызов

$a = **new** MyClass; // обращение к MyClass из текущего пространства

$a = **new** \my\name\MyClass; // тоже что в строке выше

$s = **namespace**\MyClass::static\_method(); //> 'статический метод MyClass' - вызывает статический метод "static\_method" класса my\name\MyClass.

$s = $a::static\_method(); // тоже что в строке выше

$s = $a->method(); //> 'метод MyClass' - вызывает метод "method" класса my\name\MyClass

 // namespace\MyClass->method() - такой вызов метода, вызовет ошибку - syntax error

## ВНЕДРЕНИЕ функций/методов/констант в наше пространстве из других пространств --------------

// ЗАМЕТКА: операторы use можно комбинировать: указывать через запятую

// Например: use other\name\OtherClass as Another, other\name\NSname;

**use** other\name\OtherClass as Another;

$obj = **new** Another; // создает объект класса other\name\OtherClass

**use** other\name; // теперь name = other\name

name\other\_func(); // вызывает функцию other\name\other\_func();

// импорт глобального класса

**use** WP\_Query;

$a = **new** WP\_Query(); // создаст экземпляр класса WP\_Query

 // без выражения "use WP\_Query;" создавался бы экземпляр my\name\WP\_Query

// импорт функции (PHP 5.6+)

**use** function other\name\other\_func;

$s = other\_func(); //> "Другая Функция" - работа функции other\name\other\_func()

// импорт функции под псевдонимом func (PHP 5.6+)

**use** function other\name\other\_func as func;

$s = func(); //> "Другая Функция" - работа функции other\name\other\_func()

//const other\name\HELLO2 = 'И снова здрасте!'; // вызовет ошибку синтаксиса, что странно

define('other\name\HELLO2', 'И снова здрасте!');

// импорт константы (PHP 5.6+)

**use** const other\name\HELLO2;

$s = HELLO2; //> "И снова здрасте!" - содержимое константы other\name\HELLO2

## ЕЩЕ ОДНО ПРОСТРАНСТВО в одном файле --------------

// Подробнее: http://php.net/manual/ru/language.namespaces.definitionmultiple.php

**namespace** other\name;

**class** OtherClass {}

**function** other\_func() { **return** 'Другая Функция'; }

// При описании нескольких пространств в одном файле лучше использовать синтаксис со скобками:

/\*

namespace MyProject {

 function connect() {}

}

namespace AnotherProject {

 function connect() {}

}

\*/

развернуть

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*$class::$foo* — динамичное указание класса

Это дает динамичный доступ к статическим методам/свойствам класса:

**class** C {

 **static** $foo = 'foo';

}

$class = 'C';

**echo** $class::$foo; //> foo

*const* — ключевое слово для создания констант вне классов

Сразу пример, где все понятно:

define('SHORTINIT', 'true');

// теперь можно объявить константу и так:

**const** SHORTINIT = 'true';

В отличие define(), такие константы, должны быть объявлены в самой верхней области видимости, потому что они определяются при компилировании скрипта. Это значит, что их нельзя объявлять внутри функций/циклов/выражений if или try/ catch блоков.

*static::method()* — статическое связывание

Статическое объявление метода/свойства связывает его с классом из которого оно вызывается, а не с тем в котором оно зарегистрировано. Посмотрим на примере:

**class** A {

 **static** **function** who() {

 **echo** **\_\_CLASS\_\_**;

 }

 **static** **function** test1() {

 **self**::who();

 }

 **static** **function** test2() {

 **static**::who(); // статическое связывание

 }

}

**class** B **extends** A {

 **static** **function** who() {

 **echo** **\_\_CLASS\_\_**;

 }

}

**echo** B::test1(); //> A

**echo** B::test2(); //> B

**echo** B::who(); //> B

Подробнее про статическое связывание [читайте в документации](http://php.net/manual/ru/language.oop5.late-static-bindings.php).

*goto hell;* — оператор goto

Используется для перехода в другую часть программы. Место, куда необходимо перейти указывается с помощью метки, за которой ставится двоеточие, после оператора goto указывается желаемая метка для перехода.

Целевая метка должна находиться в том же файле, в том же контексте. Т.е. нельзя выйти за границы функции или метода, а значит нельзя перейти внутрь любой функции.

Также нельзя перейти внутрь любой циклической структуры или оператора switch. Но можно выйти из любой циклической структуры, поэтому «goto» удобен как замена многоуровневых break.

Пример использования goto:

**function** zayac(){

 $i = 1;

 $out = '';

 start: $out .= ($i > 1 ? '-' : '' ) .$i;

 **if**( $i++ < 5 ){ **goto** start; }

 **return** $out . ' вышел зайчик погулять';

}

**echo** zayac(); //> 1-2-3-4-5 вышел зайчик погулять

Пример использования goto в цикле:

**for**( $i=0, $j=50; $i<100; $i++ ) {

 **while**( $j-- ) {

 **if**( $j==17 ) **goto** end;

 }

}

**echo** "i = **$i**"; // будет пропущено

end: **echo** 'j дошло до 17';

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*\_\_callStatic()*, *\_\_invoke()* — магические методы

*\_\_callStatic()* — срабатывает, когда вызывается несуществующий метод из статического контекста: *Foo::bar()*:

**class** A {

 **static** **function** \_\_callStatic( $name, $args ){

 **return** $name .' '. print\_r( $args, 1 );

 }

}

**echo** A::no\_matter\_what('bar');

/\* Выведет:

no\_matter\_what Array

(

 [0] => bar

)

\*/

*\_\_invoke()* — срабатывает, когда объект выполняется как функция: *$obj()*:

**class** A {

 **function** \_\_invoke( $var ){

 var\_dump( $var );

 }

}

$obj = **new** A;

$obj('foo'); //> string(3) "foo"

PHP 5.4

Возможности, добавленные в версии PHP 5.4. [Ссылка на офф. сайт](http://php.net/manual/ru/migration54.new-features.php).

*<?=* — короткая запись вывода на экран работает всегда

Короткая запись о которой идет речь это: *<?=* вместо *<?php echo*.

Для работы такой короткой записи вывода на экран в версиях ниже 5.4 нужно было, чтобы опция *short\_open\_tag* в php.ini была включена.

Пример длинной и короткой записи:

<a href="#"><?php **echo** $page ?></a>

<a href="#"><?= $page ?></a>

*[1,2]* — запись массива, без слова array

$a = [1, 2, 3, 4];

$a = ['one' => 1, 'two' => 2, 'three' => 3, 'four' => 4];.

*trait Class {}* — примеси (трейты)

Трейт - это аналог класса, который содержит в себе методы. Нужен он для «подмешивания» его в имеющийся класс, чтобы методы трейта стали методами класса в который он добавлен.

Несколько примесей можно задавать через запятую:

**trait** TR\_A {

 **public** $var = 'var';

 **function** foo() { **return** 'foo'; }

}

**trait** TR\_B {

 **function** bar() { **return** 'bar'; }

}

**class** A {

 **use** TR\_A, TR\_B; // подмешиваем

 **function** hello() { **return** 'hello A'; }

}

$A = **new** A();

**echo** $A->foo(); // foo

**echo** $A->bar(); // bar

**echo** $A->hello(); // hello A

**echo** $A->var; // var

**class** B **extends** A {

 **use** TR\_A, TR\_B;

 **function** hello() { **return** 'hello B'; }

}

$B = **new** B();

**echo** $B->foo(); // foo

**echo** $B->bar(); // bar

**echo** $B->hello(); // hello B

**Приоритеты трейтов**

При совпадении названий свойств/методов приоритеты расставляются так: текущий класс имеет наивысший приоритет, затем трейт, а затем расширяемый класс. Другими словами: элементы из текущего класса переопределяют элементы в трейте, которые в свою очередь переопределяют унаследованные элементы.

**Статический доступ к методу примеси из класса**

Когда в класс подмешивается trait, то его методы становятся методами класса, включая статические и статический доступ:

**trait** A {

 **static** **function** func(){ **echo** 'A'; }

}

**class** B {

 **use** A;

}

B::func(); //> A

Подробнее про трейты [читайте в документации](http://php.net/manual/ru/language.oop5.traits.php)

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*foo()[0]* — быстрое получение элемента массива

Теперь не нужно сохранять массив, который вернула функция/метод в переменную и получать элемент массива из этой переменной. Можно сразу получать элемент из функции/метода:

$foo = func()[0];

$foo = **Class**::func()[0];

*(new Foo)->method()* — доступ к элементу объекта при его создании

$foo = (**new** Foo)->method();

$foo = (**new** Foo)->property;

$foo = (**new** Foo)[0];

// было так

$obj = **new** Foo;

$foo = $obj->method();

*Class::{'foo'}()* — динамичное указание метода

Чтобы вызвать статический метод/свойство класса, не нужно запоминать его в отдельную переменную:

**class** A {

 **static** **function** foo() {

 **echo** "Hello world!";

 }

 }

 $x = "f";

 A::{ $x .'oo' }();

*callable* — новый тип для аргументов функции/метода

Авто-проверка передаваемых данных в функции/методы, известная как «контроль типа» (typehint), продолжает развиваться и теперь понимает слово *callable*.

Раньше для автоматической проверки типа передаваемого параметра, в аргументах функции/метода можно было указывать только: *array* или *имя класса*.

Теперь, можно указать еще: *callable* — значит, что передаваемый аргумент должен быть вызываемым, т.е. удовлетворяет условию is\_callable( $arg, false ).

Пример:

**function** func( callable $callback ){

 **return** **true**;

}

func('trim'); //> true

func( **function**(){} ); //> true

$db = **new** wpdb();

func( **array**($db, 'query') ); //> true

func('my\_trim'); //> fatal error: Argument 1 passed to func() must be callable, string given

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*@* — улучшена производительность

Оператор *@* нужен для подавления вывода ошибок любого уровня. Вообще его использовать не рекомендуется, но иногда с ним короче:

**if**( @ $\_GET['foo'] ) **echo** 'OK';

// или так

**if**( **isset**($\_GET['foo']) && $\_GET['foo'] ) **echo** 'OK';

// раньше так работало быстрее раз в 20, теперь раз в 5

Использовать *@* нужно как можно реже и очень осторожно, потому что часто заметки и предупреждения дают понять, что логика кода работает неправильно. Например, у меня бывало что лезу поправить казалось бы безобидный NOTICE, но при анализе выясняется что ошибка появилась из-за неправильной логики кода, которая изменилась в процессе расширения кода...

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

PHP 5.5

Новые возможности в PHP 5.5 ([ссылка на офф.сайт](http://php.net/manual/ru/migration55.new-features.php)):

*[1,3,4][2]*, *"foobar"{2}* — разыменования только-созданных массивов и строк

**echo** **array**(1, 2, 3)[0]; // 1

**echo** [1,3,4][2]; // 4

**echo** "foobar"{2}; // o

**echo** "foobar"[0] // f

// это может пригодиться для быстрой генерации:

**echo** 'abcdefghijk'{ rand(0,10) }; // получим одну из букв: 'abcdefghijk'

*empty()* — можно применять к результатам функций и выражений

Раньше empty() мог принимать только переменные, теперь можно передавать сами выражения без необходимости сохранять результат в отдельную переменную:

**empty**( $object->get() );

*list() в foreach*

В foreach стало возможным использовать [list()](http://php.net/manual/ru/function.list.php):

$array = [

 [1, 2],

 [3, 4],

];

**foreach**( $array **as** **list**($a, $b) ){

 **echo** $a;

 **echo** $b;

}

// получим: 1234

*finally* — в конструкции try/catch

Выбрасывать и ловить исключения можно с PHP 5. Такой подход позволяет контролировать выполнение кода, если есть подозрение, что в нем что-то может пойти не так.

А с версии 5.5. в эту конструкцию добавили третий блок *finally*. Блок finally выполняется всегда после завершается конструкции try/catch. Он выполняется даже когда код try вызвал фатальную ошибку:

**try** {

 **echo** 'Тут что-то деламе... ';

 // выбрасываем

 **throw** **new** **Exception**('Лови меня! ');

}

// ловим

**catch**( **Exception** $e ){

 **echo** $e->getMessage(); // выведет: Лови меня!

}

**finally** {

 **echo** 'А это выводиться всегда!';

}

Нужен finally для удобства, и дополнительных возможностей. С ним можно будет писать меньше кода и можно, например, удобно чистить память, когда это нужно.

Пару домонстрационных примеров:

**Меньше кода**

Допустим, нам нужно выполнить функцию close() в любому случае, было выброшено исключение или нет:

**try** {

 my\_function();

}

**catch**( **Exception** $e ){

 // close(); // эта строка нужна была бы без finally

 **echo** $e->getMessage(); // выведет: Лови меня!

}

**finally** {

 close();

}

//close(); // эта строка нужна была бы без finally

**Больше возможностей**

Допустим мы открыли соединение с БД до выполнения кода и есть вероятность что код вызовет ошибку и открытое соединение не закроется, а нам нужно его закрыть в любом случае. finally как раз кстати:

$db = mysqli\_connect();

**try** {

 my\_function( $db ); // результат работы функции может вызвать фатальную ошибку...

}

// исключение можно не обрабатывать

**finally** {

 mysqli\_close($db);

}

*Подробнее про finally читайте*[*статью на хабре*](https://habrahabr.ru/post/149314/)*.*

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*Class::class* — для получение имени класса в пространствах

Появилось ключевое слово *class* для классов, которое выводит название класса. В обычном режиме нам это не нужно, а вот при работе с пространствами (namespace) — это удобно:

**namespace** test;

**class** A {}

**echo** A::class; //> test\A

*yield* — создание генераторов

Если говорить простым языком: *yield* похожа на return, она также возвращает значение, но она не обрывает работу функции, а приостанавливает её до тех пор пока не будет запрошено следующее значение. Благодаря этому создавать генераторы стало удобнее.

Пример генератора:

**function** generator() {

 **for**( $i = 1; $i <= 3; $i++ ){

 **yield** $i; // выброс значения

 }

}

**foreach**( generator() **as** $value ){

 **echo** "**$value** ";

}

// выведет: '1 2 3 '

Как это работает на самом деле?

*yield* возвращает специальный объект — Generator. Когда функция generator() вызывается в цикле, например foreach, PHP выполнит код функции до первой встречи слова yield, на котором PHP прервет работу функции, запомнит позицию и выбросит значение (объект Generator). Затем, foreach обработает значение и вызовет метод next() у полученного объекта Generator. PHP снова выполнит код функции generator(), только начнет его не с начала, а с прошлой позиции, и опять, до слова yield, которое опять выбросит объект Generator. Работа цикла прервется тогда, когда функция generator() дойдет до конца (не вернет yield), или если она будет прервана с помощью *return;*.

Пример генератора который возвращает пару: ключ/значение:

**function** generator( $input ){

 **foreach**( explode('.', $input) **as** $part ){

 **list**( $num, $name ) = explode(' - ', $part );

 **yield** $num => trim($name);

 }

}

$input = '1 - один. 2 - два. 3 - три';

**foreach**( generator( $input ) **as** $num => $name ){

 **echo** "**$num** (**$name**) ";

}

Кратко о генераторах

— Не добавляют нового функционала в язык
— Быстрее
— Возобновление работы генератора происходит с последнего «выброса» yield
— В генератор можно отправлять значения и исключения (через метод throw())
— Генераторы однонаправлены, т.е. нельзя вернуться назад
— Меньше кода в большинстве случаев, более простые для понимания конструкции

Чтобы лучше понять генераторы прочитайте [эту статью на Хабре](https://habrahabr.ru/post/189796/).

[Подробно о генераторах в документации (англ.)](http://php.net/manual/en/language.generators.syntax.php)

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

API для хэширования паролей

Теперь PHP из коробки предлагает правильный способ хэшировать пароли. Новый API хэширования паролей предоставляет четыре функции:

1. *password\_hash()* — используется для хэширования пароля. В WP для этого есть своя функция [wp\_hash\_password()](http://wp-kama.ru/function/wp_hash_password).

$hash = password\_hash( $passwod, PASSWORD\_DEFAULT );

1. *password\_verify()* — используется для проверки пароля на соответствие хэшу. В WP для этого есть своя функция [wp\_check\_password()](http://wp-kama.ru/function/wp_check_password).
2. **if**( password\_verify( $password, $hash ) ){
3. // Success!

}

1. *password\_needs\_rehash()* — используется для проверки необходимости создать новый хэш.
2. *password\_get\_info()* — возвращает имя алгоритма хеширования и различные параметры, используемые при хэшировании.

*Подробнее*[*читайте в статье на хабре*](http://habrahabr.ru/post/184220/)*.*

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

PHP 5.6

Новые возможности PHP 5.6. [Ссылка на офф.сайт](http://php.net/manual/ru/migration56.new-features.php).

*const PLUS = 1 + 2;* — скалярные выражения в константах/свойствах/аргументах функции

Теперь стало возможным указывать в значения констант примитивные PHP выражения (выражения из скаляров).

Точнее, новинка касается не только констант, а всего где раньше PHP ожидал статическое значение. Теперь вместо статики можно указать выражение из чисел/строк/констант. Если точнее, то PHP выражение можно указывать: в константах/свойствах класса и в значении аргумента функции по умолчанию.

**const** ONE = 1;

**const** TWO = ONE \* 2;

**class** C {

 **const** THREE = TWO + 1;

 **const** ONE\_THIRD = ONE / **self**::THREE;

 **const** SENTENCE = 'Значение THREE равно '. **self**::THREE;

 **public** **function** f( $a = ONE + **self**::THREE ){

 **return** $a;

 }

}

**echo** (**new** C)->f() .' - '. C::SENTENCE; //> 4 - Значение THREE равно 3

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*const ARR = ['a', 'b'];* — константа может хранить массив

Стало возможным держать в константе массивы:

**const** ARR = ['a', 'b'];

**echo** ARR[0]; //> a

*func( ...$args )* или *func( ...[2, 3] )* — неизвестное число аргументов функции или распаковка массива с помощью '...' (splat оператор)

Когда мы не знали заранее, сколько параметров может получить функция, нам приходилось внутри функции обрабатывать переданные параметры с помощью специальных функций: func\_num\_args(), func\_get\_arg(), func\_get\_args().

Теперь они не нужны и мы можем получить все параметры в одной переменной, для этого перед этой переменной нужно указать оператор *...*:

**function** sum( ...$numbers ){

 $plus = 0;

 **foreach**( $numbers **as** $n ){

 $plus += $n;

 }

 **return** $plus;

}

**echo** sum(1, 2, 3); //> 6

Оператор *...* еще называют «Splat Оператор», например в языке Ruby

Еще пример:

**function** func( ...$numbers ){

 **return** print\_r( $numbers, 1 );

}

**echo** func(1, 2, 3);

/\*

Получим:

Array

(

 [0] => 1

 [1] => 2

 [2] => 3

)

\*/

**Быстрая распаковка передаваемых параметров функции**

Теперь с помощью splat оператора *...*, можно указать параметры функции сразу из значений массива:

**function** plus( $a, $b, $c ){

 **return** $a + $b + $c;

}

$array = [2, 3];

**echo** plus( 1, ...$array ); //> 6

// или так

**echo** plus( 1, ...[2, 3] ); //> 6

**Замена медленной функции call\_user\_func\_array()**

Теперь *call\_user\_func\_array(callable $callback, array $param\_arr)*, которая обычно не самая быстрая, можно заменить так:

$params = [1, 2, 3];

$callback( ...$params );

Пример: известная своим неторопливым временем исполнения функция
call\_user\_func\_array(callable $callback, array $param\_arr)
может быть заменена на

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*\*\** — оператор возведения в степень

До php 5.6, чтобы возвести число в степень нужно было использовать функцию *pow(2,2);*, а теперь есть оператор *\*\**:

// пример 1

**echo** $a = 2 \*\* 2; //> 4

// пример 2

$a = 2;

**echo** $a \*\*= 2; //> 8

// пример 3

**echo** $a = 2 \*\* 3 \*\* 2; //> 512 = 2^9

*use function* и *use const* — импорт функций и констант в пространство

Теперь стало возможным при помощью ключевого слова *use* подключать функции или константы другого пространства в наше:

**namespace** our\space {

 const FOO = 42;

 **function** func() { **echo** **\_\_FUNCTION\_\_**; }

}

**namespace** my\space {

 use const our\space\FOO;

 **use** function our\space\func;

 **echo** FOO .' - '. func(); //> 42 - our\space\func

}

-

Остальные новинки PHP 5.6 не связанные с синтаксисом, смотрите [в статье на Хабре](https://habrahabr.ru/post/234899/).

Куда делся PHP 6?

Умер не родившись... В ядро PHP 6 планировали внедрить полную поддержку юникода, но затея оказалась слишком амбициозной, а объем работ слишком велик. К тому моменту, когда это стало понятно, о PHP 6 уже было написано не мало статей. Чтобы не было путаницы, из-за того что новая версия стала преследовать совсем другие цели (производительность) и сильно отличалась по концепции от PHP 6, было решено пропустить PHP 6. Еще одной причиной стало наличие весомого количества недоделанного кода в репозитории PHP, который решили не трогать, чтобы тот в ответ тоже никого не трогал...

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

PHP 7

Что нового в PHP 7. [Ссылка на офф.сайт](http://php.net/manual/ru/migration70.new-features.php).

3 декабря 2015 года было объявлено о выходе PHP 7. Новая версия основывается на экспериментальной ветке PHP, которая изначально называлась phpng (PHPNextGeneration - следующее поколение), и разрабатывалась с упором на увеличение производительности и уменьшение потребления памяти.

Самой важной новинкой стало изменение ядра интерпретатора: теперь он называется PHPNG (Next Generation). Благодаря PHPNG удалось увеличить скорость обработки скриптов почти в двое по сравнению с PHP 5.x. Так же появился более эффективный менеджер памяти.

Прирост в скорости на практике хорошо виден [на этой картинке](http://wp-kama.ru/wp-content/uploads/2016/01/php7-infographic.png). А для WordPress прирост в скорости выглядит так:



[Подробнее смотрите в тестах PHP 7](http://talks.php.net/oz15#/wpbench)

Синтаксические новинки PHP 7:

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*$a ?? ''* — одновременная проверка isset и получение значения

Новый оператор слияния с NULL (NULL coalescing operator) *??* — это сокращение проверки isset и получения значения, если проверка пройдена.

Такая проверка часто была нужна в тернарном операторе *?:*:

// Получит значение $\_GET['foo'], если переменная установлена или не пустая, иначе получит 'default'

$foo = $\_GET['foo'] ?? 'default';

// Запись равносильна этой

$foo = **isset**($\_GET['foo']) ? $\_GET['foo'] : 'default';

// или этой

$foo = @ $\_GET['foo'] ?: 'default';

// удобная проверка при получении $\_GET параметра

**if**( $\_GET['foo'] ?? 0 ){ }

// раньше писали так

**if**( **isset**($\_GET['foo']) && $\_GET['foo'] ){ }

Так же, проверять можно по цепочке:

$foo = $\_GET['foo'] ?? $\_POST['foo'] ?? 'default';

// вернет: $\_GET['foo'], если его нет, то $\_POST['foo'], если нет, то 'default'

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*$a <=> $b* — одновременное выполнение трех сравнений: больше, равно или меньше.

Новый оператор сравнения *<=>* — «spaceship operator». Сравнивает 2 переменные и возвращает результат сравнения в виде числа:

* *-1* — если в сравнении подходит первый символ оператора *<*
* *0* — подходит второй символ *=*
* *1* — подходит третий символ *>*

// Числа

**echo** 1 <=> 1; // 0

**echo** 1 <=> 2; // -1

**echo** 2 <=> 1; // 1

// Дробные числа

**echo** 1.5 <=> 1.5; // 0

**echo** 1.5 <=> 2.5; // -1

**echo** 2.5 <=> 1.5; // 1

// Строки

**echo** "a" <=> "a"; // 0

**echo** "a" <=> "b"; // -1

**echo** "b" <=> "a"; // 1

*Удобен для использования в колбэках для usort().*

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*define('FOO', [1,2]);* — передача массива константе через define()

Константы могут содержать массивы еще с PHP 5.6. Но тогда их можно было передавать только через ключевое слово const. Теперь их можно указывать еще и через define().

define('ANIMALS', ['dog', 'cat', 'bird']);

**echo** ANIMALS[2]; //> bird

*use name\space\{A, B, C as c};* — группировка импорта при помощи use

Теперь для краткой записи, импорт данных в наше пространство можно группировать:

// PHP 7

**use** some\namespace\{ ClassA, ClassB, ClassC as C };

**use** function some\namespace\{ fn\_a, fn\_b, fn\_c };

**use** const some\namespace\{ СonstA, ConstB, ConstC };

// тоже самое до PHP 7

**use** some\namespace\ClassA;

**use** some\namespace\ClassB;

**use** some\namespace\ClassC as C;

**use** function some\namespace\fn\_a;

**use** function some\namespace\fn\_b;

**use** function some\namespace\fn\_c;

**use** const some\namespace\ConstA;

**use** const some\namespace\ConstB;

**use** const some\namespace\ConstC;

*int, float, bool* — новые типы для аргументов функции/метода

Авто-проверка типа передаваемых данных в функции/методы, известная как «контроль типа» (typehint), продолжает развиваться и теперь понимает скаляры: *int*, *float*, *bool*, *string*. Раньше понимались только типы: *array*, *имя класса* или *callable* (с версии 5.4).

Пример:

**function** foo( int $a, bool $b, callable $с, **array** $d, WP\_Post $e ) {

 **return** var\_dump( $a, $b, $c, $d, $e );

}

foo( 1, **true**, 'trim', **array**(1), get\_post(1) );

/\* выведет:

int(1)

bool(true)

NULL

array(1) { [0]=> int(1) }

object(WP\_Post)#2660 (24) { ...данные объекта... }

\*/

// если указать неверный тип:

foo( 'foo', **true**, 'trim', **array**(1), get\_post(1) );

// Получим ошибку Fatal error: Argument 1 passed to A::foo() must be of the type integer, string given

**Режим строгой типизации**

Если указан тип *int* и передать строку *'123'* то проверка все равно будет пройдена, и php превратить строку в число.

***function*** func( int $num ){

 var\_dump( $num );

}

func('123'); //> int(123)

Но что, если нужно получать именно число 123? Для этого можно включить режим строгой типизации, поместив в самое начало файла такую строку:

**declare**(strict\_types=1);

Это объявление должно быть первой строкой в файле, до выполнения какого-либо кода. Оно затрагивает только код файла и только вызовы и возвращаемые значения в этом файле.

Заметка: если строгая типизация указана в файле X, но не указана в файле Y и в файле Y вызывается функция из файла X. То вызов такой функции не будет подвержен строгой типизации!

Читайте по типизации [статью на Хабре](https://habrahabr.ru/post/248721/) и [вот еще интересная статья](https://habrahabr.ru/post/267799/).

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*int, float, bool, array* — указание возвращаемых типов для функции/метода

Указывать принимаемый тип, можно еще с версии PHP 5.3. А вот указать какой тип функция/метод должна вернуть доступно только с версии PHP 7. Тут понимаются все типы: *string*, *int*, *float*, *bool*, *array*, *callable*, *self* (в методах), *parent* (в методах) , *Closure*, *имя класса*, *имя интерфейса*.

Синтаксис:

**function** func( $var ): int{ /\* код функции \*/ }

**function** func( $var ): string{ }

**function** func( $var ): float{ }

**function** func( $var ): bool{ }

**function** func( $var ): **array**{ }

**function** func( $var ): callable{ }

**function** func( $var ): Closure{ }

**function** func( $var ): WP\_Post{ } // может вернуть только объект класса WP\_Post

**class** A **extends** B {

 **function** func( $var ): **self**{ }

 **function** func( $var ): **parent**{ }

}

Рабочие примеры:

// Пример 1:

**function** func( $var ): int {

 **return** $var;

}

**echo** func( 123 ); //> 123

**echo** func( 'asfd' ); //> вызовет ошибку: Fatal error: Uncaught TypeError: Return value of func() must be of the type integer, string returned

// Пример 2: Closure

**function** func(): Closure {

 **return** **function**( $var ){ **return** $var .' + 2 = 3'; };

}

**echo** func()( 1 ); //> 1 + 2 = 3

**Возвращаемые типы при наследовании методов класса**

При наследовании в классах, дочерние методы должны иметь такие же возвращаемые типы как и в родительском классе/интерфейсе:

**class** A {

 **function** func() : int {

 **return** 123;

 }

}

**class** B **extends** A {

 **function** func() : string {

 **return** '123';

 }

 // такое объявление функции вызовет ошибку:

 // Fatal error: Declaration of B::func(): string must be compatible with A::func(): int

 // т.е. тип int должен совпадать!

}

**Навороченный пример того, как можно писать в PHP 7**

Тут сразу несколько новинок:

1. принимаемый и возвращаемый тип;
2. объединение и распаковка параметров с помощью *...*;
3. пример создания анонимной функции с указанием возвращаемого типа данных.

**function** arraysSum( **array** ...$arrays ): **array** {

 **return** array\_map( **function**( **array** $array ): int {

 **return** array\_sum( $array );

 }, $arrays );

}

print\_r( arraysSum( [1,2,3], [4,5,6], [7,8,9] ) );

/\*

Выведет:

Array

(

 [0] => 6

 [1] => 15

 [2] => 24

)

\*/

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*foo()(), $a::$b::$c, $$foo->bar* — единый синтаксис переменных: СЛЕВА НАПРАВО

Очень важная новинка! Теперь обращения к сложносочиненным переменным разбираются последовательно СЛЕВА НАПРАВО.

Примеры новых возможностей:

// можно не указывать комбинирующие скобки

$foo()['bar']()

[$obj1, $obj2][0]->prop

getStr(){0}

// поддерживает вложенность ::

$foo['bar']::$baz // значение: ($foo['bar'])::$baz

$foo::$bar::$baz // значение: ($foo::$bar)::$baz

$foo->bar()::baz() // значение: ($foo->bar())::$baz

// поддерживает вложенные ()

foo()() // вызывает результат вызова foo(): (foo())()

$foo->bar()() // ($foo->bar())()

Foo::bar()() // (Foo::bar())()

$foo()() // ($foo())()

// Операторы над выражениями заключенными в ()

(**function**() { ... })() // IIFE синтаксис JS

($obj->closure)()

// и т.д.

(...)['foo']

(...)->foo

(...)->foo()

(...)::$foo

(...)::foo()

(...)()

// все операции по разименованию скаляров (не очень полезно)

"string"->toLower()

[$obj, 'method']()

'Foo'::$bar

Примеры разницы старого и нового распознавания:

// строка // старое понимание // новое понимание

$$foo['bar']['baz'] ${$foo['bar']['baz']} ($$foo)['bar']['baz']

$foo->$bar['baz'] $foo->{$bar['baz']} ($foo->$bar)['baz']

$foo->$bar['baz']() $foo->{$bar['baz']}() ($foo->$bar)['baz']()

Foo::$bar['baz']() Foo::{$bar['baz']}() (Foo::$bar)['baz']()

Старый код написанный с использованием *{}* для обработки переменных возможно не будет работать в новой версии PHP7.

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

foreach — изменена логика работы

Теперь foreach не переключает автоматически внутренний указатель перебираемого массива, т.е. next() не работает автоматически.

Так же, если значение массива передается в foreach по ссылке, то он всегда работает с оригиналом массива. Если значение не передается по ссылке, то foreach всегда работает с копией массива и оригинальный массив не затрагивается при изменении его внутри foreach.

Переключение указателей и влияние на работу цикла в PHP 7:

// Пример 1: автоматически не переключает внутренний переключатель next()

$a = [1,2,3]; **foreach**($a **as** $v) { **echo** $v ."-". current($a) ." "; } // 1-1 2-1 3-1 (php5: 1-2 2-2 3-2)

$a = [1,2,3]; $b = &$a; **foreach**($b **as** $v) { **echo** $v ."-". current($b) ." "; } // 1-1 2-1 3-1 (php5: 1-2 2-3 3-)

$a = [1,2,3]; $b = $a; **foreach**($b **as** $v) { **echo** $v ."-". current($b) ." "; } // 1-1 2-1 3-1 (php5: 1-1 2-1 3-1)

$a = [1,2,3]; **foreach**($a **as** & $v) { **echo** $v ."-". current($a) ." "; } // 1-1 2-1 3-1 (php5: 1-2 2-3 3-)

// В PHP 7 все строки выведут один результат: 1-1 2-1 3-1

// В PHP 5 он будет разный

// Пример 2: внутренний переключатель можно переключить с помощью next()

$a = [1,2,3]; **foreach**($a **as** $v) { **echo** $v ."-". current($a) ." "; next($a); } // 1-1 2-2 3-3 (php5: 1-2 2-3 3-)

$a = [1,2,3]; $b = &$a; **foreach**($b **as** $v) { **echo** $v ."-". current($b) ." "; next($b); } // 1-1 2-2 3-3 (php5: 1-2 2-3 3-)

$a = [1,2,3]; $b = $a; **foreach**($b **as** $v) { **echo** $v ."-". current($b) ." "; next($b); } // 1-1 2-2 3-3 (php5: 1-2 2-3 3-)

$a = [1,2,3]; **foreach**($a **as** & $v) { **echo** $v ."-". current($a) ." "; next($a); } // 1-1 2-2 3-3 (php5: 1-2 2-3 3-)

// В PHP 7 все строки выведут один результат: 1-1 2-2 3-3

// В PHP 5 результат тоже будет один, но другой: 1-2 2-3 3-

// Пример 3:

// при &$v foreach работает с оригиналом $a и изменение массива влияет на цикл

$a = [1,2,3]; **foreach**($a **as** &$v) { **echo** "**$v** "; **unset**($a[1]); } // 1 3 (php5: 1 3)

$a = [1,2,3]; $b = &$a; **foreach**($b **as** &$v) { **echo** "**$v** "; **unset**($a[1]); } // 1 3 (php5: 1 3)

// в PHP 5 и PHP 7 совпадают

// при $v foreach работает с копией $a и изменение массива НЕ влияет на цикл

$a = [1,2,3]; **foreach**($a **as** $v) { **echo** "**$v** "; **unset**($a[1]); } // 1 2 3 (php5: 1 2 3)

$a = [1,2,3]; $b = &$a; **foreach**($b **as** $v) { **echo** "**$v** "; **unset**($b[1]); } // 1 2 3 (php5: 1 3)

// в PHP 5 и PHP 7 результаты отличаются

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*$class = new class{}* — анонимные классы

Анонимные классы позволяют делать тоже самое что и обычные классы: передавать данные в конструктор, наследовать другие классы, использовать трейты и т.п.

$class = **new** **class** {

 **public** **function** **echo**( $msg ){

 **echo** $msg;

 }

};

$class->echo('Привет!'); // выведет на экран: "Привет!"

Расширение классов работает как и ожидается:

**class** Foo {}

$child = **new** **class** **extends** Foo {};

var\_dump( $child **instanceof** Foo ); //> true

Использование треитов:

**trait** Foo {

 **public** **function** method() {

 **return** "bar";

 }

}

$class = **new** **class** {

 **use** Foo;

};

var\_dump( $class->method() ); //> string(3) "bar"

Подробнее про анонимные классы [читайте в документации](http://php.net/manual/ru/language.oop5.anonymous.php) и на [wiki.php.net](https://wiki.php.net/rfc/anonymous_classes).

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*yield ... return 99;* — возврат (return) выражений в генераторах

Функции-генераторы появились в PHP 5.5. Но там можно было использовать return, только чтобы прервать работу генератора. Теперь return может возвращать выражение (значение/массив/другой генератор), а не только NULL. Но сделать это можно только в конце работы генератора.

Получить возвращенное значение можно методом *getReturn()*, но только по завершении работы генератора.

Возможность явно вернуть последнее значение упрощает работу с генераторами:
теперь не нужно проверять является ли значение последним, а просто вызываем getReturn().

**function** gen() {

 **yield** 1;

 **yield** 2;

 **return** 3;

}

$gen = gen();

// если генератор еще ничего не вернул, то вызов такой строки

// echo $gen->getReturn();

// вызовет ошибку: Fatal error: Uncaught Exception: Cannot get return value of a generator that hasn't returned

**foreach**( $gen **as** $val ) {

 **echo** $val;

}

**echo** $gen->getReturn();

// результат работы этого кода выведет на экран: 123

Подробнее: [Generator Return Expressions](https://wiki.php.net/rfc/generator-return-expressions)

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

*yield from gen()* — делегирование генераторов

Позволяет разбить сложный генератор на несколько простых.

Для этого используется новый синтаксис: *yield from <expr>*, где <expr> может быть значением (скаляром), массивом или другим генератором.

<expr> будет работать до тех пор, пока возвращает данные, затем выполнение продолжится в генераторе откуда <expr> был вызван. Смотрите пример:

**function** gen() {

 **yield** 1;

 **yield** from gen2();

 **yield** 4;

}

**function** gen2(){

 **yield** 2;

 **yield** 3;

}

$gen = gen();

**foreach** ( $gen **as** $val ) {

 **echo** $val;

}

// результат работы этого кода: 1234

Пример с массивом:

**function** g() {

 **yield** 1;

 **yield** from [2, 3, 4];

 **yield** 5;

}

$g = g();

**foreach** ( $g **as** $yielded ) {

 **echo** $yielded;

}

// выведет: 12345

Пример с return из дочернего генератора:

**function** gen() {

 **yield** 1;

 $sub\_gen = **yield** from sub\_gen();

 **yield** 4;

 **return** $sub\_gen;

}

**function** sub\_gen() {

 **yield** 2;

 **yield** 3;

 **return** 42;

}

$gen = gen();

**foreach**( $gen **as** $val ) {

 **echo** $val;

}

**echo** ' - '. $gen->getReturn();

// выведет: 1234 - 42

Подробнее: [Generator Delegation](https://wiki.php.net/rfc/generator-delegation)

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

Еще новинки PHP 7.0

1. Синтаксис конструкторов в стиле PHP 4 (имя метода конструктора совпадает с именем класса) теперь считается устаревшим.
2. Статичные вызовы *::* нестатичных методов теперь считаются устаревшими.
3. *list()* — изменение поведения. В PHP 5, list() устанавливал значения начиная с правого крайнего значения указанного массива, в PHP 7 параметры устанавливаются начиная с левого крайнего значения массива. Так же в PHP 5 list() умела разбивать строки на символы, в PHP 7 не работает со строками вообще...
4. // Пример 1: обратное чтение
5. // Если используются обычные переменные, то разницы нет
6. **list**( $a, $b, $c ) = ['apple', 'bannana', 'cherry', 'damson'];
7. var\_dump( $a, $b, $c ); // php5 и php7 вернут: apple bannana cherry
8. // А вот если устанавливаются элементы массива, то порядок будет отличаться
9. $arr = [];
10. **list**( $arr['a'], $arr['b'], $arr['c'] ) = ['apple', 'bannana', 'cherry', 'damson'];
11. print\_r( $arr );
12. /\*
13. PHP 7
14. Array
15. (
16. [a] => apple
17. [b] => bannana
18. [c] => cherry
19. )
20. PHP 5
21. Array
22. (
23. [c] => cherry
24. [b] => bannana
25. [a] => apple
26. )
27. \*/
28. // Пример 2: разбивание строк
29. $str = 'ab';
30. **list**( $a, $b ) = $str;
31. var\_dump( $a, $b );
32. // В PHP 7: NULL NULL

// В PHP 5: string(1) "a" string(1) "b"

1. Поддержка юникод управляющих (escape-) последовательностей. Т.е. в строках *""* и heredoc можно использовать конструкцию *\uXXXX* для создания юникод символа. Вот так:

**echo** "\u{1F602}"; //> ?

Подробнее читайте: [Unicode Codepoint Escape Syntax](https://wiki.php.net/rfc/unicode_escape)

1. Класс IntlChar. Cодержит методы и константы для работы с юникодом.
2. printf('%x', IntlChar::CODEPOINT\_MAX); // 10ffff
3. **echo** IntlChar::ord('@'); //> 64
4. **echo** IntlChar::chr( 64 ); //> @
5. **echo** "\u{1F602}"; //> ?
6. **echo** IntlChar::ord("\u{1F602}"); //> 128514

**echo** IntlChar::chr( 128514 ); //> ?

1. Функция *intdiv()* — делит 2 числа и возвращает только целую часть от деления:
2. **echo** intdiv(10, 3); //> 3

**echo** intdiv(5, 2); //> 2

1. *session\_start()* умеет получать параметры (стандартные настройки сессий из php.ini):

session\_start(['cache\_limiter' => 'private']);

1. Функция *preg\_replace\_callback\_array()* — альтернатива preg\_replace\_callback(). Позволяет передать в качестве обратной функции - массив *['/regex'/ => callback, ...]*:
2. $str = 'a1a2a3';
3. $array = [
4. '~[0-9]~' => **function** ( $m ){ **return** $m[0] \* 2; },
5. '~a~' => **function** ( $m ){ **return** $m[0] . '-'; }
6. ];

**echo** preg\_replace\_callback\_array( $array, $str ); //> a-2a-4a-6

1. Можно использовать глобальные ключевые слова в названиях методов. Т.е. раньше нельзя было назвать метод словами: with/new/for/foreach/... — это приводило к ошибке. Теперь можно:
2. **Class**::new('Project Name');

$class->for('purpose here');

Подробнее о новинках PHP 7 читайте в [этой статье](http://www.skillz.ru/dev/php/article-php7-whats-new.html) и [вторая часть](http://www.skillz.ru/dev/php/article-php7-whats-new-part-2.html).

[к началу](http://wp-kama.ru/id_6310/sintaksis-php-5-3-5-4-5-5-5-6-7-chto-novogo.html#kcmenu)

PHP 7.1

*void* — возвращаемый тип

Теперь функции и методы, которые не должны ничего возвращать, можно помечать возвращаемым типом void. Оператор return при этом должен отсутствовать или должен быть пустым - *return;*. Вызов *return null;* вызовет ошибку.

**function** someMethod(): void {

 // работает если return отсутствует

 // работает с return;

 // не работает если return null;

 // не работает если return 123;

}

*iterable* — новый псевдо-тип

Введен новый тип iterable для передаваемых/возвращаемых значений. Может использоваться при передаче массивов или объектов, которые соответствуют интерфейсу Traversable.

**function** myfunc( iterable $data ){

 **foreach**( $data **as** $k => $v ){

 **echo** $k, ':', $v, PHP\_EOL;

 }

}

// массив

myfunc([10, 20, 30]); // 0:10 1:20 2:30

// объект

myfunc( **new** SplFixedArray(3) ) // 0: 1: 2:

// генератор

**function** myGen(){

 **yield** 10;

 **yield** 20;

 **yield** 30;

}

myfunc( myGen() ); // 0:10 1:20 2:30

Подробнее [читайте по ссылке](https://wiki.php.net/rfc/iterable).

*null* — тип передаваемых/возвращаемых значений

В PHP 7.0 стало возможным указать тип возвращаемых/передаваемых значений, но типизация не допускала использование null в качестве значения параметра.

В PHP 7.1 для разрешения null-значений перед типом параметра указывается "?":

**function** myfunc(?int $i) : ?int {

 var\_dump($a);

 **return** $a;

}

myfunc(20); // int(20)

myfunc(**null**); // null

myfunc(); // Ошибка: Uncaught Error: Too few arguments to function name(), 0 passed

Подробнее [читайте по ссылке](http://wp-kama.ru/id_6310/%5Bhttps%3A/wiki.php.net/rfc/iterable%5D%28https%3A/wiki.php.net/rfc/nullable_types%29).

*{-1}* — отрицательное значение смещения в строках

Добавлена возможность использовать отрицательное значение для смещения в строках

**echo** $msg[-1]; // последний символ

**echo** $msg{-1}; // последний символ

*['key'=>$a] = ['key'=>'Значение']* — поддержка ключей и новый синтаксис list()

Теперь, можно использовать новый синтаксис - аналог list(), в котором разрешено использовать строковые ключи:

['test'=>$a, 'name'=>$b] = ['name'=>'Hello', 'test'=>'World!'];

var\_dump($a); // World!

var\_dump($b); // Hello

Замена list():

**list**($one, $two) = ['один', 'два']; // list style

[$one, $two] = ['один', 'два']; // [] style

**echo** "**$one**, **$two**";

Область видимости констант в классах

Конец публичным константам, теперь для констант можно указать видимость:

**class** ConstClass {

 **const** CONST\_ONE = 1; // public

 **public** **const** CONST\_TWO = 2;

 **protected** **const** CONST\_THREE = 3;

 **private** **const** CONST\_FOUR = 4;

}

Заметки по PHP 7.1

PHP движется в сторону строгой типизации данных и при переходе на 7.1 я столкнулся с ФАТАЛЬНОЙ ошибкой. И мне это показалось очень странным. Приведу пример:

$foo = '';

$foo['bar'] = 'мир'; // Warning: Illegal string offset 'bar'

$foo['bar'][] = 'мир'; // Fatal error: Uncaught Error: Cannot use string offset as an array

 // фатальная ошибка: нельзя использовать отступ строки как массив...

При Warning PHP еще работает, а дальше уже нет! А еще в 7.0 код просто работал, даже без предупреждений и нотисов... Похоже на недоработку в PHP 7.1.

К примеру, ошибка такого типа есть в популярном плагине WP Super Cache (отписал авторам, надеюсь скоро поправят).

Весь список изменение [смотрите по ссылке](https://github.com/php/php-src/blob/PHP-7.1.0/UPGRADING) и [по этой](http://www.skillz.ru/dev/php/article-php71-whats-new.html).