[Документация PHP](https://www.php.net/manual/ru/langref.php)

[Установка PHP](https://docs.google.com/document/d/1ZTcTq9D3u1sCHe1w-coF_nv4OxyD_86K5v6j_ZswgKE/edit#heading=h.sdumf9k5gfiq)

[Онлайн-песочница PHP](https://www.w3schools.com/php/phptryit.asp?filename=tryphp_compiler)

[Задания для самостоятельной работы](https://docs.google.com/document/d/1-TZHwul6t3XxytLiRfBTOPTBW92BnqqvU4a-dNUkINU/edit#heading=h.jppre9nusz8z)

Курс по PHP

# Введение

## История создания

У PHP очень необычная история в сравнении с другими языками. В 90-е годы XX века, на заре развития популярности и массовости интернета, для формирования динамического ответа от веб-сервера обычно использовалось обращение к специальным CGI-программам (Common Gateway Interface - "интерфейс общего шлюза"), которые запускались веб-сервером. И да, такие программы до сих пор существуют, хотя уже не популярны и малочисленны. В своей работе вы неоднократно встретите в корне проекта директорию с названием cgi или cgi-bin, предназначенную для хранения таких скриптов. CGI-скрипты могут быть написаны на любом языке программирования, главное чтобы они были написаны по стандарту CGI, поддерживали работу со стандартным вводом/выводом). Итак, в 1994 году датский программист Расмус Лердорф написал несколько CGI-скриптов на Perl (это язык программирования, предлагающий удобные и быстрые методы обработки строк). Единственной задачей тех скриптов был учет и вывод статистики просмотров его страницы с онлайн-резюме.

Спустя несколько месяцев развития, когда функционал и возможности скриптов увеличились, а быстроты языка Perl для проектов стало не хватать, Лердорф переписал скрипты на язык Си. В июне 1995 Расмус открыл общественности исходный код первой версии набора скриптов Personal Homepages Tools (PHP Tools - "Инструменты для персональных домашних страниц"). Данный набор скриптов позволял вести файлы логов и просматривать их содержимое, предоставлять доступ к страницам по паролю, отслеживать посещение страниц пользователями по их email и переходам с referring адресов, позволял создавать простые формы.

В сентябре того же года Расмус расширил PHP и на короткое время убрал из названия PHP. Теперь в виде инструмента FI (сокращение от Form Interpreter - "Интерпретатор Форм"), новая реализация включала в себя некоторые основные функциональные возможности того PHP, который мы знаем сегодня. Она имела Perl-подобные переменные, автоматическую интерпретацию форм и встраиваемый в HTML синтаксис. Синтаксис языка был похож на Perl, хотя и был гораздо более ограниченным, простым, и в некоторой степени противоречивым. Для того, чтобы вставлять код в HTML-файл, разработчикам пришлось использовать HTML комментарии. Хотя этот метод был не совсем хорошо принят, FI по-прежнему набирал популярность в качестве CGI-инструмента, но всё-таки не в качестве языка. Однако, перемены начались в следующем месяце, когда в октябре 1995 года Расмус выпустил полностью переписанный код. С вернувшимся именем PHP, но уже сокращенным от "Personal Home Page Construction Kit".

Код получил ещё одно существенное преобразование в апреле 1996 года. Объединив названия предыдущих версий, Расмус представил PHP/FI. Реализации второго поколения начали по-настоящему развивать PHP из набора инструментов в самостоятельный язык программирования. PHP включал в себя встроенную поддержку для различных баз данных, cookies, поддержку определяемых пользователем функций и многое другое. В июне PHP/FI была присвоена версия 2.0.

Пример кода на PHP/FI.

<!--include /text/header.html-->

<!--getenv HTTP\_USER\_AGENT-->

<!--ifsubstr $exec\_result Mozilla-->

Эй, у вас Нетскейп!<p>

<!--endif-->

<!--sql database select \* from table where user='$username'-->

<!--ifless $numentries 1-->

Извините, такой записи нет<p>

<!--endif exit-->

Добро пожаловать, <!--$user-->!<p>

На вашем счету осталось <!--$index:0--> "кредитов".<p>

<!--include /text/footer.html-->

Знать эту историю успеха важно, как минимум, по одной причине. Создание языков программирования - серьезное занятие, требующее определенной квалификации от тех, кто это делает. Расмус признавался сам, что он не понимал ничего в том, как создавать языки. И более того, он и не пытался сделать язык. Все это оставило большой отпечаток на PHP:

* Разнородное именование функций: они добавлялись по ходу дела в те времена, когда об этом никто не думал.
* В целом именование внутри языка очень непоследовательное и часто прямо противоречит PSR (официальные рекомендации по оформлению и организации кода в PHP).
* Путаница с передачей параметров: одни и те же по смыслу функции могут принимать на вход параметры по-разному: например, в обратном порядке. Поэтому в PHP сложнее запомнить сигнатуры функций.
* Странная система ошибок и предупреждений (Fatal, Warning, Notice).

Этот список можно продолжать и дальше, но в процессе работы вы и сами столкнетесь с неожиданностями.

Но, несмотря на вышесказанное, современный PHP - мощный язык с большим количеством различных возможностей, и на котором пишут миллионы программистов по всему миру. PHP - самый простой и удобный способ войти в веб-разработку.

## Версии

* 3.x - июнь 1998 года. Отличительной особенностью была возможность расширения ядра дополнительными модулями, что привлекло к PHP множество сторонних разработчиков, работающих над своими модулями, что дало PHP возможность работать с огромным количеством баз данных, протоколов, поддерживать большое число API. Большое количество разработчиков привело к быстрому развитию языка и стремительному росту его популярности. С этой версии акроним php расшифровывается как «PHP: hypertext Preprocessor», вместо устаревшего «Personal Home Page».
* 4.x - май 2000 года. Новый переписанный и более производительный движок, названный Zend Engine, поддержка сессий, буферизация вывода и др.
* 5.x - июль 2004 года. Переработка и оптимизация, новое ядро Zend Engine 2, введены пространства имён, замыкания и др..
* 7.x - декабрь 2015 года. Упор на увеличение производительности и уменьшение потребления памяти. В новой версии добавлена возможность указывать тип возвращаемых из функции данных, добавлен контроль передаваемых типов для данных и др..
* 8.x - ноябрь 2020 года. Добавлена поддержка union типизации (множественной), [JIT-компиляция](https://habr.com/ru/company/otus/blog/509598/) (динамическая компиляция кода непосредственно во время работы программы, с помощью чего достигается высокая скорость выполнения, сравнимая с обычными компилируемыми языками) и атрибуты (также известны как аннотации).

## Область применения

PHP может все. Существуют три основных области применения PHP.

* Создание скриптов для выполнения на стороне сервера. PHP традиционно и наиболее широко используется именно таким образом. Для этого вам будут необходимы три вещи. Интерпретатор PHP (в виде программы CGI или серверного модуля), веб-сервер и браузер. Для того чтобы можно было просматривать результаты выполнения PHP-скриптов в браузере, нужен работающий веб-сервер и установленный PHP. Просмотреть вывод PHP-программы можно в браузере, получив PHP-страницу, сгенерированную сервером.
* Создание скриптов для выполнения в командной строке. Вы можете создать PHP-скрипт, способный запускаться без сервера или браузера. Все, что вам потребуется - парсер PHP. Такой способ использования PHP идеально подходит для скриптов, которые должны выполняться регулярно, например, с помощью cron (на платформах \*nix или Linux) или с помощью планировщика задач (Task Scheduler) на платформах Windows. Эти скрипты также могут быть использованы в задачах простой обработки текстов, парсера сайтов, генераторов изображений, файлов PDF и пр.
* Создание оконных приложений, выполняющихся на стороне клиента. Конечно, PHP является не самым лучшим языком для создания подобных приложений. Для создания таких приложений потребуется использовать модуль PHP-GTK (проект не развивается с 2015 года), который не поставляется вместе с основным дистрибутивом PHP.

# Основы

## Синтаксис

Синтаксис PHP подобен синтаксису языка Си. Некоторые элементы, такие как ассоциативные массивы и цикл foreach, заимствованы из Perl. Для написания простейшего скрипта не требуется описывать какие-либо переменные, используемые модули и т. п. Любой скрипт может начинаться непосредственно с оператора **<?php** . Простейшая программа Hello world на PHP выглядит следующим образом:

<?php

echo 'Hello, world!';

?>

Также возможен более короткий вариант вывода строки:

<?='Hello, world!'?>

Открывающий тег вида **<?=** используется для сокращенной записи конструкций, используемых для вывода строки.

PHP исполняет код, находящийся внутри ограничителей, таких как **<?php** **?>**. Всё, что находится вне ограничителей, выводится без изменений. В основном это используется для вставки PHP-кода в HTML-документ, например, так:

<html>

<head>

<title>Тестируем PHP</title>

</head>

<body>

<?php echo 'Привет мир!';?>

</body>

</html>

Помимо ограничителей **<?php ?>**, допускается (но признано устаревшим) использование сокращенного варианта **<? ?>**. Работа сокращенных конструкций определяется в конфигурационном файле php.ini.

Имена переменных начинаются с символа **$**, тип переменной объявлять не нужно. Имена переменных и констант чувствительны к регистру символов. Имена классов, методов классов и функций к регистру символов не чувствительны.

Инструкции разделяются с помощью точки с запятой **;** , за исключением некоторых случаев, после объявления конструкции **if/else** и циклов.

Переменные в функцию можно передавать как по значению, так и по ссылке (используется знак **&**).

PHP поддерживает три типа комментариев: в стиле языка Си (ограниченные **/\* \*/**), C++ (начинающиеся с **//** и идущие до конца строки) и оболочки UNIX (с **#** до конца строки).

## Переменные

В PHP имя переменной и её содержимое - это разные вещи, поэтому одно содержимое может иметь разные имена. Переменная - это именованная ячейка в памяти, содержащая адрес в памяти на присвоенное значение. Если значение не присвоено, то адрес на значение нулевой.

В PHP переменные объявляются в символа **$**. Имя переменной чувствительно к регистру. Правильное имя переменной должно начинаться с буквы или символа подчеркивания и состоять из букв, цифр и символов подчеркивания в любом количестве.

<?php

// это разные переменные

$foo = 'foo';

$Foo = 'Foo';

$4site = 'значение'; // неверно; начинается с цифры

$\_4site = 'значение'; // верно; начинается с символа подчеркивания

Существует два основных стиля именования переменных. Это camelCase и snake\_case. Есть и другие, но в работе мы их не используем.

Важно именовать переменные корректно, чтобы другому программисту, работающему с вашим кодом, было предельно понятно как все устроено. В конечном итоге это значительно сократит время на изучение кода и разработку в целом. Также снизится количество багов.

Для правильного именования переменной достаточно ответить на вопрос: Что будет храниться в переменной? Название переменной является существительным, желательно не более 3 слов. Предлоги обычно опускаются для краткости.

## Типы данных

В любом языке программирования существует такое понятие как типы данных. По этому принципу языки подразделяются на строго типизированные и динамически типизированные.

Различие в том, что в строго типизированных языках при объявлении переменной необходимо указывать, какой тип данных она будет содержать. В функциях необходимо указывать, какой тип данных у каждого аргумента и какой тип данных она будет возвращать.

PHP является языком программирования с динамической типизацией, не требующим указания типа при объявлении переменных, равно как и самого объявления переменных.

К скалярным типам данных относятся:

* целочисленный тип (int)
* число с плавающей точкой (float)
* логический тип (bool)
* строковый тип (string)

К нескалярным типам относятся:

* массив (array)
* объект (object)
* внешний ресурс (resource)
* неопределенное значение (null)

## Типизация данных в PHP

Язык допускает типизацию аргументов, возвращаемых значений функций и методов классов, а также, начиная с версии 7.4 типизацию полей классов.

### Преимущества статической типизации

* Проверки типов происходят только один раз — на этапе компиляции. А это значит, что нам не нужно будет постоянно выяснять, не пытаемся ли мы делить число на строку (и либо выдать ошибку, либо осуществить преобразование).
* Скорость выполнения. Из предыдущего пункта ясно, что статически типизированные языки практически всегда быстрее динамически типизированных.
* При некоторых дополнительных условиях, позволяет обнаруживать потенциальные ошибки уже на этапе компиляции.
* Ускорение разработки при поддержке IDE (отсеивание вариантов, заведомо не подходящих по типу).

### Преимущества динамической типизации

* Простота создания универсальных коллекций — куч всего и вся (редко возникает такая необходимость, но когда возникает динамическая типизация выручит).
* Удобство описания обобщенных алгоритмов (например сортировка массива, которая будет работать не только на списке целых чисел, но и на списке вещественных и даже на списке строк).
* Легкость в освоении — языки с динамической типизацией обычно очень хороши для того, чтобы быстро начать программировать.

## Манипуляции с типами переменных

PHP не требует (и не поддерживает) явного типа при определении переменной; тип переменной определяется по контексту, в котором она используется. То есть, если вы присвоите значение типа string переменной $var, то $var изменит тип на string. Если вы затем присвоите $var значение типа int, она станет целым числом (int).

<?php

// переменная $foo получит целочисленный тип

$foo = 2022;

// переменная $foo получит вещественный тип

$foo = 2022.0;

// переменная $foo получит строковый тип

$foo = 'some string value';

// как и здесь

$foo = '2022';

// и здесь тоже строка

$foo = '2022.00';

Специальное значение **null** представляет собой переменную без значения. **null** - это единственно возможное значение **типа** **null**. Переменная считается **null**, если:

<?php

// переменной не было присвоено ранее никакого значения

var\_dump($some\_unknown\_var);

// переменной присвоена константа null

$foo = null;

var\_dump($foo);

// переменная была удалена с помощью unset(), после чего при обращении к ней считается, что ей ранее не было присвоено какое-либо значение

$foo = 'some value';

unset($foo);

var\_dump($foo);

// можно удалять сразу несколько переменных

unset($foo, $bar, $and\_more\_vars);

$bar = 'some value';

// проверить, что переменной было присвоено какое-то значение, отличное от null

echo isset($bar); // 1

echo isset($some\_unknown\_var);

Для проверки типа переменной используют функцию, соответствующую проверяемому типу.

<?php

$foo = 10;

// проверяет, является ли переменная булевого типа

echo is\_bool($foo);

// проверяет, является ли переменная целым числом

echo is\_int($foo);

// проверяет, является ли переменная числом с плавающей точкой

echo is\_float($foo);

// проверяет, является ли переменная строкой

echo is\_string($foo);

// проверяет, является ли переменная объектом

echo is\_object($foo);

// проверяет, является ли переменная массивом

echo is\_array($foo);

В процессе вычисления результата части выражения, например при присвоении значения переменной, может произойти **неявное преобразование** типа. Примером может служить операция умножения. Если какой-либо из операндов является **float**, то все операнды интерпретируются как **float**, и результатом также будет **float**. В противном случае операнды будут интерпретироваться как **int**, и результат также будет **int**. Обратите внимание, что это НЕ меняет типы самих операндов; меняется только то, как они вычисляются и сам тип выражения.

<?php

// переменная $foo получит строковый тип со значением '2022'

$foo = '2022';

// переменная $bar получит целочисленный тип со значением 2022, при этом тип переменной $foo остался строковым

$bar = $foo \* 1;

$bar = $foo \* '1';

// переменная $bar получит вещественный тип со значением 2022, при этом тип переменной $foo остался строковым

$bar = $foo \* 1.0;

$bar = $foo \* '1.0';

// переменная $bar получит вещественный тип со значением 2022

$bar = 2022 \* 1.0;

$bar = 2022 \* '1.0';

Если вы хотите, чтобы переменная принудительно вычислялась как определенный тип, то используются **явное приведение типа**. Синтаксис приведения типов в PHP схож с C: имя требуемого типа записывается в круглых скобках перед приводимой переменной. На проектах более распространено явное приведение типов, используя функцию соответствующую нужному типу.

<?php

// $foo - это целое число

$foo = 10;

// преобразование в булев тип

$bar = (bool) $foo;

$bar = (boolean) $foo;

$bar = boolval($foo);

// преобразование в целочисленный тип

$bar = (int) '2022';

$bar = (integer) '2022.99';

$bar = intval('-2022.99');

// преобразование в вещественный тип

$bar = (float) $foo;

$bar = floatval($foo);

// преобразование в строковый тип

$bar = (string) $foo;

$bar = strval($foo);

$bar = "$foo"; // в двойные кавычки

// преобразование в массив

$bar = (array) $foo;

$bar = [$foo];

Следует упомянуть некоторые особенности **приведения к целочисленному** типу.

<?php

// значение null всегда преобразуется в ноль

echo intval(null); // 0

// false преобразуется в 0, а true - в 1.

echo intval(false);

echo intval(true); // 1

// вещественное значение переменной при преобразовании в int будет округлено в сторону нуля

echo intval(2022.99); // 2022

echo intval(-2022.99); // 2022

// если переменная строкового типа содержит число или ведущую числовую последовательность, тогда она будет преобразована в соответствующее целое число, в противном случае она преобразуется в ноль

echo intval('2022'); // 2022

echo intval('-2022'); // -2022

echo intval('-2022.43'); // -2022

echo intval('-2022.99'); // -2022

echo intval('2o22'); // 2

echo intval('--2022'); // 0

echo intval('Vasya\_95'); // 0

Одно из самых важных умений в программировании на PHP - это быстрое и ясное восприятие **приведения к булеву** значения переменной любого типа, т.к. именно булевые значения (истина/ложь) используются в вычислении логических операций и условиях ветвления кода. *При преобразовании в bool, значения, которые можно воспринять как ноль, пустота, отсутствие какого-либо значения, рассматриваются как* ***false****. И наоборот, значения, которые можно воспринять как отличные от нуля и не пустота, рассматриваются как* ***true****.*

<?php

// само значение false

var\_dump(boolval(false)); // bool(false)

// integer 0 (ноль)

var\_dump(boolval(0)); // bool(false)

// float 0.0 (ноль) и -0.0 (минус ноль)

// пустая строка, и строка "0"

var\_dump(boolval('')); // bool(false)

var\_dump(boolval('0')); // bool(false)

// массив без элементов

var\_dump(boolval([])); // bool(false)

// особый тип NULL (включая неустановленные переменные)

var\_dump(boolval(null)); // bool(false)

var\_dump(boolval($some\_unknown\_var)); // bool(false)

var\_dump(boolval(1)); // bool(true)

var\_dump(boolval(-2)); // bool(true)

var\_dump(boolval('foo')); // bool(true)

var\_dump(boolval(2.3e5)); // bool(true)

var\_dump(boolval([-1, 0, 1, 2, 3])); // bool(true)

## Область видимости переменных

Область видимости переменной - это контекст, в котором эта переменная определена. В большинстве случаев все переменные PHP имеют только одну область видимости.

Однако определение (тело) пользовательской функции задает локальную область видимости данной функции. Любая используемая внутри функции переменная по умолчанию ограничена локальной областью видимости функции. Например:

<?php

error\_reporting(0);

// глобальная область видимости

$a = 1;

// переменная $a будет доступна внутри включённого скрипта b.php

include 'b.php';

function test() {

// переменная $b в локальной области видимости тела функции

$b = 2;

// никакого вывода, т.к. $a локально не определена

echo $a;

}

// не сгенерирует никакого вывода

test();

// никакого вывода, т.к. $b глобально не определена

echo $b;

Доступ к переменным глобальной области видимости можно получить используя ключевое слово **global**  или используя специальный суперглобальный массив **$GLOBALS**.

<?php

error\_reporting(0);

// глобальная область видимости

$a = 1;

function test\_global() {

// указали, что используется переменная $a из глобальной области видимости

global $a;

echo $a;

}

function test\_superglobal() {

// изменим переменную $a из глобальной области видимости

$\_GLOBALS['a'] = 2;

echo $\_GLOBALS['a'];

}

test\_global(); // 1

test\_superglobal(); // 2

## Включаемые файлы

Выражения **include** и **require** включают и выполняют указанный файл. Файлы включаются исходя из пути указанного файла. Конструкция **include** выдаст **E\_WARNING**, если не сможет найти файл, которое позволит скрипту продолжить выполнение. Поведение отлично от **require**, который выдаст **E\_ERROR**.

<?php

// Так как include - это специальная языковая конструкция, круглые скобки не обязательны вокруг аргумента

// Подключение скрипта consts.php из директории текущего скрипта

include ('consts.php');

// Подключение скрипта autoload.php с относительным путем к директории текущего скрипта

// в случае отсутсвия файла скрипт прекратит свою работу

require '../autoload.php';

// Подключение скрипта bootstrap.php с абсолютным путем от корня системы к файлу

include '/home/kach/project/bootstrap.php';

Когда файл включается, его код наследует ту же область видимости переменных, что и строка, на которой произошло включение. Все переменные, доступные на этой строке во включающем файле, будут также доступны во включаемом файле. Однако все функции и классы, объявленные во включаемом файле, имеют глобальную область видимости.

Если включение происходит внутри функции включающего файла, тогда весь код, содержащийся во включаемом файле, будет вести себя так, как будто он был определён внутри этой функции. То есть, он будет в той же области видимости переменных этой функции.

<?php

// Все переменные, объявленные в скрипте vars.php будут иметь ту же область видимости, что и строка на которой произошло включение

include 'vars.php';

function some\_function($foo) {

include 'some\_function\_body.php';

}

// Все классы будут иметь глобальную область видимости

include '../classes/user.php';

Оператор **include** возвращает значение **FALSE** в случае возникновения ошибки и выдаёт предупреждение. Успешные включения, пока это не переопределено во включаемом файле, возвращают значение **1**. Возможно выполнить выражение **return** внутри включаемого файла, чтобы завершить процесс выполнения в этом файле и вернуться к выполнению включающего файла. Кроме того, возможно вернуть значение из включаемых файлов.

<?php

// В зависимости от результата подключения возвращает true или false

if (include 'script.php') {

echo 'OK';

}

Выражения **include\_once** и **require\_once** включают и выполняют указанный файл во время выполнения скрипта. Их поведение идентично выражению **include** и **require**, с той лишь разницей, что если код из файла уже один раз был включён, он не будет включён и выполнен повторно и вернёт **true**. Как видно из имени, они включат файл только один раз. **include\_once** и **require\_once** могут использоваться в тех случаях, когда один и тот же файл может быть включён и выполнен более одного раза во время выполнения скрипта, в данном случае это поможет избежать проблем с переопределением функций, классов и т.д.

<?php

// Может возвращать определенное значение с помощью оператора return внутри подключаемого скрипта

$items = include 'get\_items.php';

// include\_once не выполнит повторно ранее подключенный скрипт, что позволит избежать ошибок переопределения классов, функций и др.

include\_once '../classes/user.php';

## Суперглобальные массивы

Суперглобальными массивами в PHP называются предопределенные массивы, имеющие глобальную область видимости без использования директивы **global**.

* **$GLOBALS** - массив всех глобальных переменных (в том числе и пользовательских).
* **$\_SERVER** - содержит переменные окружения, которые операционная система передает серверу.
* **$\_ENV** - текущие переменные среды. Их набор специфичен для платформы, на которой выполняется скрипт.
* **$\_GET** - содержит параметры GET-запроса, переданные в URI после знака вопроса «?».
* **$\_POST** - ассоциативный массив значений полей HTML-формы при отправке методом POST. Индексы элементов соответствуют значению свойства name объектов (кнопки, формы, радиокнопки, флажки и т. д.) HTML-формы.
* **$\_FILES** - ассоциативный массив со сведениями об отправленных методом POST файлах.
* **$\_COOKIE** - ассоциативный массив с переданными агентом пользователя значениями куки.
* **$\_REQUEST** - содержит элементы из массивов $\_GET, $\_POST, $\_COOKIE. С версии PHP 4.1 включает $\_FILES.
* **$\_SESSION** - содержит данные сессии.

# Логические операции

Логические операции позволяют комбинировать логические значения для получения новых логических значений. В языке PHP поддерживаются стандартные логические операции (and, or, not и xor).

## Логическая операция И, операторы AND и &&

Это операция, результат которой принимает истинное значение тогда и только тогда, когда оба операнда имеют истинное значение. Их единственное отличие между собой в том, что оператор **&&** имеет более высокий приоритет, чем оператор AND.

<?php

$a = true;

$b = true;

echo $a && $b; // 1

$c = false;

echo $a && $c;

echo $a && $b && $c;

## Логическая операция ИЛИ, операторы OR и ||

Операция, результат которой принимает истинное значение, если хотя бы один из операндов имеет истинное значение. Их единственное отличие между собой в том, что оператор **||** имеет более высокий приоритет, чем оператор OR.

<?php

$a = true;

$b = true;

echo $a || $b; // 1

$c = false;

echo $c || $a; // 1

echo $a || $b || $c; // 1

$d = false;

echo $c || $d;

При использовании логических операций И и ИЛИ в программе могут появиться трудноуловимые логические ошибки. Если PHP обнаружит, что первый операнд имеет значение TRUE, он не станет вычислять булево значение второго операнда. Это позволяет экономить время исполнения, но вы должны внимательно следить за тем, чтобы код, от которого зависит корректная работа программы, не был помещен во второй операнд.

<?php

function some\_function() {

echo ' in some\_function ';

return true;

}

echo some\_function(); // in some\_function

$a = true;

$b = false;

echo some\_function() && $a; // in some\_function 1

echo $a && some\_function(); // in some\_function 1

echo some\_function() && $b; // in some\_function

// функция не выполнится

echo $b && some\_function();

echo some\_function() || $b; // in some\_function

echo $b || some\_function(); // in some\_function

echo some\_function() || $a; // in some\_function 1

// функция не выполнится

echo $a || some\_function(); // 1

## Логическая операция НЕ, опертор !

Операция, результат которой принимает истинное значение, если ее единственный операнд (задаваемый справа от знака операции) имеет ложное значение, и ложное значение, если операнд имеет истинное значение. Операцию **!** обычно называют not, поскольку она становится отрицанием для того операнда, к которому применяется (инвертирует булево значение). Для простоты понимания кода, воспринимайте оператор **!** перед операндом как ожидание того, что его операнд имеет ***булево ложное*** значение. И наоборот, если перед операндом нет оператора **!** , то воспринимайте это как ожидание того, что этот операнд имеет ***булево истинное*** значение.

<?php

$a = true;

$b = true;

echo $a; // 1

echo !$a;

$c = false;

echo $c;

echo !$c; // 1

echo $a && !$b;

echo !$a && !$b;

echo $a && !$c; // 1

$d = false;

echo !$c && !$d; // 1

## Логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ, опертор xor или ^

Операция, результат которой принимает истинное значение, если только один из ее операндов (не оба одновременно) имеет истинное значение. Если оба операнда имеют одинаковые ложные/истинные значения, то результатом операции будет ложное значение.

<?php

$a = true;

$b = true;

$c = false;

$d = false;

echo $a xor $b;

echo $a xor $c; // 1

echo $c xor $d;

## Операторы сравнения

Операторы сравнения, как это видно из их названия, позволяют сравнивать между собой два значения.

| $a **==** $b | Равно | true если $a равно $b после преобразования типов |
| --- | --- | --- |
| $a **===** $b | Тождественно равно | true если $a равно $b и имеет тот же тип |
| $a **!=** $b | Не равно | true если $a не равно $b после преобразования типов |
| $a **<>** $b | Не равно | true если $a не равно $b после преобразования типов |
| $a **!==** $b | Тождественно не равно | true если $a не равно $b, или они разных типов |
| $a **<** $b | Меньше | true если $a строго меньше $b |
| $a **>** $b | Больше | true если $a строго больше $b |
| $a **<=** $b | Меньше или равно | true если $a меньше или равно $b |
| $a **>=** $b | Больше или равно | true если $a больше или равно $b |
| $a **<=>** $b | Космический корабль (spaceship) | число типа int  -1 , когда $a меньше $b,  1, когда $a больше $b  0, когда $a равно равно $b  Одинаково применимо к операндам числам и строкам |

<?php

echo 2022 == 2022.0; // 1

echo 2022 == '2022'; // 1

echo 2022 === 2022.0;

echo 2022 === '2022';

echo 2022 != 2022.0;

echo 2022 != '2022';

echo 2022 !== 2022.0; // 1

echo 2022 !== '2022'; // 1

$a = [2022, 2023, 2024];

$b = [2022, '2023', 2024];

$c = [2023, 2024];

echo $a == $b; // 1

echo $a == $c;

echo $a === $b;

В случае, если оба операнда являются строками, содержащими числа или один операнд является числом, а другой - строкой, содержащей числа, то сравнение выполняется численно. Эти правила также применяются к оператору **switch**. Преобразование типа не происходит при сравнении **===** или **!==**, поскольку это включает сравнение типа, а также значения. *До PHP 8.0.0, если строка (string) сравнивалась с числом или строкой, содержащей число, то строка (string) преобразовывалось в число перед выполнением сравнения.*

<?php

// до PHP 8.0

var\_dump(0 == 'a'); // bool(true)

var\_dump('1' == '01'); // bool(true)

var\_dump('10' == '1e1'); // bool(true)

var\_dump(100 == '1e2'); // bool(true)

switch ('a') {

case 0:

echo '0';

break;

case 'a':

echo 'a';

break;

} // 0

// начиная с PHP 8.0 результаты будут следующие

// bool(false)

// bool(true)

// bool(true)

// bool(true)

// a

# Условные выражения

В скриптах любого языка программирования условные выражения обеспечивают создание структур ветвления кода. По существу они управляют тем, будет ли выполняться часть сценария в зависимости от результата конкретного выражения (т.е. возвращает ли выражение логическое значение true или false).

## Конструкция if

Значение выражения вычисляется в булево значение. Первая строка такого выражения состоит из конструкции **if**, за которым следует выражение в круглых скобках **()**. В случае, когда требуется выполнить несколько выражений при истинном результате условного выражения, тогда блок кода с выражениями обрамляется фигурными скобками **{}**. Инструкции **if** могут быть бесконечно вложены в другие инструкции **if**, что даёт большую гибкость в организации условного выполнения различных частей программы.

<?php

$a = 2022;

$b = 2000;

$c = 1000;

if ($a > $b)

echo 'a больше b ';

echo 'этот текст выведется даже если условие не выполнено';

// инструкции if могут быть бесконечно вложены в другие инструкции if, что даёт большую гибкость в организации условного выполнения различных частей программы

if ($a > $b)

if ($b > $c)

echo 'a больше b и b больше c';

// аналогично записи

if ($a > $b && $b > $c)

echo 'a больше b и b больше c';

// часто необходимо, чтобы условно выполнялось более одной инструкции

if ($a > $b) {

echo 'a больше b';

$b = $a;

}

## Конструкция else

Часто необходимо выполнить одно выражение, если определенное условие верно, и другое выражение, если условие не верно. Оператор **else** расширяет оператор **if**, чтобы выполнить выражение, в случае, если значение условия в операторе **if** равно **false**.

<?php

$a = 2022;

$b = 2000;

if ($a > $b) {

echo 'a больше b';

} else {

echo 'a НЕ больше b';

}

## Конструкция elseif/else if

Конструкция **elseif** есть сочетание **if** и **else**. Аналогично **else**, она расширяет оператор **if** для выполнения различных выражений в случае, когда условие начального оператора **if** эквивалентно **false**. Однако, в отличие от **else**, выполнение альтернативного выражения произойдет только тогда, когда условие оператора **elseif** будет являться равным **true.**

<?php

$a = 2022;

$b = 2000;

if ($a > $b) {

echo 'a больше b';

} elseif ($a == $b) {

echo 'a равен b';

} else {

echo 'a меньше b';

}

## Конструкция switch

В случаях когда необходимо сравнивать одну и ту же переменную (или выражение) с множеством различных значений и выполнять различные участки кода в зависимости от того, какое значение принимает эта переменная, тогда вместо конструкций **if elseif else** можно использовать оператор **switch** . Следующие два примера иллюстрируют два различных способа написать то же самое.

<?php

$i = 'шоколад';

if ($i == 'яблоко') {

echo 'i это яблоко';

} elseif ($i == 'шоколад') {

echo 'i это шоколад';

} elseif ($i == 'пирог') {

echo 'i это пирог';

} else {

echo 'i это что-то';

}

switch ($a) {

case 'яблоко':

echo 'i это яблоко';

break;

case 'шоколад':

echo 'i это шоколад';

break;

case 'пирог':

echo 'i это пирог';

break;

default:

echo 'i это что-то';

}

// В операторе switch выражение вычисляется один раз и этот результат сравнивается с каждым оператором case. В выражении elseif, выражение вычисляется снова. Если ваше условие более сложное, чем простое сравнение и/или находится в цикле, конструкция switch может работать быстрее.

function my\_food() {

// do something with $food and return

$food = 'пирог';

return $food;

}

if (my\_food() == 'яблоко') {

echo 'я ем яблоко';

} elseif (my\_food() == 'шоколад') {

echo 'я ем шоколад';

} elseif (my\_food() == 'пирог') {

echo 'я ем пирог';

} else {

echo 'я ем что-то';

}

Важно понять, как оператор **switch** выполняется, чтобы избежать ошибок. Оператор **switch** исполняет строчка за строчкой (на самом деле выражение за выражением). В начале никакой код не исполняется. Только в случае нахождения оператора case, значение которого совпадает со значением выражения в операторе **switch**, PHP начинает исполнять операторы. PHP продолжает исполнять операторы до конца блока **switch** либо до тех пор, пока не встретится оператор [**break**](#_ja0o6qsglovr). Если вы не напишете оператор **break** в конце секции **case**, PHP будет продолжать исполнять команды следующей секции case.

<?php

// в этом примере, если $i равно 0, то PHP исполнит все операторы echo

$i = 0;

switch ($i) {

case 0:

echo 'i равно 0';

case 1:

echo 'i равно 1';

case 2:

echo 'i равно 2';

}

// список операторов для исполнения в секции case также может быть пустым, что просто передает управление списку операторов в следующей секции case.

switch ($i) {

case 0:

case 1:

case 2:

echo 'i меньше чем 3, но неотрицательный';

break;

case 3:

echo 'i равно 3';

}

## Тернарный оператор

Это еще один условный оператор. Тернарный оператор работает почти также как и оператор **if**, но при использовании тернарного оператора, мы вместо ключевых слов пишем **?** и **:** .

<?php

// Переменная $foo примет значение $bar1, если $condition приводится к true, иначе примет значение $bar2.

$foo = $condition ? $bar1 : $bar2;

// Приведённый выше код аналогичен следующему блоку с использованием if/else

if ($condition) {

$foo = $bar1;

} else {

$foo = $bar2;

}

Начиная с версии PHP 5.3 можно использовать тернарный оператор в таком варианте **?:**

<?php

// Изменить значения переменной, если ее значение false

$foo = $foo ? $foo : $other\_value;

// Эквивалентная запись

$foo = $foo ?: $other\_value;

Также добавился новый оператор **??**

<?php

// Изменить значения переменной, если ее значение не определено

$foo = isset($foo) ? $foo : $other\_value;

// Эквивалентная запись

$foo = $foo ?? $other\_value;

# Циклы

## Конструкция for

Цикл со счетчиком используется для выполнения тела цикла определенное число раз. Он начинает свою работу с выполнения инициализирующих команд. Данные команды выполняются только один раз. После этого проверяется условие цикла, если оно **true**, то выполняется тело цикла, иначе выполнение цикла заканчивается. После того, как будет выполнен последний оператор тела, выполняются команды после итерации. Затем снова проверяется условие цикла. Если оно **true**, выполняется тело цикла и команды после итерации, и.т.д.

<?php

// выводит числа от 1 до 10

for ($i = 1; $i <= 10; ++$i) {

echo $i;

}

Каждое из 3 выражений, разделенных знаком точка с запятой **;** внутри круглых скобок конструкции **for** может быть пустым или содержать несколько выражений, разделенных запятыми **,** . Если условие цикла отсутствует, то это означает, что цикл будет выполняться бесконечно.

<?php

// выводит числа от 1 до 10 не в теле цикла, а в выражении после итерации

for ($i = 1; $i <= 10; print\_r($i), ++$i);

// выводит числа от 1 до 10 без выражения условия выполнения. Для выхода из бесконечного цикла используется оператор break

for ($i = 1, $j = 0; ; ++$i) {

echo $i;

if ($i >= 10) {

break;

}

}

Поскольку условие выполнения цикла проверяется перед каждой итерацией, *то в выражении условия цикла не должно быть вычислений, результат которых не меняется после после каждой итерации*. Такие вычисления должны быть выполнены в выражении инициализации цикла или ранее и сохранены в промежуточную переменную.

<?php

$id\_list = [

24,

345,

// ...

];

for ($i = 0, $size = count($id\_list); $i < $size; ++$i) {

echo $id\_list[$i];

}

## Конструкция foreach

Предоставляет удобный и простой способ перебора массивов, как для индексированных, так и для ассоциативных. Выражения в теле цикла выполняются для каждого элемента массива. При этом отдельно задаются значения пары переменных: c ключем и со значением элемента.

<?php

$id\_list = [

24,

345,

// ...

];

// выводятся только первые 2 элемента массива

foreach ($id\_list as $i => $id) {

echo $id;

if ($i > 2) {

break;

}

}

$products\_prices = [

'priority' => 20000,

'allcorp' => 30000,

'market' => 40000,

];

foreach ($products\_prices as $name => $price) {

echo "{$name} => {$price} <br />";

}

Если ключ элемента не интересует, то синтаксис допускает написание конструкции без переменной для значения ключа.

<?php

$id\_list = [

24,

345,

// ...

];

foreach ($id\_list as $id) {

echo $id;

}

Изменение значения элемента внутри тела цикла возможно при обращении напрямую через ключ элемента массива, либо по ссылке на значение элемента массива. *В последнем случае после выражения цикла ссылку на значение элемента массива следует разорвать конструкцией* ***unset*** *.*

<?php

$products\_prices = [

'priority' => 20000,

'allcorp' => 30000,

'market' => 40000,

];

// наценка 10% на товары

foreach ($products\_prices as $name => $price) {

$products\_prices[$name] \*= 1.1;

}

$products\_stickers = [

'priority' => 'hot',

'allcorp' => 'new',

'market' => 'sale',

];

// сброс стикеров товаров

foreach ($products\_stickers as $name => &$sticker) {

$sticker = '';

}

// разорвать ссылку на значение последнего элемента

unset($sticker);

// если ссылку не разорвать, то следующее выражение изменило бы значение последнего элемента массива $products\_stickers

$sticker = 'sale';

## Оператор break

Прерывает выполнение текущей структуры [**for**](#_t4ffl5lhurgb), [**foreach**](#_iqzwghmhup9q), [**while**](#_qy3fuyj5vcao), [**do-while**](#_zbsxuz2u8cdv) и [**switch**](#_9rzhd7lte654). Принимает необязательный числовой аргумент, который сообщает ему выполнение какого количества вложенных структур необходимо прервать. Значение по умолчанию **1**, только ближайшая структура будет прервана.

<?php

// вывод не полной таблицы Пифагора

for ($i = 1; $i <= 9; ++$i) {

for ($j = 1; $j <= 9; ++$j) {

if ($j > 7) {

// второй множитель не более 7

break;

}

$m = $i \* $j;

if ($m > 50) {

// выход из обоих циклов for

break 2;

}

echo "$i x $j = $m <br >";

}

}

## Оператор continue

Используется для пропуска оставшейся части текущей итерации цикла и, при соблюдении условий выполнения цикла, начала следующей итерации. Принимает необязательный числовой аргумент, который указывает на скольких уровнях вложенных циклов будет пропущена оставшаяся часть итерации. Значением по умолчанию является **1**, при которой пропускается оставшаяся часть текущего цикла.

<?php

// вывод не полной таблицы Пифагора

for ($i = 1; $i <= 9; $i += 2) {

for ($j = 1; $j <= 9; ++$j) {

if (!($j % 2)) {

// получение остатка от целочисленного деления на 2 позволяет проверить на четность. Четные числа пропускаем

continue;

}

$m = $i \* $j;

if ($m > 30) {

// переходим к следующей итерации первого цикла

continue 2;

}

echo "$i x $j = $m <br >";

}

}

## Конструкция while

Цикл с предусловием позволяет выполнять вложенные выражения повторно до тех пор, пока вычисленное значение выражение условия цикла является **true**. Значение выражения условия цикла проверяется каждый раз перед началом цикла. Если значение выражения условия цикла равно **false** с самого начала, то тело цикла ни разу не будет выполнено.

<?php

$i = 1;

// выводит числа от 1 до 10

while ($i < 10) {

echo $i;

++$i;

}

while (false) {

echo 'Никогда не будет выведено';

}

## Конструкция do while

Цикл с постусловием проверяет значение выражения условия цикла не до, а после каждого прохода (итерации). Таким образом, тело цикла выполняется хотя бы один раз.

<?php

$i = 1;

// выводит числа от 1 до 10

do {

echo $i;

++$i;

} while ($i < 10);

# Альтернативный синтаксис управляющих структур

Для управляющих структур [**if**](#_5fxoswn5ldf), [**else**](#_e6yl3tpukv43), [**elseif/else if**](#_9wj6o94u6156), [**while**](#_qy3fuyj5vcao), [**for**](#_t4ffl5lhurgb), [**foreach**](#_iqzwghmhup9q) и [**switch**](#_9rzhd7lte654) допускается альтернативный синтаксис. В каждом случае основной формой альтернативного синтаксиса является изменение открывающей фигурной скобки на двоеточие **:** , а закрывающей скобки на **endif;** , **endwhile;** , **endfor;** , **endforeach;** или **endswitch;** соответственно. *Смешивание разного синтаксиса в одном и том же блоке управления не поддерживается.*

<?php

if ($i == 1):

$j = $i;

echo $j;

endif;

if ($i % 2):

echo "$i нечетное <br />";

else:

echo "$i четное <br />";

endif;

if (!$i):

echo "$i пустое или ноль <br />";

elseif ($i % 2):

echo "$i нечетное <br />";

else:

echo "$i четное <br />";

endif;

$items = [];

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

$items[$i] = $i;

endfor;

foreach ($items as $item):

echo $item;

endforeach;

$i = 1;

while ($i <= 10):

echo "$i <br />";

++$i;

endwhile;

?>

Альтернативный синтаксис (кроме конструкции **switch**) чаще всего употребляется при вставке PHP кода в страницу HTML-документ.

<?php

$products = [

'priority' => 20000,

'allcorp' => 30000,

'market' => 40000,

];

?>

<div class="products">

<?php if ($products):?>

<?php foreach ($products as $name => $price):?>

<div class="product">

<span class="product\_\_name"><b><?=$name?></b></span>

<span class="product\_\_name"><?=$price?></span>

</div>

<?php endforeach;?>

<?php else:?>

<div class="alert alert--warning">

<span>Нет товаров</span>

</div>

<?php endif;?>

</div>

# Константы

Константа - это имя для простого значения. Как следует из названия, их значение не может изменится в ходе выполнения скрипта. Константы чувствительны к регистру. По принятому соглашению, имена констант всегда пишутся в верхнем регистре. Имена констант следуют тем же правилам, что и имена переменных. Корректное имя константы начинается с буквы или знака подчеркивания, за которым следует любое количество букв, цифр или знаков подчеркивания.

Константы доступны из любой области видимости. Константа может быть определена с помощью ключевого слова **const** или с помощью функции **define**() . Получить значение константы можно, указав её имя, или через функцию **constant**().

<?php

// Константа может быть определена с помощью ключевого слова const или с помощью функции define()

const API\_KEY = 'd#s;f768sd\*f76b';

// Получить значение константы можно, указав её имя

echo API\_KEY;

// Или через функцию constant()

echo constant('API\_KEY');

// Значение константы - простые значения скалярного типа (bool, integer, float, string) и массивы

define('DEBUG', false);

define('MAX\_DEPTH\_LEVEL', 3);

define('MAX\_SCORE', 100.0);

define('INCLUDE\_PATH', '/app/include/');

define('ANIMALS', [

'dog',

'cat',

'bird',

]);

Константы, определенные через **const** определяются при компилировании скрипта, поэтому такие константы не могут быть определены внутри условия/функции и т.п., не могут иметь вычисляемое значение.

<?php

if (true) {

// Здесь произойдет parse ошибка при запуске скрипта

// const PRIVATE\_KEY = 'qvbr4$\*olkm%no';

// Здесь тоже ошибка

// const MIN\_SCORE = floatval(0);

// В отличие от const нижеследующий код сработает

define('PRIVATE\_KEY', 'qvbr4$\*olkm%no');

define('MIN\_SCORE', floatval(0));

echo PRIVATE\_KEY;

echo MIN\_SCORE;

}

До PHP 8 значение неопределенных констант интерпретировалось как строка, равная названию константы. Начиная с PHP 8 выбрасывается ошибка. Перед использованием константы нужно убедиться, что константа была определена. Это можно с делать с помощью функции **defined**() .

<?php

// Вывод значения неопределенной константы

var\_dump(UNDEFINED\_CONSTANT); // PHP <8: warning & string(18) "UNDEFINED\_CONSTANT",

var\_dump(SECOND\_UNDEFINED\_CONSTANT); // PHP 8+: fatal error

// Перед использованием константы нужно убедиться, что константа была определена. Это можно сделать с помощью функции defined()

if (

defined('API\_KEY') &&

defined('REST\_URL')

) {

$url = REST\_URL.'?api\_key='.API\_KEY;

}

## Магические константы

Есть девять магических констант, которые меняют своё значение в зависимости от контекста, в котором они используются. Например, значение **\_\_LINE\_\_** зависит от строки в скрипте, на которой эта константа указана.

| **\_\_LINE\_\_** | Текущий номер строки в файле. |
| --- | --- |
| **\_\_FILE\_\_** | Полный путь и имя текущего файла с развернутыми симлинками. Если используется внутри подключаемого файла, то возвращается имя данного файла. |
| **\_\_DIR\_\_** | Директория файла. Если используется внутри подключаемого файла, то возвращается директория этого файла. Это эквивалентно вызову dirname(\_\_FILE\_\_). Возвращаемое имя директории не оканчивается на слеш, за исключением корневой директории. |
| **\_\_FUNCTION\_\_** | Имя функции или {closure} в случае анонимной функции. |
| **\_\_CLASS\_\_** | Имя класса. Это имя содержит название пространства имён, в котором класс был объявлен (например, Foo\Bar). При использовании в методах трейтов \_\_CLASS\_\_ является именем класса, в котором эти методы используется. |
| **\_\_TRAIT\_\_** | Имя трейта. Это имя содержит название пространства имён, в котором трейт был объявлен (например, Foo\Bar). |
| **\_\_METHOD\_\_** | Имя метода класса. |
| **\_\_NAMESPACE\_\_** | Имя текущего пространства имён. |
| **ClassName::class** | Полное имя класса. |

# Функции

Функция — это блок кода, который может быть именован и вызван повторно. Имена функций следуют тем же правилам, что и имена переменных. Корректное имя функции начинается с буквы или знака подчеркивания, за которым следует любое количество букв, цифр или знаков подчеркивания. Имена функций регистронезависимы.

Функции не обязаны быть определены до их использования. Функции могут быть определены по условию, тогда в этом случае их обработка описания должна предшествовать из вызову.

Внутри функции может быть определена другая функция или даже класс. Но все они получат глобальную область видимости.

Вызов функции можно произвести через обращение к переменной, значением которой будет строка с названием вызываемой функции.

Проверить, определена ли некая функция, можно с помощью **function\_exists**(), передав в нее название проверяемой функции.

## Возвращаемое значение функции

Значения возвращаются при помощи необязательного оператора возврата **return**. Если конструкция **return** не указана, то функция вернет значение **null**. В объявлении функции можно указать тип возвращаемого значения.

## Аргументы функции

Аргументы вычисляются слева направо перед фактическим вызовом функции. По умолчанию аргументы в функцию передаются по значению (это означает, что если вы измените значение аргумента внутри функции, то вне ее значение все равно останется прежним). Если вы хотите разрешить функции модифицировать свои аргументы, вы должны передавать их по ссылке. Чтобы аргумент всегда передавался по ссылке, нужно указать амперсанд **&** перед именем аргумента в описании функции.

Для аргументов допустимо добавить объявление типа значения. В случае, если переданное значение не будет того же типа, как указано в объявлении функции, то возникнет ошибка выполнения.

В функцию можно передать больше аргументов, чем указано в описании функции. Все переданные значения аргументов внутри функции доступны через методы **func\_get\_args**() , **func\_get\_arg**() . Функции с переменным количеством аргументов также можно реализовывать с использованием оператора многоточие **...** . Аргументы в этом случае будут переданы в виде массива. Многоточие также можно использовать при вызове функции, чтобы наоборот *распаковать* массив в список аргументов.

<?php

// Функция принимает не менее 2 аргументов.

// Все аргументы, начиная со второго будут доступны внутри функции как элементы массива $strings

function concatenate($transform, ...$strings) {

$string = '';

foreach ($strings as $piece) {

$string .= $piece;

}

return ($transform($string));

}

echo concatenate('strtoupper', 'I\'d ', 'like ', 4 + 2, ' apples');

// Распаковка массива как аргументов функции на примере

// отправки электронного письма через

// функцию mail('address@example.com', 'Title', 'Text');

$mail\_params = [

'someone@example.com',

'Hi there',

'Thanks for registering, hope you like it',

];

mail(...$mail\_params);

## Значения аргументов функции по умолчанию

Для аргументов скалярного типа, для массивов, и для типа null, в описании функции можно указать значение по умолчанию. Значение по умолчанию должно быть константным, а не вычисляемым выражением. Это значение должно соответствовать указанному типу аргумента, если этот тип указан в объявлении функции. *Т.к. в функцию обязательно должны быть переданы аргументы, для которых значения по умолчанию не заданы, то все аргументы, для которых установлены значения по умолчанию, должны находиться после обязательных аргументов.*

## Именованные аргументы функции

В PHP 8+ при вызове функции можно не следовать заданному объявлением функции порядку следования в ней аргументов. Этого можно добиться с помощью именованных аргументов, которые передаются путем добавления через двоеточия имени параметра перед его значением. Имя параметра должно быть идентификатором, а не вычисляемым выражением.

<?php

// При вызове функции можно не следовать заданному объявлением функции порядку следования в ней аргументов. Этого можно добиться с помощью именованных аргументов

function some\_function($param1, $param2) {

// do something

}

some\_function(param2: $value2, param1: $value1);

// Здесь будет ошибка.

// Имя параметра должно быть идентификатором, а не вычисляемым выражением

some\_function($param\_1\_name: $value1, $param\_2\_name: $value2);

## Анонимные функции

Также известны как **замыкания**. Функции не имеющие определенных имен могут быть использованы в качестве значений переменных. Замыкания могут также наследовать переменные из родительской области видимости. Любая подобная переменная должна быть объявлена в конструкции **use** . *Наследование переменной из родительской области видимости происходит по значению.*

## Статические переменные функций

При выходе из тела функции все ранее объявленные в ней переменные исчезают. Чтобы это не происходило, переменную внутри функции объявляют через ключевое слово **static** . Статическая переменная существует только в локальной области видимости функции, но не теряет своего значения, когда выполнение программы выходит из этой области видимости. Значение статической переменной можно использовать при следующем вызове функции - она по прежнему будет иметь то же значение, которое получила при последнем вызове функции.

<?php

// Рекурсивная функция получения структуры раздела в виде дерева.

// Статическая переменная $depth\_level инициализируется 1 раз со значением 0 и используется для подсчета текущего уровня вложенности вызова функции.

function get\_dir\_tree($dir) {

static $depth\_level = 0;

$result = [];

// увеличиваем на единицу

++$depth\_level;

// не будем углубляться в подразделы, если текущий уровень вложенности более 3

if (

$depth\_level <= 3 &&

is\_dir($dir)

) {

$filenames = @scandir($dir);

if ($filenames) {

foreach($filenames as $filename) {

if (

$filename == '.' ||

$filename == '..'

) {

continue;

}

$file = $dir.DIRECTORY\_SEPARATOR.$filename;

$is\_dir = is\_dir($file);

$result[] = [

'file' => $file,

'depth\_level' => $depth\_level,

'is\_dir' => $is\_dir,

];

if ($is\_dir) {

$result = array\_merge($result, get\_dir\_tree($file));

}

}

}

}

// уменьшаем на единицу

--$depth\_level;

return $result;

}

$tree = get\_dir\_tree(\_\_DIR\_\_);

# Массивы

Массив - это список элементов, которые располагаются в определенном порядке. Каждый элемент имеет определенный индекс (ключ), даже если это явно не указано в определении значения элемента массива. Ключами массива могут быть числа или строки. Если все ключи массива числа (или строки, содержащие целое число, которое может быть приведено к типу **integer**), то такой массив называется **индексным**. Если все ключи массива строки, то такой массив называется **ассоциативным**. **Смешанный** массив содержит ключи типа **integer** и **string** одновременно.

<?php

// В ассоциативном массиве ключами элементов являются строки

$array = [

'id' => 100, // ключ id

'name' => 'Optimus', // ключ name

];

// Добавление нового элемента с ключом price в конец массива

$array['price'] = 30000;

// В индексном массиве ключами элементов являются числа

// Для определения массива можно использовать ключевое слово array() вместо квадратных скобок []

$array = array(

'foo', // индекс 0

'bar', // индекс 1

5 => 'some', // указан индекс 5

3 => 'another', // указан индекс 3

);

// Добавление нового элемента в конец массива

$array[] = 'beautiful';

// Добавление нового элемента в конец массива с заданным индексом

$array[7] = 'things';

Если ключ элемента не был указан, то для его вычисления будет взят максимальный из существующих целочисленных индексов, и новым ключом будет это максимальное значение (в крайнем случае 0) плюс 1. Если целочисленных индексов ещё нет, то ключом будет 0.

<?php

// Приведение значений ключей к целому числу

$array[5.90] = 'one'; // перезапишет элемент с индексом 5

$array['7'] = 'thing'; // перезапишет элемент с индексом 7

$array[false] = 'it'; // перезапишет элемент с индексом 0

$array[true] = 'is'; // перезапишет элемент с индексом 1

$array[null] = 'here'; // добавит элемент с индексом пустая строка ''

Так как значением массива может быть другой массив PHP, можно также создавать деревья и многомерные массивы.

<?php

// Простой многомерный массив

// Массив товаров. Значением каждого элемента массива является отдельный товар - тоже массив

$products = [

[

'id' => 100,

'name' => 'Optimus',

'price' => 30000,

],

[

'id' => 200,

'name' => 'Maximum',

'price' => 60000,

],

];

// Список пунктов меню сайта по продаже автомобильных шин и дисков

$menu = [

[

'name' => 'Главная страница',

'link' => '/',

],

[

'name' => 'Каталог',

'link' => '/catalog/',

'childs' => [

[

'name' => 'Шины',

'link' => '/catalog/tires/',

'childs' => [

[

'name' => 'Nokian',

'link' => '/catalog/tires/nokian/',

],

[

'name' => 'Yokohama',

'link' => '/catalog/tires/yokohama/',

],

],

],

[

'name' => 'Диски',

'link' => '/catalog/wheels/',

'childs' => [

[

'name' => 'Replica',

'link' => '/catalog/wheels/replica/',

],

[

'name' => 'Alutec',

'link' => '/catalog/wheels/alutec/',

],

],

],

],

],

[

'name' => 'Контакты',

'link' => '/contacts/',

],

];

Для удаления элемента из массива необходимо использовать функцию [unset()](https://www.php.net/manual/ru/function.unset.php). При этом элементы массива не будут заново проиндексированы.

Обращение к элементу массива по несуществующему ключу вернет значение **null** и вызовет ошибку уровня Notice.

<?php

$names = [

'Саша',

'foo' => 'Таня',

'bar' => 'Маша',

'Вася',

];

// Удаление элементов из массива

unset($names[0], $names['bar'], $names[1]);

// Обращение к несуществующему элементу

var\_dump($names['another']); // null

// Обращение к переменной другого типа как к массиву

$foo = 100;

var\_dump($foo['bar']); // null

$foo = 'string value';

var\_dump($foo[0]); // s

var\_dump($foo[11]); // e

## Функции для работы с массивами

Кратко опишем только некоторые часто используемые функции. [Полный список в документации](https://www.php.net/manual/ru/ref.array.php).

**array\_column**() - возвращает массив из значений одного столбца входного массива

**is\_array**() - провеяет, является ли переменная массивом.

**in\_array**() - провеяет, входит ли элемент в массив.

**count**() - подсчитывает количество элементов массива.

**implode**() - объединяет элементы массива в строку с разделителем между ними.

**explode**() - разбивает строку с помощью разделителя на элементы массива.

**array\_push**() - добавляет один или несколько элементов в конец массива.

**array\_pop**() - извлекает последний элемент массива.

**array\_search**() - осуществляет поиск данного значения в массиве и возвращает ключ первого найденного элемента в случае успешного выполнения.

**array\_key\_exists**() - проверяет, присутствует ли в массиве указанный ключ или индекс.

**array\_shift**() - извлекает первый элемент массива.

**array\_unshift**() - добавляет один или несколько элементов в начало массива.

**array\_count\_values**() - подсчитывает количество всех значений массива.

**array\_unique**() - убирает повторяющиеся значения из массива.

**array\_reverse**() - возвращает массив с элементами в обратном порядке.

**array\_diff**() - вычисляет расхождение массивов.

**array\_values**() - выбирает все значения массива. Она также заново индексирует возвращаемый массив числовыми индексами.

**array\_keys**() - возвращает ключи, содержащиеся в массиве, или некоторое подмножество ключей массива.

**array\_sum**() - вычисляет сумму значений массива.

**array\_merge**() - сливает один или большее количество массивов. Если входные массивы имеют одинаковые строковые ключи, тогда каждое последующее значение будет заменять предыдущее. Однако, если массивы имеют одинаковые числовые ключи, значение, упомянутое последним, не заменит исходное значение, а будет добавлено в конец массива. В результирующем массиве значения исходного массива с числовыми ключами будут перенумерованы в возрастающем порядке, начиная с нуля.

**array\_filter**() - фильтрует элементы массива с помощью callback-функции.

**array\_map**() - применяет callback-функцию ко всем элементам указанных массивов.

**min**() и **max**() - находит наименьшее и наибольшее значение в значениях переданных аргументов.

**sort**() и **rsort**() - сортирует элементы массива по возрастанию и убыванию значения.

**usort**() и **uasort**() - сортирует элементы массива в порядке, определяемой пользовательской функцией с переиндексацией элементов и без.

<?php

$users = [

[

'name' => 'Саша',

'age' => 8,

],

[

'name' => 'Таня',

'age' => 6,

],

[

'name' => 'Маша',

'age' => 12,

],

[

'name' => 'Вася',

'age' => 9,

],

];

// array\_column() - возвращает массив из значений одного столбца входного массива

$names = array\_column($users, 'name');

// is\_array() - провеяет, является ли переменная массивом

var\_dump(is\_array($names)); // true

// in\_array() - провеяет, входит ли элемент в массив

var\_dump(in\_array('Витя', $names)); // false

// count() - подсчитывает количество элементов массива

echo count($names);

// implode() - объединяет элементы массива в строку с разделителем между ними

$all\_names = implode(', ', $names);

echo $all\_names;

// explode() - разбивает строку с помощью разделителя на элементы массива

$names = explode(', ', $all\_names);

// array\_push() - добавляет один или несколько элементов в конец массива

array\_push($names, 'Петя', 'Катя');

// array\_pop() - извлекает последний элемент массива

$last\_name = array\_pop($names);

// array\_search() - осуществляет поиск данного значения в массиве и возвращает ключ первого найденного элемента в случае успешного выполнения

var\_dump(array\_search('Саша', $names)); // 0

var\_dump(array\_search('Катя', $names)); // false

// array\_key\_exists() - проверяет, присутствует ли в массиве указанный ключ или индекс

var\_dump(array\_key\_exists(5, $names)); // false

// array\_shift() - извлекает первый элемент массива

$first\_name = array\_shift($names);

// array\_unshift() - добавляет один или несколько элементов в начало массива

array\_unshift($names, 'Петя', 'Катя');

// array\_count\_values() - подсчитывает количество всех значений массива

$names\_qnt = array\_count\_values($names);

// array\_unique() - убирает повторяющиеся значения из массива

$names = array\_unique($names);

// array\_reverse() - возвращает массив с элементами в обратном порядке

$names = array\_reverse($names);

// array\_diff() - вычисляет расхождение массивов

$names = array\_diff($names, ['Маша', 'Лера', 'Даша'], ['Вася', 'Петя']);

// array\_values() - выбирает все значения массива. Она также заново индексирует возвращаемый массив числовыми индексами.

$names = array\_values($names);

// array\_keys() - возвращает ключи, содержащиеся в массиве, или некоторое подмножество ключей массива

$all\_keys = array\_keys($names);

$some\_keys = array\_keys($names, 'Катя');

// array\_sum() - вычисляет сумму значений массива

$sum = array\_sum($all\_keys);

// array\_merge() - сливает один или большее количество массивов. Если входные массивы имеют одинаковые строковые ключи, тогда каждое последующее значение будет заменять предыдущее. Однако, если массивы имеют одинаковые числовые ключи, значение, упомянутое последним, не заменит исходное значение, а будет добавлено в конец массива. В результирующем массиве значения исходного массива с числовыми ключами будут перенумерованы в возрастающем порядке, начиная с нуля.

$some\_boys\_names = ['Данил', 'Миша', 'Кирилл'];

$some\_girls\_names = ['Света', 'Лена', 'Вика'];

$all\_names = array\_merge($names, $some\_boys\_names, $some\_girls\_names);

// array\_filter() - фильтрует элементы массива с помощью callback-функции

$names = array\_filter($all\_names, function($name) use ($some\_boys\_names) {

return !in\_array($name, $some\_boys\_names);

});

// array\_map() - применяет callback-функцию ко всем элементам указанных массивов

$names = array\_map(function($name){

return 'это '.$name;

}, $some\_boys\_names);

$names = array\_map(function($boy\_name, $girl\_name){

return $boy\_name.' и '.$girl\_name;

}, $some\_boys\_names, $some\_girls\_names);

// min() и max() - находит наименьшее и наибольшее значение в значениях переданных аргументов

$ages = array\_column($users, 'age');

$min\_age = min($ages);

$max\_age = max($ages);

// sort() и rsort() - сортирует элементы массива по возрастанию и убыванию значения

sort($ages);

// usort() и uasort() - сортирует элементы массива в порядке, определяемой пользовательской функцией с переиндексацией элементов и без

usort($users, function($user1, $user2) {

return $user1['age'] <=> $user2['age'];

});

## Деструктуризация массива

Деструктуризация массива - это способ извлечения значений из массива и записывать их в новые переменные с минимумом кода. Для присвоения значения переменной при деструктуризации используется конструкция **list**() .

<?php

$user = [1, 'Вася', 'Пупкин'];

// Без деструктуризации

$id = $user[0];

$first\_name = $user[1];

$last\_name = $user[2];

// Деструктуризация массива - это способ извлечения значений из массива и записывать их в новые переменные с минимумом кода

list($id, $first\_name, $last\_name) = $user;

Можно использовать короткий синтаксис **[]** для деструктуризации массивов с целью присвоения. Можно присвоить значение меньшему количеству переменных, чем значений в массиве. И даже пропустить некоторые значения.

// Можно использовать короткий синтаксис [] для деструктуризации массивов с целью присвоения

[$id, $first\_name, $last\_name] = $user;

// Можно присвоить значение меньшему количеству переменных, чем значений в массиве

[$id, $first\_name] = $user;

// И даже пропустить некоторые значения

[$id, , $last\_name] = $user;

Деструктуризация может быть вложенной для присвоения значений из многомерного массива.

<?php

// Деструктуризация может быть вложенной для присвоения значений из многомерного массива

$users = [

[1, 'Вася', 'Пупкин'],

[2, 'Иван', 'Петров'],

];

list($first\_user, list($second\_user\_id, $second\_user\_first\_name)) = $users;

Для переменной можно указать ключ значения в деструктурируемом массиве. Это позволяет удобно деструктурировать ассоциативный массив или присвоить переменным только некоторые значения массива.

<?php

// Для переменной можно указать ключ значения в деструктурируемом массиве. Это позволяет удобно деструктурировать ассоциативный массив или присвоить переменным только некоторые значения массива

$user = [

'id' => 1,

'first\_name' => 'Вася',

'last\_name' => 'Пупкин',

'login' => 'vasya@mail.ru',

];

// В $id и $login будет null, т.к. в данном ассоциативном массиве нет таких ключей

[$id, , , $login] = $user;

// Все получится, если указать ключи присваиваемых значений

['login' => $login, 'id' => $id] = $user;

Использовать присвоение переменной значения из массива деструктуризацией можно прямо в конструкции foreach.

<?php

$users = [

[1, 'Вася', 'Пупкин', 'vasya@mail.ru'],

[2, 'Иван', 'Петров', 'ivan@mail.ru'],

];

// Без деструктуризации

foreach ($users as $user) {

echo $user[0].' '.$user[3].'<br />';

}

// Деструктуризация внутри foreach уменьшает расход памяти

foreach ($users as [$id, , , $login]) {

echo $id.' '.$login.'<br />';

}

// Аналогично деструктурируем с указанием ключей присваиваемых значений

foreach ($users as [0 => $id, 3 => $login]) {

echo $id.' '.$login.'<br />';

}

# Функции для работы со строками

Кратко опишем только некоторые часто используемые функции. [Полный список в документации](https://www.php.net/manual/ru/ref.strings.php).

<?php

// Конкатенация

$a = 'Привет';

$а = $a.', Мир!';

// Приравнивание с конкатенацией

$a = 'Привет';

$a .= ', Мир!';

// trim() - удаляет пробелы (или другие символы) из начала и конца строки

echo trim(' word '); // word

echo trim('/word/', '/'); // word

echo trim('\nword\n', '/'); // word

// ltrim(), rtrim() - удаляет пробелы (или другие символы) из начала/конца строки

echo ltrim('/word/', '/'); // word/

echo rtrim('/word/', '/'); // /word

// strlen() - возвращает длину строки

echo strlen(' word '); // 6

// str\_replace() - замена подстроки в строке

$path = '/catalog//tires//nokian/';

echo $path = str\_replace('//', '/', $path); // /catalog/tires/nokian/

echo str\_replace('//', '/', '/catalog///tires//nokian/'); // /catalog//tires/nokian/

// strpos() - возвращает позицию первого вхождения подстроки

echo strpos($path, '/tires/'); // 9

// substr() - возвращает подстроку заданной длины от заданной позиции в строке

echo substr('word', 0, 2); // wo

echo substr('word', 2); // rd

echo substr('word', -2); // rd

echo substr('word', -4, 1); // w

// strtolower(), strtoupper() - преобразует строку в нижний/верхний регистр

echo strtolower('HELLO WORLD!'); // hello world!

echo strtolower('Hello world!'); // HELLO WORLD!

// strip\_tags() - удаляет из строки все теги HTML и PHP кроме заданных

echo $text = '<p>Текст в параграфе.</p> <a href="#fragment">Cсылка на документ.</a> <br/><div>Текст после перевода строки</div>';

// echo strip\_tags($text);

echo strip\_tags($text, '<a><br>'); // разрешаем ссылки и переводы строк

// md5() - вычисляет MD5-хеш строки

echo md5('qwerty'); // d8578edf8458ce06fbc5bb76a58c5ca4

// htmlspecialchars() - преобразует специальные символы <>& в HTML-сущности

$content = '<a href="http://www.php.net">Руководство по PHP</a>';

echo $content; // текст отображается как HTML

echo htmlspecialchars($content);

# Функции для работы с файлами

Кратко опишем только некоторые часто используемые функции. [Полный список в документации](https://www.php.net/manual/ru/ref.filesystem.php).

<?php

$dir = \_\_DIR\_\_;

echo 'Путь к папке этого скрипта: '.$dir.'<br />';

// basename() - возвращает последний компонент имени из указанного пути

echo 'Название папки: '.basename($dir).'<br />';

$file = \_\_FILE\_\_;

echo 'Путь к этому скрипту: '.$file.'<br />';

// basename() - возвращает последний компонент имени из указанного пути

echo 'Название файла этого скрипта: '.basename($file).'<br />';

// dirname() - возвращает родительский каталог из указанного пути на заданное количество уровней вверх

echo dirname($file).'<br />';

echo dirname($file, 2).'<br />';

// scandir() - получает список файлов и каталогов, расположенных по указанному пути

$files = scandir($dir);

// is\_dir() - определяет, является ли имя файла директорией

if (is\_dir('level\_1')) {

echo 'level\_1 - это папка <br />';

}

// is\_file() - определяет, является ли файл обычным файлом

if (is\_file('2-xss.php')) {

echo '2-xss.php - это файл <br />';

}

// file\_exists() - проверяет наличие файла/папки по указанному пути

if (!file\_exists('test.php')) {

echo 'test.php не существует <br />';

}

// получим файлы текущей директории через opendir()

$files = [];

if (is\_dir($dir)) {

$handle = opendir($dir);

while(($file = readdir($handle)) !== false) {

$files[] = $file;

}

closedir($handle);

}

// получим файлы текущей директории через glob()

$files = glob($dir.'/\*');

// только .php файлы

$files = glob($dir.'/\*.php');

// включая скрытые

$files = glob($dir.'/{,.}\*', GLOB\_BRACE);

// mkdir() - создание директории с указанными правами доступа

mkdir('child\_dir');

// rmdir() - удаление пустой папки

rmdir('child\_dir');

// рекурсивное создание вложенного подраздела

mkdir('child\_dir/subchild\_dir', 0777, true);

// file\_put\_contents() - запись строки в файл

$products = [

[

'id' => 100,

'name' => 'Optimus',

'price' => 30000,

],

[

'id' => 200,

'name' => 'Maximum',

'price' => 60000,

],

];

$file = 'child\_dir/subchild\_dir/products.csv';

$seperator = ';';

file\_put\_contents($file, implode($seperator, ['id', 'name','price']).PHP\_EOL);

foreach ($products as $item) {

file\_put\_contents($file, implode($seperator, $item).PHP\_EOL, FILE\_APPEND);

}

// file\_get\_contents() - возвращает содержимое файла в виде строки

$content = file\_get\_contents($file);

// чтение удаленного ресурса

$content = file\_get\_contents('https://ya.ru');

// чтение файла блоками

$content = '';

if ($handle = fopen($file, 'r')) {

while(!feof($handle)) {

$buffer = fread($handle, 4096);

$content .= $buffer;

echo $buffer.'<br />';

}

fclose($handle);

}

// чтение файла построчно

$content = '';

if ($handle = fopen($file, 'r')) {

while(!feof($handle)) {

$buffer = fgets($handle);

$content .= $buffer;

echo $buffer.'<br />';

}

fclose($handle);

}

// запись в файл

$lines = [];

if ($handle = fopen($file, 'c')) {

// пишем в начало файла

$writed = fwrite($handle, 'BEGIN'.PHP\_EOL);

echo 'Записано: '.$writed.'<br/>';

// перемещаем указатель в конец файла

fseek($handle, 0, SEEK\_END);

// пишем в конец файла

$writed = fwrite($handle, 'END'.PHP\_EOL);

echo 'Записано: '.$writed.'<br/>';

fclose($handle);

}

// unlink() - удаление файла

unlink($file);

# Регулярные выражения

Регулярные выражения (regex или regexp) очень эффективны для извлечения информации из текста. Для этого нужно произвести поиск одного или нескольких совпадений по определённому шаблону (т. е. определённой последовательности символов). Области применения regex разнообразны, от валидации до парсинга/замены строк и др. Например, можно распарсить log-файл, номер телефона, дату, время и т.д.

Одна из любопытных особенностей регулярных выражений в их универсальности, стоит вам выучить синтаксис, и вы сможете применять их в любом (почти) языке программирования.

## Тестирование работы регулярных выражений

Сервис <https://regex101.com/> позволяет набирать и тестировать регулярные выражения онлайн на предмет нахождения искомой шаблонной подстроки в тексте. Также позволяет протестировать результат замены в тексте шаблонной подстроки на другую строку.

## Строение регулярного выражения

Любое регулярное выражение имеет строение

**<разделитель>**<шаблон строки>**<разделитель>***<опции>*

Шаблон строки в свою очередь может состоять из

<якорь><набор символов><квантификатор><жадность><якорь>

## Разделители в регулярном выражении

Шаблон строки заключен в разделители. Разделителем может быть любой символ не являющийся буквой, цифрой, обратной косой чертой или каким-либо пробельным символом. Часто используемыми разделителями являются косые черты **/**, знаки решётки **#** и тильды **~**. Ниже представлены примеры шаблонов с корректными разделителями.

/foo bar/

#^[^0-9]$#

+php+

%[a-zA-Z0-9\_-]%

## Метасимволы в шаблоне регулярных выражений

Сила регулярных выражений исходит из возможности использовать условия и повторения в шаблоне. Они записываются при помощи метасимволов, которые специальным образом интерпретируются.

## Якоря **^** и **$** в регулярных выражениях

*^Предложение*

соответствует строке, начинающейся с *Предложение*

*слов$*

соответствует строке, заканчивающейся на *слов*

*^Предложение из нескольких слов$*

точное совпадение (начинается и заканчивается как *Предложение из нескольких слов*)

*из*

соответствует любой строке, в которой есть текст *из*

## Квантификаторы — \* + ? и {}

*abc\**

соответствует строке, в которой после ab следует 0 или более символов c

*abc+*

соответствует строке, в которой после ab следует один или более символов c

*abc?*

соответствует строке, в которой после ab следует 0 или один символ c

*abc{2}*

соответствует строке, в которой после ab следует 2 символа c

*abc{2,}*

соответствует строке, в которой после ab следует 2 или более символов c

*abc{2,5}*

соответствует строке, в которой после ab следует от 2 до 5 символов c

*a(bc)\**

соответствует строке, в которой после ab следует 0 или более последовательностей символов bc

*a(bc){2,5}*

соответствует строке, в которой после ab следует от 2 до 5 последовательностей символов bc

## Оператор ИЛИ — | или []

*ab|c*

соответствует строке ab или c

*a(b|c)*

соответствует строке ab или ac

*a[bc]*

как и в предыдущем примере

## Символьные классы — \d \w \s и .

*\d*

соответствует одному символу, который является цифрой

*\w*

соответствует символу из слова (может состоять из букв, цифр и подчеркивания)

*\s*

соответствует символу пробела (включая табуляцию и прерывание строки)

*.*

соответствует любому символу

У операторов *\d*, *\w* и *\s* также есть отрицания ― *\D*, *\W* и *\S* соответственно.

\D

соответствует одному символу, который не является цифрой

*\W*

соответствует символу, который не участвует в словообразовании

*\S*

соответствует символу, который не является символом пробела

## Скобочные группы в регулярных выражениях ― ()

a(bc)

создаём группу со значением bc  [тест](https://regex101.com/r/cO8lqs/11)

a(?:bc)\*

оперетор ?: отключает группу  [тест](https://regex101.com/r/cO8lqs/12)

a(?<foo>bc)

так, мы можем присвоить имя группе  [тест](https://regex101.com/r/cO8lqs/17)

Любые множественные совпадения, по нескольким группам, будут представлены в виде классического массива: доступ к их значениям можно получить с помощью индекса из результатов сопоставления. Если присвоить группам имена (используя (?<name>...)), то можно получить их значения, под ключами равными именам групп.

## Наборы символов внутри скобочных выражений - []

*[клм]*

соответствует символу к или л или м, такой же эффект от к|л|м

*[к-м]*

то же, что и выше

*[0–9]%*

символ от 0 до 9 перед знаком %

*[a-fA-F0–9]*

одна шестнадцатеричная цифра

*[а-яА-Я]*

буква от а до я или от А до Я.

*[^а-яА-Я]*

Любой символ кроме буквы от а до я или от А до Я. В этом случае ^ используется как отрицание (любой символ кроме перечисленных)

## Экранирование метасимволов в регулярном выражении

Некоторые символы, например *^.[$()|\*+?{\* , необходимо выделять обратным слешем *\* .

\$\d

соответствует строке, в которой после символа $ следует одна цифра

Непечатаемые символы также можно искать, например табуляцию *\t*, новую строку *\n*, возврат каретки *\r*.

## Жадные и ленивые сопоставления

Квантификаторы *\* + {}* ― это «жадные» операторы, потому что они продолжают поиск соответствий, как можно глубже ― через весь текст.

Например, выражение

*<.+>*

соответствует <div>simple div</div> в This is a <div> simple div</div> test. Чтобы найти только тэг div ― можно использовать оператор ?, сделав выражение «ленивым»:

*<.+?>*

соответствует любому символу, один или несколько раз найденному между < и >  [тест](https://regex101.com/r/cO8lqs/24)

Обратите внимание, что хорошей практикой считается не использовать оператор *.* , в пользу более строгого выражения:

*<[^<>]+>*

соответствует любому символу, кроме < или >, один или более раз встречающемуся между < и > [тест](https://regex101.com/r/cO8lqs/23)

## Границы слов ― \b и \B

\babc\b

выполняет поиск слова целиком [тест](https://regex101.com/r/cO8lqs/25)

\b

соответствует границе слова, наподобие якоря (он похож на $ и ^), где предыдущий символ ― словесный (например, \w), а следующий ― нет, либо наоборот, (например, это может быть начало строки или пробел).

\B

соответствует несловообразующей границе. Соответствие не должно обнаруживаться на границе \b .

\Babc\B

соответствует, только если шаблон полностью окружён словами [тест](https://regex101.com/r/cO8lqs/26)

# 

# ООП в PHP

## Определение объекта и класса

### Объект

Объекты в PHP — это просто ещё один тип данных. Объект позволяет хранить в переменной набор из свойств и их значений, а также встроенные функции. Это делает объекты похожими по своей структуре на ассоциативные массивы. Но отличие от массивов всё-таки есть, и при этом достаточно важное — объекты могут иметь внутреннее состояние.

Давайте разберёмся, что такое PHP-объект. Как сказано выше, объекты похожи на массивы, но со своими особенностями. Объекты могут содержать отдельные значения, каждое под своим ключом. Эти значения называются свойствами объекта.

Также объекты могут иметь внутри себя функции — их называют методами объекта. Методы могут обращаться к любым свойствам объекта, читать и записывать туда данные.

Значение свойства объекта может быть любого типа: число, строка, массив, другой объект. Но, в отличие от массива, объекты не позволяют добавлять в себя новые свойства.

То есть объект всегда имеет конечное число своих свойств и методов. Менять значения существующих свойств можно, а удалять и заменять их — нельзя. Что в корне отличается от поведения массива, ведь там добавлять и удалять значения можно в любое время.

Но самая большая особенность объектов — это то, как они создаются. Если массив создается либо пустым, либо сразу с набором значений, то объекты устроены иначе. Дело в том, что объекты не существуют сами по себе. Чтобы создать новый объект, вам придется сначала создать его описание — класс.

### Класс

Класс — это шаблон объекта. Класс описывает то, из чего состоит объект. Мы не можем создать объект «на лету», как это происходит с массивами. Объект может быть создан только на основе своего описания — класса.

Сам по себе класс не является чем-то физическим и осязаемым, то есть мы не можем использовать его в коде непосредственно. Вместо этого класс является схемой, структурой, на основе которой будет создан объект.

В классе мы фиксируем из каких свойств и методов будет состоять каждый его экземпляр. Также в классе можно задать начальные значения для каждого свойства.

## Синтаксис объявления класса, его свойств и методов

<?php

class SimpleClass

{

// объявление свойства

$var = 'значение по умолчанию';

// объявление метода

function display\_var() {

echo $this->var;

}

}

$object = new SimpleClass();

$object->display\_var();

?>

## Статические и нестатические свойства и методы

В PHP существует два вида свойств и методов. Одни относятся к самому классу как шаблону, другие к объекту.

Свойства и методы, которые относятся к объекту, называют нестатическими. Те, которые относятся к шаблону объекта (классу) называют статическими.

Статическими свойства и методы становятся, если добавить ключевое слово static.

<?php

class Product {

// объявление свойства

public static $company = 'ООО Аспро';

public static $price = 50000;

public $name = 'Максимум';

// объявление метода

function display\_company() {

echo self::$company;

}

static function display\_price() {

return self::$price;

}

}

$object = new Product();

$object->display\_company();

echo '<br>';

echo Product::$company;

echo '<br>';

echo Product::display\_price();

К статическим свойствам можно обратиться как в статическом, так и в нестатическом методе. К нестатическим свойствам можно обратиться только в рамках нестатического метода.

К статическим свойствам можно обратиться не создавая экземпляр класса. Если свойство публичное - то можно получить или установить свойство в любом месте кода.

## Конструктор и деструктор класса

### Конструктор

Конструктор вызывается в момент, когда мы создаем экземпляр класса. В конструкторе можно учесть получение аргументов и произвести операции над ними. Например записать в свойства класса.

Передаются аргументы при создании объекта.

<?php

$product = new Product('Prioity', 30000);

В классе конструктор объявляется следующим образом.

<?php

class Product {

public $price = 10000;

public $name;

function \_\_construct($name, $price) {

$this->name = $name;

$this->price = $price;

print "Конструктор класса Product\n";

}

function get\_label() {

return "Решение: {$this->name}. Цена - {$this->price}";

}

}

### Деструктор

Деструктор вызывается в момент, когда экземпляр класса уничтожается (удаляется из памяти). Может быть использован, чтобы закрыть соединение с Базой Данных, для удаления дополнительных переменных и очистки памяти и т.д.

<?php

class DataBase {

function \_\_destruct() {

print "Уничтожается " . \_\_CLASS\_\_ . "\n";

}

}

## Магические методы

Магические методы - это методы, которые вызываются на различных этапах жизненного цикла объекта. Если магический метод определен у класса, мы можем вмешаться и изменить ход какого либо процесса, происходящего с объектом.

Следующие названия методов считаются магическими (<https://www.php.net/manual/ru/language.oop5.magic.php>)

\_\_construct

\_\_destruct

\_\_call

\_\_callStatic

\_\_get

\_\_set

\_\_isset

\_\_unset

\_\_sleep

\_\_wakeup

\_\_serialize

\_\_unserialize

\_\_toString

\_\_invoke

\_\_set\_state

\_\_clone

\_\_debugInfo

### \_\_call и \_\_callStatic

Если метода, который вы пытаетесь вызвать у класса, не существует - вызовется один из магических методов. \_\_call для нестатических методов и \_\_callStatic для статических.

В аргументах будет название несуществующего метода и массив с его аргументами (переданными при вызове).

<?php

class User {

public $firstname;

public $lastname;

public static $type = 'human';

function \_\_construct($firstname, $lastname) {

$this->firstname = $firstname;

$this->lastname = $lastname;

}

public function \_\_call ($name, $arguments)

{

echo "\_\_call";

echo '<br>';

switch($name) {

case 'display\_greetings':

echo "Здравствуйте {$this->firstname} {$this->lastname}!<br>";

break;

}

}

public static function \_\_callStatic ($name, $arguments)

{

echo "\_\_callStatic";

echo '<br>';

switch($name) {

case 'display\_object\_type':

$type = self::$type;

echo "Объект типа {$type}<br>";

break;

}

}

// public function display\_greetings() {

// echo "Здравствуйте {$this->firstname} {$this->lastname}!<br>";

// }

}

### \_\_get и \_\_set

Данные методы используются, чтобы получить либо установить значение свойства, которого не существует в объекте. То есть данные магические методы работают только с нестатическими свойствами.

<?php

class User {

public $firstname;

public $lastname;

function \_\_construct($firstname, $lastname) {

$this->firstname = $firstname;

$this->lastname = $lastname;

}

public function \_\_get ($name)

{

echo "\_\_get";

echo '<br>';

switch($name) {

case 'full\_name':

return "{$this->firstname} {$this->lastname}";

}

}

public function \_\_set ($name, $value)

{

echo "\_\_set";

echo '<br>';

switch($name) {

case 'full\_name':

[$firstname, $lastname] = explode(' ', $value);

$this->firstname = $firstname;

$this->lastname = $lastname;

break;

}

}

}

### \_\_toString

Иногда нам нужно определить строковое представление объекта для команды echo. Если у класса определен магический метод \_\_toString, мы сможем вывести строковое представление.

<?php

class User {

public $firstname;

public $lastname;

function \_\_construct($firstname, $lastname) {

$this->firstname = $firstname;

$this->lastname = $lastname;

}

public function \_\_toString ()

{

return "Пользователь: <b>{$this->firstname} {$this->lastname}</b>";

}

}

## Наследование

Иногда нам требуется реализовать одну и ту же логику в нескольких классах. Можно продублировать код, однако в будущем это приведет к проблемам - некто может модифицировать метод одного класса, но не знать что та же логика существует еще в двух. Как итог - часть программы сломается.

В подобной ситуации возникает желание создать некий шаблон объекта, который включает в себя только общие свойства, методы, константы для неких будущих классов.

В таком случае можно будет менять общую логику только в нем, точно зная, что все дочерние классы уже будут включать в себя новую логику.

*Синтаксис наследования*

У одного класса может быть множество наследников, иногда наследнику требуется переопределить общую логику родителя, дополнить ее.

*Переопределение родительского метода*

Иногда бывают случаи, когда метод настолько важен для класса, что вы не хотите позволять его переопределять в наследниках. Для этого существует ключевое слово final.

*Метод с final*

В логике программы нам может понадобиться точно знать, является ли переданный аргумент экземпляром определенного класса. Это нужно, чтобы понимать есть ли у него нужные методы и свойства для работы.

Можно указать тип аргумента (с версии 7.\*) либо использовать конструкцию instanceof.

Instanceof определяет относится ли объект к классу. Также эта конструкция работает для дочернего объекта, который унаследован от интересующего нас класса.

*Пример instanceof и аргумента с типом*

### Позднее статическое связывание

Это частный случай работы со статическими методами и наследованием. Представьте, что вам нужно создать в родительском классе общее статическое свойство. Также у вас есть некий метод, который это свойство использует. Но вот незадача - мы хотим, чтобы статическое свойство могло предопределяться наследниками, но используем мы его в родительском классе!

Соответственно метод родительского класса через ключевое слово self обращается к себе и своему свойству, а не к свойству наследника.

Чтобы решить эту проблему было введено ключевое слово static. Оно позволяет обратиться к статическим свойствам и методам класса, исходя из контекста использования.

## Открытые и закрытые методы и свойства. Инкапсуляция

Существует три уровня доступа к свойствам и методам класса. Публичный, приватный и защищенный.

Уровни доступа по другому называются областью видимости. Сокрытие части алгоритма программы внутри области видимости родительского класса называется инкапсуляцией.

Скрывают часть реализации алгоритма по нескольким причинам. Во первых некоторые свойства и методы могут быть настолько важны, что любое несанкционированное вмешательство неопытного разработчика может привести в падению системы.

Представьте, что у Вас не заводится машина и Вы, увы, не механик и плохо разбираетесь в машинах. Вы открываете капот и начинаете выдергивать какие-то шланги, что-то откручивать. и т.д. Хорошо, если Вы запомнили что, где и как выдергивали и откручивали. А если нет ?

Или, у Вас стрелка уровня топлива стоит на нуле, а Вы считаете, что у Вас полно топлива и полезете со спичками внутрь бензобака проверять уровень топлива. Какие последствия Вас могут ожидать ? В лучшем случае Вы и Ваша машина останутся живы, если Вам очень повезет.

Аналогично и с нашими объектами, которые могут быть чрезвычайно сложными, а Вы пытаетесь что-то в них подправить, не представляя их внутреннюю организацию.

Для того, чтобы починить машину не причинив себе и самой машине вреда (ну, разве только финансовые затраты с Вашей стороны) необходимо пригласить квалифицированных автослесарей, причем каждый из которых отлично разбирается только в определенной части Вашей машины. Если Вы скажете, что у Вас не горит лампочка подсветки в салоне, то замену лампочки проведет специалист по электрооборудованию автомобилей. И т.д.

Аналогично и в нашем объекте. Есть "мастера" - методы, которые "специализируются" в определенных областях, но свою область они знают на "5" баллов. А самое главное, они знают как можно изменить состояние объекта так, чтобы не повредить его.

Во вторых для удобства, если снаружи доступно мало методов - в них проще разобраться, чем в большом объеме функциональности, которая вне класса не нужна.

Современные системы для разработки позволяют увидеть доступные методы и свойства класса при его использовании в коде.

Например, если вы хотите завести машину, вы не начинаете открывать капот и расталкивать поршни. Вы просто поворачиваете ключ и происходит зажигание, хотя под капотом происходит множество процессов.

Понятие инкапсуляции тесно связано с наследованием. Например публичный уровень доступа позволяет обратиться к методу или свойству в любом месте программы.

Защищенный уровень позволяет обратиться к свойству или методу только в рамках самого класса или его наследников.

Приватный уровень позволяет обратиться к свойствами и методам только в рамках текущего класса и больше нигде, даже в наследниках не получится.

В классах наследниках вы можете расширить область видимости свойств и методов. Например сделать из приватного защищенный или даже публичный. Но обратный процесс недопустим, сделать приватным метод, который был публичным в родителе не получится.

## Интерфейсы

# Практика

## Инициализация проекта

1. Создаем репозиторий Git на любой удобной платформе (github, bitbucket).
2. Создаем файл /bootstrap.php
3. Регистрируем функцию spl\_autoload\_register с подключением класса из папки /classes. Название подключаемого файла = название класса в нижнем регистре.
4. Создаем класс Layout согласно паттерну Singleton
5. Описываем логику подключения статики. Метод должен принимать путь к файлу относительно папки со статикой (static/css или static/js).
6. Статика для стилей и скриптов должна собираться отдельно, в свойства класса, в виде строки (HTML).
7. Реализуем подключение шрифта из Google Fonts (можно выбрать любой). Метод должен принимать название шрифта и вызываться в конструкторе класса.
8. Создаем файл с общими стилями static/css/bootstrap.css и подключаем его в конструкторе класса. Подключаем в нем шрифт для всего сайта.
9. Создаем index.php и подключаем в нем bootstrap.php. Можно что то вывести на экран при желании
10. Создаем файл /catalog/index.php
11. Описываем четыре товара в произвольном формате и встраиваем в верстку
12. Создаем файл стилей, подключаем и делаем сетку из 4 товаров с адаптивом на гридах. Также можно стилизовать сами товары.

## Настройки проекта

1. Создаем класс Config. В нем должно быть свойство, куда сохраняются настройки проекта. Также метод для разработчика, который позволяет получить настройку проекта по названию файла настроек и ключу массива настроек.
2. Необходимо написать так, чтобы файл настроек запрашивался лишь тогда, когда обратились к его настройкам. При повторном обращении настройки должны отдаваться уже из кеша.
3. Создаем конфиг configs/layout.php. В нем указываем ключ font и значение Montserrat.
4. В классе Layout переписываем логику так, чтобы подключался шрифт из настроек. Также необходимо проследить, чтобы правильный шрифт подключался в стилях проекта.

## Подключение к Базе Данных

1. Создаем класс DB согласно паттерну Singleton
2. В конструкторе класса вызываем метод connect. Он должен инициировать подключение к БД по PDO.
3. Соединение с БД должно храниться в приватной переменной класса и инициализироваться лишь один раз.
4. Создаем конфиг db. В нем указываем необходимые настройки для подключения к БД.
5. Необходимо написать метод, который будет создавать таблицу в БД с указанным именем и колонками. У колонок может быть указано название, тип (завести константами), является ли null по умолчанию, является ли primary key. Можете добавить еще настроек по желанию.

## Создание моделей

1. Дополняем класс DB новыми методами. Table\_exists должен принимать название таблицы и возвращать bool.
2. Insert должен отвечать за запись в таблицу. Должен принимать название таблицы и данные массивом. Возвращать должен id новой записи, если все прошло успешно.
3. Select получает данные из таблицы. Принимает название таблицы, необязательное условие для построения WHERE и необязательный LIMIT. Возвращает результат выборки.
4. Создаем класс Model. Его задача в том, чтобы описывать таблицы БД и их содержимое в более привычном виде.
5. Для начала нужны свойства table\_name и table\_columns. Они позволят модели понимать структуру таблицы и как с ней правильно работать.
6. Продумаем требования к описанию колонок в будущих моделях. Первый вариант формата - когда ключ является названием колонки, а значение типом. Расширенный, когда ключ опять же является названием колонки, а значение - массив с описанием параметров колонки. Например type, null.
7. Создадим метод create\_table он должен по названию таблицы и массиву с колонками создать таблицу в БД. Необходимо учесть оба способа задания колонок. Также учтем, что существует PRIMARY KEY. Сразу заложим его автоматическое создание в методе и создадим свойство primary\_key, в котором будет указано название колонки.
8. Также нам нужен некий динамический набор свойств. Ведь у моделей будут разные таблицы и колонки, не хотелось бы описывать каждый набор свойств в классах моделей. Для этого создадим свойство properties и магические методы \_\_get и \_\_set. В них реализуем логику, если устанавливаемое свойство есть в списке колонок таблицы (по свойству модели), тогда сохраняем в массив properties либо получаем из него значение.
9. Теперь нужно позаботиться о том, чтобы при обращении к модели таблица гарантированно существовала в БД. В рабочих системах эта проблема решается тем, что таблицы создаются при установке модуля или регистрации пользователя. В нашем случае сделаем проверку на наличие таблицы и ее создание при необходимости. Создадим метод check\_table в конструкторе класса.
10. Метод для проверки существования таблицы у нас уже есть, воспользуемся им. Но есть проблема. Каждый раз, при обращении к классу, мы будем проверять наличие одной и той же таблицы. Закешируем результат в переменной класса. Сделаем массив с ключами - названия таблиц, и значениями - существует ли в системе.
11. Далее необходима логика сохранения модели. Создадим метод save, который будет вызывать методы create или update в зависимости от того, загружена ли уже запись из базы. Также нам понадобится свойство loaded, которое будет хранить состояние загруженности.
12. Метод create будет производить запись в базу. Также нужно получить id новой записи и сохранить в модели. Состояние что модель загружена также забывать не стоит.
13. Теперь создадим функционал задания свойств для модели (значений колонок). Создадим метод set, который будет принимать массив со свойствами и присваивать их модели.
14. Теперь нужно учесть инициализацию модели по ее id. Заложим это в конструктор. Пусть он будет принимать параметр с id записи. Если он передан - загружаем модель по id (стоит написать для этого отдельный метод).
15. Теперь схема взаимодействия принимает следующий вид - получаем экземпляр класса, устанавливаем ему свойства, сохраняем. Либо инициализируем экземпляр класса сразу по id и получаем значения. Создайте класс Product и опишите эту модель. Создайте товары в БД с помощью только что прописанной логики.

## Получение товаров из БД

1. Нужно понять, с какой целью мы будем получать записи из базы? Вариантов несколько. Во первых получить все записи из таблицы. При этом могут понадобиться по некому фильтру либо ограниченное количество. Во вторых мы можем захотеть получить одну запись по некому фильтру. Например если хотим понять, существует ли вообще элемент, подходящий по условию, при этом нам не хочется собирать их все. Или нам нужен любой элемент, подходящий по условию.
2. Из этого следует вывод, что нам требуется два метода - find и find\_all. Первый будет искать первое совпадение, второй все совпадения. Оба метода должны принимать условие для конструкции WHERE. Также методу find ограничение количества элементов ни к чему.
3. Метод find должен возвращать готовый экземпляр класса модели с заполненными свойствами либо вернуть не загруженную модель. Также нам требуется проверять из внешнего кода, загружена ли модель. Напишем метод loaded. Метод нужен для того, чтобы разработчик мог получить состояние, но не установить его.
4. Метод find\_all делает в целом то же самое. Но он должен вернуть массив с моделями либо пустой массив.

## Роутинг

1. Настроим редиректы на главную страницу независимо от реального URL. Для этого у нас должен быть сервер Apache либо вы можете сделать настройку самостоятельно для Nginx. У меня получилась такая конфигурация

Options -Indexes

RewriteEngine On

RewriteCond %{REQUEST\_URI} !\?

RewriteCond %{REQUEST\_URI} !\&

RewriteCond %{REQUEST\_URI} !\=

RewriteCond %{REQUEST\_URI} !\.

RewriteCond %{REQUEST\_URI} !.\*/$

RewriteRule ^(.\*)$ /$1/ [R=301,L]

RewriteCond %{REQUEST\_URI} !(^/index.php)

RewriteCond %{REQUEST\_URI} !(^/static/\*)

RewriteRule .\* /index.php [L]

1. Теперь создадим класс Route (Singleton). Он будет отвечать за обработку перенаправлений. Нам нужно реализовать два механизма - поиск и подключение файла по регулярному выражению, установку и получение параметров из URL согласно регулярному выражению.
2. Создадим конфиг routes. Там будет массив массивов с ключами url и file. В url регулярное выражения для поиска совпадения, в file путь к файлу относительно корня.
3. В конструкторе класса реализуем загрузку правил из конфига, а также получение текущего URL из переменной $\_SERVER.
4. Реализуем метод load\_file он должен пройтись по всем правилам роутинга, если найдено совпадение с текущим URL - подключить указанный файл и установить именованные параметры из регулярного выражения. Если совпадение не найдено либо файла физически не существует нужно инициировать исключение с кодом 404.
5. Затем реализуем метод get\_param он должен принимать ключ параметра и вернуть его значение, если таковой имеется. Либо не принимать ничего и вернуть все доступные параметры.
6. Реализуем два роута. Для списка товаров /catalog/ и для будущей детальной страницы товара /catalog/(?<product\_id>[^/]\*)/ пусть ведет на страницу /catalog/product.php
7. Модернизируем код главной страницы. Вызовем метод load\_file и обернем его в try catch. Если случилась ошибка - получим ее код и попробуем подключить файл ошибки из директории /errors/<error\_code>.php. Если его нет - просто выведем текст ошибки.
8. Теперь со страницы списка товаров можно убрать подключение bootstrap.php. А на детальной странице товара вывести его product\_id.

## Взаимодействие с Vue.js

1. Для начала скачаем Vue.js в файлы статики. Как вариант можно использовать ссылки на CDN, но мы не хотим зависеть от сторонних сервисов. Ссылка с инструкцией <https://vuejs.org/v2/guide/> . Подключим Vue на странице товара.
2. Создаем rest для получения информации о товаре. Пусть URL будет /catalog/rest/product/<product\_id>/ . Файл подключим catalog/rest.php
3. Далее нужно дописать логику класса Route, чтобы он составлял корректное регулярное выражение на основе подобной ссылки.
4. Займемся ответом сервера на запрос. Необходимо вернуть информацию о товаре в виде JSON. Для этого нужно в классе Model дописать метод, который будет возвращать свойства и их значения в виде массива.
5. Далее нужно прописать шаблон компонента в JS файле и подключить его. В методе created жизненного цикла отправляем ajax запрос на rest получения информации о товаре.
6. Также на странице товара нужно передать ID текущего товара для корректного формирования URL запроса. При получении ответа на запрос подставляем данные в data.