

# ПРОСТРАНСТВА ИМЕН, ПЕРЕГРУЗКА И ВСТРОЕННЫЕ ИНТЕРФЕЙСЫ И КЛАССЫ



ЕВГЕНИЙ КОРЫТОВ / РУКОВОДИТЕЛЬ ГРУППЫ ВЕБ-РАЗРАБОТКИ, CONDÉ NAST



# ЕВГЕНИЙ КОРЫТОВ

Руководитель группы веб-разработки,  
Condé Nast



korytoff@gmail.com




korytoff



# ЧТО МЫ УЖЕ ДОЛЖНЫ ЗНАТЬ?

- Классы, объекты, свойства, методы
- Наследование, переопределение методов и свойств
- Область видимости свойств и методов
- Абстрактные классы и интерфейсы
- Статические свойства и методы
- Примеси



# **ПЕРЕГРУЗКА И МАГИЧЕСКИЕ МЕТОДЫ**



# ПЕРЕГРУЗКИ В PHP НЕ СУЩЕСТВУЕТ!

Вообще, понятие перегрузки означает несколько иное в других языках программирования.

По сути, перегрузка - это возможность объявить два разных метода даже в рамках одного класса, с одним названием, но разным количеством аргументов. И в зависимости от количества аргументов будет вызываться тот или иной метод.

## ПЕРЕГРУЗКИ В PHP НЕ СУЩЕСТВУЕТ! (Ч.2)

```
1 class TestClass
2 {
3     public function testMethod($a) { // тут одно поведение
4     public function testMethod($a, $b, $c) { // тут другое
5     // но так в PHP делать НЕЛЬЗЯ
6 }
```

Вместо этого, перегрузка в PHP подразумевает возможность динамически создавать и работать с недоступными свойствами и методами.

# ЧТО ЗНАЧИТ С НЕДОСТУПНЫМИ?

1. Защищенные или приватные свойства и методы (да, да, да)
2. Несуществующие свойства и методы, которые мы вообще не создавали

Вообще, в PHP можно так:

```
1 class TestClass { } //пустой класс без свойств
2 $test = new TestClass();
3 $test->a = 100; //динамическое создание и присваивание зн
4 echo $test->a; // обращение к динамическому свойству
```

# ЗАЧЕМ ЭТО ВСЁ?

На самом деле, целесообразность и зачем это всё, проще объяснить, когда будет понятно, о чём именно речь и как этим пользоваться.

Итак, помните, мы говорили, что в PHP есть несколько магических методов, которые вызываются, если происходит определенное событие?

Один из них - это метод `__construct()`; Событие - момент создания объекта

Все магические методы должны быть объявлены, как `public`.





# НЕМНОГО МАГИИ ДЛЯ СВОЙСТВ

Для свойств, PHP под перегрузкой подразумевает работу со следующими методами:

1. `__get($property)`; - вызывается при обращении к недоступному свойству
2. `__set($property, $value)`; - вызывается при попытке дать значение недоступному свойству
3. `__isset($property)`; - вызывается при вызове `isset()` на недоступном свойстве
4. `__unset($property)`; - вызывается при вызове `unset()` на недоступном свойстве

## \_\_GET(\$PROPERTY)

С помощью данного метода мы можем разрешить обращаться к недоступному свойству

```
1 class TestClass {
2     public function __get($property) // $property
3     {
4         echo 'Обратились к недоступному свойству - ' . $property;
5     }
6 }
7 $test = new TestClass();
8 echo $test->a; //аналогично вызову $test->__get('a');
9 echo $test->lalala; //аналогично вызову $test->__get('lalala');
```

## \_\_GET(\$PROPERTY) (4.2)

```
1  class TestClass {
2      private $privateProperty;
3      protected $protectedProperty;
4
5      public function __get($property) // $property
6      {
7          echo 'Обратились к недоступному свойству - ' . $property;
8      }
9  }
10 $test = new TestClass();
11 echo $test->privateProperty; //аналогично вызову $test->privateProperty;
12 echo $test->protectedProperty; //аналогично вызову $test->protectedProperty;
```

## \_GET(\$PROPERTY) (4.3)

```
1  class TestClass {
2      private $privateProperty = 10;
3      protected $protectedProperty;
4
5      public function __get($property) // $property
6      { // лайфхак - так мы открываем доступ к обращению к
7          if (isset($this->$property))
8              return $this->$property;
9      }
10 }
11 $test = new TestClass();
12 echo $test->privateProperty; //выведет 10 (!!!)
```

## **\_GET(\$PROPERTY) (Ч.4)**

К вопросу зачем нам открывать доступ к приватным и защищенным свойствам.

На самом деле, так делать не надо, иначе мы теряем все преимущества областей видимости и можем столкнуться с неожиданным поведением.

Но при этом, это даёт нам возможность полного контроля доступа к свойствам, т.е. делает работу с объектами более гибкой.

## \_\_GET(\$PROPERTY) (4.5)

```
1  class TestClass {
2      public $property = 10;
3
4      public function __get($property) // $property - строка
5      { // лайфхак - так мы открываем доступ к обращению к
6          if (isset($this->$property))
7              echo 'Вызвано недоступное свойство'.
8      }
9  }
10 $test = new TestClass();
11 echo $test->property; //для public - __get не вызывается!
```

## **\_\_GET(\$PROPERTY) (4.6)**

Мини-резюме по `__get`

1. Единственный аргумент содержит название недоступного свойства в виде строки
2. Должен быть объявлен, как `public`
3. Вызывается при событии: обращение к недоступному свойству
4. Не вызывается, для публичных свойств класса
5. Вызывается для `private` и `protected`
6. Может содержать внутри любую логику для разных свойств

## \_\_SET(\$NAME, \$VALUE)

\_\_set вызывает при присвоении значения недоступному свойству

```
1 class TestClass {
2     private $privateProperty = 10;
3     // $name - строка-название свойства, $value - значение
4     public function __set($name, $value)
5     { // запретили менять значения недоступных свойств
6         return 'Ошибка - попытка присвоения значения недо
7     }
8 }
9 $test = new TestClass();
10 $test->privateProperty = 20; // == __set('privateProperty
11 $test->asdasd = 'lalala'; // == __set('asdasd', 'lalala')
```



## \_\_SET(\$NAME, \$VALUE) (4.2)

И да, таким образом мы можем разрешить менять приватные свойства

```
1 class TestClass {
2     private $privateProperty = 10;
3     public function __set($name, $value)
4     {
5         $this->$name = $value;
6     }
7 }
8 $test = new TestClass();
9 $test->privateProperty = 20; // == __set('privateProperty', 20)
10 echo $test->privateProperty; // == 20, мы поменяли значение
```

## \_\_SET(\$NAME, \$VALUE) (4.3)

\_\_set ведет себя аналогично \_\_get для существующих public свойств

```
1 class TestClass {
2     public $publicProperty = 10;
3     // $name - строка-название свойства, $value - значение
4     public function __set($name, $value)
5     { // запретили менять значения недоступных свойств
6         return 'Ошибка - попытка присвоения значения недоп
7     }
8 }
9 $test = new TestClass();
10 $test->publicProperty = 10; // __set не вызывается!
```

## **\_SET(\$NAME, \$VALUE) (Ч.4)**

Таким образом мы можем, например, разрешать доступ только к приватным переменным, или создавать динамические свойства, запрещая доступ к существующим, если они приватные или защищенные.

Еще одним способом применения может быть создание автоматических геттеров и сеттеров

# АВТОМАТИЧЕСКИЕ ГЕТТЕРЫ И СЕТТЕРЫ

```
1  class TestClass {
2      private $data = [];
3      public function __set($name, $value)
4      {
5          $this->data[$name] = $value;
6      }
7      public function __get($name, $value)
8      {
9          if (isset($this->data['name']))
10             return $this->data[$name];
11      }
12 }
```

## **\_SET(\$NAME, \$VALUE) (4.5)**

### Мини-резюме

1. \$name - строка, название свойства, \$value - значение, которое нужно присвоить
2. Должен быть объявлен, как public
3. Вызывается при событии: присвоение значения недоступному свойству
4. Не вызывается, для публичных свойств класса
5. Вызывается для private и protected
6. Может содержать внутри любую логику для разных свойств

## \_\_ISSET(\$PROPERTY)

\_\_isset срабатывает при вызове isset не несуществующем свойстве

```
1  class TestClass {
2      protected $property = 10;
3      // $property - строка-название свойства
4      public function __isset($property)
5      {
6          return 'Ошибка - недоступное свойство';
7      }
8  }
9  $test = new TestClass();
10  isset($test->property); // ошибка
```

## \_\_ISSET(\$PROPERTY) (4.2)

\_\_isset также не вызовется для public свойств

```
1 class TestClass {
2     public $property = 10;
3     // $property - строка-название свойства
4     public function __isset($property)
5     {
6         return 'Ошибка - недоступное свойство';
7     }
8 }
9 $test = new TestClass();
10 isset($test->property); // true - нет ошибки
```

# \_\_UNSET(\$PROPERTY)

\_\_unset вызывается при попытке удалить свойство

```
1 class TestClass {
2     private $property = 10;
3     // $property - строка-название свойства
4     public function __unset($property)
5     {
6         return 'Ошибка - наши свойства нельзя удалять';
7     }
8 }
9 $test = new TestClass();
10 unset($test->property); // ошибка
```



## \_UNSET(\$PROPERTY) (4.2)

А вот public - можно

```
1  class TestClass {  
2      public $property = 10;  
3      // $property - строка-название свойства  
4      public function __unset($property)  
5      {  
6          return 'Ошибка - наши свойства нельзя удалять';  
7      }  
8  }  
9  $test = new TestClass();  
10 unset($test->property); // нет ошибки
```

# `__ISSET($NAME)` И `__UNSET($NAME)`

## Мини-резюме

1. `$name` - строка, название свойства
2. Должны быть объявлен, как `public`
3. Вызывается при событии: проверка свойства на существование (`__isset`) либо удаление свойства (`__unset`)
4. Не вызывается, для публичных свойств класса
5. Вызывается для `private` и `protected`
6. Может содержать внутри любую логику для разных свойств

## **`__ISSET($NAME)` И `__UNSET($NAME)` (4.2)**

Например, мы можем захотеть, чтобы ни одно наше свойство нельзя было заансетить.

В этом случае можно объявить все свойства приватными или защищенными, прописать соответствующую ошибку в `__unset`, а в `__get` и `__set` прописать к каким свойствам можно обращаться

Разберем это в коде, а потом перейдём к методам.

## О МЕТОДАХ

Помимо вызова недоступных свойств мы также можем полностью контролировать и обрабатывать вызов недоступных или несуществующих методов. Для этого используются

1. `__call` - для вызова недоступного метода (в контексте объекта)
2. `__callStatic` - для вызова недоступного метода (в контексте класса)

Почти всё аналогично работе со свойствами, поэтому рассмотрим их кратко.

## \_\_CALL(\$NAME, \$ARGUMENTS)

\_\_call вызывается для недоступных методов, первый аргумент - строка, название метода, а второй - массив аргументов

```
1 class TestClass {
2     public function __call($methodName, $arguments)
3     {
4         echo "Вызван метод - " . $methodName
5             . ' с параметрами: ';
6         var_dump($arguments);
7     }
8 }
9 $test = new Test();
10 $test->someMethod(123, 234, 345); //аналогично __call('sc
```

# \_\_CALLSTATIC(\$NAME, \$ARGUMENTS)

`__callStatic` вызывается для недоступных статических методов, первый аргумент - название метода, а второй - массив аргументов

```
1 class TestClass {
2     public static function __callStatic($methodName, $arguments)
3     {
4         echo "Вызван статический метод - " . $methodName . "\n";
5         var_dump($arguments);
6     }
7 }
8 //аналогично __callStatic('someMethod', [123, 234, 345])
9 Test::someMethod(123, 234, 345);
```

## **\_\_CALLSTATIC И \_\_CALL**

1. `__callStatic` должен быть объявлен как статический метод
2. `__callStatic` вызывается, если мы обращаемся к методу класса
3. `__call` вызывается, если мы обращаемся к методу объекта
4. Все они должны быть объявлены, как `public`
5. Первый аргумент - всегда строка, с названием метода к которому обращаемся
6. Существующие публичные методы не вызывают срабатывания этих методов



## РЕЗЮМЕ ПО МАГИЧЕСКИМ МЕТОДАМ

Для чего это может быть нужно? На самом деле в простых системах потребности в этих методах немного.

Обычно подобные конструкции активно применяются во фреймворках либо для интересных и специфичных задач, когда нужно определенным образом организовать взаимодействие классов и объектов.

На данном этапе попробуйте осознать те возможности, что это даёт. Фактически, вы можете перехватывать определенные события и обрабатывать их как душе угодно





# АВТОЗАГРУЗКА

---

## О ЗАГРУЗКЕ КЛАССОВ

Для того, чтобы класс можно было использовать, сперва его нужно подключить в приложение. Самый "прямой" (но не самый простой и правильный) путь - это вызывать для каждого класса `require / include`

Когда у вас 3-5 классов, это не создаёт никаких проблем. Но если вы создаёте более-менее сложную систему, загружать через `require` - грусть и печаль.

Может поменяться название файла - и в каждом месте, где класс подключается, нужно не забыть предусмотреть эти изменения.



## О ЗАГРУЗКЕ КЛАССОВ (Ч.2)

Поэтому для начала давайте примем как факт:

- Каждый класс должен находится в отдельном файле
- Классы могут находится в разных директориях
- Нам нужно сделать так, чтобы мы могли в нужный момент получить доступ к соответствующему классу
- Названия файлов классов лучше делать в виде определенного шаблона

## **\_\_autoload(\$classname)**

В PHP есть магическая функция (не метод!) `__autoload`

- Она принимает на вход единственный параметр - строка с именем вызываемого класса
- Как и всё магическое, она вызывается автоматически при возникновении определенного события - попытки обращения к классу, который ещё не был загружен

Итак, предположим, что все классы лежат в папке `classes`, а название классов строится по маске `*.class.php`

## \_\_autoload(\$className) (4.2)

```
1 // $className содержит название класса, который мы вызываем
2 function __autoload($className) //это функция! не метод!
3 {
4     $filePath = './classes/' . $className . '.class.php';
5     if (file_exists($filePath)) {
6         include "$filePath";
7     } else {
8         die("Класса $className не существует");
9     }
10 }
11 $test = new TestClass(); // вызывается __autoload('TestClass')
```

## **`__autoload($classname)` (4.3)**

У данной функции есть один недостаток. Если у нас классы могут находиться во множестве разных директорий, то она превращается в сборник if'ов, который затруднительно поддерживать.

В связи с этим, для решения этой проблемы существует альтернатива - `spl_autoload_register()`, а данная функция является устаревшей

Поэтому не нужно её использовать :)

# SPL\_AUTOLOAD\_REGISTER

`spl_autoload_register` может регистрировать любое количество функций автозагрузчиков, которые мы создадим. Они будут вызываться последовательно.

Как выглядит алгоритм?

1. Создаём любое количество функций, которые будем использовать для автозагрузчиков
2. Регистрируем их через `spl_autoload_register`
3. Получаем счастье и радость

## SPL\_AUTOLOAD\_REGISTER (4.2)

Мы можем как угодно называть наши функции автозагрузки, а первым аргументом в них всегда будет `$className`

```
1 // $className содержит название класса, который мы вызываем
2 function myAutoload($className)
3 {
4     $filePath = './classes/' . $className . '.class.php';
5     if (file_exists($filePath)) {
6         include "$filePath";
7     }
8     //заметьте, die мы убрали, т.к. предполагаем, что
9     //будут и иные автозагрузчики после
10 }
```



## SPL\_AUTOLOAD\_REGISTER (4.3)

Создадим ещё автозагрузчик

```
1 // $className содержит название класса, который мы вызыва
2 function coreAutoloader($className)
3 { //прошлый искал классы в classes, этот берет из core
4     $filePath = './core/' . $className . '.class.php';
5     if (file_exists($filePath)) {
6         include "$filePath";
7     }
8     //и тут без die (
9 }
```

## SPL\_AUTOLOAD\_REGISTER (4.4)

Теперь регистрируем автозагрузчики

```
1 spl_autoload_register( 'myAutoload' );  
2 spl_autoload_register( 'coreAutoloader' );  
3 $test = new TestClass();
```

1. Сперва TestClass будет искаться в myAutoload - myAutoload('TestClass');
2. Затем TestClass будет искаться в coreAutoloader - coreAutoloader('TestClass');
3. Если класс не будет нигде найден - будет fatal

## SPL\_AUTOLOAD\_REGISTER (4.5)

В результате, мы можем создавать любое количество автозагрузчиков и гибко их подключать в нужной нам последовательности. Можно даже так регистрировать автозагрузчики без создания функций:

```
1 spl_autoload_register(  
2     function ($className) {  
3         include $className . '.php';  
4     }  
5 );
```



# ПРОСТРАНСТВА ИМЁН

# ПРОСТРАНСТВА ИМЁН

А теперь рассмотрим иную проблему. Допустим у нас в проекте используются **две** библиотеки от разных разработчиков. При этом **каждая** библиотека содержит следующие классы:

- Cache - используется для кэширования
- Log - используется для логирования
- Conf - для конфигурации

И у нас написаны автозагрузчики для каждой библиотеки

```
1 | $conf = new Conf(); // какой из Conf будет вызван?  
2 | // как вызвать конкретно второй?
```

## ПРОСТРАНСТВА ИМЁН (Ч.2)

Для этой цели в php существуют пространства имён. Они позволяют задать свои собственные области видимости для классов и функций и назвать их как угодно.

```
1 //допустим, файл \core\Test.php
2 namespace Core;
3
4 class Test {}
```

## ПРОСТРАНСТВА ИМЁН (Ч.3)

```
1 //допустим, файл \libs\Test.php
2 namespace Libs;
3
4 class Test {}
```

## ПРОСТРАНСТВА ИМЁН (Ч.4)

Теперь, мы можем обратиться к каждому конкретному классу следующим образом:

```
1 $test1 = new \Core\Test(); // myAutoloader('\Core\Test');  
2 $test2 = new \Libs\Test(); // myAutoloader('\Libs\Test');
```

Правда теперь наши функции автозагрузки нуждаются в переработке. Дело в том, что к ним будет приходить не только имя класса, но и всё пространство имён для него.



## ВЕРНЁМСЯ К АВТОЗАГРУЗКЕ

```
1 function myAutoload($classNameWithNamespace)
2 {
3     //учитываем пространство имён
4     $pathToFile = $_SERVER['DOCUMENT_ROOT'] //ищем файлы
5         . str_replace('\\', DIRECTORY_SEPARATOR, $classNameWithNamespace)
6         . '.php'; //добавляем расширение
7     if (file_exists($pathToFile)) {
8         include "$pathToFile";
9     }
10    // Данная функция будет работать, если namespace дублируется
11 }
```



## ПРОСТРАНСТВА ИМЁН (4.5)

- Пространства имён актуальны также для констант и функций (т.е. они также могут находиться в своём пространстве имён)
- Если пространство имён не указано, то класс / константа / функция принадлежат глобальному пространству имён

## ПРОСТРАНСТВА ИМЁН (Ч.6)

```
1 namespace MyProject;  
2 // всё это принадлежит пространству имён  
3 const MY_CONST = 10; // не define() !  
4 function myFunction() {}  
5 class MyClass {};
```

## ПРОСТРАНСТВА ИМЁН (Ч.7)

Таким образом, если указано пространство имён в файле, всё его содержимое, кроме переменных, содержится в нём.

```
1  echo \MyProject\MY_CONST;  
2  \MyProject\myFunction();  
3  echo \MyProject\MyClass::class; //выведет имя класса
```

Заметьте, что мы начинаем с символа \ - это означает, что мы ищем, начиная с корневого пространства имён.

## ПРОСТРАНСТВА ИМЁН (Ч.8)

На самом деле пространство имён ничем не ограничено, оно может быть и таким:

```
1 //подпространства имён
2 namespace MyProject\Long\MyNamespace;
```

Поэтому, что произойдёт, если мы будем обращаться к классу не от корня?

```
1 namespace MyProject;
2 $test = Long\MyNamespace\Test();
3 // == \MyProject\Long\MyNamespace\Test();
```

## ПРОСТРАНСТВА ИМЁН (Ч.9)

Если у нас есть два класса, в одном пространстве имён, нам не нужно писать полный путь до него.

Допустим у нас есть первый файл

```
1 namespace MyProject;  
2 class TestClass1 {};
```

И второй файл

```
1 namespace MyProject;  
2 class TestClass2 {};
```

## ПРОСТРАНСТВА ИМЁН (Ч.10)

Чтобы обратиться к классу из первого файла, достаточно указать его имя, т.к. пространство имён будет автоматически подставлено.

```
1 namespace MyProject;
2 class TestClass2 {
3     public $object;
4     public function __construct()
5     {
6         $this->object = new TestClass1();
7         // == \MyProject\TestClass1(); - потому что указ
8     }
9 };
```



## НЕКОТОРЫЕ ВЫВОДЫ

- Поэтому мы можем, разрабатывая конкретный модуль в одном пространстве имён, спокойно оперировать только названиями классов
- Это также решает вопрос с пересечениями названий классов
- При этом, если нам нужно вызвать один конкретный класс - мы всегда можем указать полный путь к нему
- По простому: старайтесь всегда указывать namespace'ы по аналогии с каталогами, в которых у вас лежат классы - для среднего проекта этого более чем достаточно
- Тут должно быть много вопросов )





# ВСТРОЕННЫЕ КЛАССЫ И ИНТЕРФЕЙСЫ

---

# ВСТРОЕННЫЕ КЛАССЫ И ИНТЕРФЕЙСЫ

PHP содержит некоторое количество встроенных классов и интерфейсов. На самом деле, их довольно много, но мы рассмотрим один - `ArrayAccess`, чтобы дать общее понимание, как это работает и почему.

Итак, иногда действительно неплохо было бы иметь возможность обращаться с объектами, как... с массивами. )

Например, мы можем захотеть написать свою "обёртку" над сессиями, но работать с ней, как с массивом, т.е., по сути, как и раньше, но расширяя возможности самого массива объектными "фишками".

# ЧТО ТАКОЕ ARRAYACCESS?

ArrayAccess - это простой встроенный интерфейс

```
1 interface ArrayAccess {  
2     //проверяет, существует ли смещение  
3     abstract public offsetExists ( mixed $offset )  
4     //получает значение с индексом  
5     abstract public offsetGet ( mixed $offset )  
6     //задает значение с соответствующим индексом  
7     abstract public offsetSet ( mixed $offset , mixed $value )  
8     //удаляет значение с соответствующим индексом  
9     abstract public offsetUnset ( mixed $offset )  
10 }
```

## КАК ЭТО РАБОТАЕТ?

По сути, при попытке обратиться к объекту, как к массиву - PHP будет смотреть, существуют ли соответствующие методы.

- Нужно получить значение? Ну-ка, вызовем `offsetGet`.
- Есть ли вообще значение, куда сдвигаться? Проверим с помощью `offsetExists`

От нас требуется только реализовать соответствующие методы. В чем профит? Ну, иногда так бывает привычней и удобней (мы при этом можем работать с объектом, как с массивом). Плюс, например, мы можем вообще запретить удалять значения из нашего массива-объекта.

# ARRAYACCESS

```
1  class TestClass implements \ArrayAccess //в глобальном пр
2  {
3      private $data = ['test', 'test2', 'test3'];
4
5      public function offsetSet($index, $value) {
6          if (is_null($index)) {
7              $this->data[] = $value; //добавление значения
8          } else {
9              $this->data[$index] = $value;
10         }
11     }
12 }
```

## ARRAYACCESS (4.2)

```
1  class TestClass implements \ArrayAccess //в глобальном пр
2  {
3      public function offsetExists($index) {
4          return isset($this->data[$index]);
5      }
6
7      public function offsetUnset($index) {
8          unset($this->data[$index]);
9      }
10
11     public function offsetGet($offset) {
12         return isset($this->data[$index]) ? $this->data[$
```

## ARRAYACCESS (4.3)

Работать с этим можно будет примерно так:

```
1 $test = new TestClass();  
2 $test[] = 'Новый элемент массива';  
3 $test[] = 'Еще новый элемент массива';  
4 echo $test[1]; //выведет test2  
5 isset($test[1]); //true  
6 unset($test[1]);  
7 isset($test[1]); //false
```



## ARRAYACCESS (Ч.4)

При этом мы можем определять иную хитрую логику внутри методов, которую захотим. Можем ничего не делать на `unset`. Можем дальше пользоваться всеми прелестями объектов (создать иные методы и обращаться к ним напрямую)

По большому счёту это просто небольшой лайфхак. Например, мы можем подобным образом делать работу с сессиями, при этом инициализируя отложено сессию, когда это будет нужно.





## ЕЩЕ О ВСТРОЕННЫХ КЛАССАХ И ИНТЕРФЕЙСАХ

Полный список - <http://php.net/manual/ru/reserved.interfaces.php>

Также, рекомендую обратить внимание на <http://php.net/manual/ru/class.reflectionclass.php> - в крупных проектах с хитрыми связями бывает полезно.

По большому счёту, это просто ещё немного магии языка PHP, которые расширяют возможности его использования. Но жить и без этого вполне себе можно (и даже нужно).



# ИСКЛЮЧЕНИЯ



## ЧТО ТАКОЕ ИСКЛЮЧЕНИЕ?

Исключение - один из способов работы с ошибками. В чем плюсы?

- Мы можем обрабатывать не только ошибки на уровне языка, но и ошибки на уровне логики
- Мы можем принимать решения, как действовать в случае ошибки, а не просто её игнорировать или выводить пользователю
- Мы можем обрабатывать сколько угодно ошибок самыми разными способами
- Исключение содержит цепочку вызовов функций и классов

# EXCEPTION

Исключения - это класс PHP Exception либо любой, который наследуется от него. Исключения можно "выбрасывать" и "ловить"

```
1 //Наш класс с исключениями
2 class MyException extends \Exception {}
3
4 try { //блок, который может выбрасывать исключения
5     $z = 10 / 0;
6 } catch (\Exception $e) { // отлавливаем исключения
7     echo 'Деление на ноль! ' . $e->getMessage;
8 }
```

## EXCEPTION (Ч.2)

Исключения можно выбрасывать принудительно

```
1 class MyException extends \Exception {}
2 function throwMyException() //функция просто выбрасывает
3 {
4     throw new MyException('Сработало исключение');
5 }
6
7 try {
8     throwMyException();
9 } catch (\MyException $e) {
10     echo 'Отловлено моё исключение, но ничего страшного';
11 }
```

## EXCEPTION (Ч.3)

Цепочка отлавливания может быть сколь угодно длинной

```
1  try {  
2      throwMyException();  
3      $z = 10 / 0;  
4  } catch (\MyException $e) {  
5      echo 'Отловлено моё исключение, но ничего страшного';  
6  } catch (\Exception $e) {  
7      echo 'Деление на ноль!';  
8      $z = 10; //исправляем сразу проблему  
9  }
```



## EXCEPTION (Ч.4)

Таким образом, с помощью исключений мы можем обработать любую ситуацию и сразу предложить для неё решение, при необходимости.

При этом в нашем распоряжении есть цепочка вызовов, по которой легче найти в какой момент произошла ошибка.

А ещё мы можем сделать так, чтобы все ошибки PHP по умолчанию были исключениями (и мы также могли их отлавливать)

## EXCEPTION (4.5)

```
1 function exception_error_handler($severity, $message, $file, $line) {
2     if (!(error_reporting() & $severity)) {
3         // Этот код ошибки не входит в error_reporting
4         return;
5     }
6     throw new ErrorException($message, 0, $severity, $file, $line);
7 }
8 //теперь все наши ошибки PHP будут выбрасывать исключения
9 set_error_handler("exception_error_handler");
```





## EXCEPTION (Ч.6)

На самом деле, исключения - отдельная большая тема. Я лишь хочу дать вам общий обзор

Если вы хотите научиться с ними работать более эффективно и узнать о всех возможностях - добро пожаловать в документацию

- <http://php.net/manual/ru/language.exceptions.php>
- <http://php.net/manual/ru/class.errorexception.php>
- <http://us1.php.net/manual/en/spl.exceptions.php>



**КОЕ-ЧТО ЕЩЁ**

## КОЕ ЧТО ЕЩЁ

Если вы хотите больше разобраться с встроенными интерфейсами PHP, можно обратить внимание на

<http://php.net/manual/ru/language.oop5.iterations.php>

Объект можно вызывать, как функцию.) Для этого обратите внимание на магический метод `__invoke`. Список всех магических методов -

<http://php.net/manual/ru/language.oop5.magic.php>

Если вы хотите знать, как я вас нагло обманул, сказав, что все объекты передаются по ссылке -

<http://php.net/manual/ru/language.oop5.references.php>



# ЧТО ПОЧИТАТЬ И ЧТО ДЕЛАТЬ ПОСЛЕ КУРСА?

Элизабет Фримен, Эрик Фримен - Паттерны проектирования

Много-много практики)

Обязательно ДЗ и Дипломная работа



Спасибо за внимание!

# ЕВГЕНИЙ КОРЫТОВ



[korytoff@gmail.com](mailto:korytoff@gmail.com)



[korytoff](https://www.telegram.me/korytoff)