ООП. Модуль 4. Функции и стандарты по работе с классами

В этом уроке мы рассмотрим встроенные функции для работы с классами и объектами.

Функции по работе с классами

Объекты в php можно создавать не только статически.

Т.е. просто указав внутри программы ключевое слово new и полное имя класса	

```php
// shop.php
namespace App\Shop;

class Order implements HasPrice
{
    use WithPrice;

    private $price;

    public function __construct($price = 0.0)
    {
        $this->price = $price;
    }

    public function test()
    {
        echo 'test' . PHP_EOL;
```

```
    }
}

trait WithPrice
{
    public function getPrice()
    {
        return $this->price;
    }
}

interface HasPrice
{
    public function getPrice();
}

// index.php
require_once 'shop.php';

$order = new \App\Shop\Order();
$order->test();
// test
```

Но и динамически, т.е. когда имя класса для объекта не указано непосредственно в коде программы, а вычисляется в процессе ее выполнения, и значение, например, хранится внутри переменной.

Например, можно указать имя класса, с помощью конфигурационных файлов.

Необходимо помнить, что при динамическом создании при определении полного имени класса - **не будет** участвовать текущее пространство имен. Т.е. при динамическом создании обязательно указывать **полное имя класса**.

```php
$config = [
    'order' => '\App\Shop\Order',
];

$order = new $config['order'];
$order->test();
// test
```

Но при таком создании необходимо убедиться, что такой класс действительно существует. Кроме того было бы хорошо узнать какие параметры принимает конструктор такого класса. Для этого в php предусмотрены специальный функции.

Функцию проверки существования класса

Существует несколько функций для проверки существования класса, интерфейса, трейта

class_exists($className) - возвращает true, если класс объявлен.

У нее есть аналоги **trait_exists()** и **interface_exists()**, которые соответственно проверяют: существует ли трейт, интерфейс

```php
if (! class_exists($config['order'])) {
    die('Class not exists');
}
$order = new $config['order'];
$order->test();
// test
```

Функции получения объявленных классов

Есть вспомогательные функции для получения списка всех объявленных классов в виде массива. Их также три разных функции для классов, трейтов и интерфейсов

get_declared_classes() И аналоги **get_declared_traits()** и **get_declared_interfaces()**	`print_p(get_declared_classes());`

Функции определения класса объекта

Следующие функции помогут определить полное имя класса, для переданного объекта

Функция get_class($object) принимает объект, а в качестве результата возвращает полное имя класса в виде строки.	`echo get_class($order);`
	`//`
	`App\Shop\Order`

У этой функции есть схожая, но только в качестве результата возвращает имя родительского класса, или false, если у класса нет родителей

Функция get_parent_class($object)	`var_dump(get_class($order));`
	`//`
	`bool(false)`

Эти функции можно было бы использовать для некоторых условий. Например у нас есть метод для вывода на экран объекта. При этом разные объекты должны быть выведены по-разному. То, например, если имя класса или базового класса равно “Table” - то метод бы вывел на экран каким-то образом таблицу, если “Image” - то изображение и т.п.

Но обычно для этого используется специальные оператор, которые проверяет не только сам класс, но также и его родителей. Этим же оператором можно проверять реализует ли класс тот или иной интерфейс.

Оператор instanceof.

Пример использования оператора instanceof	`if ($order instanceof \App\Shop\HasPrice) {`
	`  echo $order->getPrice() . PHP_EOL;`
	`} else {`
	`  $order->text();`
	`}`
	`// 0`

У этого оператора есть два метода - полных его аналога is_a() и is_subclass_of(), которые по-сути делают тоже самое что и этот оператор. Только как видно из их названия, второй метод будет рассматривать только дерево классов - родителей.

Т.е. такое условие будет ложно, т.к. объект является экземпляром этого класса, а не его наследником.

```php
is_subclass_of($order, \App\Shop\Order::class)
// false
```

Получение списка свойств и методов класса

Следующий метод в исследовании класса - это метод получения списка свойств класса get_class_vars()

В качестве параметра get_class_vars() принимает имя класса, и возвращает массив свойств, только каких… давайте посмотрим на результат выполнения функции в данном примере.

Кстати имя класса можно указывать не только в виде строки, но также указав имя класса, а затем через двоеточие указав ключевое слово class. Такая конструкция вернет полное имя класса. и является более предпочтительной чем строки. Во первых для построения полного имени класса будет рассчитываться с учетом пространств имен. Во вторых, это удобно в IDE редакторах.

```php
namespace App\Shop2;

class Order
{
    private $number;

    protected $price;

    public $products;

    private function getNumber()
    {
        //...
    }

    protected function getPrice()
    {
```

Вернемся к результату функции..

Он оказался не совсем очевидным. Как видно в результат попали только функции с публичным уровнем доступа.

```
        //...
    }

    public function getProducts()
    {
        //...
    }
}

print_r(get_class_vars(Order::class));
//
Array
(
    [products] =>
)
```

Если быть совсем точным, то функция возвращает видимые нестатические свойства указанного объекта в соответствии с областью видимости. Т.е. если вызвать эту функции внутри одного из методов нашего класса, то в результат попадут все свойства.

Например, вот так.

```
class Order
{
    //...
    public function get_class_vars()
    {
        print_r(get_class_vars(static::class));
    }
}
```

```php
$order = new Order();
$order->get_class_vars();
// Array
(
    [number] =>
    [price] =>
    [products] =>
)
```

Кроме получения списка видимых свойств класса, можно также получить список видимых методов класса, с помощью функцией **get_class_methods()**

При этом видимость учитывается также как и в случае со свойствами

```php
print_r(get_class_methods(Order::class));

// Array
(
    [0] => getProducts
    [1] => get_class_vars
)
```

Функции проверки существования свойств и методов

К методам и свойствам, тоже можно обращаться динамически. И также как и в случае с проверкой существования класса, нужно также предварительно проверять существование метода или свойства.

Для проверки существование метода или свойства используются функции: **method_exists(), property_exists()** - синтаксис функций одинаков, первым параметром передается экземпляр класса, вторым в виде строки название метода или свойства.

```php
var_dump(method_exists($order, 'getNumber'));
var_dump(property_exists($order, 'color'));
//
bool(true)
bool(false)
```

Но это не всегда, может помочь, потому что не проверяется уровень доступа к методу или свойству.

Такой пример, вызовет ошибку уровня доступа. Но проверка на существования выполнится успешно.

```php
$methodName = 'getPrice';

if (method_exists($order, $methodName)) {
   echo $order->{$methodName}();
}

//
HP Fatal error:  Uncaught Error: Call to
protected method
```

Это основные функции для исследования классов. Чаще всего на практике применяется оператор instanceof. А также функции проверки существования классов. Но эти методы и функции не позволяют более детально изучить методы и свойства класса. Например, нельзя получить список параметров, которые принимает метод. Для решения таких задач используются специальные классы \Reflection. Их вы можете изучить самостоятельно

Подгрузка классов. Стандарт psr-0 и psr-4

До текущего момент для подключения классов в нашем проекте нам приходилось самостоятельно писать include или require и помнить где же наши классы лежат. Но в php как и во многих других языках существует возможность создать автоподгрузку классов.

Для этого существует специальная функция spl_autoload_register(), которая принимает в качестве параметра функцию, которая будет вызвана, при попытке обращения к несуществующему классу

Создадим пример такой функции.

при создании нового объекта App\Shop\Order() - как раз и произойдет вызов нашей функции autoload().

$className - будет содержать полное имя вызываемого класса.

```php
function autoload($className)
{
    // как-то определяем $fileName и подключаем его
    require $fileName;
}

spl_autoload_register('autoload');

// Здесь произойдет обращение к несуществующему классу
new App\Shop\Order();
```

Внутри такой функции мы можем реализовать свою логику для подгрузки файлов с классами. Но если каждый программист будет придумывать свою реализацию такой загрузки классов - то при подключении

сторонней библиотеке каждая из них будет регистрировать свою функцию автозагрузки. Поэтому придумали стандарты автоподгрузки psr-0 и psr-4.

Psr-0

Psr-0 - это базовый формат автоподгрузки. Оба стандарта утверждают соответствие структуры пространства имен и имени класса структуре файла, содержащего этот класс в файловой системе.

1. Каждое имя класса должно соответствовать структуре:
   \<Vendor Name>\(<Namespace>\)*<Class Name>
   a. Vendor Name - Верхний уровень пространства имен
   b. Namespace - любой уровень вложенности пространства имен
   c. Class Name - имя класса
2. При сопоставлении файлу в директории на сервере каждое разделение пространства имен (символ \) - будет заменен на разделитель директорий.
3. Каждый символ подчеркивания “_” в имени Класса будет также заменен на разделитель директорий.
4. Последний неотделенный блок - будет именем файла, к нему добавится .php
5. При этом пространства имен и имя класса могут состоять из любой комбинации символов в верхнем или нижнем регистре

В документации в качестве разделителя директорий указан специальная константа php - DIRECTORY_SEPARATOR - в которой хранится, как ни странно, разделитель директорий, подходящий для операционной системы в которой сейчас выполняется php-скрипт.

Исходя из описания в пространству имен класса - будет соответствовать такая же структура директорий на сервере.

Примеры	

```
\myClass;
/myClass.php

\App\Shop\Order();
/App
   /Shop
       /Order.php

\Zend\some_package\Request();
/Zend
   /some_package
       /Request.php


\namespace\package\Class_Name();
/namespace
   /package
       /Class
           /Name.php
```

Вот так выглядит общепринятая функция для автоподгрузки в формате psr-0

```php
function autoload($className)
{
    $className = ltrim($className, '\\');
    $fileName  = '';
    $namespace = '';

    if ($lastNsPos = strrpos($className, '\\'))
{
        $namespace = substr($className, 0,
$lastNsPos);
        $className = substr($className,
$lastNsPos + 1);
        $fileName  = str_replace('\\',
DIRECTORY_SEPARATOR, $namespace) .
DIRECTORY_SEPARATOR;
    }

    $fileName .= str_replace('_',
DIRECTORY_SEPARATOR, $className) . '.php';

    require $fileName;
}

spl_autoload_register('autoload');
```

Psr-0 не слишком строгая функция для автоподгрузки, и оставлена в php для обратной совместимости с базовыми классами php, вместо этого стандарта рекомендуется использовать стандарт psr-4

Psr-4

Psr-4 более строгий и используемый стандарт автоподгрузки.

1. Правила именования относятся как классам, так и интерфейсам и трейтам и т.п. структурам языка.
2. Полное имя класса состоит из:
   - 2.1. \<NamespaceName>(\<SubNamespaceNames>)*\<ClassName>
     - NamespaceName - обязательно должно присутствовать верхний уровень пространства имен, также как и в предыдущем стандарте, может состоять из нескольких подпространств.
     - SubNamespaceNames - может быть любой уровень вложенности подпространств имен
     - ClassName - обязательное имя класса
   - 2.2. Символы подчеркивания, ни на что не влияют
   - 2.3. Пространство имен и имя класса могут состоять из любой комбинации символов в верхнем или нижнем регистре
   - 2.4. Все имена классов ДОЛЖНЫ быть использованы с соблюдением регистрочувствительности
3. При загрузке файла, соответствующего полностью определённому имени класса, используются следующие правила
   - 3.1. Последовательность из одного и более пространств и подпространств имён (не включая ведущий разделитель пространств имён) в полностью определённом имени класса (т.н. «префикс пространств имён») должна соответствовать хотя бы одному «базовому каталогу».
   - 3.2. Последовательность подпространств имён после «префикса пространства имён» соответствует подкаталогу в «базовом каталоге», при этом разделители пространств имён \ соответствуют

разделителям каталогов /. Имя подкаталога и имя подпространства имён ДОЛЖНЫ совпадать вплоть до регистра символов.

3.3. Имя класса, соответствует имени файла с расширением .php. Имя файла и имя класса ДОЛЖНЫ совпадать вплоть до регистра символов.

В этом стандарте появляется новое понятие - базовый каталог, он указывается отдельно при автозагрузке, и означает корневую директорию, которая будет соответствовать верхнему уровню пространства имен. Затем также как и в предыдущем стандарте, последующая структура подпространств имен должна сопоставляться структуре директорий внутри базового каталога.

Примеры

Полностью определённое имя класса	**Префикс пространства имён**	**Базовый каталог**	**Итоговый путь к файлу**
\Acme\Log\Writer\File_Writer	Acme\Log\Writer	./acme-log-writer/lib/	./acme-log-writer/lib/File_Writer.php
\Aura\Web\Response\Status	Aura\Web	/path/to/aura-web/src/	/path/to/aura-web/src/Response/Status.php
\Symfony\Core\Request	Symfony\Core	./vendor/Symfony/Core/	./vendor/Symfony/Core/Request.php
\Zend\Acl	Zend	/usr/includes/Zend/	/usr/includes/Zend/Acl.php

Желтым выделены базовые пространства имен, синим - базовый базовый каталог, который соответствует этим пространствам имен, а оставшаяся часть собирается автоматически по правилам подгрузки.

Вот так выглядит общепринятая функция для автоподгрузки в формате psr-4

В ней явно указано базовая директория и базовое пространство имен, которое ей соответствует

```
spl_autoload_register(function ($class) {

    // project-specific namespace prefix
    $prefix = 'Foo\\Bar\\';

    // base directory for the namespace prefix
    $base_dir = __DIR__ . '/src/';

    // does the class use the namespace prefix?
    $len = strlen($prefix);
    if (strncmp($prefix, $class, $len) !== 0) {
        // no, move to the next registered
autoloader
        return;
    }

    // get the relative class name
    $relative_class = substr($class, $len);

    // replace the namespace prefix with the
base directory, replace namespace
    // separators with directory separators in
the relative class name, append
    // with .php
    $file = $base_dir . str_replace('\\', '/',
```

```
$relative_class) . '.php';

    // if the file exists, require it
    if (file_exists($file)) {
        require $file;
    }
});
```