Технологии разработки Web-приложений
PHP. ООП

Объе́ктно-ориенти́рованное программи́рование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования

Принципы ООП:

- **Наследование**
- **Абстракция**
- **Инкапсуляция**
- **Полиморфизм**

Синтаксис

Класс

```php
<?php
class SimpleClass
{
    // объявление свойств
    public $var = 'значение по умолчанию';
    private $secret = 'secret';//доступно только внутри класса

    // объявление метода
    public function displayVar() {
        echo $this->var;
    }
}
```

Именем класса может быть любое слово, если

- *оно не входит в список зарезервированных слов PHP,*
- *начинается с буквы или символа подчёркивания,*
- *состоит из букв, цифр или символов подчёркивания*

Класс. Типизация

Используйте типизацию, она избавляет от лишних проверок в коде

```php
<?php
class SimpleClass
{
    public string $name; //доступно с PHP 7.4

    //доступно с PHP 7.1
    public function setName(?string $name): void
    {
        $this->name = $name;
    }
}
```

```php
<?php
$instance = new SimpleClass();
 ?>
```

Области видимости

- **public** — свойства или методы доступны в любом месте.
- **protected** — свойства и методы доступны внутри класса и в дочерних классах.
- **private** — доступ к свойствам и методам имеет только класс, в котором эти свойства или методы объявлены

Переменная $this

Переменная $this нужна, чтобы обратиться к свойству или методу внутри того же класса

```php
<?php
class User
{
    public $name;
    public $age;

    public function show()
    {
        return $this->name; // обращение к свойству
    }

    public function setName($name)
    {
        $this->name = $name; // запись в свойство
    }
}
```

Конструктор - метод, вызывающийся при создании объекта.

Это полезно, например, для инициализации состояния объекта перед его использованием.

Как и обычные методы, **конструкторы могут иметь произвольное количество аргументов**.

Конструктор

```php
<?php
class User
{
    public $name;
    public $age;

    public function __construct(string $name, int $age)
    {
        $this->name = $name; // запишем данные в свойство name
        $this->age = $age; // запишем данные в свойство age
    }
}

$user = new User('Дарья', 30); // создадим объект, сразу заполнив его данными

echo $user->name; // выведет 'Дарья'
echo $user->age; // выведет 30
```

Деструктор

```php
<?php
class MyDestructableClass
{
   function __destruct() {
       print "Уничтожается " . __CLASS__ . "\n";
   }
}
```

Деструктор будет вызван

- при освобождении всех ссылок на определённый объект
- при завершении скрипта (в т.ч. вызове exit())

На практике почти не используется.

Статические свойства и методы

Статические свойства/методы для своего вызова **не требуют создания объекта**.

Чтобы объявить метод статическим, нужно после модификатора доступа (public, private или protected) написать **ключевое слово static**.

Статические методы **принадлежат не какому-то объекту класса, а самому классу**, поэтому **внутри статических методов нельзя использовать `$this`.**

Для обращения к статическим свойствам/методам **внутри класса** используем **self::**

Статические свойства и методы

```php
<?php
class Math
{
    private static $pi = 3.14;

    public static function getSum($a, $b)
    {
        return $a + $b;
    }

    public static function getProduct($a, $b)
    {
        return $a * $b;
    }

    public static function getCircleSquare($radius)
    {
        return self::$pi * $radius * $radius;
    }
}
```

```php
echo Math::getSum(1, 2) + Math::getProduct(3, 4);

echo Math::getCircleSquare(10);
```

Константы

```php
<?php
class MyClass
{
    const CONSTANT = 'значение константы';

    function showConstant() {
        echo  self::CONSTANT;
    }
}

echo MyClass::CONSTANT;
```

Константы задаются для всего класса, а не отдельно для каждого созданного объекта (как статические свойства).

Константы нельзя изменять.

Пространства имен

Пространства имён PHP позволяют группировать логически связанные классы, интерфейсы, функции и константы.

Решают проблемы:

- Конфликт имён между вашим кодом и внутренними классами/функциями/константами PHP или сторонними.

- Возможность создавать псевдонимы (или сокращения) для Ну_Очень_Длинных_Имён, чтобы облегчить первую проблему и улучшить читаемость исходного кода.

Пример

```php
<?php
namespace Admin;

class Page
{

}
```

```php
<?php
namespace Users;

class Page
{

}
```

```php
<?php
require_once '/admin/page.php';
require_once '/users/page.php';

$adminPage = new \Admin\Page();

$usersPage = new \Users\Page();
```

Пространства имен. Use

Конструкция **use** позволяет подключить класс по его полному имени.

После этого можно будет обращаться к этому классу просто по имени класса.

Можно использовать псевдонимы для классов с помощью конструкции **as**.

```php
<?php
namespace \Core\Admin;

class Data
{
    public function __construct($num)
    {

    }

 }
```

```php
<?php
namespace Users;
use \Core\Admin\Data; // подключаем класс
use \Core\User\Data as UserData; // подключаем класс

class Page extends Controller
{
    public function __construct()
    {
        $data1 = new Data('1');
        $data2 = new UserData('2');
    }

  }
```

Наследование

```php
<?php
class Person
{
    private $name;
    private $age;

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

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

    public function getAge()
    {
        return $this->age;
    }

    public function setAge($age)
    {
        $this->age = $age;
    }

}
```

```php
<?php
class Student extends Person
{
    private $university;

    public function getUniversity()
    {
        return $this->university;
    }

    public function
setUniversity($university)
    {
        $this->university = $university;
    }
}
```

Перезапись конструктора родительского класса

Если в дочернем классе не определён конструктор, то он может быть унаследован от родительского класса как обычный метод.

Если дочерний класс определяет собственный конструктор:

Чтобы вызвать конструктор, объявленный в родительском классе, требуется вызвать parent::__construct() внутри конструктора дочернего класса.

```php
<?php
class Person
{
    private $name;
    private $age;

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

```php
<?php
class Student extends Person
{
    private $university;
    public function __construct($name, $age,
$university)
    {
        parent::__construct($name, $age);

        $this->university = $university;
    }
}
```

Множественное наследование

Множественного наследования в php нет.

Чтобы обойти это ограничение, вы можете использовать трейты

Трейт - набор свойств и методов, которые можно включить в другой класс. При этом свойства и методы трейта будут восприниматься классом будто свои.

Экземпляр трейта нельзя создать - трейты предназначены только для подключения к другим классам.

```php
trait Hello {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait World {
    public function sayWorld() {
        echo 'World';
    }
}

class MyHelloWorld {
    use Hello, World;
    public function sayExclamationMark() {
        echo '!';
    }
}
```

```php
$o = new MyHelloWorld();
$o->sayHello();
$o->sayWorld();

$o->sayExclamationMark();
```

Ключевое слово final

Ключевое слово **final** перед объявлениями методов или констант класса, запрещает их переопределение в дочерних классах.

Если сам класс определяется с этим ключевым словом, то он не сможет быть унаследован.

```php
<?php
final class BaseClass {
   public function test() {
       echo "Вызван метод BaseClass::test()\n" ;
   }

   // Поскольку класс уже является final, ключевое слово final является избыточным
   final public function moreTesting() {
       echo "BaseClass::moreTesting() called\n" ;
   }
}

class ChildClass extends BaseClass {
}
// Выполнение заканчивается фатальной ошибкой:
// Class ChildClass may not inherit from final class (BaseClass)
```

Абстрактные классы и методы

Абстрактные классы предназначены для наследования от них. При этом объекты таких классов нельзя создать.

Абстрактные классы также могут содержать **абстрактные методы**.

Такие методы не должны иметь реализации, а нужны для того, чтобы указать, что такие методы должны быть у потомков.

А собственно реализация таких методов - уже задача потомков.

Абстрактные классы

```php
<?php
abstract class User
{
    private $name;

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

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

```php
<?php
class Employee extends User
{
    private $salary;

    public function getSalary()
    {
        return $this->salary;
    }

    public function setSalary($salary)
    {
        $this->salary = $salary;
    }

}
```

```php
$user = new User(); // выдаст ошибку

//будет работать
$employee = new Employee;
$employee->setName('Иван'); // метод родителя, т.е. класса User

$employee->setSalary(200000); // свой метод, т.е. класса Employee
```

Абстрактные методы

```php
<?php
abstract class User
{
    private $name;

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

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

    // Абстрактный метод без тела:
    abstract public function
           increaseRevenue($value);

 }
```

```php
<?php
class Employee extends User
{
    private $salary;

    public function getSalary()
    {
        return $this->salary;
    }

    public function setSalary($salary)
    {
        $this->salary = $salary;
    }


    // Напишем реализацию абстрактного метода:
    public function increaseRevenue($value)
    {
        $this->salary = $this->salary + $value;
    }

 }
```

Интерфейсы

Интерфейс - набор публичных методов класса, обязательных для реализации (без самой реализации).

Это нужно, чтобы при программировании совершать меньше ошибок.

Описав все необходимые методы в классе-родителе, мы можем быть уверены в том, что все потомки их действительно реализуют.

Каждый класс может реализовывать любое количество интерфейсов.

Интерфейсы, могут наследоваться друг от друга с помощью оператора extends.

```php
<?php
interface iMath
{
   // сложить
    public function sum($a, $b);

   // вычесть
    public function subtract($a, $b);

    // умножить
    public function multiply($a, $b);

     // разделить
    public function divide($a, $b);

 }
```

```php
<?php
class Math implements iMath
{
    public function sum($a, $b)
    {
        return $a + $b;
    }

    public function subtract($a, $b)
    {
        return $a - $b;
    }

    public function multiply($a, $b)
    {
        return $a * $b;
    }

    public function divide($a, $b)
    {
        return $a / $b;
    }

}
```

Особенности, отличия от других языков

Динамический вызов функций

PHP поддерживает концепцию переменных функций. Это означает, что если к имени переменной присоединены круглые скобки, PHP ищет функцию с тем же именем, что и результат вычисления переменной, и пытается её выполнить.

* Объявлено устаревшим начиная с php 8.2
https://www.php.net/manual/ru/migration82.deprecated.php

```php
<?php
function foo() {
    echo "Вызвана функция foo()";
}

$func = 'foo';
$func();        // Вызывает функцию foo()
```

Динамический вызов свойств и методов

Свойства и методы классов тоже можно вызывать динамически

Пример 1

```php
<?php
class User
{
    public $name;
    public $age;

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

$user = new User('Дарья', 30);
$prop = 'name';
echo $user->$prop; // выведет 'Дарья'
```

Динамический вызов свойств и методов

Пример 2

```php
<?php
class User
{
    public $surname; // фамилия
    public $name; // имя
    public $patronymic; // отчество

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

$user = new User('Иванов', 'Иван', 'Иванович');
$props = ['surname', 'name', 'patronymic'];
echo $user->{$props[0]}; // выведет 'Иванов'
```

Динамический вызов свойств и методов

Пример 3

```php
<?php
class User
{
    private $name;
    private $age;

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

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

    public function getAge()
    {
        return $this->age;
    }
}
```

```php
<?php

$user = new User('Дарья', 30);


$method = 'getName';

echo $user->$method(); // выведет 'Дарья'
```

Перегрузка. Магические методы

Перегрузка в PHP - возможность динамически создавать свойства и методы класса.

Методы перегрузки вызываются при обращении к свойствам или методам, которые не были объявлены или не видны в текущей области видимости.

Для работы с ними используются **“магические методы”**.

https://www.php.net/manual/ru/language.oop5.magic.php

Магические методы __get(), __set()

```
public __set(string $name, mixed $value): void
```

будет выполнен при записи данных в недоступные (защищённые или приватные) или несуществующие свойства.

```
public __get(string $name): mixed
```

будет выполнен при чтении данных из недоступных или несуществующих свойств

```
public __call(string $name, array $arguments): mixed
```

будет выполнен при вызове недоступных методов

```
public static __callStatic(string $name, array $arguments):
mixed
```

будет выполнен при вызове недоступных статических методов

Магические методы __get(), __set(), __call()

Пример 1

```php
<?php
class Test
{
    public function __set($property, $value)
    {
        // устанавливаем значение
        $this->$property = $value;
    }

    // Магический геттер свойств:
    public function __get($property)
    {
        return $this->$property;
    }

    public function __call($arg) {
        //…
    }
}
```

```php
<?php
$test = new Test();

$test->prop1 = 1; // запишем 1
$test->prop2 = 2; // запишем 2

echo $test->prop1; // выведет 1
echo $test->prop2; // выведет 2

$test->myMethod();
```

stdClass

stdClass - общий пустой класс в PHP (похож на Object в Java или object в Python). Объекты этого класса могут наполняться любыми свойствами.

```php
<?php

$obj = new stdClass();

$obj->key = 'value';


var_dump($obj);
```

```php
<?php

$userAsArray = [

  'name' => 'George',

  'age' => 18

];


$userAsObject = (object) $userAsArray;


var_dump($userAsObject);
// class stdClass#2 (2) {
//   public $name =>
//   string(6) "George"
//   public $age =>
//   int(18)
// }
```

Другие магические методы

```
public __serialize(): array
```

```
public __unserialize(): void
```

Метод **__serialize()** срабатывает при сериализации объекта. Полезен для определении удобного для сериализации произвольного представления объекта.

Метод **__unserialize**() срабатывает при вызове **unserialize()** (восстановлении сериализованного значения в объект)**.**

```
public __toString(): string
```

позволяет классу решать, как он должен реагировать при преобразовании в строку. Например, что вывести при выполнении `echo` $obj;

```
static __set_state(array $properties): object
```

Определяет, что вернет функция var_export() для объекта этого класса

https://www.php.net/manual/ru/language.oop5.magic.php

Позднее статическое связывание. Отличие self:: и static::

Позднее статическое связывание

позволяет объектам все также наследовать методы у родительских классов, но помимо этого дает возможность унаследованным методам иметь доступ к статическим константам, методам и свойствам класса потомка, а не только родительского класса.

Позднее статическое связывание. Отличие self:: и static::

```php
<?php
class BaseEntity
{
    public static function getTable()
    {
        return self::$table;
    }

    public function save()
    {
        // Какой-то код
        $table = self::getTable();
        // Дальнейшая обработка
    }
}

class User extends BaseEntity
{
    protected static $table = 'users';
}

echo User::getTable(); // Error

$user = new User();
$user->save(); // Error
```

```php
<?php
class BaseEntity
{
    public static function getTable()
    {
        return static::$table;
    }

    public function save()
    {
        // Какой-то код
        $table = self::getTable();
        // Дальнейшая обработка
    }
}

class User extends BaseEntity
{
    protected static $table = 'users';
}

echo User::getTable(); // Выведет users

$user = new User();
$user->save(); // отработает корректно
```

Reflection API

Reflection API дает возможность получать **информацию о внутренней структуре классов, интерфейсов, функций, методов и модулей.**

Кроме того, Reflection API позволяет получать doc-блоки комментариев функций, классов и методов.

Механизм рефлексии **широко используется генераторами документаций, а так же в фреймворках для конфигурирования роутингов, параметров сериализации, настройки прав доступа и т.д.**

https://www.php.net/manual/ru/book.reflection.php

Пример использования https://symfony.com/doc/current/doctrine.html#creating-an-entity-class

Reflection API

```php
/**
 * Class Profile
 */
class Profile {
    /**
     * @return string
     */
    public function getUserName(): string
    {
        return 'Foo';
    }

}
```

```php
// получаем объект
$reflectionClass = new
ReflectionClass('Profile');

// получаем имя класса
var_dump($reflectionClass->getName());
=> output: string(7) "Profile"

// получаем комментарии
var_dump($reflectionClass->getDocComment());
=> output:
string(24) "/**
 * Class Profile

   */"
```

Composer.
Полезные пакеты.
Автозагрузка классов

Composer - это **менеджер зависимостей для PHP.** Вы можете описать от каких библиотек зависит ваш проект и Composer установит нужные библиотеки за вас.

https://packagist.org/

При установке php пакетов Composer заодно устанавливает все зависимости, от которых эти пакеты зависят.

Загрузку сторонних библиотек Composer выполняет в папку vendor в корневой директории проекта.

Также composer создаёт специальный файл **autoload.php**. Если вы подключите этот файл в проекте, вы сразу сможете использовать все загруженные библиотеки.

```
$ composer require monolog/monolog
```

composer.json # описание основных пакетов, включая требования к их версиям.

```
{
    "require": {
        "monolog/monolog": "^2.3"
    }
}
```

composer.lock # реальные версии пакетов, которые были установлены на компьютер пользователя

пример

https://gist.github.com/daria-popova/65440ce2b101d0e6428d599fab338b28

В папке `vendor` располагаются исходники библиотек и файл `autoload.php`. Вы можете подключить `autoload.php` и начать использовать классы, которые эти библиотеки предоставляют:

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

$log = new Monolog\Logger('name');
$log->pushHandler(new Monolog\Handler\StreamHandler('app.log', Monolog\Logger::WARNING));

$log->addWarning('Foo');
```

Автозагрузка. Стандарт PSR-4

Стандарт **PSR-4** (https://www.php-fig.org/psr/psr-4/) описывает, каким образом загрузчик будет искать класс с определенным неймспейсом среди файлов

В таблице ниже представлены примеры соответствий полностью определённого имени класса, префикса пространства имён, базового каталога и итогового пути к файлу.

Fully Qualified Class Name	Namespace Prefix	Base Directory	Resulting File Path
\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

Автозагрузка с помощью composer

Вы даже можете добавить свой код в автозагрузчик, добавив поле autoload в **`composer.json`**

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

После добавления поля `autoload` в `composer.json` необходимо повторно выполнить команду **`dump-autoload`** для повторной генерации файла `vendor/autoload.php`

Composer зарегистрирует автозагрузчик PSR-4 для пространства имен `App`.

Все ваши классы из папки **src** будут доступны в любом месте кода (без require).