ООП (объектно-ориентированное программирование)

Классы и объекты
Конструктор и деструктор
Наследование
Константы классов
Область видимости
Интерфейсы
Абстрактные классы
Инкапсуляция
Полиморфизм
Ключевое слово "final"

Классы и объекты

ООП **- (объектно-ориентированное программирование) парадигма программирования, в которой основными концепциями являются понятия объектов и классов.**

Класс **- это способ описания сущности, и является своего рода чертежом. Определение класса начинается с ключевого слова class**, затем следует имя класса, и далее пара фигурных скобок, которые заключают в себе определение свойств и методов этого класса. Именем класса может быть любое слово, при условии, что оно не входит в список зарезервированных слов PHP, начинается с буквы или символа подчеркивания и за которым следует любое количество букв, цифр или символов подчеркивания:

```php
class Human
{
    public $name;

    public function talk()
    {
        echo 'Привет, меня зовут - '. $this->name;
    }

    public function walk($steps = 2)
    {
        echo str_repeat($this->name .' делает шаг...<br/>', $steps);
    }

}
```

Класс может иметь *свойства* - своего рода переменные, в нашем примере это *$name*. А так же *методы* - своего рода функции в нашем примере это *talk(), walk()*.

public - ключевое слово области видимости (всего их три *public, protected* или *private*). Пока достаточно знать что они есть, чуть позже будет понятно что это и зачем.

Объект **- это конкретный экземпляр класса. Т.е. если класс - Human (человек), то объект - это конкретный человек, например Иванов Петр Сергеевич. Для создания объекта используется директива new.** Пустые круглые скобки можно опустить:

```php
$vasya = new Human();
$petya = new Human;
```

После создания объекта, он получает доступные свойства и методы объявленные в родительском классе. Чтобы задать или получить значение свойства объекта, используется следующий синтаксис:

```
// Задаем значения
$vasya->name = 'Вася';
$petya->name = 'Петя';
// Выводим значения
echo $vasya->name; // Вася
echo $petya->name; // Петя
```

Точно так же и при вызове метода:

```
$vasya->talk(); // Привет, меня зовут - Вася
$petya->walk(1); // Петя делает шаг...
```

Псевдо-переменная **$this** доступна в том случае, если метод был вызван в контексте объекта. **$this** является ссылкой на вызываемый объект. Именно поэтому в методы подставляются свойства *$name* вызванного объекта и мы видем имена *Вася* и *Петя*, соответственно. Точно так же можно вызывать методы объекта.

```
public function talk()
{
    echo 'Привет, меня зовут - '. $this->name;
}

public function walk($steps = 2)
{
    $this->talk();
    echo str_repeat($this->name .' делает шаг...<br/>', $steps);
}
```

Конструктор и деструктор

Было бы здорово при создании нашего объекта сразу производить какие-то действия, например, передавать имя человека. PHP 5 позволяет объявлять методы-конструкторы **__construct()**. Классы, в которых объявлен метод-конструктор, будут вызывать этот метод при каждом создании нового объекта. Добавим конструктор в наш класс, который будет принимать имя человека.

```
class Human
{
    public $name;

    public function __construct($name = 'Аноним')
    {
        $this->name = $name;
    }

    public function talk()
    {
        echo 'Привет, меня зовут - '. $this->name;
    }

    public function walk($steps = 2)
    {
        echo str_repeat($this->name .' делает шаг...<br/>', $steps);
    }

}

$vasya = new Human('Вася');
$petya = new Human('Петя');
```

Параметры в конструктор передаются при создании объекта в те самые круглые скобки.

В php4 конструктором был метод совпадающий с именем класса. С 5 версии совместимость осталась но в приоритете __construct(). Начиная с версии PHP 5.3.3, методы с именами, совпадающими с последним элементом имени класса, находящимся в пространстве имен, больше не будут считаться конструкторами. Это изменение не влияет на классы, не находящиеся в пространстве имен.

Так же есть метод обратный конструктору, называемый - деструктором **__destruct()**. Деструктор будет вызван при освобождении всех ссылок на определенный объект или

при завершении скрипта (*порядок выполнения деструкторов не гарантируется*). Появился в версии php 5.

```php
public function __destruct()
{
    echo 'Это конец...';
}
```

Деструктор будет вызван даже в том случае, если скрипт был остановлен с помощью функции exit(). Вызов exit() в деструкторе предотвратит запуск всех последующих функций завершения.

Наследование

Наследование - позволяет описать новый класс на основе уже существующего. Класс, от которого производится наследование, называется базовым, родительским или супер-классом. Новый класс – потомком, наследником или производным классом. Чтобы наследовать класс используется ключевое слово **extends**, например создадим классы Man (мужчина) и Woman (женщина) и унаследуем их от Human (человек). Тогда наши классы получат свойства и методы базового класса.

```php
class Man extends Human
{
    public $gender = 'мужчина';
}
class Woman extends Human
{
    public $gender = 'женщина';
}

$vasya = new Man('Вася');
$yulya = new Woman('Юля');
// Выводим значения
echo $vasya->name; // Вася
echo $yulya->name; // Юля
echo $vasya->gender; // мужчина
echo $yulya->gender; // женщина
// Вызов методов
$vasya->talk(); // Привет, меня зовут - Вася
$yulya->walk(1); // Юля делает шаг...
```

Перегрузка методов позволяет перекрыть работу родительского метода в дочернем. Например, при вызове метода talk() мужчина хочет добавлять “Я - мужчина”. Для этого объявим точно такой же метод в классе Man. А если нам надо чтобы еще отработал родительский метод то это делается с помощью **parent::method_name()**:

```php
class Man extends Human
{
    public $gender = 'мужчина';

    public function talk()
    {
        // вызываем метод talk() родительского класса Human
        parent::talk();
        // выводим пол
        echo ' Я '. $this->gender;
    }
}
```

Константы классов

Константы также могут быть объявлены и в пределах одного класса. Отличие переменных и констант состоит в том, что при объявлении последних или при обращении к ним не используется символ $. Негласное правило - имя константы писать прописными буквами, если несколько слов разделять нижними подчеркиваниями. Значение должно быть неизменяемым выражением, не переменной, свойством, результатом математической операции или вызовом функции. Константа принадлежит классу а не к объекту. Объявляется с помощью const:

```php
class DbConfig
{
    const DB_NAME = 'test_db';

    public function getDbName()
    {
        return self::DB_NAME;
    }
}

$config = new DbConfig();
echo $config->getDbName(); // test_db
echo DbConfig::DB_NAME; // test_db
```

Область видимости

Область видимости свойства или метода может быть определена путем использования следующих ключевых слов в объявлении: public, protected или private. Доступ к свойствам и методам класса, объявленным как *public* (общедоступный), разрешен отовсюду. Модификатор *protected* (защищенный) разрешает доступ наследуемым и родительским классам. Модификатор *private* (закрытый) ограничивает область видимости так, что только класс, где объявлен сам элемент, имеет к нему доступ. Давайте немного перепишем наш класс Human.

```php
class Human
{
    public $age = 0;
    protected $name;
    private $type = 'человек';

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

    public function talk()
    {
        echo 'Привет, меня зовут - '. $this->name;
    }

    public function walk($steps = 2)
    {
        echo str_repeat($this->name .' делает шаг...<br/>', $steps);
    }

    public function getType()
    {
        return $this->type;
    }

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

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

Теперь попробуем получить доступы в классе наследнике:

```php
class Woman extends Human
{
    public $gender = 'женщина';

    public function test()
    {
        // Свойства
        $this->age; // Доступно (public)
        $this->name; // Доступно (potected)
        $this->type; // НЕ доступно (private) ОШИБКА
        // Методы
        $this->getType(); // Доступно (public)
        $this->getName(); // Доступно (potected)
        $this->setName('Юлия'); // НЕ доступно (private) ОШИБКА
    }
}
```

Теперь попробуем получить доступы в объекте:

```php
$yulya = new Woman('Юля');
// Свойства
$yulya->age; // Доступно (public)
$yulya->name; // НЕ доступно (potected) ОШИБКА
$yulya->type; // НЕ доступно (private) ОШИБКА
// Методы
$yulya->getType(); // Доступно (public)
$yulya->getName(); // НЕ доступно (potected) ОШИБКА
$yulya->setName('Юлия'); // НЕ доступно (private) ОШИБКА
```

Интерфейсы

Интерфейсы объектов позволяют создавать код, который указывает, какие методы и свойства должен включать класс, без необходимости описывания их функционала. Интерфейсы объявляются так же, как и обычные классы, но с использованием ключевого слова interface. Тела методов интерфейсов должны быть пустыми, а область видимости - *public*.

```
interface Moneymaker
{
    public function makeMoney();
    public function spendMoney($money);
}
```

Для реализации интерфейса используется оператор **implements**. Класс должен реализовать все методы, описанные в интерфейсе; иначе произойдет фатальная ошибка.

```
// Класс тракторист реализует интерфейс Moneymaker
class TractorDriver implements Moneymaker
{
    public function makeMoney()
    {
        echo 'Заработал 100 рублей<br>';
    }

    public function spendMoney($money)
    {
        echo "Потратил $money рублей<br>";
    }
}

$tractor_driver = new TractorDriver();
$tractor_driver->makeMoney(); // Заработал 100 рублей
$tractor_driver->spendMoney(50); // Потратил 50 рублей
```

Сигнатуры методов в классе, реализующем интерфейс, должны точно совпадать с сигнатурами, используемыми в интерфейсе, в противном случае будет вызвана фатальная ошибка.

При желании классы могут реализовывать более одного интерфейса за раз, реализуемые интерфейсы должны разделяться запятой. Создадим еще один интерфейс

```php
interface Educable
{
    public function acquireKnowledge($knowledge);
}
```

Добавим новый класс который и реализует два наших интерфейса

```php
// Класс программист реализует 2 интерфейса
class Programmer implements Moneymaker, Educable
{
    public function makeMoney()
    {
        echo 'Заработал 1000 рублей<br>';
    }

    public function spendMoney($money)
    {
        echo "Потратил $money рублей<br>";
    }

    public function acquireKnowledge($knowledge)
    {
        echo "Получил новые знания - $knowledge<br>";
    }
}

$programmer = new Programmer();
$programmer->makeMoney(); // Заработал 1000 рублей
$programmer->spendMoney(500); // Потратил 500 рублей
$programmer->acquireKnowledge('PHP'); // Получил новые знания - PHP
```

Интерфейсы могут быть унаследованы друг от друга, так же как и классы, с помощью оператора *extends*.

```php
// Интерфейс наследуется от другого интерфейса
interface Marketer extends Moneymaker
{
    public function sellGoods();
}
```

```php
// Класс продавец тоже реализует 2 интерфейса
class Seller implements Marketer
{
    public function makeMoney()
    {
        echo 'Заработал 500 рублей<br>';
    }

    public function spendMoney($money)
    {
        echo "Потратил $money рублей<br>";
    }

    public function sellGoods()
    {
        echo "Продал товар<br>";
    }
}

$seller = new Seller();
$seller->makeMoney(); // Заработал 500 рублей
$seller->spendMoney(250); // Потратил 250 рублей
$seller->sellGoods(); // Продал товар
```

Интерфейсы могут содержать константы. Константы интерфейсов работают точно так же, как и константы классов, за исключением того, что они не могут быть перекрыты наследующим классом или интерфейсом. Так же интерфейсам доступно множественное наследование. Некоторые начинают название интерфейса с префикса “**I_**”

```php
interface I_Healing extends Educable, Moneymaker
{
    const HIPPOCRATIC_OATH = 'Клятва Гиппократа';
}
```

```php
// Доктор наследует 3 интерфейса
class Doctor implements I_Healing
{
    public function acquireKnowledge($knowledge)
    {
        echo "Получил новые знания - $knowledge<br>";
    }

    public function makeMoney()
    {
        echo 'Заработал 700 рублей<br>';
    }

    public function spendMoney($money)
    {
        echo "Потратил $money рублей<br>";
    }
}
```

```php
$doctor = new Doctor();
$doctor->makeMoney(); // Заработал 700 рублей
$doctor->spendMoney(350); // Потратил 350 рублей
$doctor->acquireKnowledge('о болезни'); // Получил новые знания - о болезни
echo I_Healing::HIPPOCRATIC_OATH; // Клятва Гиппократа
```

Абстрактные классы

Абстрактные классы. PHP поддерживает определение абстрактных классов и методов. Методы, объявленные абстрактными, несут, по существу, лишь описательный смысл и не могут включать реализации (так же как в интерфейсе). В отличии от интерфейсов абстрактные методы могут иметь разные области видимости.

Класс, который содержит по крайней мере один абстрактный метод, должен быть определен как абстрактный. В то время как абстрактный класс не обязательно должен содержать абстрактные методы.

Например создадим абстрактный класс A_Employee (работник) который реализует общую часть для всех классов и имеет абстрактный метод getMoney(). Некоторые начинают название абстрактного класса с префикса “**A_**”

```php
abstract class A_Employee
{
    abstract public function getMoney();

    public function makeMoney()
    {
        echo 'Заработал '. $this->getMoney() .' рублей<br>';
    }

    public function spendMoney($money)
    {
        echo "Потратил $money рублей<br>";
    }
}
```

Теперь создадим класс наследник

```php
// Класс учитель наследует абстрактный класс
class Teacher extends A_Employee
{
    // Обязательная реализация getMoney()
    public function getMoney()
    {
        return 300;
    }
}

$teacher = new Teacher();
$teacher->makeMoney(); // Заработал 300 рублей
$teacher->spendMoney(150); // Потратил 150 рублей
```

При наследовании от абстрактного класса, все методы, помеченные абстрактными в родительском классе, должны быть определены в классе-потомке; кроме того, область видимости этих методов должна совпадать (или быть менее строгой). Например, если абстрактный метод объявлен как protected, то реализация этого метода должна быть либо protected либо public, но никак не private.

```php
abstract class A_Car
{
    abstract protected function drive();
}
```

```php
class Lada extends A_Car
{
    // public protected можно, private НЕЛЬЗЯ
    public function drive()
    {
        echo 'Lada не едет, а летит!';
    }
}
```

Сигнатуры методов должны совпадать, т.е. контроль типов (type hint) и количество обязательных аргументов должно быть одинаковым. К примеру, если в дочернем классе указан необязательный параметр, которого нет в сигнатуре абстрактного класса, то в данном случае конфликта сигнатур не будет. Это правило также применяется к

конструкторам начиная с версии PHP 5.4, ранее сигнатуры конструкторов могли отличаться.

```php
abstract class A_Text
{
    abstract public function showText($str);
}
```

```php
class TextEditor extends A_Text
{
    public function showText($str, $prefix = '')
    {
        echo '<p>'. $prefix . $str .'</p>';
    }
}

$editor = new TextEditor();
$editor->showText('Тест текста', 'Каламбур: ');
```

Инкапсуляция

Инкапсуляция это защита внутренних данных класса от внешнего по отношению к данному классу коду и сокрытие внутренней реализации.
Представьте мы делаем механизм оплаты для банка. У класса Account есть свойство balance. Если оно будет public и доступно для изменений, то это может привести к плачевным последствиям. Поэтому лучше сделать её private, и реализовать доступы через публичные методы getBalance() - возвращает текущий баланс, а addMoney() - добавляет средства. При выводе баланса использовались рубли, но руководство банка решило что теперь лучше выводить баланс в евро. Если просто выводить свойство balance то пришлось бы в каждом месте менять вывод, а так мы внесем изменения только в метод getBalance(), а остальные части останутся без изменений. Или при добавлении денежных средств надо изменить процент комиссии, сделав это в методе addMoney(). Инкапсуляция внутренних данных и реализации методов позволяет объектно-ориентированным программным системам защищать свои данные и управлять доступом к ним, а также скрывать детали реализации, создавая, таким образом, гибкие и стабильные приложения.

```php
class Account
{
    private $_balance = 0; // Баланс в рублях

    public function getBalance()
    {
        // Возвращаем баланс переведенный в евро (61 грубое значение)
        return $this->_balance / 61;
    }

    public function addMoney($sum)
    {
        $sum = floatval($sum);
        // Рассчитываем сумму минус процент комиссии
        $sum -= ($sum / 100 * 10);
        // Прибавляем сумму с комиссией к текущему балансу
        $this->_balance += $sum;
    }

}

$account = new Account();
$acount->addMoney(100); // balance = 100 - 10 = 90
echo $acount->getBalance(); // 90 / 61 = 1.4754...
```

Полиморфизм

Полиморфизм - способность приложения выполнять различные функции в зависимости от конкретного объекта, на который воздействует это приложение. Например нам необходимо вывести все записи на одной страницы причем записи разного типа - новости, посты блога и т.д. Создадим интерфейс IArticle с методом

```php
interface IArtical
{
    public function showText();
}
```

Создаем классы новостей и постов, реализующих данный интерфейс

```php
class News implements IArtical
{
    public function showText()
    {
        echo '<h3>Выводим текст новости</h3>';
    }
}
```

```php
class Post implements IArtical
{
    public function showText()
    {
        echo '<h3>Выводим текст поста блога</h3>';
    }
}
```

А теперь можем работать с разными классами без необходимости знать что за объект используем т.к. они реализуют один интерфейс.

```php
$articles = [new News(), new Post()];

foreach ($articles as $article)
{
    if ($article instanceof IArtical)
    {
        $article->showText();
    }
}
```

Ключевое слово "final"

PHP 5 представляет ключевое слово final, разместив которое перед объявлениями методов класса, можно предотвратить их переопределение в дочерних классах. Если же сам класс определяется с этим ключевым словом, то он не сможет быть унаследован. Следующий код закончится фатальной ошибкой т.к. дочерний класс пытается переопределить финальный метод родительского класса

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

    final public function moreTesting()
    {
        echo "Вызван метод BaseClass::moreTesting()\n";
    }
}

class ChildClass extends BaseClass
{
    public function moreTesting()
    {
        echo "Вызван метод ChildClass::moreTesting()\n";
    }
}
```

Данный код тоже закончится фатальной ошибкой т.к. нельзя наследоваться от финального класса

```php
final class BaseFinalClass {}

class ChildOfFinalClass extends BaseFinalClass {}
```

Трейты

Начиная с версии 5.4.0 PHP вводит инструментарий для повторного использования кода, называемый трейтом.

Трейты (англ. traits) - это механизм обеспечения повторного использования кода в языках с поддержкой единого наследования, таких как PHP. Трейты предназначены для уменьшения некоторых ограничений единого наследования, позволяя разработчику повторно использовать наборы методов свободно, в нескольких независимых классах и реализованных с использованием разных архитектур построения классов. Семантика комбинации трейтов и классов определена таким образом, чтобы снизить уровень сложности, а также избежать типичных проблем, связанных с множественным наследованием и с т.н. mixins.

Трейт очень похож на класс, но предназначен для группирования функционала хорошо структурированным и последовательным образом. Невозможно создать самостоятельный экземпляр трейта. Это дополнение к обычному наследованию и позволяет сделать горизонтальную композицию поведения, то есть применение членов класса без необходимости наследования.

```php
trait EmployeeTrait
{
    function justWork()
    {
        echo 'Я работаю';
    }
}
```

```php
class Worker
{
    use EmployeeTrait;
}

$worker = new Worker();
$worker->justWork(); // Я работаю
```

Наследуемый член из базового класса переопределяется членом, находящимся в трейте. Порядок приоритета следующий: члены из текущего класса переопределяют методы в трейте, которые в свою очередь переопределяют унаследованные методы.

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

```php
trait SayWorld
{
    public function sayHello()
    {
        parent::sayHello();
        echo 'World!';
    }
}
```

```php
class MyHelloWorld extends Base
{
    use SayWorld;
}

$o = new MyHelloWorld();
$o->sayHello(); // Hello World
```

Несколько трейтов могут быть вставлены в класс путем их перечисления в директиве use, разделяя запятыми.

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

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

class JustHelloWorld
{
    use Hello, World;
}
```

Если два трейта вставляют метод с одним и тем же именем, это приводит к фатальной ошибке в случае, если конфликт явно не разрешен. Для разрешения конфликтов именования между трейтами, используемыми в одном и том же классе, необходимо использовать оператор insteadof для того, чтобы точно выбрать один из конфликтных методов. Так как предыдущий оператор позволяет только исключать методы, оператор as может быть использован для включения одного из конфликтующих методов под другим именем.

```
trait A
{
    public function smallTalk() { echo 'a'; }

    public function bigTalk() { echo 'A'; }
}

trait B
{
    public function smallTalk() { echo 'b'; }

    public function bigTalk() { echo 'B'; }
}
```

```
class Talker
{
    use A,
        B {
            B::smallTalk insteadof A;
            A::bigTalk insteadof B;
        }
}
```

```
class Aliased_Talker
{
    use A,
        B {
            B::smallTalk insteadof A;
            A::bigTalk insteadof B;
            B::bigTalk as talk;
        }
}
```

Пространство имен

Пространство имен, в широком смысле - это один из способов инкапсуляции элементов. Если проводить аналогию представьте файлы с одинаковыми именами они могут располагаться только в разных директориях. Пространства имен PHP предоставляют возможность группировать логически связанные классы, интерфейсы, функции и константы.

В PHP пространства имен используются для решения двух проблем, с которыми сталкиваются авторы библиотек и приложений при создании повторно используемых элементов кода, таких как классы и функции:

1. Конфликт имен между вашим кодом и внутренними классами, функциями, константами PHP или сторонними.
2. Возможность создавать псевдонимы (или сокращения) для очень длинных имен, чтобы облегчить первую проблему и улучшить читаемость исходного кода.

Хотя любой исправный PHP-код может находиться внутри пространства имен, только классы (включая абстрактные и трейты), интерфейсы, функции и константы зависят от него. Пространства имен объявляются с помощью зарезервированного слова **namespace**. Файл, содержащий пространство имен, должен содержать его объявление в начале перед любым другим кодом, кроме зарезервированного слова declare.

```php
namespace Loftschool\Php;

class PhpLesson {}
// Использование класса
$lesson_1 = new \Loftschool\Php\PhpLesson();
```

Названия пространств имен PHP и php, и составные названия, начинающиеся с этих (такие как PHP\Classes), являются зарезервированными для нужд языка и их не следует использовать в пользовательском коде.

Возможность ссылаться на внешнее абсолютное имя по псевдониму или импортирование - это важная особенность пространств имен. Это похоже на возможность файловых систем unix создавать символические ссылки на файл или директорию.Все версии PHP, поддерживающие пространства имен, поддерживают три вида создания псевдонима имени или импорта: создание псевдонима для имени класса, создание псевдонима для имени интерфейса и для имени пространства имен. PHP 5.6+ также поддерживает импорт функций и имен констант.

В PHP создание псевдонима имени выполняется с помощью оператора use. Вот пример, показывающий 5 типов импорта:

```php
// Чтобы постоянно не писать весь namespace используем use
use Loftschool\Php\PhpLesson;
// Теперь можно обращаться к классу просто PhpLesson
$lesson_2 = new PhpLesson();
```

PHP поддерживает два способа к абстрактно доступным элементам в текущем пространстве имен таким, как магическая константа __NAMESPACE__ и ключевое слово namespace. Значение константы __NAMESPACE__ - это строка, которая содержит имя текущего пространства имен. В глобальном пространстве, вне пространства имен, она содержит пустую строку.

```php
echo __NAMESPACE__; // Loftschool\Php
```