ООП. Модуль 3. Расширенные понятия

Продолжим наше знакомство с ООП.

Статичные свойства и методы. Позднее статическое связывание

Познакомившись с классами, мы охарактеризовали их как шаблоны, которые не содержат в себе никаких данных и служат для создания объектов. На самом деле мы можем получить доступ к методам и свойствам из контекста класса, т.е. из шаблона класса. Для этого свойство или метод должны быть статичными, т.е. не относиться к каждому объекту, а относиться непосредственно ко всему классу.

Чтобы сделать метод или свойство статичным необходимо добавить модификатор *static*. Обычно он указывается после модификатора доступа.

```php
class Zoo
{
    protected static $animalsCount = 0;

    public static function getAnimalsCount()
    {
        //…
    }

}
```

Статичные метода и свойства существуют вне объектов, поэтому внутри их методов нельзя обращаться к нестатичным свойствам и объектам. Также внутри статичного метода не существует указателя $this.

Но к статичным методам и свойствам обращаться можно изнутри метода, используя указатель *self*, который, как вы помните, указывает на текущий класс.

```php
class Zoo
{
    protected static $animalsCount = 0;

    public static function getAnimalsCount()
    {
        return self::$animalsCount;
    }

    public static function hasAnimals()
    {
        return self::getAnimalsCount() <= 0;
    }

}
```

Также к статичным методам можно обращаться извне класса, используя сигнатуру класса, т.е. указав полное имя класса, затем двоеточие и название публичного метода или свойства

```php
echo Zoo::getAnimalsCount();
// 0
```

И из объекта этого класса также можно обращаться к статичным методам, как и к обычным, принадлежащим ему методам.

```php
$zoo = new Zoo();
echo $zoo->getAnimalsCount() . PHP_EOL;
// 0
```

Добавим приватную переменную *$name* - заполним ее в конструкторе и опишем методы **getName()** и **describeZoo()**

Такой пример уже вызовет ошибку.

Метод **describeZoo()** - попытается вызвать статичный метод **getName()**, который обращается к нестатичной переменной *$name*. Произойдет ошибка при выполнении, даже не смотря что мы вызываем метод у существующего объекта, указателя *$this* внутри **getName()** не будет.

```php
class Zoo
{
    protected static $animalsCount = 0;
    private $name;

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

    public static function getAnimalsCount()
    {
        return self::$animalsCount;
    }

    public static function hasAnimals()
    {
        return self::getAnimalsCount() <= 0;
    }

    public static function describeZoo()
    {
        return self::getName() . ': ' .
self::getAnimalsCount();
    }

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

	`}`
	`}`
	
	`$zoo = new Zoo('Simple Zoo');`
	`echo $zoo->getAnimalsCount() . PHP_EOL;`
	
	`echo $zoo->describeZoo();`

Так тоже сделать не получится. Метод не может быть статичным **describeZoo()** здесь, т.к. обращается к нестатичному методу, а тот в свою очередь к нестатичной перменной.	`public static function describeZoo()`
	`{`
	`return $this->getName() . ': ' .`
	`self::getAnimalsCount();`
	`}`

Только так будет работать.	`public function describeZoo()`
	`{`
	`return $this->getName() . ': ' .`
	`self::getAnimalsCount();`
	`}`
	`$zoo = new Zoo('Simple Zoo');`
	
	`echo $zoo->describeZoo() . PHP_EOL;`
	`//`
	`Simple Zoo: 0`

Также стоит помнить, что объекты не имеют своего значения статичной переменной, все статичные переменные относятся к исходному шаблону, а не какому-либо объекту.

Это значит, что при изменении свойства одним объектом, его значение изменится для всех объектов.

```php
class Zoo
{
    protected static $animalsCount = 0;

    // ….

    public function addAnimal()
    {
        self::$animalsCount++;
    }
}

$zoo = new Zoo('Simple Zoo');
$zoo2 = new Zoo('School');

$zoo->addAnimal();
$zoo->addAnimal();


echo $zoo->describeZoo()  . PHP_EOL;
echo $zoo2->describeZoo() . PHP_EOL;

//
Simple Zoo: 2
School: 2
```

Позднее статическое связывание

Мы помним, что указатель self указывает непосредственно на класс, в котором он вызывается, поэтому все методы, которые будут вызваны или обращения к свойствам - будут относиться непосредственно к текущему классу, и не будут учитывать классы наследники.

Рассмотрим пример.

Везде выведется 1, т.е. используя *self* мы не можем сослаться на статичное свойство класса наследника (на вызывающий контекст)

```php
class BaseTimer
{
    public static $time = 0;
    public function tic()
    {
        self::$time++;
    }

    public static function getTime()
    {
        return self::$time;
    }
}

class TimerA extends BaseTimer
{
    public static $time = 0;
}

class TimerB extends BaseTimer {}
```

```php
$timerA = new TimerA();
$timerB = new TimerB();

$timerA->tic();

echo $timerA->getTime()    . PHP_EOL; // ?
echo $timerB->getTime()    . PHP_EOL; // ?
echo BaseTimer::getTime() . PHP_EOL; // ?

//
1
1
1
```

Для решения этой проблемы в php был добавлен специальный указатель static, который, в отличие от self, ссылается на вызывающий его класс, а не на вызывающий код.

Изменим пример, воспользуемся указателем *static* при обращении к переменной *$time*.

Теперь при изменений статичного свойства *$time* переопределенного у класса *TimerA* - благодаря статическому связыванию, будет меняться именно его внутреннее свойство, а не свойство класса родителя.

```php
class BaseTimer
{
    public static $time = 0;
    public function tic()
    {
        static::$time++;
    }

    public static function getTime()
    {
        return static::$time;
```

```
    }
}
//…

$timerA = new TimerA();
$timerB = new TimerB();

$timerA->tic();

echo $timerA->getTime()    . PHP_EOL; // ?
echo $timerB->getTime()    . PHP_EOL; // ?
echo BaseTimer::getTime() . PHP_EOL; // ?

//
1
0
0
```

При этом вызов метода *tic* у объекта *TimerB*, изменит значение таймера в базовом классе, т.к у него нет своего переопределенного свойства.

```
$timerA = new TimerA();
$timerB = new TimerB();

$timerB->tic();

echo $timerA->getTime()    . PHP_EOL; // ?
echo $timerB->getTime()    . PHP_EOL; // ?
echo BaseTimer::getTime() . PHP_EOL; // ?
//
0
1
1
```

А где на практике стоить применять статичные методы и свойства. Первый пример - для доступа к неизменяемым свойствам, перечислениям или вспомогательным методам объекта, без необходимости создания экземпляра класса.

Например, перечисление статусов у заказа. Список доступных статусов задан статичной переменной. Ведь он не зависит от конкретного объекта заказа. На какой-нибудь странице, например на странице редактирования заказа, будет удобно показать выпадающий список для выбора нового статуса, тут подойдет метод Order::getStatuses();

Также может понадобиться получить конкретный статус, и вариант со статичной функцией лучше чем прямое обращение к константе Order::getDoneStatus();

А вот функция isDone() в данном примере, выглядит как полезная, но на самом деле вряд ли пригодится. Скорее всего вам придется проверять статус конкретного заказа - т.е. конкретного экземпляра, и гораздо лучше вызвать этот метод для него, т.е. $order->isDone();

```php
class Order
{
    public $status;

    const STATUS_CREATED   = 'created';
    const STATUS_CANCELLED = 'cancel';
    const STATUS_DONE      = 'done';

    protected static $statuses = [
        self::STATUS_CREATED,
        self::STATUS_CANCELLED,
        self::STATUS_DONE,
    ];

    public static function isDone($orderStatus)
    {
        return $orderStatus === self::STATUS_DONE;
    }

    public static function getDoneStatus()
    {
        return self::STATUS_DONE;
    }

    public static function getStatuses()
```

```
    {
        return static::$statuses;
    }
}

// Получить список статусов для select'a
foreach (Order::getStatuses() as $status) {
    ?><option
value="<?=$status?>"><?=$status?></option><?
}

// Где-то в программе нам нужно установить фильтр
для выбора только завершенных заказов
$filter = ['status' => Order::getDoneStatus()];

// А вот функция isDone() - вряд ли пригодится
$order = new Order();

Order::isDone($order->status); // Выглядит
странно
$order->isDone(); // Так гораздо понятнее
```

Вторым популярным способом применения статичных являются так называемые именованные конструкторы. Внутри них происходит создание объекта каким-то особым способом, например с предустановленными начальными значениями, без необходимости каждый раз указывать такие значения при создании нового объекта.

Вот пример такого именованного конструктора. В данном виде он никакой пользы не несет.	`class Example`
	`{`
	`  public static function createMe()`
	`  {`
	`  return new static();`
	`  }`
	`}`

Хорошо проиллюстрирует полезность таких конструкторов лучшая библиотека php для работой с датой, временем и календарем nesbot/carbon, и ее основной пакет Carbon\Carbon. Вы можете легко найти ее исходный код на github, а мы приведем реальные удобные примеры из этой библиотеки, по использованию именованных конструкторов.

Создание объекта с текущей датой временем.	`$carbon = Carbon::now();`
Создание объекта с указанием года, месяца и дня	`$carbon = Carbon::createFromDate($year, $month, $days);`
Сегодня и вчера.	`$carbon = Carbon::yesterday();` `$carbon = Carbon::tomorrow();`
Создание из формата (1-е февраля) Создание из timestamp метки - в данном примере - час назад.	`$carbon = Carbon::createFromFormat('d.m.Y H:i:s', '01.02.2018 13:14:15');` `$carbon = Carbon::createFromTimestamp(time() - 3600);`

По-сути, все что делают эти методы - это создают новый объект Carbon\Carbon, основываясь на предустановленных данных, вшитых в тот или иной метод (именованный конструктор), или создают его из ограниченного но достаточного набора данных.

Абстрактные классы и методы

Абстрактные объекты открывают большое количество возможностей при построении архитектуры системы. Помните пример про повара, из предыдущей лекции. Он готовил еду, при этом на выходе мы получали конкретную реализацию еды, конфеты, суп и т.п. По своей сути они являются едой - некой абстракцией в нашем приложении. А может ли повар приготовить просто еду. Нет, только конкретные блюда. Это значит, что было бы очень хорошо для нашего приложения, если бы нельзя было создать экземпляр объекта Еда.

т.е. нельзя было бы сделать так.	`$food = new Food();`

Именно это позволяют сделать абстрактные классы. Другими словами, абстрактные класса - это базовые классы, которые не предполагают создания объектов, а сделаны для расширения (наследования).

Чтобы указать, что класс абстрактный - необходимо добавить ключевое слово *abstract* в описании класса.
Попытка создать новый экземпляр абстрактного класса вызовет ошибку.
	`abstract class Food`
	`{`
	
	`}`
	
	`$food = new Food();`

Абстрактный класс также как и обычный класс может содержать реализации методов с разными уровнями доступа, иметь свойства с различными уровнями доступа, иметь статичные свойства и метода. Может быть наследником от обычного или другого абстрактного класса. Также может реализовывать интерфейсы и использовать трейты (о тех и других расскажем чуть позже). В отличие от обычных классов только Абстрактный класс может содержать абстрактные методы - методы без реализации.

Чтобы указать, абстрактный метод - необходимо добавить модификатор *abstract* к описанию метода.

Также у абстрактного метода не может быть реализации, т.е. после перечисления всех аргументов у метода должна стоять *;*

```
abstract class Food
{
    abstract public function getReceipt();
}
```

При этом, если класс содержит хоть один абстрактный метод - он обязан быть абстрактным.

Что же дает нам использование абстрактного класса, какие преимущества. С его помощью можно получить все преимущества и полиморфизма и наследования. Можно реализовать часть методов объекта, при этом в обязательном порядке делегировать необходимость реализации части функциональности дочернему класса. Т.е. дочерние классы гарантированно реализуют необходимый метод, или они также обязаны быть абстрактными, т.е нельзя будет создать их экземпляр.

В нашем примере с едой, мы обязываем всех наследников реализовать метод получения рецепта приготовления еды.

```php
abstract class Food
{
    abstract public function getReceipt();
}

class Chocolate extends Food
{
    public function getReceipt()
    {
        // какао и молоко
    }
}
```

Абстрактный класс помогает реализовать стратегию выполнения своей функциональности, без необходимости реализации деталей этой функциональности.

Рассмотрим пример такой стратегии. Выведем информацию о продукте в виде шаблона:

“Чтобы приготовить: <Название продукта>, нам понадобится(-ятся): <Рецепт>. ”

Дополним наш абстрактный класс Продукта именем и методом который будет возвращать количество ингредиентов.

```php
abstract class Food
{
    protected $name;

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

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

Сделаем вспомогательную функцию, которая выведет нашу строку, на основе абстрактного полиморфного объекта *Food*.

Таким образом мы построили Стратегию формирования описания о продукте, используя абстрактный класс, и обязали реализовать всех наследников класса *Food* два метода.

Тем самым мы абсолютно уверены в том, что можем свободно использовать эти методы в своем приложении.

```
    }

    abstract public function getReceipt();
    abstract public function ingredientsCount();
}


function describeFood(Food $food) {
    echo 'Чтобы приготовить: ' .
$food->getName() . ', нам ' .
($food->ingredientsCount() > 1 ? 'понадобятся'
: 'понадобится') .': ' . $food->getReceipt() .
'.' . PHP_EOL;
}
```

Чтобы добавлять все новые и новые продукты в программу, нужно наследоваться от *Food* и реализовать в каждом наследнике два абстрактных метода: получение рецепта и количество ингредиентов. При этом как мы реализуем внутренние методы - это уже наше дело, и не зависит от абстрактного класса.

Так мы можем добавить практически бесконечное число продуктов в наше приложение.

```
class Chocolate extends Food
{
    public function __construct()
    {
        parent::__construct('Шоколад');
    }

    public function getReceipt()
    {
        return 'какао и молоко';
    }

    public function ingredientsCount()
    {
```

```
        return 2;
    }
}

class BakedApple extends Food
{
    private $ingredients = ['яблоко'];

    public function __construct()
    {
        parent::__construct('Печеное яблоко');
    }

    public function getReceipt()
    {
        return implode(', ',
$this->ingredients);
    }

    public function ingredientsCount()
    {
        return count($this->ingredients);
    }
}
//
Чтобы приготовить: Шоколад, нам понадобятся:
какао и молоко.
Чтобы приготовить: Печеное яблоко, нам
понадобится: яблоко.
```

Интерфейсы

Интерфейсы внешне очень схожи с абстрактными классами, но в отличие от них они вообще не могут иметь реализаций методов. Все метод должны быть публичными. Также в интерфейсах нельзя объявлять свойства, но константы интерфейс может содержать.

Объявляются интерфейсы также как и классы, но вместо ключевого слова *class* используется *interface*	`interface Walkable`
	`{`
	`  public function walk();`
	`}`
	
	`interface Sleepable`
	`{`
	`  public function sleep(int $seconds);`
	`}`

При попытке создать свойство, установить приватный или защищенный уровень доступа или реализовать метод будут возникать фатальные ошибке еще на уровне интерпретации.

Также как и для абстрактного класса - нельзя создать экземпляр интерфейса. Интерфейс можно только реализовать, внутри другого объекта.

Чтобы реализовать интерфейс, необходимо использовать ключевое слово *implements* при описании	`class Cat implements Walkable, Sleepable` `{`

класса (по аналогии с *extends*). После него необходимо указать полные имена интерфейсов, через запятую. В отличие от наследования, один класс может реализовывать сколько угодно интерфейсов.

```
    public function walk() {
        //
    }

    public function sleep(int $seconds) {
        //
    }
}
```

Класс реализующий интерфейс обязан реализовать все методы этого интерфейса, или он должен быть абстрактным. При этом сигнатура метода должна совпадать с сигнатурой в интерфейсе. При попытке изменить параметры метода возникнет ошибка на уровне интерпретации.

Интерфейсы позволяют еще более гибко чем абстрактные классы использовать полиморфизм внутри приложения. Если в случае с абстрактным классом - наши классы должны быть наследниками исходного абстрактного класса, то в случае реализации интерфейса - объекты могут вообще не иметь ничего общего.

Продемонстрируем в программе.

Реализуем методы в классе Кот - *Cat*

Создадим новый класс Человек - *Person*. Человек также как и кот может гулять, поэтому реализуем интерфейс *Walkable*.

Создадим вспомогательную функцию пойти на прогулку, на вход которой нужен объект реализующий интерфейс *Walkable*.

```
class Cat implements Walkable, Sleepable
{
    public function walk() {
        echo 'Cat Walk';
    }

    public function sleep(int $seconds) {
        echo 'Cat Sleep: ' . $seconds;
    }
}
```

Протестируем нашу программу.

Несмотря на то, что кот не человек, а человек не кот, т.е. объекты ничего общего не имеют между собой, но гулять умеют оба, т.к. реализуют интерфейс. Поэтому наша вспомогательная функция выполнена успешно. Ей совершенно не важно какие конкретные реализации были в нее переданы.

```php
class Person implements Walkable
{
    public function walk()
    {
        echo 'Person Walk';
    }
}

function goForAWalk(Walkable $walkable) {
    $walkable->walk();
    echo PHP_EOL;
}

$cat = new Cat();
$person = new Person();

goForAWalk($cat);
goForAWalk($person);
//
Cat Walk
Person Walk
```

Интерфейсы также как классы могут наследоваться друг от друга, при этом возможно множественное наследование.

Создадим интерфейс животное *Animal* расширяющее наши интерфейсы, и расширим класс *Cat*.

Теперь кот (*Cat*) реализует интерфейс *Animal*, а значит и реализует интерфейсы *Walkable* и *Sleepable*. И наша программа все еще работает корректно.

```php
interface Animal extends Walkable, Sleepable
{
    public function say();
}

class Cat implements Animal
{
    public function walk() {
        echo 'Cat Walk';
    }

    public function sleep(int $seconds) {
        echo 'Cat Sleep: ' . $seconds;
    }

    public function say()
    {
        echo "Mau";
    }
}
```

Именно за счет интерфейсов достигается гибкость и наименьшая связность компонентов приложения между собой. Что облегчает замену компонентов системы и ее доработку.

Трейты

В PHP как и во многих других языках нет поддержки множественного наследования, частично это ограничение можно обойти с помощью интерфейсов. Класс будет соответствовать типам всех реализуемых интерфейсов. Но интерфейсы не могут иметь реализации, они только описывают поведение, а иногда было бы хорошо, как бы “подмешать”, готовую функциональность к классу. Для этого и существуют Трейты.

Трейты очень схожи с классами, за исключением того что нельзя создать экземпляр трейта. Трейты могут быть включены в любой класс, не меняя его тип. Класс в который включен трейт получает доступ ко всем методам и свойствам описанным в трейте.

Для объявления трейта используется ключевое слово *trait*.

Создадим трейт для отправки уведомления из класса.

И второй трейт Для форматирования числа. Можно использовать разные уровни доступа к методам внутри трейта, при этом модификатор *private* не ограничивает область видимости только трейтом. Т.к. трейт “подмешивается” к исходному классу, то и область видимости рассчитывается относительно класса, содержащего трейт.

```php
trait SendMail
{
    public function sendMail($email)
    {
        echo "Отправляю Email по адресу
{$email}" . PHP_EOL;
    }
}

trait FormatNumber
{
    private function formatNumber($number)
    {
        return $number . 'rub.';
    }
}
```

Трейты удобно использовать для вынесения общей логики приложения в отдельные подключаемые “блоки”, для избежания дублирования кода.

Для реализации трейта классом используется ключевое слово *use* после которого идет полное наименование трейта. Как и интерфейсы, класс может содержать сразу несколько трейтов. Их можно указывать через запятую, но согласно общепринятым стилям, лучше указывать каждый трейт на отдельной строке с отдельным оператором *use*.

В этом примере, как раз видно, что класс, событие на момент оплаты заказа *OrderPayed*, внутри своего метода без ошибок вызывает приватный метод трейта *formatNumber*.

```php
class OrderPayed
{
    use SendMail;
    use FormatNumber;

    public function formatPrice($orderPrice) {
        return $this->formatNumber($orderPrice);
    }
}

$orderPayed = new OrderPayed();
$orderPayed->sendMail('order_payed_email@example.com');
echo $orderPayed->formatPrice(100.00). PHP_EOL;

//
Отправляю Email по адресу
order_payed_email@example.com
100rub.
```

Трейты можно использовать внутри трейтов. Благодаря этому можно разбивать программу на очень маленькие специфичные блоки, собирать их в более крупные и подключать в классе.

Рассмотрим пример.

Трейт для уведомлений использует внутри себя другой трейт - отправки.

Интересным моментов в реализации метода **notify()** трейта *Notification* является использование свойства *$this->email* из класса родителя. В теле трейта это свойство не объявлено, но так как трейт является частью класса, то он может обращаться ко всем его свойствам и методам. Трейт наследует контекст выполнения у класса реализующего этот трейт. Следует с особой осторожностью использовать свойства из контекста класса, так как появляется жесткая связь с классом, использующим трейт, такие связи сложно отследить, что означает потенциальное место для ошибки в будущем.

Также трейт может реализовывать интерфейс или его часть. В нашем случае для классов Человек и Кот, трейт реализует интерфейс *Notifiable*.

Еще одна особенность - переопределение методов. Если в классе есть метод с таким же названием, что и в трейте - то метод будет

```php
trait Notification
{
    use SendMail;

    protected function needSendEmail()
    {
        return true;
    }

    public function notify()
    {
        if ($this->needSendEmail()) {
            $this->sendMail($this->email);
        }

        $this->toLog();
    }

    public function toLog()
    {
        echo "Лог об отправке уведомления class:
" . substr(strrchr(__CLASS__, "\\"), 1) .
PHP_EOL;
    }
}

interface Notifiable {
    public function notify();
}
```

переопределен и именно он будет вызван при обращении к методу текущего объекта.

У кота ведь нет email. Благодаря переопределению - *email* коту отправлен не будет, но другой функционал уведомления выполнится. *Email у кота может и не быть, но голубями кот посылку с сосисками с удовольствием примет. Если конечно трейт Notification будет поддерживать отправку уведомления этим каналом.*

Но стоит помнить, что происходит именно переопределение метода, а не его наследование, т.е. обратиться к методу трейта используя указатель parent: *parent::needSendEmail();* - не получится

```php
class Person implements Notifiable
{
    use Notification;
    protected $email;

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

class Cat implements Notifiable
{
    use Notification;

    protected function needSendEmail()
    {
        return false;
    }
}

$person = new
Person('person_email@example.com');
$person->notify();

$cat = new Cat();
$cat->notify();

//
Отправляю Email по адресу
```

```
person_email@example.com
Лог об отправке уведомления class: Person
Лог об отправке уведомления class: Cat
```

Трейт не может сам по себе реализовывать интерфейсы, возможно это было бы даже удобно, но так сделать нельзя. Трейт как и абстрактный класс может содержать абстрактные методы.

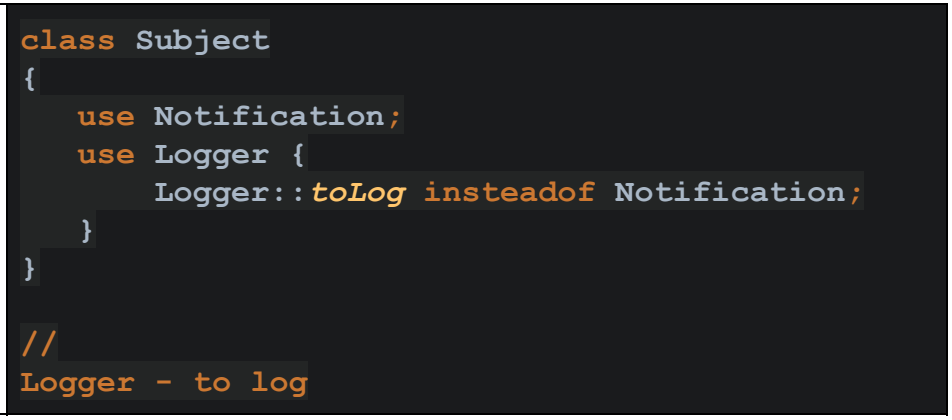
Класс может содержать множество трейтов, и не исключено, что названия методов внутри них совпадут. Пространства имен и т.п. штуки не подойдут, ведь Трейт наследует контекст выполнения у класса реализующего этот трейт, т.е. все методы “подмешиваются” внутрь одного класса.

Рассмотрим пример в котором два трейта имеют один и тот же метод. Используем трейт *Notification* и новый трейт *Logger*.

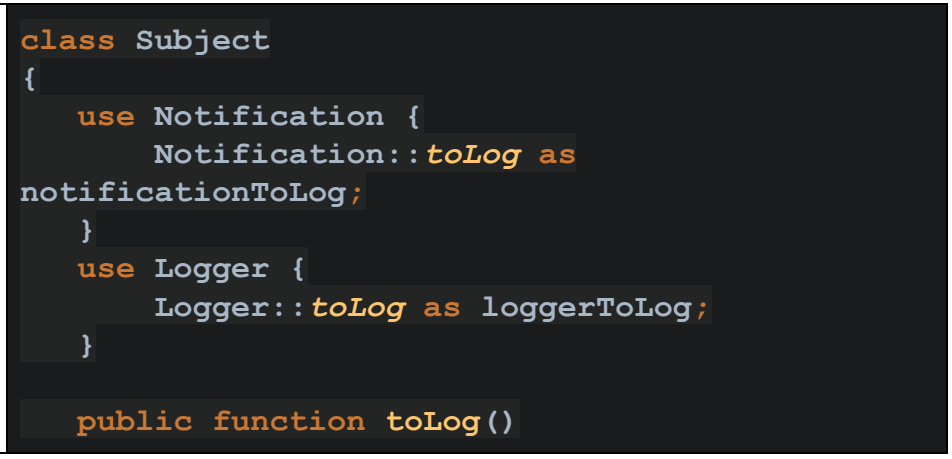
В этом случае при попытке выполнения возникнет фатальная ошибка

```php
trait Logger
{
    public function toLog()
    {
        echo 'Logger - to log' . PHP_EOL;
    }
}

class Subject
{
    use Notification;
    use Logger;
}

$subject = new Subject();
$subject->toLog();
```

Для решения подобный проблем в языке имеется конструкция “*insteadof*”, которая позволяет явно указать из какого трейда необходимо использовать метод.

Используется такая конструкция. В данном случае выполнится метод из трейта *Logger*. Конструкция указывает на то, что метод *Logger::toLog* нужно использовать вместо аналогичного метода трейта *Notification*.	

Но что делать если требуется использовать сразу два метода? Для этого в php предусмотрены псевдонимы трейтов, которые можно задать с помощью ключевого слова “*as*”. Для того чтобы понять как это работает дополним предыдущий пример использованием псевдонимов.

Нам пришлось переопределить метод **toLog()** и в нем вызвать два новых алиаса, приписанных нашим трейтам.
Важный момент: псевдоним не переименовывает метод, а лишь дополнительно “создает” новый метод в котором происходит перенаправление на исходный. Поэтому не получится решить конфликт имен просто используя псевдонимы.
	 class Subject
	{
	use Notification {
	Notification::toLog as
	notificationToLog;
	}
	use Logger {
	Logger::toLog as loggerToLog;
	}
	public function toLog()

Также отдельно будут доступны эти методы для класса.

```
    {
        $this->notificationToLog();
        $this->loggerToLog();
    }
}

$subject = new Subject();
$subject->toLog();

$subject->notificationToLog();
$subject->loggerToLog();


//
Лог об отправке уведомления class: Subject
Logger - to log
Лог об отправке уведомления class: Subject
Logger - to log
```

Еще одной возможностью оператора “*as*” является переопределение уровня доступа. Если вместо название нового метода указать один из модификаторов “*public*”, “*protected*”, “*private*”, то вместо создания псевдонима метод получит новый уровень доступа в рамках класса.

Теперь при попытке вызвать **$subject->toLog()** возникнет ошибка доступа к методу.

```
class Subject
{
    use Notification {
        Notification::toLog as private;
    }
}
```

Финальные классы и методы

Используя наследование можно создать новый класс имеющий функциональность родительского, а также переопределить в нем методы базового класса. Таким образом вызов метода для разных классов, являющихся наследником исходного - будет приводить к разным результатам. Это часто очень удобно, но иногда требуется запретить переопределение метода или даже класса, чтобы его поведение нельзя было изменить.

Для этого используется ключевое слово *final*. Оно должно быть использовано до применения любых других модификаторов, уровня доступа, или *static*.

Попробуем создать класс. Для этого в начале описание класса нужно добавить ключевое слово *final*.

Теперь попробуем наследоваться от нашего класса.

Еще на этапе интерпретирования кода мы получим ошибку Фатальную ошибку.

```php
final class ICantHaveChildren
{
    public function make()
    {
        return 'hello from Parent';
    }
}

class Child extends ICantHaveChildren
{
    public function make()
    {
        return 'hello from Children';
    }
}
// Class Child may not inherit from final class
(ICantHaveChildren)
```

Продолжим наш пример, теперь попробуем запретить переопределение метода, для этого добавим ключевое слово *final* не к классу, а к его методу.

Ошибка также появится еще на этапе интерпретирования.

```php
class ICantHaveChildren
{
    final public function make()
    {
        return 'hello from Parent';
    }
}

class Child extends ICantHaveChildren
{
    public function make()
    {
        return 'hello from Children';
    }
}

// Cannot override final method
ICantHaveChildren::make()
```

Ключевое слово “*final*” обеспечивает неизменность определенной части кода, для его использования должны быть веские причины, так как впоследствии на том, что класс является неизменным могут быть тесно завязаны другие части системы. Если вам когда нибудь потребуется указать метод или класс как “*final*” обязательно задокументируйте причину.

Магические методы

Помните, когда мы рассказывали базовые понятия ООП, мы упоминали о паре специальных методов у класса, конструктор и деструктор. Таких методов у классов гораздо больше и называются они “магические методы”.

Все магические методы начинают свое название с двойного символа подчеркивания, и создатели php, условно считают, что в новых версиях могут появиться и другие магические методы, поэтому такое начало названия метода считается зарезервированным. Не используйте двойное подчеркивание в начале названия своих методов класса.

Следующие магические методы существуют у классов.

__construct() и **__destruct()** - конструктор и деструктор - мы с ними уже познакомились ранее

Методы перегрузки

__call() и **__callStatic()** - будут вызваны при попытке вызвать несуществующий метод класса.

```php
class MagicClass
{
    public function __construct()
    {
    }

    public function __destruct()
    {
    }

    public function __call($name, $arguments)
    {
    }

    public static function __callStatic($name, $arguments)
    {
```

__get() и **__set()** - будут вызваны при попытке обратиться к несуществующему свойству класса

__isset() и **__unset()** - будут выполнены при использовании *isset()* (или *empty()*) и *unset()* соответственно на несуществующих свойствах класса.

Остальные магические методы

__sleep() и **__wakeup()** будут вызваны, когда к объекту будет применена функция **serialize()** и **unserialize()** соответственно.

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

```
    }

    public function __get($name)
    {
    }

    public function __set($name, $value)
    {
    }

    public function __isset($name)
    {
    }

    public function __unset($name)
    {
    }

    public function __sleep()
    {
    }

    public function __wakeup()
    {
    }

    public function __toString()
    {
        return '';
    }
```

Этот метод должен возвращать строку, иначе произойдёт фатальная ошибка.

__invoke() вызывается, когда скрипт пытается выполнить объект как функцию.

__debugInfo() - вызывается функцией **var_dump()**

__set_state() - вызывается для тех классов, которые экспортируются функцией **var_export()**

__clone() - вызывается при клонировании объекта.

```
    public function __invoke()
    {

    }

    public function __debugInfo()
    {
    }

    public static function __set_state($array)
    {
    }

    public function __clone()
    {
    }

}
```

Мы не будет рассматривать все методы, они не часто используются, но остановимся подробнее на методах перегрузки.

Перегрузка свойств

К методам перезагрузки свойств, как мы отметили, относятся **__get()**, **__set()**, **__isset()** и **__unset()**;

Создадим пустой класс и попробуем присвоить значения несуществующим свойствам.

Как видите, все, вроде бы неплохо получилось. Но использование несуществующих свойств - очень плохо тон.

Ведь класс о таких свойствах не знает ничего, а следовательно и все кто будут пользоваться вашим классом о таких свойствах ничего не узнают.

```php
class OverloadProperties
{

}

$overload = new OverloadProperties();
$overload->test = 'new Test Value';
$overload->TEST = 'second Test Value';
echo $overload->test . PHP_EOL;
echo $overload->TEST . PHP_EOL;
echo (isset($overload->test)  ? 'yes' : 'no') .
PHP_EOL;
echo (isset($overload->Other) ? 'yes' : 'no') .
PHP_EOL;

//
new Test Value
second Test Value
yes
no
```

Добавим метод пустые **__set()** и **__get()**.

```php
class OverloadProperties
{
   private $data = [];
```

Теперь наши свойства созданы не будут, потому что мы изменили поведение по-умолчанию нашим пустым поведением.

```
    public function __set($name, $value)
    {
    }

    public function __get($name)
    {
    }
}
//


no
no
```

Специально для таких обращений к классу мы создадим приватное свойство *$data*.

Давайте сохраним присваемые свойства в него, при этом будем игнорировать символы разного регистра для свойства и будем немного изменять входное значение, при сохранении.

Обращение к методу **__set()** происходит, когда мы присвоим свойству *test* новое значение (на строке *$overload->test = 'new Test Value';*)

при этом первый параметр *$name = 'test'*, а второй параметр *$value = 'new Test Value'*;

Обработаем данные так, как мы придумали.

```
public function __set($name, $value)
{
    $this->data[strtolower($name)] = 'custom:' .
$value;
}

public function __get($name)
{
    return $this->data[strtolower($name)];
}

//
custom:second Test Value
custom:second Test Value
no
no
```

Теперь в методе **__get()** также в регистронезависимом варианте вернем значение запрашиваемого свойства. Параметр *$name* - название свойства	

Добавим два метод, **__isset()** и **__unset()** Для завершения нашей функциональности.	`public function __isset($name)`
	`{`
	`  return array_key_exists(strtolower($name), $this->data);`
	`}`
	
	`public function __unset($name)`
	`{`
	`  if (isset($this->{$name})) {`
	`  unset($this->data[strtolower($name)]);`
	`  }`
	`}`
	
	`//`
	`custom:second Test Value`
	`custom:second Test Value`
	`yes`
	`no`

Что ж мы получили нужный нам код, используя магические методы перегрузки свойств.
Теперь небольшой рефакторинг и готово.
	`class OverloadProperties`
	`{`
	`  private $data = [];`
	
	`  public function __set($name, $value)`

```
    {
        $this->data[$this->convertName($name)] =
'custom:' . $value;
    }

    public function __get($name)
    {
        return
$this->data[$this->convertName($name)];
    }

    public function __isset($name)
    {
        return
array_key_exists($this->convertName($name),
$this->data);
    }

    public function __unset($name)
    {
        if (isset($this->{$name})) {

unset($this->data[$this->convertName($name)]);
        }
    }

    private function convertName($name) {
        return strtolower($name);
    }
}
```

Иногда бывает очень удобно пользоваться перезагрузкой свойств. Кроме того вы можете выполнять свой произвольный код внутри этих магических методов. И самое главное - теперь любой программист, посмотрев на ваш класс, поймет, что класс поддерживает динамическое использование свойств, и сможет посмотреть как эти свойства устроены внутри него.

Перегрузка методов

Помимо перегрузки свойств, есть также перегрузка методов, для этого используются магические методы **__call()** и **__callStatic()**. Как видно из названия, первый относится к динамическим вызовам методов, второй к статическим.

Создадим пример.

При попытке обращения к несуществующим методам, будет вызвана фатальная ошибка.

```php
class OverloadMethods
{
}

$overload = new OverloadMethods();
echo $overload->sayAll_Is_Good() . PHP_EOL;
echo OverloadMethods::sayHello_World() . PHP_EOL;
echo $overload->summ(1, 2, 3, 4, 5)    . PHP_EOL;
```

Давай-те научим программу складывать из слов предложения, и складывать сумму.

оба метода **__call()** и **__callStatic()** принимают два аргумента - в первом *$name* наименование вызываемой функции, а во втором

```php
class OverloadMethods
{
    private static function say($words)
    {
        return implode(' ', $words);
    }

    private static function
```

$arguments - параметры, переданные в функцию в виде массива.

Мы придумали такую логику: Если название метода начинается с *say* - то мы разбиваем название метода на слова и передаем на выполнение внутреннему методу **say()**, который вернет нам предложение из полученных слов.

В других случаях при вызове статического метода - ничего не будет происходить, а при вызове нестатичного метода - вернется сумма всех аргументов.

Проверим результат.

```
getWordsFromMethodName($name)
    {
        return array_filter(explode('_',
substr($name, 3)));
    }

    public function __call($name, $arguments)
    {
        if (strpos($name, 'say') === 0) {
            return
$this->say($this->getWordsFromMethodName($name));
        }

        return array_sum($arguments);
    }

    public static function __callStatic($name,
$arguments)
    {
        if (strpos($name, 'say') === 0) {
            return
static::say(static::getWordsFromMethodName($name));
        }
    }
}
//
All Is Good
Hello World
15
```

Остальные магические методы используются не так часто и вы можете ознакомиться с ними самостоятельно.

Теперь у вас есть полный набор знаний по ООП. Мы познакомились с основами ООП, разобрали что такое классы и что такое объекты. Узнали что классы состоят из свойств и методов, могут наследоваться от других классов, а могут сами быть родителями для них. Могут реализовывать интерфейсы или быть абстрактными. Освоили и, я надеюсь, запомнили принципы ООП. Научились создавать статичные методы, привязанные к классам в целом, а не к объектам, созданным на их основе. Познакомились со множеством расширенных понятий о классах и об их некоторых нюансах.

Эти знания помогут вам создавать сложные, расширяемые, стабильные и масштабируемые приложения.