ООП. Модуль 2. Принципы

В этом модуле мы поговорим об основных принципах ООП, их всего четыре: Абстракция, Наследование, Инкапсуляция и Полиморфизм. Постараемся понять зачем они нужны. А также узнаем как эти принципы реализуются в языке php.

Абстракция

Первый принцип, который мы рассмотрим - абстракция. Многие в современности выделяют абстракцию отдельно, поэтому и мы не будет от них отставать (В классическом представлении ООП абстракция отдельно не выносится). Абстракция в ООП - это выделение наиболее значимой (наиболее важной для нашей программы) информации в программный объект и исключение из рассмотрения менее важной информации.

На словах, возможно, это не так и просто представить, поэтому рассмотрим примеры. “Поабстрагируем на лету”.

Представим студенческую столовую, все же любят покушать. В нашей виртуальной столовой есть три основных объекта: голодные студенты, повар и еда. Фактически это и есть три различных абстракции.

Нам совершенно не важно как зовут повара, какого он пола или что-либо еще, важным для нас является то, что он умеет готовитьЕду.

Что касается студентов, то тут все то же самое, имена не важны, главное, что они голодные и содержат метод естьЕду. И не важно, что Наталья пришла попить кофе, а Миша объест пол столовой, все они пришли естьЕду.

И, ее величество Еда, в нашей абстракции достаточно знать, что это ее можно приготовить и съесть, и не важно какая это еда, сколько ее готовить, зачем в йогурт положили чеснок и т.п.

Наша программа ничего не выводит, но выполнится, потому что все условия и абстракций соблюдены.

```php
<?php

class Cook
{
    public function makeFood()
    {
        return new Food;
    }
}

class Food
{
    public function eat() {}
}

class HungryStudent
{
    public function eatFood(Food $food)
    {
        $food->eat();
    }
}

$cook = new Cook();
$student1 = new HungryStudent();
$student2 = new HungryStudent();
$student1->eatFood($cook->makeFood());
$student2->eatFood($cook->makeFood());
```

Хотя в нашей программе есть одна неточность. Повар вряд ли может приготовить абстрактную еду. Он все-таки должен готовить конкретную еду, доработаем нашу программу, пусть повар готовит конкретную еду.

Создадим три новых объекта Конфетка (Candy), Суп (Soup) и Кофе (Coffee). Мы применили наследование здесь, т.е. наши новые объекты все равно являются едой, о наследовании подробнее расскажем чуть позже.

Изменим метод приготовления еды, не будем вдаваться в подробности как он это делает, но теперь наш повар готовит конкретную еду. Но при этом наша остальная программа будет оперировать именно с абстрактной едой. Повар, несмотря на то, что готовит конкретные реализации еды, по сути все равно готовит еду. И именно еду едят Студенты.

Оставшаяся часть программы осталась без изменений. Ведь метод eatFood взаимодействует с абстрактным объектом Еда, и не важно, что повар готовит конкретную реализацию этой еды.

```php
class Cook
{
    public function makeFood()
    {
        switch ($this->whatToCook()) {
            case 'Candy':
                return new Candy();
            case 'Soup':
                return new Soup();
            case 'Coffee':
                return new Coffee();
        }
    }

    public function whatToCook()
    {
        // ....
        // return $foodType;
    }
}

class Candy extends Food {}
class Soup extends Food {}
class Coffee extends Food {}
```

```php
$cook = new Cook();
$student1 = new HungryStudent();
$student2 = new HungryStudent();

$student1->eatFood($cook->makeFood());
$student2->eatFood($cook->makeFood());
```

Наследование

Наследование - это возможность описать новый класс на основе уже существующего. Цель такого наследования - использовать функциональность базового класса, полностью или частично, а также возможность ее переопределить. Класс, от которого производится наследование, называется базовым или родительским классом, а новый класс называется наследником или дочерним классом.

Рассмотрим пример

В программе есть домашние животные: кролики, котики и собачки, они умеют ходить, спать и говорить.

Вспоминая принцип абстракции, для нашей программы можно выделить абстрактное домашнее животное, которое умеет ходить, спать и говорить.

Опишем такое домашнее животное. Наше животное топает когда ходит, и посапывает когда спит, но ничего не говорит - ведь разные животные разговаривают по разному.

Константа PHP_EOL - суперконстанта встроенная в php, выводит перенос строки, подходящий для ОС, на которой выполняется скрипт. (в браузере будет как пробел)

```php
class Pet
{
    public function walk()
    {
        echo 'топ-топ-топ'. PHP_EOL;
    }

    public function sleep()
    {
        echo 'zZZZz'. PHP_EOL;
    }

    public function say()
    {

    }
}
```

Теперь наследуем конкретных животных от родительского класса Pet.

Кролик, не помню чтобы он как-то разговаривал, кроме шуршания, поэтому мы используем простое наследование, без переопределения и расширения функциональности.

Наследование указывается в строке объявления класса после его имени, используется ключевое слово extends и за ним следует полное имя класса, от которого производится наследование.

```php
<?php

class Rabbit extends Pet
{

}
```

Создадим котика и собачку, каждый из них разговаривает по-своему.

Но наш котик, в отличие от собачки, умеет ловить мышек. Добавим соответствующий метод классу кота.

```php
class Cat extends Pet
{
    public function say()
    {
        echo 'Мяу'. PHP_EOL;
    }

    public function catchMouse()
    {
        echo 'Котик поймал мышку'. PHP_EOL;
    }
}

class Dog extends Pet
{
    public function say()
    {
```

```
        echo 'Гав'. PHP_EOL;
    }
}
```

Протестируем. Несмотря на то, что ни в одном классе мы не описывали методы walk и sleep - у всех классов эти методы существуют, т.к. они существуют у класса родителя.

И при этом функциональность метода say - для каждого класса своя, т.к. мы ее переопределили.

У переменной $cat, хранящий ссылку на объект класса Кот, есть метод поймать Мышь, при этом если мы попробуем вызвать этот метод у собачки или кролика - то получим ошибку, т.к. такого метода они не содержат

```
$rabbit = new Rabbit();
$rabbit->walk();
$rabbit->say();
$rabbit->sleep();

$cat = new Cat();
$cat->walk();
$cat->say();
$cat->catchMouse();
$cat->sleep();

$dog = new Dog();
$dog->walk();
$dog->say();
$dog->sleep();
//
топ-топ-топ
zZZZz
топ-топ-топ
Мяу
Котик поймал мышку
zZZZz
топ-топ-топ
Гав
zZZZz
```

Порядок наследования может быть практически бесконечным, т.е. мы можем наследоваться от объекта, который унаследован от объекты, который унаследован и т.д.

Кто-то держит и тигров дома, а тигры это почти коты	`class Tiger extends Cat` `{` `}`

С точки зрения принципов понятий ООП, возможно и множественное наследование, когда у класса много родителей - и наследуются все их методы, но в некоторых языках, в том числе и php, множественное наследование не разрешено на уровне самого языка.

Так сделать нельзя. Жаль, я бы посмотрел на такого монстра.	`class FlyTigerWithBigGun extends Cat, Pegas, Tank` `{` `}`

Указатель parent, self

parent

При наследовании часто возникает необходимость обратиться к исходному родительскому методу в переопределенном методе наследника. Для этого существует специальный указатель parent.

Очень часто parent используется в конструкторах, но ограничений нет, мы можем использовать его везде. Продолжим наш пример с животными.

В нашей программе появился новый зверь Болтливый Кот, который Ну очень любит мяукнуть прежде чем что-нибудь сделать. Чтобы реализовать болтуна в программе очень пригодиться указатель parent.

Унаследуем болтливого кота, от обычного кота, и переопределим его методы.

Чтобы вызвать родительский метод нужно написать parent затем двойное двоеточие и название родительского метода

Посмотрим, как отработает наша программа. Ну очень болтливый)

```php
class TalkativeCat extends Cat
{
    public function walk()
    {
        $this->say();
        parent::walk();
    }

    public function sleep()
    {
        $this->say();
        parent::sleep();
    }

    public function catchMouse()
    {
        $this->say();
        parent::catchMouse();
    }
}

$talkativeCat = new TalkativeCat();
$talkativeCat->walk();
$talkativeCat->say();
$talkativeCat->catchMouse();
$talkativeCat->sleep();

//
Мяу
топ-топ-топ
```

	`` Мяу Мяу Котик поймал мышку Мяу zZZZz ```
Фактически, если переопределить метод и сделать в нем только вызов родительского метода - это все равно что не переопределять метод.	``` public function walk() { parent::walk(); } ```

self

Указатель self - указывает непосредственно на текущий класс, не на наследника и не на родителя, а именно на текущий.

Для демонстрации добавим вызов self::say(); в метод кота поймать Мышку. А тигру добавим свой голос. Несмотря на то, что говорит тигр теперь по-другому, после того, как он поймает мышку, он все равно мяукнет как котик.	``` class Tiger extends Cat { public function say() { echo 'Roar' . PHP_EOL; } } ```

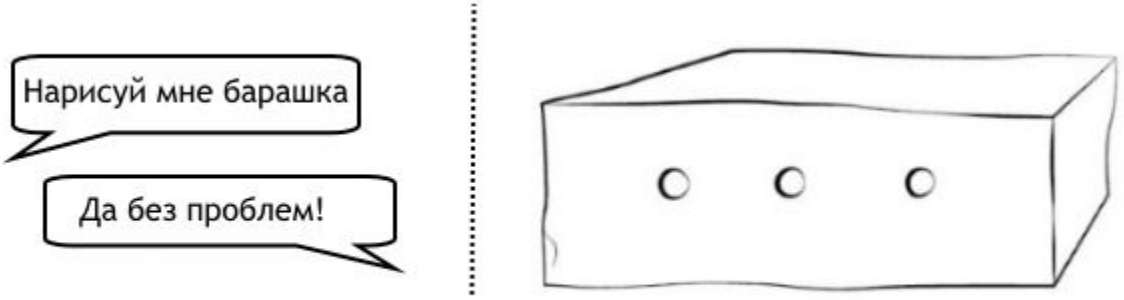
```php
class Cat extends Pet
{
    public function say()
    {
        echo 'Мяу' . PHP_EOL;
    }

    public function catchMouse()
    {
        echo 'Котик поймал мышку'. PHP_EOL;
        self::say();
    }
}


$tiger = new Tiger();
$tiger->say();
$tiger->catchMouse();
//
Roar
Котик поймал мышку
Мяу
```

Инкапсуляция и модификаторы доступа

Инкапсуляция - это возможность объекта объединять данные и методы работающие с ними, а также скрывать свое внутреннее устройство и данные от пользователя. От кого скрываем - от нашей программы и других объектов и классов. А зачем скрывать - для локализации изменений при их необходимости и прогнозируемости таких изменений. Ведь если мы уверены, что этим участком кода пользуется только текущий объект, т.к. мы его скрыли от внешней программы и других объектов, то не так страшно его менять.



Инкапсуляция позволяет скрывать детали реализации

Для изменения области видимости метода или свойства класса нужно указать ему соответствующий модификатор private, protected или public. По-умолчанию, если модификатор не указан, подразумевается public уровень доступа, но рекомендуется (а по стилям оформления кода - обязательно) указывать модификатор у всех методов и свойств класса.

Модификатор public - разрешает вызов метода и чтение/изменение свойства объекта из любого места программы.

Модификатор private - разрешает вызывать метод и читать/изменять свойство только внутри текущего класса.

Создадим класс Игрушка, с тремя разными по уровню доступа свойствами.

Внутри класса можно изменять все свойства.

Публичное свойство можно менять из любого места программы.

```php
<?php

class Toy
{
   private $scheme;
   protected $resources;
   public  $name;

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

}

$toy = new Toy('scheme', ['wood', 'steel'], 'Игрушка');

$toy->name = 'Можно изменить это свойство';
// $toy->scheme = 'Нельзя прочитать или изменить это свойство';
// $toy->resources = 'Нельзя прочитать или изменить это свойство';
var_dump($toy);
```

Модификатор protected - разрешает вызывать метод и читать/изменять свойство внутри текущего класса и классов наследников текущего. Но не извне.

Создадим игрушку наследника. Здесь можно менять свойства с модификатором protected и public	`class KinderSurpriseToy extends Toy` `{` `  public function __construct($scheme, $resources, $name)` `  {` `  parent::__construct($scheme, $resources, $name);` `  // $this->scheme = $scheme; - нельзя обратиться к этому свойству из дочернего класса` `  $this->resources[] = 'chocolate';` `  $this->name = 'Kinder Toy: ' . $name;` `  }` `}`

Пример для методов:

Создадим Фабрику игрушек и класс игрушку.

Для нашей внешней программы объект - фабрика игрушек инкапсулирован, кроме метода - получить новую игрушку - никакой другой метод этого класса вызвать нельзя. Т.е. весь процесс создания новой игрушки скрыт. от посторонних глаз.

Причем за получение ресурсов и чертежа игрушки отвечает именно базовый класс фабрики игрушек.

Но сам метод по созданию игрушки на основе ресурсов и чертежа - объявлен protected, т.е. его можно переопределить в классе наследнике.

```php
<?php

class ToyFabric
{
    public function getNewToy()
    {
        return $this->createToy();
    }

    private function createToy()
    {
        $resources = $this->collectResources();
        $scheme = $this->getToyScheme();

        return $this->makeToy($resources, $scheme);
    }

    private function collectResources()
    {
        return [];
    }

    private function getToyScheme()
    {
        return [];
    }

    protected function makeToy($resources, $scheme)
    {
```

```
        return new Toy($scheme, $resources, 'new
Toy');
    }
}
```

Создадим новую фабрику, переопределим метод создания игрушки.

И проверим наш результат.

Для класса KinderSurprizeToyFabric - также скрыто как фабрика получает ресурсы и схему игрушки. Переопределение метода collectResources не повлияет на функциональность.

```
class KinderSurpriseToyFabric extends ToyFabric
{
    protected function makeToy($resources, $scheme)
    {
        return new KinderSurpriseToy($scheme,
$resources, 'new surprise Toy');
    }
}

$toyFabric = new ToyFabric();
$surpriseToyFabric = new
KinderSurpriseToyFabric();

$toy1 = $toyFabric->getNewToy();
$toy2 = $surpriseToyFabric->getNewToy();

var_dump($toy1, $toy2);

//
object(Toy)#3 (3) {
  ["scheme":"Toy":private]=>
  array(0) {
  }
  ["resources":protected]=>
  array(0) {
  }
```

```
  ["name"]=>
  string(7) "new Toy"
}
object(KinderSurpriseToy)#4 (3) {
  ["scheme":"Toy":private]=>
  array(0) {
  }
  ["resources":protected]=>
  array(1) {
    [0]=>
    string(9) "chocolate"
  }
  ["name"]=>
  string(28) "Kinder Toy: new surprise Toy"
}
```

Взглянем в целом на нашу программу по производству игрушек. Если нам понадобиться изменить метод getNewToy или изменить тип данных в свойстве игрушки $name - то в этом случае, придется искать все обращения к методу и свойствам во всем коде проекта. Что, согласитесь, может оказаться очень не просто.

Если же нужно изменить логику работы метода сбора ресурсов - то это не составит труда, потому что мы знаем, что этот метод вызывается только из класса ToyFabric и нигде более.

В связи с этим можно вывести рекомендацию: используйте самые закрытые модификаторы доступа, всегда, если нет необходимости раскрывать шире их область видимости.

Полиморфизм

Полиморфизм - это возможность системы, использовать разные объекты с разной реализацией, но одинаковой спецификацией. Вспомним пример про столовую. Помните - нам было совершенно не важно какую еду готовит повар. Объекту Студенту, чтобы съесть ее, достаточно было просто знать, что объект является едой, и не важно какой.

Другой пример - вы отправляете документ на печать, и вам совершенно не важно какая модель принтера будет работать - все принтеры поддерживают интерфейс “Печать”. Более того печатью может заниматься и не принтер вовсе, а ваш коллега, с каллиграфическим почерком.

```php
class Printer
{
   public function printer()
   {
   }
}

class SomePrinter extends Printer {}

class OtherPrinter extends Printer {}

function goPrint(Printer $printer) {
   $printer->printer();
}

goPrint(new SomePrinter());
goPrint(new OtherPrinter());
```

Полиморфизм позволяет легче абстрагировать одну от другой части программы и выделять из них отдельные переиспользуемые модули.

Другой пример одна часть системы собирает данные, а другая их сохраняет.

Читать данным можно из разных источников, для нашей программы подойдут любые, кто наследован от базового класса Reader.

Аналогично с Writer.

Вы можете читать из базы данных, писать в файл. Читать из файла, записать в файл в облаке, и т.п. любые комбинации. Для этого только достаточно будет сделать ОДИН свой драйвер для записи и чтения, в конкретное место. И подменить один класс на другой при вызове. В популярных фреймворках такие драйвера уже есть.

```php
class Reader
{
    public function read() {}
}

class Writer
{
    public function write($data) {}
}

function convert(Reader $reader, Writer $writer) {
    $writer->write($reader->read());
}
```

Обычно для описания такого “базового” класса, используется специальный объект - интерфейс, но о нем мы расскажем позднее.

Делая модули программы переиспользуемыми - их часто выносят в отдельные “библиотеки”, поэтому часто какие-то части системы писать и вовсе не нужно - достаточно найти свободное готовое решение, разработанное

другими разработчиками, и только подключить его в своем проекте. Использование стабильной и проверенной временем, поддерживаемой сторонней библиотеки, часто гораздо выгоднее, чем написание своего кода. И это возможно именно благодаря принципам ООП, а в особенности - Полиморфизмом.

Итоговый пример Стоимость в корзине

Напишем программу, которая будет считать сумму стоимости товаров, у пользователя в корзине и выведет эту сумму в удобном для нашей веб-страницы формате, а также сможет показать стоимость одного из товаров, также в отформатированном (возможно, в другом отличном от корзины формате) виде.

Первое определим какие абстрактные элементы должны быть в нашей программе. Корзина и Товар.

У товара должна быть цена и название.

В корзине должен быть список таких товаров.

Создадим два таких класса.

```php
class Basket
{
    /** @var Product[] */
    private $products = [];

    public function addProduct(Product $product)
    {
        $this->products[] = $product;
    }
}

class Product
{
    protected $name;
    protected $price;

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

	} class HasPrice { public function getPrice() { return 0; } } class Basket extends HasPrice { /** @var Product[] */ private $products = []; public function addProduct(Product $product) { $this->products[] = $product; } public function getPrice() { $price = 0.0; foreach ($this->products as $product) { $price += $product->getPrice(); } return $price; } }
В нашей программе мы может рассчитать как сумму корзины, так и сумму товара. Значит у двух объектов должны быть методы для расчета цены. Похоже на место для полиморфизма Как и ранее здесь выгоднее использовать интерфейсы, а не базовый класс, но пока вы не знаете интерфейсы - используем базовый класс Создадим базовый класс HasPrice с методом getPrice. Унаследуем этот класс в корзине и в товаре, и переопределим (мы бы сказали, реализуем если бы использовали интерфейс) этот метод.	

	``php class Product extends HasPrice { protected $name; protected $price; public function __construct($name, $price) { $this->name = $name; $this->price = $price; } public function getPrice() { return $this->price; } public function getName() { return $this->name; } } ```
Посмотрим что Выведет наша программа теперь Сумма посчитана верно. Да, у нас очень дорогой и прибыльный магазин)	```php $basket = new Basket(); $basket->addProduct(new Product('Кубик', 10000.1)); $basket->addProduct(new Product('Матрешка', 1700.0)); $product = new Product('Колобок', 1700.348); $basket->addProduct($product); ```

```
echo 'Сумма в корзине'    . ' - ' .
$basket->getPrice()   . PHP_EOL;
echo $product->getName() . ' - ' .
$product->getPrice() . PHP_EOL;

//
Сумма в корзине - 13400.448
Колобок - 1700.348
```

Теперь нам нужно научиться форматировать результат. Для разных мест нашей веб-страницы могут понадобиться разные форматы вывода. Здесь, похоже, снова пригодится Полиморфизм и Наследование.

Создадим класс базового форматирования цены, который ничего не будет делать нового с ценой, и класс для нужного нам формата цены, а также вспомогательную функцию.

Функция formatItemPrice работает как бы с интерфейсами HasPrice и PriceFormatter. Благодаря полиморфизму, эта функция может работать как с отдельным товаром, так и с целой корзиной. И опять

```
class PriceFormatter
{
    public function format($value)
    {
        return $value;
    }
}

class NumberPriceFormatter extends PriceFormatter
{
    public function format($value)
    {
        return number_format($value, 2, '.', ' ');
    }
}


function formatItemPrice(HasPrice $hasPrice,
```

же благодаря полиморфизму и наследованию - мы можем менять формат вывода цены с помощью этой функции на странице.

В первом случае - мы выводим без изменения.

Во втором мы форматируем стоимость, добавляя пробел между тысячами, уменьшая количество знаков после запятой до двух.

```
PriceFormatter $formatter) {
   return
$formatter->format($hasPrice->getPrice());
}


echo 'Сумма в корзине'    . ' - ' .
formatItemPrice($basket,  new PriceFormatter())
. PHP_EOL;
echo 'Сумма в корзине'    . ' - ' .
formatItemPrice($basket,  new
NumberPriceFormatter()) . PHP_EOL;
echo $product->getName() . ' - ' .
formatItemPrice($product, new PriceFormatter())
. PHP_EOL;
echo $product->getName() . ' - ' .
formatItemPrice($product, new
NumberPriceFormatter()) . PHP_EOL;

//
Сумма в корзине - 13400.448
Сумма в корзине - 13 400.45
Колобок - 1700.348
Колобок - 1 700.35
```

Стоимость в корзине очень важна для нашей страницы, поэтому выведем эту стоимость в теге h1. Благодаря нашей структуре сделать это очень легко.

Добавим новый класс форматтер, и подставим его в параметр нашей функции. Готово.

Мы можем посчитать стоимость корзины, вывести ее в различном формате, а также вывести стоимость товара, в любом другом или том же что и корзина формате.

```
class HtmlNumberPriceFormatter extends
NumberPriceFormatter
{
   public function format($value)
   {
       return '<h1>' . parent::format($value) .
'</h1>';
   }
}

echo 'Сумма в корзине'    . ' - ' .
formatItemPrice($basket,  new
HtmlNumberPriceFormatter()) . PHP_EOL;

//
Сумма в корзине - <h1>13 400.45</h1>
```