Урок 2

ООП в PHP. Расширенное изучение

Продолжение изучения ООП и его реализации в PHP. Архитектурные аспекты ООП. Магические методы, контроль типов, трейты, паттерны.

Ключевое слово parent

При использовании наследования в PHP иногда нужно применять реализованный в классе-родителе функционал частично. Например, когда речь идет о конструкторе. Тогда к реализации можно обратиться при помощи указателя **parent.** Наравне с указателем **self**, который говорит, что мы работаем с текущим классом, указатель **parent** направляет нас на метод в родительском классе:

```
<?php
class BaseClass {
     function __construct() {
          echo "Конструктор класса BaseClass\n";
     }
}
class SubClass extends BaseClass {
     function __construct() {
          parent::__construct();
          echo "Конструктор класса SubClass\n";
     }
}
$obj = new BaseClass();
$obj = new SubClass();
?>
```

Абстрактные классы

Класс, который содержит хотя бы один абстрактный метод, должен быть определен как абстрактный. Такие классы реализуют на практике один из принципов ООП – полиморфизм. Нужно помнить, что нельзя создать экземпляр абстрактного класса. Методы, объявленные абстрактными, лишь описывают смысл и не могут включать реализации.

При наследовании от абстрактного класса все методы, помеченные абстрактными в родительском классе, должны быть переопределены в классе-потомке. Кроме того, область видимости этих методов должна совпадать (или быть менее строгой). Контроль типов (который мы рассмотрим чуть позже) и количество обязательных аргументов должно быть одинаковым.

Пример абстрактного класса:

```php
<?php
abstract class MyAbstractClass {
    /* Данный метод должен быть переопределен в дочернем классе */
    abstract protected function getValue();
    /* Общий метод */
    public function printValue() {
        print $this->getValue() . "\n";
    }
}
class ChildClass extends MyAbstractClass
{
    protected function getValue() {
        return " ChildClass ";
    }
}
$class1 = new ChildClass();
$class1->printValue();
?>
```

Интерфейсы

С помощью интерфейсов можно описать методы, которые должны быть реализованы в классе, без описания их функционала.

Интерфейсы объявляются, как и обычные классы, но с использованием ключевого слова interface. Тела методов интерфейсов должны быть пустыми.

Методы внутри интерфейса должны быть определены как публичные.

Пример описания интерфейса:

```
<?php
// Объявим интерфейс 'CarTemplate'
interface CarTemplate
{
    public function getId();         // Получить id автомобиля
    public function getName(); // Получить название
    public function add();          // Добавить новый автомобиль
}
?>
```

Для реализации интерфейса используется оператор **implements**. Класс должен реализовать все методы, описанные в интерфейсе, иначе произойдет фатальная ошибка. Если нужно, классы могут реализовывать несколько интерфейсов (они должны разделяться запятой).

Пример:

```
<?php
// Объявим интерфейс 'CarTemplate'
class Audi implements CarTemplate {
     function getId() {
          return "1-ATHD98";
    }
    function getName() {
         return "Audi";
    }
    function add() {    }
}
class Bmw implements CarTemplate {
    function getId() {
          return "2-HHFY14";
    }
    function getName() {
         return "BMW";
    }
    function add() {  }
}
?>
```

И абстрактный класс, и интерфейс имеют свои области применения. Рассмотрим их отличия:

Abstract Class	Interface
Абстрактные методы нужно указывать явно с помощью ключевого слова **abstract**	Все методы являются абстрактными
Абстрактные методы можно объявлять с идентификаторами доступа (**public**, **protected**, **private**). При реализации в классе-потомке методы должны иметь такой же модификатор (или менее ограниченный).	Все методы – публичные
Может содержать методы с реализацией, свойства и константы	Может содержать константы
В силу отсутствия множественного наследования, класс может наследовать только один абстрактный класс	Класс может наследовать много интерфейсов

Когда использовать классы, а когда – интерфейсы? Нужно руководствоваться следующим подходом.

Интерфейс описывает поведение и возможности своих реализаций. Обратите внимание на классические названия интерфейсов: **Throwable**, **Countable**, **Comparable**, **Iterable**, **Rollable** (катящийся), **Foldable** (складывающийся). А абстрактный класс описывает сущность – например, стол: **Table_Abstract**. Стол может быть деревянным: **Table_Wood extends Table_Abstract**. Или хирургическим: **Table_Surgical extends Table_Abstract**. В таком случае **Table_Abstract** объединяет общие свойства всех столов (площадь поверхности, наличие ножек и т.п.). А конкретный класс описывает сущность определенного типа столов.

Связь интерфейсов и классов описывает свойства. Например, стол можно катить: **Table_Abstract implements Rollable**. Деревянный стол можно сложить: **Table_Wood implements Foldable**.

Перегрузка и магические методы

Перегрузка в PHP дает возможность динамически «создавать» свойства и методы. Такие методы и свойства обрабатываются с помощью «волшебных» методов, которые можно создать в классе для различных видов действий.

Обращения к свойствам объекта могут быть перегружены с использованием методов **__get** и **__set**. Они будут срабатывать, если объект не содержит свойства, к которому осуществляется доступ.

Синтаксис:

```
<?php
public void __set (string $name , mixed $value)
public mixed __get (string $name)
?>
```

Пример использования:

```
<?php
class MyClass {
      public $c = "c value";
      public function __set($name, $value) {
            echo "__set, property - {$name} is not exists \n";
      }

      public function __get($name) {
            echo "__get, property - {$name} is not exists \n";
      }
}
$obj = new MyClass;
$obj->a = 1;     // Запись в свойство (свойство не существует)
echo $obj->b; // Получаем значение свойства (свойство не существует)
echo $obj->c; // Получаем значение свойства (свойство существует)
?>
```

Результат:

__set, property - a is not exists

__get, property - b is not exists

c value

Вызовы методов могут быть перегружены с использованием методов **__call**. Они будут срабатывать в том случае, если объект не содержит метод, к которому осуществляется доступ. Синтаксис:

```
<?php
public mixed __call (string $name , array $arguments)
?>
```

Пример:

```
<?php
class MyClass {
	public function __call($name, $arguments) {
		return "__call, method - {$name} is not exists \n";
	}
	public function getId() {
		return "АН-15474";
	}
}
$obj = new MyClass;
echo $obj->getName(); // Вызов метода (метод не существует)
echo $obj->getId();   // Вызов метода (метод существует)
?>
```

Результат:

__call, method - getName is not exists (при вызове **getName**)

АН-15474 (при вызове **getId**)

Магический метод __toString

Метод **__toString()** будет срабатывать при попытке преобразования класса в строку. Например, **echo $obj;**.

Синтаксис:

```
<?php
public string __toString ()
?>
```

Пример:

```
<?php
class MyClass {
	public function __toString() {
		return "MyClass class";
	}
}
$obj = new MyClass;
echo $obj; // Результат: MyClass class
?>
```

Контроль типа

В PHP 5 можно использовать контроль типов. При передаче параметром есть возможность проверить данные на следующие типы: объекты (указывая имя класса в прототипе функции), интерфейсы, массивы, колбеки с типом **callable** (начиная с PHP 5.4).

Начиная с PHP 7, можно принимать тип возвращаемого значения и добавлять туда скалярные типы.

Синтаксис вызова – двоеточие с аргументом-суффиксом перед скобками.

```
function isValidStatusCode(int $statusCode): bool {
      return isset($this->statuses[$statusCode]);
}
```

Пример использования:

```
<?php
class MyClass {
      public function names(array $names) {
// Тип Array
            $res = "<ul>";
            foreach($names as $name) {
                  $res .= "<li>{$name}</li>";
            }
            return $res .= "</ul>";
      }
      public function otherClassTypeFunc(OtherClass $otherClass) { // Тип
OtherClass
            return $otherClass->var1;
      }
}
$obj = new MyClass;
$names = array(
      'Иван Андреев',
      'Олег Симонов',
      'Андрей Ефремов',
      'Алексей Самсонов'
);
echo $obj->names($names);
// Работает
$names = "Олег Симонов";
// Получим фатальную ошибку: Argument 1 passed to MyClass::names() must be of
the type array, string given
echo $obj->names($names);
// Получим фатальную ошибку: Argument 1 passed to MyClass::names() must be an
instance of OtherClass, string given
echo $obj->otherClassTypeFunc("test string");
?>
```

Пространства имен

Пространства имен – это один из способов инкапсуляции элементов. Такое абстрактное понятие можно увидеть во многих местах. Например, в операционной системе директории служат для группировки файлов и выступают в качестве пространства имен для находящихся в них файлов.

В качестве примера файл **text.txt** может находиться сразу в нескольких директориях: **/files** и **/docs**, но две копии **text.txt** не могут существовать в одной директории. Для доступа к **text.txt** извне мы должны добавить имя директории перед именем файла, используя разделитель (**/files /text.txt**). Такой же принцип распространяется и на пространства имен.

В PHP пространства имен используются для решения двух проблем:

- конфликт имен между вашим кодом и сторонними;
- возможность создавать псевдонимы (или сокращения) для длинных имен, чтобы облегчить первую проблему и улучшить читаемость исходного кода.

Пример использования

Допустим, у нас такая файловая структура:

-- App

--- Main

---- MyClass.php

- namespace.php

Опишем класс **MyClass.php**:

```
<?php
// App/Main/MyClass.php
namespace App\Main;
class MyClass {
	function hello() {
		return "hello";
	}
}
?>
```

С помощью пространств имен мы можем получить доступ к классу **MyClass** (файл **namespace.php**):

```
<?php
// namespace.php
namespace App\Main;
require_once "App/Main/MyClass.php";
$obj = new \App\Main\MyClass;
echo $obj->hello(); // hello
?>
```

Исходя из описания, мы можем создать такой же класс, только в другой директории. Создадим класс с таким же названием в папке **App/Core**:

```
<?php
// App/Core/MyClass.php
namespace App\Core;
class MyClass {
	function hello() {
		return "hello, it's core";
	}
}
?>
```

Получим доступ к этому классу:

```
<?php
namespace App\Core;
require_once "App/Core/MyClass.php";
$obj = new \App\Core\MyClass;
echo $obj->hello(); // hello it's core
?>
```

Создание псевдонима имени

Псевдонимы для пространств имен используются для более простого доступа к нужному пространству.

Например, у вас такая структура: **namespace App/Core/Controller/**. Чтобы получить доступ к одному из классов, нужно будет написать весь этот путь: **App/Core/Controller/AppController.php**. Намного проще было бы написать **CoreController/AppController.php**. Это можно реализовать с помощью псевдонимов.

Для создания псевдонима используют ключевое слово **use**.

Пример:

```
<?php
use App/Core/Controller as CoreController;
// …
$app = new CoreController\AppControooler.php;
?>
```

Трейты

С версии 5.4.0 в PHP появился инструментарий для повторного использования кода – **трейт**.

Трейт похож на класс, но предназначен для группирования функционала структурированным образом. Невозможно создать самостоятельный экземпляр трейта. Это дополнение к обычному наследованию, то есть возможность использовать функционал класса без наследования.

Пример написания:

```
<?php
trait MyTrait {
      public function myFunc() {
            return 2 + 2;
      }
}
?>
```

Использование

Рассмотрим пример использования трейта на основе создания транслитератора (переводит буквы кириллицы в латиницу):

```
<?php
trait MyTransliterator {
      private $letters = array(
        'а' => 'a',      'б' => 'b',      'в' => 'v',
        'г' => 'g',      'д' => 'd',        'е' => 'e',
        'е' => 'e',      'ж' => 'zh',       'з' => 'z',
        'и' => 'i',      'й' => 'y',        'к' => 'k',
        'л' => 'l',      'м' => 'm',        'н' => 'n',
        'о' => 'o',      'п' => 'p',        'р' => 'r',
        'с' => 's',      'т' => 't',        'у' => 'u',
        'ф' => 'f',      'х' => 'h',        'ц' => 'c',
        'ч' => 'ch',     'ш' => 'sh',       'щ' => 'sch',
        'ь' => '',       'ы' => 'y',        'ъ' => '',
        'э' => 'e',      'ю' => 'yu',       'я' => 'ya',
        'А' => 'A',      'Б' => 'B',        'В' => 'V',
        'Г' => 'G',      'Д' => 'D',        'Е' => 'E',
        'е' => 'E',      'Ж' => 'Zh',       'З' => 'Z',
        'И' => 'I',      'Й' => 'Y',        'К' => 'K',
        'Л' => 'L',      'М' => 'M',        'Н' => 'N',
        'О' => 'O',      'П' => 'P',        'Р' => 'R',
        'С' => 'S',      'Т' => 'T',        'У' => 'U',
        'Ф' => 'F',      'Х' => 'H',        'Ц' => 'C',
        'Ч' => 'Ch',     'Ш' => 'Sh',       'Щ' => 'Sch',
        'Ь' => '',       'Ы' => 'Y',        'Ъ' => '_',
        'Э' => 'E',      'Ю' => 'Yu',       'Я' => 'Ya',
```

```
        'є' => 'ye',     'ї' => 'yi',       'і' => 'i',
        'Є' => 'YE',     'Ї' => 'YI',       'І' => 'I',
        ' ' => '_'
    );
      public function translate($str) {
            // Заменяем символы кириллицы на латиницу
            return strtr(trim($str), $this->letters);
      }
}
class MyClass {
      use MyTransliterator;
      private $data;
      /**
      *     Некая функция для добавления данных в наш массив
      */
      public function setData(array $data) {
            $this->data = $data;
      }
      /**
      *      Некая функция для подготовки данных
      */
      public function getPreparedData() {
            // Допустим, мы хотим сделать адрес страницы по названию
            // Тогда нам нужно перевести название с кириллическими символами
на латиницу
            $this->data['url']                                              =
strtolower($this->translate($this->data['title']));
            return $this->data;
      }
}
$obj = new MyClass;
$obj->setData([
      'title' => 'Не очень простое название для страницы',
      'content' => 'Текст страницы'
]);
$data = $obj->getPreparedData();
echo "<pre>";
print_r($data);
echo "</pre>";
?>
```

Результат:

```
Array
(
   [title] => Не очень простое название для страницы
    [content] => Текст страницы
    [url] => ne_ochen_prostoe_nazvanie_dlya_stranicy
)
```

Паттерн Singleton

Шаблон проектирования или **паттерн** в разработке программного обеспечения – этоповторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках часто возникающего контекста.

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код. Это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

«Низкоуровневые» шаблоны, учитывающие специфику конкретного языка программирования, называются идиомами. Это хорошие решения проектирования, характерные для конкретного языка или программной платформы, и поэтому не универсальные.

На наивысшем уровне существуют **архитектурные шаблоны** – они охватывают архитектуру всей программной системы.

Алгоритмы по своей сути тоже являются шаблонами, но не проектирования, а вычисления, так как решают вычислительные задачи.

Одиночка (англ. **Singleton**) – шаблон проектирования, гарантирующий, что в однопоточном приложении будет единственный экземпляр класса с глобальной точкой доступа. Его плюс – удобное создание контролируемого доступа к единственному экземпляру класса в программе. К минусам стоит отнести то, что этот объект практически всегда глобальный. А это может быть вредно для объектного программирования, так как в некоторых случаях приводит к созданию немасштабируемого проекта.

Итак, у нас должен быть единственный экземпляр класса, легко доступный всем клиентам. Он должен расширяться, порождая подклассы, и клиентам нужна возможность работать с расширенным экземпляром без модификации своего кода.

```php
<?php
 class Singleton {
    private static $instance;  // Экземпляр объекта
// Защищаем от создания через new Singleton
    private function __construct() { /* ... @return Singleton */ }
// Защищаем от создания через клонирование
                private function __clone() { /* ... @return Singleton */ }
// Защищаем от создания через unserialize
     private function __wakeup()  { /* ... @return Singleton */ }
// Возвращает единственный экземпляр класса. @return Singleton
      public static function getInstance() {
            if ( empty(self::$instance) ) {
            self::$instance = new self();
        }
        return self::$instance;
    }
    public function doAction() { }
 }
/* Применение*/
 Singleton::getInstance()->doAction();
?>
```

Это решение было актуальным до появления трейтов в версии 5.4, предоставившей другую реализацию паттерна. Попытайтесь реализовать ее в домашнем задании!

Практическое задание

1. Создать структуру классов ведения товарной номенклатуры.
    a. Есть абстрактный товар;
    b. Есть цифровой товар, штучный физический товар и товар на вес;
    c. У каждого есть метод подсчета финальной стоимости;
    d. У цифрового товара стоимость постоянная. У штучного товара стоимость зависит от количества штук, у весового – в зависимости от продаваемого количества в килограммах. У всех формируется в конечном итоге доход с продаж.

    Что можно вынести в абстрактный класс, наследование?
2. * Реализовать паттерн **Singleton** при помощи **traits**.

Дополнительные материалы

1. https://geekbrains.ru/events/285 – о профессиональном использовании пространств имен в php.
2. Мэт Зандстра. PHP. Объекты, шаблоны и методики программирования.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Мэтт Зандстра. PHP. Объекты, шаблоны и методики программирования.
2. Мэтт Вайсфельд. Объектно-ориентированное мышление.