Въведение в PHP

Обектно-ориентирано програмиране

Георги Гроздев
g.grozdev@viscomp.bg
Атанас Василев
a.vasilev@viscomp.bg

www.viscomp.bg

About us

- Лектори
  - Георги Гроздев – g.grozdev@viscomp.bg
  - Атанас Василев – a.vasilev@viscomp.bg

- Инфо и слайдовете ще намерите на:
  - http://phplab.viscomp.bg

viscomp
we develop the web

ООП

- Класово-базирани езици
- Прототипно-базирани езици

ООП

- Класово-базирани езици
  - Обектите са два типа: клас и инстанция
  - класът обединява структурата и поведението на обекта
  - инстанцията представлява състоянието (данните) на обекта
  - Йерархията се осъществява чрез класова верига
  - Дефиницията на класа определя всички свойства на всички негови инстанции. (Принципно) не могат да се добавят нови свойства динамично. Не могат да се променят свойства на вече инстанциирани обекти чрез промяна на свойство в родителския обект
  - Създаване на обекти – посредством конструктори и евентуално подадените им аргументи. Получената инстанция е базирана на структурата и поведението, определени от избрания клас
  - Привържениците на прототипно-базираното програмиране смятат, че класово базираните езици насърчават модел на разработка, който най-вече се фокусира върху таксономията и връзката между класовете. От друга страна, прототипно-базираният модел се счита, че провокира програмистите отначало да се фокусират върху поведението на няколко отделни обекта и едва след това да се опитват да ги класифицират в по-общи архетипни обекти, които по-късно да се ползват по подобен на класовете начин.

ООП

- Прототипно-базирани езици
  - Липсват класове – всички обекти са инстанции
  - Йерархията се осъществява посредством прототипна верига
  - Прототипът определя само първоначалните свойства – впоследствие динамично могат да се добавят или променят свойства както на отделни обекти, така и на всички свързани чрез общ прототип
  - Създаване на обекти
    - клониране на съществуващ обект
    - *ex nihilo* (от нищото)

```
var foo = {one: 1, two: 2};
var bar = {three: 3};

// Gecko & Webkit
bar.__proto__ = foo; // bar is now the child of foo

// Opera, IE
child.prototype = new Parent;
```

ООП

- Клас
  - Най-общо - съвкупност от данни и код. Обединява структурата и поведението на обектите.

```
class Dog
{
    public $name;
    public $breed = 'Golden retriever';

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

  - Стойностите по подразбиране трябва да са константни изрази, а не променлива, извикване на функция или метод на клас

ООП

- Инстанция
  - Използваем обект, базиран на шаблона на определен клас.

```
$lassie = new Dog('Lassie');
```

- Метод
  - Способностите на обекта

```
$lassie->bark();
$lassie->sit();
```

  - В рамките на програмата извикването на метод обикновено касае един обект. Всички кучета могат да лаят, но ние искаме едно определено куче да излае в дадения момент.

- Предаване на съобщения
  - Процес, при който един обект изпраща данни на друг обект или изисква от него да изпълни даден метод. Например обект Breeder може да изпрати съобщение sit към обект Lassie, което да го накара да изпълни своя sit() метод

ООП

- Наследяване
  - Дъщерните класове са специализирани версии на базовия клас, които наследяват свойства и поведение от него, като също така могат да дефинират свои собствени такива.

```
class Dog extends Animal {
    public function __construct($name) {
        $this->name = $name;
        parent::__construct();
    }
}
```

ООП

- Абстракция
  - Опростяване на комплексен проблем, чрез моделиране на класовете по подходящ за него начин и чрез работа на най-подходящото ниво на наследяване за всеки отделен аспект на проблема.
  - Кучето Lassie може да се третира като Dog през по-голямата част от времето, като Collie, когато искаме да достъпим специфични за Коли свойства и поведения, както и като Animal, когато броим домашните любимци на стопанина.
  - Абстракция чрез композиция – клас Car може да включва отделни компоненти като Двигател, Скоростна кутия, Кормилна уредба и т.н. За да изградим класа Car не е нужно да познаваме как точно работят отделните компоненти вътрешно за себе си – достатъчно е да знаем как да взаимодействаме с тях – да изпращаме съобщения към тях и да получаваме съобщения от тях.

ООП

- Енкапсулиране
  - Скриване на функционалните особености на един клас от обектите, които изпращат съобщения към него
  - Класът Dog има метод bark(). Кодът на този метод описва как точно да се осъществи лаенето – например inhale(), последвано от exhale() с определена сила и честота на гласа. Стопанинът на Lasie обаче, не се интересува от това, как точно става лаенето.
  - Енкапсулирането се постига чрез дефиниране на това, кои класове трябва да имат достъп до методите на даден обект. Като резултат казваме, че даден обект показва пред конкретния клас определен свой интерфейс – методите, достъпни за него.
  - Обосновката за енкапсулирането е да се предпазят клиентите на даден интерфейс от обвързването им с такива негови вътрешни части, които е вероятно да бъдат променяни в бъдеще. По този начин промените ще бъдат безболезнени – т.е. няма да се налагат промени и в кода на клиентите на интерфейса.
  - Интерфейсът би могъл да гарантира например, че нови кученца ще се добавят към обект от клас Dog само от код в същия клас. За целта се ползват ключовите думи *public*, *protected* и *private*, които ограничават видимостта на членовете на един клас

ООП

- Полиморфизъм
  - Полиморфизмът позволява да третираме членове на дъщерен клас като членове на родителския клас. По-точно полиморфизмът в ООП е способността на обекти, принадлежащи към различни типове данни да отговарят на извиквания на методи с еднакви имена, но да предизвикват поведение, специфично за конкретния тип.
  - *Overriding полиморфизъм*
    - Ако накараме куче да говори – speak(), това може да доведе до изпълнението на bark(); ако пък накараме прасе да говори – speak(), то ще изпълни oink(). Както Dog, така и Pig наследяват speak() от Animal, но методите в тях презаписват (override) този в родителския клас.
  - *Overloading полиморфизъм*
    - Една и съща декларация на метод, или един и същи оператор изпълняват няколко различни функции в зависимост от конкретната имплементация. В JavaScript операторът '+' може да служи както за събиране на числа, така и за слепване на низове.

ООП

- Разделяне (Decoupling)
  - Представлява разделянето на функционални части (блокове), които не би трябвало да зависят един от друг на различни нива на абстракция. Някои градивни единици са общи и не се интересуват от детайлите на други части от цялото. Често енкапсулирани компоненти се разделят полиморфно, което означава, че използваме код за многократна употреба за да предотвратим взаимодействието между различните дискретни модули.

ООП

- Конструктор и Деструктор

```
class Dog extends Animal {
    public function __construct($id) {
        $this->data = $this->find($id);
        parent::__construct();
    }
    public function __destruct() {...}
}
```

  - Съвместимост с PHP4 – конструктор с името на класа

ООП

- Видимост на свойствата и методите
  - public – достъпни отвсякъде
  - protected – достъпни от класа, в който са дефинирани, както и от неговия базов и дъщерни класове
  - private – достъпни само от класа, в който са дефинирани

```
class MyClass
{
    public $public = 'Public';
    protected $protected = 'Protected';
    private $private = 'Private';

    function printHello() {
      echo $this->public;
      echo $this->protected;
      echo $this->private;
    }
}
$obj = new MyClass();
echo $obj->public;          // Работи
echo $obj->protected;       // Fatal Error
echo $obj->private;         // Fatal Error
$obj->printHello();         // Public, Protected и Private
```

ООП

- Статични свойства и методи
  - До поле, декларирано като статично не може да се достигне през инстанция на класа – може само чрез статичен метод
  - Променливата $this не е достъпна в метод, деклариран като статичен
  - Достъп до статични свойства на обект не може да се осъществи чрез '->'
  - Статични свойства могат да бъдат инициализирани само с низ или константа
  - Статично извикване на нестатичен метод ще доведе до предупреждение

```
class Foo {
    public static $my_static = 'foo';
    public static function aStaticMethod() {
        print self::$my_static;
    }
}
Foo::aStaticMethod();
```

ООП

- Класови константи

```
class MyClass
{
    const constant = 'стойност на константата';

    function showConstant() {
        echo  self::constant . "\n";
    }
}

echo MyClass::constant . "\n";
```

ООП

- Абстрактни класове
  - Не може да се инстанциират директно
  - Абстрактните методи имат прототип, но не и имплементация
  - Ако клас има поне един абстрактен метод, то целия клас трябва да се дефинира като абстрактен
  - При наследяване на абстрактен клас всички методи, декларирани като абстрактни трябва да се имплементират и то със същата или по-слабо рестриктивна видимост

ООП

- Абстрактни класове

```
abstract class AbstractClass {
    // Методи, които трябва да бъдат дефинирани в дъщерния клас
    abstract protected function getValue();
    abstract protected function prefixValue($prefix);

    // Общ метод
    public function printOut() {
        print $this->getValue() . "\n";
    }
}

class ConcreteClass1 extends AbstractClass {
    protected function getValue() {
        return "ConcreteClass1";
    }

    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass1";
    }
}
```

ООП

- Интерфейси
  - Дефинират методите, които даден клас задължително трябва да реализира, без да се декларират самите тела на тези методи
  - Всички методи, дефинирани в даден интерфейс, трябва да бъдат public
  - За да се укаже, че даден клас реализира определен интерфейс, се използва операторът *implements*
  - Интерфейсите могат да бъдат разширявани, също както класовете, посредством оператора *extend*
  - Даден клас не може да реализира два интерфейса, ако те имат методи с еднакви имена, тъй като това води до неопределеност.

ООП

- Интерфейси

```
// Декларация на интерфейс 'iTemplate'
interface iTemplate {
    public function setVariable($name, $var);
    public function getHtml($template);
}
// Реализиране на интерфейса. Това ще работи
class Template implements iTemplate {
    private $vars = array();

    public function setVariable($name, $var) {
        $this->vars[$name] = $var;
    }

    public function getHtml($template) {
        foreach($this->vars as $name => $value) {
            $template = str_replace('{' . $name . '}',
            $value, $template);
        }
        return $template;
    }
}
```

ООП

- Разширяеми интерфейси

```
interface a {
    public function foo();
}

interface b extends a {
    public function baz(Baz $baz);
}

// Това ще работи
class c implements b {
    public function foo() { }

    public function baz(Baz $baz) { }
}
```

ООП

- Интерфейси – множествено наследяване

```
interface a {
    public function foo();
}

interface b {
    public function bar();
}

interface c extends a, b {
    public function baz();
}

class d implements c {
    public function foo() { }
    public function bar() { }
    public function baz() { }
}
```

Въпроси?

Благодаря ви за вниманието!