

1. Обектно-ориентирано програмиране	2
1.1 Прототипно- и класово-базирани езици за програмиране	3
1.2 Основни принципи	4
1.3 Терминология	4
1.4 Абстрактни класове	8
1.5 Интерфейси	9
1.6 Traits	11
2. Задача - Шахматни диаграми	14

# Обектно-ориентирано програмиране

## Лектори

- Димитър Ников - [d.nikov@viscomp.bg](mailto:d.nikov@viscomp.bg)
- Атанас Василев - [a.vasilev@viscomp.bg](mailto:a.vasilev@viscomp.bg)

Новини, лекции, задачи и други материали ще намерите на адрес <http://phplab.viscomp.bg>



# Прототипно- и класово-базирани езици за програмиране

## 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  
bar.prototype = new foo;
```

## 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.

# Основни принципи

## 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.

## Капсулиране

- 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 `Lassie` обаче, не се интересува от това, как точно става лаенето.
- Капсулирането 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 капсулирането е да се предпазят клиентите на даден интерфейс от обвързването им с такива негови вътрешни части, които е вероятно да бъдат променени 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.

## 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 пате да говори – `speak()`, то ще изпълни `quack()`. Както `Dog`, така и `Duck` наследяват `speak()` от `Animal`, но методите в тях презаписват тези в родителския клас.

### Overloading полиморфизъм

Дъщерният клас може да предефинира метод от базовия клас, но с различни по брой или тип параметри. Не се поддържа директно в PHP.

## 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 капсулирани компоненти се разделят полиморфно, което означава, че използваме код за многократна употреба за да предотвратим взаимодействието между различните дискретни модули.

- Dependency Injection

## Терминология

### 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.

## Инстанция

- Използваем обект, базиран на шаблона на определен клас.

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

## Метод

- Поведението на обекта. Неговите способности

```
$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.

## 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();
    }
}
```

## Конструктор и Деструктор

```

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

```

## 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';

    public function printHello() {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}

$obj = new MyClass();

// 'Public'
echo $obj->public;

// Fatal Error
echo $obj->protected;

// Fatal Error
echo $obj->private;

// 'Public', 'Protected', 'Private'
$obj->printHello();

```

## 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 литерал 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();
```

## Класови константи

```
class MyClass
{
    const CONSTANT_NAME = '  ';

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

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

## 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.

```
abstract class AbstractClass
{
    // These should be defined in derived classes
    abstract protected function getValue();
    abstract protected function prefixValue($prefix);

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

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

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



# Интерфейси

- Дефинират методите, които даден клас задължително трябва да реализира, без да се декларира 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 `extends`.
- Даден клас не може да реализира два интерфейса, ако те имат методи с еднакви имена, тъй като това води до неопределеност.

```
interface iTemplate
{
    public function setVariable($name, $var);
    public function getHtml($template);
}

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;
    }
}
```

## Наследяване на интерфейси

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

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

class c implements b
{
    public function foo() { }

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

## Множествено наследяване на интерфейси

```
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() { }
}
```

# Traits



## Traits (PHP 5.4+)

Механизъм, позволяващ използване на код в езици като PHP, които не поддържат множествено наследяване. Преодолява някои от ограниченията на единичното наследяване, като позволява на програмиста свободно да използва множество от методи в няколко независими класа, членове на различни класови йерархии.

### Private членовете на класа са достъпни за трейтовете

```
trait Singleton
{
    private static $instance;

    public static function getInstance() {
        if (!(self::$instance instanceof self)) {
            self::$instance = new self;
        }
        return self::$instance;
    }
}

class DbReader extends ArrayObject
{
    use Singleton;
}

class FileReader
{
    use Singleton;
}

$a = DbReader::getInstance();
$b = FileReader::getInstance();

var_dump($a); //object(DbReader)
var_dump($b); //object(FileReader)
```

### Multiple Traits

```
trait Hello
{
    function sayHello() {
        echo "Hello";
    }
}

trait World
{
    function sayWorld() {
        echo "World";
    }
}

class MyWorld
{
    use Hello, World;
}

$world = new MyWorld();
echo $world->sayHello() . " " . $world->sayWorld(); //Hello World
```

## Трейтове, съставени от трейтове

```
trait HelloWorld
{
    use Hello, World;
}

class MyWorld
{
    use HelloWorld;
}

$world = new MyWorld();
echo $world->sayHello() . " " . $world->sayWorld(); //Hello World
```

## Приоритет на методите

1. метод от трейт има по-висок приоритет от метод със същото име, наследен от родителски клас
2. метод, дефиниран в текущия клас има по-висок приоритет от същия метод, дефиниран в трейт

## Приоритет на методите

```
trait Hello
{
    function sayHello() {
        return "Hello";
    }

    function sayWorld() {
        return "Trait World";
    }

    function sayHelloWorld() {
        echo $this->sayHello() . " " . $this->sayWorld();
    }

    function sayBaseWorld() {
        echo $this->sayHello() . " " . parent::sayWorld();
    }
}

class Base
{
    function sayWorld(){
        return "Base World";
    }
}

class HelloWorld extends Base
{
    use Hello;
    function sayWorld() {
        return "World";
    }
}

$h = new HelloWorld();
$h->sayHelloWorld(); // Hello World
$h->sayBaseWorld(); // Hello Base World
```

## Разрешаване на конфликт с имена на методи

Когато трейт вмъкне метод, който вече е дефиниран в друг трейт имаме конфликт с имената на методите. Имплементиращият клас няма как да реши, кой от двата метода да изпълни при извикване.

### Временно изключване на метод

```
class Player
{
  use Game, Music {
    Music::play insteadof Game;
  }
}

$player = new Player();
$player->play(); //Playing music
```

### Алиас на метод

```
class Player
{
  use Game, Music {
    Game::play as gamePlay;
    Music::play insteadof Game;
  }
}

$player = new Player();
$player->play(); //Playing music
$player->gamePlay(); //Playing a game
```















# Задача - Шахматни диаграми

Да се напише програма, която да генерира шахматна позиция при зададен FEN низ.

[Описание на FEN нотацията в Wikipedia.](#)

Шахматната позиция трябва да представлява таблица, в която във всяка клетка да има по една от предварително дадени 26 картинки, именувани според следната конвенция:

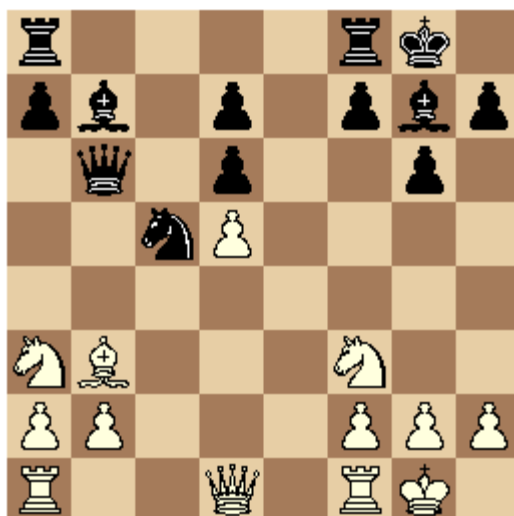
Файл	Фигура (EN)	Фигура (BG)	
bbb.gif	Black Bishop on Black	Черен офицер на черно поле	
bbw.gif	Black Bishop on White	Черен офицер на бяло поле	
bkb.gif	Black King on Black	Черен цар на черно поле	
bkw.gif	Black King on White	Черен цар на бяло поле	
bnb.gif	Black kNight on Black	Черен кон на черно поле	
bnw.gif	Black kNight on White	Черен кон на бяло поле	
bpb.gif	Black Pawn on Black	Черна пешка на черно поле	
bpw.gif	Black Pawn on White	Черна пешка на бяло поле	
bqb.gif	Black Queen on Black	Черна дама на черно поле	
bqw.gif	Black Queen on White	Черна дама на бяло поле	
brb.gif	Black Rook on Black	Черен топ на черно поле	
brw.gif	Black Rook on White	Черен топ на бяло поле	

eb.gif	Empty Black	Празно черно поле	
ew.gif	Empty White	Празно бяло поле	
wbb.gif	White Bishop on Black	Бял офицер на черно поле	
wbw.gif	White Bishop on White	Бял офицер на бяло поле	
wkb.gif	White King on Black	Бял цар на черно поле	
wkw.gif	White King on White	Бял цар на бяло поле	
wnb.gif	White kNight on Black	Бял кон на черно поле	
wnw.gif	White kNight on White	Бял кон на бяло поле	
wpb.gif	White Pawn on Black	Бяла пешка на черно поле	
wpw.gif	White Pawn on White	Бяла пешка на бяло поле	
wqb.gif	White Queen on Black	Бяла дама на черно поле	
wqw.gif	White Queen on White	Бяла дама на бяло поле	
wrb.gif	White Rook on Black	Бял топ на черно поле	
wrw.gif	White Rook on White	Бял топ на бяло поле	

Например при зададени FEN низове :

- r4rk1/pb1p1pbb/1q1p2p1/2nP4/8/NB3N2/PP3PPP/R2Q1RK1
- r2r4/5kpp/1np1B3/4Pp2/1q1Q4/pP1R3P/2P2PP1/R5K1

... програмата трябва да изведе следните диаграми:



FEN низът описва разположението на фигурите от гледната точка на белите.

Описва се всеки ред, като се започва от 8-ми (най-горе) и се свършва с 1-ви (най-долу).

В рамките на всеки ред се описва всяка фигура, като се започва от колона А и се свършва с колона Н.

Съгласно стандартната алгебрична нотация, всяка фигура се указва с една буква от английските наименования на фигурите:

- pawn (пешка) = "P"
- knight (кон) = "N"
- bishop (офицер) = "B"
- rook (топ) = "R"
- queen (дама) = "Q"
- king (цар) = "K"

Белите фигури се обозначават с главни букви ("PNBRQK"), а черните - с малки ("pnbrqk").

Празните полета се описват с цифри от 1 до 8 (указващи броя съседни празни полета).

Редовете са разделени с "/"

Задачата да се реши с обектно-ориентиран подход. Целевият интерфейс е:

```
$b = new Board( 'r4rk1/pb1p1pbp/1q1p2p1/2nP4/8/NB3N2/PP3PPP/R2Q1RK1' );
$b->render();
```