

PDA | O portálu | Podpořte nás | Ke stažení | Redakce | Kontakt | Inzerce | Partneři |

Registrace | Login

Přídejte se

i h

programujte
zaměřeno na informační technologie

Clánky | Diskuzní fórum | Kritika webů | Podcast | Kalendář akcí | Články na přání | E-shop | Login/Nový účet

Anketa

Jakou agregační službu používáte pro sledování nových článků?

Programujte » Webdesign » PHP

Přidat do vlastních článků

OOP v PHP

09. 12. 2009 | 01:00 - Jakub Kulhan (bukaj) - 13250× přečteno

Dostali jste se k PHP a najednou se na vás řítí ze všech směrů, že jste hnusný bastlíč, protože neprovozujete to krásné OOP, ale pachtíte se s nějakými procedúrkami? Tento článek by vám měl stručně, jasně a na příkladech vysvětlit, jak ukočírovat objektový svět PHP.

gugCAMP 30.4.

Článek bude rozčleněn do několika částí. Nejdříve se podíváme na základy syntaxe, která bude pro OOP potřeba – v podstatě základ k tomu, abyste mohli využít nějakou tu knihovnu, pro kterou její autor zvolil objektový styl programování. Dále se již vrhneme na modelování samotných objektů – jednotlivá témata budou vysvětlena na doufám dost názorných příkladech.

Vše je psané v PHP 5 a pro PHP 5. PHP 4 nechť archeologům, jeho objektový model je velice omezený a nemá cenu se jím již více zabývat.

Tento článek není úvod do celé problematiky programování, takže byste měli znát PHP syntaxi týkající se procedurálního programování, neměl by pro vás být problém pochopit, co je to podmínka, cyklus, jak se volají funkce, co je návratová hodnota atp.

Co je to OOP a proč by mě mělo zajímat?

Je to jeden ze stylů (neboli paradigma) imperativního (neboli „podmínky a cykly“) programování, jež pracuje se základní jednotkou zvanou objekt. Objekt je něco, co dokáže udržovat svůj stav a interagovat s okolím zasíláním a přijímáním zpráv. OOP vzniklo jako reakce na stále se zvyšující složitost programů s cílem usnadnit jejich psaní.

Člověka by mělo zajímat hlavně proto, že dnes je to majoritní styl programování, a pokud chce použít některé knihovny, prostě se bez alespoň základních znalostí neobejde.

Základy syntaxe

Vyhledávání »

Hledat

fotografie, id3, ping, thief,
omfg, gamesy, miranda
pack, partial, revoke

Přihlášení »

geniv

.....

Přihlásit se

Založit nový účet

Zapomenuté heslo

Výhody registrace

Reklama »

Reklama »

www.jobs.cz

SPOJENÍ S FUTUREM

Práce v IT

■ Programátor .NET

■ IT - programátor

■ Analyst/developer J2EE - senior

■ Product Manager - strukturovaná kabeláž a optika

■ Windows specialista

■ PL/SQL vývojář - junior

více nabídek práce »

Gopas & DAQUAS & Microsoft si Vás dovolují pozvat na největší IT odbornou konferenci TechEd 2010 TECHNICAL EDUCATION & DEVELOPER DATABASE CONFERENCE 2010 ve dnech 26. - 29. dubna 2010 v multiplexu CineStar Černý Most.
www.tech-ed.cz

Anketa »

Abychom mohli pracovat s objektem, musíme ho nejdříve vytvořit. K tomu slouží operátor `new`:

```
Ukázat zdrojový kód Zkopírovat do schránky Tisk
01. $objekt = new Trida;
```

Těsně za operátorem `new` následuje název třídy (třída je šablona budoucího objektu, více o třídách později). Název třídy může být buď posloupnost znaků vyhovujících regulárnímu výrazu `[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*` (řečí smrtelníků: písmeno anglické abecedy, podtržítka či nějaký znak z horní řady ASCII následovaný žádným nebo více písmeny anglické abecedy nebo čísly, podtržítka či znaky z horní řady ASCII), nebo proměnná. Jedná-li se o proměnnou, převede se nejdříve obsah této proměnné na řetězec a jako název třídy se vezme takto získaná hodnota:

```
Ukázat zdrojový kód Zkopírovat do schránky Tisk
01. $trida = "Trida";
02. $objekt = new $trida;
03. echo get_class($objekt); // vytiskne Trida
```

Vězte, že funkce `get_class()` vrací název třídy daného objektu.

Podobně jako při volání funkce můžete za název třídy přidat do závorek seznam parametrů, se kterými bude třída inicializována:

```
Ukázat zdrojový kód Zkopírovat do schránky Tisk
01. $objekt = new Trida($param1, $param2, ...);
```

Co se stane s těmito parametry, se dozvíte dále ve článku.

Objekt samotný je k ničemu, takže přichází na řadu interakce s okolním světem a posílání zpráv. PHP zná celkem tři základní typy zpráv:

1. získání hodnoty vlastnosti (atributu, property)

```
Ukázat zdrojový kód Zkopírovat do schránky Tisk
01. $objekt->nazev_vlastnosti;
```

2. nastavení hodnoty vlastnosti

```
Ukázat zdrojový kód Zkopírovat do schránky Tisk
01. $objekt->nazev_vlastnosti = "hodnota"; // může být číslo (kupř. 2), proměnná ($f
```

3. volání metody

```
Ukázat zdrojový kód Zkopírovat do schránky Tisk
01. $objekt->nazevMetody($param1, $param2, ...);
```

Pro zasílání zpráv, jak je vidět, je používán šipkový operátor `->`. Za ním opět může následovat sekvence znaků stejná jako v případě názvu třídy. A stejně tak lze místo této sekvence použít proměnnou, kdy se vezme její textový obsah. Takže např.:

```
Ukázat zdrojový kód Zkopírovat do schránky Tisk
01. $super_metoda = "nazevMetody";
02. $objekt->$super_metoda($param1);
```

Důležitou konstrukcí, se kterou se můžete setkat a velice často setkáte, je možnost řetězení volání metod (taktéž je vidět, že na bílé znaky kolem operátoru šipky se nebere zřetel):

Jakou agregační službu používáte pro sledování nových článků?

Desktopovou RSS čtečku - 10042 hl.

Google Reader - 9920 hl.

PraveDnes.cz - 9561 hl.

Weblogy.cz - 9616 hl.

ITzpravy.cz - 9872 hl.

Počet hlasů: 49011

Newsletter »

Máte-li zájem přihlášení/odhlášení odběru nových článků na Programujte, můžete si jej zdarma nechat posílat 1x týdně na svoji emailovou adresu:

@
Uložit

Projekty »

Diskuzní fórum
Diskuze o portále

Tkinter

Zpravodajství v oblasti IT **NOVINKA**

Partnerské portály »

Soom.cz - počítačová bezpečnost
ER.cz - přesměrování zdarma
Vyvojar.cz - vývojáři sobě

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. $vystupni_hodnota = $objekt
02.                 ->prvniMetoda()
03.                 ->druhaMetoda();
```

Řekněme, že definice prvniMetody vypadá následovně:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. function prvniMetoda()
02. {
03.     $novy_objekt = new DalsiTrida;
04.     return $novy_objekt;
05. }
```

Vytvořili jsme nějaký \$novy_objekt a ten vrátili. druhaMetoda je tedy volána na tomto novém objektu, nikoli na původním. Jestliže je druhaMetoda ve třídě DalsiTrida definována jako:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. function druhaMetoda()
02. {
03.     return "Hello, world!";
04. }
```

Pak proměnná \$vystupni_hodnota bude obsahovat řetězec "Hello, world!".

Bylo řečeno, že objekt je cosi, co udržuje svůj stav. Ale stejně tak to dokáže i třída. Dá se říci, že třída je globálně přístupný objekt s jasně definovaným jménem. Stejně jako objekt umí i třída přijímat tři základní typy zpráv. Ale syntaxe se liší – místo šipkového operátoru (->) je používána „čtyřtečka“ (::) a název vlastnosti musí být prefixovaný znakem dolaru (\$; podobně jako proměnné):

1. získání hodnoty statické vlastnosti

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. Trida::$navez_vlastnosti;
```

2. nastavení hodnoty statické vlastnosti

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. Trida::$navez_vlastnosti = "hodnota";
```

3. zavolání statické metody

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. Trida::nazevMetody($param1, $param2, ...);
```

Třídní vlastnosti a metody jsou povětšinou nazývány jako „statické“.

Nyní byste již neměli mít problém porozumět např. kódu (vypůjčeno z manuálu k Zend Frameworku):

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. $doc = new Zend_Search_Lucene_Document();
02. $doc
03.     ->addField(Zend_Search_Lucene_Field::Text('title',
04.                                                $title,
05.                                                'iso-8859-1'))
06.
07.     ->addField(Zend_Search_Lucene_Field::Unstored('contents',
08.                                                  $contents,
09.                                                  'utf-8'));
```

Objektový svět

Pokud chcete používat knihovny třetích stran, měli byste si ve většině případů vystačit s předchozí kapitolkou. Jaké konkrétní metody volat, aby požadovaný objekt dělal, co chcete, si můžete najít většinou v příkladech ke knihovně či API referenci. Teď ale půjdeme přímo k meritu věci – k tomu, jak si vytvořit nějaký smysluplný objekt.

Nechme zatím stranou, že pes a kočka jsou savci a podobné příklady, co se povětšinou uvádí, a zaměřme se na něco, s čím se můžete setkat prakticky ve všech aplikacích – subsystémem pro logování.

Abychom mohli vytvořit logovací objekt, musíme nejdříve udělat jeho šablonu – třídu. Třída se definuje konstrukcí:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. class NavezTridy
02. {
03.     // tělo třídy
04. }
```

Jako první se uvádí klíčové slovo `class`, za ním následuje název třídy (jaké jsou povolené znaky názvu třídy je uvedeno výše ve článku). Tělo třídy může být prázdné, nebo obsahovat definice jednotlivých zpráv, které objekty dané třídy mohou přijímat.

Takže teď si zadefinujeme třídu pro logování. Vy, co znáte OOP, se hned nenaštvejte a nepište do komentářů, co je to tu za malou hrudku zeleného hnusu, již jsem objevil v podpaždí jednoho jitra. Postupně bude následující třída podstupovat lifting, až se z ní stane docela úhledný kus kódu.

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. class Log
02. {
03.     var $soubor;
04.
05.     function loguj()
06.     {
07.         $args = func_get_args();
08.         $vystup = call_user_func_array("sprintf", $args);
09.
10.         file_put_contents(
11.             $this->soubor,
12.             $vystup . "\n",
13.             FILE_APPEND
14.         );
15.     }
16. }
```

Rozeberme jednotlivé konstrukce.

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. var $soubor;
```

Definuje celkem dvě zprávy – získání a nastavení vlastnosti `soubor` (`$objekt->soubor` a `$objekt->soubor = "navez_souboru";`).

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. function loguj()
02. {
03.     // ...
04. }
```

A toto definuje jednu zprávu – metodu `loguj` (`$objekt->loguj("zprava");`). Zavoláním `loguj` se vykoná kód mezi složenými závorkami. Funkce `sprintf()` volaná se stejnými

argumenty, s jakými byla zavolána metoda (viz `func_get_args()` a `call_user_func_array()`), vrátí formátovaný řetězec. Ten připojíme (`file_put_contents`) na konec souboru (`FILE_APPEND`), jehož název je uložený ve vlastnosti `soubor`. `$this` je speciální objekt, který odkazuje na aktuální instanci, které je zpráva posílána. Pokud tedy budeme mít objekt třídy `Log` uložený v proměnné `$objekt` a zavoláme metodu `loguj`, `$this` v metodě `loguj` ukazuje na stejný objekt jako `$objekt`.

Vytvoříme si `Log` :

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. $log = new Log;
```

Musíme nastavit, do jakého souboru se má logovat:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. $log->soubor = "error.log";
```

Následující kód připojí dva řádky na konec souboru `error.log` :

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. $log->loguj("Nastala naprosto neočekávatelná chyba.");
02. $log->loguj("bleh blah na řádku %d", __LINE__);
```

Nyní se pustíme na lifting.

Konstruktor

Mně osobně jako první věc vadí, že k tomu, aby se dalo začít logovat, musím vytvořit objekt a nastavit soubor a potom až je teprve možno něco dělat. V případě logování do souboru to ještě tak hrozné není, ale pokud by se muselo těch vlastností objektu nastavit více, bylo by to mnohem záladnější. Navíc by se muselo počítat s případy, kdy zavoláme metodu objektu, aniž by byla některá z vlastností inicializována. V případě, že by takové případy ošetřeny nebyly, objekt se bude mezi svým vytvořením a plnou inicializací nacházet v jakémsi nedefinovaném stavu a bůhví, co by se mohlo stát po zavolání některé z metod.

Instancování (aneb vytvoření instance třídy, aneb vytvoření objektu dle třídy) a inicializace se dají smrsknout do jednoho volání. K takovým účelům slouží konstruktor. Nechť tedy třída `Log` vypadá takto (namísto tří teček si domyslete tělo metody `loguj`):

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. class Log
02. {
03.     var $soubor;
04.
05.     function __construct($soubor)
06.     {
07.         $this->soubor = $soubor;
08.     }
09.
10.     function loguj() { ... }
11. }
```

Metoda `__construct` je taková zvláštní, speciální. Říká se jí konstruktor a je zavolána hned po vytvoření objektu operátorem `new`. Konstruktoru se právě předají ty parametry, které jsou v závorkách při vytváření pomocí `new` :

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. $log = new Log("error.log");
```

Nyní již rovnou můžete volat metodu `loguj`, vlastnost `soubor` byla nastavena v konstruktoru.

Všechny vlastnosti, bez kterých by objekt neměl smysl (tady např. `soubor`), by se měly inicializovat v konstruktoru.

Řízení přístupnosti

Další věcí, která by se nemusela líbit je, že vlastnost `soubor` může změnit kdokoli odkudkoli a stačí mu k tomu jenom reference na objekt. Člověk by neměl věřit cizímu kódu a už vůbec ne svému, a tak budeme chtít přístupy k `souboru` nějak ochránit. PHP zná celkem tři typy ochrany:

1. `public` – veřejně přístupné; aneb žádná ochrana, výchozí stav
2. `protected` – chráněné; k vlastnosti mají přístup vlastní instance a instance potomků třídy (něco o dědění bude dále ve článku)
3. `private` – jen a jen moje; k tomu se nedostane nikdo jiný než objekty dané třídy

Tato tři klíčová slova se při definici vlastnosti používají místo `var`, takže to může vypadat třeba takhle:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. private $soubor;
```

Nyní je `soubor` přístupný pouze v těle metody třídy `Log`.

Stejně jako s vlastnostmi a jejich ochranou je to i u metod. Akorát že tam klíčová slova značící přístup nenahrazují slovo `function`, nýbrž se umisťují před něj:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. public function __construct($soubor) { ... }
02. public function loguj() { ... }
```

Doporučuji nikdy nepoužívat `var` (je to relikv z PHP 4; při definování vlastností tedy používat pouze klíčová slova řízení přístupu) a u funkcí vždy uvádět, jakou ochranu mají (bůhví, co si dokážou vývojáři PHP usmyslet do dalších verzí, třeba nakonec v PHP 6 bude výchozí `private`).

Pokud nějak omezíte přístup k zasílání určitých druhů zpráv, omezuje se tím veřejné rozhraní, které třída poskytuje. Po tomto zásahu třída `Log` umí vlastně jen dvě věci – inicializovat svůj stav v konstruktoru a zalogovat zprávu předanou metodou `loguj`, nic víc, nic méně.

Skládání a dědění

Ted' ale co když někdo dostane šílený nápad, že by rád logoval do databáze? Existují dvě možnosti, jak to vyřešit:

1. skládání objektů
2. specializace (dědění)

Osobně se kloním k první variantě. Ovšem neochudím vás ani o tu druhou.

Skládání objektů spočívá v tom, že jeden objekt obsahuje referenci na jiný objekt (takže danému referencovanému objektu může zasílat zprávy). Pro `Log` by to znamenalo, že by držel referenci na nějaký „zapisovač“ a místo toho, aby zapisování do souboru a databáze atd. implementoval sám, tak „zapisovači“ podle daného protokolu zasílá požadavky na zápis a „zapisovač“ s nimi dělá vše potřebné. Přidání nového zapisovače obnáší vytvořit novou třídu, která podporuje daný protokol.

K jasné definici protokolu slouží tzv. „interface“, česky rozhraní. Jeho definice je podobná té třídní. Neuvedete klíčové slovo `class`, nýbrž `interface` a u metod neuvádíte tělo (pouze řízení přístupu, název a seznam parametrů). Také nelze pomocí interface definovat zprávy pro práci s vlastnostmi (získání, přiřazení hodnoty), ale pouze metody. Potřebujeme tedy „zapisovač“:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. interface Zapisovac
02. {
03.     public function zapis($zprava);
04. }
```

Třídy, které se zavazují k možnostem zasílat jim zprávy daného protokolu, rozhraní tzv. „implementují“:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. class SouborovyZapisovac implements Zapisovac
02. {
03.     private $soubor;
04.
05.     public function __construct($soubor)
06.     {
07.         $this->soubor = $soubor;
08.     }
09.
10.     public function zapis($zprava)
11.     {
12.         file_put_contents(
13.             $this->soubor,
14.             $zprava . "\n",
15.             FILE_APPEND
16.         );
17.     }
18. }
```

Třída nemusí implementovat žádné rozhraní, může být implementací jednoho, ale může jich být i více. Pak se jednotlivé názvy rozhraní oddělí čárkami (`class A implements B, C { ... }`).

Rovnou tu bylo využito, že název souboru, do kterého se má logovat, předáme v konstruktoru, a tak se nemusíme zabývat nastavováním vlastností.

Log potom bude vypadat takto:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```
01. class Log
02. {
03.     private $zapisovac;
04.
05.     public function __construct(Zapisovac $zapisovac)
06.     {
07.         $this->zapisovac = $zapisovac;
08.     }
09.
10.     public function loguj()
11.     {
12.         $args = func_get_args();
13.         return $this->zapisovac->zapis(call_user_func_array("sprintf", $args));
14.     }
15. }
```

Konstruktor již nepřijímá název souboru, do kterého zapisovat (Log se už vůbec o nějaké soubory nezajímá), místo toho dostává zapisovač. Uvedení „typu“ Zapisovac před názvem parametru je tzv. „type hinting“. PHP tím říkáme, že má ověřit, že se opravdu jedná o Zapisovac a ne třeba o řetězec, číslo nebo pole. Pokud znáte nějaký staticky

typovaný jazyk, nemyslete si, že byste type hintingem nahradili typovou analýzu v době kompilace, vše probíhá za běhu. Type hinting PHP pouze napoví („hint“ česky znamená naznačit, napovědět, náznak něčeho), po čem by se mělo koukat a co by mělo kontrolovat.

Když chceme zapisovat do souboru:

```
Ukázat zdrojový kód Zkopírovat do schránky Tisk
01. $error_log = new SouborovyZapisovac("error.log");
02. $log = new Log($error_log);
03. $log->loguj("42");
```

Pokud na logy kašlete, můžete si vytvořit zapisovač, který všechno zahodí:

```
Ukázat zdrojový kód Zkopírovat do schránky Tisk
01. class DevNullZapisovac implements Zapisovac
02. {
03.     public function zapis($zprava)
04.     {
05.         // nedělej nic
06.     }
07. }
```

Fantazii se meze nekladou, různé zapisovače můžete prohazovat podle prostředí, ve kterém zrovna aplikace běží (vývojové, produkční) atd. atp.

Ukažme si, jak řešit problém různých výstupů Log u specializací (děděním). Než to složitě vysvětlovat slovy, lepší je to ukázat na konkrétním kódu:

```
Ukázat zdrojový kód Zkopírovat do schránky Tisk
01. abstract class Log
02. {
03.     public function loguj()
04.     {
05.         $args = func_get_args();
06.         return $this->zapis(call_user_func_array("sprintf", $args));
07.     }
08.
09.     abstract protected function zapis($zprava);
10. }
11.
12. class SouborovyLog extends Log
13. {
14.     private $soubor;
15.
16.     public function __construct($soubor)
17.     {
18.         $this->soubor = $soubor;
19.     }
20.
21.     protected function zapis($zprava)
22.     {
23.         file_put_contents(
24.             $this->soubor,
25.             $zprava . "\n",
26.             FILE_APPEND
27.         );
28.     }
29. }
30.
31. final class DevNullLog
32. {
33.     protected function zapis($zprava)
34.     {
35.         // nedělej nic
36.     }
37. }
```


Nejdříve k novým syntaktickým prvkům. Přibyla nám nějaká klíčová slova – `extends`, `abstract` a `final`. `extends` se používá za názvem třídy a znamená, že daná třída rozšiřuje třídu za `extends`. Třída `SouborovyLog` je potomkem (dědí z) `Logu`. Rovněž tak třída `DevNullLog`. V PHP může mít každá třída maximálně jednoho rodiče.

`abstract` před `class` značí, že daná třída nemůže být instancována – nemůže být vytvořen objekt takové třídy. Před definicí metody zase, že zde je uvedena pouze deklarace (hlavička; podobně jako v interface /rozhraní/) a tělo (implementace) bude někde jinde (v potomkovi). Pokud třída obsahuje jednu abstraktní metodu, musí být deklarována jako abstraktní (tzn. že musí být `abstract` i před slůvkem `class`). Ovšem třída může být abstraktní, i když nemá ani jednu abstraktní metodu.

`final` je značka toho, že ze třídy už nemůže být dále děděno. Např. kód `class FooLog extends DevNullLog {}` vyvolá chybu.

A jak se to dá dohromady s dědičností? V základní třídě (base class) `Log` máme opět funkci pro formátování záznamu do logu, která posílá sama sobě zprávu pro zavolání metody `zapis`, ale tuto metodu sama neimplementuje (je zde pouze deklarace její hlavičky). Implementace `zapis` je přenechána potomkům `Log` – např. `SouborovyLog` a `DevNullLog`. Základní třída tedy v sobě kombinuje vlastnosti rozhraní a implementace.

Je to i hezký příklad toho, kdy ano a kdy nepoužívat `final` (alespoň doufám). Zatímco dále dědit od `DevNullLogu` je k ničemu (že se zpráva nikam nezapiše, nikam neuloží snad už ani jinak udělat nejde), u třídy `SouborovyLog` to smysl má – řekněme, že budete chtít přidávat čas, kdy se daná věc stala:

```

01. class SouboryLogSCasem extends SouborovyLog
02. {
03.     protected function zapis($zprava)
04.     {
05.         return parent::zapis(date("[Y-m-d H:i:s] ") . $zprava);
06.     }
07. }
```

Dostáváme se k další věci, se kterou se při dědičnosti setkáte – přepisování metod. Kromě toho, že potomek může do předka doplňovat metody, může také měnit chování stávajících. Pokud teď vytvoříte instanci `SouboryLogSCasem` a pošlete mu zprávu `loguj`, objekt sám na sobě zavolá metodu `zapis` s naformátovanou zprávou. Ale jelikož se jedná o instanci `SouboryLogSCasem`, bude vykonán kód, který je v těle metody `zapis` v této třídě, nikoli ta ze `SouborovyLog`. Metoda `zapis` v `SouboryLogSCasem` ale může volat metodu předka – i tu stejnou – slouží k tomu klíčové slovo `parent` následované „čtyřtečím“ a již samotným názvem metody předka.

U vlastností objektu se `parent::` neuvádí – vlastnost je prostě deklarace „tak si udělej v paměti místo“ a toto místo tam bude v základní třídě i všech potomcích.

Nyní byste měli mít základní povědomí o tom, co je skládání objektů a co dědičnost.

Ve většině textů se dočtete, že dědičnost je ta „nejvíc nejlepší“ vlastnost OOP. Podle mě je to vedlejší vlastnost, které se dostalo takové popularity, protože je prakticky ve všech mainstreamových jazycích. Hlavní je to, že si objekty mohou zprávy přeposílat (delegovat), díky čemuž můžeme implementovat různé služby hodně obecně (např. vstup a výstup) a postupným nabalováním dalších abstraktních vrstev se dostat k výslednému kódu, jenž by měl být lépe udržitelný (právě díky rozvrstvení).

Budu rád, když se v diskusi pod článkem podělíte o své názory na toto téma.

Samozřejmě že oba „přístupy“ můžete různě kombinovat a v praxi se tak často

děje. Např. pokud byste chtěli využít `Zapisovac` a konkrétně `SouborovyZapisovac`, opět při tom, když budete chtít přidávat čas, kdy k dané věci došlo, můžete podědit `SouborovyZapisovac` a tam implementovat přidávání časového razítka, nebo vytvořit `CasovanyZapisovac`, který deleguje zprávu jinému zapisovači.

Destruktor

Zatím jsme se seznámili s konstruktorem, který je volán při vytváření objektu, při jeho konstrukci. Destruktor, jak název napovídá, je volán při destrukci, při ničení, objektu. Stále zůstaneme u logů a konkrétně u zapisovače. Co když se nám nelíbí `file_get_contents()`, avšak jsme sžiti s `fopen()`, `fwrite()` atd.? Zdroje systému by se měly uvolňovat a abychom se o to nemuseli starat ručně, může to za nás udělat destruktorka:

```

01. class SouborovyZapisovac implements Zapisovac
02. {
03.     private $otevreny_soubor;
04.
05.     public function __construct($soubor)
06.     {
07.         $this->otevreny_soubor = fopen($soubor, "a");
08.         // tady by mělo být ošetření chyb
09.     }
10.
11.     public function zapis($zprava)
12.     {
13.         fwrite($this->otevreny_soubor, $zprava . "\n");
14.     }
15.
16.     public function __destruct()
17.     {
18.         fclose($this->otevreny_soubor);
19.     }
20. }
```

Až jakákoli instance této třídy nebude již nadále referencována (žádná proměnná na ni nebude odkazovat), garbage collector zavolá `__destruct`, který zavře otevřený soubor, a uvolní paměť obsazenou objektem.

Nicméně pozor, o zavolání destruktorky se stará garbage collector. PHP sice využívá reference counting, takže hned jak zmizí poslední reference na daný objekt, měl by být zavolán destruktorka. Ovšem změnilo-li by PHP někdy razantně své GC algoritmy (např. začalo využívat mark-sweep), mohly by se nám postupně hromadit objekty s otevřenými soubory a vyčerpali bychom tak systémové zdroje. Také podobná situace může nastat, ocitne-li se objekt v uzavřeném kruhu objektů (toto by měl řešit collector cyklických referencí v PHP verze 5.3).

static a self

Zase na chvíli skočíme k řešení logování do odlišných zařízení dědičností. Co když chceme, aby pro každý otevřený souborový log existovala v aplikaci jen jedna instance?

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```

01. class SouborovyLog extends Log
02. {
03.     static private $instance = array();
04.
05.     protected function __construct($soubor) { ... }
06.
07.     // ...
08.
09.     static public function instance($soubor)
10.     {
11.         $soubor = self::normalizovatCestu($soubor);
12.
13.         if (!isset(self::$instance[$soubor])) {
14.             self::$instance[$soubor] = new self($soubor);
15.         }
16.
17.         return self::$instance[$soubor];
18.     }
19.
20.     static protected function normalizovatCestu($cesta)
21.     {
22.         // tady by měl být kód pro normalizování cesty k souboru, jinak se pro
23.         // "../soubor" a "../../soubor", i když budou vlastně odkazují na stejný
24.         // soubor, vytvoří dvě instance
25.
26.         return $cesta;
27.     }
28. }

```

Za trojteččí si dosadíte kód z předchozích příkladů. Teď pokud chceme objekt pro `error.log`, voláme:

```
$error_log = SouborovyLog::instance("error.log");
```

Nejdříve jsme vytvořili statickou vlastnost `instance`. Pokud za názvem vlastnosti uvedete „rovná se něco“, přičemž „něco“ musí být odvoditelné při „kompilaci“ kódu (tedy ne např. volání funkce, ale může to být nějaké pole, číslo, řetězec, konstanta...), vlastnost čerstvě vytvořeného objektu bude „předvyplněna“ uvedenou hodnotou; toto platí i pro nestatické vlastnosti.

`protected` u konstruktoru zajistí, že při pokusu vytvořit objekt třídy `SouborovyLog` mimo tuto třídu PHP vyhodí chybu.

Nejzajímavější je asi statická metoda `instance`, která zjistí, jestli instance pro soubor existuje a neexistuje-li, vytvoří ji, a poté instanci vrátí. `self` je takové zájmeno – než abych si říkal pořád „Jakub“, řeknu „já“; než abych pořád psal `SouborovyLog`, napíšu `self`.

Problém se `self` je v tom, že u něj probíhá tzv. „early static binding“. Kdybych to opět převedl do lidského světa, tak pokud by se můj syn jmenoval třeba „Jan“ a podědil ode mne některé mé metody, které by neměnil, `self` by u něj stále znamenalo „Jakub“ a nikoli „Jan“ (v jeho metodách by ale `self` bylo to samé jako „Jan“). Tento „problém“ je řešen až v PHP 5.3, kde krom `self` můžete používat `static`, u něhož probíhá tzv. „late static binding“ – název třídy je vyřešen až za běhu (v „run-time“). Nejlepší asi bude příklad:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```

01. class SouborovyLog extends Log
02. {
03.     // ...
04.
05.     static public function instance($soubor)
06.     {
07.         $soubor = static::normalizovatCestu($soubor);
08.
09.         // ...
10.     }
11.
12.     static protected function normalizovatCestu($cesta)
13.     {
14.         // ...
15.     }
16. }
17.
18. class SilenySouborovyLog extends SouborovyLog
19. {
20.     static protected function normalizovatCestu($cesta)
21.     {
22.         $cesta = strrev($cesta);
23.
24.         return $cesta;
25.     }
26. }
```

V PHP 5.3, pokud zavoláte `SilenySouborovyLog::instance("robuos");`, získáte log, který bude zapisovat do souboru `soubor` v momentálním pracovním adresáři.

Vyšší dívčí

Tato kapitola se bude zabývat různorodými „pokročilejšími“ tématy, na která můžete při objektově orientovaném programování narazit.

Gettery a settery

Gettery a settery (anglicky „getters and setters“) jsou princip, který si programátoři zavedli kvůli nedotaženému návrhu objektového modelu jazyků jako PHP. Když si vezmete rozhraní, jediný typ zprávy, který můžete definovat, je volání metod. Jenže krom toho byste třeba chtěli objektu zaslat zprávu na získání/nastavení hodnoty některé z jeho vlastností. Gettery a settery tedy převádí tyto zprávy na volání metod.

Stále zůstáváme u problému logu a `Zapisovace`. Co když chceme z nějakého důvodu zaměnit zapisovač za jiný? Můžeme sice vytvořit nový log, ale to neovlivní současný objekt (na který mohou odkazovat jiné objekty, a tak jejich logovací záznamy budou stále zapisovány na stejné místo). Zpřístupnit vlastnost `zapisovac` jako `public` také není to nejlepší, protože PHP je dynamicky typované, a tak by se nám mohl ve vlastnosti `zapisovac` ocitnout místo `Zapisovace` třeba řetězec. Jediný způsob, při kterém PHP implicitně ověřuje, jestli se jedná opravdu o objekt daného typu, je „type hinting“ u metod. Proto ho využijeme pro `zapisovac`:

```

01. class Log
02. {
03.     private $zapisovac;
04.
05.     public function __construct(Zapisovac $zapisovac) { ... }
06.
07.     public function getZapisovac()
08.     {
09.         return $this->zapisovac;
10.     }
11.
12.     public function setZapisovac(Zapisovac $zapisovac)
13.     {
14.         $this->zapisovac = $zapisovac;
15.     }
16.
17.     // ...
18. }

```

setZapisovac je metoda, která nám umožňuje nastavit zapisovac. Je to tzv. „setter“, protože „set“ znamená v angličtině nastavit. Většinou se settery prefixují právě tím set. Pak je tu ještě getZapisovac, což naopak zapisovac vrátí. Gettery se většinou prefixují get.

Settery umožňují další kontroly, např. je-li číslo v nějakém intervalu. Není doporučené mapovat vlastnosti a gettery/settery 1:1, protože tím říkáte vnějšímu světu přesně, jaká je implementace vašeho objektu. Někde se můžete dočíst, že gettery a settery jsou zlo s velkým Z. Právě kvůli tomu, že většina lidí si řekne, že vlastnosti skryjí pomocí protected nebo private a pro každou udělají gettery a settery. Volte, co má jít do veřejného rozhraní třídy, rozvážně.

Fluent interfaces

Fluent interfaces využívají toho, že můžeme metody „řetěžit“ (\$objekt->metoda1()->metoda2()->...()); viz Základy syntaxe) a vracet v metodách referenci na současný objekt (return \$this;). Většinou se tohoto využívá u setterů. Řekněme, že máme nějaký objekt reprezentující osobu:

```

01. class Osoba
02. {
03.     private $jmeno;
04.     private $prijmeni;
05.     // ... další vlastnosti
06.
07.     // ... gettery
08.
09.     public function setJmeno($jmeno)
10.     {
11.         $jmeno = trim($jmeno);
12.         if (strlen($jmeno) < 2) {
13.             trigger_error("Nejake kratke jmeno, ne?", E_USER_ERROR);
14.         }
15.
16.         $this->jmeno = $jmeno;
17.
18.         return $this;
19.     }
20.     // ... další settery, všechny nakonec vrací $this
21. }
22.
23. $ja = new Osoba;
24. $ja
25.     ->setJmeno("Jakub")
26.     ->setPrijmeni("Kulhan");

```

O řetězení metod bylo napsáno již v kapitole o syntaxi. Když tedy vrátíme `$this`, což je reference na ten samý objekt, je další metoda v řetězci zavolána opět na tom objektu.

Další využití fluent interfaces v PHP najdete hlavně u různých „sestavovačů“ SQL dotazů. Ale i pro log není k zahoezení:

```
01. class Log
02. {
03.     public function loguj()
04.     {
05.         // ...
06.
07.         return $this;
08.     }
09. }
10.
11. // ...
12.
13. $error_log
14.     ->loguj("Je to v haji.");
15.     ->loguj("Uz se na to vyprdu a pujdu spat.");
```

Vlastní zpracovávání zpráv

PHP je jazyk dynamický, a tak je dobré využívat všech vlastností, které to přináší. Jednou z nich je, že kromě předdefinovaných/předdeklarovaných zpráv v těle třídy můžeme některé obsloužit i vlastním kódem a rozhodovat tak podle momentálního stavu objektu, třídy, či globálního stavu (i když závislosti na globálním stavu by se při programování měl člověk co nejvíce vyvarovat, protože pak nastávají problémy při konkurenčním programování).

Jednou ze základních zpráv je získání hodnoty vlastnosti objektu. Pokud není nějaká vlastnost ve třídě deklarována nebo se jedná o vlastnost, která sice deklarována je, avšak nemáme k ní přístup (kupř. když se mimo kód třídy snažíme dostat k vlastnosti označené jako `private`), PHP se podívá, jestli objekt má metodu `__get`, a má-li ji, je jí předán prvním parametrem název vlastnosti, na kterou se ptáme. „Hloupý“ příklad:

```
01. class Foo
02. {
03.     private $data = array();
04.
05.     public function __construct(array $data)
06.     {
07.         $this->data = $data;
08.     }
09.
10.     public function __get($vlastnost)
11.     {
12.         return $this->data[$vlastnost];
13.     }
14. }
15.
16. $foo = new Foo(array("bar" => "baz"));
17. echo $foo->bar; // vypíše baz
```

Podobně je to i s nastavováním vlastnosti – tady zase PHP hledá metodu `__set` a předá jí název vlastnosti a nastavovanou hodnotu.

Když toto zkombinujeme, můžeme udělat jednu zajímavou věc:

Ukázat zdrojový kód Zkopírovat do schránky Tisk

```

01. class Foo
02. {
03.     public function __get($vlastnost)
04.     {
05.         $getter = "get" . str_replace("_", "", $vlastnost);
06.         if (method_exists($this, $getter)) {
07.             return $this->$getter();
08.         }
09.
10.         // tady by mělo být ošetření pro případ, že getter neexistuje
11.     }
12.
13.     public function __set($vlastnost, $hodnota)
14.     {
15.         $setter = "set" . str_replace("_", "", $vlastnost);
16.         if (method_exists($this, $setter)) {
17.             return $this->$setter($hodnota);
18.         }
19.
20.         // tady by mělo být ošetření pro případ, že setter neexistuje
21.     }
22. }
23.
24. class Bar extends Foo
25. {
26.     private $baz = array();
27.
28.     public getBaz()
29.     {
30.         return $this->baz;
31.     }
32.
33.     public setBaz(array $baz)
34.     {
35.         $this->baz = $baz;
36.     }
37. }
38.
39. $bar = new Bar;
40. $bar->baz = array(1, 2, 3);
41. $bar->baz = "foo!";

```

Foo umí převádět zprávy získávání a nastavování vlastností na volání příslušných getterů a setterů (pozn.: předpokládá se, že pro metody je používána velbloudí notace /např. `viceSlovnnyNazevMetody/` a pro přístup k vlastnostem podtržítková /např. `vice_slovnny_nazev_vlastnosti/`; metody jsou v PHP case-insensitive).

Získání a nastavení vlastností byly uvedeny jako základní zprávy. Ale ještě by se jako další zprávy dalo pokládat volání `isset()` nad vlastností objektu (`isset($objekt->vlastnost)`) a `unset()` také nad vlastností (`unset($objekt->vlastnost);`), protože pokud daná vlastnost neexistuje, PHP se snaží tato volání převést na metody `__isset`, resp. `__unset` – obě dostanou jeden parametr, a to název vlastnosti. Obě fungují od PHP verze 5.1.0.

Další magickou metodou je `__call`, po které PHP kouká, když nemůže najít volanou metodu, popř. když k ní nemáme z aktuálního kontextu přístup. `__call` jsou předány dva parametry – název volané metody a pole parametrů, s jakými byla volána. Stejně je to s `__callStatic`, ovšem jak se dá vytušit z názvu, tak ta je volána, pokud nelze najít nějakou statickou (třídní) metodu. (`__callStatic` pracuje tímto způsobem až od PHP 5.3.)

Seznam těchto metod můžete najít [na php.net](http://php.net).

Funkční objekty

Funkční objekty umožňují, abychom mohli s objektem nakládat jako s funkcí,

tj. abychom ho mohli zavolat jako každou jinou funkci přidáním závorek se seznamem parametrů za název objektu (proměnné objektu):

```

01. class Nahrazovac
02. {
03.     private $co;
04.     private $cim;
05.
06.     public function __construct($co, $cim)
07.     {
08.         $this->co = $co;
09.         $this->cim = $cim;
10.     }
11.
12.     public function __invoke($kde)
13.     {
14.         return str_replace($this->co, $this->cim, $kde);
15.     }
16. }
17.
18. $nahrazovac = new Nahrazovac("a", "b");
19.
20. echo $nahrazovac("aaa"); // vypíše bbb
    
```

Magická metoda `__invoke` je vyvolána tehdy, zavolá-li se objekt jako funkce.

Využití je v případě, kdy chceme společně s `callbackem` předat nějaký stav, který by se jinak musel předávat odděleně. Příklad výše je trochu hloupý – *Vždyť můžu rovnou zavolat `str_replace("a", "b", "aaa");`!* Ale třeba u `array_map()` tam ty další parametry už jinak nepředáte:

```

01. $bcka = array_map($nahrazovac, array("alfa", "beta", "gamma", "delta"));
02. var_dump($bcka);
    
```

Vypíše:

```

01. array(4) {
02.     [0]=>
03.         string(4) "blfb"
04.     [1]=>
05.         string(4) "betb"
06.     [2]=>
07.         string(5) "gbmbb"
08.     [3]=>
09.         string(5) "deltb"
10. }
    
```

Takovéto zpracování `__invoke` je přístupné od PHP 5.3.

Konstanty

Už v archaických verzích PHP umožňovalo definovat konstanty pomocí `define()`. Problém je, že takto definované konstanty jdou do globálního prostoru. V PHP 5 lze definovat konstanty ve třídě pomocí klíčového slova `const`:

```

01. class Moje
02. {
03.     const MAM_RAD = "PHP";
04.     const NEMAM_RAD = "PHP";
05. }
    
```

Nikdy nezadrátovávejte různé roztodivné hodnoty do kódu (až na pár výjimek),

používejte konstanty. Věřte, že po roce si nevzpomenete, co jste tou pětkou tady u té funkce mysleli.

I kdybyste neprogramovali objektově, můžete třídu využít jako jmenný prostor pro konstanty. PHP 5.3 již jmenné prostory má, a tak nemusíte za tímto účelem zneužívat třídy.

Výjimky

Jednoduše se dá říci, že výjimky jsou dalším způsobem, jak v PHP ošetřovat chyby. Kód vyhodí výjimku a jiný ji zachytí a zpracuje. Vyhozením se ihned přeruší právě probíhající kód a výjimka postupně probublává nahoru, dokud někde nenarazí na blok, který ji zpracovává.

Vraťme se k Zapisovací, který využíval `fopen()`. S výjimkami můžeme ošetřit právě chyby, které nastanou v konstruktoru objektu (cokoli, co konstruktor vrátí, je zahazeno, tudíž výjimky jsou jediným způsobem, jak dát vědět o chybách v konstruktoru):

```

01. class IOException extends Exception {}
02.
03. class SouborovyZapisovac implements Zapisovac
04. {
05.     private $otevreny_soubor;
06.
07.     public function __construct($soubor)
08.     {
09.         $this->otevreny_soubor = fopen($soubor, "a");
10.         if (!$this->otevreny_soubor) {
11.             $vyjimka = new IOException("Nelze otevrit soubor.");
12.             throw $vyjimka;
13.             // zkráceně: throw new IOException("Nelze otevrit soubor.");
14.         }
15.     }
16.
17.     // ...
18. }
19.
20. try {
21.     $zapisovac = new SouborovyZapisovac("error.log");
22.     $log = new Log($zapisovac);
23.
24. } catch (IOException $e) {
25.     die("Logovani je nam treba.");
26.
27. } catch (Exception $e) {
28.     die("Tak toto jsem vazne necekal.");
29. }
```

Vytvořili jsme třídu `IOException`, která dědí z `Exception` (všechny vyhazované výjimky musí dědit z `Exception`). Pokud se soubor nepodaří otevřít, je vyhozena `IOException`.

Poté při vytváření instance kontrolujeme (`try`), jestli právě tato výjimka nebyla vyhozena a bylo-li tomu tak, chytíme ji (`catch`). Vidíte, že pro jedno `try` může být několik bloků `catch` – každý blok chytá určitý typ výjimky.

Nemusíte zachytávat všechny výjimky. Pokud žádný `catch` blok výjimku nezachytí, PHP hledá první obalující `try` blok (třebas ve volající metodě) a testuje k němu přidružené `catch` bloky. Nejhorší vše dopadne tak, že výjimka probublá až úplně na povrch a PHP vyhodí chybu.

Není nutné hned při zachycení výjimky umírat (`die()`) – záleží, co se stalo. Pokud chceme uložit něco do keše a nepodaří se to, svět se povětšinou nezboří, ale pokud je aplikace závislá na připojení k databázi a spojení neustanovíme, pak je zle. Ale uživateli

asi nebude moc platno, že uvidí hlášku „Nepodařilo se připojit k databázi.“ vyvedenou černým písmem na jinak úplně bílém pozadí (navíc je potřeba myslet na návratový HTTP kód).

Výjimky by se měly zachytávat tam, kde je můžete ošetřit. Sice je možno znovu je vyhodit (`try { ... } catch (Exception $e) { throw $e; }`), ale dělat toto všude je zbytečnost – ony probublají samy.

Oprávněnou příčinou znovuvyhození je, když se snažíme výjimečnou situaci opravit, ale nezadaří se. Pak se může hodit, že od PHP 5.3 mohou výjimky vytvořit řetězec pomocí vlastnosti `previous` základní třídy `Exception`, který se nastavuje třetím parametrem konstruktoru:

```

01. try {
02.     ...
03. } catch (Exception $e) {
04.     if (!spravTo()) {
05.         throw new Exception("Ouplne rozbity!", NULL, $e);
06.     }
07. }
```

Závěr

Tento článek by vám měl osvětlit základy práce s objekty v PHP. Spíše než sáhodlouhé rozbory, co ano a co ne, na něž nejsou odpovědi, jež by se daly označit za jednoznačnou pravdu, ukazuje postupy (chtěl jsem napsat vzory, ale to by se mohlo plést s návrhovými vzory), kterými se při práci s objekty můžete řídit. OOP je podle mého rozhodně oproti procedurálnímu programování krokem kupředu a neměli byste si nechat ujet vlak. Pokud vám vše nepůjde hned tak, jak chcete, nevzdávejte to. Dobře modelovat objekty se naučíte postupně pročítáním dalších materiálů, odkoukáváním odjinud a samozřejmě také praxí.

Tento článek je součástí kurzu či seriálu, ke kterému máte možnost odevzdávat úkoly. Máte-li zájem se zapojit, stačí se [přihlásit](#). Úkoly opravují lektoři, kteří v případě potřeby poradí, jak lze řešení lépe zpracovat.

Účast ve všech kurzech je **bezplatná**.

Jakub Kulhan

Autor momentálně studuje na osmiletém gymnáziu v Kralupech nad Vltavou. Programování se věnuje od 11 let, kdy ho poprvé uchvátila možnost "mít vlastní stránky". Nakrátko poté objevil PHP a už se to s ním "vezlo". Webové aplikace zůstaly jeho hlavní doménou, ale ve svém volném čase probádává nejrůznější zákoutí světa programování, programovacích jazyků a všeho kolem nich.

Korektura: Martin Šimeček (DeedX)

[Tisk](#)
[Doporučit](#)
[RSS](#)
[PDF](#)
[UVEŘAĎMĚ SK](#)
[DEL](#)
[1010 US](#)
[LINKUJ. CZ](#)
[JAGGHI TO](#)

« Soutěž o nejlepší hosting zná vítěze

Object Pascal - 07: Cyklus for »

Přečtete si také:

- [JSON: první kroky aneb používáme PHP a jQuery](#)
- [PHP a MySQL – Mysql](#)
- [Generujeme náhledy s PHP objekty](#)

Povinné ručení

Spočítejte si nejlevnější **povinné ručení**, sazby pro rok 2009.

Kvalitní www stránky

intencio - **tvorba www stránek**, eshopů, řešení reklamy na stránky

HOME

+

DISKUSE

+

KOMENTÁŘE

+

REKLA

Společenské hry

Nenuďte se a společně hrajte **společenské hry** z VeselyKlaun.cz

Hip hop shop

Hip hop shop a Skateshop obchod pro moderní lidi. Vše skladem.

Diskuze k článku (13)

Zobrazit všechny

Přidat nový

Sledovat diskuzi

Super clanek	Tom V	09. 12. 2009 20:56
Ale...	Tomáš	10. 12. 2009 12:21
Hezký článek	vrana	10. 12. 2009 16:03
Několik poznámek...	LamiCZ	11. 12. 2009 03:34
Php a oop	remmidemmi	27. 12. 2009 10:40
Skvělý článek	Jan Košář	27. 12. 2009 20:07
Good job!	Yety	19. 01. 2010 13:31

Zobrazit všechny

Přidat nový

Tagováno:

php oop

ZÁLOŽKY

☆

© 2004-2010 Programujte by Lukáš Churý, ISSN 1801-1586

Tento server dodržuje právní předpisy o ochraně osobních údajů. Všechna práva vyhrazena. Bez svolení redakce není možno texty dále rozšiřovat!
 Kontakt | Reklama | Redakce | Podmínky užívání obsahu | Podpořte Programujte.com | Ke stažení | O portálu | RSS exporty [80.78.156.4]

HOME

 RSS DISKUSE

 KOMENTÁŘE


ZÁLOŽKY


MENU
