



Distance Learning System

# Advanced PHP Programming

## Object Basics

Darko Pantović

# Plan

---

- *Klase i objekti*: Deklarisanje klasa i instanciranje objekata
- *Konstruktor metode*: Automatizovanje procesa setup-a objekta
- *Tipovi*: Zašto su tipovi važni?
- *Nasleđivanje*: Zašto je nasleđivanje potrebno i kako ga koristiti?
- *Dostupnost*: Zaštita metoda i svojstava.

# Šta podrazumeva pojam OOP?

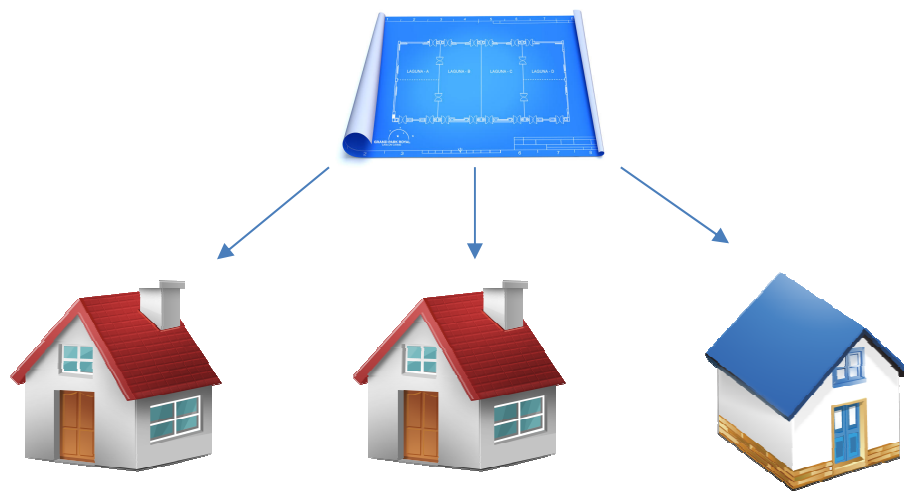
---

- Objektno-orijentisano programiranje je stil programiranja koji omogućava programeru da slične programske celine grupiše u klase.
- Ovakav pristup podrazuiva i pridržavanje DRY principa („Don't repeat yourself“).
- Osnovna prednost pridržavanja DRY principa je u tome što se kod lakše održava i promena na jednom mestu ne zahteva ažuriranje čitavog koda.
- Prednosti OOP-a dolaze do izražaja onda kada se pojavi veći broj instanci iste klase.

# Klasa - Objekat

---

- Šta je klasa, a šta objekat?
- Klasu možemo posmatrati kao šablon koda koji se upotrebljava za generisanje novih objekata.



# OOP vs Procedural (parallel)

---

1. **Re-Usability** – Upotreba OOP modela nudi sjajne mogućnosti ponovne iskoristljivosti jednom napisanog koda. Na primer, ako napravimo klasu koja se bavi izračunavanjem složenih izraza možemo je koristiti na čitavoj aplikaciji.
2. **Easy to Maintain** – OOP podrazumeva grupisanje koda u logički povezane celine čime se značajno olakšava održavanje koda. Ukoliko bismo na prethodno pomenutoj klasi želeli da modifikujemo način izračunavanja ta promena bi bila primenjena na svim delovima aplikacije.
3. **Good Level of Abstraction** – OOP podrazumeva viši nivo apstrakcije u odnosu na proceduralni pristup pisanja koda čime se ostvaruju značajni benefiti ali se gubi na preformansama.
4. **Molecularity** – Kreiranje zasebne klase za rešavanje pojedinačnih problema (grupe problema) čini aplikaciju modularnom. Na ovaj način se aktivnosti razvoja poslovne logike (business logic) mogu podeliti na timove koji aplikaciju razvijaju gotovo nezavisno.

# Klasa

---

- Deklarisanje nove klase vrši upotrebom ključne reči *class*.
- Nakon ključne reči dolazi proizvoljni naziv za klasu koji podleže pravilima pisanja identifikatora promenljivih.
- Nazivi klasa uglavnom imaju veliko početno slovo iako ovo nije uslovljavajuće sintaksno pravilo.
- Kod pridružen klasi mora biti smešten unutar vitičastih zagrada.
- Kreirajmo prvu klasu poštujući navednena pravila...

# Klasa (2)

---

```
class Product {  
    // prostor za telo klase  
}
```

- Klasa koju smo upravo kreirali poseduje urednu sintaksu iako sama po sebi nije naročito korisna.
- Kreiranjem klase definisan je novi korisnički definisan tip podatka, odnosno definisan je novi tip za čuvanje podataka koji se može upotrebljavati u skriptu.
- Već sada je moguće izvršiti instanciranje ovakve klase.

# Objekat

---

- Objekat je instanca klase.
- Kreiranje objekta kao instance određene klase naziva se **instanciranje**.
- Instanciranje se vrši operatorom *new* kojem sledi naziv klase.
- Klasa koju smo kreirali (Product) može biti korišćena za kreiranje većeg broja objekata:

```
class Product {  
    // telo klase  
}  
$p1 = new Product;  
$p2 = new Product;  
var_dump($p1);  
var_dump($p2);
```

```
object (Product)[1]  
object (Product)[2]
```



# Podešavanje svojstava u klasi

---

- Klase mogu definisati specijalne promenljive koje se nazivaju svojstva (properties).
- Svojstva možemo posmatrati kao promenljive članova jer njihove vrednosti mogu biti različite za svaki objekat.
- Prilikom definisanja svojstva mora se definisati i opseg vidljivosti (dostupnosti).
- Definisanje opsega vidljivosti se vrši ključnim rečima (*public*, *protected* i *private*) – podsetimo se pojmova *local scope* i *global scope*.
- Dodajmo zato polje za naziv i cenu proizvoda u našu klasu:

# Podešavanje svojstava u klasi (2)

---

```
class Product {  
    public $title = "unknown" ;  
    public $price = 0 ;  
}
```

- Kako smo svojstva definisali u klasi svaki obekat koji bude kreiran kao instanca ove klase biće ispunjen podrazumevanim vrednostima za definisana svojstva.
- Ključna reč *public* obezbeđuje otvoren pristup svojstvu.
- Ključne reči *public*, *private* i *protected* su uvedene u PHP sa verzijom 5 i u verziji 4 nisu podržane.
- PHP 4 poznaje ključnu reč *var* koja odgovara *public*.
- PHP 5 poznaje sve navedene reči (uključujući i *var*).

# Pristupanje svojstvima za objekat

---

- Nad svakim objektom se može izvršiti pristup vrednosti za svojstvo.
- Da bismo izvršili ovakav pristup koristimo oznaku '->'.
- Pogledajmo ovo na primeru:

```
class Product {  
    public $title = "unknown";  
    public $price = 0;  
}  
$p1 = new Product;  
$p2 = new Product;  
var_dump($p1->title);  
var_dump($p2);
```

string 'unknown' (length=7)

```
object(Product)[2]  
  public 'title' => string 'unknown' (length=7)  
  public 'price' => int 0
```

# Editovanje vrednosti svojstva

---

- Vrednosti svojstava mogu biti promenjene nakon instanciranja objekta.
- Primer:

```
class Product {  
    public $title = "unknown";  
    public $price = 0;  
}  
$p1 = new Product;  
$p1->title = "Samsung Galaxy S6";  
$p2 = new Product;  
var_dump($p1->title);  
var_dump($p2);
```

string 'Samsung Galaxy S6' (length=17)

```
object(Product)[2]  
  public 'title' => string 'unknown' (length=7)  
  public 'price' => int 0
```

# Dodavanje novih svojstava (!)

- Sintaksa PHP-a ne zahteva definisanje svih svojstava objekta u klasi već se to može raditi i kasnije na pojedinačnoj instanci.
- Primer:

```
class Product {  
    public $title = "unknown";  
    public $price = 0;  
}  
$p1 = new Product;  
$p1->title = "Samsung Galaxy S6";  
$p1->price = 580;  
$p1->os = "Android OS, v5.0.2 (Lollipop)";  
var_dump($p1);
```

```
object(Product)[1] public 'title' => string 'Samsung  
Galaxy S6' (length=17) public 'price' => int 580 public  
'os' => string 'Android OS, v5.0.2 (Lollipop)' (length=29)
```

# Rad sa metodama

---

- Pokazali smo kako koristiti svojstva da bi se čuvali podaci u objektu, a sada treba razmotiti na koji način vršiti akcije kroz objekat.
- Dok svojstva čuvaju vrednosti, metode izvršavaju zadatke.
- Za razliku od svojstava koja se mogu dodavati svakoj instanci pojedinačno, to sa metodama nije moguće (ili bar ne tako direktno).
- I za metodu kao i za svojstvo je potrebno definisati opseg vidljivosti.
- Ukoliko se metodi ne postavi neka od poznatih ključnih reči za definisanje vidljivosti podrazumevano se smatra da je metoda *public*.

# Kreiranje metode

---

- Primer:

```
class Product {  
    public $title = "unknown";  
    public $price = 0;  
    public function getInfo(){  
        return "This is product info.";  
    }  
}  
  
$p1 = new Product;  
echo $p1->getInfo();
```

# Prekid klase drugim fajlom

---

- Prekidanje klase drugim fajlom (include, require, HTML...) nije dozvoljeno. *Primer...*

```
class myClass{  
    public $abc;  
    public function test(){  
        return $this->abc;  
    }  
    include "abc.php";  
}
```

- Prekidanje tela metode (funkcije) drugim fajlom je moguće i neće izazvati grešku. *Primer...*

```
class myClass{  
    public $abc;  
    public function test(){  
        include "abc.php";  
    }  
}
```



# Konstruktor

---

- Klasi može biti definisana konstruktorska metoda koja će biti pozvana onog trenutka kada se izvrši instanciranje objekta.
- Konstruktor bi trebalo da se bavi svim operacijama koje su od značaja za konstruisanje objekta.
- Najvažnija uloga konstruktora je definisanje inicijalnih vrednosti za svojstva.
- Upotrebom konstruktora instanciranje postaje **jednostavnije i bezbednije**.
- Konstruktorska metoda izgleda ovako: `__construct`
- PHP 4 ne poznaje ovu metodu već konstruktorskom metodom smatra onu koja nosi ime koje odgovara klasi.

# Primer

---

```
class Product {
    public $title;
    public $price;

    public function __construct($title, $price){
        $this->title = $title;
        $this->price = $price;
    }
    public function getInfo(){
        return "Title: {$this->title}, price: {$this->price}.";
    }
}

$p1 = new Product("Alcetel Idol 3", 320);
echo $p1->getInfo();
```

Title: Alcetel Idol 3, price: 320.

**LINKgroup**

# Kontrola tipa

---

- PHP je slabo tipiziran jezik i kao takav dozvoljava dinamičku tipizaciju.
- Ipak, PHP prepoznaje tipove i može da ih kontroliše.
- Funkcije za testiranje tipa:
  - `is_bool()`
  - `is_integer()`
  - `is_double()`
  - `is_string()`
  - `is_object()`
  - `is_array()`
  - `is_resource()`
  - `is_null()`
- Navedene funkcije kao odgovor vraćaju boolean vrednosti true ili false.

# Kontrola tipa argumenta

---

- Ukoliko bismo želeli da definišemo novu metodu koja menja status objekata na onu boolean vrednost koja je prosleđena kao argument moramo proveriti da li je boolean tip zaista i prosleđen.
- Primer:

```
public function setStatus($s){  
    if(!is_bool($s)){  
        die("Argument must be a Boolean.");  
    }  
    $this->status = $s;  
}
```

# Prvera tipa objekta

---

- Ukoliko bismo želeli da ograničimo parametar samo na objekat određene klase mogli bismo malo modifikovati prethodni primer:

```
public function setStatus($s){  
    if(!$s instanceof Product){  
        die("Argument must be a Boolean.");  
    }  
    $this->status = $s;  
}
```

# Nagoveštavanje tipa objekata

---

- Iako dati problem možemo rešiti na prethodno prikazan način, PHP nam nudi sintaksu po kojoj vršimo nagoveštaj za tip objekta već prilikom definisanja parametra:

```
public function setStatus(Product $s){  
    $this->status = $s;  
}
```

- Na osnovu date sintakse se postavlja pitanje da li možemo obezbediti neprosleđivanje parametra definisanog prema klasi?
  - To je moguće uraditi dodavanjem podrazumevane vrednosti za parameter, NULL:

```
public function setStatus(Product $s=NULL){  
    $this->status = $s;  
}
```

# Nasleđivanje - Inheritance

---

- Jedna ili više klasa prilikom svog nastanka mogu preuzeti osobine neke druge već postojeće klase. Ovakav proces naziva se nasleđivanje.
- Klasa koja nasleđuje osobine naziva se *subclass*-a, dok se klasa koja je nasleđena naziva *superclass*-a.
- Ovakve veze se često izražavaju relacijama roditelj-dete (parent-child class) i predstavljaju stablom nasleđivanja.
- Nasleđivanje se vrši upotrebom ključne reči *extends*.

# Primer

---

```
class Product {
    public $title;
    public $price;
    public function __construct($title, $price){
        $this->title = $title;
        $this->price = $price;
    }
    public function getInfo(){
        return "Title: {$this->title}, price: {$this->price}.";
    }
}

class Dvd extends Product { }

$d1 = new Dvd("The Godfather: Part II", 8.5);
echo $d1->getInfo();
```



# Konstruktori i nasleđivanje *(magic methods)*

---

- Prilikom kreiranja novog objekta kao instance klase koja je imala nasleđivanje javlja se problem sa konstruktorima jer i roditeljska klasa i klasa potomka imaju svoje konstruktore.
- Koji će onda konstruktor biti upotrebljen i kako?
- Ukoliko klasa potomak nema konstruktor biće korišćen samo roditeljski konstruktor. Iz ovog razloga smo u prethodnom primeru morali proslediti oba argumenta konstruktoru.
- Sa druge strane, ukoliko i subclass-a ima konstruktor onda je taj konstruktor dužan da uredno pozove roditeljski i da mu prosledi sve potrebne argumente.
- Nakon ovoga, konstruktor može nastaviti sa konstruisanjem specifičnih svojstava objekta.

# Konstruktori i nasleđivanje (2)

---

- Kako bi bilo izvršeno pozivanje konstruktora roditeljske klase mora se najpre rešiti način pristupa roditeljskoj klasi.
- U ovu svrhu se koristi ključna reč *parent* u kombinaciji sa znakom ::
- Nakon pristupa roditeljskoj klasi možemo definisati metodu, što je u slučaju našeg konstruktora:

```
class Dvd extends Product {  
    public $language;  
    public function __construct($title, $price, $language){  
        parent::__construct($title, $price);  
        $this->language = $language;  
    }  
}
```

# \_\_construct i PHP distribucije

---

- Ukoliko želite da napišete konstruktor kompatibilan sa verzijom PHP 4 potrebno je konstruktor nazovete istim imenom kao i klasu.

```
class MyClass{  
    function MyClass(){  
    }  
}
```

**Napomena:** Ovakva sintaksa smatra se zastarelom za PHP 7 i u narednim verzijama će biti isključena iz sintakse.

# Prepisivanje metoda

---

- Prilikom nasleđivanja sve istoimene metode (kao što smo to videli sa konstruktorom) i svojstva bivaju prepisane novim.
- Iz ovog razloga klasi koja nasleđuje mogu se jednostavno postaviti nove vrednosti za identifikatore metoda i svojstva kako bi one prepisale postojeće metode i svojstva iz roditeljske klase koja ne odgovaraju novoj klasi.
- Pogledajmo to na našem primeru...

# Primer

---

```
class Product {
    public $title;
    public $price;
    public function __construct($title, $price){
        $this->title = $title;
        $this->price = $price;
    }
    public function getInfo(){
        return "Title: {$this->title}, price: {$this->price}";
    }
}
class Dvd extends Product {
    public $language;
    public function __construct($title, $price, $language){
        parent::__construct($title, $price);
        $this->language = $language;
    }
    public function getInfo(){
        return "Title: {$this->title}, price: {$this->price}, language: {$this->language}";
    }
}
$d1 = new Dvd("The Godfather: Part II", 8.5, " English | Italian | Spanish | Latin | Sicilian");
echo $d1->getInfo();
```

# Specijalizacija metoda

---

- Kao što se može uočiti u prethodnom primeru, javlja se redundantnost koda kojim se definišu metode.
- Zato bimo mogli iskoristiti roditeljku metodu i izvršiti specijalizaciju.
- Pogledajmo to na primeru:

```
class Dvd extends Product {  
    public $language;  
    public function __construct($title, $price, $language){  
        parent::__construct($title, $price);  
        $this->language = $language;  
    }  
    public function getInfo(){  
        return parent::getInfo().", language: {" . $this->language . " }";  
    }  
}
```

# Magične konstante *(magic constants)*

- PHP poseduje niz konstanti. Ovde ćemo pokazati nekoliko *magičnih* konstanti koje menjaju vrednosti u zavisnosti od mesta na kom su pozvane.

Naziv	Opis
<code>__LINE__</code>	Trenutna linija u fajlu.
<code>__FILE__</code>	Putanja do fajla sa nazivom fajla i ekstenzijom.
<code>__DIR__</code>	Putanja do direktorijuma u kojem se nalazi fajl koji se trenutno izvršava. Ukoliko se koristi unutar inkludovanog fajla vratiće putanju do njega. Ekvivalent mu je: <code>dirname(__FILE__)</code> .
<code>__FUNCTION__</code>	Naziv funkcije.
<code>__CLASS__</code>	Naziv klase.
<code>__METHOD__</code>	Naziv metode.

# Destruktor - \_\_destruct

---

- PHP 5 dolazi sa dodatkom destruktorskog koncepta koji odgovara pristupu korišćenom u drugim objektno-orijentisanim jezicima poput C++.
- Destruktorska metoda se aktivira onog momenta kada više **nema referenci** na konkretan objekat.

```
class MyClass{  
    function __destruct() {  
        echo "TEST" ;  
    }  
}
```



# Destruktor - `__destruct` (2)

---

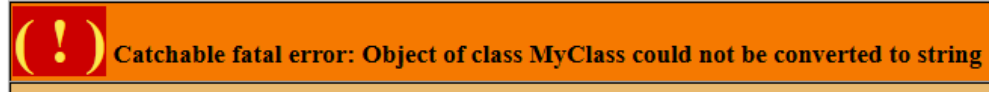
- `__destruct()` metode su pogodne za čišćenje smeća posle upotrebe objekta (npr. zatvaranje konekcije za bazom).
- Kada se izvršavanje fajla završi PHP se automatski oslobađa svih resursa. U skladu sa ovim treba biti oprezan prilikom definisanja `__destruct` metoda.

# Konvertovanje u String

---

- Ukoliko dođe do pokušaja konverovanja objekta u string doći će do greške:

```
class MyClass{  
    public $property1 ="Value1";  
    public $property2 ="Value2";  
}  
$o = new MyClass;  
echo $o;
```



# Konvertovanje u String (2)

---

- Kako bi se izbegla ovakva greška potrebno je definisati ponašanje objekta u trenutku kada neko pokušava da ga konveruje u string.
- Upravo u ovu svrhu se koristi nova magična metoda `__toString`.
- `__toString` metoda bi trebalo uvek da vraća vrednost umesto da samostalno vrši štampanje.
- Pogledajmo primer sa upotrebom `__toString` metode...

# Konvertovanje u String (3)

---

```
class MyClass{  
    public $property1 ="Value1";  
    public $property2 ="Value2";  
    public function __toString(){  
        return $this->property1 . " " . $this->property2;  
    }  
}  
$o = new MyClass;  
echo $o;
```

Value1 Value2

# Vidljivost metoda i svojstava

---

- Kako bi se kontrolisao pristup metodama i svojstvima objekata definišu se opsezi vidljivosti.
- Na ovaj način se definiše kako i sa kog mesta se može pristupiti svojstvima i metodama.
- Postoje tri ključne reči za definisanje vidljivosti:
  - public
  - private
  - protected
- Dodatno se svojstvo ili metoda može definisati kao statičko (*static*), čime se dozvoljava pristup bez instanciranja klase.

# Public Properties and Methods

---

- Najveći broj metoda i svojstva koje se kreiraju su javne: `public`.
- Public svojstvima i metodama se može pristupiti sa svih mesta (unutar i izvan klase).
- Metode kojima nije definisan drugi opseg su podrazumevano `public`.
- U verziji PHP 4 je `var` ekvivalent za `public`.
- ***Zadatak:*** Kreirati klasu koja poseduje dva `public` svojstva i jednu `public` metodu. Uloga metode je da vrati vrednosti svojstava.

# Protected Properties and Methods

---

- Metodi ili svojstvu definisanom kao protected može se pristupiti samo iz matične klase i iz klasa-potomaka (klase koje nasleđuju klasu koja sadrži protected metodu ili svojstvo).
- Pogledajmo ovo na primeru...


# Protected Properties and Methods (2)

---

```
class MyClass {
    public $prop1 = "I'm a class property!";
    protected function getProperty(){
        return $this->prop1 . "<br />";
    }
}

class MyOtherClass extends MyClass{
    public function newMethod() {
        echo "From a new method in " . __CLASS__ . ".<br />";
    }
}

$newobj = new MyOtherClass;
echo $newobj->getProperty();
```

 Fatal error: Call to protected method MyClass::getProperty()



# Protected Properties and Methods (3)

---

- Da bismo rešili pristup zaštićenoj metodi iskoristićemo klasu koja nasleđuje glavnu klasu da bismo kroz nju pristupili zaštićenoj metodi jer ona ima pravo ovog pristupa.
- Na ovaj način posredstvom subklase pristupamo glavnoj klasi i na taj način omogućavamo objektu da vidi vrednost koju vraća protected metoda.
- Pogledajmo ovo na primeru...

# Protected Properties and Methods (4)

---

```
class MyClass {
    public $prop1 = "I'm a class property!";
    protected function getProperty(){
        return $this->prop1 . "<br />";
    }
}
class MyOtherClass extends MyClass{
    public function newMethod() {
        echo "From a new method in " . __CLASS__ . "<br />";
    }
    public function callProtected(){
        return $this->getProperty();
    }
}
$newobj = new MyOtherClass;
echo $newobj->callProtected();
```

I'm a class property!

# Private Properties and Methods

---

- Metode i svojstva koja su definisana kao private su vidljiva samo iz matične klase.
- Treba imati u vidu da ni klasa-potomak, neće imati pristup privatnim metodama.
- Da bismo ovo demonstrirali getProperty metodu sa prethodnog primera ćemo definisati kao private.

# Private Properties and Methods (2)

---

```
class MyClass {
    public $prop1 = "I'm a class property!";
    private function getProperty(){
        return $this->prop1 . "<br />";
    }
}
class MyOtherClass extends MyClass{
    public function newMethod() {
        echo "From a new method in " . __CLASS__ . "<br />";
    }
    public function callProtected(){
        return $this->getProperty();
    }
}
$newobj = new MyOtherClass;
echo $newobj->callProtected();
```

 Fatal error: Call to private method MyClass::getProperty()

# Static Properties and Methods

---

- Metodama i svojstvima koja su definirana kao *static* se može pristupiti bez instanciranja klase.
- Za pristup statičkom svojstvu ili metodi dovoljno je iskoristiti naziv klase, scope resolution operator (::) i identifikator.
- Jedan od najvećih benefita statičkih svojstava je u tome što ona zadržavaju svoje vrednosti tokom trajanja skripte.
- Važno je zapamtiti da nakon operatora :: dolazi dolar znak (\$)  
pre postavljanja identifikatora svojstva.
- Pogledajmo ovo na primeru...

# Static Properties and Methods (2)

---

```
class MyClass {  
    public static $count = 0;  
    public static function plusOne() {  
        self::$count++;  
    }  
}
```

```
$newobj = new MyClass;  
$newobj->plusOne();  
$newobj->plusOne();  
echo MyClass::$count;
```

# Static Properties and Methods (3)

---

- Zadatak: Kreirati metodu koja će vraćati ukupan broj kreiranih instanci za klasu.

```
class MyClass {  
    public static $count = 0;  
    public function __construct(){  
        self::$count++;  
    }  
}
```

```
$newobj1 = new MyClass;  
$newobj2 = new MyClass;  
$newobj3 = new MyClass;  
echo MyClass::$count;
```

# Više o konstruktoru (vidljivost)

---

- Ukoliko obratite pažnju na do sada kreirane konstruktore primetićete da su svi definisani kao *public*.
- Postavljanje konstruktora na *private* ili *protected* izaziva grešku prilikom instanciranja.

```
class MyClass{  
    private function __construct(){ }  
}  
$o = new MyClass;
```

 Fatal error: Call to private MyClass::\_\_construct() from invalid context



# Više o konstruktoru (prioritet)

---

- Setimo se da za PHP4 konstruktor predstavlja metoda koja nosi isti identifikator kao i klasa.
- Šta će se dogoditi ukoliko u istoj klasi postoji konstruktor u vidu metode sa imenom klase i konstruktor tipa `__construct`?
- Primer:

```
class MyClass{  
    public function __construct(){  
        echo "__construct";  
    }  
    public function MyClass(){  
        echo "MyClass";  
    }  
}
```

PRIORITET



REGULARNA  
METODA

# Napomene dobre prakse

---

1. Umesto dodavanja vrednosti promenljivama nakon instanciranja, bolje je koristi konstruktor.
2. Voditi računa o definisanju vidljivosti metoda i svojstava. Klasa ne treba da bude ni nepotrebno zatvorena niti nepotrebno otvorena.
3. Budite konzistentni u sintaksi.
4. Svojstva i metode označene kao *protected* imenujte sa oznakom `_` na početku.
5. Pažljivo razmatrajte probleme koje ćete rešavati u jednoj klasi. Razdvajanje logike na veći broj klasa često je bolje rešenje od grupisanja koda u jednoj klasi.
6. Svaku klasu (ma koliko mala ona bila) kreirajte u zasebnom fajlu.
7. Prikom imenovanja fajlova za klase budite strogo dosledni.

# Dokumentovanje klase - DocBlocks

---

- DocBlock stil komentarisanja je najčešće korišćeni metod dokumentovanja klasa.
- Dokumentovanje klasa nije obavezno i ne podleže sintaksnim pravilima (sem pravila za postavljanje komentara u kod).
- DocBlock se definiše blokom komentara u kojem linije započinju zvezdicom (\*);
- Primer:

```
/**  
 * This is a very basic DocBlock  
 */
```

# Dokumentovanje klase – DocBlocks (2)

---

**@author:** John Doe <john.doe@email.com>.

**@copyright:** 2016 ITAcademy.

**@license:** <http://www.example.com/path/to/license.txt> License Name.

**@var:** Tip i opis promeljive ili svojstva klase.

**@param:** Tip i opis parametra funkcije ili metode.

**@return:** Tip i opis povratne vredosti funkcije ili metode.

# Konstante u klasi

---

- PHP 5 dozvoljava kreiranje svojstva sa konstantnom vrednošću unutar klase (konstanta).
- Konstantama se vrednost dodeljuje prilikom definisanja i njihova vrednost se kasnije ne može menjati.
- Konstante se podrazumevano pišu velikim slovima.
- Ispred konstanti ne stoji \$ znak.
- Podsetimo se kako se kreira obična konstanta:

```
define( "PI" , 3.14159 );  
echo PI;
```

# Konstante u klasi (2)

---

- Konstante u klasi se definišu korišćenjem ključne reči *const*.
- Primer:

```
class MyClass{  
const MYCONST = 300;  
}  
//Validna sintaksa:  
echo MyClass::MYCONST;  
//Nevalidna sintaksa:  
echo MyClass->MYCONST;
```

# Abstract Classes

---

- Uvođenje pojma apstraktnih klasa bila je jedna od glavnih novina koje je doneo PHP 5.
- Apstraktne klase se koriste onda kada želimo da kreiramo klasu koja će moći da bude nalađivana, ali neće biti dozvoljeno njeno instanciranje.
- Ako je klasa šablon za kreiranje objekata, onda je apstraktna klasa šablon za kreiranje dugih klasa.
- Apstraktne klase se definišu upotrebom ključne reči *abstract*.
- Primer:

```
abstract class MyClass{ }  
$o = new MyClass;
```



# Abstract Classes (2)

---

- Apstrakte klase mogu posedovati i metode koje mogu imati definisano telo.
- Ovakve metode se mogu prepisivati u klasama koje ih nasleđuju (*method overriding*).
- Apstrakne klase često poseduju i apstraktne metode.
- Apstraktne metode ne smeju imati definisano telo.
- Obavezno je izvršiti implementaciju apstraktne metode.
- Pogledajmo ovo na primeru...



# Abstract Classes (3)

---

```
abstract class MyClass{  
    const MYCONST = 300;  
    abstract public function abstractMethod();  
}
```

```
class myNewClass extends MyClass{  
    public function abstractMethod(){ }  
}
```

# Abstract Classes (4)

---

- Zašto ne koristiti obične klase umesto apstraktnih?
- Korišćenje apstraktnih klasa organizaciju koda čini urednijom i programsku logiku podiže na viši nivo apstrakcije.
- Apstraktne klase mogu osigurati poštovanje definisanih pravila od strane programera koji rade u timu.

# Final Classes and Methods

---

- Nasleđivanje klasa i prepisivanje metoda predstavljaju izuzetno korisne funkcionalnosti.
- Ipak, ponekad se klasa ili metoda mogu smatrati finalnim, odnosno klasama ili metodama nad kojima više ne treba da bude izmena.
- Finalizovanje klase ili metode vrši se ključnom rečju *final*.
- Klase ili metode ne treba činiti finalnim ukoliko nismo sigurni da nikada neće nastati situacija u kojoj bi njena izmena mogla biti korisna.
- Pogledajmo ovo na primeru...

# Final Classes and Methods (2)

---

```
final class MyClass {  
    public function msg() {  
        return "TEST" ;  
    }  
}  
  
class MyClass2 extends MyClass { }  
  
$m = new MyClass ;  
echo $m->msg() ;
```

 Fatal error: Class MyClass2 may not inherit from final class (MyClass)

# Final Classes and Methods (3)

---

```
class MyClass {  
    final function msg(){  
        return "TEST";  
    }  
}  
  
class MyClass2 extends MyClass{  
    public function msg(){  
        return "TEST 2";  
    }  
}  
  
$m = new MyClass;  
echo $m->msg();
```

 Fatal error: Cannot override final method MyClass::msg()