SKRIPTA IZ PREDMETA INTERNET TEHNOLOGIJE

Objektno orijentisano programiranje u PHP

SADRŽAJ

UVOD

Objektno orjentisan način progrmiranja ima nešto drugačiju logiku od proceduralnog, prvobitnog načina u php-u. U objektnom pristupu, mnogi složeni i komplikovani zadaci se znatno lakše rešavaju nego u standarnom obliku, i ne samo to, možda još što je i važnije, mnogo se lakše održavaju i menjaju.

Ideja uvođenja objekata i klasa, znatno pojednostavljuje logiku koda, takođe je i čitljiviji za programere i mnogo se lakše održava. Glavne stvari oko kojih se OOP (objektno orjentisano programiranje) vrti su objekti i klase. One su usko povezane i samim tim mnogi programeri koji se prvi put susreću sa ovakvim načinom rada, ih često mešaju.

Često se može čuti pitanje „Da li sve što mogu da uradim u OO programiranju mogu i u proceduralnom?“. Odgovor bi bio DA. Nakon toga bi usledilo pitanje „A čemu onda služi objektno orijentisano programiranje?“ Odgovor na ovo pitanje je nešto opširniji. Ukoliko se radi o jednostavnijim ili manjim aplikacijama i programima, apsolutno nema nikakve veze koji princip koristite. Medjutim, problemi nastaju kasnije, kada taj kod treba održavati i menjati na zahtev klijenata. Dobro struktuiran program napisan koristeći OO koncept se znatno lakše održava i menja. Ukoliko se primenjuju pravila dobrog kodiranja i softverski paterni, svaka modifikacija može da se uradi relativno jednostavno i lako, ma koliko ona zahtevna i velika bila. Sa druge strane, u proceduralnom programiranju, stvari ne idu baš tako lako. Programer sve može napraviti proceduralno, ali je pitanje može li kasnije taj kod održavati. Nisu retki primeri kada klijent traži neznatne promene u kodu (minimalna promena funkcionalnosti ili čak promena dizajna) a da je njih nemoguće ugraditi i sve treba pisati iz početka. Zamislite projekat na kojem raditite par meseci i neposredno pre isporuke klijent vam kaže „e, izmeni mi samo ovaj deo, umesto ovog neka radi ovako“, znači zahteva od vas minimalne promene u kodu, a s’obzirom da ste radili proceduralno i niste poštovali pravila programiranja, dolazi do velikog problema, i vrlo verovatno je da ćete morati da pišete ceo kod ponovo. Naravno, iskusni programeri ne moraju da rade sve ispočekta ako je program lepo struktuiran, ali svakako će imati mnogo više posla i komplikacija nego da su koristili OO način rada. Međutim, to ne znači da je i u OOP-u sve idealno, i tu ako se ne poštuju pravila, vrlo je moguće da se kod piše ispočetka na zhtev klijenta na minimalnu promenu.

To je glavna i možda najveća predsnost OOP-a u odnosu na proceduralno. Lakše održavanje, pregledniji kod i logičnije povezivanje entiteta!

OOP u PHP

1.1 Objekti

Objekat može da bude sve. Sve što se piše u proceduralnom režimu, može da se premesti u objekat. Bilo koji entitet možete se predstaviti objektnom. Ukoliko vam je u programu neophodno da predstavite neku životinju, recimo psa, njega možete predstaviti objektom. Ukoliko želite da predstavite drvo, školu, ulicu,.... sve to možete predstaviti objektom.

Uzećemo jedan primer, recimo treba da prikažemo psa, i napravićemo jedan objekat (biće reči nešto kasnije o tome kako se kreira objekat) koji smeštamo u promenljivu `$pas`. Taj pas ima neke karakteristike i neka ponašanja. Od karakteristika možemo da izdvojimo: dužinu dlake, boju dlake, visinu, dužinu ušiju, da li je pas ženka ili mužjak i tako dalje. Znači karakteristike opisuju životinju, tj. psa. U objektnom programiranju te karakteristike se nazivaju atributi ili polja. Znači objekat se sastoji od atributa/polja i oni opisuju stanje objekta.

Druga stvar koju možemo posmatrati je ponašanje psa ili neka radnja psa. To može da bude: hodanje, trčanje, lajanje, skakanje, ujed i tako dalje. Tako da pas pored svojih karakteristika ima i ponašanje. To ponašanje, prevedeno na OO jezik, predstavljaju metode. Znači dolazimo do zaključka da objekat ima atribute i metode. Atributi opisuju stanje objekta a metode opisuju radnju objekta. U OOP modu, to bi igledalo ovako:

```
$pas->boja_dlake = 'crna'; //postavljamo da pas ima crnu dlaku
$pas->visina = 45;  //postavljamo da je pas visok 45cm
$pas->lajanje();  //  pozivamo metudu koja simulara lajanje psa
$pas->trcanje();  //pozivamo metodu koja simuliranje trcanje psa
```

Kao što se vidi, objekti pozivaju atribute i metode tako što se prvo napiše promenljiva objekta (`$pas`), pa zatim se doda (`->`) i na kraju naziv metode ili naziv atributa. Sve ovo samo po sebi i nema nekog smisla, neophodno je objasniti i šta su klase, kako se kreiraju objekti, kako se kreiraju atributi i metode i tek onda će slika o OOP biti nešto jasnija.

1.2 Klase

Klasu možemo shvatiti kao „fabriku“ za kreiranje objekata. Klase predstavljaju neku šemu ili šablon kako će se kreirati objekti i kako će oni izgledati. U klasi definišemo i atribute i metode objekta. Ona bi iz našeg primera izgledala ovako:

```
class Pas{

      public $boja_dlake;
      public $visina;

      function lajanje(){

      }
      function trcanje(){
```

```
	}
}
```

Kao što se vidi, klase se kreira tako što se napiše ključna reč `class` i nakon toga dolazi proizvoljno ime klase, u našem slučaju pas. Nakon te deklaracije, otvara se vitičasta zagrada i na kraju i zatvara. Sve što se nalazi između vitičastih zagrada, predstavlja neku strukturu ili definiciju klase.

Atributi se definišu vrlo jednostavno, prvo se upotrebi ključna reč koja ukazuje na pristup i vidljivost atributa `public` (o tome nešto kasnije) i nakon nje, samo se naznače atributi kao obične promenljive `$boja_dlake` ili `$visina`, i tako je budućem objektu pridotata karakteristika koju on poseduje. Ponašanje objekta ili metode se definišu tako što se navede ključna reč `function` i nakon toga sledi naziv metode, obične zagrade pa vitišaste. U običnim zagradama se upisuju parametri (potpuno identično kao i kod obične funkciju u proceduralnom režimu) ili mogu da budu prazne, kao u našem slučaju, a između vitišastih se definiše ponašanje ili radnja koja treba da se izvrši nakon poziva te metode, i taj deo između vitičastih zagrada se naziva telo metode.

Dolazimo sada do pitanja, kako kreirati objekat uz pomoć klase. To je jako jednostavno i čini se na sledeći način:

```
$pas = new Pas();
```

Da bi se kreirao objekat, navodi se naziv objekta koji je proizvoljan, `$pas`, zatim sledi znak jednakosti, dolazi rezervisana reč `new` i na kraju naziv klase sa praznim zagradama (naravno, prazne su jer nema parametara, postoje primeri i kada se koriste parametri, ali o tome kasnije).

Ovo je najjednostavniji i osnovni primer kako se kreiraju klase i objekti i šta sadrže. Sada ćemo dati kompletan primer kako bi to izgledalo:

```
class Pas{

	public $boja_dlake;
	public $visina;

	function lajanje(){
		echo 'avav avav';
	}
	function trcanje(){
		echo 'trcim trcim';
	}
}


$pas = new Pas();
$pas->boja_dlake = 'crna';
$pas->visina = 45;
$pas->lajanje();
echo '<br/>';
$pas->trcanje();
echo '<br/>';
echo $pas->boja_dlake;
echo '<br/>';
```

```
echo $pas->visina;
```

na monitoru, kao rezultat ovog koda, bi pisalo:

```
avav avav
trčim trčim
crna
45
```

Pre nego što nastavio dalje, uporedićemo primer dve biblioteke mysql i mysqli. Obe se odnose na konekciju i manipulaciju sa bazom, s tim što je prva proceduralna, a druga objektna. Ovako bi to izgledalo:

mysql biblioteka:

```
$user = "username";
$password = "password";
$database = "database";
$host = "localhost";

mysql_connect($host,$user,$password);  //konekcija na server
mysql_select_db($database);  //konekcija na bazu
$query = "INSERT INTO korisnici (id, ime, prezime) VALUES (3,
'Pera','Peric')"; //pisanje upita
mysql_query($query); //izvršavanje upita
mysql_close();  //zatvaranje konekcije
```

mysqli biblioteka:

```
$user = "username";
$password = "password";
$database = "database";
$host = "localhost";

$mysqli = new mysqli($host,$user,$password, $database); //konekcija
na server i bazu
$query = "INSERT INTO korisnici (id, ime, prezime) VALUES (3,
'Pera','Peric')"; //pisanje upita
$mysqli->query($query); //izvršavanje upita
$mysqli->close();  //zatvaranje konekcije
```

Kao što se i vidi, nema velike razlike u veličini i obimu koda (u ovom primeru), i on je poslužio samo da se vidi koja je razlika u sintaksi u odnosu na proceduralno i objektno programiranje.

1.3 Atributi i modifikatori

Ranije je bilo reči o atributima i šta oni predstavljaju.. Atributi, kao što smo rekli, opisuju stanje objekta. Ukoliko imamo primer, da je objekat pas, atributi mogu bit: boja dlake, visina, pol, starost i tako dalje. Vratićemo se ponovo na naš primer sa psom:

```
class Pas{

	public $boja_dlake;
	public $visina;

	function lajanje(){
		echo 'avav avav';
	}
	function trcanje(){
		echo 'trcim trcim';
	}
}
```

Ukoliko želimo da dodelimo nekom atributu neku vrednost to radimo na sledeći način:

```
$pas = new Pas();
$pas->boja_dlake = 'crna';
echo $pas->boja_dlake;
```

Na ekranu bi se pojavilo:
`crna`

Prvo smo kreirali objekat `$pas`, a zatim pozvali atribut `boja_dlake` i dodelili joj crnu boju. E sad, dolazi na red pitanje, čemu služi `public`. Sa `public`–om označavamo da je atribut javni i da može svako da mu pristupi. Tačnije, mi smo kreirali objekat `$pas` i direkto uz pomoć `->boja_dlake` mu pristupili. Međutim, to u praksi i nije najbolje rešenje i uvek se izbegava ta mogućnost. Mnogo je bolje i pametnije koristiti rezervisanu reč `private`. `private` ograničava pravo pristupa samo na klasu, i taj atribut možete pozvati samo iz klase, dok atribut označen kao `public`, možete pozvati i iz klase i van nje. Ovako bi to izgledalo:

```
class Pas{

	private $boja_dlake;
	private $visina;

	function lajanje(){
		echo 'avav avav';
	}
	function trcanje(){
		echo 'trcim trcim';
	}
}

$pas = new Pas();
$pas->boja_dlake = 'crna';  // GREŠKA !!!!!!!!
echo $pas->boja_dlake;
```

Ovaj kod ne bi se izvršio jer pokušavamo da pristupimo atributu `boja_dlake`, a on je označen kao `private`. Onda se postavlja pitanje kako se pristupa tim atributima. To se radi enkapsulacijom, kreiraju se get i set metode. To bi izgledalo ovako:

```
class Pas{

	private $boja_dlake;
	private $visina;

	function getBojaDlake(){
		return $this->boja_dlake;
	}
	function setBojaDlake($bojaDlake){
		$this->boja_dlake = $bojaDlake;
	}
	function lajanje(){
		echo 'avav avav';
	}
	function trcanje(){
		echo 'trcim trcim';
	}
}

$pas = new Pas();
$pas->setBojaDlake('crna');
echo $pas->getBojaDlake();
```

Na ekranu bi se pojavilo:
`crna`

Time smo zaštitili atribut od direktnog pristupa i omogućili da mu se pristupa preko set i get metoda. To se u programiranju naziva enkapsulacija. Preko metode set postavljamo vrednost, a preko metode get uzimamo vrednost (naredba `return` vraca vrednost i izlazi iz metode)

Postoji još jedan modifikator, a to je `protected`. On takođe ne dozvoljava da se pristupi izvan klase, ali za razliku od `private`, on dozvoljava pristup i nasleđenoj klasi (nešto kasnije o tom) dok `private` dozvoljava pristup samo u datoj klasi.

Da rezimiramo:
Kada je neki atribut ili neka metoda privatna, tj. ispred naziva atributa i metode stoji `private` (uzmimo za primer atribut `boja_dlake` kod psa), to znači da atributu `boja_dlake` možete pristupiti SAMO i SAMO unutar date klase, tačnije iz tela klase. Svaki pokušaj pristupa tom atributu van klase predstavljao bi gresku. Telo klase predstavlja prostor između vitičastih zagrada

```
class Pas{
	// telo klase
}
```

pa prema tome, ovo bi nesumnjivo bila greška:

```
class Pas{
	private $boja_dlake;
}

$pas = new Pas();
$pas->boja_dlake = 'crna';  // GREŠKA !!!!!!!!
```

jer pokušavamo da pristupimo atributu `boja_dlake` van tela klase. Ovo se odnosi i na metode jer se i one potpuno isto ponašaju kao i atributi.

Međutim, ukoliko bi ste umesto private stavili public, mogli bi ste nesmetano pozivati atribute i metode bilo gde, pa i izvan klase:

```
class Pas{
	public $boja_dlake;
}

$pas = new Pas();
$pas->boja_dlake = 'crna';  // ISPRAVNO
```

Rezervisana reč private se uglavnom upotrebaljava za atribute, i oni po nekom pravilu uvek bi trebali da budu privatni. Ukoliko želimo da im pristupimo izvan te klase, neophodno je u klasi kreirati set i get metodu i tako bi rešili problem komunikacije sa tim atributom.

Primer:
Zamislimo studentsku službu koja ima vaš dosije. U našem primeru, studentska služba neka bude klasa, a dosije atribut koji ona poseduje. Potrebno je zatražiti od studentske službe dosije. Postoje dva načina: prvi, da slobodno uđemo u studentsku službu, bez kucanja i pitanja, malo prorovarimo po dosijeu, nađemo naš, uzmemo i izađemo. Drugi slučaj je da dođemo do šaltera i zamolimo za naš dosije. Službenica ga potraži i vrati nam ga kroz šalter.

Prvi primer je opisao situaciju u kojoj je dosije označen kao public, znači bilo ko i bilo kad mu može pristupiti i uzeti šta mu treba. Što je najgore, može ga i menjati bez pitanja. Drugi primer opisuje situaciju kada je dosije privatan, pa je moguće mu pristupiti samo preko get metode (šaltera).

Što se tiče metoda, one se uglavnom obeležavaju kao public. Naravno, ukoliko postoje i neke metode koje neće niko koristiti osim date klase, onda i one bi mogle da budu private.

Primer sa studentskom službom:
Potrebno je upisati narednu godinu. Za to je potrebno izvršiti proveru broja bodova, zatim je potrebno da se student zavede u bazu podataka i na kraju da se upiše u index godina koju je upisao. Za to su nam potrebne tri metode: `proveraBodova()`, `upisUBazu()` i `upisUIndex()`.

```
class SSluzba{

	public proveraBodova(){

	}
```

```
	public upisUBazu(){

	}
	public upisUIndex(){

	}
}

$sluzba = new SSluzba();
$sluzba->proveraBodova();
$sluzba->upisUBazu();
$sluyba->upisUIndex();
```

Ovaj kod preveden u praksi bi značio: dolazimo na šalter i dajemo index. Zatim kažemo službenici da proveri broj bodova. Ona proverava i kaže da je uredu, nakon toga, kažemo joj da nas upiše u bazu, što ona i čini, i na kraju da nam sve to upiše i u index. Ona upisuje u index i vraća nam kroz šalter. Međutim, da li je u praksi baš tako, da li vi govorite šta službenica da radi ili ona sama zna svoj posao? Naravno da zna šta joj je činiti, zato mi i ne moramo da joj pričamo korake, već samo da joj damo index i kažemo da želimo da upišemo godinu, a ona sama obavlja taj posao. U tom slučaju, mi ne moramo da znamo da li će ona proveravati bodove, da li treba da nas upiše u bazu i ili možda treba još 10ak stvari da uradi. To nas ne zanima i to je njen posao. Naše je samo da kažemo da želimo da budemo upisani u novu školsku godinu i to je to. Onda bi to predstavili na ovaj način:

```
class SSluzba{

	public upisiGodinu(){
		$this->proveraBodova();
		$this->upisUBazu();
		$this->upisUIndex();
	}
	private proveraBodova(){

	}
	private upisUBazu(){

	}
	private upisUIndex(){

	}
}

$sluzba = new SSluzba();
$sluzba-> upisiGodinu();
```

Obratite pažnju da su metode `proveraBodova()`, `upisUBazu()` i `upisUIndex()` privatne i da niko ne može da im pristupi osim klase SSluzba. Mi sada možemo da pristupimo samo metodi `upisiGodinu()` i mi samo za tu metodu i znamo u praksi, a nemamo pojma šta službenica radi iza šaltera, to je njen posao. Kada pozovemo metodu `upisiGodinu()` ona poziva ostale tri metode koje su neophodne, a mi jednostavno o tome ne moramo da vodimo računa niti da znamo za njih, pa su iz tog razloga i obeležene kao private.

1.4 Metode i parametri

Do sada smo objasnili šta su metode. One predstavljaju ponašanje klase. U predhodom primeru set i get takođe predstavljaju metode (`getBojaDlake()`, `setBojaDlake($bojaDlake)`), s tim da druga metoda ima jedan parametar `$bojaDlake`. Preko parametra metodi prosledjujemo neku vrednost, zatim se ta vrednost obradi i rezultat vrati. U predhodnom primeru, ovaj parametar je služio samo da bi atributu `$boja_dlake` u kalsi Pas dodelili vrednost. Evo primera kako metoda prima parametre i sabira ih i vraća rezultat:

```
class Matematika{

	function saberi($broj1, $broj2){
		$zbir = $broj1 + $broj2;
		return $zbir;
	}
}


$mat = new Matematika();
$rezultat = "Zbir je ".$mat->saberi(3,7);
echo $rezultat;
```

Kao što se vidi, metode potpuno na isti način funkcionišu kao i funkcije u proceduralnom programiranju, samo što se pozivaju preko objekta, u ovom slučaju preko `$mat`.

1.5 Statičke metode

Metode mogu biti i statičke. To znači da nisu vezane za konkretni objekat, već se pozivaju preko klase. Tačnije, ne mora biti kreiran objekat da bi ste pozvali metodu, već to možetu uraditi direktno preko imena klase. To bi izgledalo ovako:

```
class Matematika{

	static function saberi($broj1, $broj2){
		$zbir = $broj1 + $broj2;
		return $zbir;
	}
}

$rezultat = "Zbir je ". Matematika::saberi(3,7);
echo $rezultat;
```

Ponekad su statičke metode korisnije, i upotrebljavaju se onda kada nije neophodno da se kreira objekat, već kada je logičnije da metoda bude vezana za jednu klasu.

Metode kao i atributi mogu imati modifikatore i vladaju potpuno ista pravila kao i za atribute. Oni se dodaju prvi, ispred `function` i `static` (ako je metoda staticka)

1.6 Konkstruktori

Do sada nismo pominjali niti kreirali konstruktore, ali svaka klasa obavezno sadrži konstruktor. Onda se postavlja pitanje, kako to da nismo pisali konstruktore, ako svaka klasa mora da sadrži jedan, a program je radio ?

Odgovor je da ukoliko mi ne kreiramo konstruktor, to klasa radi automatski za nas.

Konstruktori je zapravo obična metoda koja se automatski poziva pri kreiranju objekta. Znači to je metoda koja se uvek poziva, i to pri kreiranju obejkta.

Kada napišemo ovako:

```
class Pas{

	private $boja_dlake;
	private $visina;

	function lajanje(){
		echo 'avav avav';
	}
	function trcanje(){
		echo 'trcim trcim';
	}
}
```

to ne znači da ne postoji konstruktor. On se automatski kreira. Taj kod je ekvivalentat ovom:

```
class Pas{

	private $boja_dlake;
	private $visina;

	public function __construct(){

	}

	function lajanje(){
		echo 'avav avav';
	}
	function trcanje(){
		echo 'trcim trcim';
	}
}
```

Konstruktor mora imati odredjenu formu, i metoda mora da se zove ili `__construct()` ili kao i naziv klase. Konstruktor služi da bi se inicijalizovale promenljive na samom početku. Pogledajmo primer:

```
class Pas{

	private $boja_dlake;
	private $visina;

	public function __construct(){
		$this->boja_dlake = 'crna';
		$this->visina = 45;
	}

	function lajanje(){
		echo 'avav avav';
	}
	function trcanje(){
		echo 'trcim trcim';
	}
}
```

To znači, da ukoliko napišete `$pas = new Pas();` , tada će i atributi biti inicijalizovani i bice im dodeljena vrednost. Onda nije neophodno setovati te atribute naknadno preko set metode, jedino ako hoćete da promenite te vrednosti.

S obzirom da je konstruktor metoda, i on može imati parametre. Konstruktor možemo napisati i ovako:

```
public function __construct($bojaDlake, $visina){
	$this->boja_dlake = $bojaDlake;
	$this->visina = $visina;
}
```

U tom slučaju bi objekat kreirali na ovaj način:

```
$pas = new Pas('crna', 45);
```

1.7 Konstante

Klasa može sadržati i konstante, koje nisu vezane za objekte, već samo za klasu. Tako, klasa Matematika, bi mogla imati konstantu pi. Ona je ne zavisna od bilo kog objekta i bilo koje matematičke funkcije i uvek je 3.14. One se predstavljaju tako što se ispre naziva

konstante doda ključna reč `const`. Uglavno se pišu velikim slovima, to je neko ne pisano pravilo, radi lakšeg snalaženja programera. Primer:

```
class Matematika{

	const PI = 3.14;

	static function pomnoziSaPI($broj1){
		$proizvod = $broj1 * self::PI;
		return $proizvod;
	}
}
```

Konstante se izvan klase pozivaju preko naziva klase `Matematika::PI`, a u samoj klasi preko ključne reči `self`, na ovaj način `self::PI`

1.8 Nasleđivanje

Dolazimo do dela gde objektno orijentisano programiranje dobija puno na značaju. Do sada je bilo reči o osnovim pojmovima u klasama i objektima, a sada dolaze najznačajniji delovi OOP-a.

Šta je zapravo nasleđivanje?
Svaka klasa sadrži određene atribute i metode preko kojih se opisuje stanje i ponašanje iste. Zamislimo da u programu kreiramo dve klase, od kojih obe sadrže dobar deo istog koda. Tačnije, sadrže nekoliko zajedničkih promenljivih i zajedničkih metoda, koje moraju da budu identične. Recimo, hoćemo da kreiramo klasu Pas i klasu Mačka. Obe klase predstavljaju životinje, i obe klase imaju neke zajedničke osobine kao što su: visina, težina, hodanje, trčanje i tako dalje. Ali pored zajedničkih osobina, postoje i one specifične. Recimo pas lovi mačke, dok mačka lovi miša, pas laje a mačka mjauče i tako dalje.

Analogno tome, imamo dve klase:

```
class Pas{

	private $tezina;
	private $visina;
	private $brojUhvacenihMacaka;

	public function __construct($tezina, $visina,
				$brojUhvacenihMacaka){

		$this->tezina = $tezina;
		$this->visina = $visina;
		$this->brojUhvacenihMacaka = $brojUhvacenihMacaka;
	}

	function trcanje(){
		echo 'trcim trcim';
	}
```

```
	function uloviMacku(){
		echo $this->brojUhvacenihMacaka += 1;
	}
}



class Macka{

	private $tezina;
	private $visina;
	private $brojUhvacenihMiseva;

	public function __construct($tezina, $visina,
				$brojUhvacenihMiseva){

		$this->tezina = $tezina;
		$this->visina = $visina;
		$this->brojUhvacenihMiseva = $brojUhvacenihMiseva;
	}

	function trcanje(){
		echo 'trcim trcim';
	}
	function uloviMisa(){
		echo $this->brojUhvacenihMiseva += 1;
	}
}
```

Kao što se vidi, i klasa Pas i klasa Mačka imaju neke zajedničke atribute i metode. U ovom primeru razlikuju se samo po tome što mačka lovi miša a pas mačku. Jedno od osnovnih pravila programiranja je **nikada ne imati jedan isti kod na dva različita mesta !!!** To znači da kod u celom programu mora biti jedinstven. Recimo, mi imamo metodu `trcanje()` i ona se nalazi u obe klase. Ukoliko kasnije želimo da je izmenimo, menjaćemo na oba mesta. Zamislimo da imamo 1000 životinja i na 1000 mesta metodu `trcanje()` i da treba svaku da izmenimo – posao ogroman, rezultat mali.

Nasleđivanje nam rešiva ovaj problem. Ono služi da se svi zajednički podaci smeste u roditeljsku klasu a specifični u nasleđenu. U našem slučaju, roditeljska klasa bi sadržala zajedničke metode i atribute koje imaju klase Pas i Mačka. Nju bi mogli da nazovemo Životinja. Ta klasa, sadržala bi zajedničke atribute i metode obe klase, a specifične radnje bi prepustila specifičnim klasama.

```
class Zivotinja{

	private $tezina;
	private $visina;

	public function __construct($tezina, $visina){
		$this->tezina = $tezina;
		$this->visina = $visina;
```

```
	}

	function trcanje(){
		echo 'trcim trcim';
	}
}
```

Klasa Zivotinja u ovom primeru sadži samo atribute i metode koje su sajedničke i za klasu Pas i za klasu Macka. Sada zelimo da kreiramo nove klase Pas i Macka, tako da iskoristimo funkcionalnost klase Zivotnja. Ono sto ćemo morati da uradimo, to je da nasledimo klasu Zivotinja. Nasleđivanje se čini pomoću ključne reči `extends`

```
class Pas extends Zivotinja{

	private $brojUhvacenihMacaka;

	public function __construct($tezina, $visina,
				$brojUhvacenihMacaka){

		parent::__construct($tezina, $visina);
		$this->brojUhvacenihMacaka = $brojUhvacenihMacaka;
	}
	function uloviMacku(){
		echo $this->brojUhvacenihMacaka += 1;
	}
}
```

Kao što se vidi, iz klase Pas su izbačne sve metode i atributi koji postoje u klasi Zivotinja, jer ih klasa Pas sve nasleđuje. Novina u ovoj klasi je ta, da konstruktor prima idalje parametre `$tezina` i `$visina`, a tih parametara nemamo u datoj klasi. Tako je, oni ne postoje u nasleđenoj klasi (Pas), ali postoje u roditeljskoj klasi (Zivotinja). Kada smo primili te parametre, neophodno je pozvati nadređeni konstruktor, tj. roditeljski konstruktor i proslediti im date parametre. To se čini pomoću ključne reči `parent`. Ta ključan reč može da se upotrebi bilo za pozivanje konstruktora bilo za pozivanje metoda nadređene klase.

VAŽNO!
Nadređeni konkstruktori u PHP-u se ne pozivaju automatski kao što je kod JAVE i C# slučaj, već mora se pozvati neposredno korišćenjem ključen reči `parent`.

Klasa Mačka bi izgledala ovako:

```
class Macka extends Zivotinja{

	private $brojUhvacenihMiseva;

	public function __construct($tezina, $visina,
				$brojUhvacenihMiseva){

		parent::__construct($tezina, $visina);
		$this->brojUhvacenihMacaka = $brojUhvacenihMiseva;
```

```
	}
	function uloviMisa(){
		echo $this->brojUhvacenihMiseva += 1;
	}
}
```

Iz ovog primera se vidi da smo zajedničke podatke podigli na viši (abstraktniji) nivo, a specifičene spustili na niži (specifičniji) nivo. Sada više nema dupliranja koda i ovim smo skratili vreme razvoja, preglednosti i smaljili mogućnost grešaka u daljem projektovanju.

Šta biva ako nam sada u klasi Pas zatreba njegova tezina? Recimo hoćemo da implementiramo metodu da na monitoru ispiše koliko se pas ugojio, tačnije koliko trenutno ima kilograma, ako mu se od svake ulovljenje mačke poveća težina za 100 gr. To bi izgledalo ovako:

```
class Pas extends Zivotinja{

	private $brojUhvacenihMacaka;

	public function __construct($tezina, $visina,
				$brojUhvacenihMacaka){

		parent::__construct($tezina, $visina);
		$this->brojUhvacenihMacaka = $brojUhvacenihMacaka;
	}
	function uloviMacku(){
		echo $this->brojUhvacenihMacaka += 1;
	}
	function povecajTezinu(){
		$povecanjeTezine = $this->brojUhvacenihMacaka * 0.1;
		$novaTezina = $this->tezina + $povecanjeTezine;
		echo $novaTezina;
	}

}
```

Prva linija koda u metodi `povecajTezinu()` računa vrednost koliko se ukupno pas nagojio od mačaka ako je za svako ulovljenu mačku dobio po 100gr. Druga linija koda, tu vrednost sabira sa trenutnom težinom psa, a treća linija koda prikazuje rezultat. Isprobajmo da vidimo kako radi:

```
$pas = new Pas(25, 35, 2);
$pas->povecajTezinu();
```

Kreiramo objekat pas sa težinom od 25kg, visinom 35kg i sa 2 ulovljene mačke. Prema našim proračunima, nakon izvršavanja ovog koda, rezultat bi bio

25.2

Međutim, nakon izvršavanja koda, vidimo da je rezultat 0.2. Krenimo redom da analiziramo kod:

```
$povecanjeTezine = $this->brojUhvacenihMacaka * 0.1;
```
naš slučaj: `$povecanjeTezine = 2 * 0.1;`
vrednost: `$povecanjeTezine = 0.2;`

```
$novaTezina = $this->tezina + $povecanjeTezine;
```
naš slučaj: `$novaTezina = 25 + 0.2;`
vrednost: `$novaTezina = 25.2;`

```
echo $novaTezina;
```
naš slučaj: 25.2

Po logici, ovo je sasvim kako treba, ali gde je greška. Čini se kao da on uopšte ne uzima 25, tj. težinu psa. Pa da, tu je problem !!! Zašto ?

Ukoliko pogledamo klasu Zivotinja, uočićemo ovo:

```
private $tezina;
```

Atribut težina je privatni atribut, i kao što smo kazali ranije, on dozvoljava poziv tog atributa samo iz svoje klase, tačnije klase Zivotinja. Čak ne dozvoljava ni pozivanje iz izvedene (nasleđene) klase Pas. Da bi smo izbegli `public`, a iz nekog razloga ne želimo da pravimo javne get i set metode, jedino rešenje je modifikator `protected`. On dozvoljava da se atribut poziva iz svoje klase, ali isto tako dozvoljava da se poziva i iz nasleđene klase. U svim ostalim slučajevima se ponaša kao `private` (tačnije ne dozvoljava poziv iz drugih klasa). Ukoliko to zamenimo i u klasi Zivotinja stavimo

```
protected $tezina;
```

naša metoda `povecajTezinu()` će se izvršiti baš onako kako i očekujemo.

1.9 Prikrivanje metoda

Iskoristimo predhodni primer Zivotinje, Psa i Mačke. Za sve životinje je zajedničko da love hranu, ali na koji način i šta love, razlikuje se od životinje do životinje. Mi smo u predhodnom primeru to rešili tako što smo implementirali metode `uloviMacku()` i `uloviMisa()` u zavisnosti o kojem se objektu radi. E sada, hoćemo samoinicijativno ili na zahtev korisnika to da izmenimo i da se obe metode zovu `uloviHranu()`. Pored toga, želimo da i sve životinje koje se naknadno dodaju, poseduju pomenutu metodu. Želimo da izbegnemo da se svakoj životinji dodeljuje novo ime metode, već da sve koriste jedno te isto ime, a to je `uloviHranu()`.

Ukoliko analiziramo stvar, shvatamo da je naziv metode zajednički za sve životinje, a njegova implementacija (telo metode gde se upisuje konkretna radnja) zavisi od životinje do životinje. Tako i pas i mačka i krokodil i žaba i sve ostale životinje love hranu, to im je zajedničko, ali one se ne hrane istom hranom, i to im je razlika. Pa na osnovu toga, u klasi Zivotinja, tebalo bi da kreiramo metodu `uloviHranu()` a u posebnim klasama (Pas, Macka,...) uradimo implementaciju i navedemo čime se određena životinja hrani. To bi izgledalo ovako:

```
class Zivotinja{

	private $tezina;
	private $visina;

	public function __construct($tezina, $visina){
		$this->tezina = $tezina;
		$this->visina = $visina;
	}

	function trcanje(){
		echo 'trcim trcim';
	}
	function uloviHranu(){
		echo 'lovim hranu';
	}

}



class Pas extends Zivotinja{

	private $brojUhvacenihMacaka;

	public function __construct($tezina, $visina,
				$brojUhvacenihMacaka){

		parent::__construct($tezina, $visina);
		$this->brojUhvacenihMacaka = $brojUhvacenihMacaka;
	}
}
```

Ukoliko kreiramo objekat klase Pas i pozovemo metodu `uloviHranu()` , na ekranu će se prikazati `lovim hranu(`možemo da pozovemo tu metodu jer je ona definisana u klasi Zivotinja, pa kako je klasa Pas nasleđuje, onda je sve regularno`)`. Međutim, mi ne želimo da se to prikaže jer znamo šta pas lovi, i hoćemo da se na ekranu prikaže `lovim mačke`. To ćemo uraditi tako što ćemo dodati tu metodu u klasi Pas. To bi igledalo ovako:

```
class Pas extends Zivotinja{

	private $brojUhvacenihMacaka;

	public function __construct($tezina, $visina,
				$brojUhvacenihMacaka){

		parent::__construct($tezina, $visina);
		$this->brojUhvacenihMacaka = $brojUhvacenihMacaka;
	}
	function uloviHranu(){
		echo 'lovim mačke';
```

```
	}
}
```

Sada ako kreiramo objekat klase Pas i pozovemo metodu `uloviHranu()` , na ekranu će pisati `lovim mačke`. Možda je sada nedoumica oko toga, koju će metodu pozvati, metodu `uloviHranu()` u klasi Zivotinja ili u klasi Pas. Prilikom odabira metode, uvek se prvo gleda metoda koja se nalazi u instanciranoj klasi, tačnije u klasi iz koje je kreiran objekat. Kako je naš objekat instanca klase Pas, prvo će se izvršiti metoda u toj klasa, pa ako metoda ne postoji u toj klasi (kao u malopređašnjem slučaju), onda se poziva roditeljska metoda. Ovaj sistem se zove prikrivanje metoda.

To je dosta korisno i vrlo često se primenjuje u programiranju. U našem slučaju bi bilo korisno ukoliko bi smo želeli da kreiramo klasu neke životinje, a da ne znamo čime se ona hrani. Jednostavno ne bi u toj novoj klasi dodali metodu uloviHranu(), tj. ne bi prikrili roditeljsku metodu, i uvek kada bi je iz programa pozvali, pisalo bi „lovim hranu“.

1.10 Abstraktne klase

Pogledajmo predhodni primer

```
class Zivotinja{

	private $tezina;
	private $visina;

	public function __construct($tezina, $visina){
		$this->tezina = $tezina;
		$this->visina = $visina;
	}

	function trcanje(){
		echo 'trcim trcim';
	}
	function uloviHranu(){
		echo 'lovim hranu';
	}

}
```

Ukoliko dobijemo zahtev da SVE klase koje nasleđuju klasu Zivorinja moraju da reimplementiraju metodu uloviHranu() (tačnije da dodaju tu metodu u svoju klasu), onda se postavlja pitanje, šta će nam ta metoda u roditeljskoj klasi ako nikada neće biti izvršena. Tačnije, ako za svaku životinju znamo čime se hrani, onda nam ova generalna metoda i nije potrebna. Ono što nama treba sada u roditeljskoj klasi je samo potpis metode (u potpis metode spada modifikator, rezervisana reč function, ime metode i parametri koje ona prima), ali ne i telo. To možemo napisati ovako:

```
function uloviHranu();
```

Ova linija koda je pogrešna, jer posle naziva metode i parametara koje prima, očekuju se vitičaste zagrade i telo metode. Da bi ovo rešili, možemo da napišemo ovako:

```
abstract function uloviHranu();
```

ovo je sada ok. Sada smo metodu `uloviHranu()` označili kao abstraktnu, a to znači da ima sve kao i obična metoda osim tela. Ispred `function` stoji i ključna reč `abstract` kako bi pokazala da je metoda abstraktna i da nema telo. Do sada je sve kako treba, međutim, idalje postoji problem. Obične klase ne mogu sadržati abstraktne metode. Da bi i ovo rešili, moramo i klasu proglasiti abstraktnom tako što ćemo ispred ključne reči `class` dodati `abstract`. To bi izgledalo ovako

```
abstract class Zivotinja{

	private $tezina;
	private $visina;

	public function __construct($tezina, $visina){
		$this->tezina = $tezina;
		$this->visina = $visina;
	}

	function trcanje(){
		echo 'trcim trcim';
	}
	abstract function uloviHranu();
}
```

Glavna razlika između abstraktne i obične klase je ta što abstraktna klasa ne može da ima objekat, tačnije, ovo bi sada predstavljalo grešku:

```
$zivotinja = new Zivotinja(); // graška
```

Ono što je još jako važno, a to je da svaka klasa koja nasledi abstraktnu, mora da reimplementira sve njene abstraktne metode. Tako da sve klase koje naslede klasu Zivotinja, moraće da u sebi definišu metodu `uloviHranu()`.

1.11 Interfejs

Interfejs je jako sličan abstraktnoj klasi, ali razlika u tome je što abstraktna klasa pored abstraktnih metoda može sadržati i obične, ne abstraktne metoda, dok su kod interfejsa sve metode abstraktne. On se obeležava sa ključnom reči `interface` ispred naziva interfejsa. U našem primeru bi napisali

```
interface Zivotinja{
	function uloviHranu();
}
```

Primećujemo da i pored toga što je metoda `uloviHranu()` abstraktna, nema ispred reči `function` reč `abstract`. To je zato što interfejs i ne može da ima druge metode osim abstraktnih, tako da se podrazumeva da su sve metode abstraktne i iz tog razloga ta ključna reč može da se izostavi.

Druga razlika u odnosu na abstraktne kalse je ta, što se interfejs nasleđuje ključnom reči `implements`, dok abstraktna klasa ključnom reči `extends`.

```
class Pas implements Zivotinja{
	function uloviHranu(){

	}
}
```

Kao i kod abstraktnih klasa, svaka klasa koja implementira neki interfejs, mora implementirati i sve njegove metode.

1.12 this, parent i self

Do sada smo upotrebljavali ključne reči `this`, `parent` i `self`, a sada je vreme da kažemo kada se one koriste.

`this` se koristi kada smo u nekoj klasi i želimo da pozovemo neku metodu iz te klase ili se obratimo nekom atributu iz te klase.

```
abstract class Zivotinja{

	private $tezina;
	private $visina;

	public function __construct($tezina, $visina){
		$this->tezina = $tezina;
		$this->visina = $visina;
	}

	function trcanje(){
		echo 'trcim trcim';
	}
	abstract function uloviHranu();
}
```

`parent` se koristi kada želimo da pozovemo neku metodu u roditeljskoj klasi

```
class Pas extends Zivotinja{

	private $brojUhvacenihMacaka;

	public function __construct($tezina, $visina,
				$brojUhvacenihMacaka){

		parent::__construct($tezina, $visina);
		$this->brojUhvacenihMacaka = $brojUhvacenihMacaka;
	}
	function uloviHranu(){
		echo 'lovim mačke';
	}
```

```
}
```

U ovom primeru smo pozvali nadređeni konstruktor ključnom reči parent. Međutim moguće je pozvati i bilo koju metodu u roditeljskoj klasi. Recimo mogli bi smo da stavimo i ovo:

```
class Pas extends Zivotinja{

	private $brojUhvacenihMacaka;

	public function __construct($tezina, $visina,
				$brojUhvacenihMacaka){

		parent::__construct($tezina, $visina);
		$this->brojUhvacenihMacaka = $brojUhvacenihMacaka;
	}
	function uloviHranu(){
		echo 'lovim mačke';
	}

	function prikaziTrcanjePsa(){
		parent::trcanje();
	}
}
```

self za pozivanje statičkih metoda ili konstanti iz postojeće klase.

```
class Matematika{

	const PI = 3.14;

	static function pomnoziSaPI($broj1){
		$proizvod = $broj1 * self::PI;
		return $proizvod;
	}
}
```

1.13 Rukovanje izuzecima

Greške prilikom pisanja koda se uvek dešavaju, ponekad namerno, ponekad slučajno. Nekad ih je jako teško otkriti, i morali bi ste se pozabaviti njima malo detaljnije kako bi videli uzrok lošeg rada aplikacije. Jedan od dobrih načina rukovanjem izuzecima koje podržava OO način programiranja je Exception. Zamislete recmo da je neophodno da pozovete metodu koja proverava da li je unet e-mejl sa forme ispravno formatiran, tačnije da li sadrži jedan znak @, tačku i između njih znakovne nizove. (znakovni_niz)@ (znakovni_niz).(znakovni_niz)

Želite da obezbedite upis u bazu samo pravilno unete e-mail adrese. Ukoliko se to ne dogodi, potrebno je ispaliti izuzetak. Za početak napravićemo jedan skript koji prima samo parametar „mejl“ sa forme, proverava da li je on ispravan, i na osnovu toga ispisuje poruku:

```
class Controller{
```

```
	public function prihvatArgumenataSaForme(){
		$mejl = trim($_POST['mejl']);
		try{
			$this->proveraMejla($mejl);
			$this->unosMejlaUBazu($mejl);
			echo "Vas mejl je uspesno unet u bazu!";
		} catch (Exception $e) {
			echo $e->getMessage();
		}
	}

	private function proveraMejla($mejl){
		$ispravanMejl = true;
		$ispravanMejl = //kod za proveru ispravnosti mejla
		if(!$ispravanMejl)
			throw new Exception("Unet mejl je neispravan");
	}
	private function unosMejlaUBazu($mejl){
		// kod za unos mejla u bazu
	}
}
```

Glavni deo ovog koda je try-catch blok. Sve što je „sumnjivo“ da može da dovede do greške, stavlja se u bloku try. Nama je sumnjiva metoda proveraMejla jer može dovesti do greške i ispaliti izuzetak ako mejl nije lepo formatiran. Ukoliko dođe do greške, ona se obrađuje u catch bloku. Sve naredbe koje su u try bloku, a nalaze se ispod naredbe koja je ispalila izuzetak, preskaču se i direktno se odlazi u catch blok. Analizirajmo naš primer

Neko je sa forme uneo loše formatiran mejl. Mi smo takav mejl poslali na proveru pozivom metode proveraMejla. Ta metoda je proverila mejl i u zavisnosti od toga dodelila vrednst true ili false promenljivoj $ispravanMejl. Ukoliko je dodelila vrednost false, izvršava se ispaljivanje izuzetka. To se čini uz pomoć ključne reči throw new, zatim se doda klasa Exception koja kao argument prima string koji ujedno predstavlja objašnjenje o nastaloj grešci. Čim se naiđe na throw new Exception, ispaljuje se izuzetak i vrća na mesto odakle je pozvana metoda, tačnije na $this->proveraMejla($mejl); Kako je izuzetak ispaljen, a ta metoda se nalazi u try bloku, automatski se preskače sav kod koji je ispod nje u pomenutom bloku i odlazi u catch blok. Tamo sada postoji samo jedna naredba koja iz tog izuzetka izvlači poruku i prikazuje je na ekran. U našem slučaju na ekranu bi pisalo „Unet mejl je neispravan.“ Da smo nekim slučajem uneli ispravan mejl, izuzetak se ne bi ispalio, već bi kod nastavio normalno da se izvršava, prvo bi se pozvala metoda $this->unosMejlaUBazu($mejl); pa zatim ukoliko je uspešno unet mejl u bazu, prikazala bi se poruka "Vas mejl je uspesno unet u bazu!". U tom slučaju se catch blok nikada ne bi izvršio.

1.14 Korisne funkcije

PHP standardna biblioteka nam pruža dosta funkcija kojima možemo da rukujemo klasama i objektima. Recimo ukoliko želimo da proverimo da li je objekat instanciran, da li ima metode i koliko ih ima, kog je tipa i tako dalje. Navešćemo nekoliko funkcija koje se najčešće upotrebljavaju

`class_exists`

ova funkcija utvrđuje da li postoji kalsa sa kojom želimo da rukujemo. Prima obavezan parametar string, koji predstavlja naziv klase i vraća true/false, u zavisnosti da li klasa postoji ili ne. Recimo pre instanciranja objekta, želimo da utvrdimo da li klasa postoji. To bi uradili ovako:

```
if (class_exists('Pas')) {
    $pas = new Pas();
}
```

get_class_methods

funkcija koja vraća nazive metoda koje se nalaze u datoj klasu. Nazive vraća u vidu niza a parametar koji prima je naziv klase ili objekat date klase.

```
$metode = get_class_methods('Pas');
// ili
$metode = get_class_methods(new Pas());

foreach ($metode as $naziv) {
    echo "$naziv\n";  // ispisuje naziv metode jedan ispod drugog
}
```

get_class

vraća naziv klase za dati objekat. Prima jedan parametar, instancu nekog objekta i na osnovu njega u vidu stringa vraća naziv date klase:

```
$mojPas = new Pas();

echo get_class($mojPas); // na ekranu će ispisati Pas
```

is_subclass_of

ispituje da li za dati objekat, je data klasa jedna od roditeljskih klasa tog objekta. U zavisnosti od toga, vraća true ili false

```
$mojPas = new Pas();
$sSluzba = new SSluzba();

if (is_subclass_of($mojPas, 'Zivotinja')) {
  echo "da, objekat $mojPas nasleđuje klasu Zivotinja\n";
} else {
  echo "ne, objekat $mojPas ne nasleđuje klasu Zivotinja\n ";
}

if (is_subclass_of($sSluzba, 'Zivotinja')) {
  echo "da, objekat $sSluzba nasleđuje klasu Zivotinja\n";
} else {
```

```
  echo "ne, objekat $sSluzba ne nasleđuje klasu Zivotinja\n ";
}

Kao rezultat na ekranu bi se pojavilo

da, objekat $mojPas nasleđuje klasu Zivotinja
ne, objekat $sSluzba ne nasleđuje klasu Zivotinja
```

Još sličnih funkcija možete naći na http://www.php.net/manual/en/ref.classobj.php

Primeri iz prakse

ArrayList

Rešili smo da sami kreiramo klasu koja će simulirati listu u koju mogu da se dodaju razni objekti. Ukoliko je neko blizak sa Javom i C# - om, onda zna o čemu pričamo jer zapravo želimo da napravimo baš tu klasu i u php-u.

Ono što će nama trebati su metode,

add($obj) – dodavanje elementa $obj
remove($i) – uklanjanje elementa na $i-toj poziciji
removuFirst() – uklanjanje prvog elementa
removeLast() – uklanjanje poslenjeg elementa
getSize() – dobijanje broja objekata u listi
get($i) – vraća objekat na i-toj poziciji
isEmpty() – ispituje da li je lista prazna ili ne

Implementacija ove klase bi igledala ovako:

```
class ArrayList {
        private $niz;
        private $size;

        public function __construct() {
            $this->niz = array();
            $this->size=0;
        }

        public function add($parametar) {
            $this->niz[$this->size]=$parametar;
            $this->size++;
        }
        public function remove($index) {
```

```
            if(!$this->isIndexCorrect($index)) {
                return false;
            }
            if($index == 0) {
                $this->removeFirst();
            }else if($index == ($this->size)-1) {
                    $this->removeLast();
            }else {
                for($i=($index+1);$i<$this->getSize();$i++) {
                    $this->niz[$i-1] = $this->niz[$i];
                }
                $this->niz[$this->size-1]=null;
            }
            $this->size--;
        }
        public function removeFirst() {
            array_shift($this->niz);
        }
        public function removeLast() {
            array_pop($this->niz);
        }
        public function getSize() {
            return $this->size;
        }
        public function get($index) {
            if($this->isIndexCorrect($index)) {
                return $this->niz[$index];
            }
            return false;
        }
        public function isEmpty(){
            return $this->size == 0;
        }
        private function isIndexCorrect($index) {
            return !($index>$this->size-1 || $index<0);
        }
    }
```

Nakon kreiranja ove klase, jednostavno rukujemo elementima u listi:

```
$lista = new ArrayList();   // kreiranje objekta
$lista->add('neki string');   //dodavanje stringa u listu
$lista->add('neki drugi string'); //dodavanje drugog stringa u listu
$lista->add('neki treci string'); //dodavanje treceg stringa u listu
$lista->remove(2); // brisanje stringa ciji je index 2
echo $lista->getSize(); //prikaz broja elemenata u listi
```

Rukovanje tekstalnim fajlama

Često puta u programiranju je neophodno snimiti neki podatak u tekstualni fajl, ili ga učitati. Sada ćemo kreirati klasu FileManager koja će biti odgovorna upravo za manipulaciju tekstova u fajlu. Pomenuta klasa imaće sledeće metode

open($mode) –otvaranje konekcije ka fajlu (mode ukazuje upisivanje ili čitanje podataka)
close() – zatvaranje konektcije
write($tekst) – upisivanje u fajl
read() – čitanje iz fajla

```
class FileManager {

    private $file;
    private $connection;

    const READ = "rb";
    const WRITE = "wb";
    const APPEND = "ab";

    public function __construct($file) {
        $this->file = $file;
        $this->connection = false;
    }
    public function open($mode) {
        $this->connection = fopen($this->file, $mode);
        return $this->connection;
    }
    public function close() {
        return !fclose($this->connection);
    }
    public function write($text) {
        if(!$this->connection)
            return false;

        fwrite($this->connection, $text);
    }
    public function read() {
        if(!$this->connection)
            return false;

        while(!feof($this->connection)) {
            $content .= fgets($this->connection);
        }
        return $content;
    }
}
```

Kada smo kreirali klasu, možemo vrlo jednostavno upisivati i čitati podatke iz nekog tekstualnog fajla:

```
$fajl = new FileManager('test.txt'); // učitavanje fajla
$fajl->open(FileManager::WRITE); // otvaranje konekcije za upis
$fajl->write('Ovaj tekst se treba upisati u fajl !!!'); // unos teksta u fajl
$fajl->close(); // zatvaranje konekcije

$fajl = new FileManager('test.txt'); // učitavanje fajla
```

```
$fajl->open(FileManager::READ); // otvaranje konekcije za čitanje
$tekst = $fajl->read(); // čitanje sadržaja fajla
$fajl->close(); // zatvaranje konekcije
echo $tekst; // prikaz sadržaja na ekran
```

Rukovanje slikama

PHP biblioteka poseduje jako puno funkcija preko kojih može da se manipuliše slikama. Postoje funkcije za smanjivanje slika, uveličavanje, sečenje, kreiranje i tako dalje i tako dalje. Zaista broj ovih funkcija je veliki. Ponekad, ukoliko želite da uradite nešto sa slikom, nećete moći samo uz pomoć jedne funkcije, već kombinacijom nekoliko njih. Sada ćemo kreirati klasu koja ima najosnovnije elemente preko kojih možete da manipulišete slikom. Metode koje će biti implementirane su :

save() – snima trenutnu sliku
saveAs($novaPutanja) – snima sliku na novoj putanje
getType() – vraca tip slike (jpg, png,...)
getWidth() – vraca sirinu slike
getHeight() – vraca visinu slike
resizeToHeight($visina)– vrši promenu visine slike sa očuvanjem odnosa širina:visina
resizeToWidth($sirina) – vrši promenu širine slike sa očuvanjem odnosa širina:visina
resize($sirina,$visina) – vrši promenu širine i visine slike

```
class Image {

    private $path;
    private $image;

    public function __construct($path) {
        $this->path = $path;
        $this->setImage();
    }
    public function save() {
        $this->saveAs($this->path);
    }
    public function saveAs($newImagePath) {
        if($this->getType() == IMAGETYPE_GIF ) {
            imagegif($this->image,$newImagePath);
        } elseif($this->getType() == IMAGETYPE_PNG ) {
            imagepng($this->image,$newImagePath);
        } elseif($this->getType() == IMAGETYPE_WBMP) {
            imagewbmp($this->image, $newImagePath);
        }elseif($this->getType() == IMAGETYPE_JPEG) {
            imagejpeg($this->image,$newImagePath);
        }else {
            return false;
        }
    }
    public function getType() {
          list($width, $height, $type, $attr) = getimagesize($this-
>path);
```

```
        return $type;
    }
    public function getWidth() {
          list($width, $height, $type, $attr) = getimagesize($this->path);
        return $width;
    }
    public function getHeight() {
          list($width, $height, $type, $attr) = getimagesize($this->path);
        return $height;
    }

    public function resizeToHeight($height) {
        $ratio = $height / $this->getHeight();
        $width = $this->getWidth() * $ratio;
        $this->resize($width,$height);
    }
    public function resizeToWidth($width) {
        $ratio = $width / $this->getWidth();
        $height = $this->getHeight() * $ratio;
        $this->resize($width,$height);
    }

    public function resize($width,$height) {
        $new_image = imagecreatetruecolor($width, $height);
          imagecopyresampled($new_image, $this->image, 0, 0, 0, 0, $width, $height, $this->getWidth(), $this->getHeight());
        $this->image = $new_image;
    }

    private function setImage() {
        if($this->getType() == IMAGETYPE_GIF ) {
            $this->image = imagecreatefromgif($this->path);
        } elseif($this->getType() == IMAGETYPE_PNG ) {
            $this->image = imagecreatefrompng($this->path);
        } elseif($this->getType() == IMAGETYPE_BMP) {
            $this->image = imagecreatefromwbmp($this->path);
        }elseif($this->getType() == IMAGETYPE_JPEG) {
            $this->image = imagecreatefromjpeg($this->path);
        }else {
            return false;
        }
    }
}
```

Ovom klasom sada vrlo lako rukujemo :

```
$img = new Image('../img/plan_grada.jpg '); // učitavamo sliku
echo $img->getWidth(); // prikazujemo na ekran širinu slike
$img->resizeToWidth(200);//menjamo širinu na 200px i održavamo odnos širina:visina
$img->saveAs('../img/plan_grada_thumb.jpg '); // snimamo je pod drugim nazivom i tako pravimo kopiju
```

Nakon izvršenja ovog koda, stara slika će ostati ista, dok će nova imati širunu 200px. Da smo umesto `saveAs(nova putanja)` korisnili `save()`, samo bi snimili smanjenu sliku preko postojeće.

Rukovanje bazom podataka

Zamislimo situaciju da je potrebno napraviti aplikaciju koja će rukovati sa bazom podataka. Ono što nam je neophodno je korišćenje mysql ili mysqli biblioteke. Rešili smo da taj problem rešimo uz pomoć mysqli biblioteke. Naša klasa, koja je će predstavaljti konekciju ka bazi, sadrži sledeće metode:

open() – otvaranje konekcije
close() – zatvaranje konekcije
query($upit) – izvršavanje upita

pored toga, mogli bi smo da napravimo i metode commit(), rollback(), isOpen(), setAutoCommit($bool) i tako dalje, ali zbog složenosti zadržaćemo se samo na ove tri metode.

Naša klasa bi izgledala ovako:

```
class Connection {

    private $link;

    public function __construct() {
        $this->link = null;
    }

    public function close() {
        $this->link->close();
    }
    public function query($query) {
        return $this->link->query($query);
    }

    public function open() {
            $this->link = new mysqli(HOSTNAME, USERNAME, PASSWORD,
DB_NAME);
        if (!$this->link)
           return false;
         return true;
    }
```

Ukoliko sada želimo da dodamo nekog korisnika u bazu, za to možemo kreirati neki php faj, recimo kreiranjeKorisnika.php, jednostavno ćemo napisati ovako:

kreiranjeKorisnika.php

```
<?php
```

```
$con = new Connection();
$con->open();
$con->query("INSERT INTO ........");
$con->close();

?>
```

Sve će to super raditi, dok ne dobijemo zahtev da moramo celu aplikaciju da premestimo na drugi server. To i nije toliko strašno, ali zamislite da taj drugi server ne podržava biblioteku `mysqli`, već samo `mysql`. Onda bi morali da izmenimo ovaj kod u klasi `Connection`

To ćemo modifikovati ovako:

```
class Connection {

    private $link;

    public function __construct() {
        $this->link = null;
    }

    public function close() {
        mysql_close($this->link);
    }
    public function query($query) {
        return mysql_query($query, $this->link);
    }


    public function open() {
        $link = mysql_connect(HOSTNAME, USERNAME, PASSWORD);
        if (!$link)
           return false;
         $db_selected = mysql_select_db(DB_NAME, $link) ;
         if (!$db_selected) {
           return false;
         }
         $this->link = $link;
    }
```

Sada smo rešili i taj problem. Php fajl kreiranjeKorisnika.php nećemo ništa menjati, jer on poziva date metode, samo smo promenili implementaciju u tim metodama. Međutim, šta se dešava ako dobijemo obaveštenje da na serveru od sutra nije dostupna biblioteka mysql već se uvodi mysqli. To bi značili ponovno prepravljanje postojeće klase. Međutim, mi sada hoćemo da rešimo i taj problem i da ne menjamu uvek implementaciju klase kad god se stvari promene na serveru.

Razmotrimo problem. Potrebne su nam dve klase koje će različito biti implementirane u zavisnosti da li je mysql ili mysqli biblioteka. Pored toga, mi želimo da uvek koristimo iste nazive metoda za konekciju, za izvršavanje upita i za zatvaranje konekcije. Te potpise možemo izdvojiti u jedan interfejs. To bi izgledalo ovako:

```
interface IConnection{
     public function open();
```

```
      public function query($query);
      public function close();
}
```

Želimo da u programu uvek koristimo ove metode, a na koji način i kako će oni biti implemetirane, to nas ne zanima (ne zanima nas u fajlu `kreiranjeKorisnika.php`)

Pored toga, morali bi da kreiramo i dve klase, jednu za mysql a drugu za mysqli biblioteku, i obe klase bi implementirale ovaj interfers.

```
class MySQL_I_Connection implements IConnection{

    private $link;

    public function __construct() {
        $this->link = null;
    }

    public function close() {
        $this->link->close();
    }
    public function query($query) {
        return $this->link->query($query);
    }

    public function open() {
            $this->link = new mysqli(HOSTNAME, USERNAME, PASSWORD,
DB_NAME);
        if (!$this->link)
           return false;
         return true;
    }
}
```

i druga klasa

```
class MySQL_Connection implements IConnection {

    private $link;

    public function __construct() {
        $this->link = null;
    }

    public function close() {
        mysql_close($this->link);
    }
    public function query($query) {
        return mysql_query($query, $this->link);
    }


    public function open() {
        $link = mysql_connect(HOSTNAME, USERNAME, PASSWORD);
        if (!$link)
           return false;
```

```
            $db_selected = mysql_select_db(DB_NAME, $link) ;
            if (!$db_selected) {
              return false;
            }
            $this->link = $link;
        }
}
```

Sada se postavlja pitanje kako ćemo ovo iskoristiti. To radimo na sledeći način:

Potrebno je da kreiramo jedan inicijalni php fajl, koji će se uvek prvi pozivati u php skriptu. On će određivati koju klasu koristimo, da li MySQL_I_Connection ili MySQL_Connection. Nazovimo ga init.php

init.php

```
<?php

function getConnection(){
      return new MySQL_Connection();
}

?>
```

Fajl init.php sadrži samo jednu funkciju `getConnection()` koja vraća objekat tipa `MySQL_I_Connection` ili `MySQL_Connection` . Kod nas trenutno vraća objekat tipa `MySQL_Connection`.

Skript kreiranjeKorisnika.php malo ćemo izmeniti, i on će igledati ovako:

kreiranjeKorisnika.php

```
<?php

$con = getConnection();
$con->open();
$con->query("INSERT INTO ........");
$con->close();

?>
```

i to je sve !!!

Ukoliko opet dođe do nekog zahteva, da se pređe sa mysql na mysqli biblioteku, dovoljno je samo izmeniti u skripti init.php da funkcija getConnection() vraća new `MySQL_I_Connection()` i mi smo naš problem rešili, ostatak koda neće se menjati, pa će pri tome i kreiranjeKorisnika.php skript ostati isti. To je i glavna prednost OOP-a – **kod koji se lako održava** !!!