PHP5 – Új generáció (3. rész)

...avagy hogyan használjuk okosan az osztályokat és objektumokat PHP5-ben.

Az előző részben az objektumszemléletű fejlesztés igazi erősségét adó örökléssel, mint nagy témakörrel foglalkoztunk. Szó volt az ezekkel szorosan kapcsolatban álló ***elvont (abstract) osztályokról*** illetve ***felületekről (interface)*** és a bennük rejlő lehetőségekről, végezetül pedig megnéztünk néhány mágikus tagfüggvényt a teljesség igénye nélkül.
Most megnézzük, mire jók azok a bizonyos kivételek, miként másolhatjuk az objektumainkat értékeik szerint, illetve hogyan írhatunk elő objektumtípusokat paraméter gyanánt. Ne is húzzuk az időt, kezdjünk bele!

Kivételkezelés

PHP 5-ben is bevezették a *Java*, *C#* nyelvekből már jól ismert kivételeket. Ezek olyan felhasználói „hibaüzenetek", amelyeket a programkódban használhatjuk hibajelzések, különleges események kezelésére. Működésének lényege, hogy bizonyos feltétel teljesülése esetén mondjuk egy objektumból kivételt dobunk, amelyet azonban az objektum metódusát hívó programkód fog majd elkapni, lekezelni. Ez így homályos lehet, ezért nézzünk egy példát:

```
<?php
class Negyzet {
    private $oldal = 0;

    public function __construct($oldal) {
        $this->oldalHossztBeallit($oldal);
    }

    public function oldalHossztBeallit($ertek) {
        if (is_numeric($ertek)) {
            $this->oldal=$ertek;
        } else {
            throw new Exception("Értékadási hiba:
            ➥a megadott érték nem szám");
        }
    }

    public function terulete() {
        echo $this->oldal*$this->oldal;
    }
}
$negyzet1=new Negyzet(0);
try {
    $negyzet1->oldalHossztBeallit("asd");
    $negyzet1->terulete();
} catch (Exception $ex) {
    echo $ex->getMessage();
}
?>
```

A kulcs az `oldalHossztBeallit()` metódusban van. Jelen esetben, ha olyan értéket adunk meg, ami nem szám, akkor szeretnénk, ha erről értesítene bennünket az objektum, hogy a felhasználás pillanatában tudjuk, mi a hiba. A főprogramban a `try` blokk szolgál arra, hogy megpróbáljuk beállítani az oldalhosszt. Ez természetesen nem biztos, hogy sikerülni fog. A `catch` blokkban kezeljük le azt, ha esetleg a `try` blokk valamelyik parancsa nem sikerült, s az adott típusú kivétel keletkezett. Jelen esetben megpróbálunk karaktersorozatot adni értékül az oldalnak, ami nem fog sikerülni. Ekkor a `try` blokkban az utasítások végrehajtása megszakad, a futás átugrik a megfelelő `catch` blokkba (ahol az adott típusú kivételre várunk), és végrehajtja az ott található összes utasítást.
Egy `try` blokkot tetszőleges számú `catch` blokk követ. Mint már említettem is, abba a blokkba kerül a vezérlés a hiba esetén, amilyen típusú az objektum által dobott kivétel (jelenleg `Exception` típusú). Ez akkor lehet igazán hasznos, ha mi magunk is saját kivételtípusokat készítünk, és más más típusra máshogyan szeretnénk reagálni.
Természetesen egy blokkon belül több azonos kivételt előidéző utasítás is lehet (az oldalhossz beállítása mellé valami), azonban minden esetben csak egyetlen kivétel jön létre, hiszen utána azonnal a `catch` blokkban találjuk magunkat.
A módszer előnye az, hogy `if`-ek nélkül egyszerűen kezelhetjük a hibákat, ráadásul a használt objektum belső szerkezetének ismerete nélkül. Tegyük fel, hogy a négyzetnek beállítjuk még a színét is. Ha nem megfelelő színt adunk, akkor ott is kiírathatunk egy hibaüzenetet, s ilyenekből tetszőleges számú lehet. Ekkor azonban a felhasználás során alkalmazott `try-catch` szerkezet a világon semmit sem változik, csak végzi az egyszerű hibakezelési feladatot.
Természetesen nem szükséges minden esetben lekezelni a kivételt... nem kell tehát `try` blokkba foglalni. Ilyenkor, ha mégis bekövetkezik, végzetes hibával leáll a program futá-

sa, ezért célszerű mindig alkalmazni. Olyan eseteben hagyhatjuk el mégis, mint a fenti példa. Ha ugyanis a programból mindig ugyanazt a konstans számértéket állítjuk be, sohasem fog kivétel képződni. A dolog csak akkor érdekes, ha mondjuk a felhasználó által beolvasott értéket szeretnénk megadni oldalhosszként, amelyet nem ismerhetünk előre.

Típuselőírás, típusmeghatározás

A fentiekben már volt szó a `catch` blokkok kapcsán arról, hogy a kivétel, mint osztály típusával azonosítható egy ilyen kódrészlet. Az előző változatokhoz képest szokatlan a (típus változónév) formula, amely itt arra hivatott, hogy meghatározza, milyen típusú legyen az adott paraméter. Mivel a *PHP* egy gyengén típusos nyelv, ez az előírás csak azt erőlteti, hogy a paraméter objektum legyen, mely a típusként megadott nevű osztály egyik példánya. Ez az úgynevezett ***type hinting*** minden függvényre és metódusra alkalmazható.

```
function test(Negyzet $negyzet) {
        $negyzet->terulete();
}
```

A fenti kódrészlet az előző példa kódját felhasználva szemlélteti az esetet. A test függvény csak a `Negyzet` osztály példányait fogadja paraméterül. Nem lehet más osztály példánya, egyáltalában csak osztályokat fogad paraméterül, és null sem lehet az értéke.
A fenti lehetőséget minden olyan esetben haszonnal alkalmazhatjuk, ahol előre meghatározott objektumpéldányokon kell műveletet végeznünk (így garantált, hogy létezik a paraméter megadott metódusa, stb.).

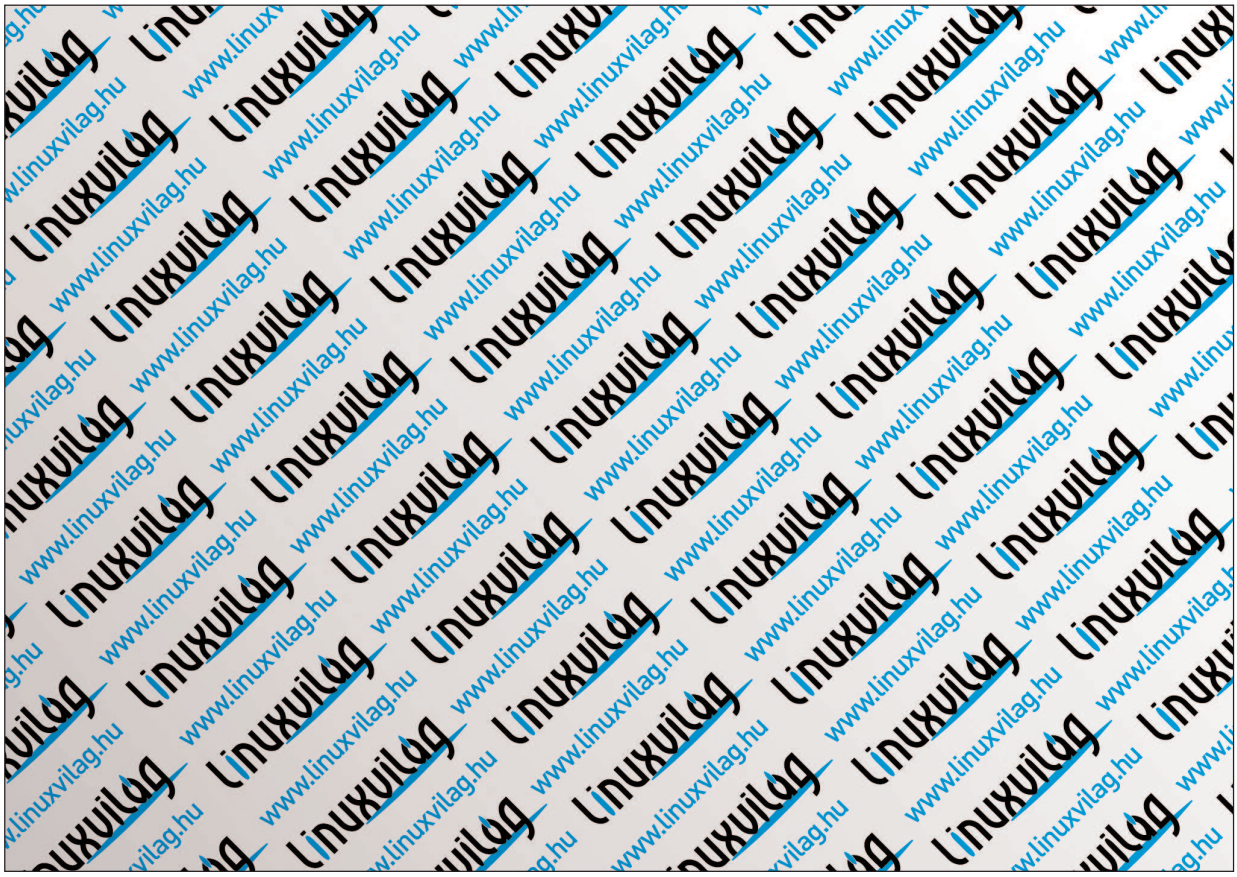
Objektumok másolása

Szó volt róla, hogy a *PHP 5* egyik legjelentősebb változtatása, hogy az objektumok átadása a régi érték szerinti megoldás helyett már cím szerinti. Ez azt jelenti, hogy `$a=$b` esetén mindkét változó ugyanarra a memóriaterületre fog mutatni (természetesen csak objektumok esetén), ezért ha az egyiknek bármilyen tulajdonsága megváltozik, valójában a másik is meg fog változni. Olyan ez, mintha két póráz lenne a kezünkben, amely ugyanazon kutyában végződik. Gyakran előfordulhat az a dolog, hogy mi valóban szeretnénk megduplázni úgy azt az objektumot, hogy valóban kettő legyen belőle, mert mindkettőn egymástól független műveleteket szeretnénk végezni, tehát kell nekünk két póráz, amely két különböző kutyában végződik. Még mindig a fenti négyzetes példánál maradva

```
$negyzet1 = new Negyzet(5);
$negyzet2 = clone $negyzet1;
$negyzet2->oldalHossztBeallit(3);

$negyzet1->terulete();
$negyzet2->terulete();
```

A fenti kódrészlet a futás után kiír egyszer 25-öt, majd 9-et, jól látszik tehát, hogy a két példány különböző. Ez olyan esetekben lehet hasznos, ahol van egy sok-sok tulajdonsá-



got tartalmazó objektumunk, be is vannak állítva ezek az adatok jól, és mi szeretnénk valamiért a rajta végzett műveletet „elágaztatni”, és az új objektumban csak bizonyos értékeket megváltoztatni.
Felvetődik itt azonban egy probléma, amelyet legjobb lesz, ha egy példán keresztül próbálok meg bemutatni:

```
<?php

class Ember {
    private $nev;
    private $szuletesiEv;

    public function __construct($nev,$szuletesiEv) {
        $this->setNev($nev);
        $this->setSzuletesiEv($szuletesiEv);
    }

    public function setSzuletesiEv($szuletesiEv) {
        $this->szuletesiEv=$szuletesiEv;
    }

    public function setNev($nev) {
        $this->nev=$nev;
    }

    public function getNev() {
        return $this->nev;
    }

    public function getSzuletesiEv() {
        return $this->szuletesiEv;
    }

}

class Anya extends Ember {
    private $gyermeke;

    public function setGyermeke(Ember $ember) {
        $this->gyermeke=$ember;
    }

    public function getGyermeke() {
        return $this->gyermeke;
    }
}

$ellie = new Anya(“Ellie”,1920);
$jockey = new Ember(“Jockey”,1950);
$ellie->setGyermeke($jockey);

$mary = clone $ellie;
$mary->setNev(“Mary”);
$ellie->getGyermeke()->setSzuletesiEv(1945);
echo $mary->getGyermeke()->getSzuletesiEv();
?>
```

A fenti kódrészlet az alábbi szituációt takarja: vannak `Emberek`, és vannak olyan speciális emberek, akik egyben `Anyák`, s van gyerekük, aki egy másik `Ember`. A példában létrehoztuk `Ellie`-t, és `Jockey`-t, mint embereket, majd azt mondtuk, hogy `Ellie` mama `Jockey` bácsi anyukája. Ez eddig rendben is volna.
Ezek után lemásoltuk `Ellie` anyukát `Mary` néven, majd `Ellie` fiának születési évét átállítottuk. Sajnálattal tapasztaltuk azonban az eredményt, hogy `Mary` fiának születési éve is megváltozott, amiből hajlamosak vagyunk arra következtetni, hogy a gyermekeik azonosak, pedig mi nem ezt szerettük volna.
A jelenség magyarázata az alábbi: az anyák gyermekükre referenciaként hivatkoznak, hisz látszik a kódból (`setGyermeke()`), hogy egyszerű egyenlőség operátorral „másoltuk” őket. Ez nem baj, ez az eredeti szándékunk, hisz így szép és helyes a megoldás. A baj abból adódik, hogy a `clone` hívás alapértelmezetten ún. ***sekély másolatot (shallow copy)*** készít, mindent egy az egyben átmásol az új objektumba, tehát referenciát is referenciába másol, így a két objektum $gyermeke tulajdonsága ugyanarra a példányra mutat. A kutyáknál maradva: hiába van nekünk két külön kutyánk a póráz végén, ha hozzájuk van kötözve egy harmadik közös kutya.
Nem mindig hasznos tehát, ha egy objektum egy az egyben másolódik (néha működésből adódóan sem célszerű – például nincs mindenre szükségünk). Erre megoldás lehet az, ha egy osztályban rögzítjük előre, hogy egy esetleges másolás esetén milyen értékek hogyan menjenek át a másolatba. Erre szolgál a `__clone()` nevű mágikus tagfüggvény, amely a `clone` utasítás kiadásakor hívódik meg az adott objektumra. Bármilyen utasítás, amit itt rögzítünk, végrehajtódik. A fenti példa javítása tehát: adjuk hozzá az Anya osztályhoz az alábbi metódust:

```
public function __clone() {
    $this->gyermeke = clone($this->gyermeke);
}
```

A hivatkozott referenciára is készítünk valódi másolatot. Fontos megjegyezni, hogy nem kell minden attribútumot átmásolnunk, azok a `__clone()` metódustól függetlenül automatikusan átmásolódnak, nekünk csak felül kell írnunk a nem kívánt értékeket.
További érdekesség, hogy ha megengedjük a `setGyermeke()` metódusban, hogy ne csak Ember, hanem Anya is lehessen gyerek (minek tulajdonsága referenciát tartalmaz), a fenti másolóalgoritmus akkor is hibátlanul működik, hisz a `__clone()` metódusban kiadott `clone` utasítás hatására a gyermekre (aki anya is lehet) szintén meghívódik annak `__clone()` metódusa, és ez így gyűrűzik egészen addig, amíg van hivatkozásunk a hivatkozott objektumban.
Már csak egy nagy témakör maradt hátra, nevezetesen a ***PHP 5*** új ***Reflection API***-ja, amely az osztályok, objektumok igen széles körű és részletes elemzésére szolgál – akár futásidőben is. A sorozat következő részében ezen a területen teszünk egy kisebb kirándulást.

Komáromi Zoltán