PHP 5 – Új generáció

...avagy hogyan használjuk okosan az osztályokat, és objektumokat PHP 5-ben.

Annak idején, amikor a *PHP 3*-at leváltotta a *PHP 4*, egy hasonló átállásnak lehettünk tanúi, mint most. A programnyelv fejlesztői nagyon odafigyeltek arra, hogy az új rendszerrel kompatibilis maradjon a kód, vagyis igyekeztek minimálisra csökkenteni azoknak a programoknak a számát, amelyek nem futnak az új *PHP*-vel. Tudjuk, hogy az átállás nem mindig volt sikeres, de legalább megpróbálták... És valljuk be, azért nem olyan rossz az eredmény, amit elértek. Az új változat tartalmazott néhány új megoldást, pár koncepcionális újítást, filozófiai jellegű helyretételt, amelyek hosszú távon igen jelentősek a programnyelv teljességére, összefogottságára nézve. A nagy változás azonban az volt, hogy megjelent a ***ZEND engine 1***, ami valójában az a virtuális gép, amely a *PHP* kódot futtatja. Ezzel vált igazából platform függetlenné a *PHP*.

Hasonló változásokról beszélhetünk mostanság is, a *PHP 5* megjelenésénél. Ami a programozást illeti, itt is olyan jellegű módosítások történtek, mint annak idején a *PHP 4*-ben. Két fő változat közben a függvény könyvtárak alakulása, bővülése természetesen folyamatos, ezzel nem is szükséges foglalkoznunk. Ebből a szempontból a *PHP 4* vége és a *PHP 5* eleje gyakorlatilag ugyanazt tudja, ám a *PHP 5* bevezetéséhez időzítettek jó néhány teljesen új függvényt, csakhogy könnyebb legyen az életünk. Ami igazán új, az a már említett virtuális gép a *ZEND*, amely már a 2-es változatszámot viseli. Ellátták néhány új képességgel, átalakult a memóriakezelés, gyorsabb lett, stabilabb lett, azonban mi programozók, ebből nem sokat látunk.

Ami viszont rendkívüli módon – mondhatni teljesen – átalakult, az az objektummodell. Ezzel a PHP talán leggyengébb bástyáját sikerült igencsak megerősíteni: olyan mértékben kibővült, hogy akár komoly dolgokra is használni lehet. A régi modell csak erős jóindulattal teljesítette azokat az alapkritériumokat, amelyek egy objektumközpontú nyelvnek sajátjai. Nem volt igazi egységbe zárás, s a többalakúság is csak úgy teljesült, ha programozóként betartottuk az ehhez szükséges szabályokat – szerencsére ez nincs többé.

Nem véletlen tehát, hogy a most induló cikksorozatunk is szinte kizárólag erről fog szólni. Egyrészt azért, hogy mindenki megismerje az új szolgáltatásokat, másrészt, hogy a tisztelt olvasó kedvet kapjon az objektumok okos, átgondolt használatához, amivel jelentősen megkönnyítheti saját dolgát – ha programozni támad kedve. Éppen ezért – a sorozat alcímének megfelelően – az alapoktól kezdve, a szintaktika mellett a szemlélet áttekintésével próbálom elővezetni mondandómat. Remélem az Olvasó is méltónak találja majd írásomat az ott említett jelzőre.

Osztályok és objektumok

Mint tudjuk, az osztályok olyan kerek, egész szerkezetek, amelyekben nem csak értékeket tárolhatunk, de azok segítségével különböző műveleteket végezhetünk, utasításokat hajthatunk végre. Olyan összetett típusok tehát, amelyekben nem csak adat, hanem logika is található.

A legfontosabb változás a *PHP 4*-hez képest, amelyet mindjárt az elején meg kell említsek, hogy az osztályok példányaira, az objektumokra ezentúl referenciaként hivatkozunk. Előzőleg, ha egy objektumot átadtunk bárhogyan a programban, az lemásolódott, és egy új példány jött létre, azon végeztük azután a műveleteket. *PHP 5*-ben, ha átadunk egy objektumot, például valamilyen függvénynek, akkor annak az hivatkozásként, referenciaként adódik át, ami a gyakorlatban azt jelenti, hogy az objektum csak egyszer szerepel a memóriában, és amikor valahonnan használjuk, valójában arra a memóriaterületre hivatkozunk, ahol az elhelyezkedik. Ha tehát átmásoljuk az objektumunkat egy másik „változóba”, az igazából ugyanaz a változó marad, amit ezután két módon is elérhetünk.

Új osztály létrehozása

Új osztályt még mindig a megszokott módon kell létrehozni, azaz a `class` kulcsszóval, amit az osztály neve követ, majd kapcsos zárójelek között az osztály definíciója. Lássunk erre egy példát:

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

      public function oldalHossztBeallit($ertek) {
        $this->oldal=$ertek;
        $this->terulet=$erte*$ertek;
      }

      public function terulete() {
        echo $this->terulet;
```

```
        }
}
$negyzet = new Negyzet();
$negyzet->oldalHosszBeallit(7);
$negyzet->terulete();
?>
```

Ez majdnem olyan, mintha *PHP 4*-ben íródott volna. Az osztály a `new` kulcsszó segítségével példányosítható. A `$this` különleges változó arra szolgál, hogy az osztályon belül a saját tulajdonságokra, tagfüggvényekre hivatkozhassunk. Az osztály egy példánya egy konkrét elemre (esetünkben a négyzetek közül egy konkrét négyzetre) vonatkozik, ebben eddig nincs is semmi különleges.
Ha azonban jobban megnézzük a kódot, ott van az a bizonyos `public`, illetve `private` kulcsszó. Mi is ez valójában?

Nyilvánosság – az egységbe zárás kulcsa

A *PHP 5*-ben is bevezették a más objektumközpontú nyelvekben már régen használatban lévő adattulajdonságokat, amelyek arra hivatottak, hogy segítségükkel elrejtsük az osztályban szereplő tagfüggvényeket, osztályváltozókat. Erre azért van szükség, hogy az osztály által nyújtott szolgáltatásokhoz, erőforrásokhoz csak az osztályban megvalósított, szabványos felületeken keresztül férhessünk hozzá, s ne tehessünk kerülőutakat, melyek használata rövid időn belül átláthatatlanná teszi kódunkat. A fenti példa alapján képzeljük el, hogy a négyzet oldalát nem az erre írt tagfüggvényen keresztül állítjuk be, hanem közvetlenül, az $oldal osztályváltozón keresztül. Ekkor a négyzet területe természetesen nem számítódik ki, nekünk kell azt megtenni, s ha netán elfelejtenénk, az osztályunk már is helytelen működést produkál. A megoldás az, ha megtiltjuk, hogy az $oldal változóhoz közvetlenül is hozzá lehessen férni (ezt jelzi a `private` kulcsszó).
Ennek megoldására a klasszikus séma szerint három tulajdonság áll rendelkezésünkre, amelyet az osztályváltozókra és tagfüggvényekre egyaránt alkalmazhatunk.

- `public`: Ez az alapértelmezett tulajdonság. Ha nem adunk meg semmit, akkor automatikusan nyilvános lesz az osztályelem. A nyilvános elemek elérhetők az osztály példányán keresztül a -> operátor segítségével – minden korlátozás nélkül. Azokat a tagfüggvényeket tehát, amelyeket kifejezetten arra célra készítünk, hogy a program egyéb részeiből használjuk, nyilvánosnak kell deklarálni.

- `private`: Az ilyen módosítóval ellátott elemek csak az adott osztály belsejében lévő programkódból érhetők el.

- `protected`: A védett tulajdonságot majd csak később fogjuk jól megérteni. Az így jelölt elemek csak azokból az osztályokból érhetők el, amelyek az adott osztályból öröklődnek (és természetesen önmagából is). Ez valójában arra szolgál, hogy azért ne zárjuk ki teljesen az adott változót a további műveletből. Ha egy osztályt továbbfejlesztünk az öröklés módszerét választva, igen nagy korlátokba ütközhetnénk, ha nem érhetnénk el az ősosztály meghatározott változóit.

Ezeket az adattulajdonságokat helyesen alkalmazva elérhetjük, hogy az általunk írt osztályokat csak olyan módon lehessen használni, ahogyan azt mi megálmodtuk. A pontosan meghatározott használhatóság pedig lehetővé teszi számunkra, hogy építsünk rá.

Konstruktorok és destruktorok

Ha megnézzük, a fenti példában az osztály használatát, láthatjuk, hogy a példányosítás során jó volna, ha nem kellene külön beállítanunk a négyzet oldalát, feltételezhető, hogy ha egy négyzetet akarunk „csinálni”, annak van valamilyen hosszú oldala. Ilyen helyzetek megoldására találták ki a konstruktorokat. Ezek valójában különleges tagfüggvények. Automatikusan meghívódnak, amikor a `new` kulcsszó segítségével létrehozunk egy példányt, ezáltal bizonyos kezdeti értékek beállítására kiválóan alkalmasak. *PHP 4*-ben azok a metódusokat tekintette a nyelv konstruktoroknak, amelyek azonosak voltak az osztály nevével. Ez mostanra megváltozott. A konstruktorokat minden körülmények között `__consturct` névvel kell illetni. Lássunk erre egy példát, egészítsük ki a fenti osztályt az alábbi tagfüggvénnyel:

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

Most pedig példányosítsuk az osztályt úgy, hogy egyből át is adjuk az oldal hosszát:

```
$negyzet = new Negyzet(7);
```

Hasonló módon kell értelmezni a destruktorokat is, amelyek akkor hívódnak meg, amikor az osztály egy példánya megszűnik létezni. Általában arra szokták őket használni, hogy az osztály által használt erőforrásokat (adatbáziskapcsolat, fájlleíró) felszabadítsák. Destruktorok nem voltak a *PHP 4*-ben, csak most kerültek bevezetésre. Egészítsük ki a példaosztályunkat egy destruktor tagfüggvénnyel:

```
public function __destruct() {
    echo "A négyzet oldalhossza elhalálozásának
        ➥időpontjában: $this->oldal"
}
```

Ha lefuttatjuk a programunkat, akkor a program végeztével a PHP automatikusan felszabadítja a változókat, tehát minden példányra meg fog hívódni a destruktor, amely kilistázza nekünk, hogy mekkora volt a négyzet a futás végén. (Ugyanez az eredmény történik, ha kézzel, az `unset()` függvénnyel szabadítjuk fel az objektumunkat.)

Statikus változók, tagfüggvények

A statikus elemek olyan résztvevői az osztályoknak, amelyek nem élnek együtt azok példányaival, sokkal inkább kötődnek az osztályhoz, magához. Statikus elemek eléréséhez nincs is szükségünk az objektumokra, az osztályra hivatkozva érhetjük el azokat. A statikus elemeket a `static` kulcsszóval vezethetjük be, amelynek minden esetben az adattulajdonságokat jelölő kulcsszó után kell következnie.

Hogy mindezt megértsük, egészítsük ki a fenti osztályunkat az alábbi tagfüggvénnyel:

```
public static leiras() {
        echo „Én egy olyan osztály vagyok, aki
              ➥négyzeteket képvisel”;
}
```

A fenti függvénynek ugyan sok értelme nincs, de jól szemlélteti a lényeget. A statikus elemekre a `Scope (::)` operátorral hivatkozhatunk. Lássuk:

```
Negyzet::leiras();
```

Mint látható, anélkül, hogy példányosítottuk volna az osztályt, leírást kérhetünk róla. Jól látható az is, hogy az ilyen tagfüggvények nem kötődnek szorosan az osztályhoz, nem használják ki azt, hogy az osztályok adatokat és logikát tartalmaznak. Egy statikus metódus csupán „logika”, nem köthető az osztály adataihoz. Ezek után szinte természetes, hogy nem is hivatkozhatunk benne hagyományos osztályváltozókra, tehát sehol nem fordulhat elő benne a `$this` kulcsszó.
Ha már a hivatkozásoknál tartunk: megszokhattuk, hogy a `$this` változón keresztül érhetjük el többek között az osztály saját tagfüggvényeit. Ez statikus esetben nem működik, mivel ott nem is beszélhetünk példányról, ezért itt a `self` kulcsszó használata a megoldás. Ha például a konstruktorban ki szeretnénk írni, hogy mely osztályról is van szó, akkor beletehetjük a

```
self::leiras();
```

sort, amely meghozza a kívánt eredményt.
Az osztályváltozók esetén is ugyanaz a helyzet, mint a tagfüggvényeknél. Ők is lehetnek statikusak, ekkor hasonló tulajdonságok érvényesek rájuk is. Logikailag azonban némiképp bonyolítható a helyzet. Ha statikus változókat használunk, akkor nem nehéz kitalálni, hogy ezek a változók az osztály minden példányából ugyanúgy, egyediként látszanak. Ha tehát az egyik példány megváltoztatja egy statikus változó értékét, azt utána minden példány „érzékelni” fogja. Példaként valósítsunk meg a négyzeteket képviselő osztályunkban példányszámlálást, amelynek segítségével megmondható, hogy adott pillanatban hány négyzetpéldány van jelen a programban. Ehhez az alábbi módon alakítsuk át az osztályunkat:

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

    private static $peldanyszam = 0;

    public function __construct($oldal) {
        self::$peldanyszam++;
        $this->oldalHossztBeallit($oldal);
    }

    public static function aPeldanyokSzama() {
        echo  self::$peldanyszam;
    }

    public function oldalHossztBeallit($ertek) {
        $this->oldal=$ertek;
        $this->terulet=$ertek*$ertek;
    }

    public function terulete() {
        echo $this->terulet;
    }

    public function __destruct() {
        self::$peldanyszam–;
    }
}
$negyzet1 = new Negyzet(7);
$negyzet2 = new Negyzet(2);
$negyzet3 = new Negyzet(9);
unset($negyzet1);   //elpusztitom a negyzet1
                      ➥peldanyt
Negyzet::aPeldanyokSzama();
?>
```

A kód futáskor kiírja, hogy két darab példány van jelen épp az adott `Négyzet` osztályból. Ha ugyanis létrejön egy új, meghívódik a konstruktor, ami átállítja a számlálót, ami mint tudjuk, minden „példányt” érint. (Azért helytelen kicsit a kifejezés, mert a statikus változók a statikus tagfüggvényekhez hasonlóan igazából nem kötődnek az osztály példányaihoz – csak annyiban, hogy ha nem statikus tagfüggvényben hivatkozunk rájuk, akkor minden példány „ugyanazt” az értéket látja.)
A példában statikus tagfüggvénnyel fértünk hozzá a hasonló tulajdonságú változóhoz, mert egy „globális”, példányfüggetlen szolgáltatásra volt szükségünk, de ez nem feltétel. Nem statikus metódusokból is módosíthatjuk, lekérhetjük ezeket a változókat, sőt, ekkor válik érdekessé a dolog. (Valójában ez történt akkor is, amikor a konstruktorból, destruktorból növeltük ill. csökkentettük az osztálytulajdonság értékét, de ez nem mindig tudatosul a példa tanulmányozásakor, ezért hívtam fel rá a figyelmet.)
Nagyon fontos, hogy megértsük a statikus változók és tagfüggvények szerepét, jelentőségét, mert ezzel egy halomnyi „bűvésztrükköt” tudunk később megvalósítani – olyanokat, amelyektől tényleg „szép” lesz az, amit csinálunk.

Állandók

A ***PHP 5*** lehetővé teszi állandók (konstansok) bevezetését osztályváltozók gyanánt. Ezek nagyon közeli rokonságban állnak a fentebb emlegetett statikus változókkal. Egyik fontos különbség, hogy értékük nem változtatható meg, az egyedüli értékadás a deklaráció során történik. A bevezetés a `const` kulcsszóval történik, s fontos megjegyezni, hogy nem használjuk a `$` változóelőtagot sem a deklarációnál, sem később az érték kiolvasásánál!
A változó értékét szintén a `self` előtaggal és a `scope` operátor használatával kaphatjuk meg, s hasonlóan a statikus testvéreikhez, ezek sem érhetők el az objektumpéldányokból, csakis az osztályon keresztül.

```
<?php
class ProbaOsztaly {
        public const allando = 'barmi';

        public function konstansErteke() {
                echo self::allando;
        }
}
echo ProbaOsztaly::allando;       //ily módon is
                                    ➥elérhetjük
$osztaly = new ProbaOsztaly();
$osztaly->konstansErteke();       //ily módon is
                                    ➥elérhetjük
?>
```

Osztályok összehasonlítása

Az esetek jelentős részében szükségünk lehet arra, hogy megmondjuk két objektumról, hogy azok egyenlőek-e. A helyzet némiképp bonyolódott a *PHP 4*-hez képest, ugyanis ott csak a azonosság (===) operátort használtuk két objektum összehasonlítására. Két objektum mindig két külön memóriaterületet, azaz két külön „változót” jelentett. *PHP 5*-ben azonban bevezették, hogy az objektumokra referenciaként hivatkozunk. Ez máris szükségessé tette az összehasonlítás átalakítását: egyenlőséget és azonosságot egyaránt vizsgálhatunk

Az objektumok egyenlőségét az „összehasonlító” (==) operátorral ellenőrizhetjük. Ekkor a két objektum megegyezik, ha ugyanazon tulajdonságai ugyanazokat az értékeket tartalmazták, és ugyanannak az osztálynak voltak példányai (Ez a *PHP 4*-es megegyezőség feltétele is). Mivel azonban itt referenciákkal dolgozunk, szükségünk lehet szigorúbb ellenőrzésre is, ha azt akarjuk tudni, hogy két hivatkozás ugyanarra az objektumra mutat-e. Ehhez használjuk a már említett azonosság operátort. Két objektum akkor azonos, ha az adott osztály ugyanazon példányaira hivatkoznak mindketten. Alapesetben, ha egy objektumot átmásolok egy másikba, a két objektum „azonos” lesz. Ez jelentősen bonyolítja az esetleges „valódi” másolást, de a későbbiek folyamán megtanulhatjuk, hogy hogyan legyünk úrrá a problémán. A fentiek alkalmazásával egy sor fejlett osztályt készíthetünk az egyszerűbb problémák intelligens elhárítására. Fontos, hogy feleslegesen ne alkalmazzunk objektumokat, mert azzal csak elbonyolítjuk az egyszerű feladatot. Gondosan válasszuk meg, hogy mikor és milyen osztályokat készítünk. Nos, a cikknek ugyan végére értünk, de a mondanivalónknak messze nem. Hátra van még például az öröklődés – mint nagy témakör, az elvont osztályok, felületek létrehozása, a különleges tagfüggvények használata. Ezekkel és más egyéb érdekességékkel foglalkozunk cikksorozatunk következő epizódjában.

Komáromi Zoltán
(komi@kiskapu.hu)
23 éves, a BME hallgatója, mellette PHP-programozóként dolgozik. Kedvenc területe a multimédia.