PHP5 – Új generáció (2. rész)

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

Cikksorozatom előző részében képet kaphattunk arról, hogy valójában mik is azok az objektumok, milyen tulajdonságaik, *PHP* vonatkozású különlegességeik vannak, illetve néhány példaprogramon keresztül megismerkedhettünk a konkrét használatukkal is. Ebben a részben központi szerepet kap az objektumközpontúság savát-borsát adó öröklődés, az ezzel kapcsolatos elvont (***abstract***) osztályok és felületek (***interface***) létrehozása, alkalmazása, valamint egy-két különleges tagfüggvény használata.

Vágjunk bele – mi is az az öröklődés

Az objektumközpontú programozás egyik ismérve a nagyfokú újrahasznosíthatóság. Ezt egyrészt annak köszönheti, hogy ezek a jól beburkolt, jól felépített objektumok komponensekként viselkednek, remekül lehet velük LEGO-zni. Másrészt ezeket az objektumokat egymással rokoni kapcsolatba állíthatjuk. A gyakorlatban ezt hívják öröklődésnek. Ha egy objektum egy másik (szülő)objektumtól örököl (gyermekobjektummá válik), akkor megkapja annak minden tulajdonságát és tagfüggvényét – a láthatóság által megfogalmazott feltételek mellett természetesen.
Az örökölt tagfüggvények teljes értékűen használhatók, felülírhatók, átalakíthatók, s új tagfüggvényeket adhatunk az osztálynak, egyszóval programozóként igen nagy szabadságot élvezünk, s mindemellett az ős összes képességét kihasználhatjuk. Lássunk erre egy példát, vegyük az előző epizódban bemutatott négyzet osztályt, kissé átalakítva

```
class Negyzet {
    protected $oldal = 0;

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

    public function terulete() {
        echo $this->oldal*$this->oldal;
    }
}
```

Közben a programunkban szeretnénk speciális négyzetekkel, rombuszokkal foglalkozni. Tudjuk, hogy a négyzetek és a rombuszok hasonlóak abban, hogy minden oldaluk egyenlő, de a rombusz esetében fontos az oldalak által bezárt szög, s a területe is másként számítandó. Használjuk ki a hasonlóságokat, és csak a különbségeket valósítsuk meg.

```
class Rombusz extends Negyzet {

    protected $szog = 0;

    public function szogetBeallit($szog) {
        $this->szog=$szog;
    }

    public function terulete() {
        echo $this->oldal*sin($this->szog)*
            ➥$this->oldal;
    }
}
```

Nos, ebben a példában felülírtuk a területszámító algoritmust, és az új tagfüggvény hozzáadásával elintéztük az oldalak által bezárt szög kezelését. Ha jól megnézzük, megspóroltuk az oldalak kezeléséért felelős függvények megírását, az osztályunk mégis teljes értékű. Fontos tény továbbá, hogy megmaradt az eredeti négyzet osztályunk, s nincs a kódunkban ismétlés.
Az így megspórolt munka természetesen nem túl sok, de ez a példa igen egyszerű. Említést érdemel még a változók előtti `protected` módosító: Most már láthatjuk, hogy mire jó: az ilyen változók elérhetők az örökös osztály belsejéből, de csak onnan.

A konstruktorok, destruktorok, és az öröklődés

PHP 5-ben ha az ősosztályunknak volt egy konstruktora, majd az abból örököltettünk másik osztály esetében nem határoztunk meg ilyen tagfüggvényt, akkor a példányosítás során az ősosztály konstruktora automatikusan meghívódik, lefut, elvégzi a szükséges lépéseket.
Ha azonban az örököltetett osztály is kapott konstruktort, akkor az ősosztály konstruktorának meghívásáról magunknak kell gondoskodnunk (ha szükséges). Ez azért van így, mert ellenkező esetben jelentősen meg volna kötve a kezünk a származtatásokat illetően. Lássunk erre is egy példát, módosítsuk az ősosztályt. Az előző cikkben is ismertetett módszer szerint adjunk hozzá egy konstruktort, amely beállítja az oldalhosszt.

```
class Negyzet {
    protected $oldal = 0;

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

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

    public function terulete() {
        echo $this->oldal*$this->oldal;
    }
}
```

Ilyenkor ugye a példányosítás után nem kell bíbelődni az oldalhossz beállításával, elintézhetjük a dolgot egy lépésben is. Na de mi van ilyenkor az előbbiekben bemutatott örökössel? Szeretnénk, ha az is egy lépésben elvégezné ezeket a módosításokat, de mint tudjuk, az oldalhossz kezelésével nem is foglalkoztunk, az az ősosztály feladata. Nézzük!

```
class Rombusz extends Negyzet {

    protected $szog = 0;

    public function __construct($oldal,$szog) {
        parent::__construct($oldal);
        $this->szogetBeallit($szog);
    }

    public function szogetBeallit($szog) {
        $this->szog=$szog;
    }

    public function terulete() {
        echo $this->oldal*sin($this->szog)*$this->
        ➥oldal;
    }
}
```

Látható, hogy az oldalkezelés továbbra is az ősé maradt. Mi csupán annyit tettük, hogy megkértük az örökös belsejéből, hogy foglalkozzon az ő érdekeltségébe tartozó adatokkal. Erre szolgál az a bizonyos `parent` előtag a `scope (::)` operátorral: az ősosztály függvényeire hivatkozhatunk vele. Ez természetesen csak akkor érdekes, ha felülírunk egy függvényt, ugyanis ha ezt nem tesszük, a `$this->függvénynév()` módon elérhetjük, mint az osztály saját tagfüggvényét. Ha azonban az öröklés során felülírjuk, és úgy szeretnénk hivatkozni, az előbbi módszerrel egy végtelen rekurzív függvényt kapunk, ami bizonyos, hogy senkinek sem jó.
A `parent::__construct()` hívás helyett jelen esetben a `$this->oldalHossztBeállít()` tagfüggvényt is használhattuk volna, ám úgy logikailag összefolyna a két objektum. Ez most még egy sokadrangú döntés, amely bonyolultabb esetekben azonban életet menthet.

Függvénytúlterhelés (overloading)

Számos – főként erősen típusos nyelvekben – létezik ez a fogalom. Ez nem is annyira az objektumokhoz kötődik, hanem úgy általánosságban létezik, ám most az öröklődés és a felülírás kapcsán érdemes néhány szót ejteni róla a félreértések elkerülése végett.
A függvénytúlterhelés azt jelenti, hogy több ugyanolyan nevű, de más paramétereket fogadó (esetleg más visszatérési típussal rendelkező) függvényt is definiálhatunk, s a meghívás során az a függvény hajtódik végre, melynek paraméterei (száma, típusa) illeszkednek a hívó paraméterekre. Ezzel lehet megoldani, hogy egy hasonló funkciójú függvényt különböző típusú és számú esetben is alkalmazni lehessen.
A *PHP*-ben ez mindkét okból szükségtelen. Mint tudjuk a *PHP* egy gyengén típusos nyelv, egy változó (paraméter) értéke lehet egész, lebegőpontos, karaktersorozat, tömb, akármi. Így tehát mindegy, hogy az adott paraméter gyanánt milyen típust adunk át.
A másik ok a paraméterek száma volt. Ez a lehetőség azért válik feleslegessé, mivel megadhatunk függvényparaméterként alapértelmezett értékeket a meghatározásban. A *PHP* tudja, hogy ha a meghívás során nem adunk át paramétert, akkor behelyettesíti a definícióban megadott alapértelmezett értéket.
Ennek folyományaként a *PHP*-ben sehol sem engedélyezett a függvénytúlterhelés, ne is keressük.

Tagfüggvény felülírásának megakadályozása

Számos esetben előfordulhat, hogy ki szeretnénk kötni néhány metódus számára, hogy azokat az örökösök bizony ne írhassák felül, ne változtathassák meg az algoritmust. Erre olyankor lehet szükség, ha több programozó által használt ősosztályt készítünk, és szeretnénk, ha a mi szabályaink szerint kódolnának – mert az úgy egységes, ellenőrizhető, konzisztens, és így tovább.
A problémára a `final` módosító kínál megoldást, amelyet tagfüggvények definíciói előtt használhatunk.

```
class OsOsztaly {
    final public function teszt() {
        echo 'Ősosztály teszt() metódusa lefutott';
    }
}

class GyermekOsztaly {
    public function teszt() {
        echo 'felülírt teszt() metódus lefutott';
    }
}
```

Ha ilyet szeretnénk csinálni, programunk futása végzetes hibával megszakad.

Elvont osztályok

Az eddigiekben tárgyalt objektumaink, még ha rokonságba is állíthatók egymással, meglehetősen szétszórt szerkezetet alkothatnak, s ennek csakis a programozó szabhat határt. Mint tudjuk, ez messze nem elégséges feltétel. Sokszor szükség lenne arra, hogy egy jól átgondolt hierarchiát építsünk, s a fejlesztés során, mint karácsonyfáról a szaloncukrot, csak

leakasszunk egyet a megfelelő helyről. Ezen szabályozott öröklés, hierarchia felépítését segítik az elvont osztályok és a felületek, nézzük most ez előbbit.

PHP 5-ben lehetőségünk van elvonatkoztatott osztályok, elvont tagfüggvények létrehozására. Ezek általában olyan tagfüggvények, amelyeket az adott osztály nem tud megvalósítani, mondjuk mert nem ismeri a megoldás miként jét, de azt pontosan tudja, hogy ilyen szolgáltatásra szükség lesz majd valamikor, és azt is tudja, hogy a gyermekosztályok már képesek lesznek a függvény megvalósítására.

Az absztrakt osztályoknak tehát csak a definíciója ismert, nincs is lehetőségünk a konkrét algoritmus megvalósítására az adott osztályon belül.

Az az osztály, amely majd megvalósítja az absztrakt metódust, legfeljebb olyan erős láthatósági paraméterrel rendelkezhet, mint maga az absztrakt metódus. Ha tehát ez a bizonyos elvont tagfüggvény `protected` besorolású, a megvalósító gyermekosztályban csak `protected`, vagy `public` lehet, `private` nem. Ennek az az oka, hogy az elvonatkoztatott osztályt készítő programozónak valószínűleg jó oka volt arra, hogy az adott láthatósági paramétert választotta, s ha ezt szűkítenénk az öröklés során, a további öröklések folyamán megváltozna a tagfüggvény jellege – esetleg teljesen el is tűnne.

Fontos megjegyezni, hogy ha egy osztálynak van legalább egy elvont tagfüggvénye, akkor az osztálynak is elvontnak kell lennie, továbbá az ilyen osztályok nem példányosíthatók, csak a gyermekosztályaik.

Az érthetőség kedvéért íme egy összetett példa: A geometriánál maradva szeretnénk objektumokkal modellezni a szabályos sokszögeket, s elég egyértelmű, hogy a valóságban ezek egy igen egyszerű hierarchiába szervezhetők, próbáljuk ki a programunkban megalkotott világunkban is – az eddigi példáktól teljesen függetlenül!

```
abstract class SzabalyosSokszog {
    protected oldalakSzama = 0;
    protected oldalHossz = 0;

    public function
        ➥__construct($oldalHossz,$oldalakSzama) {
        $this->oldalHossztBeallit($oldalHossz);
        $this->oldalakSzamatBeallit($oldalakSzama);
    }

    public function oldalHossztBeallit($oldalHossz)
    ➥{
        $this->oldalHossz =$oldalHossz;
    }

    public function oldalakSzamatBeallit
    ➥($oldalakSzama) {
        $this->oldalakSzama=$oldalakSzama;
    }

    public function oldalakAltalBezartSzog() {
        return 360/$this->oldalakSzama;
    }

    abstract public function terulet();
}
```

```
class NegySzog extends SzabalyosSokszog {
    public function __construct($oldalHossz) {
        parent::__construct($oldalHossz,4);
    }

    public function terulet() {
        return $this->oldalHossz*$this->oldalHossz;
    }
}


class HatSzog extends SzabalyosSokszog {
    public function __construct($oldalHossz) {
        parent::__construct($oldalHossz,6);
    }

    public function terulet() {
        return 3*$this->oldalHossz*$this->
        ➥oldalHossz*sin(60);
    }
}
```

Kicsit hosszúra sikeredet, de mint látszik, annál egyszerűbb. Íme a `SzabalyosSokszog` osztályunk. Jól látható, hogy vannak olyan szolgáltatások, amelyek minden gyermekre ugyanúgy vonatkoznak. Ezek az oldalhossz beállítása, oldalak számának beállítása, oldalak által bezárt szög kiszámítása. Ez minden sokszög esetében ugyanaz.

De ott van a terület, melynek kiszámítási módját az alaposztály nem ismeri, de tud róla, hogy minden szabályos sokszögnek van területe. Ekkor ezt megjelöli, de nem valósítja meg.

Felületek

A fenti elvont osztályos példa remek akkor, ha olyan tagfüggvényeket szeretnénk készíteni, amit nem feltétlenül kötelező minden örökösnek megvalósítani. Felmerülhet a kérdés, hogy ezeken a gyermekosztályokon valamilyen közös műveleteket végezzünk, mondjuk mindnek kiszámítsuk a területét. Ilyenkor elengedhetetlen az, hogy minden gyermekosztály rendelkezzen a `terulet()` tagfüggvénnyel, különben programhibával le fog állni a futás. A megoldás a felületek alkalmazása.

A felületek olyan osztálydefiníciók, amelyek tartalmazzák, hogy az őket megvalósító osztályoknak pontosan milyen tagfüggvényeket kell megvalósítaniuk. Hasonló a függvénydefinícióhoz, csak ez az osztályokra vonatkozik. A felületek az absztrakt osztályokkal ellentétben nem tartalmazhatnak semmilyen megvalósítást, csak az osztály minimális „tervét”. Nem tiltott, hogy a megvalósító osztály az előírtnál több tagfüggvényt tartalmazzon, ám ha nem készít el minden megadott metódust, a program végzetes hibával leáll. Egy ilyen felület minden metódusa nyilvános (`public`) kell legyen – ez a felületek természetéből adódik.

Felületeket tényleg csak akkor használjunk, ha minden egyes metódusra szükségünk van. Így az osztályaink egymásnak megfelelők (`compatible`) lesznek. Sok esetben nincs szükség az ilyesmire, azon esetekben inkább az elvont osztályokat használjuk. Nézzünk egy példát ismét a `SzabalyosSokszog` osztálytól függetlenül.

```
interface SikidomTulajdonsagok {
    public function terulet();
    public function kerulet();
    public function oldalakSzama();
    public function tengelyesenSzimmetrikus();
    public function kozeppontosanSzimmetrikus();
}

class Teglalap implements SikidomTulajdonsagok {
    //implementálni kell minden tagfüggvényt, de
    //ezen túl készíthetünk konstruktort, oldalbe-
    //állítót,stb. Az a későbbi feldolgozást
    // nem érinti
}
```

Előfordulhat, hogy több felületnek is meg kell felelnünk, ekkor az `impements` kulcsszó után vesszővel elválasztva kell megadni az egyes felületek nevét, amit megvalósítunk.
A felületek megvalósítása természetesen nem zárja ki azt, hogy az osztályunk örököljön is.
Módosítsuk a `SzabalyosSokszog` példánkat úgy, hogy kötelező legyen a `terulet()` tagfüggvény használata.

```
interface Teruletes {
    public function terulet();
}

abstract class SzabalyosSokszog implements
➥Teruletes {
    protected oldalakSzama = 0;
    protected oldalHossz = 0;

    public function __construct
    ➥($oldalHossz,$oldalakSzama) {
        $this->oldalHossztBeallit($oldalHossz);
        $this->oldalakSzamatBeallit($oldalakSzama);
    }

    public function oldalHossztBeallit
    ➥($oldalHossz) {
        $this->oldalHossz =$oldalHossz;
    }

    public function oldalakSzamatBeallit
    ➥($oldalakSzama) {
        $this->oldalakSzama=$oldalakSzama;
    }

    public function oldalakAltalBezartSzog() {
        return 360/$this->oldalakSzama;
    }

}
```

Ekkor a gyermekosztályokhoz nem is kell hozzányúlni, de minden későbbit úgy kell elkészíteni,hogy tartalmazz a `terulet()` tagfüggvényt. Felmerülhet a kérdés, hogy miért nem jelez hibát a PHP a fenti esetben, holott az osztályban nem is valósítottuk meg azt a bizonyos `terulet()` metódust. Ennek az az oka, hogy ez egy elvont osztály, tehát nem példányosodhat, ennek értelmében biztosan nem fogja megsérteni a szabályt. Megsértik viszont azok az örökösök, akik elmulasztják eme tagfüggvény megvalósítását. Jelen esetben tehát csak ennyi szerepe van osztályunk elvont voltának, mert mint láthatjuk, nem tartalmaz egyetlen elvont metódust sem. Az is egy megoldás lett volna továbbá, ha a szülőt változatlanul hagyjuk és a gyermekosztályoknál az öröklés után megmondjuk, hogy ezek a `Teruletes` nevű felületet valósítják meg. (A megoldás hátránya többek között az, hogy így minden sokszögfajtára le kell ellenőriznünk a használat során, hogy implementálják-e a várt felületeket)
A gyakorlatban remekül lehet kombinálni az öröklést a felületeket és az elvont osztályok alkalmazását. Sok esetben vezet igen-igen érdekes eredményre. Ha jobban megnézzük, a fenti esetben is ezt alkalmazzuk. Természetesen a sok osztálynak, melyek ugyanazt a felületet valósítják meg, nem kell feltétlenül egy helyről öröklődniük, lehetnek teljesen függetlenek, látni fogjuk később, hogy lehetőségünk van ellenőrizni egy osztályról, objektumról, hogy megvalósítja-e a szerintünk szükséges felületet, felületeket.

Néhány mágikus tagfüggvény

Az összes ilyen különleges metódusnak közös tulajdonsága, hogy ezeket nem mi, programozók hívjuk, hanem a *PHP*. Hogy tiszta legyen a kép: ide tartoznak a konstruktorok és destruktorok, ám azok ismertetése elengedhetetlen volt az alapvetésnél. Most a `__set()`, `__get()` és a `__call()` tagfüggvényeket vizsgáljuk meg közelebbről. Megjegyezném, hogy a *PHP 5* ezen kívül is tartalmaz még olyan függvényeket, amelyek __-rel kezdődnek (ilyen előtag azonosítja ugyanis a különleges tagfüggvényeket), ám ezek tárgyalása a szűkös terjedelmi keretek miatt most elmarad.

A __set() tagfüggvény

Ha egy osztály, objektum tartalmazza ezt a metódust, akkor minden olyan esetben lefut, ha mi az adott objektum nem létező tulajdonságának adunk értéket. Ez olyankor lehet hasznos, ha egy dinamikusan bővülő adathalmaz alkotja osztályunk tulajdonságait. Példaképp, ha a program futása során különböző gyümölcsök színét szeretnénk összegyűjteni. Célszerű ilyenkor a gyümölcsöket nem külön-külön felvenni, mint osztálytulajdonságot, hiszen az fix, ehelyett jó lenne dinamikusan bővíteni. Mivel azért a nyelv nem teszi lehetővé, hogy futásidőben ily módon piszkáljuk az osztályokat, osztálytulajdonság gyanánt használjunk struktúrát, asszociatív tömböt a megoldásra. Ez kellően rugalmas. Az esetleges beállító tagfüggvényekkel ugyanez a helyzet: nem elég rugalmasak. Itt jön a képbe a `__set()` metódus, nézzük, hogyan:

```
<?php
class Gyumolcsok {
    private $gyumolcsok = array();

    public function __set($name,$value) {
        $this->gyumolcsok[$name]=$value
    }
}
$deliGyumolcsok = new Gyumolcsok();
$deliGyumolcsok->narancs="sarga";
```

```
$deliGyumolcsok->citrom="citromsarga";
$deliGyumolcsok->banan="erdekesen sarga";
?>
```

A __set() tagfüggvény első paramétere a változó neve (ami a ->után szerepel), a második paraméter pedig az érték, ami az egyenlőségjel után szerepel.
A feladat, tehát megoldva. Bármilyen gyümölcsöt beletehetünk, kötöttségek nélkül, az érték nem fog elveszni. Vigyáznunk kell azonban, hogy semmilyen gyümölcsnév ne szerepeljen osztálytulajdonságként, mert ha teszem azt van $narancs nevű osztályváltozó, akkor a második hívás annak értékére fog vonatkozni, nem fut le a __set() metódus.

A __get() metódusa

Ha már van egy ilyen szabadon feltölthető osztályunk, nem ártana, ha legalább ilyen szabadon hozzáférhetnénk. A __set() párja, a __get() siet ilyenkor a segítségünkre. Teljesen analóg módon: ez akkor fut le, ha olyan változó értékére vagyunk kíváncsiak, amely nem szerepel az osztálytulajdonságok között. A fentiek ismeretében bővítsük tovább az osztályunkat.

```
<?php
class Gyumolcsok {
    private $gyumolcsok = array();

    public function __set($name,$value) {
        $this->gyumolcsok[$name]=$value
    }

    public function __get($name) {
        if (array_key_exists($name,
           ➥$this->gyumolcsok))
            return $this->gyumolcsok[$name];
        else
            echo 'Nincs ilyen gyümölcs!';
    }
}
$deliGyumolcsok = new Gyumolcsok();
$deliGyumolcsok->narancs="sarga";
$deliGyumolcsok->citrom="citromsarga";
echo $deliGyumolcsok->narancs;
echo $deliGyumolcsok->banan;
?>
```

A példa az első esetben kiírja, hogy sárga, a második esetben, hogy Nincs ilyen gyümölcs. A megoldás egyértelmű: a __get() paramétere tartalmazza, hogy milyen változóra hivatkoztunk. Megnézzük, hogy létezik-e az adott kulccsal érték a tömbben. Ha igen, akkor visszaadjuk azt, különben kiírjuk, hogy nincs olyan.

A __call() metódus

Ez a tagfüggvény is illeszkedik az előző kettőre, ám ez akkor fut le, ha olyan tagfüggvényt hívunk meg az objektumunk esetében, amely nem létezik. Az előző kettő esetben a program semmilyen hibát nem ír ki, a __get() ill. __set() metódusok hiányában sem, ha olyan változóra hivatkozunk, ami nem létezik. A tagfüggvények hívása esetén ez nincs így, ezért még akár hibaelnyelés céljából is hasznos lehet. A metódusnak két paramétere van. Az első a meghívott tagfüggvény neve, a második pedig a híváskor átadott paraméterek tömbje. Ennek segítségével bizonyos hívásokat elkaphatunk, s különböző műveleteket végezhetünk, mielőtt visszatérnénk. (A visszatérési érték természetesen ugyanúgy viselkedik, mintha valóban létezne az a metódus). A cél itt is az volt, hogy az objektumok futás idejű dinamikus viselkedését növeljék.

Egy kakukktojás – az automatikus betöltés

Bár nem kapcsolódik szorosan az objektumközpontú viselkedésmódhoz, de mindenképp hasznos és érdekes móka az egyes osztályok igény szerinti betöltésének lehetősége.
A *PHP* számára eddig is előnynek számított, hogy maga a forráskód futásidőben alakítható volt az include, require parancsokkal – ha például feltételhez kötöttük őket. Így ugyanis elérhettük, hogy mindig csak a minimálisan szükséges méretű kódot kellett a *PHP*-nak értelmeznie, lefordítania, amivel jelentős sebességnövekedést eredményezett.
Az osztályokat gyakorta helyezzük el külön-külön fájlokban, ami átláthatóvá teszi kódunkat, ám elég kényelmetlen, hogy minden egyes példányosítás előtt be kell töltenünk az adott fájlt. Erre nyújthat megoldást az __autoload() nevű különleges függvény, amely minden olyan esetben meghívódik, ha nem definiált osztályt szeretnénk használni, példányosítani. A függvény meghívása után a *PHP* még egyszer megpróbálja használni azt a bizonyos osztályt, s ha ekkor sem sikerül, hibaüzenettel leáll. A gyakorlatban mindez így nézhet ki:

```
<?php
function __autoload($className) {
  include_once($className.'.php');
}

$obj = new ProbaOsztaly();
$obj2 = new MasikProbaOsztaly();
?>
```

Ez természetesen nem az egyetlen felhasználási mód, egész sor automatikát vihetünk általa programunkba.

Végszó

Nos, még mindig nem értünk a téma végére, de már közel járunk. Hátra van még a nagy újításnak számító ***Refleciton API***, amely az objektumok, osztályok részletes elemzésére szolgál, de nem beszéltünk még a típus előírásról (***type hinting***), a kivételkezelésről és az objektumok másolásáról sem. A cikksorozat következő részében ezekre lehet tehát számítani.

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