© Kiskapu Kft. Minden jog fenntartva

Kávéfőzés lépésről lépésre (2. rész)

Az öröklődés, avagy mindig legyen nálad egy angol szótár ha programozol.

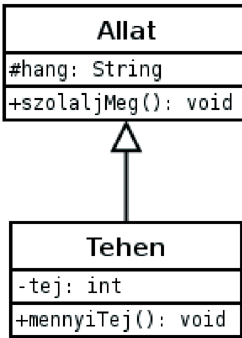
Múlt hónapban feltelepítettük a *Java* fejlesztői környezet 5.0-ás változatát, megírtuk a kötelezőnek számító „Helló világ!” alkalmazást, végül egy tehén, egy ló és egy kutya képében megismertük az objektum fogalmát. Megtudtuk, hogy az objektum nem más, mint egy osztály példánya. Az osztály általános leírást jelent, az objektum ennek az általánosításnak egy megvalósítása. Szó volt még a tagváltozókról, vagy tulajdonságokról, és az ezeken műveletet végző tagfüggvényekről, azaz metódusokról. Ezek igen közel állnak a hagyományos programozási nyelvek változó- és függvényfogalmához.
Elevenítsük fel az állatok hangját utánzó program main(String[]) tagfüggvényt:

```
...
public static void main(String[] args) {
   Allat tehen = new Allat(“múúú”);
   Allat lo = new Allat(“nyihaha”);
   Allat kutya = new Allat(“vauvau”);
   tehen.szolaljMeg();
   lo.szolaljMeg();
   kutya.szolaljMeg();
}
...
```

A metódus nyilvános (public), ami azt jelenti, hogy az osztályon kívülről elérhető. Továbbá statikus (static), ami azt jelenti, hogy az osztály minden példányára közös, és ezért akár példányosítás nélkül is használható. Mivel ez jelenti az alkalmazás belépési pontját, ennek szükségképpen így kell lennie. Nem rendelkezik visszatérési értékkel (void), paraméterként pedig szövegfüzérek tömbjét kapja (String[]), mely a parancssori paramétereket tartalmazza. A C-hez szokott szemnek talán furcsa, de ennek mindig így kell lennie, nem hozható létre belépési pont például int típusú visszatérési értékkel.
A tagfüggvényben először létrehozunk három változót, melyek Allat típusúak, és rögtön kezdőértéket is kapnak egy-egy példányosítás révén. A tehen, a lo és a kutya úgynevezett referenciák, segítségükkel objektumokra hivatkozhatunk. Az objektumok létrehozása az osztály konstruktorának meghívásával történik, és ezen keresztül állítjuk be a hangjukat. Végül az objektumoknak egyesével meghívjuk a szolaljMeg() metódusát, így a képernyőn megjelennek az állatok hangjai.
Teljhatalmú programozóként állatokat hoztunk létre, és láttuk, hogy ez jó. Mégis, a *Java* mögött álló szemlélet azt sugallja, hogy a megoldás még nem tökéletes. Bármilyen objektumközpontú alkalmazás fejlesztéséhez modellalkotáson keresztül vezet az út. Az általunk használt modellben minden állat ugyanazokkal a tulajdonságokkal rendelkezik. Így, ha a megbízónk tehenek esetében meg szeretné jeleníteni a naponta adott tej mennyiségét, már gondban lennénk. Az Allat osztályt nem bővíthetjük ilyen tulajdonsággal, mert azzal azt állítanánk, hogy lovaink és kutyáink is tejtermelő jószágok. Kézenfekvő megoldásnak tűnik, hogy képviseljen a tehén önálló osztályt. Kérdés viszont, hogy mi legyen a meglévő tagváltozókkal és tagfüggvényekkel. Egyszerű, másolás-beillesztés módszerrel kaphatunk egy megoldást, de érezhető, hogy ez nem csak csúnya, de rossz hatékonyságú kódot eredményező módszert is jelentene. Az objektumközpontúság egyik nagy előnye pedig a nagy mértékű kódújrafelhasználás, aminek itt valahogy érvényt kell szerezni. Ekkor lép képbe az ***öröklődés (inheritance)***.

A tehén egy állat, de...

Öröklődésről akkor beszélünk, amikor egy meglévő osztályból kiterjesztéssel egy új osztályt hozunk létre. Az eredeti, vagyis az ősosztály megfelelő láthatóságú elemeit megörökli az új, mintha csak saját tulajdonságokról és metódusokról lenne szó. Az egyetlen kérdés a láthatóság, de mielőtt elvesznénk ennek részleteiben, fogadjuk meg azt a bölcsességet, mely szerint egy kép többet mond ezer szónál, és rajzoljuk le a modellt, mielőtt megvalósítanánk.
Az ***UML*** a ***Unified Modeling Language*** rövidítése, és pont az, amit a neve is mutat: egységesített modellező nyelv. Objektumközpontú problémák leírására szolgál, programozási nyelvtől és környezettől függetlenül. A szabvány által bevezetett diagrammok segítségével könnyen emészthetően fogalmazhatók meg objektumközpontú állítások, melyek megvalósítása egy-egy programozási nyelvben már kevés önálló ötletet igénylő kódolási feladatot jelent. Nem célom az ***UML*** részletes bemutatása, csak annyira használom, amennyire segítheti az objektumközpontúság alapfogalmainak megértését. Lássunk egy ***UML*** diagrammot az öröklődésre.
Az ábrán két osztály látható, ezeket három rekeszre osztott dobozok jelképezik. Az első rekesz az osztály nevét, a második a tulajdonságokat, a harmadik a műveleteket mutatja.
Az üres hegyű, folytonos vonalú nyíl mutatja az öröklődést. A nyíl irányára több magyarázat is létezik. Szerintem a legfontosabb annak a ténynek a hangsúlyozása, hogy ennél a kapcsolatnál a Tehen osztály hivatkozik az Allat-ra, és az ősosztály semmilyen formában nem értesül az öröklődésről, pusztán elszenvedi azt. Az osztályok elemei mögött, egy ket-



tőspont után azok típusa látható. Az elemek nevét pedig egy most minket jobban foglalkoztató jelölés előzi meg. Egyetlen különleges karakter, ami a láthatóságot mutatja. A + a nyilvános, a - a személyes, a # pedig a védett (`protected`) tagváltozót vagy -függvényt jelenti.
Ez utóbbiról még nem volt szó. Nézzük meg, hogyan alakul a láthatóság öröklődés esetén.
Egy nyilvános elem mindenki számára szabadon hozzáférhető, és ezen az öröklődés nem változtat. Egy személyes elem ezzel szemben a teljes elrejtés megvalósítása. Ha származtatunk egy osztályt, abban nem lesz látható egy személyesre állított elem. Érezhető, hogy szükség van egy köztes megoldásra a láthatóság tekintetében. Ez a védett, ami idegen osztályoknak továbbra sem elérhető, öröklődés esetén pedig megtartja védett tulajdonságát, ezért tetszőleges számú öröklés után is látható osztályon belülről.
Ezért, noha a múlt hónapban személyesre állítottuk az `Allat` osztály `hang` tagváltozóját, ennek láthatóságát védettre kell állítani. Nagyon fontos, hogy ezt nem a `szolaljMeg()` metódus érdekében tesszük! A `Tehen` örökli ezt a metódust, és az meghívható egy `Tehen` típusú objektum esetén. A metódus használja a `hang` változót, de a láthatóság kérdésének eldöntésekor az számít, hogy a művelet hol helyezkedik el, és nem az, hogy melyik osztály példányáról van szó.

Ha nem lenne olyan tagfüggvény a `Tehen` osztályban, amely felhasználja a hang tulajdonságot, nem lenne szükség a védett láthatóságra. Viszont ha egy tehéntől megkérdezzük, hogy hány liter tejet ad, hozzáteszi, hogy „múúú”, ezért kell a `protected` módosító.
Lássuk a megvalósítást. Két osztályunk, így két forrásállományunk lesz. Az első a már múlt hónapban is látott `Allat.java`, azzal a módosítással, hogy a `hang protected`:

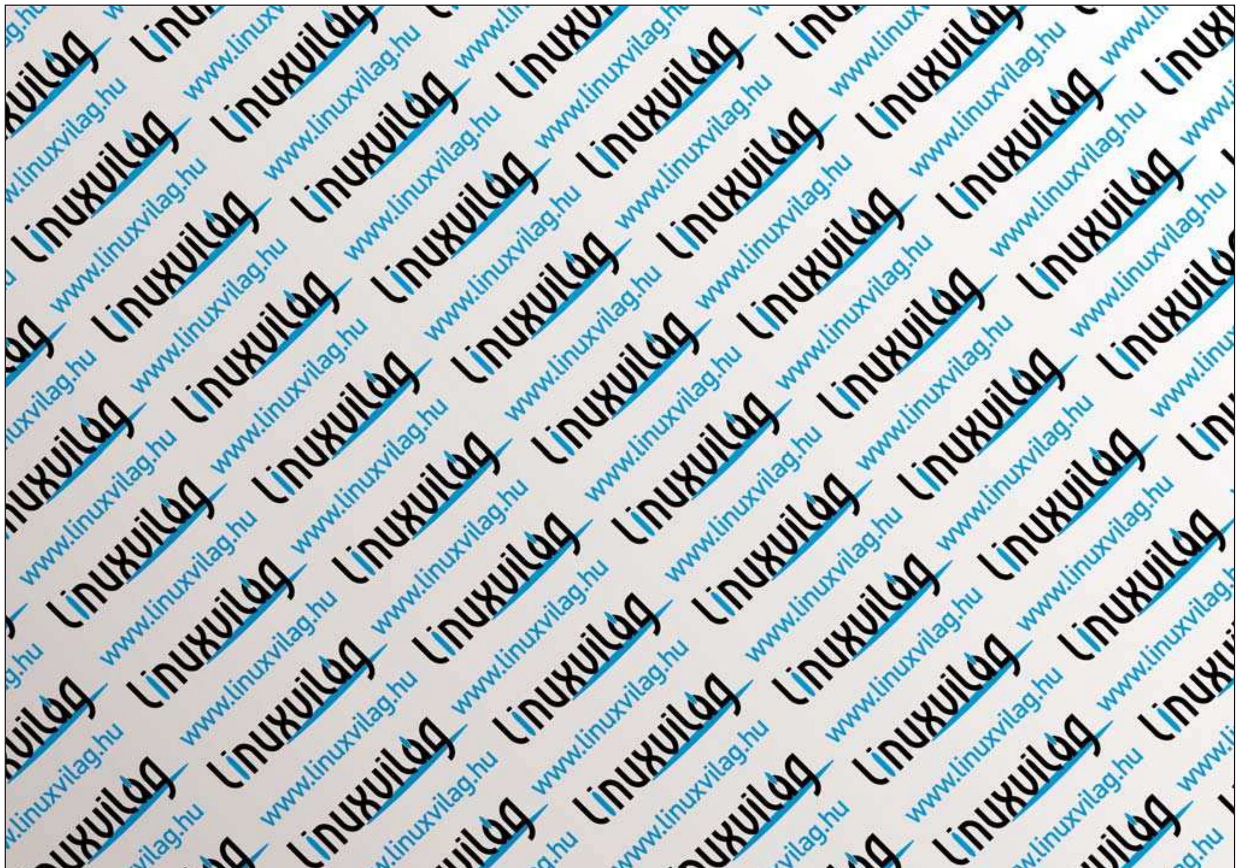
```
/**
* Ez az osztaly egy allatot ir le.
* Tulajdonsaga a hangja.
* Meg lehet szolaltatni.
*/
public class Allat {

    /**
    * Az allat hangja.
    */
    protected String hang;

    /**
    * Letrehoz egy uj allatot
    * a megadott hanggal.
    */
    public Allat(String h) {
       hang = h;
    }

    /**
```

© Kiskapu Kft. Minden jog fenntartva



© Kiskapu Kft. Minden jog fenntartva

```
    * Kiirja a kepernyore az
    * allat hangjat.
    */
    public void szolaljMeg() {
       System.out.println(hang);
    }

    /**
    * A program belepesi pontja.
    * Letrehoz harom allatot, es
    * megszolaltatja oket.
    */
    public static void main(String[] args) {
       Tehen tehen = new Tehen("múúú", 20);
       Allat lo = new Allat("nyihaha");
       Allat kutya = new Allat("vauvau");
       tehen.szolaljMeg();
       tehen.mennyiTej();
       lo.szolaljMeg();
       kutya.szolaljMeg();
    }

}
```

Következzen a ***Tehen.java*** állomány:

```
/**
* Ez az osztaly egy tehenet ir le
* Tulajdonsaga a tej mennyisege, amit egy nap ad.
* Ezt ki lehet iratni a kepernyore.
*/
public class Tehen extends Allat {

        /**
        * A naponta adott tej mennyisege.
        */
        private int tej;

        /**
        * Letrehoz egy uj tehenet a megadott
        * hanggal, és napi tej mennyiseggel
        */
        public Tehen(String h, int t) {
                super(h);
                tej = t;
        }

        /**
        * Kiirja a kepernyore a napi
        * tej mennyiseget
        */
        public void mennyiTej() {
                System.out.println(tej + " liter,
                ➥" + hang);
        }

}
```

A fordítás és futtatás a megszokott módon történik:

```
$ javac Allat.java Tehen.java
$ java Allat
```

Az új osztály bevezetése akkor nyer értelmet, ha használatba is vesszük, így a program belépési pontja is változott. Tehenet immár a `Tehen` osztály példányosításával hozunk létre.
Az öröklés miatt meghívhatjuk az ősosztályban szereplő `szolaljMeg()` mellett a kibővítés révén nyert `mennyiTej()` metódust is. Így a képernyőn az alábbi kimenet látható:

```
múúú
20 liter, múúú
nyihaha
vauvau
```

Időzzünk el még egy keveset az öröklődés példaképét jelentő `Tehen` osztály elemzésével. Magának a kapcsolatnak a kifejezése nagyon egyszerű, csupán az osztály fejében kell ezt jeleznünk az `extends` (kiterjeszt) kulcsszó segítségével. Az egyetlen kérdés az, hogy pontosan mi is történik a példányosításkor meghívódó konstruktorokkal. Nyilvánvalóan akkor hasznos az öröklődés, ha a meglévő metódusokat nem kell újraírni, így a tagváltozókat értékkel ellátó konstruktort sem. Lehet valahogy hivatkozni az ős kosntruktorára?
A válasz igen, de nem magától értetődő, hogy hogyan.
A konstruktor különleges függvény abban a tekintetben, hogy minden más metódussal ellentétben nem jelezzük a visszatérési értékét, hiszen meghívásakor objektum készül, amit viszont nem a függvény ad. Pontosan ezért különleges bánásmódot igényel. Meghívása a `super` hivatkozással történik, melyet az ős konstruktorának megfelelően kell paraméterezni.
Két dolgot hangsúlyoznék ezzel kapcsolatban. Ha az ősosztály tartalmazna paraméter nélküli konstruktort, és a származtatott osztály konstruktorában nem lenne `super` hívás, akkor a származtatott osztály példányosításakor először önműködően meghívódna az ősosztály kosntruktora. Továbbá nagyon fontos, hogy ha így hivatkozunk egy ősosztálybeli konstruktorra, a `super` hívásnak a legelső műveletnek kell lennie, egyébként a fordító hibát is jelez.
A `mennyiTej()` metódusban a kiíratásnál a + operátor segítségével fűzzük eggyé a szövegfüzéreket. Ez egy igen kényelmes, és könnyen olvasható megoldást jelent. Ugyanakkor ez azt is jelenti, hogy a + több értelemmel is bír, hiszen számok aritmetikai összeadása mellett `String` típusú objektumok összefűzését is lehetővé teszi. Ez egy beépített megoldás, amit az objektumközpontúság szakzsargonjával élve operátor túlterhelésnek hívunk (`operator overload`). Jó tudni, hogy egyes nyelvek, például a ***C++*** ezt a programozó számára is lehetővé teszik. Jelen helyzetre lefordítva, ha azt mondanánk, hogy két tehén egyenlősége akkor teljesül, ha ugyanannyi tejet adnak, túlterhelhetnénk az == operátort, hogy kényelmesen hasonlítsunk össze két `Tehen` típusú objektumot. Ugyanez ***Java***-ban hatékonysági okokból nem megvalósítható.
Mindezek alapján már bátran belegondolhatunk, mi is az ami miatt hasznos az objektumközpontú tervezés.
Egy kellően átgondolt modell segítségével olyan alkalmazást építhetünk, ami nem csak könnyebben átlátható, de egyszerűen bővíthető is. A sorozattal kapcsolatban várom az észrevételeket, javaslatokat. Következő hónapban az interfészek lesznek terítéken.

Fülöp Balázs (bigwig42@gmail.com)