Kávéfőzés lépésről lépésre (3. rész)
Tömbök és interfészek, avagy a változatosság gyönyörködtet

Az előző részben megismerkedtünk az öröklődés fogalmával. Ennek a módszernek a segítségével egy meglévő osztályt bővíthetünk újabb tulajdonságokkal és műveletekkel. Ehhez gondos tervezés esetén nincs szükség az ősosztály módosítására, valóban nincs másról szó, mint egyszerű kiterjesztésről.

Példánkban egy állatokat jelképező osztályból örököltettünk egy `Tehen` nevű osztályt, amely új tulajdonságát, a naponta termelt tej mennyiségét ki is tudta íratni a képernyőre.
Ahhoz, hogy ennél a kiíratásnál megjelenítsük az állat hangját is, elveinknek ellentmondva módosítást hajtottunk végre az ősosztályban. Az `Allat` osztály `hang` változójának láthatóságát személyesről átállítottuk védettre. Erre nem lett volna szükség, ha tudjuk előre, hogy öröklődést szeretnénk majd alkalmazni. Ez tehát elkerülhető egy átgondolt modell esetén, aminek megalkotásában nagy segítségünkre lehetnek az ***UML*** diagramok, melyek közül egyre szintén láttunk példát.
Vessünk most egy pillantást az elmúlt alkalommal készített alkalmazás belépési pontjára:

```
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();
}
```

Noha a forráskód az objektumközpontú filozófia számos ismertetőjegyét magán viseli, még mindig bőbeszédűnek érezhetjük a leírtakat. Felmerül a kérdés, hogy ha a `tehen`, a `lo` és a `kutya` objektumot is megszólaltatjuk, miért kell ezt három, független metódushívással megtenni. Gondoljunk bele, ha nem csak három objektumunk lenne, hanem mondjuk száz. Ez esetben százszor kellene leírni gyakorlatilag ugyanazt? Szerencsére nem.
Mindhárom objektumban van valami közös, mégpedig az, hogy az `Allat` osztályból származnak. A `lo` és a `kutya` közvetlenül, hiszen ezek az `Allat` példányai. A `tehen` ugyan a `Tehen` osztály példánya, de ez az `Allat` kiterjesztése, így rá is igaz az állítás. Ez a közös vonás pedig egyszerűen megragadható úgy, hogy objektumainkat egy tömbbe gyűjtjük össze.

Tömböm, tömböm mondd meg nékem

Java-ban a tömb szerkezet is objektumként jelenik meg. Ez azt jelenti, hogy egy tömb létrehozásához ugyanúgy a `new` kulcsszót kell használnunk mint eddig. Fontos különbség viszont, hogy a referencia változónk, mellyel a létrejövő objektumra tudunk hivatkozni valamilyen tömb típusú, és ezért egyben indexelhető is. Lássuk, hogyan néz ki az átírt `main()` függvény:

```
public static void main(String[] args) {
   Tehen tehen = new Tehen("múúú", 20);
   Allat lo = new Allat("nyihaha");
   Allat kutya = new Allat("vauvau");

   Allat[] allatok = new Allat[3];
   allatok[0] = tehen; allatok[1] = lo;
   ➥allatok[2] = kutya;

   for (int i = 0; i < allatok.length;
   ➥i ++) {
          allatok[i].szolaljMeg();
   }

   tehen.mennyiTej();
}
```

A létrehozott objektumokból egy `allatok` nevű tömböt készítünk, amint az a metódus 4. és 5. sorában látható. Az `allatok` létrehozása után újabb példányosítás már nem történik, csupán feltöltjük a tömböt az objektumok referenciáival. Így az eredeti néven túl egy másik úton

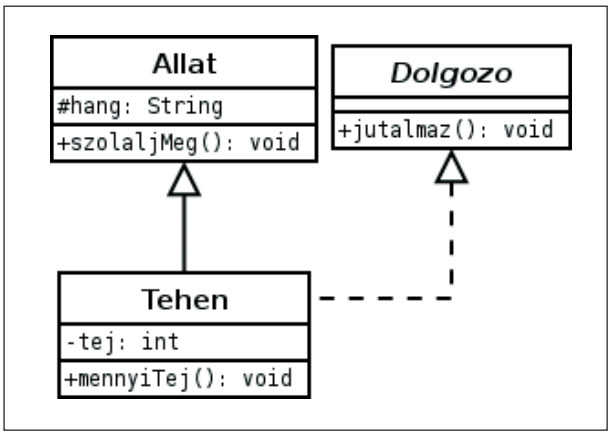
is megcímezhetjük ugyanazokat az objektumokat.
Az = operátorral csupán az objektumok elérhetőségeit másoltuk át a tömbbe, nem a teljes objektumokat.
Az ezután következő `for` ciklus *C*-ből ismerős lehet, viszont több ponton segít rajtunk a Java. A ciklus fejében három rész található, melyeket `;` választ el egymástól. Az első rész készíti elő a használandó változókat, mivel ez még a törzsbe való első belépés előtt lefut. Kényelmes és hasznos, hogy a ciklusváltozót elég itt létrehozni, és nem kell a teljes függvény legelején. Ennek az a haszna is megvan, hogy a ciklus lefutását követően ez a segédváltozó megszűnik. A második rész a törzs minden lefutása előtt kiértékelődik, és csak akkor folytatódik a ciklus, ha ez a feltétel teljesül. Itt ellenőrizzük, hogy a ciklusváltozó, amellyel a tömböt fogjuk indexelni, nem lépte-e túl a tömb méretét. Kihasználjuk, hogy a tömb egy objektum, és szerencsénkre a mérete egy tagváltozón keresztül lekérdezhető. Végül az utolsó rész az egyes lefutások után hajtódik végre, ezen a helyen növeljük a ciklusváltozót.
Így a ciklus háromszor fut le, előbb a tehenet, majd a lovat, végül a kutyát szólaltatja meg. Ezzel a módszerrel elértük, hogy ha növelni szeretnénk az állataink számát, a kiíratás részébe már nem kellene belenyúlnunk. Tetszőleges számú referenciát tartalmazhat az `allatok` tömb, az összes hivatkozott objektumnak meghívásra kerül a `szolaljMeg()` metódusa.
Jó kérdés, hogy ha ez ilyen remekül működik egy egyszerű metódus meghívásakor, miért ne oldhatnánk meg hasonlóan az objektumok létrehozását. Ekkor nem is lenne szükség másra, csak a tömbre, melynek elemei egy hasonló ciklusban egyenként kaphatnának egy új objektumot. Ennek jelen helyzetben több dolog is ellentmond. Egyrészt a tehén `Tehen` osztályból származik, és így noha hivatkozhatunk rá egy `Allat` típusú referenciával, de csak az `Allat` osztályban meglévő tagokat érhetjük el. Ha létrehoznánk egy `Tehen` típusú objektumot, de csak olyan referenciánk lenne rá, ami `Allat` típusú, elveszítenénk azt, amivel az ősosztályt kiegészítettük. Másrészt gond lenne a konstruktorok paraméterezésével is, hiszen honnan tudjuk egy ciklus belsejéből, hogy épp melyik hangot kell átadni. Ebben az esetben tehát ez nem egy jó ötlet, viszont több hasonló objektum létrehozásakor kényelmes megoldást jelenthet.

Díjazzuk a teheneket

Most játszunk el a gondolattal, hogy főnökünk hazaért az első vakációból és elmondja, milyen kiváló ötlete támadt, mialatt a Balaton partján pihent. Teheneink látható módon keményen dolgoznak, hiszen minden tehén objektumnak tulajdonsága a naponta termelt tej mennyisége. Jó lenne külön jutalommal díjazni ezt a megfeszített munkát. Ezért szükséges a jól dolgozó állatoknak egy `jutalmaz()` metódus, mellyel kifejezhetjük elismerésünket a teljesítményért.
Első ötletünk az lehetne, hogy egyszerűen vegyük fel a `Tehen` osztályban az említett eljárást, és mi is menjünk nyaralni. Emlékezzünk azonban vissza saját hibánkra, mikor egy egyszerű öröklöttetéshez bele kellett nyúlnunk az ősosztály kódjába. Ahhoz, hogy ne essünk újabb csapdába, gondoljuk végig, mit állítanánk azzal, ha jelenlegi modellünket úgy egészítenénk ki, hogy a `Tehen` osztályba minden további nélkül belevésnénk egy ilyen metódust. Ezzel azt fejeznénk ki, hogy a tehenek, és csak a tehenek jutalmazhatóak. Bármennyire is kedveljük ezeket a béke szimbólumaként ismert, szeretetreméltó állatokat, ezt nem jelenthetjük ki.
Ha már az objektumközpontúság a magas fokú kódújrahasznosítás kecsegtet, ne mondjunk ennek ellent egy helytelen modellel. A teheneket valóban meg lehet jutalmazni, ugyanakkor ez később igaz lehet a lovakra, vagy a kutyákra is, mi több, alkalmazásunk fejlődésével még emberekre is. Ezért az `Allat` osztály sem rendelkezhet egyértelműen a művelettel. Mit tehetünk ebben a helyzetben?
Elsőként gondolhatunk arra, hogy készítünk egy `Dolgozo` osztályt, és ebből örököltetjük az állatokat, később az embereket jelképező osztályt is. Ez már egy fokkal jobb legelső megközelítésünknél, de még mindig nem helyes. Egy alkalmazás osztályhierarchiája a valóság egy képe. Ha létrehoznánk egy ilyen nevű közös ősosztályt, az azt jelenthetné, hogy dolgozó az egész világ, és dolgozik benne minden állat és ember. Ne felejtsük el, egyelőre a lovak és a kutyák nem jutalmazhatóak!
Érezzük, hogy az új metódus, ami keresztbe tett a nyarunknak, valamilyen formában önálló osztályként kell, hogy fellépjen. Kényelmes lenne, ha készíthetnénk egy `Dolgozo` osztályt, de ez nem közös ősosztály volna, hanem egy független szerkezet. Ebből a szerkezetből lenne jó örököltetni azokat és csak azokat az osztályokat, amelyeknek jutalom szabható. Ez jelen esetben a `Tehen` osztályra lenne igaz, aminek viszont már van egy őse.
Ha a *C++* nyelv mellett döntöttünk volna a sorozat elején, ez nem lenne gond. Ott ugyanis megengedett a többszörös öröklődés. *Java*-ban ez már nem igaz, itt egy osztály közvetlenül csak egy ősből származhat. Ennek az az oka, hogy problémás annak a helyzetnek a kezelése, amikor egy tag két osztályban is szerepel, és ebből a kettőből örököltetünk. Azért, hogy ez még meggondolatlan tervezés eredményeként se fordulhasson elő, a *Java* nem támogatja a többszörös öröklődést. Amit e szándékosan kifelejtett módszer helyett használhatunk az az interfész.

Interfészek, madárfiókák nélkül

Az interfész hasonló a már megismert osztályokhoz, viszont nélkülözi a változtatható tulajdonságokat, azaz a nem állandó tagváltozókat, és a metódusok törzseit. Ezért tényleg nem más, mint az angol kifejezés, az *interface* magyar jelentése: *felület*. Egy interfész önmagában nem használható, ha nem áll mögötte olyan osztály, amely megvalósítja. A megvalósítás azt jelenti, hogy az interfész által előírt összes metódus törzzsel együtt előfordul az adott osztályban. Lássuk, hogyan fest ez egy *UML* osztálydiagramon (*1. ábra*).
Az ábrán is látható, hogy az interfész a gyakorlatban eléggé közel áll az osztályhoz. Egy hasonló felépítésű doboz jelképezi mindkettőt. Az interfésznek a fejléce azonban dőlt betűs, ami annak elvontabb, idegen kifejezéssel élve absztrakt voltát mutatja. Tagváltozói csak állandók lehetnek, jelen esetben ez a rekesz üres. Műveletei helyén most egyetlen metódus áll, ez a `jutalmaz()`.



1. ábra

Az osztály és interfész közötti fogalmat akkor lehet jól megfogni, ha tisztán modellezési eszközként tekintünk rájuk. Egy osztály tulajdonságok és ezeken végezhető műveletek összessége. Az interfész ezzel szemben annak a leírása, hogy egy osztály milyen műveletekkel kell, hogy rendelkezzen, ha ennek egy megvalósítása. Azt is fontos észrevenni, hogy a megvalósítás jele az *UML* diagramon a szaggatott nyíl, ami megkülönbözteti az örökléstől. Itt ugyanis nem meglévő tagok átvételéről, hanem adott fejlécű műveletek kialakításáról van szó.
Ennek egy fontos következménye, hogy ha készítenénk `Ember` osztályt is, melynek példányait szeretnénk jutalmazni, akkor ebben a `jutalmaz()` metódust külön kell megvalósítani. Ez jól illeszkedik jelen modellünkbe, hiszen valószínűleg másképp szeretnénk jutalmazni egy embert, mint egy tehenet. Azokban az esetekben, ahol ez nem így van, át kell gondolni, hogyan lehetne megoldani a problémát tiszta öröklődéssel.
Lássuk a `Dolgozo` interfész forráskódját. A már megszokott módon, `Dolgozo.java` néven kell elmenteni az alábbiakat:

```
/**
* Ez az interfesz egy olyan dolgozot
* ir le, amelyet meg lehet jutalmazni
* A hogyan kerdese a megvalosito
* osztaly feladata
*/
public interface Dolgozo {
        /**
        * Megjutalmazza a dolgozot.
        */
        public void jutalmaz();
}
```

A kód a megjegyzésekkel együtt is tömör és lényegre törő. Az interfész az azonos nevű kulcsszóval hozható létre. A metódusoknak csupán a fejlécét tartalmazhatja, melyet ; jellel kell lezárni. Nézzük most meg az új `Tehen` osztályt, amely már megvalósítja ezt az interfészt.

```
/**
* 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 implements
➥Dolgozo {

        /**
        * 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 tehen megjutalmazasa
        * eseten megszolal es elmosolyodik
        */
        public void jutalmaz() {
                System.out.println(hang + " :-)");
        }

}
```

A változás mindössze az osztály fejlécének kiegészítését és az új metódus bevezetését jelenti. A fejlécben az `implements` kulcsszó mutatja a megvalósítást. Ennek megadásakor, ha elfelejtünk elkészíteni egy tagfüggvényt, az már fordításkor hibát okoz. Az új metódus a megjegyzés és az eddigiek alapján magától értetődő.
Az elmondottak egy kicsit talán rávilágítanak annak a ténynek a miértjére, hogy egy jó programozó egy nap nem ír többet 20 sor kódnál. Rengeteg időt és fáradtságot takaríthatunk meg, ha a vad billentyűtépés helyett először belegondolunk abba, hogy mit szeretnénk elérni. Ezután érdemes elgondolkozni azon, hogy a fejünkben alakuló megoldás eléggé általános-e ahhoz, hogy joggal nevezhessük objektumközpontúnak.
Miután felébredtünk a mély önelemzésből, térjünk vissza teheneinkhez. Mint azt már említettem, az egy osztályból származó egyedek közös vonását kihasználva egyszerű ciklusszerkezettel lépdelhetünk végig az objektumokon. Miért ne tehetnénk ezt meg az ugyanazt az interfészt megvalósító osztályok példányai esetében? Ennél mi sem egyszerűbb, amire jó példa az alábbi, újraírt `main()` függvény:

```
public static void main(String[] args) {
    Tehen tehen1 = new Tehen("múúú", 20);
    Tehen tehen2 = new Tehen("múúú", 30);
    Allat lo = new Allat("nyihaha");
    Allat kutya = new Allat("vauvau");

    Allat[] allatok = new Allat[4];
    allatok[0] = tehen1;
    allatok[1] = tehen2;
    allatok[2] = lo;
    allatok[3] = kutya;

    for (int i = 0; i < allatok.length;
    ➥i ++) {
            allatok[i].szolaljMeg();
    }

    Dolgozo[] dolgozok = new Dolgozo[2];
    dolgozok[0] = tehen1;
    dolgozok[1] = tehen2;

    for (int i = 0; i < dolgozok.length;
    ➥i ++) {
            dolgozok[i].jutalmaz();
    }
}
```

Az alkalmazás fordítása és futtatása után a várt kimenetet láthatjuk:

```
múúú
múúú
nyihaha
vauvau
múúú :-)
múúú :-)
```

Most csak két tehenünket illettük jutalommal, de a dicséretet osztó ciklus ennél sokkal többre képes. Ha a `Dolgozo` interfészt megvalósító osztályok tömbje, a `dolgozok` 1000 elemből állna, és akár az összes objektum más-más osztály példánya volna, akkor is működne az alkalmazás ennek a programrészletnek a módosítása nélkül. Az interfész ugyanis biztosítja azt az egységes felületet, hogy mindnek van `jutalmaz()` metódusa.

Egy hétköznapi példa

Mivel elképzelhető, hogy az Olvasók egy része nem teljes munkaidőben foglalkozik lovakkal, kutyákkal, és tehenekkel, egy gondolatkísérlet erejéig próbáljunk meg egy, az üzleti életre végletekig kihegyezett világunkhoz közelebb álló problémát modellezni. Tegyük fel, hogy egy olyan alkalmazás készítése a célunk, amelyben egy cég munkavállalóit kell kezelni. Erre a jelen cikksorozaton edzett programozó azonnal rávágja, hogy természetesen minden alkalmazott önálló objektum kell, hogy legyen. Ez egy jó megközelítés, hiszen minden dolgozónak van neve, életkora, munkaköre, fizetése, amik tekinthetők az objektum tulajdonságainak. Emellett minden dolgozónak lehet feladatot kiadni, szabadságra küldeni, illetve cégünk humán erőforrás politikájától függően bért emelni, illetve csökkenteni. Ezek pedig az objektumon végezhető műveletek.

Eldöntöttük tehát, hogy mi fog objektumnak számítani. Noha a válasz itt kézenfekvő volt, a tervezésnek ez a lépése nem mindig ilyen könnyű, ezért mindig éljünk a legnagyobb alkotói szabadsággal, és gondoljuk meg, mit nyerhetnénk, ha másképp fogalmaznánk meg a modellt. Adott esetben elképzelhető, hogy az egyes feladatköröket érdemes objektumoknak tekinteni. Most azonban maradjunk az egy dolgozó - egy objektum meggondolásnál, ami a valóságnak is egy jó képe. Miután az objektumok osztályok példányai, meg kell fontolni, hogy milyen osztályhierarchiát szeretnénk kialakítani. Elérkeztünk az első buktatóhoz. Elsőre eszünkbe juthat, hogy van a vállalatnak egy belső struktúrája, ahogy a dolgozók egymás fölé vannak rendelve, használjuk ezt osztályhierarchiának. Az objektumközpontúság tényleg egyfajta leírása a világnak, amiben élünk, de nem szabad az életben használt modelleket gondolkodás nélkül ráerőltetni alkalmazásainkra.

Több, mint valószínű, hogy alkalmazásunkban egy főnök objektumon többféle műveletet végezhetünk, mint egy egyszerű dolgozón. Visszatérve a humán erőforrás politikához, lehet, hogy fizetést csak a főnök objektumok esetén lehet emelni. Ebben az esetben hibás lenne a vállalat hierarchiáját átvenni, hiszen ha a dolgozó kiterjeszti a főnököt, akkor neki is önműködően meglesz az összes metódus az ősosztályból.

A legelső megoldási kísérlet kudarca ellenére érezhetjük, hogy nem járunk messze az igazságtól. Van valami köze a kettőnek egymáshoz, csak nem ugyanazok. Gondoljuk meg, hogy egy vállalat szervezeti felépítésének diagrammján a felelősség lentről felfelé nő, a lehetőségek is ebben az irányban szélesednek. Egy ***UML*** diagramon az ősosztály jellemzően az öröklő osztály felett áll, ami pont fordított irányt mutat az előzőhöz képest. Egy, a ***Balaton*** partjáról érkező szellő azt súgja, hogy fordítsuk meg a szervezeti felépítés diagrammját, és így fejjel lefelé nézve pont megadja a megoldást.

Így azonban többszörös öröklődésbe futunk, ami, mint tudjuk nem megengedett Java-ban. A megközelítés tehát jó, csak meg kell fontolnunk, mit tehetünk meg interfésznek, és mit valódi osztálynak. A sorozatban most először egy próba erejéig megteszem ezt házi feladatnak. A probléma nem túl konkrét, így ki-ki saját ízléséhez mérten bonyolíthatja. E-mailben várom tehát a különféle dolgozók fizetésének, korának, szabadsága időpontjának kiíratását.

Ha a kedves olvasó elakadt valahol, akkor is írjon bátran, szívesen segítek.

Ha már ennyire szép formájú megjegyzéseket készítettünk eddig, következő hónapban kipróbáljuk a ***javadoc***-ot, és próbára tesszük a ***Java*** kivételkezelését is.

Fülöp Balázs (bigwig42@gmail.com)
21 éves, imádja a Túró Rudit, a Debian Linuxot és a teheneket. Kedvenc írója Slawomir Mrožek. Leginkább a számítógépes hálózatok biztonsága érdekli. A BME VIK műszaki informatikus szak hallgatója.