Kávéfőzés lépésről lépésre (1. rész)

Lubickolás Java partjainál, avagy programozzunk divatosan.

Ez a cikk egy olyan sorozatnak az első darabja, amely azoknak szól, akik egy keveset programoztak már C-ben, és szeretnék megtanulni, mit is jelent ez a manapság agyondicsért objektumközpontúság. A szükséges C előismeret igazán alapszintű, ezért aki még soha semmilyen programnyelvvel nem találkozott, kis töprengés árán annak is megérthetőek az alábbiakban közölt példák. Csupán egy kis türelemre, és egy adag kávéra lesz szükség, a többiről én gondoskodom.

Az objektumközpontúság egy programozási módszertan. Iránymutató gondolatok és elvek összessége, aminek elsődleges célja az, hogy a programozó munkáját megkönnyítse. Tudatosságra tanít, aminek a betartásával mind magunk, mind mások számára átlátható kódot, könnyen továbbfejleszthető programot kapunk. Talán azonnal érthető, hogy egy nagyobb szoftver készítésekor csapatjátékosként szempont a könnyen emészthető forráskód. Amit a legtöbben csak a saját kárukon tanulnak meg, az a másik fele. Saját magunkra is gondolnunk kell akkor, amikor programot írunk, mert eltelik egy hónap, egy év, akár több is, és egy sebtiben összecsapott kódról fogalmunk sem lesz, hogy mit takar.

Először is el kell, hogy szomorítsam az elméletek embereit: ez a sorozat nem a módszertanról, hanem annak gyakorlati alkalmazásáról fog szólni. Az elv legfontosabb fogalmait a Java szemüvegén keresztül fogom bemutatni. Azért erre a programozási nyelvre esett a választás, mert érzésem szerint a manapság széles körben használt objektumközpontú nyelvek közül ez a leginkább letisztult megvalósítás, és igen sok támogatásra lelhetünk az interneten bevezetők, kézikönyvek és fórumok formájában.

Mielőtt még belevágnánk a munkába, vessünk egy pillantást az alcímre, és az első pár mondatra. Noha nem merném állítani, hogy a szójátékok koronázatlan királya volnék, ez a rövid bevezető akkor is árulkodó. A következőkben használt programozási nyelv Java szigetéről kapta a nevét, amely igen híres kávéültetvényeiről. Így többen komoly valóságalapot tulajdonítanak annak a legendának, miszerint a készítők a nyelv kifejlesztése közben elfogyasztott kávé mennyisége okán hódoltak a névadással a szigetnek. Vágjunk is bele! A ***Sun***-féle ***Java*** megvalósítást fogjuk használni, mivel ez tekinthető a hivatalos és kiforrott változatnak.

Telepítés

Minden, ***Java***-val kapcsolatos igényünk kielégítést nyerhet a ***Sun Java*** oldalán (➲ ***http://java.sun.com/***), legyen szó könyvekről, ingyenes leírásokról, példaprogramokról, vagy magáról a fejlesztői csomagról. Ezért érdemes egy kicsit elidőzni az oldalon, esetleg feliratkozni a fórumra, ha komolyak a szándékaink a programozással kapcsolatban. Amire a cikk példáinak kipróbálásához szükség lesz, az egy ***JDK 5.0***.

A rövidítés a ***J2SE Development Kit (fejlesztői csomag)*** kifejezést fedi, amit még mindig lehet tovább boncolgatni. A ***J2SE*** jelentése ***Java 2 Standard Edition***. Hogy mi köze a 2-nek az 5.0-hoz, azt néhány további változatszám ismertetésével tudom csak érzékeltetni. Az 1.3-as változat óta a ***Sun Java 2*** néven is emlegeti szeretett gyermekét. Ezt követte az ***1.4***-es, amire még mindig illett a ***Java 2*** jelző, majd kisvártatva megérkezett az ***1.5***-ös változat. Ez már egyszerre ***Java 2***, ***1.5***, sőt mi több, 5.0. Aki követte, kérem e-mailben jelentkezzen.

Miután megbarátkoztunk a ***Sun*** változatszámkezelési logikájával, töltsük le a ***Java***-t az alábbi címről:
➲ ***http://java.sun.com/j2se/1.5.0/download.jsp***. Itt többféle összeállítással találkozhatunk. A ***NetBeans IDE + JDK 5.0*** egy olyan csomag, ami a fordítón kívül tartalmaz egy ***integrált fejlesztői környezetet*** (***Integrated Development Environment***) is. Ez gyönyörű, nagyon kényelmes, viszont kisebb programokhoz felesleges és egy kicsit lassú is. A ***JDK 5.0*** cím alatt tölthető le az, amire nekünk szükségünk lesz. Ugyanerről az oldalról elérhető még a ***JRE (J2SE Runtime Environment)***, ami kizárólag a futtatókörnyezetet jelenti. Erre azért érdemes odafigyelni, mert a ***Java*** kapcsán sokat hangoztatott felületfüggetlenség, azaz hogy szinte bármilyen operációs rendszer alatt módosítás nélkül fut ugyanaz az alkalmazás, azzal a kitétellel teljesül, hogy ez telepítve van. A cikk végére a kedves olvasó már elkészítette az első Java programját. Ha szeretné ezt másoknak is megmutatni, hívja fel a figyelmüket a futtatókörnyezet telepítésének szükségességére.

Meg kell még említenem, hogy a teljes ***API (Application Programming Interface)*** leírás szintén letölthető, ami erősen ajánlott, ha programozás közben nem szeretnénk újból feltalálni a kereket. Erre még később visszatérünk. Ami viszont minden ***Linux***-hívő szívét megmelengeti, az ezután jön: a teljes forráskód úgyszintén szabadon letölthető. Ennek letöltése a nagyon sok szabadidővel rendelkező Olvasóknak ajánlott.

Térjünk vissza arra, amiért meglátogattuk az oldalt. Töltsük le tehát a legújabb fejlesztői csomagot. A cikk írásának pillanatában ez a ***JDK 5.0 Update 3*** nevet viseli. A licensz szerződés elfogadása után már csak egy kattintásra vagyunk a hőn áhított csomagtól. ***Linux*** egy önkicsomagoló ***.bin*** állomány érhető el, ***RPM*** és sima változatban. A sima itt azt jelenti, hogy minden csomagkezelő nélkül csak beleömleszti egy könyvtárba a fájlokat. Mindkét változat 45 MB körüli méretet képvisel.

A nekünk jobban tetsző változat letöltése után egy grafikus telepítő lépésről lépésre végigvezet a szükséges lépéseken. Természetes igényként merülhet fel többünkben, hogy ne kelljen rendszergazdai jogosultsággal használni a grafikus felületet. Erre szolgálna a `-console` kapcsoló, amire azonban én a következő hibaüzenetet kaptam:

```
The wizard cannot continue because of the
➥following error: Invalid command line option:
➥console is not supported (1001) (403)
```

Könnyen elképzelhető, hogy ezt a hibát azóta kijavították. Ha nem így lenne, kénytelen kelletlen a grafikus telepítővel kell megküzdenünk a fejlesztői csomagért. Mindkét esetben szükséges még egy utolsó lépés a kényelmes használat érdekében. Miután több, mint valószínű, hogy nem egy olyan könyvtárba telepítettük a `java`, illetve a `javac` programokat, amely benne lenne a `PATH` környezeti változóban, érdemes ezekre egy szimbolikus hivatkozást létrehozni egy olyan könyvtárban, amit tartalmaz a `PATH`, például:

```
$ ln -s /opt/jdk1.5.0_03/bin/java
➥/usr/local/bin/java
$ ln -s /opt/jdk1.5.0_03/bin/javac
➥/usr/local/bin/javac
```

Hasonló eljárással érdemes még a `javadoc`-ra is létrehozni egy hivatkozást, mivel a későbbiekben még jól jöhet. Indulásnak azonban bőven elég ennyi.
Elég a telepítésből, próbáljuk már ki!

Első Java programunk

Először is indítsunk egy kényelmes szövegszerkesztőt. Grafikus felület alatt nyugodt szívvel tudom ajánlani a ***SciTE***-t. Ez egy nagyon gyors, ***GTK***-s eszközkészletet használó alkalmazás, ami mindent tud, amire csak szükségünk lehet, és a legtöbb terjesztés tartalmazza. Van benne színkiemelés, sorszámozás, és képes több forráskódot is kezelni egy, a ***Mozilla***-nál látott „füles” megoldással. Gépeljük tehát be az alábbi kódot a szövegszerkesztőbe:

```
public class Hello {
    public static void main(String[] args) {
        System.out.println(“Hello világ!”);
    }
}
```

Bizony, ez a jól ismert „Helló világ!” program. Mentsük el a forrásunkat ***Hello.java*** néven, majd konzolban adjuk ki a következő parancsokat:

```
$ javac Hello.java
$ java Hello
```

A `javac` a ***Java Compiler***, azaz a ***Java*** fordító. A `Hello.java` forrásból hoz létre egy ***Hello.class*** nevű, úgynevezett bájtkódot. Ez a forrás és a gépi kód között valahol félúton helyezkedik el, és ez az, amit bármelyik ***Java*** értelmező képes futtatni, legyen szó ***Windowsról***, ***Linuxról***, vagy akár egy ***Mac OS X***-ről. A második sorban meghívott `java` az előbb említett értelmező, vagy futtatókörnyezet, ahogy tetszik.
Figyelem: bár a Hello.class fájlt futtatjuk, nincs szükség a kiterjesztés megadására, a fent látott sor nem elírás!
Ez a példa a létező legrövidebb Java program. Ahelyett azonban, hogy nekiállnánk átrágni magunkat az egyes kulcsszavak jelentésén, nézzük meg, mit is jelent az osztály és az objektum fogalma az objektumközpontú szemléletben.

Tehén mondja „múúú”

A most következőkben megszólaltatunk néhány állatot. Már nagyon régóta tudjuk, hogy a tehén azt mondja „múúú”, a ló azt mondja „nyihaha”, a kutya pedig azt mondja „vauvau”. Képzeljük el, hogy programot kell írnunk, amiben ezek az állatok beszélnek. Egy objektumokat nem ismerő programozási nyelvben, például C-ben ez valahogy így nézne ki:

```
#define TEHEN 1
#define LO 2
#define KUTYA 3

void beszelj(int allat) {
    if (allat == TEHEN) {
        printf(“múúú”);
    } else if (allat == LO) {
        printf(“nyihaha”);
    } else if (allat == KUTYA) {
        printf(“vauvau”);
    }
}
```

Másképp ezt nem is igazán lehetne megoldani C-ben.
A `beszelj(int)` függvénynek az állat azonosítóját átadva a képernyőn megjelenik az állat hangja. Akkor mi a probléma ezzel? Röviden szólva az, hogy itt nem az állatok beszélnek, hanem mi utánozzuk őket.
Gondoljuk meg, mi lenne, ha az állatok beszéde nem csak egy egysoros kiíratás lenne, hanem valami bonyolultabb. A feltételes elágazások hasa megnőne, esetleg még rosszabb, létre kellene hoznunk három, szinte ugyanolyan segítő függvényt. Vagy tegyük fel, hogy nem csak a beszédüket kell kiíratnunk a képernyőre, hanem azt is, hogy mit esznek. Ez egy külön függvényt igényel, amiben pontosan ugyanezt az elágazást kell megvalósítanunk, csak más kiíratással. Felmerül a kérdés: miért kódoljuk le kétszer ugyanazt? Mellesleg mit keres a tehén kódja a lóé mellett?
Az egyik nagy ötlete az objektumközpontú filozófiának az egységbezárás. Ez az adat, és a rajta végezhető művelet egységét jelenti. Jelen esetben az adatot az állatok hangjai, a műveletet pedig a kiíratás jelenti. Próbáljuk meg általánosan megfogalmazni a kérdést. A feladatban állatok szerepelnek. Minden állatnak van egy hangja, amit felfoghatunk úgy, mint az adott állat egy tulajdonsága. Minden állat megkérhető arra, hogy szóljon meg.
Ez az általánosítás noha egyszerű szöveges leírása a problémának, rögtön megadja a megoldás módszerét. Létrehozzuk az állatok közös leírását, amiben megadjuk a tulajdonságuk típusát, és a rajtuk végezhető műveletet. Ezt a leírást hívják osztálynak. Majd minden állathoz létrehozunk egyegy példányt, melyek már egyediek lesznek abban az értelemben, hogy különböznek tulajdonságaikban. Ezeket objektumoknak hívjuk. Hozzunk létre egy ***Allat.java*** állományt az alábbi tartalommal:

```
/**
* Ez az osztaly egy allatot ir le.
* Tulajdonsaga a hangja.
```

```
* Meg lehet szolaltatni.
*/
public class Allat {

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

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

    /**
    * 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) {
       Allat tehen = new Allat("múúú");
       Allat lo = new Allat("nyihaha");
       Allat kutya = new Allat("vauvau");
       tehen.szolaljMeg();
       lo.szolaljMeg();
       kutya.szolaljMeg();
    }
}
```

Ez első ránézésre sokkal hosszabbnak tűnik a C-beli megoldáshoz képest. Valójában nem az, mert a forráskód fele megjegyzés, ami komoly fejtöréstől szabadít meg minket, ha a jövőben szeretnénk újra felhasználni a kódunkat. Továbbá egy nagyon egyszerű programnál még kevéssé érezhetőek az objektumközpontúság áldásos tulajdonságai, ez az alkalmazás méretének növekedtével azonban egyre élesebben előjön.
Miután lefordítottuk, és futtattuk az alkalmazást, próbáljuk meg lépésről lépésre megérteni, mit is csinál. Az első és legszembetűnőbb dolog a sok *-os sor. Java-ban kétféle megjegyzésjel van. A /*, illetve */ jelek között szereplő szöveget a fordító figyelmen kívül hagyja, és ez lehet több soros is. A // jel ezzel szemben csak a sor végéig „hat”, egy sortörés mindenképpen lezárja. Azért formáztuk ilyen különlegesen a megjegyzéseket, mert később így nagyon egyszerű lesz az osztályhoz leírást készíteni. Az ehhez használatos segédprogram a *javadoc*, amivel egy-két részen belül találkozunk.
Az osztály megadása a `class` kulcsszóval történik. Ezt egy név követi, ami meg kell, hogy egyezzen a forrásfájl nevével, a .java kiterjesztést leszámítva. A `class` előtt álló módosító azt határozza meg, hogy az osztály mindenki számára látható (`public`). Az osztály leírása az ezután következő blokkban van, aminek elejét a {, végét a } jelzi. Nem lehet ***Java*** programot írni legalább egy osztály létrehozása nélkül, ezért készítettünk már a ***„Hello világ!”*** példában is egy `Hello` osztályt.
Az osztály tulajdonságait szokás tagváltozóknak is hívni. Ilyen ebben a példában a `String` típusú `hang`. A String egy szövegfüzért jelképez. Láthatóságát személyesre állítottuk (`private`), ami annyit tesz, hogy csak az osztály műveletein keresztül hozzáférhető. Egy külső kód tehát pusztán azáltal, hogy elérhet egy `Allat` típusú objektumot, még nem módosíthatja közvetlenül a `hang` változóját. Megszokott gyakorlat az osztály összes tagváltozóját személyesre állítani, és csak azokhoz biztosítani beállító, lekérdező műveletet, amelyeknél ez megengedhető.
A feladat szöveges megfogalmazásából eredő megoldási módszernél említettem, hogy osztályok példányaival, azaz objektumokkal dolgozunk. Felfoghatjuk úgy is a dolgot, mint a házépítést. Az osztályleírás adja a szerkezeti vázat, az objektumok pedig az emberektől nyüzsgő, kész építményt. A példányosítás ez esetben a ház átadását jelenti.
Az ehhez kapcsolódó művelet az úgynevezett konstruktor, ami a példányosításkor fut le, és legtöbbször értelmes értékekkel tölti fel a tagváltozókat.
A konstruktor mindig az osztály nevével megegyező művelet, vagy tagfüggvény, és szükségképpen nyilvános, különben a példányosítás meghiúsulna. Jelen példában egy paraméterrel is rendelkezik, ami `String` típusú, és ezt adja értékül `hang` nevű tagváltozójának. A tagfüggvény nagyon hasonlít például egy C-beli függvényhez. Jelen esetben a fontos különbség abban áll, hogy a konstruktor, mint kitüntetett tagfüggvény, `Allat` típusú objektum létrehozásakor hívódik meg.
Létrehozunk egy másik tagfüggvényt, a `szolaljMeg()`-et, amely kiíratja a képernyőre az állat hangját. Ez a ***„Hello világ!”*** példában is látott módon történik. Utolsó tagfüggvényünk a `main(String[])`, ami a program belépési pontját jelenti. Paraméterül a futtatókörnyezettől a parancssori argumentumok tömbjét kapja, amit itt nem használunk fel.
A függvény törzsében létrehozunk három változót, és ezekhez rögtön példányosítunk is egy-egy `Allat`-ot. Majd minden objektumnak meghívjuk a `szolaljMeg()` tagfüggvényét (metódusát).
A figyelmes olvasóban bizonyára felmerült a kérdés, hogy ha egy osztály csak a szerkezeti vázát adja egy háznak, akkor azon közvetlenül nem is lehet műveletet végezni, kizárólag objektumon. Akkor hogyan jelentheti egy tagfüggvény a program belépési pontját, hiszen induláskor `Allat` típusú objektumból egy darab sincs? A válasz a metódus `static` módosítójában rejlik. Az ilyen kulcsszóval ellátott elemek ugyanis közösek az osztályra és anélkül elérhetők, hogy akár példány is létezne az osztályból.
Ennek tudható be az is, hogy képesek voltunk a képernyőre írni! A `System` ugyanis nem más, mint egy osztály. Ennek az out egy olyan tagváltozója, ami statikus. Ráadásul ez egy olyan tagváltozó, ami egy objektum, és ezért van metódusa. A `System.out.println()` elsőre végiggondolva zavarbaejtő, de ez ne rémisszen meg. Ezzel kezdődik az objektumközpontúság, ami a kezdeti nehézségek után hasznos befektetésnek bizonyulhat.

Fülöp Balázs