2.1 Osztályok

Osztályok létrehozása

Ahogy korábban már tárgyaltuk, a konkrét életbeli objektum(csoportok) formai leírása lesz az osztály. Osztályokat kell létrehoznunk ahhoz, hogy majd létre tudjunk hozni a memóriában objektumokat. Osztályokkal olyan objektumokat formalizálunk, melyek azonos tulajdonságokkal és operációkkal rendelkeznek, mint például az emberek. Sok különböz ember létezik, de mégis rendelkeznek közös tulajdonságokkal: van `név` és k `or` tulajdonságuk, valamint mindenki tudja magáról, hogy `férfi-e`. Készítsünk egy ilyen osztályt, melynek kulcsszava a `class`. Általában osztályokat külön fájlokba készítünk, melynek neve megegyezik a létrehozni kívánt osztállyal, kiegészítve a `.java` kiterjesztéssel. Tehát készítsük el a `Ember.java` fájlt:

```
public class Ember {

}
```

Az embereknek van néhány közös tulajdonságuk, amelyet UML-ben tulajdonságnak, attribútumnak hívtunk, és a `név`, `kor` , valamit a `férfi-e` t ulajdonságot szeretnénk a programunkban használni. Ezek legyenek `String`, `int` és `boolean` típusú változók.

```
public class Ember {
    String nev;
    int kor;
    boolean ferfi;
}
```

Elkészült az osztályunk, már csak a viselkedést kellene valahogy a forráskódban is reprezentálni. Erre metódusok lesznek a segítségünkre, melyeket az osztályba írunk bele, és ezek azt jelentik, hogy az adott osztályból létrejöv objektumok milyen viselkedéssel rendelkezhetnek. Például, az emberek tudnak köszönni, ez operációjuk. Készítsük el a `koszon` metódust.

```
public class Ember {
    String nev;
    int kor;
    boolean ferfi;

    public void koszon(){
        System.out.println("Szia! " + nev + " vagyok és " + kor + " éves, mellesleg " + (ferfi ? "férfi." :
"n."));
    }
}
```

Az így elkészült osztályból objektumokat készítünk, ezt **példányosításnak** hívjuk. Minden egyes objektum pontosan egy osztály példánya, azonban egy osztályból sázmtalan objektumpéldány is létrehozható. Hozzuk létre Józsit a main függvényben a `new` operátor segítségével.

```
public static void main(String[] args){
    Ember jozsi = new Ember();
    jozsi.koszon();
}
```

Fordítsuk, majd futtassuk le a programot. Ennek kimenete: `Szia! null vagyok és 0 éves, mellesleg n`. Nem éppen lett Józsi, még férfi sem igazán. Ennek az az oka, hogy az osztályban lév adattagok nincsenek rendesen inicializálva, a típusokhoz tartozó default értéket vették fel (elz gyakorlat). Ahhoz, hogy ezt megfelel módon megtessük, konstruktort kell létrehoznunk.

Konstruktor

Ez a speciális függvény fogja létrehozni a példányokat (objektumokat) az osztályokból, beállítani az osztályban deklarált változókat a konkrét, adott objektumra jellemz értékekre. Ilyen alapból is létezik (hiszen az elbb mi sem írtunk ilyet), de mi paramétereseket is megadhatunk, ami segítségével gyorsan és könnyen tudjuk az adott objektumot inicializálni. Megjegyzend, hogy ha mi készítünk **bármilyen** konstruktort, akkor a fordító nem készít egy default, paraméter nélküli konstrukotrt (ahogy azt tette korábban).

- A konstruktor nevének meg kell egyezzen az osztály nevével
- Visszatérési értéke/típusa nincs (nem is lehet!)
- Új objektum létrejöttekor fut le

```
/*
default konstruktor

Ez alapból létezik, nem muszáj kiírni, hacsak nem szánunk neki speciális szerepet.
*/
public Ember() {}

/*
paraméteres konstruktor

minden esetben létre kell hoznunk (ha szeretnénk paraméteres konstruktort).
*/
public Ember(String nev, int kor, boolean ferfi) {
      this.nev = nev;
      this.kor = kor;
      this.ferfi = ferfi;
}
```

`this` kulcsszó - az objektum saját magára tudni hivatkozni vele, ennek akkor van gyakorlati haszna például, amikor egy metódusban szerepl paraméter neve megegyezik az osztályban deklarált adattag nevével, akkor valahogy meg kell tudnunk különböztetni, hogy melyik az objektum tulajdonsága, és melyik a paraméter.

A paraméteres konstruktorral könnyen, egy sorban létrehozhatjuk és értéket is adhatunk a példányoknak, ami energia- és helytakarékos módszer a paraméter nélkülihez képest.

Az így kiegészített osztály után hozzuk létre mostmár valóban Józsit.

```
public static void main(String[] args){
    Ember jozsi = new Ember("Jozsi", 20, true);
    jozsi.koszon();
}
```

A program kimenete: `Szia! Jozsi vagyok és 20 éves, mellesleg férfi`.

Láthatóságok

A láthatóságok segítségével tudjuk szabályozni adattagok, metódusok elérését. Ugyanis ezeket az objektumorientált paradigma értelmében korlátozni kell, kívülrl csak és kizárólag ellenrzött módon lehessen ezeket elérni, használni. Része az implementáció elrejtésének, azaz akár a programot, akár az osztályt úgy használják kívülrl a felhasználók, hogy pontosan ismernék annak mködését.

Eddig ismert láthatóságok:

`public` - mindenhonnan látható

`private` - csak maga az osztály látja

nem írtunk ki semmit - "friendly" láthatóság, csomagon belül `public`, csomagon kívül `private`

A késbbiekben majd megismerkedünk a 3. láthatósági módosítószóval is.

`protected` - a csomag, az osztály és az azokból származtatott gyermekosztályok látják

Getter/Setter

Az adattagok értékeinek lekérésére (getter), valamint inicializálás utáni módosítására (setter) használjuk. Ezek összefüggenek azzal, hogy egy objektum adatait csak ellenrzött módon lehet lekérni, illetve módosítani, ezeken a függvényeken keresztül. Általában csak simán lekérésre, és beállításra használjuk, de ide tehetünk mindenféle ellenrzéseket is például.

Ezekkel tulajdonképpen elrejtjük a változókat és akár még módosíthatunk is az értékeken lekéréskor vagy megadáskor (mondjuk setternek megadhatjuk, hogy ne hagyja 0-ra állítani a változó értékét (mondjuk például egy osztás nevezjét))

Hagyományos elnevezése: `get + adattag neve` illetve `set + adattag neve`.

```java
//getter
public String getNev() {
    return this.nev;
}

//setter
public void setNev(String nev) {
    this.nev = nev;
}

//setter
public void setKor(int kor) {
    if (kor > 0) {
        this.kor = kor;
    } else {
        System.err.println("A kor csak pozitív szám lehet!");
    }
}
```

Boolean értékek esetében a getter függvényt általában `is + adattag neve` formában szoktuk elnevezni.

```java
/*
Kiegészítés: boolean érték gettere
A boolean érték gettert get kulcsszó helyett is-zel szokás jelölni.
*/
public boolean isFerfi(){
    return this.ferfi;
}
```

Az Eclipse kedvesen segít ezek elállításában is (de a ZH-n ki kell írni ket), a munkaterületen jobb klikk és **Source > Generate Getters and Setters...** menüpontban találjuk meg az ide kapcsolódó varázslót.

toString metódus

Írassuk ki az elkészült `jozsi` nev, ember típusú objektumunkat:

```java
System.out.println(jozsi);
```

A program kimenete valami hasonló: `Ember@677327b6`. Ez nem túl szerencsés, pláne nem olvasható. Ezt megoldhatjuk úgy, hogy az `Ember` os ztály összes adattagját egyesével kiírogatjuk, ám ez macerás lehet, ha több helyen is szeretnénk ezt a kiíratást használni.

Ha minimális munkával szeretnénk emberi nyelv leírószövegeket adni az objektumról minden esetben, amikor például átadjuk a kész objektumot a `System.out.println` metódusnak, akkor felül kell definiálnunk a `toString()` metódust. Ez egy `public` láthatóságú, `String`-gel visszatér metódus.

```java
public String toString() {
    return "Ez egy ember, neve " + this.nev +
        ", szuletett " + this.szuldatum +
        ", ferfi=" + this.ferfi;
}
```

Ami ugyanolyan mintha egy sorba írtuk volna a return-t, csak így nem fut ki a sor a végtelenbe. Ezek után nincs más dolgunk, nézzük meg mi történik a következ sor hatására:

```java
System.out.println(jozsi);
```

Ennek eredménye: `Ez egy ember, neve Jozsi, kora 20, ferfi=true`

Az Eclipse kedvesen segít ezek elállításában is (de a ZH-n ki kell írni ket), a munkaterületen jobb klikk és **Source > Generate toString()...** menüp ontban találjuk meg az ide kapcsolódó varázslót.

Static

Adattag

Ugyanazon a helyen tárolódik a memóriában, az összes példány esetében ugyanaz lesz az értékük, gyakorlatilag az osztály összes példánya osztozik rajtuk, példányosítás nélkül is hivatkozhatunk rájuk. Tulajdonképpen ezek az adattagok nem az objektumokhoz, hanem az osztályhoz tartoznak.

Gyakorlati jelentsége lehet például akkor, ha egy változóban szeretnénk letárolni egy olyan értéket - például a létrehozott objektumok

darabszámát - amelyeknek azonosnak kell lenniük minden objektumpéldány esetén, st, akár példányosítás nélkül is értelmesek.

Hivatkozni rájuk `Osztály.adattag` módon lehetséges, amennyiben látható az adattag az adott helyrl.

Metódus

Ezek a metódusok gyakorlatilag nem az objektumokhoz, hanem az osztályhoz tartoznak. Objektum példányosítása nélkül is meghívhatóak.

Ezekre példát már sokat láttunk, ilyen többek között a `public static void main()`, `Integer.parseInt()` metódusok.

Statikus metódusban csak statikus adattagok és a kapott paraméterek használhatóak, hiszen ezek nem kapcsolódnak egyetlen objektumhoz sem, azok létezése nélkül is meghívhatunk egy statikus metódust.

Ezek a metódusokat nem lehet felüldefiniálni, de errl a késbbiekben fogunk tanulni.

Final

Adattag

A `final` adattag kezdeti értékét nem változtathatjuk meg a futás során. Ehhez persze adni kell kezdértéket, vagy a létrehozás helyén (pl.: `final int TOMEG = 10`) vagy pedig a konstruktorban. Ha a final értéken változtatni próbálnánk meg, akkor fordítási hibát kapunk. Próbáljuk ki a következt.

```
private final int tomeg = 10;
tomeg=20; // Fordítási hiba!!
```

Metódus

Ezeket a metódusokat a gyerekosztályban nem lehet felüldefiniálni. Errl a késbbiekben fogunk tanulni, de itt érdemes visszaemlékezni az UML-nél tanultakra.

Osztály

Nem lehet gyerekosztálya.

Konstans

Nincs rá külön kulcsszó, az eddigiekbl merítve azonban könnyen kitalálható: valódi konstans létrehozása a static és final kulcsszavak együttes használatával, hiszen az így létrejöv változó: statikus lesz, bármely objektumpéldány esetén ugyanazt az egy konkrét adatot látjuk/módosítjuk; valamint a final miatt a kezdérték nem változtatható meg.

Konstans változók nevét általában csupa nagybetvel szoktuk írni, szóhatárnál aláhúzást teszünk.

```
public class Alma {
 public static final int ALMA_TOMEG = 10;
}
```

Ennek elérése kívülrl:

```
int almaTomege = Alma.ALMA_TOMEG;
```

Garbage collection - Szemétgyjtés

Objektumok élettartama Java-ban: (élettartam = objektum létrejöttétl a memória felszabadításig) Amikor létrehozunk egy objektumot a new kulcsszóval, az objektum a heapen jön létre (ellentétben a primitív típusokkal). A memória felszabadítása automatikus Javaban, a garbage collector (Szemétgyjt) végzi el a feladatokat. A Java szemétgyjtrl bvebben ezen, ezen és ezen az oldalon olvashatunk. Ha egy objektumot nem használunk tovább, beállíthatjuk `null`-ra, ez a kulcsszó jelöli, hogy az adott referencia nem hivatkozik egyetlen objektumra sem (pl. `jozsi = null;`) Garbage collector hívása manuálisan (nem biztos hogy lefut):

```
System.gc();
```

Lehetségünk van az osztályainknak egy `finalize()`-nak nevezett metódust készíteni. Ez a metódus akkor fog lefutni, amikor a szemétgyjt felszabadítja a feleslegesnek ítélt memóriát, és egy adott objektum, amit már nem használunk, törldni fog. Azt természetesen nem tudni, hogy mikor fog lefutni, vagy hogy le fog-e egyáltalán. A metódus célja az objektum által használt valamilyen erforrás felszabadítása (erre késbb látunk példát).

Feladatok

1. Hozzunk létre egy 7*10-es `int`-eket tartalmazó tömböt, töltsük fel ket, az alábbi séma szerint: `tömb[x][y] = x*y;` (pl.: tömb[5][8] = 40;)
2. Hozzunk létre egy karakter tömböt `'t' 'e' 'l' 'e' 'f' 'o' 'n'` karakterekkel. Másoljuk egy új tömbbe a `'l' 'e'` karaktereket !
3. Írj egy osztályt, amely téglalapot reprezentál, annak oldalhosszait tárolja. Készíts neki konstruktort, amely az oldalakat inicializálja. Írj az osztálynak még egy konstruktort, amely csak egy paramétert vár és amellyel négyzetet lehet létrehozni. Készíts metódusokat a kerület és terület kiszámítására. Írj egy másik osztályt, amely futtatható (van benne main függvény), és a parancssori paramétereknek megfelelen létrehoz téglalap objektumokat a Téglalap osztályból, és kiszámolja a Téglalapok területének és kerületének átlagát. Példa a main függvényre: számhármasok, az els szám jelöli, hogy 1 vagy 2 paraméterbl inicializálódik a téglalap, azaz négyzetet vagy téglalapot szeretnénk létrehozni, majd az ezt követ 1 vagy 2 szám tartalmazza a téglalap oldalhosszait. `java TeglalapMain 1 5 2 10 22 2 9 8 1 100`. Ennek jelentése: Elször létrehozunk egy négyzetet, 5-ös oldalhosszal, majd téglalapot 10, 22 oldalhosszakkal, majd megint téglalapot 9 és 8 oldalhosszakkal, majd egy négyzet, melynek 100 az oldalhossza.

Kocsmaszimulátor part 1:

Bvítsük ki a már létez Ember osztályt egy privát *pénz*, és *részegség* int, és egy *kocsmában*boolean változókkal. Legyen egy új konstruktor, ez fogadjon a már meglév paramétereken kívül egy pénz paramétert is, amit állítson be az Ember pénzének. A részegség 0, a kocsmában false legyen alapértelmezetten. Legyen az Embernek egy `iszik(Kocsmáros kocsmaros)`metódusa, ami egy Kocsmárost vár majd. Ha ezt meghívják, akkor ha az illet a kocsmában van, fogyjon 1 a pénzébl, njön 1-gyel a részegsége, generáljon 1 koszos poharat, és adjon 1 pénzt a kocsmárosnak, akit paraméterül kapott. Majd látjuk, hogy a poharat hova kell eltárolni, és mi a Kocsmáros. Ha nincs a kocsmában, akkor írjon ki egy üzenetet errl. Legyen egy `alszik()`metódusa is, ami nullázza a részegséget és kiírja, hogy elaludt, egy `hazamegy()` metódusa, ami false-ra állítja a kocsmában változót, és egy `jön()` metódusa, ami true-ra. Ezekrl is történjen kiírás.

Legyen egy Kocsmáros osztály is. Neki is legyen privát *pénze*, amit konstruktorban is meg lehet adni. Az összes kocsmáros ugyanazokon a *kosz os poharakon* osztozzon (static), és legyen egy `elmos()` metódusa, ami csökkenti eggyel a koszos poharak számát, és kiírja, hogy elmosott egy poharat. Ha nincs koszos pohár, akkor azt írja ki.

Legyen egy `Ital` osztály is, aminek a következ privát tulajdonságai lesznek: *ár*, *alkoholtartalom*.

Az Embernek legyen egy olyan iszik metódusa is, aminek fejléce `iszik(Kocsmáros kocsmáros, Ital ital)`, azaz italt is tud fogadni. Ekkor az ital árát adja át az Ember a Kocsmárosnak 1 helyett. Az Ember részegsége az ital alkoholtartalmával njön.

Ha a részegség eléri a 40-et, akkor az Ember mindkét `iszik()` függvényénél automatikusan aludjon el.

Az összes osztály privát változóihoz legyenek getter, setter metódusok, és az osztályokhoz értelmes toString metódus.

Legyen egy main függvény, mondjuk Main nev osztályban, itt írjatok egy rövidke futtatást, amiben eljátszogattok egy kicsit az emberekkel, bemutattok pár esetet, igyanak, aludjanak, stb...