Állatverseny 2.

Vészesen közeleg az állatverseny időpontja, és a megrendelő most jött rá, hogy módosítani kellene a programunkat (a megrendelők már csak ilyenek ☺)

Hiába dugjuk az orra alá az eredeti kérését, mégpedig ezt:

„Az állatmenhely alapítvány március közepén kisállat-versenyt rendez. Mindegyik **állat** regisztrálásakor meg kell adni az állat *nevét* és a *születési évét*. Ezek a verseny során nyilván nem változhatnak. Mindegyikőjüket *pontozzák*, pontot kapnak a *szépségük*re és *a viselkedésük*re is. A *pontszám* meghatározásakor figyelembe veszik a korukat is (csak év): egy egységesen érvényes *maximális kor* fölött 0 pontot kapnak, alatta pedig az életkor arányában veszik figyelembe a szépségre és a viselkedésre adott pontokat. Minél fiatalabb, annál inkább a szépsége számít, és minél idősebb, annál inkább a viselkedése. (Ha pl. 10 év a maximális kor, akkor egy 2 éves állat pontszáma: (10-2)*a szépségére adott pontok + 2* a viselkedésére kapott pontok.)”

ő közli, hogy bocsánat, tévedett, és kéri a módosítást. Kiderült ugyanis, hogy a versenyen kutyák és macskák vesznek részt, nem teljesen egyforma feltételekkel. A regisztrációra és a pontozásra való előírás nagyjából marad, de ezeket a módosításokat kéri:

Kutyák esetén a gazdához való *viszony*t is pontozzák. Ez hozzáadódik a szépségért és viselkedésért kapott pontokhoz, de ezt a viszony-pontot a verseny előtt adja a zsűri. Ha nincs viszony-pontja, akkor a végső pontszáma nulla lesz.

Mivel kutyák és macskák együtt szerepelnek, ezért csak olyan macskák versenyezhetnek, akiknek *van* macskaszállító dobozuk. A doboz létét már a regisztráció során be kell jelenteni, de a verseny pillanatáig módosítható. Ha a verseny pillanatában nincs ilyen doboz, akkor az ő végső pontszáma is nulla lesz.

Bár a megrendelő csak ennyit mondott, de azért figyelmeztessük rá, hogy így nem lehet egyértelműen azonosítani az állatokat, hiszen miért ne lehetne köztük két azonos nevű. Most állapodjunk meg abban, hogy mindegyikőjük kap majd egy rajtszámot, mégpedig a regisztrálás sorrendjének megfelelő értéket.)

A toString() metódusban adjuk meg azt is, hogy az illető állat kutya-e vagy macska.

1. Először próbáljuk ki az elkészült osztályokat két konkrét példányra.

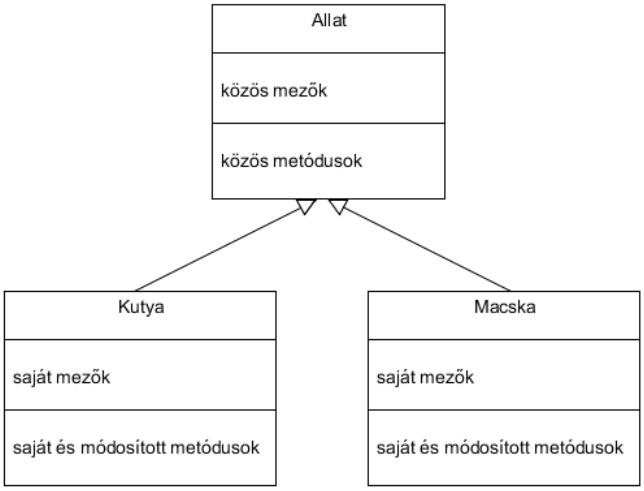
2. Utána kapcsolja össze a feladatot a korábban tanultakkal, azaz regisztráljon valahány állatot (vegyesen kutyákat és macskákat), majd versenyeztesse őket (legegyszerűbb, ha véletlen pontokat ad). A regisztráció után is és a verseny után is írassa ki az adataikat. Az adatokat az *allatok.txt* fájlból olvassa. (Fájlszerkezet: állat_neve;szül.éve, és ha macska, akkor utolsó adatként az, hogy van-e doboza (true vagy false).)

Megoldás-javaslat

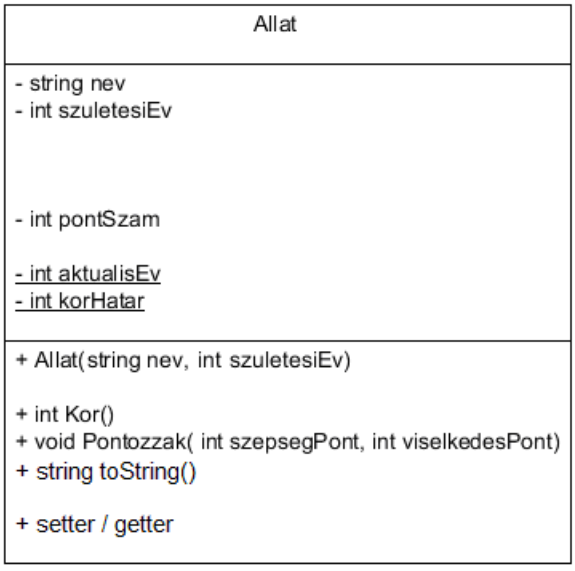
(A legtöbb program esetén több jó megoldás is lehet. Ezt se magolja be, hanem **értse meg**, **csinálja végig** a leírtak alapján, majd próbálja meg önállóan is.)

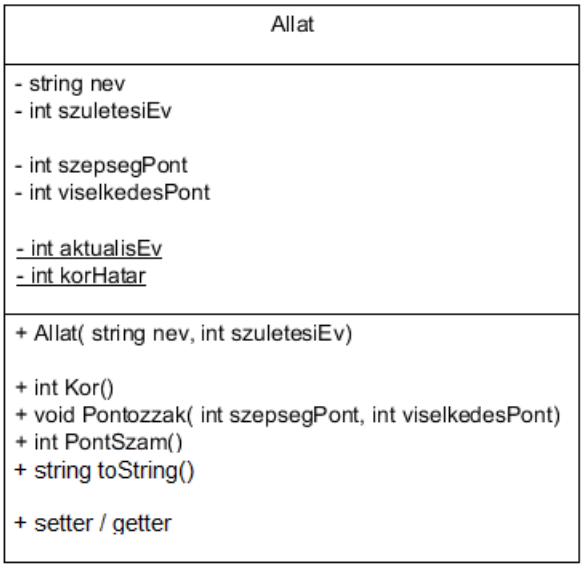
Természetesen minél kevesebb plusz munkával szeretnénk módosítani a már meglévő megoldást, ezért nem kezdjük elölről az egészet, hanem végiggondoljuk, mit lehetne felhasználni a korábbiból. No és persze, a kódismétlés is kerülendő. Ezért célszerű igénybe venni az öröklődés fogalmát. Tudjuk ugyanis, hogy a kutya is, macska is állat, és vannak olyan feltételek, amelyek mindkettőre vonatkoznak. Ezeket célszerű csak egyszer megfogalmazni. A közös tulajdonságok kerülnek majd az ősosztályba (`Allat`), az utódosztályokban (`Kutya`, `Macska`) pedig csak a módosításokat kell majd leírnunk.

Az osztályszerkezet sematikus UML ábrája:



Induljunk ki tehát a már meglévő `Allat` osztályból. Erre kétféle megoldást is megbeszéltünk. Ismételjük át mindkét UML ábrát:





Mint látjuk, az a különbség, hogy a baloldaliban a pontszámot adattagként kezeltük és a `pontozzak()` metódusban számoltuk ki az értékét, a jobboldaliban pedig egy külön metódust írtunk a kiszámítására. Mindkét megoldás jó, mindkét megoldást alapul tudjuk venni, csak eltérő lesz a továbblépés (mint ahogy a kiindulás is az volt ☺).

Mindkét változatot megbeszéljük. Kezdjük a baloldalival.

A kutyának is, macskának is van neve, születési éve, mindkettő kap szépségpontot és viselkedéspontot, sőt, meg is engedjük, hogy ezeket bármikor le is lehessen kérdezni (ezért definiáljuk mezőként és nem csak egyszerű paraméterként), mindkettőnek kiszámolhatjuk a korát, és mindkettőre ugyanaz a korhatár-előírás vonatkozik. Vagyis az `Allat` osztály `nev`, `szuletesiEv`, `szepsegPont`, `viselkedesPont`, `aktualisEv`, `korHatar` mezőit nem kell (nem szabad) újraírnunk az utód osztályokban, és az életkort is ugyanúgy számoljuk, vagyis a `kor()` metódust sem írjuk újra.

Mi a helyzet a `pontozzak()` metódussal? Itt már van egy kis probléma, mert nem egyformán pontozzák őket. Ugyanakkor azonban a szépségpont és viselkedéspont alapján kiszámított rész most is egyforma. Ezt úgy lehet megoldani, hogy a pontozásnak ezt a részét a közös ősben írjuk meg, az utód osztályokban pedig hozzárakjuk az eltérő részleteket, vagyis az utód osztályban módosítjuk (felüldefiniáljuk) ezt a metódust. A `Macska` osztályban nincs is gond, hiszen csak annyi a módosítás, hogy egy feltételtől tesszük függővé, hogy meghívjuk-e az ősosztály `pontozzak()` metódusát, de a `Kutya` osztályban a kiszámítás módja is változik, az eddigi értékhez hozzáadódik még egy pontszám. Igen ám, de minden adattag `private`, és nem is engedtük meg, hogy a `pontSzam` változó értékét a `pontozzak()` metóduson kívül bárki más megváltoztathassa. Most kénytelenek vagyunk engedni ebből a szigorú megszorításból, és megengedni azt, hogy az utód mégiscsak módosíthassa a `pontSzam` értékét. Emiatt a változóhoz `protected` hozzáférést kell majd rendelnünk.

Lássuk, mi új lesz a `Kutya` osztályban!

Lesz két új mező, az egyik a `gazdaViszonyPont` (ehhez nem feltétlenül kellene mezőt rendelnünk, csak akkor, ha a teljes pontszámtól függetlenül is meg akarjuk kérdezni az értékét), illetve egy logikai változó, amelyik azt mutatja, hogy kapott-e már ilyen pontot. (Ez lesz a `kapottViszonyPontot` változó.) Szükség lesz még a `viszonyPontozas()` metódusra, ez kéri be a `gazdaViszonyPont` értékét, és felül kell definiálnunk a `pontozzak()` metódust.

Gondoljuk végig a `Macska` osztályt is!

Itt kell egy logikai változó, melynek értéke mutatja, hogy van-e macskaszállító doboz (ezt `vanMacskaSzallitoDoboz` néven nevezzük). Itt is felül kell definiálnunk a `pontozzak()` metódust.

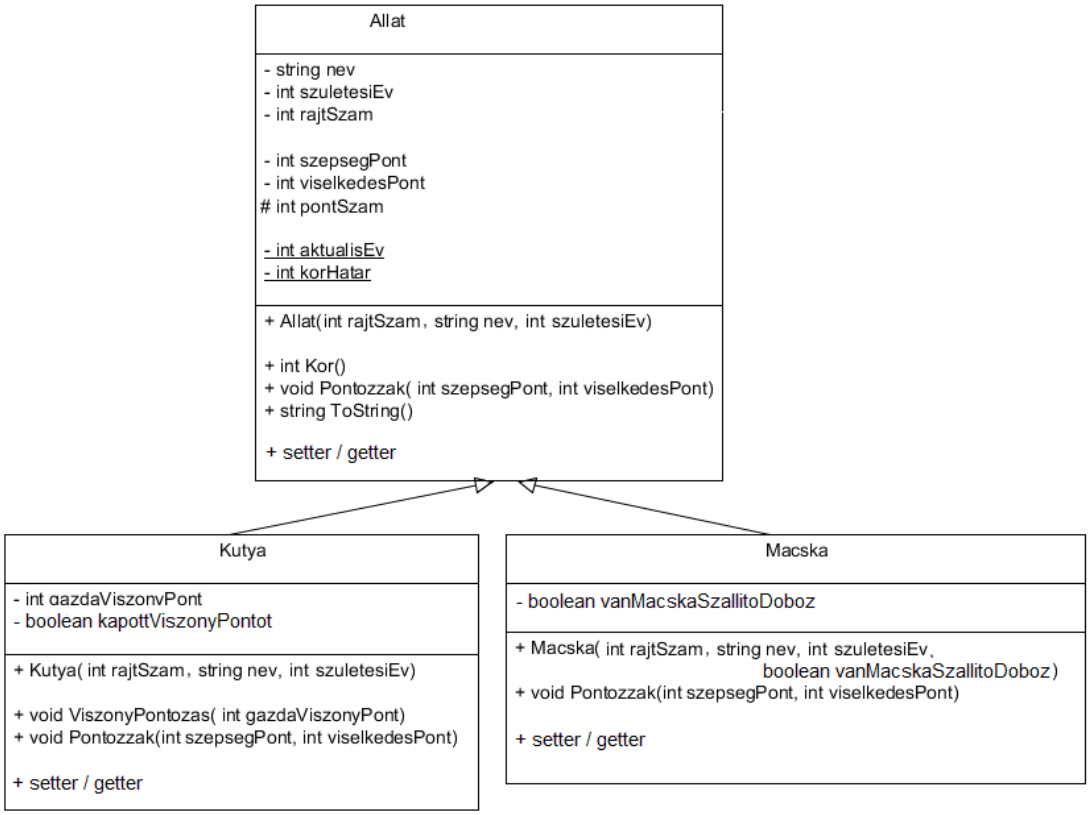
Fontos tudnunk, hogy az utód osztály konstruktorát mindig újra kell gondolnunk. A `Kutya` osztály konstruktorának ugyanazok a paraméterei, mint amelyeket az `Allat` osztályban is megadtunk, hiszen egy kutya regisztrálásakor csak ezeket az adatokat kérik. A `Macska` osztály konstruktorát azonban ki kell bővítenünk a `vanMacskaSzallitoDoboz` paraméterrel, hiszen esetükben már a regisztráció során nyilatkozni kell arról, hogy van-e ilyen doboz vagy sincs.

Kérdés még a `toString()` sorsa. A megrendelő azt is kéri, hogy az állat neve mellé írjuk ki, kutyáról vagy macskáról van-e szó. Első hallásra úgy tűnik, hogy emiatt a `toString()` metódust is át kell alakítanunk, és felül kell definiálnunk az utód osztályokban. De ha ügyes módon választunk osztályneveket, akkor egyszerűbben is megoldható a feladat. Elég, ha csak az ősosztály `toString()` metódusát alakítjuk át, mégpedig úgy, hogy a visszaadott `string`-be foglalja bele az aktuális osztály nevét is. Így `Kutya` példány esetén a kutya szót írja ki, `Macska` példány esetén a macska szót (feltéve, hogy arra is odafigyelünk, hogy alakítsa a nevet csupa kisbetűssé).

Végül ne feledkezzünk el a saját javaslatunkról, vagyis legyen minden állatnak rajtszáma. Mivel ez minden állatra ugyanúgy vonatkozik, ezért ezt most úgy oldjuk meg, hogy az `Allat` osztályban felveszünk egy újabb mezőt `rajtSzam` néven. Ez a mező a konstruktorban kap értéket, amely a versenykiírásnak megfelelően később nem változhat.

(Megjegyzés: ez a fajta megoldás nem garantálja azt, hogy a rajtszámok egyediek legyenek, úgyhogy majd ezt is megbeszéljük a megoldás során.)

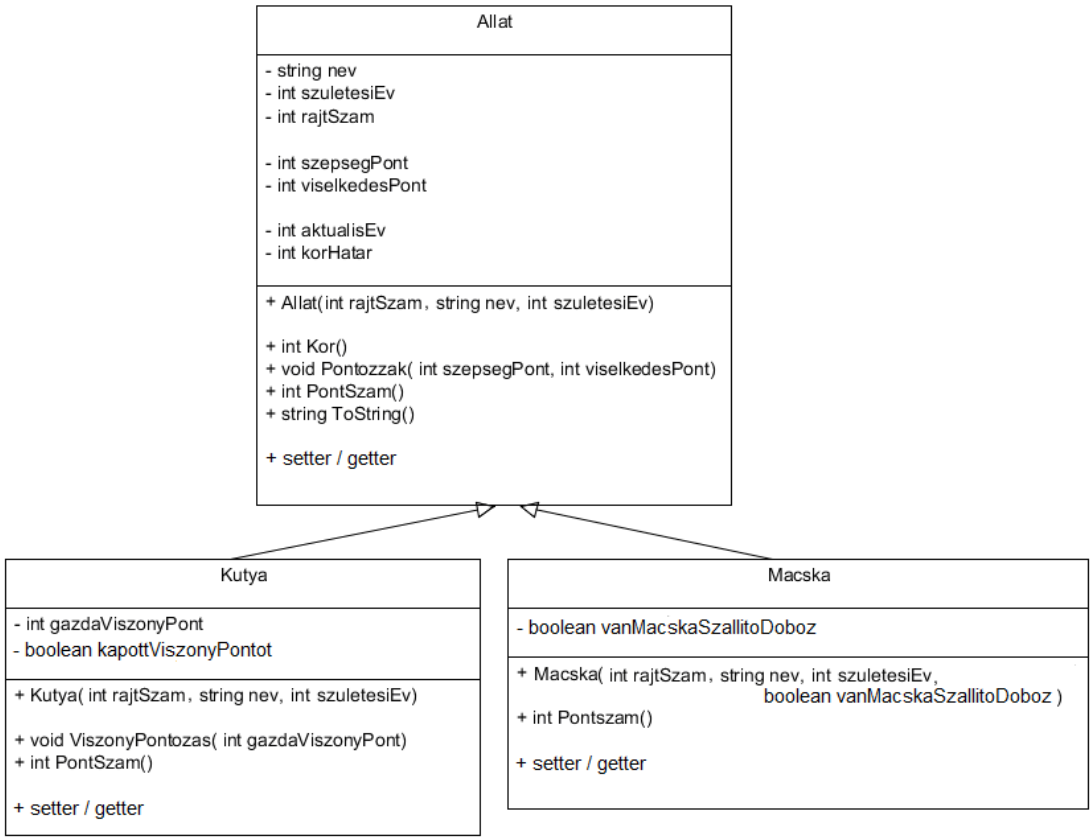
Ezek után lássuk az UML ábrát:



Láttuk, hogy egy kicsit körülményessé tette a megoldást az, hogy a pontszámot mezőként definiáltuk. Ráadásul nem is szerencsés `protected` hozzáférésűnek deklarálni egy mezőt, hiszen az nem csak az utódokban lesz elérhető, hanem csomag szinten. Vagyis nem igazán biztonságos. Ezért lehetőleg mindig a `private` hozzáférés a javasolt.

Várhatóan egyszerűsíti a kódot, ha az ősosztályban külön metódust írunk a pontszám kiszámítására (ez lesz a `pontSzam()` metódus). Ekkor az ős `pontozzak()` metódusa változatlan marad (csak arra szolgál, hogy a zsűri megadhassa a pontszámokat), és a `pontSzam()` metódust kell majd felüldefiniálnunk a `Kutya` osztályban.

Az ennek megfelelően módosított UML:



Ez utóbbi megoldást beszéljük meg részletesen.

Az `Allat` osztály megírása már nem jelenthet problémát. Ha mégis, akkor alaposan végiggondolva oldja meg újra az erre vonatkozó korábbi feladatokat.

Ennek ellenére ide kerül majd a kód, de előbb beszéljünk meg egy kis módosítást, mégpedig a rajtszám kérdését. Ha ezt a konstruktor paraméterében, azaz kívülről adjuk meg, akkor semmi garancia nincs rá, hogy nem lesz két azonos rajtszám. Az egyediséget csak belülről tudjuk megoldani, mégpedig így: a konstruktoron belül egy statikus változó értékét növeljük, és ez lesz az egyedi rajtszám értéke. Mivel a statikus változó csak egy példányban létezik, ezért minden egyes példányosításkor ezt az egy értéket növeljük, vagyis azt mutatja majd, hogy összesen hány példány létezik. Így aktuális értéke valóban tekinthető az aktuálisan létrehozott

példány egyedi azonosítójának. Természetesen sem a statikus mezőhöz, sem a rajtszámhoz nem tartozhat setter, hiszen az elrontaná az egyediséget.

A másik módosítás: a `toString()` metódus tartalmazza az osztály nevét is, ami futáskor értelemszerűen kutya vagy macska lesz.

Ezt is figyelembe véve az `Allat` osztály (setterek, getterek nélkül):

```
public class Allat {

    private String nev;
    private int szulEv;
    private int rajtszam;

    private int szepsegPont;
    private int viselkedesPont;

    private static int aktualisEv;
    private static int korHatar;
    private static int utolsoIndex;

    /** konstruktor ...6 lines */
    public Allat(String nev, int szulEv) {
        this.nev = nev;
        this.szulEv = szulEv;
        rajtszam = ++utolsoIndex;
    }

    /** Visszaadja a kiszámított életkort ...5 lines */
    public int eletKor(){
        return aktualisEv - szulEv;
    }

    /** Pontot kap ...6 lines */
    public void pontotKap(int szepsegPont, int viselkedesPont) {
        this.szepsegPont = szepsegPont;
        this.viselkedesPont = viselkedesPont;
    }

    /** A feltételeknek megfelelően kiszámolja a pontszámot ...5 lines */
    public int pontSzam() {
        int kor = this.eletKor();
        if (kor > Allat.korHatar) {
            return 0;
        }
        return (Allat.korHatar - kor) * szepsegPont + kor * viselkedesPont;
    }

    @Override
    public String toString() {
        return nev + " " + this.getClass().getSimpleName().toLowerCase() + " "
                   + this.eletKor() + " éves";
    }
```

Megjegyzések:

1. Egy alaposztályt úgy definiálunk, hogy a lehető legáltalánosabban lehessen használni, ezért esetünkben minden mezőhöz írunk gettert, settert viszont csak az aktuális év és a korhatár értékének megadásához.

2. Mivel nincs is konkrét Allat típusú példány, csak kutya és macska van, ezért az osztályt absztraktként is deklarálhattuk volna: `public abstract class Allat`

Lássuk a `Kutya` osztályt! Ennek kapcsán két dolgot kell megbeszélnünk: a konstruktor és a felüldefiniált metódus kérdését.

Azt hogy az osztály az `Allat` osztály leszármazottja, az `extends` kulcsszóval jelöljük, a közvetlen ősre pedig a `super` kulcsszóval hivatkozunk. Esetünkben ez annyit jelent, hogy a konstruktor hivatkozik az ős konstruktorára, és teljes mértékben elfogadja az általa létrehozott példányt.

Egy kutya pontszámának kiszámításakor a gazdához való viszonyára kapott pontot is figyelembe kell venni, ezért felül kell definiálnunk az `Allat` osztály `pontSzam()` metódusát. Ezek után az osztály kódja:

```
public class Kutya extends Allat{

    private int gazdaViszonyPont;
    private boolean kapottViszonyPontot;

    public Kutya(String nev, int szulEv) {
        super(nev, szulEv);
    }

    public void viszonyPontotKap(int viszonyPont){
        this.gazdaViszonyPont = viszonyPont;
        this.kapottViszonyPontot = true;
    }

    @Override
    public int pontSzam() {
        int pont = 0;
        if(kapottViszonyPontot){
            pont = this.gazdaViszonyPont + super.pontSzam();
        }
        return pont;
    }

    public int getGazdaViszonyPont() {
        return gazdaViszonyPont;
    }

    public boolean isKapottViszonyPontot() {
        return kapottViszonyPontot;
    }
}
```

Megjegyzés: Ha nagyon merev előírás lenne az, hogy csak a viszonypontozás után kaphatja meg a másik két pontot, akkor elvileg a `pontotKap()` metódust is felül kellene írnunk, de ha elképzelhető, hogy tetszőleges sorrendben mehet a kétféle verseny, akkor nem.

A `Macska` osztály:

Az osztály konstruktora hivatkozik az ős konstruktorára, segítségével létrehoz az objektumból annyit, amennyit tud, majd kiegészíti még a `vanMacskaSzallitoDoboz` változó inicializálásával.

A feladat szerint azt, hogy van-e szállítódoboz, a pontozás pillanatában veszik figyelembe. Ez azt jelenti, hogy most nem a `pontSzam()` metódust kell felülírnunk, hanem a `pontotKap()` metódust, hiszen ez a pontozás pillanata.

```
public class Macska extends Allat{

    private boolean vanSzallitoDoboz;

    public Macska(String nev, int szulEv, boolean vanSzallitoDoboz) {
        super(nev, szulEv);
        this.vanSzallitoDoboz = vanSzallitoDoboz;
    }

    @Override
    public void pontotKap(int szepsegPont, int viselkedesPont) {
        if (vanSzallitoDoboz) {
            super.pontotKap(szepsegPont, viselkedesPont);
        }
    }

    public boolean isVanSzallitoDoboz() {
        return vanSzallitoDoboz;
    }

    public void setVanSzallitoDoboz(boolean vanSzallitoDoboz) {
        this.vanSzallitoDoboz = vanSzallitoDoboz;
    }
}
```

Rövid kitérő: Korábban már szó volt róla, hogy az `@Override` annotáció nem kötelező (de nem célszerű elhagyni, mert információt ad a fordítóprogram számára). De ha nem kötelező, akkor honnan tudja a futtató környezet, hogy ez egy öröklött metódus? Onnan, hogy a Javaban minden metódus virtuális, vagyis a metódusok implementációja a leszármazott osztályokban felülírható. Az ilyen metódusok meghívásakor a hívásban végrehajtásra kerülő implementációt az adott objektumpéldány típusa határozza meg. Vagyis futáskor, és nem fordításkor derül ki, hogy az eredeti vagy valamelyik felülírt metódus hajtódik-e végre. Ezt nevezik kései kötésnek, és ez a polimorfizmus lényege.

Írjunk rövid teszt osztályt az eddigiek ellenőrzésére. Egyúttal azt is megbeszéljük majd, hogy mi a teendő, ha nem `Kutya` vagy `Macska` típusúnak deklaráljuk őket, hanem általánosabban `Allat` típusra. Ezt az is indokolja, hogy a feladat szerint a vezérlésben vegyesen akarjuk regisztrálni őket, vagyis olyan listába rakni, amely vegyesen tartalmaz kutyákat is macskákat. Ez úgy oldható meg, ha a listába `Allat` típusú példányokat helyezünk.

A teszt osztály eleje nagyon egyszerű:

```
public class Teszt {

    public Teszt() {
    }

    private Allat kutya;
    private Allat macska;

    @Before
    public void setUp() {
        Allat.setAktualisEv(2018);
        Allat.setKorHatar(10);
        kutya = new Kutya("Bodri", 2016);
        macska = new Macska("Cirmos", 2016, false);
    }


     @Test
    public void allatTeszt() {
        kutya.pontotKap(5, 4);
        macska.pontotKap(5, 4);

        /* Feltételezzük, hogy most a kutya pontszáma 0
           (mert nem kapott viszonypontot),
           és a macskái is ennyi, mert nincs szállító doboza.
        */
        assertTrue(kutya.pontSzam() == 0);
        assertEquals(kutya.pontSzam(), macska.pontSzam());
```

Következő lépésként azt szeretnénk ellenőrizni, hogy ha a kutya megkapja a viszony-pontot, és csak ez után pontozzák, akkor viszont már helyesen számolja a metódus a pontszámát.

Csakhogy most jön a probléma: mivel `Allat` típusra deklaráltuk, ezért a kutya példánynak nincs `viszonyPontotKap()` metódusa! Ez azonban nem jelent nagy problémát, ugyanis egy általános típust mindig lehet speciálisra kényszeríteni, vagyis esetünkben egy `Allat` típust `Kutya` vagy `Macska` típusúra, pl.: `(Kutya)kutya`. Csakhogy még mindig nem látjuk a keresett metódust, ugyanis a pont (.) operátor erősebb, mint a típuskényszerítés. Ha azt szeretnénk, hogy előbb történjen meg a kényszerítés, akkor meg kell változtatnunk a műveletek prioritását, vagyis zárójeleznünk kell: `((Kutya)kutya)`.

Végre megvan a metódus, vagyis szintaktikusan helyes ez a sor:

```
((Kutya) kutya).viszonyPontotKap(3);
```

De örömünk még korai. Most persze, a beszédes változónév miatt elég valószínűnek tűnik, hogy valóban Kutya típusú példányt kényszerítettünk, de vajon mi a helyzet, ha egy `List<Allat> allatok;` módon deklarált lista esetén írjuk azt, hogy

```
((Kutya) allatok.get(i)).viszonyPontotKap(pont);
```

Ekkor már korántsem egyértelmű, hogy a lista i-edik eleme valóban `Kutya` típusú példányt tartalmaz-e. Emiatt nagyon veszélyes a típuskényszerítés, és hacsak lehet, célszerű kerülni. De persze, nem mindig lehet. Az viszont megoldható, hogy legalább figyeljük, hogy valóban megfelelő típusú példányt akarunk-e kényszeríteni.

További magyarázat nélkül nézzük a teszt osztály folytatását:

```
        if (kutya instanceof Kutya) {
            ((Kutya) kutya).viszonyPontotKap(3);
            kutya.pontotKap(5, 4);

            assertTrue(kutya.pontSzam() > 0);

            /*
            A következő két vizsgálat egyenértékű:
            */
            assertEquals(kutya.pontSzam(), ((Kutya) kutya).getGazdaViszonyPont()
                    + kutya.getSzepsegPont() *
                                        (Allat.getKorHatar() - kutya.eletKor())
                    + kutya.getViselkedesPont() * (kutya.eletKor()));

            assertEquals(kutya.pontSzam(), 3 + 5 * 8 + 4 * 2);
        }

        if (macska instanceof Macska) {
            ((Macska)macska).setVanSzallitoDoboz(true);
            assertEquals(macska.pontSzam(), 0);

            macska.pontotKap(5, 4);
            assertEquals(macska.pontSzam(), 5 * 8 + 4 * 2);
        }
    }
}
```

A vezérlésben most nem sok újat tárgyalunk, hiszen alig kell valamit módosítanunk a korábbi megoldáshoz képest. Összesen két változtatást végzünk majd, egyrészt kicsit módosítjuk a beolvasást, másrészt pedig beillesztjük a gazdához való viszony pontozását. (Az, hogy a legjobbak meghatározását, illetve a rendezést most nem írjuk át, igencsak megkérdőjelezhető, hiszen külön kellene összehasonlítani a kutyákat és a macskákat, de most épp elég egyéb új információt vettünk, úgyhogy ezt majd egy későbbi feladatban beszéljük meg, de persze, próbálkozhat vele, ha kedve van hozzá.)

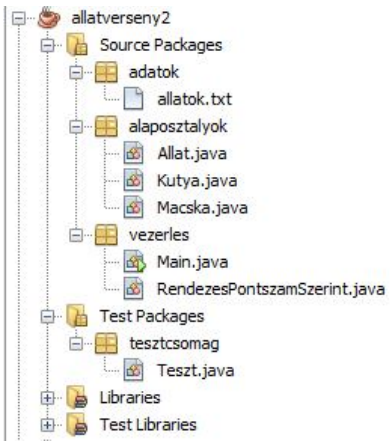
A beolvasásban két dolgot módosítunk. Egyrészt ilyen szerkezetű adatokat olvasunk:

```
Bodri;2010
Rexi;2013
Cirmos;2011;true
Pamacs;2015
Bolhazsák;2016;true
Buksi;2010
Picúr;2012;false
```

Másrészt azt is megbeszéljük, hogy hol legyen az adatfájl.

A korábbi megoldásban az adatfájlt a projekt gyökerébe helyeztük. Egyszerű is volt a beolvasás, működik is, ha a NetBeans-ből indítjuk a programot (akkor is, ha más fejlesztőkörnyezetből). Ha viszont átadjuk a megrendelőnek a programot, neki nem írhatjuk elő, hogy bármilyen fejlesztőkörnyezetet is használjon, ő a .jar kiterjesztésű fájlt kapja meg, és ezt futtatja. De ha így akarjuk futtatni, akkor – mivel a program írásakor nem jeleztük az adatfájl elérési útját –, ezért abban a mappában keresi, ahol a .jar fájl is van. Nem mindig szerencsés dolog, ha a felhasználó kezébe adjuk az adatfájlt, sokszor jobb, ha az is bele van csomagolva a .jar fájlba. (Hátránya viszont, hogy a futtatás során nem lehet úgy beleírni, hogy a módosítás is bekerüljön a futtatható állományba.)

A .jar állományba az src mappa tartalma kerül, ezért most ide rakjuk az adatfájlt. Egyúttal logikailag szétválasztjuk az osztályokat, és értelemszerűen más-más csomagba írjuk őket.



Ezen feladatok megoldása során végig egyszerű csomagneveket használunk, de a névkonvenció szerint ezeket kicsit összetettebben szokták megadni:

https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html

Ahhoz, hogy az *adatok* mappából lehessen olvasni az adatfájlt, meg kell adnunk az elérési útvonalát. Ez viszont nem lehet abszolút, hiszen nem tudhatjuk, hogy a felhasználó gépén milyen mappa-szerkezet van. Ha azt szeretnénk, hogy .jar-ból is el lehessen érni az adatfájlt, akkor inputstream-en keresztül fogjuk elérni azt. Egyúttal gondolunk a fájlban esetlegesen szereplő ékezetes karakterekre is, ezért a kódolás módját is megadjuk.

A `Main` osztály „bevezető” részlete:

```
public class Main {

    private String CHAR_SET = "UTF-8";
    private final String ADATFAJL = "/adatok/allatok.txt";
    private final int AKTUALIS_EV =
                                    Calendar.getInstance().get(Calendar.YEAR);
    private final int KORHATAR = 10;
    private final int SZEPSEG_PONT_HATAR = 11;
    private final int VISELKEDES_PONT_HATAR = 11;
    private final int VISZONYPONT_HATAR = 6;

    private List<Allat> allatok = new ArrayList<>();

    public static void main(String[] args) {
        new Main().start();
    }

    private void start() {
        statikusAdatok();
        adatBevitel();
        resztvevok();
        kutyaGazdaVerseny();
        verseny();
        eredmenyek("\nA verseny eredménye:");
        legjobbak();
        pontszamSzerintRendezve();
        eredmenyek("\nPontszám szerint rendezve:");
    }
```

Látható, hogy a `kutyaGazdaVerseny()` metódus hívásának kivételével ugyanazok a metódushívások szerepelnek a `start()` metódusban, mint a korábban megoldott feladatban, ezért most csak ezt a metódust írjuk meg, illetve a már említett `adatBevitel()` metódust.

Kezdjük az adatbevitellel. Az adatfájl helyét relatív módon adjuk meg, vagyis a hívó osztály (jelenleg a `Main` osztály) forrásának (vezerles mappa) helyéhez képest. Mivel a \ karakter már foglalt a formázáshoz, ezért útvonalmegadáshoz a / karaktert használjuk.

Egyúttal kicsit pontosítjuk a kivételkezelést is. Beolvasás után illik lezárni a megnyitott objektumokat (scanner, illetve inputstream), még akkor is, ha a szemétgyűjtő algoritmusnak hála a beolvasás ezek elmulasztása esetén is működne (ha azonban a hasonló módon működő fájlba-íráskor felejtjük el, akkor előfordulhat, hogy üres marad az eredményfájl). Mivel ezeket akkor is be kell zárni, ha helyesen futott le a beolvasás, de akkor is, ha esetleg közben valamilyen kivétel keletkezett, ezért a megnyitott objektumokat a `finally` ágon zárjuk le (illetve később mutatunk egy másik lehetőséget is)

A megbeszéltek alapján ez a metódus:

```
private void adatBevitel() {
    InputStream ins = this.getClass().getResourceAsStream(ADATFAJL);
    Scanner sc = new Scanner(ins, CHAR_SET);
    try {
        String[] adatok;
        String sor;
        while (sc.hasNextLine()) {
            sor = sc.nextLine();
            if (!sor.isEmpty()) {
                adatok = sor.split(";");
                if (adatok.length == 3) {
                    allatok.add(new Macska(adatok[0],
                            Integer.parseInt(adatok[1]),
                            Boolean.parseBoolean(adatok[2])));
                } else {
                    allatok.add(new Kutya(adatok[0],
                            Integer.parseInt(adatok[1])));
                }
            }
        }
    } catch (Exception ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    } finally {
        try {
            sc.close();
            ins.close();
        } catch (IOException ex) {
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

A 7-es Java verzió óta az objektumok lezárása automatizálható az úgynevezett „try with resources” kivételkezelő mechanizmus segítségével:

http://tutorials.jenkov.com/java-exception-handling/try-with-resources.html

Ez azt jelenti, hogy a lezárható (`Closeable`) objektumokat a `try` blokk fejében nyithatjuk meg, ekkor nincs szükség a `finally` ágra, hanem dolguk végeztével a megnyitott objektumok automatikusan bezáródnak.

A metódus így átalakított változata olvasható a következő oldalon:

```
private void adatBevitel() {

    try (InputStream ins = this.getClass().getResourceAsStream(ADATFAJL);
            Scanner sc = new Scanner(ins, CHAR_SET)) {
        String[] adatok;
        String sor;
        while (sc.hasNextLine()) {
            sor = sc.nextLine();
            if (!sor.isEmpty()) {
                adatok = sor.split(";");
                if (adatok.length == 3) {
                    allatok.add(new Macska(adatok[0],
                            Integer.parseInt(adatok[1]),
                            Boolean.parseBoolean(adatok[2])));
                } else {
                    allatok.add(new Kutya(adatok[0],
                            Integer.parseInt(adatok[1])));
                }
            }
        }
    } catch (Exception ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Ez a megoldás első nekifutásra elfogadható, illetve ha nem generált a `try-catch` szerkezet (most azért volt az, mert az előző feladatban generálódott), akkor a `catch` ágba célszerű inkább ezt gépelni: `ex.printStackTrace();` – ez ugyanis sorrendben kiírja az elkapott hibákat. Ezek közül az első olyat kell ellenőrizni, amelyiket mi írtunk. Ez nagyon sokat segít az esetleges hibakeresésben.

Ha az adatfájl elérését adtuk meg hibásan, akkor nyilván semmit sem tud beolvasni, és bár elindul a többi metódus, de adatok hiányában nem tud rendesen futni. Ha jó az elérési út, de az adatfájl valamelyik sora hibás, akkor a hibás sor előttieket beolvassa, és ezekkel az adatokkal dolgozik a továbbiakban. Azonban úgy is meg lehetne írni a beolvasást, hogy a hibás sort egyszerűen hagyja ki, és folytassa a beolvasást a következő sorral. Ekkor külön kellene választani a hibás fájlmegadás elkapását, illetve az egyes sorokban keletkezett hibák elkapását. Még egy dologra kell gondolnunk: mi van, ha egy sorban háromnál több adat van, amelyből az első kettő helyes? Akkor bizony abból azonnal `Kutya` típusú példányt hoz létre, holott nem kellene. Természetesen ezt is ki lehet védeni.

A jelzett hibalehetőségeket is figyelembe vevő megoldás:

```
private void adatBevitel() {

    try (InputStream ins = this.getClass().getResourceAsStream(ADATFAJL);
            Scanner sc = new Scanner(ins, CHAR_SET)) {

        String[] adatok;
        String sor;
        while (sc.hasNextLine()) {
            sor = sc.nextLine();
            try {
                if (!sor.isEmpty()) {
                    adatok = sor.split(";");
                    switch (adatok.length) {
                        case 3:
                            allatok.add(new Macska(adatok[0],
                                    Integer.parseInt(adatok[1]),
                                    Boolean.parseBoolean(adatok[2])));
                            break;
                        case 2:
                            allatok.add(new Kutya(adatok[0],
                                    Integer.parseInt(adatok[1])));
                            break;
                        default:
                            throw new Exception();
                    }
                }
            } catch (Exception e) {
                System.out.println("Hibás a " + sor + " adatsor.");
            }
        }
    } catch (NullPointerException ex) {
        System.out.println("Hibás fájlmegadás");
        System.exit(-1);
    } catch (IOException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Az újonnan megírandó metódus pedig:

```
private void kutyaGazdaVerseny() {
    for (Allat allat : allatok) {
        if(allat instanceof Kutya){
            ((Kutya) allat).viszonyPontotKap((int) (Math.random()*
                                                    VISZONYPONT_HATAR));
        }
    }
}
```