Állatverseny – 1.

Az állatmenhely alapítvány 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.)

Az állatok adatainak kiíratásához írjuk meg az állat nevét és pontszámát tartalmazó *toString()* metódust.

Írjon versenyeztető programot, vagyis:

- Legyen regisztráció (azaz olvassa be az állatok adatait – lehetőleg fájlból)
- Versenyeztesse őket – ez azt jelenti, hogy minden egyes állatot véletlen pontszámmal értékel a zsűri
- Verseny előtt és után is írassa ki a résztvevőket (név, pontszám)
- Állapítsa meg, hogy ki(k) a nyertes(ek).
- Írassa ki a résztvevőket pontszám szerint csökkenően.

Megoldásjavaslat

A legtöbb program esetén több jó megoldás is lehet. Ezért egyetlen programot se magoljon be, hanem értse meg, csinálja végig a leírtak alapján, majd próbálja meg önállóan is, illetve oldja meg az önálló feladatként megadott részleteket is.

Először is gondolkodjunk el a feladaton, és tervezzük meg a megoldását.

A feladat állatokkal foglalkozik. Ezért célszerű avval kezdeni, hogy definiáljuk az állat fogalmát – legalábbis azt, amit a program során állat fogalmán értünk majd.

Amikor meg akarjuk alkotni az állat fogalmát, akkor (lévén, hogy most program szintjén akarjuk ezt megfogalmazni) egy osztályt definiálunk hozzá. Ez az osztály tartalmazza majd az állat fogalmával kapcsolatos elképzeléseinket, azt, hogy milyen adatok és milyen viselkedés

(vagyis az adatokkal végzett műveletek) jellemző egy általunk elképzelt állatra. Ez még csak „fogalom”, „tervrajz”, úgy is mondhatjuk, hogy ebben az osztályban írjuk le, azaz itt határozzuk meg az állat fogalmát. Ha ezt megtettük, vagyis definiáltuk az `Allat` nevű osztályt, akkor ettől kezdve deklarálni tudunk `Allat` típusú változókat, és létre tudunk hozni ilyen típusú példányokat.

A feladat szövegének kiemelései segítenek abban, hogy végiggondoljuk, mi minden jellemez egy állatot.

Az állatot jellemző adatok (ezeket mezőknek nevezzük, néha az adattag elnevezés is használatos): Az állat *neve* és *születési éve* – ezeket azonnal, vagyis már a regisztráció során meg kell adnunk, értékük később nem változhat.

Pontozzák őket – ez már érezhetően valamilyen metódus, azaz az adatokkal végzett művelet. A szöveg megfogalmazásából az is érződik, hogy a pontozás valamilyen külső cselekvés, vagyis `void` metódus tartozik majd hozzá, amelynek során az állat pontszámot kap. Egészen pontosan: két pontszámot kap (a szépségére és viselkedésére), mégpedig nem saját magától, hanem valami külső szereplőtől, vagyis a paraméterlistán. Ezek alapján számoljuk ki a versenyben figyelembe vehető tényleges pontszámot. Ennek kapcsán kétféle módon gondolkozhatunk:

a/ Ez a metódus számolja ki a tényleges pontszámot is. Ekkor a pontszám is adattagnak (mezőnek) tekintendő.

b/ Ez a metódus csak a két adott pontszám értékének megadására szolgál, és egy külön `int` típusú metódus számolja ki a pontszám értékét.

Ez utóbbi esetben egy másik metódus is használja a pontozáskor kapott értékeket, vagyis ekkor a *szépségpont*ot és *viselkedéspont*ot is mezőként kell kezelnünk. De elképzelhető, hogy az első megoldás esetén is szükségünk lehet ezekre a mezőkre, ugyanis lehet, hogy az adott részpontok értékét is szeretnénk ismerni, ekkor a pontozási metóduson kívül is szükség van rájuk.

Először maradunk az első elképzelésnél, de később kitérünk a másodikra is. Ezek szerint most kell még egy *pontszám* mező is.

Még két további speciális mezőt kell végiggondolnunk. Ki kell számolnunk az állatok életkorát, ezért ehhez meg kell adnunk az *aktuális év*et is. Mivel ez a verseny éve, és ugyanakkor versenyzik az összes állat, ezért ez az évszám nyilván mindegyik esetén azonos, vagyis fölösleges annyiszor létrehozni, ahány állat van, elég, ha csak egyetlen példány létezik belőle. Emiatt ezt majd statikusnak (static) deklaráljuk. (De mivel ezt a programot esetleg több évben is használni szeretnénk, ezért nem konstans évszámmal dolgozunk, és nem is a gépidőt kérjük le.)

Ugyancsak ez a helyzet a *korhatár*ral is, hiszen ez is egyformán vonatkozik minden állatra.

A mezők áttekintése után gondoljuk végig az `Allat` osztályhoz tartozó metódusokat is. Ezek: Az állat *korának kiszámítása*, a *pontozás*, illetve a feladat kér még egy speciális metódust, ez az úgynevezett `toString()`, amely az állatok kiválasztott adatainak `string` formában való megadására szolgál. Erről később majd kicsit részletesebben is lesz szó.

Még egy dologra szükségünk van. Arról volt szó, hogy az állat nevét és születési évét a regisztráció során azonnal meg kell adni, nélküle nem is versenyezhet. (Akár azt is mondhatjuk, hogy a program számára enélkül nem is létezhet egy állatpéldány.)

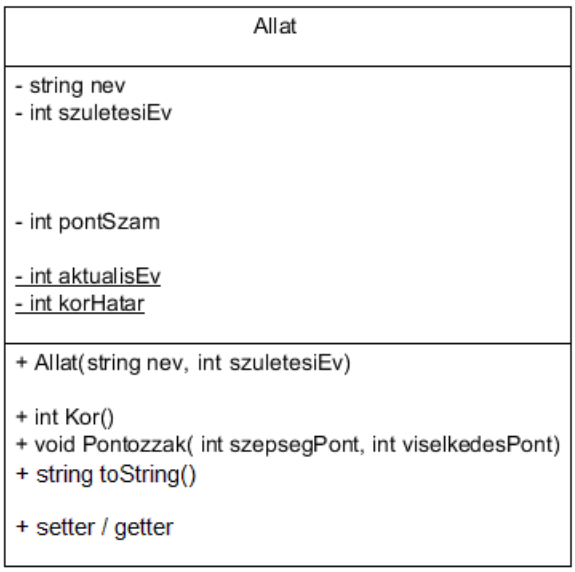
A példányt az úgynevezett konstruktor hozza létre (ez konstruálja). A konstruktor a példány létrehozásáért is, de bizonyos mezők inicializálásáért is felelős. Alapértelmezetten minden mező inicializálva van (a típusától függően 0, `false` vagy `null` az értéke), de nyilván lehetnek olyanok is, amelyeket nem ilyen kezdőértékkel akarunk beállítani. Esetünkben ilyen például a név és a születési év. Ezek a mezők majd a konstruktoron belül kapnak értéket, mégpedig azokat, amelyek a konstruktor paraméterében szerepelnek. Tehát definiálnunk kell majd egy kétparaméteres konstruktort is.

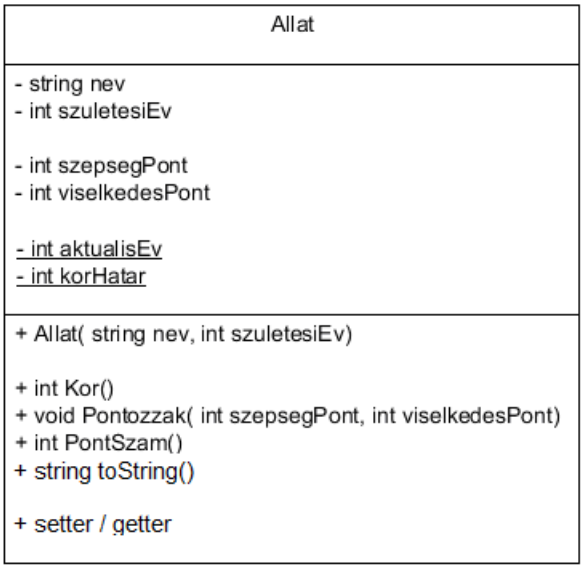
A mezőkkel kapcsolatban még egy fontos dolgot meg kell beszélnünk. Az objektumorientált programozás egyik alapelve az úgynevezett egységbezárás elve. Ez többek között azt jelenti, hogy a mezőket kizárólag csak az osztályban definiált módon lehet lekérdezni és módosítani. Ezért az összes mezőt mindig `private` jelzővel látjuk el. Ez azonban azt jelentené, hogy soha senki nem fér hozzá ezekhez az adatokhoz, kizárólag az osztály belügye lenne, hogy melyikkel mit csinál. Nyilván lehetnek olyan mezők is, amelyek értékét bárki lekérdezheti, illetve olyanok is, amelyeket szabad módosítani. Ezekhez azonban külön metódusokat kell írnunk, az értékadáshoz `set…()`, a lekérdezéshez `get…()` metódust (setter /getter). Felmerülhet a kérdés, hogy ha ezekkel a metódusokkal megváltoztathatjuk a mező értékét, illetve lekérdezhetjük azt, akkor miért nem egyszerűbb közvetlenül publikusnak deklarálni? Ennek biztonsági okai vannak. Ha csak metódusokon keresztül lehet hozzáférni az adatokhoz, akkor ezekben a metódusokban hozzáférési feltételeket is megadhatunk, sőt, az is lehet, hogy meg sem írjuk ezeket a metódusokat.

Ha megalkottuk az állat fogalmát (azaz megírtuk az `Allat` osztályt), akkor ettől kezdve már hivatkozhatunk erre a fogalomra, vagyis definiálhatunk ilyen típusú változókat, létrehozhatunk ilyen típusú példányokat, és dolgozhatunk is velük.

Ezt természetesen egy másik osztályban tesszük, esetünkben most a `Main` osztályban.

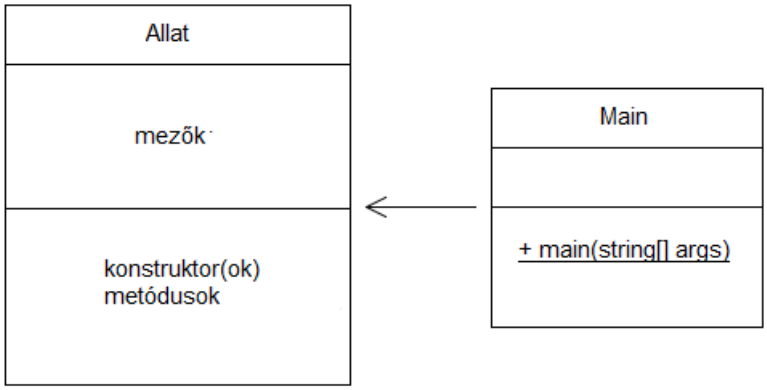
Foglaljuk össze az eddigieket egy, illetve a kétféle megoldási lehetőség miatt kétféle ábrába:





Most szubjektív döntés, hogy a baloldali esetben is akarjuk-e külön mezőkként kezelni a szépség- és viselkedéspontot.

A program szerkezete:



Jelölések:

- : privát

+: publikus

aláhúzás: statikus

Az ábra mutatja az `Allat` osztály felépítését, és azt is, hogy a `Main` osztályból fogjuk használni. (Ez az úgynevezett UML ábra – UML: Unified Modeling Language.)

Ezek után írjuk meg az `Allat` osztályt!

(Egyelőre ugyanabban a csomagban dolgozunk, amelyben a `Main` osztály is szerepel, ezért a csomagot a legelső kódrészlet kivételével sehol sem jelezzük.
Csomag: a logikailag összetartozó osztályokat rakjuk egy csomagba – ezt a fogalmat később még bővítjük.)

Először lássuk az osztály szerkezetét:

```
package allatverseny;

public class Allat {

    private String nev;
    private int szulEv;

    private int pontSzam;
    private static int aktualisEv;
    private static int korHatar;

    public Allat(String nev, int szulEv) {...4 lines }

    public int eletKor() {...3 lines }

    public void pontotKap(int szepsegPont, int viselkedesPont) {...8 lines

    @Override
    public String toString() {...3 lines }

    public String getNev() {...3 lines }

    public int getSzulEv() {...3 lines }

    public int getPontSzam() {...3 lines }

    public static int getAktualisEv() {...3 lines }

    public static int getKorHatar() {...3 lines }

    public static void setAktualisEv(int aktualisEv) {...3 lines }

    public static void setKorHatar(int korHatar) {...3 lines }
}
```

A konstruktor és az első két metódus nem okozhat problémát. A `toString()` metódusról ejtünk még néhány szót. Ez egy speciális metódus, a metódusfej fölött látható `@override` annotáció azt jelenti, hogy a minden osztály közös ősosztályából, azaz az `Object` osztályból származik. Az objektum `string` formáját adja meg, de természetesen azt mi döntjük el, hogy mi legyen ez a `string`. Nyilvánvalóan bármilyen más `string` típusú metódust is írhatnánk, a `toString()` annyiban speciális, hogy minden típushoz tartozik ilyen metódus, illetve még abban, hogy kiíratáskor nem kell megadnunk a metódus nevét, elég csak az objektumét.

Még két **megjegyzés**:

1. Az annotáció olyasmi, mint a komment, csak a komment a kódot olvasó ember számára ad valamilyen információt, egy annotáció pedig a fordítóprogram számára. Az `@override` annotáció például azt jelzi, hogy ez a metódus öröklődik az ősosztályból. De a program akkor is működne, ha ezt kitörölnénk (próbálja ki). Akkor mi szükség van rá? Segíti a szintaktikus

ellenőrzést. Ha nem lenne ott az annotáció, és például `tostring()`-et írnánk `toString()` helyett, akkor is lefordulna a program, de persze, ez a metódus már nem az örökölt metódus, ami csak a futás során derülne ki. Ha viszont ott van az annotáció, akkor fordításkor azonnal jelzi, hogy baj van, az általunk írt (elgépelt) metódus nem öröklődik sehonnan. Egy fordítási hibát mindig könnyebb javítani, mint a futáskor keletkezőt, ezért fontos ez az annotáció.
Annotációkat egyébként nagyon gyakran használnak a Java nyelvben, ennél jóval bonyolultabb esetekben is, de ez most nem tartozik a feladatunkhoz.

2. Felmerülhet a kérdés, hogy miért nem írunk inkább egy kiíró metódust. Ennek az oka, hogy nem tudhatjuk, hol akarjuk felhasználni ezt az osztályt. Másképp kell ugyanis kiíratni az adatokat, ha konzolos felületen dolgozunk, és másképp, ha grafikuson. De hiszen tudjuk, hogy most konzolos felületen dolgozunk, nem? Most igen, de nagyon fontos alapelv a programok, illetve osztályok újrahasznosíthatósága, vagyis az, hogy ha esetleg később mégis grafikus felületen szeretnénk dolgozni, akkor se kelljen módosítani az `Allat` osztályt. Emiatt alaposztályokban soha sincs kiíratás, és hasonló okok miatt beolvasás sem.

Ezek után nézzük a deklarációt, konstruktort és a metódusokat:

```
public class Allat {

    private String nev;
    private int szulEv;

    private int pontSzam;
    private static int aktualisEv;
    private static int korHatar;

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

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

    /** A megadott szépségpont és viselkedéspont alapján kiszámolja a ...8 lines */
    public void pontotKap(int szepsegPont, int viselkedesPont) {
        int kor = this.eletKor();
        if (kor > korHatar) {
            pontSzam = 0;
        } else {
            pontSzam = (korHatar - kor) * szepsegPont + kor * viselkedesPont;
        }
    }

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

Ismét két **megjegyzés**:

1. Mint látható, a kódkiemelés segít abban, hogy lássuk, melyik változó lokális és melyik mező, de ha igazán szépen szeretnénk programozni, akkor a mezőkre, saját metódusokra mindig a `this` operátorral együtt illik hivatkozni, mert így garantált, hogy nem keverjük össze a mezőt egy lokális változóval. De ezt a szabályt maguk a fejlesztőkörnyezetek sem tartják be, mert sokszor enélkül generálják a kódot. A `this` az aktuális objektumpéldányt jelenti. Ha statikus mezőre szeretnénk hivatkozni, akkor az nem egy konkrét objektumhoz tartozik, hanem az osztályhoz, vagyis azokra az osztályon keresztül hivatkozunk.

2. Bár tanulás közben magunknak célszerű időnként kommentezni a kódot, de egy jól megírt, szép, tiszta kódban nincsenek kommentek, ugyanis ha jól megválasztott változó-, illetve metódus-neveket használunk, akkor kommentek nélkül is jól olvasható a kód, sőt, a kommentek inkább zavaróak. Viszont dokumentálni kell a programot. Erre szolgálnak az úgynevezett dokumentációs kommentek, vagyis a `/** ...*/` jelek közé írt szöveg. Ezeket a metódusok elé írjuk (generálhatóak is a /** + enter begépelésével). Nem muszáj minden metódust kommentezni, csak amelyikről bővebb információt szeretnénk megadni. A generált html formátumú dokumentációban ezek a kommentek is megjelennek, illetve ezek jelennek meg az online help alkalmazásakor is. (NetBeans-ben így lehet generálni: Run menüpont, Generate Javadoc).

Az egyik ilyen dokumentációs kommenttel ellátott metódus:

```
/**
 * A megadott szépségpont és viselkedéspont alapján kiszámolja a
 * pontszám értékét. Ez az érték a korhatár fölött 0,
 * egyébként pedig függ az állat korától is.
 *
 * @param szepsegPont
 * @param viselkedesPont
 */
public void pontotKap(int szepsegPont, int viselkedesPont) {
    int kor = this.eletKor();
    if (kor > Allat.korHatar) {
        this.pontSzam = 0;
    } else {
        this.pontSzam = (Allat.korHatar - kor) * szepsegPont
                                                + kor * viselkedesPont;
    }
}
```

Itt is egy **megjegyzés**: a kor nevű változót azért volt célszerű bevezetni, hogy ne kelljen háromszor meghívni az életkort számoló metódust. Most egyszerű metódusról van szó, vagyis nem jelentene hatékonyságromlást, ha többször is meghívnánk, de egy bonyolultabb metódus kiszámításakor már erre sem árt gondolni.

Már csak a setterek / getterek megbeszélése maradt hátra.

Arról volt szó, hogy a név és a születési év nem változtatható. De kívülről nem változtathatjuk meg a pontszám értékét sem, hiszen az csalás lenne, mert ez a változó kizárólag a `pontotKap()` metódusban kaphat értéket. Ugyanakkor egyértelmű, hogy bárki tudhatja az állatpéldány nevét, születési évét, megkérdezheti a kapott pontszámát. Vagyis lekérdezni engedjük ezeket az adatokat, de módosítani nem. Ezért ezekhez get hozzáférést biztosítunk. Elvileg bármilyen nevű metódusban visszaadhatnánk ezeket az értékeket, de a konvenció az, hogy ezeket a metódusokat `getMezonev()` módon nevezzük, például:

```
public String getNev() {
    return nev;
}

public int getSzulEv() {
    return szulEv;
}
```

Kívülről, mégpedig egyszerű értékbeállítással adjuk meg az aktuális év és a korhatár értékét. Az ilyen mezőkhöz a set hozzáférést is biztosítani kell. A set (beállítás) azt jelenti, hogy a változó értéket kap, mégpedig azt, amelyet az osztály (illetve objektum) használója ad. Itt a névkonvenció: `setMezonev()`

Az, hogy az osztály vagy az objektum használója, attól függ, hogy milyen mezőről van szó. Az olyan mezők, melyek minden példány esetén egyediek, a konkrét objektumon keresztül kapnak majd értéket. Az aktuális év és a korhatár minden egyes példányra egyformán érvényes, ezért csak egyetlen memóriahelyen tároljuk őket, vagyis ezeket osztály szinten adjuk meg. Ezt az érték megadása esetén is ugyanúgy jelöljük, mint a mezők esetén, vagyis a `static` kulcsszóval. Például:

```
public static void setKorHatar(int korHatar) {
    Allat.korHatar = korHatar;
}

public static int getKorHatar() {
    return korHatar;
}
```

Kérdés lehet még a szépségpont és viselkedéspont helyzete. Ezek nyilván kívülről kapnak értéket, hiszen egy zsűri mondja meg, hogy melyik mekkora, ugyanakkor mégsem írhatunk hozzájuk módosító (vagy beállító, azaz set hozzáférést), hiszen a zsűrin kívül senki más nem módosíthatja ezeket a pontokat. Ők viszont a `pontotKap()` metódusban adják meg ezeket az értékeket. Ha kizárólag csak a végső pontszámot akarjuk tudni, akkor ezzel nem is kell tovább foglalkoznunk, hiszen a metódus belügye, hogy mit csinál ezekkel az értékekkel. Ha viszont úgy gondoljuk, hogy esetleg valaki kíváncsi lehet rá, hogy vajon ki mekkora szépségpontot kapott, akkor ezeket a részpontokat is mezőként kell kezelnünk. Ez esetben a kód bővül még

két meződeklarációval (`private int szepsegPont, viselkedesPont;`), a hozzájuk tartozó getterekkel, illetve így alakul a `pontotKap()` metódus:

```
public void pontotKap(int szepsegPont, int viselkedesPont) {
    this.szepsegPont = szepsegPont;
    this.viselkedesPont = viselkedesPont;
    int kor = this.eletKor();
    if (kor > Allat.korHatar) {
        this.pontSzam = 0;
    } else {
        this.pontSzam = (Allat.korHatar - kor) * szepsegPont
                                                + kor * viselkedesPont;
    }
}
```

Készen vagyunk az `Allat` osztály definiálásával. Nézzük meg, hogyan lehet felhasználni az itt megírt fogalmat. De mielőtt belemélyednénk az összetettebb vezérlés megírásába, célszerű lenne ellenőrizni, hogy valóban jó-e a megírt osztály, úgy viselkednek-e az egyes metódusok, ahogy szeretnénk. Ennek egyik – de igencsak „fapados” módja –, hogy írunk egy nagyon egyszerű `main()` metódust, amelyben kipróbáljuk az osztály működését. Például:

```
public static void main(String[] args) {

    int aktEv = 2018;
    int korhatar = 10;
    String nev = "Bodri";
    int szulEv = 2015;
    int szepsegPont = 7;
    int viselkedesPont = 4;

    Allat.setAktualisEv(aktEv);
    Allat.setKorHatar(korhatar);

    Allat allat = new Allat(nev, szulEv);
    System.out.println(allat);

    allat.pontotKap(szepsegPont, viselkedesPont);
    System.out.println(allat.getNev() + " pontszáma: " +
                       allat.getPontSzam() + " pont");
}
```

Ez – kivéve talán a tanulás nagyon kezdeti állapotát – szerencsétlen megoldás, hiszen ha kiderül, hogy úgy működik, ahogy szeretnénk, akkor utána dobhatjuk a szemétbe, hiszen nem ilyen `main()` metódusra van szükségünk. Ha valami miatt később még bővíteni akarjuk az osztályt, és csak egyszerű módon tesztelni, akkor írhatjuk újra az egészet. Nyilván nem megoldás a „szokásos” programozói módszer sem, vagyis az, hogy bizonyos kódsorokat hol

kikommentezünk, hol meg visszarakjuk a kommentet, ha mégsem azt akarjuk futtatni. Néha nem rossz ez a módszer, de sokszor nem szerencsés.

Ugyanakkor jogos igény az, hogy a komolyabb használat előtt ellenőrizzük az elkészült osztályt. Szerencsére van erre egy jól használható megoldás, mégpedig a JUnit teszt. Írjunk máris egy kicsi tesztet az `Allat` osztályhoz. (A teszt osztályt minden fejlesztőeszközben lehet generálni, NetBeans-ben: TestPackages / jobb egérgomb / New Java package, majd ezen a csomagon belül new JUnitTest – ezt első alkalommal a new /other menüpont alatt lehet megtalálni.) Ha esetleg nincs TestPackages mappa, akkor a projektnévre kattintva kérhetjük a teszt létrehozását.

A teszt egy speciális osztály, annotációk irányítják a metódusok végrehajtásának sorrendjét. Most csak a legelemibbeket említjük: a `@Test` annotációval ellátott metódusok futnak le teszteléskor, de ezek lefutási sorrendje esetleges. A `@Before` annotációval ellátott metódus minden egyes teszt metódus lefutása előtt lefut – itt érdemes megadni azokat a beállításokat, amelyek mindegyik teszt metódus esetén érvényesek. Az `@After` annotációval ellátott metódusok pedig minden egyes metódus lefutása után fut le. Sok egyéb annotáció is van még, de most elégedjünk meg ennyivel.

A teszt metódusokban feltételezésekkel élünk: feltesszük, hogy egy állítás igaz vagy hamis, hogy két érték azonos egymással, és még rengeteg mindent, most azonban csak ezzel a három esettel foglalkozunk.

A teszt futtatása: a tesztfájl nevén jobb egérgomb, Run file. Szerencsés esetben lefut a teszt (the test passed), ha nem, akkor pedig kiírja, hogy hol és milyen hiba miatt szállt el. A hiba oka lehet az, hogy hibásan írtuk meg a tesztet, de az is, hogy az alaposztályban vétettünk hibát.

A tesztvezérelt programozás ennél jóval bővebb, de sokszor előfordul, hogy a programozó egy teszt osztályt kap, és olyan kódot kell írnia, amely átmegy ezen a teszten.

```
package tesztcsomag;

import allatverseny.Allat;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class AllatTeszt {

    private final int aktEv = 2018;
    private final int korhatar = 10;
    private final String nev = "Bodri";
    private final int szulEv = 2015;
    private final int szepsegPont = 7;
    private final int viselkedesPont = 4;

    private Allat allat;
```

```
    @Before
    public void setUp() {

        Allat.setAktualisEv(aktEv);
        Allat.setKorHatar(korhatar);

        allat = new Allat(nev, szulEv);
    }

    @Test
    public void teszt() {

        // Feltételezzük, hogy az állat életkorát jól számoltuk:
        // Mindhárom feltételezés ugyanazt vizsgálja, fölösleges a három,
        // csak azért szerepel mindhárom, hogy láthassa, ugyanazt többféle
        // módon is meg lehet oldani.

        assertTrue(allat.eletKor() == Allat.getAktualisEv() - allat.getSzulEv());
        assertEquals(allat.eletKor(), Allat.getAktualisEv() - allat.getSzulEv());
        assertEquals(allat.eletKor(), 3);

        // Feltételezzük, hogy a verseny előtt még nincs pontszáma:

        assertEquals(allat.getPontSzam(), 0);

        allat.pontotKap(szepsegPont, viselkedesPont);

        // Feltételezzük, hogy ilyen adatok mellett a verseny után már
        // nem nulla a pontszám - ez most fölösleges, hiszen utána azonnal
        // azt vizsgáljuk, hogy jól számolta-e a pontszámot, most azért
        // szerepel, hogy a hamis feltételezésre is lásson példát:

        assertFalse(allat.getPontSzam() == 0);

        assertEquals(allat.getPontSzam(), 61);

        // Még azt is ellenőrizni kell, hogy ha az életkora a korhatár
        // fölött van, akkor valóban nulla pontot kap-e.
        // Ezért most egy újabb - idősebb - példányt hozunk létre, pontozzuk,
        // majd ellenőrizzük a pontszámát:

        allat = new Allat(nev, Allat.getAktualisEv() - Allat.getKorHatar() - 1);

        // Ha már ilyen kacifántosan adtuk meg az életkort :) (pedig lehetett
        // volna így is: new Allat(nev, 2007);) akkor ellenőrizzük, hogy jó-e
        // az életkora:

        assertEquals(allat.eletKor(), 11);

        // Végül ellenőrizzük, hogy ilyen szépség- és viselkedéspont esetén
        // valóban nulla lesz-e az össz pontszáma:

        assertEquals(allat.getPontSzam(), 0);
    }
}
```

A teszt azt mutatja, hogy helyesen írtuk meg az `Allat` osztályt, vagyis következhet a vezérlő osztály. (Tesztelésről kicsit bővebben: http://tutorials.jenkov.com/java-unit-testing/index.html)

Beszéljük meg a vezérlést. Ez lesz a `Main` osztály. Ebben a következő teendőink vannak: regisztrálni kell a jelentkező állatokat, vagyis be kell olvasnunk a megfelelő adatokat, és létrehozni belőlük a versengő példányokat. Természetesen a statikus adatok értékét is be kell állítanunk. Hogy lássuk a beolvasás eredményét, azonnal írassuk ki a regisztrált állatokat (most a nevüket és életkorukat), majd jöhet a verseny. Ez után ismét írassuk ki őket, de most a név mellé a pontszám kerüljön. Végül határozzuk meg, hogy ki(k) nyerté(k) a versenyt, majd írassuk ki őket pontszám szerint csökkenő sorrendben.

Mivel több állat szerepel a versenyen, ezért vagy tömbben, vagy listában kell őket tárolnunk. A lista szerencsésebb megoldás, hiszen így nem kell előre megadni, hogy vajon hányan jelentkezhetnek. Kevés unalmasabb dolog van, mint billentyűzetről olvasni, ezért adatfájlból olvassuk majd az adatokat.

Természetesen mindezt nem ömlesztve, hanem metódusokra bontva oldjuk meg. Ezzel kapcsolatban még egy dolgot kell megbeszélnünk. Ha ezeket a metódusokat közvetlenül a `main()` metódusból hívjuk, akkor – mivel a `main()` metódus statikus – az összes belőle hívott metódusnak is statikusnak kell lennie. Ez nem feltétlenül baj, sőt bizonyos értelemben hatékonyabb és helytakarékosabb is, ugyanakkor kicsit merevvé teszi a programot. Ez vezérlő osztály esetén szintén nem baj, de alacsonyabb szinten szerencsésebb, ha rugalmasabb a kód.

Egy kicsit szubjektív, hogy melyik változatot választja a programozó. Most úgy oldjuk meg, hogy ne legyenek benne statikus metódusok (a másik változatban viszont statikus metódusokat használunk majd). Ennek legegyszerűbb módja, ha létrehozunk egy példányt a `Main` osztályból, és meghívjuk ennek az indító metódusát. Ez a metódus „helyettesíti” majd az eredeti `main()` metódust, de mivel példányon keresztül hivatkozunk rá, ezért már nem kell statikusnak lennie.

Az osztály elején megadjuk a programban szereplő konstansokat. Alapvető dolog, hogy konstansokat nem égetünk be a kódba. Vagyis metódusokon belül nem használunk konstansokat, például nem mondjuk azt, hogy a szépségpont mondjuk 0 és 10 közötti érték legyen, ugyanis mi van akkor, ha egy másik versenyen csak 5-ig engednek pontozni? Akkor dobjuk sutba ezt a programot? Vagy bogarásszuk végig a kódot, hogy hol kell módosítani rajta? Nyilván nem. Ezért a használni akart konstansok mindegyikét belerakjuk egy változóba, és erre a változóra hivatkozunk. A konstans értékeket meg lehet adni a vezérlő osztály elején – itt könnyen lehet módosítani őket, de akár az is lehet, hogy kigyűjtjük őket egy külön osztályba vagy egy konfigurációs fájlba.

Az állatok listájára az összes metódusban hivatkozunk, ezért mezőként deklaráljuk. Szintén szubjektív, hogy a deklarációhoz azonnal hozzákapcsoljuk-e a lista példány létrehozását, vagy ezt a konstruktorban tesszük meg, de jelenleg a konstruktornak semmi más feladata nem lenne, ezért talán kényelmesebb a deklaráció során példányosítani:

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

A deklarációt később még részletesebben tárgyaljuk, most csak annyit, hogy az, amit látunk, nagyon gyakori megoldás, mégpedig az, hogy a változót interfész típusra deklaráljuk, de példányosítani nyilván csak az interfészt implementáló osztályt lehet. Hogy ez miért jó, arról majd később lesz szó. Esetünkben a `List` interfész, az `ArrayList` osztály, a <> zárójelben pedig az olvasható, hogy milyen típusú elemeket tartalmaz majd a lista.

Lássuk tehát a vezérlő osztály deklarációs részét, indító metódusát, sőt, a statikus adatok megadását is. Az aktuális év értékét a számítógépünktől kérjük le (erre van más lehetőség is):

```
public class Main {

    private final String adatfajl = "allatok.txt";
    private final int aktualisEv = Calendar.getInstance().get(Calendar.YEAR);
    private final int korHatar = 10;
    private final int szepsegPonthatar = 11;
    private final int viselkedesPonthatar = 11;

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

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

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

    private void statikusAdatok() {
        Allat.setAktualisEv(aktualisEv);
        Allat.setKorHatar(korHatar);
    }
```

(Egyszerű dátumkezelés: http://www.tutorialspoint.com/java/java_date_time.htm – 8-as Java ennél jóval többet tud, de erről majd később.)

Az adatokat az *allatok.txt* fájlból (vagy hozzá hasonlóan megadott adatokat tartalmazó fájlból szeretnénk olvasni). Mint látható, egy-egy állat adatai alkotnak egy-egy sort: név;születési_év. Most ezt az adatfájlt a projekt gyökeréből fogjuk olvasni. Az adatfájl létrehozásakor figyelni kell rá, hogy UTF-8-as kódolással mentsük. Lehetőleg ne legyen üres sor a végén, bár ezt akár kódból is kezelhetjük.

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

Többféle módon lehet fájlból olvasni, most a legegyszerűbb változatát vesszük. Olvasásra a `Scanner` osztály egy példányát használjuk, és egy fájlra irányítjuk a scanner-t:

```
Scanner scanner = new Scanner(new File("allatok.txt"));
```

Sem a `Scanner` osztály, sem a `List` interfész, `ArrayList` osztály nem része az alap Java csomagnak (`java.lang`), ezért importálni kell őket, de ez generálható.

A fájlból addig tudjuk olvasni a sorokat, amíg a scanner talál új sort. A fájlt soronként tudjuk olvasni, a sor szétvágásához a `String` osztály `split()` metódusát használhatjuk. A beolvasott adatokból létre tudunk hozni egy új állat példányt, majd a lista `add()` metódusával hozzá tudjuk adni az eddigiekhez.

Fájlból való olvasáskor mindig kötelező a kivételkezelés. A Java ezt valóban komolyan veszi, és anélkül nem is hajlandó lefordítani a programot. Ugyanakkor nem muszáj direktben leírnia Ha a sor melletti kis lámpácskára kattint, akkor két vagy három javítási ötletet kap (attól függ, hogy mikor kattint a lámpácskára, akkor, amikor már megírta a teljes metódust, vagy az első sor után azonnal). Ha érti a kiírt javaslatokat, akkor döntsön belátása szerint. Ha még nem igazán érti, akkor válassza a „Surround … with try-catch” változatot. Ha már a blokkot is felkínálja (ezt akkor teszi, ha több kódsort is megírt már), akkor válassza ezt, ha csak az adott sort kínálja fel, akkor azt, de ez utóbbi esetben is írjon mindent a try blokkba. A rendes kivételkezelésről kicsit később lesz majd szó.

Ezek után a beolvasó metódus:

```
private void adatBevitel() {

    try {
        Scanner sc = new Scanner(new File(adatfajl));
        String[] adatok;
        String sor;
        while (sc.hasNextLine()) {
            sor = sc.nextLine();
            if (!sor.isEmpty()) {
                adatok = sor.split(";");
                allatok.add(new Allat(adatok[0],
                                      Integer.parseInt(adatok[1])));
            }
        }
        sc.close();
    } catch (FileNotFoundException ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Az adatok kiíratás nagyon egyszerű. Arra kell csak figyelni, hogy kiíratás előtt mindig kell egy cím, hiszen udvariatlan dolog értelmezetlen adatokat dobni a felhasználó orra alá. Célszerű az iteráló `for` ciklust használni a klasszikus helyett. (Ezt szokták foreach szerű ciklusnak is nevezni.) Ez végigiterál a ciklusfejben lévő gyűjtemény elemein. (Itt láthatjuk a `toString()` metódus „használatát” – csak az `allat` példányt íratjuk ki, melynek eredményeként azt látjuk, amit a `toString()` metódusba írtunk.)

```
private void resztvevok() {
    System.out.println("\nA verseny résztvevői:");
    for (Allat allat : allatok) {
        System.out.println(allat);
    }
}
```

A verseny és az eredmény kiíratása:

```
private void verseny() {
    int szepseg, viselkedes;
    for (Allat allat : allatok) {
        szepseg = (int) (Math.random()*szepsegPonthatar);
        viselkedes = (int) (Math.random()*viselkedesPonthatar);
        allat.pontotKap(szepseg, viselkedes);
    }
}

private void eredmenyek(String cim) {
    System.out.println(cim);
    for (Allat allat : allatok) {
        System.out.println(String.format("%s %d pont", allat.getNev(),
                                                       allat.getPontSzam()));
    }
}
```

Mivel az eredményt kétszer is ki akarjuk íratni (egyszer a zsűri pontozása után, egyszer pedig a rendezést követően, ezért célszerű végiggondolni, hogy hogyan lehet elkerülni a kódismétlést (egy jó kódban nincs ismétlés). Látjuk, hogy a két kiíratás csak az adatok előtt szereplő címben térnek el, ezért ezt érdemes paraméterként kezelni, és máris tudjuk két különböző szituációban is hívni ugyanazt a metódust.

Egy „igazi” verseny úgy zajlana, hogy a zsűri valóban pontokat ad, amelyeket valamilyen módon be kellene olvasni a programba. Most ezt helyettesítjük azzal, hogy véletlenül generáljuk a pontszámokat. A véletlen generálására két módszer is lehetséges. Az egyik az, ami följebb látható, vagyis a `Math` osztály `random()` metódusát használjuk. Ez egy `double` érték: 0 <= `Math.random()` < 0.

A másik lehetőséghez példányosítanunk kell a `Random` osztályt (ez sincs benne a `java.lang` csomagban, vagyis importálni kell). Ennek az osztálynak több metódusa is van, ezeket is alkalmazhatjuk véletlen érték generálására. Ha ezt használjuk, akkor így alakul a `verseny()` metódus:

```
private void verseny() {
    Random rand = new Random();
    int szepseg, viselkedes;
    for (Allat allat : allatok) {
        szepseg = rand.nextInt(szepsegPonthatar);
        viselkedes = rand.nextInt(viselkedesPonthatar);
        allat.pontotKap(szepseg, viselkedes);
    }
}
```

A legjobb(ak) keresése esetén mindig gondolnunk kell a holtversenyre. Ezért első körben megállapítjuk, hogy mekkora a legnagyobb pontszám, majd kiíratjuk az összes olyan állat nevét, akihez ekkora pontszám tartozik:

```
private void legjobbak() {
    int max = allatok.get(0).getPontSzam();
    for (Allat allat : allatok) {
        if(allat.getPontSzam() > max){
            max = allat.getPontSzam();
        }
    }

    System.out.printf("\nA legjobbak %d pontszámmal:\n", max);
    for (Allat allat : allatok) {
        if(allat.getPontSzam() == max){
            System.out.println(allat.getNev());
        }
    }
}
```

Mint látható, ez már a harmadik féle kiíratási mód. Kiírathatunk egy darab `String`-et, ha előtte konkatenáljuk a kiírandó adatokat, hivatkozhatunk a `String` osztály `format()` metódusára, és konzolra alkalmazhatjuk a `printf()` metódust is, amelynek paraméterlistája ugyanolyan, mint a `String.format()` metódusé, vagyis az első paraméter egy formázó `String`, amelyet vesszővel felsorolva követnek a kiírandó argumentumok.

Végül foglalkozzunk a rendezéssel. Írhatnánk egy saját rendezési metódust – pl. a buborék algoritmust, de jóval hatékonyabb, ha a `Collections` osztály `sort()` metódusát használjuk. Ez kétféle módon paraméterezhető, ill. ily módon kétféleképpen oldható meg a rendezés. Mindkét esetben az okozza a problémát, hogy meg kell mondanunk a rendezési szempontot is, hiszen az állatokat rendezhetnénk névsorba, vagy életkor szerint, illetve most pontszám szerint.

Az egyik lehetőség ez:

```
private void pontszamSzerintRendezve() {
    Collections.sort(allatok);
}
```

Ekkor kicsit módosítanunk kell az `Allat` osztályt, ugyanis rendezni csak olyan példányokat tudunk, amelyeket össze lehet hasonlítani, azaz implementálják a `Comparable` interfészt. Ennek egyetlen implementálandó metódusa van, a `compareTo()` (generálható). Ez összehasonlítja az aktuális példányt a paraméterében lévő példánnyal, és értelemszerűen pozitív, negatív vagy 0 értéket ad vissza, attól függően, hogy az összehasonlítás két eleme közül melyik a nagyobb.

Ha így módosítjuk az állat osztályt, akkor az osztály feje és a kötelezően implementálandó metódus:

```
public class Allat implements Comparable<Allat>{

    @Override
    public int compareTo(Allat o) {
        return o.pontSzam - this.pontSzam;
    }
```

A másik lehetőség, hogy hozzá sem nyúlunk az eredeti `Allat` osztályhoz, hanem létrehozunk egy külső rendező osztályt, és ezt használjuk a rendezéskor. Ennek az osztálynak `Comparator` típusúnak kell lennie, és a `compare()` metódust kell implementálnia. A metódus az előzőekben tárgyalthoz hasonló módon pozitív, negatív egész vagy 0 értéket ad vissza, attól függően, hogy a két paraméter közül melyik a nagyobb. Az ennek megfelelő rendezés és a rendező osztály:

```
    private void pontszamSzerintRendezve() {
        Collections.sort(allatok, new RendezesPontszamSzerint());
    }
```

```
public class RendezesPontszamSzerint implements Comparator<Allat>{

    public RendezesPontszamSzerint() {
    }

    @Override
    public int compare(Allat o1, Allat o2) {
        return o2.getPontSzam() - o1.getPontSzam();
    }
}
```

Megjegyzések:

1. Az 1.8-as Java verziótól kezdve a `Collections` osztály nélkül is megoldható a rendezés:

```
    private void pontszamSzerintRendezve() {
        allatok.sort(new RendezesPontszamSzerint());
    }
```

2. Egy teszt osztályban ellenőriztük, hogy jól definiáltuk-e az `Allat` osztályt. Legalább ilyen fontos lenne tesztelni a vezérlő osztályt, hiszen abban bonyolult, sőt, az alaposztály metódusainál jóval bonyolultabb metódusok lehetnek. Azonban ezek a metódusok általában privát hozzáférésűek, vagyis nem lehet elérni őket egy külső tesztosztályból. Természetesen tesztelhető a vezérlő osztály is, illetve az egyes osztályok közötti kapcsolat is, de ez túlmutat a mostani célunkon.

Még két **adósság:**

1. A megoldás megtervezése során szó volt arról, hogy a pontszám kiszámítását külön metódusba is írhatnánk (sőt, a továbbfejlesztés szempontjából ez lenne a jobb). Ekkor az `Allat` osztályban nem kell (nem szabad) definiálnunk a `pontSzam` változót és a hozzá tartozó gettert sem, a megfelelő metódusok pedig így alakulnak:

```
public void pontotKap(int szepsegPont, int viselkedesPont) {
    this.szepsegPont = szepsegPont;
    this.viselkedesPont = viselkedesPont;
}

public int pontSzam(){
        int kor = this.eletKor();
    if (kor > Allat.korHatar) return 0;
    return (Allat.korHatar - kor) * szepsegPont + kor * viselkedesPont;
}
```

Ennek megfelelően természetesen a felhasználás során is megváltozik a pontszámok használata, mégpedig úgy, hogy a kódban szereplő összes korábbi `getPontSzam()` hivatkozást kicseréljük `pontSzam()` hívásra, pl:

```
private void legjobbak() {
    int max = allatok.get(0).pontSzam();
    for (Allat allat : allatok) {
        if(allat.pontSzam() > max){
            max = allat.pontSzam();
        }
    }

    System.out.printf("\nA legjobbak %d pontszámmal:\n", max);
    for (Allat allat : allatok) {
        if(allat.pontSzam() == max){
            System.out.println(allat.getNev());
        }
    }
}
```

2. Arról is szó volt, hogy kicsit szubjektív, hogy a vezérlő osztályban statikus metódusokat használunk-e vagy sem. Ha a statikus metódusok mellett döntünk, akkor viszont a hivatkozott változókat is statikusra kell deklarálnunk. Az ilyen elvű megoldás néhány részlete:

A program belépési pontja:

```
public static void main(String[] args) {
    start();
}
```

A deklaráció és néhány metódus:

```
private static final String ADATFAJL = "allatok.txt";
private static final int AKTUALIS_EV =
                            Calendar.getInstance().get(Calendar.YEAR);
private static final int KORHATAR = 10;
private static final int SZEPSEG_PONT_HATAR = 11;
private static final int VISELKEDES_PONT_HATAR = 11;

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

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

private static void statikusAdatok() {
    Allat.setAktualisEv(AKTUALIS_EV);
    Allat.setKorHatar(KORHATAR);
}

private static void adatBevitel() {
    ...
```

Értelemszerűen a többi hivatkozott metódus is statikus.

Hurrá! Akkor biztosan van még más jó megoldás is, hiszen eleve úgy indult ez a leírás, hogy több jó megoldás lehetséges. Ez igaz, de mint kiderül, nem is olyan sok. A most tárgyalt ötlet például jó. De vajon jó-e az – az egyébként akár működőképes megoldás –, hogy mindhárom pontszámfajtát mezőként deklaráljuk, a `pontotKap()` metódus helyett a szépségponthoz és a

viselkedésponthoz nemcsak get, hanem set hozzáférést is biztosítunk, és csak a most tárgyalt `pontSzam()` metódust írjuk meg?

Ez első olvasásra jónak tűnhet, de ha alaposabban belegondolunk, akkor mégsem az. Miért?

A feladat szerint a zsűri egyszerre kétféle pontot ad. A most említett megoldás lehetőséget adna rá, hogy valamelyik pontozást elsinkófáljuk, vagy esetleg utólag, korrupt módon megváltoztassuk a zsűri döntését. Az eredeti megoldásban tárgyalt paraméterezés erre nem ad lehetőséget.

Utolsó **megjegyzés**: Programírás közben bizony előfordulhat, hogy időnként módosítanunk kell a kódot, aztán egyszer csak csúnya lesz a külalak, pedig nagyon fontos, hogy jól tabulált és szépen formázott legyen a kód, hiszen ez is segíti az olvashatóságát. Bármelyik fejlesztőkörnyezet segíti a kódformázás. A NetBeans-ben például az Alt+Shift+F billentyűkombináció segítségével lehet formázni a kijelölt részt.

Sok egyéb kód-generálási vagy kódírást segítő lehetőség is van benne, érdemes időt szánni a megismerésükre.