JAVA-TRAVEL

Feladat

Folytassuk a múlt órai feladatot!

Eddig jutottunk:
A JAVA-TRAVEL utazási iroda több hajót működtet, ezeken utaztatja a vendégeit.

Minden hajóút esetén adott annak neve és egy egyedi azonosítója, ezen kívül pedig az utaztatható személyek maximális száma. A hajóra akkor tud felszállni egy utas, ha az utaslétszám még ezen a korláton belül van, és az utas beszállhat. Ha beszállhat, akkor az utas bekerül a hajó utas-listájába. Egy-egy út költsége egy-egy adott hajón minden utas számára azonos.

Az utasokat a nevük és egy egyedi kód azonosítja. Mindenkinek van valamennyi pénze, és bármikor tud költeni valamennyit és kapni is valamennyit. A társaság csak akkor enged beszállni valakit, ha ki tudja fizetni az adott út költségét, plusz még ezen felül marad nála egy minden utas számára egyformán kötelező alaptőke.

Írjunk programot az iroda működtetésére, vagyis olvassuk be néhány hajó és néhány utas adatait, szimuláljuk az utazást, majd írassuk ki az adatokat.
A szimuláció ezt jelenti: minden hajó esetén adjuk meg az illető hajóra érvényes útiköltséget (bizonyos határok között véletlenül generált érték), majd valahányszor egymás után válasszunk ki egy véletlen hajót, amelyre egy véletlen utas megpróbál felszállni.

Folytatás:

a/Bővítsük az eddigieket:

- A cég kedvezményt ad a Java programozók számára ☺, de mivel nem mindenki egyformán jó programozó, ezért a kedvezmény százalékát egyénenként dönti el. (A Java programozók azonban reménykednek abban, hogy esetleg a családtagjuk, barátjuk is kedvezményt kaphat, ezért úgy írják meg a programot, hogy csak az legyen érdekes, hogy valaki kedvezményezett. ☺) Kiíratáskor az is kerüljön a kedvezményezett neve mellé, hogy hány százalék kedvezményt kap.

- További feladatként számoljuk ki a cég teljes bevételét. Gondolja végig, hogy ehhez mit, melyik osztályban és hogyan kell megadni.

b/ Teszteljük a megoldásban szereplő osztályokat. Hf-ként írjon olyan teszt-eseteket, amelyekre nem jutott idő az órán.

További feladatok a „szokásosak”: melyik hajónak legtöbb a bevétele, melyiken utaznak a legtöbben/legkevesebben, stb.

Azt gondolom, ez a feladatmegoldás nem sok magyarázatot igényel, így csak néhány részletet emelek ki belőle.

1. Ismétlésként: az utas lista gettere: az eredeti lista helyett annak egy másolatát adjuk át:

```
public List<Utas> getUtasLista() {
    return new ArrayList<>(utasLista);
}
```

2. A kedvezményes utasokról azt tudjuk, hogy ők is utasok, csak kedvezményesen utazhatnak. Vagyis a `KedvezmenyesUtas` osztály az `Utas` osztály leszármazottja lesz. A kedvezmény-százalékot most a konstruktor paraméterlistáján adjuk meg, de természetesen tartozhat hozzá setter is és getter is. Az osztály setter és getter nélkül:

```
public class KedvezmenyesUtas extends Utas{

    private int kedvezmenySzazalek;

    public KedvezmenyesUtas( String nev, String kod, int kedvezmenySzazalek) {
        super(nev, kod);
        this.kedvezmenySzazalek = kedvezmenySzazalek;
    }

    @Override
    public boolean beszallhat(int utiKoltseg) {
        return super.beszallhat((int) (utiKoltseg*(1-kedvezmenySzazalek/100.))
    }

    @Override
    public String toString() {
        return super.toString() + "kedvezmeny: " + kedvezmenySzazalek + " %" ;
    }
```

Megjegyzés: a `kedvezmeny` változó lehetne `double` is (sőt, talán logikusabb is volna), de akkor nem tudtam volna kihangsúlyozni azt, hogy két `int` típusú változóval végzett művelet eredménye `int` típusú. A konstans egész értéket is valóssá tehetjük, nem csak a változót. Például úgy, ahogy látja, vagy a `100.0` vagy `100f` módon is.

Ahhoz, hogy a cég bevételét ki tudjuk számolni, célszerű bevezetni egy-egy újabb változót a `HajoUt` és az `Utas` osztályban is.

Az `Utas` osztály `jegyAr` változója a `beszallhat()` metódusban kap értéket. Ez az az érték, amelyet az utasnak ténylegesen fizetnie kell az útért.

```
public boolean beszallhat(int utiKoltseg){
    if(this.penz >= utiKoltseg + alapToke) {
        this.jegyAr = utiKoltseg;
        return true;
    }
    return false;
}
```

Természetesen meg kell írnunk a `jegyAr` mezőhöz tartozó gettert is.

Nem árt végiggondolni, hogy az így megadott jegyár a kedvezményes utasok esetén valóban kedvezményes ár-e. Ehhez azt kellene végiggondolni, hogy mennyit kell fizetnie egy kedvezményes utasnak. Az ő `beszallhat()` metódusa az ősosztály `beszallhat()` metódusát hívja meg, de már a kedvezményes költséggel. Vagyis ha a `jegyAr` változó a metódus paraméterének értékét veszi fel, akkor a kedvezményes utas esetén ez pontosan a kedvezményes összeg.
(Ha még ezután sem biztos a dolgában, írjon a vezérlésben egy próba-metódust, amelyben az egyik hajón állítsa be az útiköltséget pl. 1000-re vagy 100-ra, szálljon fel az összes utas a hajóra, majd írassa ki az utasokat a jegyárakkal együtt.)

Ha ismerjük a ténylegesen fizetendő jegyárat, akkor a hajóra való felszálláskor könnyedén ki tudjuk számolni egy-egy hajó bevételét, vagyis a `HajoUt` osztályban:

```
public void felszall(Utas utas) {
    if (utasLista.size() < maxUtasSzam
            && utas.beszallhat(utiKoltseg)
            && !utasLista.contains(utas)) {
        utasLista.add(utas);
        utas.penztKolt(utas.getJegyAr());
        bevetel += utas.getJegyAr();
    }
}
```

Természetesen meg kell írnunk a `bevetel` mezőhöz tartozó gettert is.

Megjegyzés: A feladatot úgy is meg lehetett volna oldani, hogy itt, a `felszall()` metódusban ellenőrizzük, hogy az illető utas kedvezményezett-e. (Az `utas` objektum példánya-e a `KedvezmenyesUtas` osztálynak.) Ha igen, akkor az útiköltség alapján kiszámítjuk, hogy mennyit fizet, és evvel növeljük a bevételt, sima utas esetén pedig az útiköltséggel. Ez is elfogadható megoldás lenne, csak ha így oldjuk meg, akkor két helyen is meg kell írnunk a kedvezmény kiszámítására vonatkozó összefüggést (itt is és a kedvezményes utas `beszallhat()` metódusában is). Ez első látásra nem tűnik nagy bajnak, de ha belegondolunk, hogy esetleg később változhat a kedvezmény kiszámításának módja, akkor máris baj van, mert két helyen is módosítani kell a programot, és könnyen előfordulhat, hogy pl. a másik helyen elfelejtjük, vagy nem ugyanúgy módosítjuk. Ez az oka annak, hogy egy jó programban soha nincs kódismétlés. (A kódismétlést többek között pont az öröklődéssel lehet elkerülni.)

A `Vezerles` osztály `adatBevitel()` metódusában csak az utasok beolvasását kell módosítanunk (az utasok.txt fájlból olvasunk):

```
        fileScanner = new Scanner(new File(UTASOK_PATH));
        Utas utas;
        while (fileScanner.hasNextLine()) {
            sor = fileScanner.nextLine();
            if (!sor.isEmpty()) {
                adatok = sor.split(";");
                if (adatok.length == KEDVEZMENYES_ADATHOSSZ) {
                    utas = new KedvezmenyesUtas(adatok[0], adatok[1],
                                                Integer.valueOf(adatok[3]));
                } else {
                    utas = new Utas(adatok[0], adatok[1]);
                }

                utas.penztKap(Integer.valueOf(adatok[2]));
                utasok.add(utas);
            }
        }
        fileScanner.close();
    } catch (FileNotFoundException ex) {
        Logger.getLogger(Vezerles.class.getName()).
                log(Level.SEVERE, null, ex);
    }
```

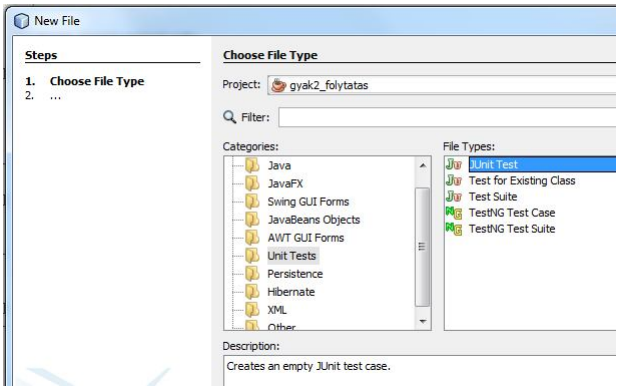
Megjegyzés: A beolvasás kikerüli a fájlban lévő esetleges üres sorokat, de azt már nem tudja kezelni, hogy ha egy sorban pl. háromnál kevesebb adat szerepel. Ha van rá ideje, próbálja ezt is megoldani.

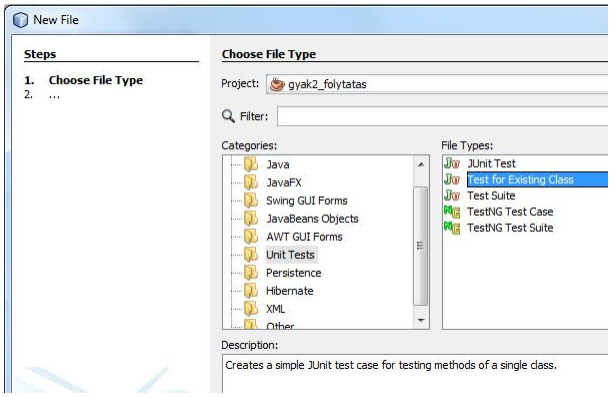
A bevétel kiszámítása:

```
    private void bevetel() {
        int osszBevetel = 0;
        for (HajoUt hajoUt : hajoutak) {
            osszBevetel += hajoUt.getBevetel();
        }
        System.out.printf("Az iroda bevétele: %d peták\n", osszBevetel);
    }
```

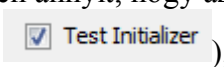
Hátra van még a tesztelés. Ez kicsit unalmas, de nagyon fontos része egy programnak. Manapság e nélkül már el sem fogadnak egy-egy projektet (legalábbis a komolyabb szoftvercégek).

Java programokhoz a JUnit (egységteszt) keretrendszert használjuk, jelenleg a 4-es verziót.
Ez a NetBeans-ből is elérhető: projektnév, jobb egérgomb, majd az Others csoportból:





A legjobb, ha mindkettőt kipróbálja, most csak annyit, hogy az első egy gyakorlatilag üres teszt osztályt hoz létre, a második pedig a kiválasztott osztály(ok)-hoz egy-egy osztályt, sőt, az összes metódusához is egy-egy teszt metódust. Viszont nyilván nem kell mindent tesztelni, pl. egy gettert vagy settert teljesen fölösleges, ill. lehet más olyan metódust is, amelyet nem fontos ellenőrizni. Ezt egy kicsit ügyesebb programozó már „érzi”.

Nézze át a javasolt tutoriálokat, most csak röviden annyit, hogy az egyszerűbb tesztekhez elég a `@Before` annotációjú metódus (generálás: 

), és a `@Test` annotációjút megírni.

A `@Before` metódus minden egyes teszt-metódus végrehajtása előtt lefut (ezért itt írjuk meg az inicializációkat), a `@Test` metódusokat pedig a futtató környezet futtatja (és csak ezeket futtatja, vagyis ha hiányzik ez az annotáció, akkor a teszt metódus nem kerül sorra).

A tesztelés célja a programunk ellenőrzése, ezt egyelőre két (vagy akár csak egy) metódus segítségével oldjuk meg. A metódusok szerepe, hogy beszúrjanak egy-egy igaznak vagy hamisnak vélt állítást, a teszt futása pedig azt ellenőrzi, hogy valóban igaz vagy hamis-e ez a metódus. Ha nem olyan, amilyennek véljük (vagyis ha pl. nem igaz a true-nak képzelt), akkor a teszt nem fut le, és azt is jelzi, hogy melyik sorban talált ellentmondást.

Ez a két metódus az `assertTrue()`, illetve az `assertFalse()`. De ha kicsit is utánanéz, láthatja, hogy még rengeteg másik, hasonló célú metódus is létezik, vagyis az általunk vett egyszerű tesztek nyilván csak kis ízelítőt adnak, ezeknél jóval komolyabb teszteket is lehet (és a majdani munkája során kell is) készíteni.

Futtatás: fájlnév, jobb egérgomb (a későbbiekre meg önállóan is rájön ☺)

Végül egy lehetséges teszt-osztály:

```java
public class UtazasiIrodaTeszt {

    public UtazasiIrodaTeszt() {
    }

    private final String NEV = "hajonev";
    private final String AZONOSITO = "azonosito";
    private final int MAX_UTASSZAM = 2;
    private final int UTIKOLTSEG = 1000;
    private final int ALAP_TOKE = 500;

    private HajoUt hajoUt;

    @Before
    public void setUp() {
        hajoUt = new HajoUt(NEV, AZONOSITO, MAX_UTASSZAM);
        hajoUt.setUtiKoltseg(UTIKOLTSEG);
        Utas.setAlapToke(ALAP_TOKE);
    }

    @Test
    public void tesztFelszall() {
        // Feltételezés: induláskor üres az utaslista és nincs bevétel
        assertTrue(hajoUt.getUtasLista().isEmpty());
        assertTrue(hajoUt.getBevetel() == 0);

        Utas utas = new Utas("nev1", "kod1");
        utas.penztKap(5000);
        hajoUt.felszall(utas);

        // Feltételezés: már nem üres az utaslista, hanem 1 a mérete
        // és tartalmazza ezt az utast
        assertFalse(hajoUt.getUtasLista().isEmpty());
        assertTrue(hajoUt.getUtasLista().size() == 1);
        assertTrue(hajoUt.getUtasLista().contains(utas));

        // Feltételezés: a hajó bevétele 1000
        assertTrue(hajoUt.getBevetel() == 1000);

        hajoUt.felszall(utas);

        // Feltételezés: ugyanaz az utas nem tud még egyszer felszállni
        // (van annyi pénze, hogy fel tudna szállni, ha már nem lenne fent)
        assertTrue(hajoUt.getUtasLista().size() == 1);

        utas = new Utas("nev2", "kod2");
        utas.penztKap(1200);
        hajoUt.felszall(utas);

        // Feltételezés: ennek az utasnak kevés pénze van, nem tud felszállni
        // emiatt nem is fizet
        assertTrue(hajoUt.getUtasLista().size() == 1);
        assertTrue(hajoUt.getBevetel() == 1000);
```

```
        utas = new KedvezmenyesUtas("nev3", "kod3", 30);
        utas.penztKap(1200);
        hajoUt.felszall(utas);

        // Feltételezés: ő felszállhat
        assertTrue(hajoUt.getUtasLista().size() == 2);
        assertTrue(hajoUt.getBevetel() == 1700);

        utas = new Utas("nev4", "kod4");
        utas.penztKap(2000);
        hajoUt.felszall(utas);

        // Feltételezés: már nincs hely, nem tud felszállni,
        // és a lista nem tartalmazza őt.
        assertTrue(hajoUt.getUtasLista().size() == 2);
        assertFalse(hajoUt.getUtasLista().contains(utas));
    }

    @Test
    public void utasTeszt(){
        Utas utas = new Utas("nev", "kod");

        utas.penztKap(2000);

        // Feltételezés: el tud menni egy 1500 -as útra
        assertTrue(utas.beszallhat(1500));

        // Feltételezés: nem tud elmenni egy 1600-asra
        assertFalse(utas.beszallhat(1600));

        utas.penztKolt(400);

        // Feltételezés: már nem tud elmenni az 1200-asra
        assertFalse(utas.beszallhat(1200));

       // De el tud menni az 1100-asra
        assertTrue(utas.beszallhat(1100));
    }
}
```