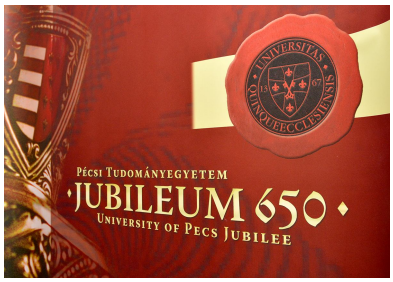
PTE JUBILEUM

Feladat:



Ha már a PTE jubileumával kezdődik az év, vegyük ki mi is a részünket belőle. Bár szerencsére a rendezvényeket ingyenesen lehetett látogatni, a feladat kedvéért fizetőssé tesszük őket.

Az ünnepséghez tehát rendezvények tartoznak. Minden egyes rendezvény a címével, időpontjával (jelenleg sima String) és a belépőjegy árával jellemezhető.
Természetesen vannak résztvevők is, sőt, a PTE azonosítóval rendelkezők még kedvezményt is kapnak. Minden résztvevőnek van neve (a név egyébként sohasem lehet egyedi azonosító, de most az egyszerűség kedvéért feltesszük, hogy minden név más). A PTE-s résztvevőt ezen kívül még egy azonosító is jellemzi.
Egy résztvevő akkor vesz részt egy adott rendezvényen, ha kifizeti a rendezvény részvételi díját. A PTE-s résztvevők egységesen 10% kedvezményt kapnak.
Minden egyes résztvevő esetén tudjuk felsorolni azokat a rendezvényeket, amelyeken részt vett az illető, illetve minden rendezvény esetén állapítsuk meg a résztvevők számát és a rendezvény bevételét.

Írjunk programot a rendezvények szimulálására: Olvassuk be a tervezett rendezvényeket, ill. a potenciális résztvevőket, a résztvevők kapjanak véletlenszerűen valamennyi zsebpénzt, majd rendezvényenként véletlenszerűen döntsék el, hogy megpróbálnak-e részt venni rajtuk, vagy sem. (A részvételi kedv kb. 80%-os.)

Egy lehetséges megoldás vázlata:

A feladatot OOP szemlélettel oldjuk meg. Ha úgy gondolja, nem érti biztosan a fogalmakat, érdemes a megoldás előtt elolvasni az **oop_alapok.pdf** fájlt.

A feladat megoldásához három alaposztályt definiálunk: a rendezvényt és a résztvevőket leíró osztályt, illetve mivel speciális résztvevők is vannak (a PTE-s polgárok), ezért az ő speciális tulajdonságaikat a résztvevők osztályának kiterjesztéseként definiált utód osztályban fogalmazzuk majd meg.

Az egyes osztályok feladata:

`Rendezveny:`
Azt kell figyelnie, hogy ha be akar lépni egy résztvevő, akkor egyáltalán részt vehet-e a rendezvényen. Ha igen, akkor növelni kell a résztvevők számát és a bevétel. Ha nem lennének speciális résztvevők, azaz mindenkinek azonos jegyárat kellene fizetnie, akkor a bevételt célszerű lenne mező helyett metódusként deklarálni, hiszen ez esetben a bevétel a résztvevőszám és a jegyár szorzata lenne. Mivel azonban lesznek majd kedvezményes árat fizető résztvevők is, ezért a bevételt az aktuális részvételi díj figyelembe vételével a rendezvényre való belépéskor számoljuk.

`Resztvevo:`
Hozzá a „részt vesz” esemény tartozik. Ekkor megmondjuk, hogy egy adott rendezvényen akkor vehet részt, ha ki tudja fizetni a rendezvény árát, vagyis van legalább ennyi pénze. Ha van, akkor fizet is, illetve az adott rendezvény bekerül az általa látogatott rendezvények listájába.
Első nekifutásra gondolhatjuk azt, hogy egy cselekvésről lévén szó, `void` metódust kell írnunk, de ha jobban végiggondoljuk, rájöhetünk, hogy később azt is tudnunk kell, hogy vajon valóban sikerült-e részt vennünk, vagyis ez a metódus nem `void`, hanem `boolean` lesz.

`PTEsResztvevo:`
Ő az utód osztály, ezért örökli az ős mezőit, metódusait. Az egységes kedvezmény miatt még kap egy statikus mezőt. Azt kell majd megoldanunk, hogy kezelni tudjuk valahogy a kedvezményt, mégpedig úgy, hogy minél kevesebbet ismételjünk meg az ős műveletei közül. Az nyilván nem lenne jó, ha a teljes `resztVesz()` metódust írnánk felül, hiszen akkor sok műveletet megismételnénk. Az lesz majd a használható ötlet, hogy egy olyan metódust írunk, amelyik csak a részvételi díjat határozza meg, és ezt írjuk felül. Az ősosztályban ez egy fölösleges lépésnek tűnik, hiszen közvetlenül is lekérdezhetnénk a rendezvény árát, de ez az ötlet kimondottan kényelmessé teszi majd az öröklődést.

Az osztályokhoz célszerű osztálydiagramot (UML) készíteni. Ezt a korábban tanultak alapján próbálja meg önállóan megoldani. Egy jó UML ábra sokat segíthet a program megírásakor.

Az UML elkészítésekor (vagy egyszerűbb esetben egy rövid vázlat megírásakor) azt kell végiggondolni, hogy milyen mezők (adattagok), és milyen metódusok alkotják majd az osztályt, illetve még ezeket is figyelembe kell vennünk:
- Milyen **konstruktor**okra van szükség? Ilyenkor azt gondoljuk végig, hogy milyen adatok szükségesek egy példány létrejöttéhez. Több konstruktor definiálásának akkor van értelme, ha esetleg többféle módon akarjuk „adminisztrálni”, vagyis létrehozni a példányokat.
- Mely adatokhoz kellenek **getter**ek. Az adattagok (biztonsági okok miatt) mindig `private` láthatóságúak. Viszont lehetőséget adhatunk rá, hogy lekérdezhessük az értéküket, erre szolgálnak a `get…()` metódusok. Az, hogy csak metóduson keresztül érhetjük el az adatot, lehetőséget ad rá, hogy akár teljesen megtiltsuk az elérést, akár pedig a metódusban megfogalmazott feltételekhez kössük azt.
- Mely adatokhoz kellenek **setter**ek. Ekkor azt gondoljuk végig, hogy melyeket engedjük kívülről módosítani ennek a metódusnak a segítségével.

Különösebb indoklás és részletezés nélkül összefoglalóan leírom az egyes osztályok mezőit, illetve azt, hogy ezek szerepelnek-e a konstruktor paraméterlistáján (i/n), van-e hozzájuk getter/setter.

`Rendezveny:`

Mező	konstruktor	getter	setter
`cím`	i	i	?
`idoPont`	i	i	i
`jegyAr`	i	i	i
`resztvevokSzama`	n	i	n
`bevetel`	n	i	n

Az, hogy megengedjük-e a cím módosítását, a megrendelő elképzelésén múlik. A résztvevők száma és a bevétel számolt érték, vagyis nem írhatunk hozzájuk settert.

`Resztvevo:`

Mező	konstruktor	getter	setter
`nev`	i	i	n
`penz`	?	n	?
`rendezvenyek`	n	i(!)	n

Bár kérdőjelesen szerepel, vagyis elvileg megadható a konstruktorban az, hogy induláskor mennyi pénze van az illetőnek, de az emberek általában ezt nem szokták elárulni, vagyis életszerűbb, ha nem szerepel a konstruktorban. Azt pláne nem szokták megengedni, és nem kötik a rendezvényszervezők orrára, hogy aktuálisan mennyi pénzük van, ezért nem írunk gettert. A pénzhez tartozó setter azért kérdéses, mert ha garantáltan csak egyszer szeretnénk pénzt adni az illetőnek, akkor jó, ha esetleg többször is, akkor figyelni kell arra is, hogy ne vesszen el a régi pénze – ezt majd kicsit részletesebben a kódolási részben.
A meglátogatott rendezvények listája menet közben, egy metódus hatására alakul ki, ezért sem a konstruktor paramétereként nem szerepelhet, sem settert nem írhatunk hozzá. Gettert azonban nyilván, ugyanakkor figyelnünk kell majd rá, hogy ez a getter ne sértse az egységbezárás elvét, vagyis ne az eredeti listát adjuk át, hanem ennek csak egy másolatát.

A speciális résztvevőket, azaz a PTE polgárokat a `Resztvevo` osztály utód osztálya írja majd le. Az utód osztályban csak azt tüntetjük fel, ami új vagy módosítás az őshöz képest.

`PTEsResztvevo:`

Mező	konstruktor	getter	setter
`pteAzonosito`	i	i	n
`kedvezmenySzazalék`	n	i	i

Statikus mező soha nem szerepelhet a konstruktor paraméterlistáján, az mindig setteren keresztül kap értéket.

Lássuk a kódot!

Bár célszerű egy-egy konstruktort, settert, gettert „gyalog” is végiggépelni, hogy alaposabban végig tudja gondolni a szerkezetüket, de a többi generálható: NetBeans-ben az Alt+Ins gomb hatására megjelenik a generálható elemek listája (konstruktor, getter, setter, toString(), stb.).

Az osztályok kódját a generálható setterek, getterek nélkül másolom ide, de – ahogy az elején tettük – ezeket mindig végig kell gondolni.

Rendezveny:

```
public class Rendezveny {

    private String cim;
    private String idoPont;
    private int jegyAr;
    private int resztvevokSzama;
    private int bevetel;

    public Rendezveny(String cim, String idoPont, int jegyAr) {
        this.cim = cim;
        this.idoPont = idoPont;
        this.jegyAr = jegyAr;
    }

    /**
     * Ha egy résztvevő részt vesz ezen a rendezvényen, akkor növekszik a
     * résztvevők száma is és a bevétel is.
     *
     * @param resztvevo
     */
    public void resztVesz(Resztvevo resztvevo) {
        if (resztvevo.belep(this)) {
            resztvevokSzama++;
            bevetel += resztvevo.reszveteliDij(this);
        }
    }

    @Override
    public String toString() {
        return cim + ", időpontja: " + idoPont + ", jegyár: " + jegyAr;
    }
```

A `belepes()` metódus előtt látható, /** kezdetű kommentet dokumentációs kommentnek nevezik, ezekből egy gombnyomás hatására generálható egy html formátumú dokumentáció. Ez a komment is generálható: /** + Enter.

Resztvevo:

```
public class Resztvevo {

    private String nev;
    private int penz;
    private List<Rendezveny> rendezvenyek = new ArrayList<>();

    public Resztvevo(String nev) {
        this.nev = nev;
    }
```

```java
/**
 * Ha ki tudja fizetni a rendezvény részvételi díját, akkor "részt vesz"
 * rajta, vagyis az általa látogatott rendezvények listájához ez a
 * rendezvény is hozzáadódik.
 *
 * @param rendezveny
 * @return
 */
public boolean belep(Rendezveny rendezveny){
    int reszveteliDij = reszveteliDij(rendezveny);
    if(penz >= reszveteliDij){
        this.fizet(reszveteliDij);
        rendezvenyek.add(rendezveny);
        return true;
    }
    return false;
}

/**
 * Ez lesz az a metódus, amelyet az utód felül tud írni, és nála már
 * a részvételi díj kedvezményes lesz.
 *
 * @param rendezveny
 * @return
 */
protected int reszveteliDij(Rendezveny rendezveny) {
    return rendezveny.getJegyAr();
}

private void fizet(int dij) {
    if(penz >= dij) penz -= dij;
}

@Override
public String toString() {
    return nev;
}
```

Néhány megjegyzés:

A `rendezvenyek` lista deklarálásakor a típust a `List<>` interfész határozza meg, de mivel interfészt nem lehet példányosítani, példányosításkor egy, az interfészt implementáló osztályt kell megadni – jelenleg az `ArrayList` osztályt példányosítottuk. Ha ez a megoldás még idegen, ne zavarja, majd kicsit később részletesebb magyarázatot kap rá.

A `reszveteliDij()` metódust célszerű `protected`-ként kezelni, hogy csak az utód tudja felülírni.

A `fizet()` metódussal kapcsolatban jogosan merül fel néhány kérdés:

1. Először is az, hogy egyáltalán szükség van-e rá, miért nem lehet a `resztVesz()` metóduson belül azonnal levonni a részvételi díjat? Természetesen úgy is meg lehetne oldani a feladatot. Esetünkben kicsit szubjektív, hogy írunk-e külön metódust, vagy sem. A mostani megoldás mellett az szól, hogy

- így talán olvashatóbb a kód, hiszen jó metódusnévvel jelzi, hogy mit is csinálunk (a jó metódus- vagy változónév roppant fontos a kód olvashatósága szempontjából);
- később könnyebben bővíthető az osztály, vagy úgy, hogy egy esetleg később megírandó metódusból is meg tudjuk hívni az osztályon belül, vagy úgy, hogy a módosítót publikussá tesszük, és ekkor a példányosítás után bárhonnan meg lehet hívni.

2. Eleve csak akkor hívtuk meg ezt a metódust, ha teljesült a feltétel (penz >= reszveteliDij), akkor miért kell ezt megismételni a `fizet()` metóduson belül is? Ha garantáltan csak ennyi az osztály, és az életben soha nem akarjuk bővíteni, akkor persze fölösleges (mint ahogy az is, hogy külön metódust írjunk), de ha gondolunk az esetleges bővíthetőségre, akkor viszont fontos, hogy ezt a metódust bárhonnan lehessen hívni, vagyis nincs rá semmi garancia, hogy a hívás előtt ellenőriznénk a feltételt.
Persze, erre jogos válasz az, hogy ha a `fizet()` metódusban szerepel ez a feltétel, akkor miért kell a `resztVesz()` metódusban is? Azért, mert ott a feltételhez nem csak a fizetés tartozik, hanem más is – esetünkben a lista bővítése.

A most megbeszéltekkel kapcsolatban még egy **fontos**, általános megjegyzés: Nagyon lényeges alapelv az, hogy könnyen módosítható, bővíthető legyen a programunk. Ezt akár úgy is biztosíthatjuk, ahogy most a `fizet()` metódus kapcsán láttuk.

Külön kiemelem a `rendezvenyek` listához tartozó gettert. Ha „sima” gettert írnánk, vagyis a getter ezt a listát adná vissza, akkor bármelyik másik osztályban módosítani tudnánk ezt a listát (pl. `resztvevo.getRendezvenyek().add(rendezveny);`), ami erősen sérti az egységbezárás elvét. Emiatt nem magát a listát, hanem annak egy másolatát adjuk át. Természetesen ezt is lehet módosítani a másik osztályban, de ha ismét a gettert hívjuk meg, akkor ismét az eredeti listát kapjuk, vagyis az eredetit nem tudjuk módosítani. Figyeljen rá, hogy mindig ezt a megoldást válassza. (Illetve választhatja azt is, hogy módosíthatatlanná tesszük a listát, de erről még nem volt szó.)

Tehát a getter:

```
public List<Rendezveny> getRendezvenyek() {
    return new ArrayList<>(rendezvenyek);
}
```

Van még egy adósságom: hogyan kap pénzt az illető?

Egyik, nem túl szép megoldás az, hogy „hagyományos” setteren keresztül. Evvel az a baj, hogy ha egynél többször hívjuk meg, akkor a későbbi pénzösszeg felülírja a korábbit, ami nem túl életszagú.

A másik lehetőség, hogy nem „hagyományos” settert írunk, hanem ilyet:

```
    public void setPenz(int penz) {
        this.penz += penz;
    }
```

vagyis az újabb (azaz a paraméterben szereplő) pénzadag hozzáadódik az eddigiekhez. Ez már életszagúbb, működőképes is, csak ha ezt a metódusnevet használjuk, akkor nem igazán olvasható a kód. Ezért célszerű átnevezni (az átnevezés sohasem átgépelés, hanem jobb egérgomb, refactor, rename, mert ekkor minden előfordulást kijavít).
Vagyis ezt a metódust célszerű használni:

```
    public void penztKap(int penz) {
        this.penz += penz;
    }
```

Végül következzen a `Resztvevo` osztály utódja. Java-ban az utódlásra a „kiterjesztés” szót használjuk (`extends`).

PTEsResztvevo:

```
public class PTEsResztvevo extends Resztvevo{
    private String pteAzonosito;
    private static double kedvezmenySzazalek;

    public PTEsResztvevo(String nev, String pteAzonosito) {
        super(nev);
        this.pteAzonosito = pteAzonosito;
    }

    @Override
    protected int reszveteliDij(Rendezveny rendezveny) {
        return (int) (super.reszveteliDij(rendezveny)* (1-kedvezmenySzazalek/100));
    }
```

Jöhet a vezérlés:

```
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        new Vezerles().start();
    }
}
```

Gyakorlaton vettünk példát a billentyűzetről való olvasásra is, ezt most kihagyom, ha át akarja nézni, a lényege benne van a gyak1_temp.txt fájlban.

Mielőtt idemásolnám a kódot, még néhány megjegyzés:

1. A kódban használunk majd néhány konstansot (pl. a kedvezmény mértéke, vagy a fájlok neve, stb.). Azt már mindenki tudja, hogy NEM ÉGETÜNK be adatokat a kódba. A beégetés elkerülésének egyik módja, hogy a konstansokat `final` változókként a vezérlő osztály elejére írjuk, ekkor könnyen lehet majd módosítani. (Egyéb szokásos megoldás: a konstansokat kirakjuk egy külön osztályba, vagy egy config fájlban adjuk meg – ez utóbbi a legszebb megoldás, ekkor ugyanis egyáltalán nem kell hozzányúlni a kódhoz.)

2. A feladat szerint a résztvevők beolvasáskor kapnak valamennyi pénzt, illetve csak akkor vesznek részt egy rendezvényen, ha van hozzá kedvük. Természetesen ezt meg lehetne oldani úgy, hogy minden egyes alkalommal bekérjük a billentyűzetről, hogy mennyi pénzt kapjon az illető, vagy hogy van-e kedve elmenni a rendezvényre, de remélem, senki sem gondolja komolyan, hogy ezt így szeretné csinálni. A billentyűzetről való olvasásnál nem sok unalmasabb dolog van ☺. Ezért az olvasás helyettesítésére véletlen értékeket generálunk. Ezt lehet hasonlóan, mint a C#-ban, vagyis egy `Random` osztály segítségével, de szerintem egyszerűbb, ha a `Math` osztály `rand()` metódusát használjuk. Ez egy [0,1) közötti `double` értéket ad vissza, ha ezt megszorozzuk a határokkal, akkor bármekkora véletlen értéket tudunk generálni. A pontos kiszámítási módot gondolja végig!

Különösebb magyarázat nélkül lássuk a kódot – remélem, hogy a kód magyarázza magát.

```
public class Vezerles {

    private final double KEDVEZMENY_SZAZALEK = 10;
    private final String RENDEZVENY_ELERES = "rendezvenyek.txt";
    private final String RESZTVEVOK_ELERES = "resztvevok.txt";
    private final int ALSO_PENZ = 1000;
    private final int FELSO_PENZ = 10000;
    private final double KEDV_SZAZALEK = 0.8;

    private List<Rendezveny> rendezvenyek = new ArrayList<>();
    private List<Resztvevo> resztvevok = new ArrayList<>();


    public Vezerles() {
    }

    void start() {
        adatBevitel();
        adatKiiras();
        jubileum();
        eredmenyKiiras();
    }
```

```
private void adatBevitel() {

    PTEsResztvevo.setKedvezmenySzazalek(KEDVEZMENY_SZAZALEK);

    try {
        Scanner fajlScanner = new Scanner(new File(RENDEZVENY_ELERES));
        String cim, idoPont;
        int ar;
        String sor, adatok[];
        Rendezveny rendezveny;
        while (fajlScanner.hasNextLine()) {
            sor = fajlScanner.nextLine();
            adatok = sor.split(";");
            cim = adatok[0];
            idoPont = adatok[1];
            ar = Integer.parseInt(adatok[2]);
            rendezveny = new Rendezveny(cim, idoPont, ar);
            rendezvenyek.add(rendezveny);
        }
        fajlScanner.close();

        fajlScanner = new Scanner(new File(RESZTVEVOK_ELERES));
        Resztvevo resztvevo = null;
        while (fajlScanner.hasNextLine()) {
            sor = fajlScanner.nextLine();
            adatok = sor.split(";");
            if(adatok.length == 1) {
                resztvevo = new Resztvevo(adatok[0]);
            }
            if(adatok.length == 2) {
                resztvevo = new PTEsResztvevo(adatok[0], adatok[1]);
            }
            resztvevo.setPenz((int) (Math.random()*
                                (FELSO_PENZ - ALSO_PENZ + 1) + ALSO_PENZ));
            resztvevok.add(resztvevo);
        }
        fajlScanner.close();

    } catch (FileNotFoundException ex) {
        Logger.getLogger(Vezerles.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Megjegyzés: Tanulták, hogy fájlból való olvasáskor kötelező a kivételkezelés, de a Java ezt nem bízza a belátásunkra, hanem ténylegesen kötelezővé teszi, vagyis enélkül le sem fordul a program. De ne ijedjen meg, ha leírta a
`Scanner fajlScanner = new Scanner(new File(RENDEZVENY_ELERES));`
sort, akkor azonnal felkínálja, hogy generálja.
Egyébként azt is, hogy importálni kell a `Scanner` osztályt a `util` csomagból, de ha kódkiegészítéssel írja az osztály nevét, akkor azonnal be is írja ezt az importot.

```
    private void adatKiiras() {
        System.out.println("A rendezvények: ");
        for (Rendezveny rendezveny : rendezvenyek) {
            System.out.println(rendezveny);
        }
        System.out.println("\nA résztvevők:");
        for (Resztvevo resztvevo : resztvevok) {
            System.out.println(resztvevo);
        }
    }

    /**
     * Minden egyes rendezvény esetén "végigkérdezzük" a résztvevőket, és ha
     * a "van kedve" - vagyis a véletlen érték kisebb a kedv-százaléknál,
     * akkor részt vesz a rendezvényen.
     */
    private void jubileum() {
        for (Rendezveny rendezveny : rendezvenyek) {
            for (Resztvevo resztvevo : resztvevok) {
                if(Math.random() < KEDV_SZAZALEK)rendezveny.resztVesz(resztvevo);
            }
        }
    }

    private void eredmenyKiiras() {
        System.out.println("\nÖsszesítés: ");
        for (Rendezveny rendezveny : rendezvenyek) {
            System.out.printf("%10s, %3d résztvevő, %5d Ft bevétel \n",
                    rendezveny.getCim(),
                    rendezveny.getResztvevokSzama(),
                    rendezveny.getBevetel());
        }

        for (Resztvevo resztvevo : resztvevok) {
            System.out.println("\n" + resztvevo +
                                "ezeken a rendezvényeken vett részt:");
            for (Rendezveny rendezveny : resztvevo.getRendezvenyek()) {
                System.out.println(rendezveny);
            }
        }
    }
}
```

Szorgalmi hf:
Ha végiggondolják, látható, hogy redundáns a megoldás, hiszen a rendezvényeket sok példányban tároljuk (minden egyes résztvevőnél). Gondolkozzon el rajta, hogy hogyan lehetne megszüntetni ezt a redundanciát.