JAVA-TRAVEL

Feladat

Ha már láttuk, hogy milyen szép a Java szigete, tegyünk ott egy hajókirándulást, és tegyük ezt lehetővé mások számára is.

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

Minden hajóút esetén adott a hajó neve és egy, az utat definiáló egyedi azonosító, 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 hajóúton 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.

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.

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

További feladatok:

- A szimuláció során egy-egy véletlen utas időnként kapjon valamennyi pénzt.
- Ha már költségekbe veri magát, akkor utazhasson az utas, vagyis az utazik() metódusának hatására az aktuális utat adja hozzá a hajóútjai listájához.
- Számolja ki a cég teljes bevételét. Gondolja végig, hogy ehhez mit, melyik osztályban és hogyan kell megadni.
- A „szokásosak”: melyik hajónak legtöbb a bevétele, melyiken utaznak a legtöbben, legkevesebben, stb.

Még **további** feladat: bővítse ki a szereplő osztályokat további értelmes funkciókkal!

Egy lehetséges megoldás vázlata:

Ebben a vázlatban nagyjából csak addig jutunk el, mint órán, vagyis még nem foglalkozunk a kedvezményekkel.

1. Először néhány szót általában az OOP szemléletről – ha evvel tisztában van, akkor ezt a részt át is ugorhatja.

A feladat megoldása során, illetve még a konkrét megoldás előtt a feladat végiggondolásakor azt kell eldöntenünk, hogy milyen fogalmakkal akarunk dolgozni.
Ez a projekt három fogalom köré épül: hajóút, utas, utazási iroda.

Ezekről a fogalmakról mindenkinek van elképzelése, de ahhoz, hogy egy projektbe tudjuk foglalni őket, pontosítani kell, hogy mit is értünk rajtuk. A pontosítás azt jelenti, hogy kiemeljük a feladatmegoldás szempontjából fontos tulajdonságokat és viselkedést. A többivel nem foglalkozunk. Mondhatjuk azt is, hogy megadjuk a fogalom vázát (vagy tervrajzát). Ez a „tervrajz", vagy másképpen ez a pontosított fogalom az **osztály**. Ez egy leírást ad arról, hogy a továbbiakban mit is értünk az aktuális fogalom alatt.
De ez még valóban csak fogalom vagy tervrajz, dolgozni azonban konkrét dolgokkal tudunk (konkrét hajóutakkal és konkrét utasokkal). Ez a konkretizálás a **példányosítás**, amelynek során kapjuk az objektumokat. Ezekkel az **objektum**okkal tudunk majd dolgozni.
Ezekre sokszor gondolhatunk úgy, mint valamilyen adminisztrálandó adatra – természetesen összetett adatra, vagyis azok együttesére, amelyek ezt a bizonyos objektumot jellemzik.

Programozóként azt is mondhatjuk, hogy az osztály definiálásakor létrehozunk egy saját típust, pontosan ugyanolyan típust, mint amilyen pl. a String. Ez utóbbi is osztály, ennek is vannak tulajdonságai (ezeket nevezzük mezőknek vagy adattagoknak), és ennek is van viselkedése, ezt írják le a metódusok.

Nézzük meg ezeket a pontosításokat:

Hajóút:
A feladat szempontjából csak ezek a fontosak:
Tulajdonságok (mezők): az utat végző hajónak van neve, az útnak van azonosítója, ismernünk kell, hogy mennyi a maximálisan szállítható utaslétszám, mennyi az aktuális útiköltség, illetve még az utasok listáját is.
A *viselkedés*éből (metódus) nem sok érdekli a feladat kitűzőjét, összesen csak annyi, hogy felszáll rá egy-egy utas. Az már, hogy a kényelmesebb kiíratás kedvéért még a kiírandó szöveget is itt határozzuk meg, inkább a programozó igénye, de persze, ez is fontos igény.

Ha majd megpróbálja kibővíteni a feladatot, akkor elgondolkozhat további funkciókon is, pl. ilyesmin: az utas le is szállhat ☺, úszik a hajó, és közben számolja, hogy hány km-t tett meg élete során. Vagy: ismerve a gyártási évet és az előírt karbantartási időintervallumokat, mikor kell legközelebb műszaki vizsgára vinni, stb. Ezeknek a funkcióknak a kezeléséhez nyilván újabb tulajdonságokat és metódusokat kell majd definiálni.

Utas:
A feladat szempontjából csak ezek a fontosak:
Tulajdonságok (mezők): az utasokat a nevük és kódjuk alapján különböztetik meg, az is fontos, hogy legyen pénzük. A feladatot kitűző cégnek van még egy speciális előírása: a kifizetett jegy mellett mindenkinek rendelkeznie kell egy előírt pénzmennyiséggel ahhoz, hogy hajóra szállhasson. Ez az előírt összeg azonos minden egyes utas esetén. (Elismerem, hogy ez egy kicsit(?) erőltetett feltétel, de talán ha kauciónak nevezzük, ahogy az egyik kollégájuk javasolta, akkor elfogadhatóbb.)
A *viselkedés*éből (metódus) ezek a fontosak: kaphat valamennyi pénzt, illetve költhet valamennyit, és megállapítható, hogy beszállhat-e a hajóba, vagyis eleget tesz-e bizonyos feltételeknek. Első körben (vagyis ezen a szinten) az lényegtelen, hogy mik ezek a feltételek.

Természetesen itt is kitalálhat újabb funkciókat, pl. csak bizonyos kor fölöttiek szállhatnak fel, vagy számolhatja a hajón töltött napokat (ez már kicsit összetettebb funkció), vagy…

Utazási iroda:
Lényegileg ő a megbízó, vagyis az ő igényeit majd a vezérlő osztály írja le.

A fentiek alapján már el is készíthető az osztálydiagram (UML). Ezt a leírtak és 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 végiggondolásakor) még további dolgokat kell figyelembe 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. (Mondjuk, az egyik hajóút esetén már az első adminisztráció során ismert a név, azonosító és a maximális létszám, a másik hajóút esetében viszont először csak a nevet és azonosítót tudják, és csak valamikor később tudják megadni a létszámot.)
- 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.

Ezt is gondoljuk át a két osztályra vonatkozóan:

Hajóút:

Mező	konstruktor	getter	setter
`nev`	i	i	?
`azonosito`	i	i	n
`maxUtasSzam`	?	i	i
`utiKoltseg`	n	i	i
`utasLista`	n	i(!)	n

A `maxUtasSzam` szerepelhet a konstruktor paraméterei között, de nem muszáj, hogy szerepeljen. Akár két konstruktort is írhatunk.
Az, hogy megengedjük-e a név módosítását, a megrendelő elképzelésén múlik.

Az `utasLista` természetesen minden út során módosul, de kizárólag csak a felszállás hatására. A gettere mellett azért van a felkiáltójel, mert egy listát nem szabad sima getterrel átadni, hiszen akkor manipulálható lenne, hanem vagy módosíthatatlanra kell alakítanunk, vagy csak egy másolatot szabad átadnunk, erről majd később lesz szó kicsit részletesebben.

Utas:

Mező	konstruktor	getter	setter
`nev`	i	i	?
`kod`	i	i	n
`penz`	?	n	n
`alapToke`	n	i	i

Bár kérdőjelesen szerepel, vagyis elvileg megadható a konstruktorban az, hogy induláskor mennyi pénze van az utasnak, de az emberek általában ezt nem szokták elárulni, vagyis életszerűbb, ha nem szerepel a konstruktorban. Azt pláne nem szokták megengedni, és nem kötik az utazási iroda orrára, hogy aktuálisan mennyi pénzük van, ezért nem írunk gettert.

Az, hogy egy út alatt megváltozhat-e egy utas neve, nem túl valószínű, de ha a megrendelő megengedi (vagyis pl. direkt esküvős utat szervez), akkor esetleg lehet. De ez a megrendelő és a programozó közötti megállapodás kérdése.

A `penz` csak a pénzt kap, ill. pénzt költ metódus során változhat. Az `alapToke` pedig statikus, hiszen mindenki számára egyforma értékű.

2. A kód szintű megvalósítás (getterek, setterek nélkül):

A `HajoUt` osztállyal kezdjük.

```
public class HajoUt {

    private String hajoNev;
    private String azonosito;

    private int maxUtasSzam;
    private int utiKoltseg;
    private List<Utas> utasLista = new ArrayList<>();

    public HajoUt(String hajoNev, String azonosito, int maxUtasSzam){
        this.hajoNev = hajoNev;
        this.azonosito = azonosito;
        this.maxUtasSzam = maxUtasSzam;
    }
```

A felszállásnál kicsit elidőzünk. Ennek során egy utas száll majd fel a hajóra, vagyis a metódus paraméterként tartalmazza ezt az utast (azaz egy `Utas` típusú példányt). A kód megírásakor elég annyit tudnunk, hogy létezik ilyen osztály. Sőt, ha az írás pillanatában még nem létezik, akkor a NetBeans hajlandó generálni ezt az osztályt.
A felszállás feltételhez kötött: az első feltétel az, hogy egyáltalán legyen hely, és ha van, akkor az utas beszállhasson. A hajót (vagy mondhatjuk azt, hogy a beszállást irányító tengerészt) nem érdekli az, hogy az utasnak mit kell teljesítenie ahhoz, hogy beszállhasson, őt csak az érdekli, hogy az utas lobogtatja-e a beszállásra jogosító jegyét vagy sem, vagyis az `utas` példány `beszallhat()` metódushívásának értéke `true` vagy sem. Hogy konkrétan mik ezek a feltételek, azt majd az `Utas` osztályban kell megfogalmaznunk. Itt csak egyetlen dolgot kell tudni, mégpedig azt, hogy az, hogy beszállhat-e az utas, egyetlen külső körülménytől függ, az útiköltségtől.

Van még egy feltétel: ugyanaz az utas nem szállhat fel kétszer, vagyis azt is ellenőriznünk kell, hogy az utaslista nem tartalmazza-e az aktuális utast.
Ha a feltételek teljesülnek, akkor az utast hozzá kell adnunk az utaslistához. Ha nem teljesülnek a feltételek, akkor nem történik semmi.

```
/**
 * Felszáll egy utas, ha tud.
 * Ha nem tud, nem történik semmi.
 *
 * @param utas
 */
public void felszall(Utas utas) {
    if (utasLista.size() < maxUtasSzam
            && utas.beszallhat(utiKoltseg)
            && !utasLista.contains(utas)) {
        utasLista.add(utas);
        utas.penztKolt(utiKoltseg);
    }
}
```

Megjegyzések:

1. Egy esetleges bővítés során elképzelhető, hogy módosítani kell a visszatérési típust. Sőt, akár már most is megteheti, gondolva egy esetleges továbbfejlesztésre. Célszerű lehet `boolean` típusra deklarálni (és ha majd megfigyeli, láthatja, hogy sok beépített metódus is ilyen), ekkor ugyanis egy esetleges továbblépés során azt is lehetne vizsgálni, hogy mi történik, ha az utas nem szállhatott fel. A jelen feladatban erre nincs szükség, de programozás során érdemes gondolni a továbbfejlesztésre is.

2. A /** - gal kezdődő komment az úgynevezett javadoc komment. Ezek alapján lehet generálni a dokumentációt, illetve ez jelenik meg az online help-ben is.
Generálása: közvetlenül a metódus előtt: /** + enter.

A `toString()` metódusban nincs semmi újdonság:

```
@Override
public String toString() {
    return "Az út azonosítója: " + azonosito +
           "\n a hajó neve: " + hajoNev +
           "\n utiköltség: " + utiKoltseg + " peták" +
           "\n utasLista=" + utasLista;
}
```

A metódus fölött látható `@Override` egy úgynevezett annotáció. Erről később még lesz szó, most csak annyit, hogy a metódus akkor is működik, ha ezt kitöröljük, de nem célszerű kitörölni. Hogy miért nem, arról majd később lesz szó.

Az utas lista kiíratása így elég csúnya lesz, próbálja meg szépíteni (vagyis egy ciklusban hozzáadni a korábbiakhoz az aktuális utas `toString()` értékét.)

Szó volt róla, hogy listához nem írunk a listát direktben visszaadó gettert, mert ekkor a listát kívülről is lehetne módosítani, pedig az sérti az egységbezárás elvét, és biztonsági kockázatot jelent. Helyette a lista másolatát adjuk át (később lesz még szó másfajta megoldásról is). Erre akár mi magunk is írhatnánk egy egyszerű kis ciklust, de sokkal egyszerűbb, ha az ArrayList osztály megfelelő konstruktorát alkalmazzuk:

```
/**
 * Másolatot adunk át!!
 *
 * @return
 */
public List<Utas> getUtasLista() {
    return new ArrayList<>(utasLista);
}
```

Az eddig megbeszéltek alapján az `Utas` osztály könnyen megírható. A kódja getterek és setterek nélkül:

```
public class Utas {

    private String nev;
    private String kod;
    private int penz;
    private static int alapToke;

    public Utas(String nev, String kod) {
        this.nev = nev;
        this.kod = kod;
    }
```

```java
/**
 * Az utas pénzt kap, ennek hatására a már
 * meglévő pénze a paraméterben lévő pénzértékkel növekszik.
 *
 * @param penz
 */
public void penztKap(int penz){
    this.penz += penz;
}

/**
 * Az utas pénzt költ, ennek hatására a pénze csökken a
 * paraméterben lévő értékkel, feltéve, hogy még ki
 * tud fizetni ennyit.
 * Ha nem, akkor nem történik semmi.
 *
 * @param penz
 */
public void penztKolt(int penz){
    if(this.penz >= penz){
        this.penz -= penz;
    }
}
```

```java
/**
 * Beszállhat, ha ki tudja fizetni az útiköltséget,
 * és még marad legalább alapToke mennyiségű pénze.
 *
 * @param utiKoltseg
 * @return
 */
public boolean beszallhat(int utiKoltseg){
    if(this.penz >= utiKoltseg + alapToke) {
        return true;
    }
    return false;
}

@Override
public String toString() {
    return nev + "(" + kod + ')';
}
```

Néhány megjegyzés:

1. A `penztKolt()` metódust – a hajóra való felszálláshoz hasonlóan – egy esetleges későbbi továbbfejlesztésre gondolva célszerű lehet `boolean`-ként deklarálni.

2. A `beszallhat()` metódus tömörebben:

```java
public boolean beszallhat(int utiKoltseg){
    return this.penz >= utiKoltseg + alapToke;
}
```

Végül beszéljünk a vezérlésről Azt gondolom, ehhez nem kell magyarázatot fűzni:

A `Main` osztály `main()` metódusa:

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

A `Vezerles` osztály:

```
class Vezerles {

    // final jelentése: nem módosítható, azaz konstans
    private final int kotelezoToke = 5000;
    private final int FELSO_AR = 50000;
    private final int ALSO_AR = 20000;

    private List<HajoUt> hajoUtak = new ArrayList<>();
    private List<Utas> utasok = new ArrayList<>();


    void start() {
//        probaAdatBevitel();
        adatBevitel();
        utaztatas();
        kiiratas();
    }

    // Csak arra szolgál, hogy lásson példát billentyűzetről való
    // olvasásra is.
    private void probaAdatBevitel() {
        Scanner scanner = new Scanner(System.in);
        System.out.print("A hajó neve: ");
        String nev = scanner.nextLine();
        System.out.print("azonosítója: ");
        String azonosito = scanner.nextLine();
        System.out.print("max utasszám: ");
        int maxUtasSzam = scanner.nextInt();
        hajoUtak.add(new HajoUt(nev, azonosito, maxUtasSzam));
    }
```

```
private void adatBevitel() {
    // Beállítjuk az utasok alaptőkéjét.
    Utas.setAlapToke(kotelezoToke);

    // kötelező kivételkezelés van, lehet generáltatni
    try {
        Scanner fileScanner = new Scanner(new File("hajoutak.txt"));
        String sor;
        String[] adatok;
        while (fileScanner.hasNextLine()) {
            sor = fileScanner.nextLine();
            // Pearl of Java;PJ001;200
            adatok = sor.split(";");
            hajoUtak.add(new HajoUt(adatok[0], adatok[1],
                    Integer.parseInt(adatok[2])));
        }
        fileScanner.close();

        fileScanner = new Scanner(new File("utasok_sima.txt"));
        Utas utas;
        while (fileScanner.hasNextLine()) {
            sor = fileScanner.nextLine();
            // Jani;HUJ001;30000
            adatok = sor.split(";");
            utas = new Utas(adatok[0], adatok[1]);
            utas.penztKap(Integer.valueOf(adatok[2]));
            utasok.add(utas);
        }
        fileScanner.close();

        // a generált kivétel-elkapás - log fájlba írja a hibaüzenetet
    } catch (FileNotFoundException ex) {
        Logger.getLogger(Vezerles.class.getName()).
                                        log(Level.SEVERE, null, ex);
    }
}
```

```java
    /**
     * Minden hajóút esetén beállít egy véletlen útiköltséget,
     * majd valahányszor véletlenszerűen kiválaszt egy-egy utast
     * és egy-egy hajóutat, és a kiválasztott hajóra felszáll a
     * kiválasztott utas.
     */
    private void utaztatas() {
        int valahany = 30;
        int utasIndex, hajoIndex;
        int utiKoltseg;

        for (HajoUt hajoUt : hajoUtak) {
            // ALSO_AR <= utiKoltseg < FELSO_AR
            utiKoltseg = (int) (Math.random()*(FELSO_AR - ALSO_AR)+ALSO_AR);
            hajoUt.setUtiKoltseg(utiKoltseg);
        }

        for (int i = 0; i < valahany; i++) {
            utasIndex = (int) (Math.random()*utasok.size());
            hajoIndex = (int) (Math.random()*hajoUtak.size());
            // A kiválasztott indexű hajóra felszáll a kiválasztott utas.
            hajoUtak.get(hajoIndex).felszall(utasok.get(utasIndex));
        }
    }

    private void kiiratas() {
        System.out.println("Hajóutak:");
        for (HajoUt hajoUt : hajoUtak) {
            System.out.println(hajoUt);
        }

        System.out.println("\nUtasok: ");
        for (Utas utas : utasok) {
            System.out.println(utas);
        }
    }
}
```