Rövid(? ☺) segítség a 4. gyakorlaton vett feladat megoldásához

Feladat:

A feladat lényege:
Adatfájlban lévő adatok alapján létre kell hozni sima és alkoholos italokat, majd különböző módon rendezve megjeleníteni az itallapon.
Az itallapra kattintva mellette olvasható a kiválasztott ital alkoholfoka – persze, csak ha alkoholos italról van szó.
Lehet rendelni, ekkor megjelenik a rendelt mennyiség ára.

Egyelőre ezt beszéljük meg – zömében csak vázlatosan.

1. Először is létre kell hoznunk egy `JFrame` alapú osztályt (mondjuk, `InditoFrame` néven). Beállíthatjuk a szélességét, magasságát, feliratát, stb.

2. Hozzunk létre egy `JPanel` alapú osztályt (mondjuk, `KocsmaPanel` néven). Erre a panelre rakjuk fel az „Itallap” feliratot, és az árlapként szolgáló `JList`-et. Ennek legyen a neve mondjuk `lstItalLap`.

Ennyi elég is ahhoz, hogy máris megpróbáljuk rárakni a panelt a frame-re.
A kész panel ugyanúgy húzható rá a frame felületére, mint a „gyári” komponensek. Egyetlen feltétel, hogy helyesen lefordúljon.

A ráhelyezés módja: a frame-t állítsuk `Border Layout`-ra. (Frame felületén jobb egérgomb, Set Layout – itt lehet beállítani.)

A többi komponenst később is felrakhatjuk, futáskor frissül majd, de csak akkor, ha már van rajta legalább egy komponens. Emiatt nem érdemes az üres panelt azonnal rárakni a frame-re. Viszont célszerű minél hamarabb, mert akkor sem sikerül majd ráhúzni, ha bár lefordul a program, de tartalmi hiba (pl. végtelen ciklus, nullpointer exception) van a kódban. Vagyis az a legjobb, ha akkor próbálja meg ráhúzni, amikor még csak egy-két komponens van rajta, lehetőleg minden egyéb kód nélkül. Sőt, mivel az egész projektnek le kell fordulnia, vagyis más osztályokban sem lehet hiba, ezért célszerű a felület kialakításával kezdenünk a munkát.

3. Kezdjük el a programozást!

a/ Adatbevitel.

Fájlból már mindenki tud olvasni, ezért azt nem részletezem túlzottan, de a biztonság kedvéért néhány szót mégis ejtek róla.

Az a jó projekt-struktúra, ha minden fontos részletet külön írunk meg, külön osztályba szervezünk, mert így könnyen lehet módosítani, illetve könnyen átalakítható úgy, hogy más projektben is használhassuk. Emiatt külön csomagban írtuk meg az adatbevitelt, mégpedig úgy, hogy előbb egy interfészben végiggondoltuk, hogy mire is van szükségünk. Arra, hogy „kezünkben" legyen a fájlban lévő adatokból készült itallista. Ezt kétféle módon oldhatjuk meg:

1. az előzőekben tanultak szerint, ekkor a metódusnak egy `List<Ital>` típusú gyűjteményt kell visszaadnia;
2. már eleve figyelembe vesszük, hogy egy grafikus `JList` típusú felületre akarjuk kiírni. Ebben az esetben nem `List<Ital>` típusú lesz az eredmény, hanem `DefaultListModel<Ital>` típusú.

```
public interface AdatInput {
    public List<Ital> italLista() throws Exception;
    public DefaultListModel<Ital> italModell() throws Exception;
}
```

(Az interfészt ennél általánosabban is megírhatnánk, de evvel most nem foglalkozunk.)

Az első változatnak az az előnye, hogy a `List<>` típus általánosabb, több helyen tudjuk felhasználni az így kapott kollekciót. A második viszont célirányosabb.
A legjobb megoldás az lenne, ha az interfész mindkét metódust tartalmazná, és két implementáló osztályt írnánk:

```
public class ListaBevitel implements AdatInput{

    @Override
    public List<Ital> italLista() throws Exception {
        List<Ital> italok = new ArrayList<>();
        // megírjuk a metódust
        return italok;
    }

    @Override
    public DefaultListModel<Ital> italModell() throws Exception {
        // üresen hagyjuk
        return null;
    }
}
```

A `public class ModellBevitel implements AdatInput{` osztályban pedig pont fordítva írjuk meg a metódusokat, azaz a modellre vonatkozót írjuk meg és a listára vonatkozót hagyjuk üresen. És természetesen lehet olyan osztály is, amelyben mindkét metódust megírjuk.

Most a modellt kezelő osztályt használjuk majd. Mivel ezt már mindenkinek kellene érteni, ezért nem részleteztem túlzottan. Egyetlen megjegyzés: `DefaultListModel<>` típusú objektumhoz az `addElement()` metódussal tudunk újabb elemet hozzáadni.

```
public class FajlbolModellInput implements AdatInput{
    private String adatEleres;
    private final int SIMA_HOSSZ = 3;

    public FajlbolModellInput(String adatEleres) {
        this.adatEleres = adatEleres;
    }

    @Override
    public List<Ital> italLista() throws Exception {
        throw new UnsupportedOperationException("Not supported yet.");
    }

    @Override
    public DefaultListModel<Ital> italModell() throws Exception {
        DefaultListModel<Ital> italModell = new DefaultListModel<>();

        try (InputStream ins = this.getClass().getResourceAsStream(adatEleres)
                Scanner fajlScanner = new Scanner(ins, Global.CHAR_SET)) {
            String sor;
            String[] adatok;
            Ital ital;
            // narancslé;123456;300
            // sör;555555;600;Java;5
            while (fajlScanner.hasNextLine()) {
                sor = fajlScanner.nextLine();
                if (!sor.isEmpty()) {
                    adatok = sor.split(";");
                    if (adatok.length == SIMA_HOSSZ) {
                        ital = new Ital(adatok[0], adatok[1],
                                Integer.parseInt(adatok[2]));
                    } else {
                        ital = new AlkoholosItal(adatok[0], adatok[1],
                                Integer.parseInt(adatok[2]),
                                adatok[3], Double.parseDouble(adatok[4]));
                    }
                    italModell.addElement(ital);
                }
            }
        }
        return italModell;
    }
}
```

Megjegyzések:
1. A sort nem csak a `split()` metódussal lehetne szétvágni.
Az is jó megoldás lenne, ha a `sor`-ra is definiálunk egy `Scanner`-t, (`Scanner sorScanner`) azzal pedig addig megyünk, ameddig van újabb elem (vagyis `hasNext()`). Ehhez persze meg

kell mondani, hogy meddig tart egy elem. Ezt a `Scanner.useDelimiter(";")` módon adhatjuk meg.

2. A `Scanner` osztály példányosításakor az inputcsatorna mellett a kódolás módját is illik megadni, hiszen egyébként nem tudja kezelni az ékezetes karaktereket. Órán a kódolási módot meghatározó `String`-et egy külön osztályban (`Global`) adtuk meg, bár az sem lett volna baj, ha ennek az osztálynak az elején állítottuk volna be: A `Global` jellegű osztállyal azért érdemes megismerkedni, mert célszerű a program által használt konstansokat, sőt, esetleg bizonyos speciális metódusokat is egy külön osztályban definiálni. Egyrészt azért, hogy a projekt bármelyik osztálya könnyen el tudja érni, másrész a későbbi könnyű módosíthatóság miatt. Ebben a projektben a `Global` osztályba került a kódolás beállítása és az adatfájl elérési útjának megadása:

```
public static final String CHAR_SET = "UTF-8";
public static final String ADAT_ELERES = "/adatok/arlista.txt";
```

3. Bár valami hasonló lenne a szép megoldás, de ahogy gyakorlaton is mondtam: ha valakinek nincs rá elég ideje, akkor egy iskolai szintű feladatot úgy is megoldhat, hogy a beolvasási metódust abban az osztályban írja meg, ahol fel akarja használni az adatokat, vagyis ott, ahol meghívjuk a metódust.

Ha megírtuk az adatbevitelre vonatkozó metódust, akkor az a további kérdés, hogy hol hívjuk meg ezt a metódust, illetve hogy hova rakjuk az adatok alapján létrehozott példányokat.

Erre is több lehetőség van:

1. Az `ItalFrame` osztályban.
2. Az `ItalPanel` osztályban.

Órán az első változatot vettük, most mindkettőt megmutatom. Most elsősorban azért vettük az elsőt, hogy az osztályok közötti „kommunikációról” is szó legyen, de azért is, mert ez áll közelebb a korábbi konzolos alkalmazáshoz. Egyébként pedig akkor érdemes ezt a megoldást választani, ha több osztály is használni akarja a beolvasás során létrejött modellt (vagy általánosabb esetben a beolvasás során létrejött listát).

A második változat inkább akkor célszerű, ha a beolvasott adatokra csak ennek az osztálynak van szüksége. (Esetünkben ez is elég lett volna.)

Nézzük tehát a két változatot:

1. A frame-n hívjuk meg az olvasást.
Nyilván a `main()` metódus indítja az egész programot. Ez generált `JFrame` esetén bekerül a `JFrame` osztályba, de természetesen lehetne külön `Main` osztályt is írni, amelyet ugyanúgy használhatnánk, mint eddig.
Most a `JFrame`-ben generált `main()` metódust mutatom. Ennek belsejében van egy másik metódus (`run()`) amivel egyelőre nem foglalkozunk. Akár ki is törölhető, de persze, óvatosan, egyébként pedig kicsit segíti a hatékonyságot. Ebben a metódusban ugyanúgy dolgozunk, mint eddig. Eddig azt csináltuk, hogy példányosítottuk a vezérlő osztályt, és meghívtuk annak indító metódusát. Most is ugyanezt tesszük. A vezérlő osztály jelenleg maga az `ItalFrame` osztály, ezért őt kell példányosítanunk, és ennek a `start()` metódusát kell

meghívnunk. Arra kell figyelni, hogy a láthatóságra vonatkozó metódust is meg kell hívni, egyébként ugyanolyan, mint a konzolos alkalmazás:

```
public void run() {
    new InditoFrame().start();
}
```

```
private void start() {
    this.setVisible(true);
    try {
        kocsmaPanel1.adatBevitel();
    } catch (Exception ex) {
        ex.printStackTrace();
        JOptionPane.showMessageDialog(this, "Hibás fájl");
        Logger.getLogger(InditoFrame.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Most összesen csak annyit tettünk, hogy megkértük a felületen lévő kocsmapanel példányt, hogy olvassa be az adatokat. Az `ex.printStackTrace()` most nem kell, csak azért van a kódban, hogy jelezzem: ha nem generált a `catch()` blokk, akkor ezt érdemes beleírni, mert ez ugyanúgy kilistázza a dobott kivételeket, mint a generált logger.

A `KocsmaPanel` osztály megfelelő metódusa:

```
void adatBevitel() throws Exception {
    AdatInput adatInput = new FajlbolModellInput(Global.ADAT_ELERES);
    italModell = adatInput.italModell();
    lstItallap.setModel(italModell);
}
```

1.b/ Természetesen előfordulhat, hogy több panelen is szeretnénk használni ugyanazokat az adatokat. Ekkor is a frame-n lenne célszerű beolvasni őket, de akkor inkább így:

```
private void indit() {
    try {
        AdatBevitel adatBe = new ModellBevitel(this.path);
        DefaultListModel<Ital> italModell = adatBe.italModellBevitel();
        italPanel1.listabaIr(italModell);
    } catch (Exception ex) {
        Logger.getLogger(ItalFrame.class.getName()).log(Level.SEVERE, null, ex);
        JOptionPane.showMessageDialog(this, "Hibás adatbevitel");
        System.exit(0);
    }
}
```

A szereplő kódrészletben nem az órán használt nevek szerepelnek, talán ez is jelzi, hogy most nem így csináltuk.

A beolvasás után a frame már ismeri az italokat, de a panel felületén kell megjelenítenünk őket. Ezért „meg kell kérnie” a panelt, hogy rakja ki őket a felületére, azaz szól az

`italPanel1` példánynak, hogy írj listába az `italModell`-t. (Az `italPanel1` megnevezés az `ItalPanel` példány frame-re való húzásakor keletkezett. (Olyasmi ez, hogy a kocsma főnöke már tudja, hogy milyen italokat akarnak árulni, és megkéri a beosztottját, hogy írja meg az itallapot.)
Ettől kezdve a „főnöknek” (frame) semmi dolga, mindent elintéz a „beosztott” (`italPanel1` példány).

2. A másik lehetőség, hogy a főnök „nyaral”, és mindent a beosztottra bíz, vagyis mindent a `KocsmaPanel` osztály végez, még az adatbevitel meghívását is. De ha ebben az osztályban hívjuk meg a beolvasást, akkor pontosan hol? A konstruktorban nem lehet, mert akkor hibás adatbevitel esetén borul az egész, még maga a panel példány sem jön létre. Ha az itteni olvasást választjuk, akkor az olvasó metódus hívását egy esemény bekövetkeztéhez kell fűznünk. Ez az esemény a form betöltése. Itt ezt nem formLoad-nak hívják, hanem `ancestorAdded` eseménynek. (Talán előzményhez adásnak lehetne fordítani – és valóban nem csak formLoad-ról van szó, hanem hozzárendelhetjük pl. egy gombhoz is, és akkor a gomb megjelenésekor végrehajtandó dolgokat írhatjuk ide.) Az eseményhez tartozó metódus-fejet (mint minden esemény esetén) az objektumra kattintva lehet generálni. (Jobb egérgomb, Events)

Ekkor itt hívjuk meg az `indit()` metódust, és magát a metódust is ebben az osztályban adjuk meg.

```
private void formAncestorAdded(javax.swing.event.AncestorEvent evt) {
    indit();
}
```

```
private void indit() {
    try {
        AdatBevitel adatBe = new ModellBevitel(this.path);
        italModell = adatBe.italModellBevitel();
        italModell = adatInput.italModell();
        lstItallap.setModel(italModell);
    } catch (Exception ex) {
        Logger.getLogger(ItalFrame.class.getName()).log(Level.SEVERE, null, ex);
        JOptionPane.showMessageDialog(this, "Hibás adatbevitel");
        System.exit(0);
    }
}
```

A panel osztály további viselkedése mindkét előző megoldás esetén azonos: meg kell írni az árlistát, azaz ki kell rakni az italok adatait a `JList` típusú felületre. A `JList` csak egy felületet biztosít arra vonatkozóan, hogy hol jelenjenek meg az ital-adatok. (Ugyanezek az adatok megjelenhetnének pl. a konzolon is.) Az, hogy mi jelenjen meg a felületen, a `JList`-hez tartozó modellen múlik. A modell tartalmazza a kiírandó adatokat. Ha menet közben másik modellt rendelünk a `JList` példányhoz, akkor más adatokat fog kiírni. A modell tárolja tehát az adatokat, sőt, valamennyire manipulálni is tudja azokat. Ha a modell változik, akkor a felületen is azonnal megjelenik a változás eredménye.

Összesen annyi a dolgunk, hogy közöljük a listafelülettel, hogy melyik modell tartozik hozzá. Ezt a `setModell()` metódussal tudjuk beállítani. Mivel a modell pontosan abban a lényegi dologban különbözik a listától, hogy minden változásról tudja értesíteni a listafelületet, ezért a listafelületet és a modellt **csak egyszer** kell egymáshoz rendelni.

Ennyi, és máris látjuk az árlistát.

Megjegyzés:
A `JList` felületén a benne lévő objektumok `String` formája jelenik meg, vagyis az, amit az objektumhoz tartozó `toString()` metódusban megírtunk.

b/ Alkoholfok kiíratása

A listára kattintva jelenjen meg egy felirat. Ehhez egy label-t kell felraknunk a felületre. Ennek neve legyen mondjuk `lblAlkoholFok`. A listára kattintáskor ez történik (a metódus generálódik az esemény kiválasztásakor, csak a törzset kell megírni):

```
private void lstItallapValueChanged(javax.swing.event.ListSelectionEvent evt) {
    Ital ital = (Ital) lstItallap.getSelectedValue();
    if(ital instanceof AlkoholosItal){
        lblAlkoholFok.setText("Alkoholfok: " +
                              ((AlkoholosItal)ital).getAlkoholFok() + " %");
    }else{
        lblAlkoholFok.setText("");
    }
}
```

(A `getSelectedValue()` metódus egy `Object`-et ad vissza, itt ezért kell a típuskényszerítés, az ital pedig Ital típusú, vagyis kényszerítés nélkül nincs `getAlkoholFok()` metódusa. Ugyanakkor figyelni kell rá, hogy nem tudunk vízből pálinkát varázsolni, azaz valóban csak az alkoholos ital fajtára vonatkozzon a típuskényszerítés.)

c/ Rendezések

Majd tanulnak elegánsabb megoldást is, most azt tehetjük, hogy kiszedjük az adatokat a modellből, rendezzük őket, majd visszarakjuk. Erre azért van szükség, mert a `DefaultListModell` típushoz nem tartozik közvetlen rendezési metódus. (Az lesz majd az elegánsabb megoldás, hogy nem a default modellt használjuk, hanem írunk egy sajátot.)

A modellből nyilván egy klasszikus `for` ciklussal is át lehet pakolni az elemeket egy listába (default modellt nem tudunk foreach ciklussal kezelni), de egyetlen utasítással is megoldható a feladat: `List<Ital> italok = Collections.list(italModell.elements());`
és erre a listára már alkalmazható a `Collections` osztály `sort()` metódusa.

Kitérő: az is lehet, hogy a default modellből a `toArray()` metódushívással egy tömböt hozunk létre. A kapott tömb ugyanúgy rendezhető, mint egy `ArrayList`, annyi különbséggel, hogy most nem a `Collections.sort()`, hanem az `Arrays.sort()` metódust használjuk. Ennek a metódusnak több paraméterezési lehetősége van, de szerintem az

használható a legegyszerűbben, ha az egyparaméteres változatot választjuk. Ehhez azonban `Comparable` típusúvá kell alakítanunk az `Ital` osztályt, és a feladatkiírásnak megfelelő módon kell megírnunk a szükséges `CompareTo()` metódust.

Az `Arrays.sort()` metódusnak is van kétparaméteres változata (a második paraméter egy `Comparator` típusú példány), de ez itt sajnos csak némi nehézség árán alkalmazható, ugyanis a `toArray()` metódussal kigyűjtött tömb `Object` típusú elemeket tartalmaz, és helyenként típuskényszerítést kellene alkalmaznunk. (kitérő vége)

Másik lehetőség, hogy eleve a listába (`List<>`) való beolvasást választja, ezt a listát is átadja a panelnek (ill. csak ezt adja át – most ugyanis a frame csak a listát fogja látni), és majd az szükség szerint időnként átrakja a modellbe a lista adatait. (Mondjuk a lista rendezése után, de persze, rögtön az adatbevitel után is.) Ennek az a hátránya, hogy a program teljes életciklusa alatt két helyen is tároljuk az adatokat, míg az első változat esetében csak a rendezés lefutásáig.

Mindegyik megoldásban van némi gányolás, de amíg nem beszéltük meg, hogy hogyan lehet saját modellt írni, addig kénytelenek vagyunk kicsit gányolni.

Közlök majd egy-egy változatot a Comparable és a Comparator típusú megoldásra is, de előbb még néhány szó a rádiógombok kezeléséről.

A rádiógombok hatására tudjuk beállítani a rendezési szempontokat.
A rádiógombok felrakásakor arra kell figyelni, hogy külön csoporba (`ButtonGroup`) tegyük a rendezési szempont kiválasztására vonatkozó gombokat, és egy másikba az irány meghatározására vonatkozókat. Csoportot a palettáról tudunk ráhúzni az alkalmazásra, legegyszerűbb, ha húzáskor elengedjük a form felülete mellett.
Azt, hogy melyik rádiógomb melyik csoporthoz tartozzon, a rádiógomb tulajdonságai között tudjuk beállítani. (Properties\buttonGroup). Ugyancsak itt állítható be, hogy átlátszó legyen-e a háttér vagy sem (opaque tulajdonság).

Az egyes gombokhoz rendelt események:

```
private void rdbNevsorActionPerformed(java.awt.event.ActionEvent evt) {
    nevsorba();
}

private void rdbNovekvoActionPerformed(java.awt.event.ActionEvent evt) {
    novekvoCsokkeno();
}

private void rdbCsokkenoActionPerformed(java.awt.event.ActionEvent evt) {
    novekvoCsokkeno();
}

private void rdbArActionPerformed(java.awt.event.ActionEvent evt) {
    arSzerint();
}

private void rdbAlkoholFokActionPerformed(java.awt.event.ActionEvent evt) {
    alkoholfokSzerint();
}
```

A növekvő/csökkenő beállítás nagyon egyszerű:

```
private void novekvoCsokkeno() {
    if(rdbNevsor.isSelected())nevsorba();
    if(rdbAr.isSelected()) arSzerint();
    if(rdbAlkoholFok.isSelected())alkoholfokSzerint();
}
```

Vagyis most azt figyeljük, hogy a növekvő/csökkenő rendezést jelző gombok megnyomásakor melyik szempont van bejelölve, és az annak megfelelő metódust hívjuk.

A többi is egyszerű, csak ott már függ a megoldás attól, hogy melyik fajta összehasonlítási módot választotta. Ha az `Ital` osztályt tette `Comparable` típusúvá, akkor pl. ilyesmi lehet a megfelelő metódus:

```
private void nevsorba() {
    Ital.setRendezesiSzempont(Ital.Szempont.FAJTA, rdbNovekvo.isSelected());
    rendez();
}
```

Ha pedig külön `Rendezes` osztályt ír, akkor valami ilyesmi:

```
private void nevsorba() {
    Rendezes.setRendezesiSzempont(Rendezes.Szempont.FAJTA,
                                  rdbNovekvo.isSelected());
    rendez();
}
```

A `rendez()` metódus:

```
private void rendez() {
    // átrakjuk a modell elemeit egy listába
    List<Ital> italok = Collections.list(italModell.elements());

    // Ehhez a megoldáshoz Comparable típusúvá kell tenni az Ital osztályt
    Collections.sort(italok);

    // Ehhez külön meg kell írni a Rendezes osztályt
    Collections.sort(italok, new Rendezes());

    // visszarakjuk a modellbe a rendezett adatokat
    italModell.clear();
    for (Ital ital : italok) {
        italModell.addElement(ital);
    }
}
```

Természetesen a kétféle rendezés (`sort()`) közül egyszerre csak az egyiket kell használni.

Jöjjön tehát a rendezés ígért kétféle előkészítési módja. Előbb a külön rendezési osztályt mutatom be. Ezt `enum` típus használatával oldottam meg (részben azért, hogy arra is lásson egy újabb példát, de főleg azért, mert ha csak néhány értéket vehet fel egy változó, akkor azt lehetőleg `enum`-ként kell kezelni). Különösebb magyarázat nélkül mutatom meg.

```
public class Rendezes implements Comparator<Ital>{

    // Megadjuk, hogy milyen szempont alapján rendezünk.
    public enum Szempont {FAJTA, AR, ALKOHOLFOK}

    // Milyen módon.
    public final static boolean NOVEKVOEN = true;
    public final static boolean CSOKKENOEN = false;

    private static Szempont valasztottSzempont;
    private static boolean miModon;

    public Rendezes() {
    }

    @Override
    public int compare(Ital t1, Ital t2) {
        switch(valasztottSzempont){
            case FAJTA: {
                return(miModon)?t1.getFajta().compareTo(t2.getFajta()) :
                        t2.getFajta().compareTo(t1.getFajta());
            }
            case AR: {
                return(miModon)?t1.getLiterAr()-t2.getLiterAr() :
                        t2.getLiterAr() - t1.getLiterAr();
            }
            case ALKOHOLFOK: {
                double t1fok, t2fok;
                t1fok = (t1 instanceof AlkoholosItal) ?
                                ((AlkoholosItal)t1).getAlkoholFok() : 0;
                t2fok = (t2 instanceof AlkoholosItal) ?
                                ((AlkoholosItal)t2).getAlkoholFok() : 0;
                return (int) ((miModon) ? Math.signum(t1fok - t2fok) :
                                          Math.signum(t2fok - t1fok));
            }
        }
        return 0;
    }
```

Mivel a `valasztottSzempont` és a `miModon` változók értékét mindig együtt kezeljük, ezért célszerű egyetlen setterben megadni őket.

```
public static void setRendezesiMod(Szempont valasztottSzempont, boolean miModon) {
    Rendezes.valasztottSzempont = valasztottSzempont;
    Rendezes.miModon = miModon;
}

public static Szempont getValasztottSzempont() {
    return valasztottSzempont;
}

public static boolean isMiModon() {
    return miModon;
}
```

A másik változat az `Ital` osztályt készíti fel. Ekkor az osztálynak implementálnia kell a `Comparable` interfészt:

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

és definiálni kell benne a `compareTo()` metódust. (A többi ugyanolyan, mint a `Rendezes` osztályban.)

A `compareTo()` metódus lényegileg ugyanolyan, mint egy `Comparator` típusú osztály (`Rendezes`) `compare()` metódusa, egyetlen különbséggel: a `compareTo()` az aktuális példányt hasonlítja össze a paraméterében lévő példánnyal, a `compare()` metódusban pedig mindkét összehasonlítandó példány a metódus paraméterében szerepel.

A metódus:

```
@Override
public int compareTo(Ital t) {
    //Csak azért vezettem be ezt a két változót, hogy lássa,
    // lényegileg ugyanarról a metódusról van szó, mint a külön osztályban.
    Ital t1 = this, t2 = t;
   switch(valasztottSzempont){
        case FAJTA: {
            return(miModon)?t1.getFajta().compareTo(t2.getFajta()) :
                    t2.getFajta().compareTo(t1.getFajta());
        }
        case AR: {
            return(miModon)?t1.getLiterAr()-t2.getLiterAr() :
                    t2.getLiterAr() - t1.getLiterAr();
        }
        case ALKOHOLFOK: {
            double t1fok, t2fok;
            t1fok = (t1 instanceof AlkoholosItal) ?
                            ((AlkoholosItal)t1).getAlkoholFok() : 0;
            t2fok = (t2 instanceof AlkoholosItal) ?
                             ((AlkoholosItal)t2).getAlkoholFok() : 0;
            return (int) ((miModon) ? Math.signum(t1fok - t2fok) :
                                      Math.signum(t2fok - t1fok));
        }
    }
    return 0;
}
```

Ezek után eldöntheti, hogy melyik megoldás tetszik jobban. ☺

d/ Rendelés

Ez elég egyszerű, nem is igényel sok magyarázatot.

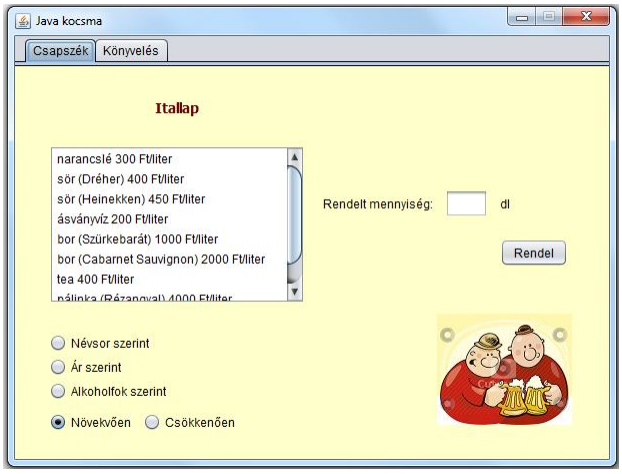
1. Az Ital osztályt ki kellene egészíteni egy `rendel()` metódussal, plusz az összmennyiséghez és összbevételhez tartozó getterekkel:

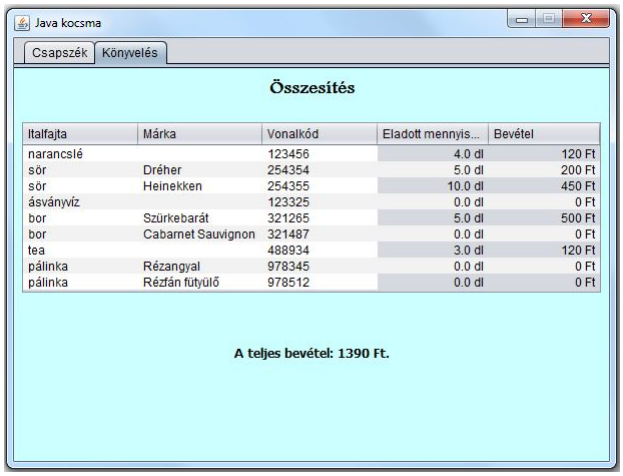
```
public void rendel(double deciMennyiseg) {
    this.deciMennyiseg = deciMennyiseg;
    osszMennyiseg += deciMennyiseg;
    osszBevetel += fizetendoAr();
}
```

A `KocsmaPanel` osztály gombnyomás hatására lefutó metódusa:

```
private void btnRendelActionPerformed(java.awt.event.ActionEvent evt) {
    Ital ital = (Ital) lstArLista.getSelectedValue();
    try {
        double mennyiseg = Double.valueOf(txtMennyiseg.getText());
        if(mennyiseg > 0){
            ital.rendel(mennyiseg);
            lblFizetendo.setText("Fizetendő: " + ital.fizetendoAr() + " Ft");
        }
    } catch (Exception e) {
        if (e instanceof NullPointerException) {
            JOptionPane.showMessageDialog(this, "Nem választott semmit");
        } else {
            JOptionPane.showMessageDialog(this, "Hibás a mennyiség.");
        }
    }
}
```

A feladat további része ezt kéri:



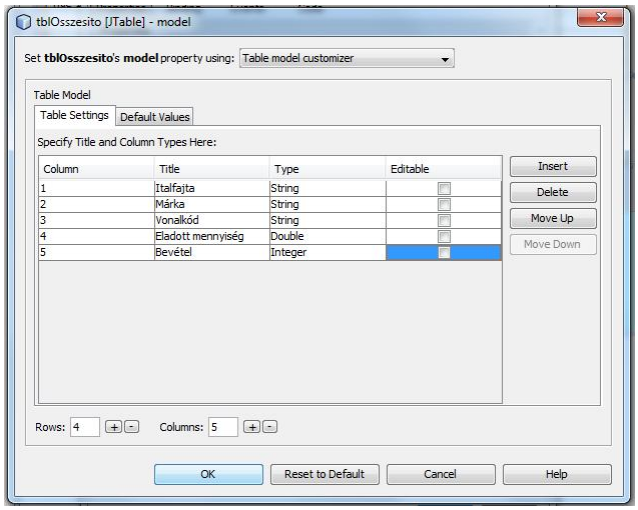


Vagyis hirtelen meg kellene változtatni az egészet, és többregiszteres alkalmazássá alakítani. Persze, mondhatná, hogy miért nem gondolkoztunk időben, de sajnos az „éles” alkalmazások esetén is elég gyakori az olyasmi, hogy már csak a csaknem késznek vélt megoldás bemutatása után jön rá a megrendelő, hogy ő még mást is szeretne, vagy nem pont így gondolta. Ezért fontos az, hogy a programunk minél rugalmasabb, és minél könnyebben módosítható legyen.
Szerencsére a mienk pont ilyen. Kb. fél perc alatt alkalmassá tehetjük az előző megoldást arra, hogy több regiszteres legyen.

Ehhez csak ennyi kell: szedjük le a frame felületéről a `kocsmaPanel1` példányt, rakjunk rá a frame-re egy `tabbedPane` objektumot, majd erre rakjuk vissza a `kocsmaPanel` példányt. Máris megvan az első fül. Ha erre a `tabbedPane` objektumra rárakunk egy másik panelt, akkor megvan a második fül.

Beszéljük meg a **másik panel**t is.

Erre két label-t teszünk fel (a felirat, plusz a bevétel kiíratására szolgáló label), és egy `JTable` példányt. Ez utóbbit is ki lehet választani a palettáról, de úgy is megcsinálhatja, ahogy az előző órán tanulták. Így remélem, önállóan is meg tudná csinálni. Most azt mutatom meg, hogy hogyan lehet bánni a palettáról választott `JTable` objektummal.
A palettáról kiválasztott tábla modell tulajdonságát be tudjuk állítani (Properties/model – gomb), pl. így:



Legyen a `JTable` neve `tblOsszesito`, a bevételt tartalmazó labelé `lblBevetel`.

A panel generált részek nélküli forrása:

```
private DefaultTableModel tablaModell ;
private int bevetel;

public OsszesitoPanel() {
    initComponents();
    // egy kis külcsín
    tblOsszesito.setShowGrid(true);

    // Mivel a palettáról leszedett tábla modelljét már létrehoztuk akkor,
    // amikor beállítottuk a tábla fejlécét, ezért most ezt a modellt
    // akarjuk használni.
    // Sajnos a getModel() metódus TableModel típust ad vissza, ezért
    // típuskényszerítés kell (ha nem akarunk saját modell osztályt írni).
    tablaModell = (DefaultTableModel) tblOsszesito.getModel();
}

public void italokTablazatba(Object[] italok) {

    // kitöröljük a tábla korábbi sorait
    int n = this.tablaModell.getRowCount();
    for (int i = n - 1; i >= 0; i--) {
        this.tablaModell.removeRow(i);
    }

    Ital ital;
    String marka;
    // létrehozzuk az új sorokat
    for (Object obj : italok) {
        ital = (Ital) obj;
        marka = (ital instanceof AlkoholosItal) ?
                                    ((AlkoholosItal) ital).getMarka() : "";
        Object[] tablaSor = {ital.getFajta(), marka, ital.getVonalKod(),
            ital.getOsszMennyiseg(), ital.getOsszBevetel()};
        this.tablaModell.addRow(tablaSor);

        // kiszámoljuk a bevételt
        bevetel += ital.getOsszBevetel();
    }
    lblBevetel.setText("A teljes bevétel: " + bevetel + " Ft.");
}
```

A `DefaultTableModel` osztály `addRow()` metódusa egy `Object[]` tömböt vár paraméterként, ilyet a listamodellből is nagyon könnyen tudunk készíteni, ezért célszerű ezt átadni.

Már csak az a kérdés maradt hátra, hogy hol hívjuk meg ezt az `italokTablazatba()` metódust.

Nyilván ott, ahol ismerjük az italok adatait is, de elérjük ezt a metódust is.

Az ital objektumokat most a `kocsmaPanel` példány listamodellje tartalmazza. Az `italokTablazatba()` metódus pedig egy `osszesitoPanel` objektumon keresztül érhető el. A két panel a frame-n szerepel együtt. No meg a `jTabbedPane` is a frame-re került. Vagyis a regiszter-fülre való kattintáskor a `kocsmaPanel` ital objektumait át kellene adni az `osszesitoPanel` metódusának. Ezt az `InditoFrame` osztályban tehetjük meg.

```
private void jTabbedPane1MouseClicked(java.awt.event.MouseEvent evt) {
    osszesitoPanel1.italokTablazatba(kocsmaPanel1.getItalok());
}
```

A metódusban látható példányneveket (`jTabbedPane1`, `osszesitoPanel1`, `kocsmaPanel1`) a NetBeans generálta akkor, amikor ráhúztuk az objektumokat a frame felszínére. Amikor megírjuk a metódust, figyelnünk kell ezekre a generált nevekre. (De át is nevezheti őket, ha úgy szimpatikusabb.)

Még nem mutattam meg az `KocsmaPanel` osztályban definiált `getItalok()` metódust. Ennyi:

```
public Object[] getItalok(){
    return italModell.toArray();
}
```

Megjegyzések:

1. Vegye észre, hogy itt is másolatot adunk át, hiszen a `toArray()` metódus hatására az `italModel` elemei átkerülnek egy `Object` típusú tömbbe.

2. Bár a módosíthatóság szempontjából elegáns megoldás az, hogy minden fülre külön saját panel osztályt írunk, nem biztos, hogy mindig élni kell vele. A nagyon összetartozó fülek esetén nem feltétlenül muszáj mindent külön panelre tenni, bár kétségtelen, hogy úgy elegánsabb.

A rövidséget ígérő cím ellenére nem lett túl rövid, de talán elég sok mindenről nyerhetett képet, ill. sok mindenhez kaphatott ötletet, ha végigolvasta.