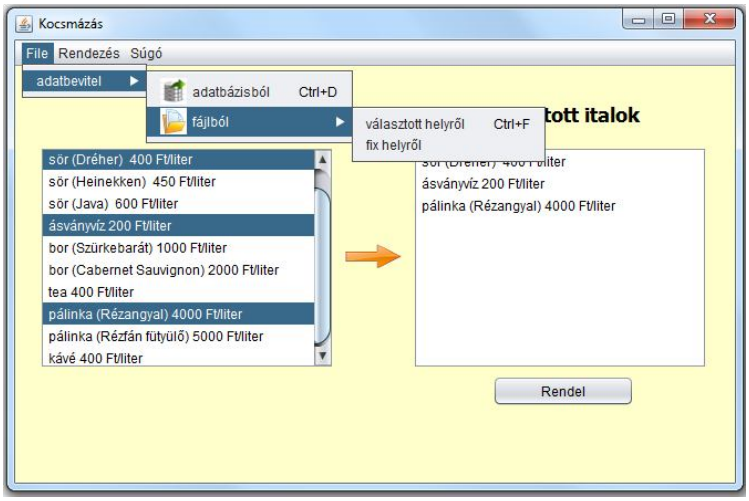
Interfész „haszna” – 4. gyakorlat második része

Feladat:

Kocsmázás ürügyén sok mindent szerettem volna megbeszélni, egyelőre csak egy részét sikerült, most csak ezt beszéljük meg.



A 650*400-as belső felületű alkalmazásban választhassuk ki, hogy honnan olvassuk be az adatokat.

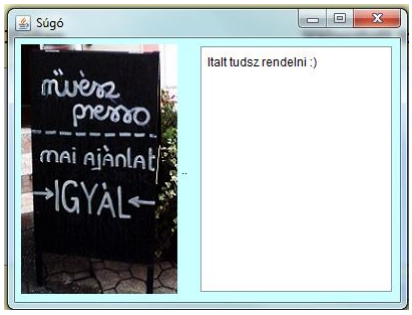
Ahogy látható, az itallapon szereplő italok között vannak alkoholos és nem alkoholos italok is.
Az ital definiálásakor meg kell adnunk a fajtáját (bor, tea, víz, stb), vonalkódját és literenkénti árát.

Az alkoholos italt a fentieken kívül még egy márkanév és az alkoholfok is jellemzi.

Ebből is csak azt a kettőt beszéljük most meg, amelyet vettünk, vagyis a fixen adott helyen lévő fájlból való olvasást és az adatbázisból valót.

És bár nem mindegyik csoportban került rá sor, de most megbeszéljük a súgót is.

A Súgó menüpont hatása:



A szóban forgó feladatrészlet (többféle olvasás) elsődleges célja az interfész használatának bemutatása, de nyilván az sem elhanyagolható újdonság, hogy megbeszéljük, hogyan lehet elérni egy adatbázist.

Nyilván a menüpontok használata is újdonság volt, de ezt önállóan is kitapasztalhatja.

Kezdjük a súgóval. Ebben pusztán annyi az érdekes, hogy két frame szerepel az alkalmazásban. Mindkettőben generálódik egy-egy `main()` metódus, de nyilván a súgó framet nem akarjuk önállóan indítani, ezért ebből kitöröljük a `main()` metódust. Nem ez teszi majd láthatóvá, hanem a súgó menüpont megnyomása. Itt azt is beállíthatjuk, hogy ez a frame az indító frame közepén jelenjen meg.

Az `InditoFrame` osztályban tehát:

```
private SugoFrame sugofrm = new SugoFrame();

private void sugoMenuPontActionPerformed(java.awt.event.ActionEvent evt) {
    sugofrm.setLocationRelativeTo(this);
    sugofrm.setVisible(true);
}
```

A `SugoForm` osztályban egyetlen dologra kell még odafigyelni, mégpedig arra, hogy a bezáró gomb hatására ne záródjon be az alkalmazás, hanem csak „tűnjön” el ez a frame.
Ezt a konstruktorban állíthatjuk be, mégpedig így:

```
this.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
```

Nézzük a beolvasásokat.

A valóságban ritkán fordul elő, hogy egy projektben választani lehet, hogy fájlból vagy adatbázisból olvassuk az adatokat, de az könnyen előfordulhat, hogy bár eredetileg pl. fájlból való olvasással oldották meg a feladatot, egy későbbi módosítás során átváltanak adatbázisra. És persze, az is lehet, hogy egy korábbi Derby adatbázist később pl. MySQL-re váltanak.

Ezért célszerű úgy megírni a beolvasást, hogy minél könnyebb legyen módosítani.

Nagyon könnyű lesz a módosítás, ha előbb egy interfészt írunk az olvasáshoz, majd ezt különböző osztályokkal implementáljuk. Ekkor a használat során csupán egyetlen dolgot kell változtatni, mégpedig azt, hogy az interfész típusra deklarált példányt melyik osztállyal példányosítjuk.

Az interfész törzs nélküli metódusok gyűjteménye (lehetnek benne konstans értékek is, de ez most lényegtelen). Megírásakor azt kell végiggondolni, hogy egy későbbi olvasás során milyen metódusokra lehet szükségünk. (Általában: egy későbbi feladatmegoldás során milyen metódusokra lehet szükség.)

Most úgy oldottuk meg a feladatot, hogy az olvasás használható legyen olyankor is, ha listába akarjuk rakni az adatokat (pl. konzolos alkalmazás, vagy bármi más), és akkor is, ha listamodellbe. Mivel a beolvasás mindig rizikós dolog, ezért kötelező kivételkezelést is előírunk.

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

Mint látható, most csak annyit írtunk elő, hogy az az osztály, amelyik implementálja ezt az interfészt „szerezzen” valahonnan egy itallistát, illetve egy italmodellt.
Implementáláskor minden metódust kötelező megírni, de ha valamelyikre nincs szükségünk, akkor akár üresen is maradhat. (Pontosabban: benne hagyható az a kivételdobás, amelyik jelzi, hogy még nincs megírva, vagy lehet, hogy egy null értéket adjon vissza.) De az implementálás

lehet az is, hogy a metódus egy konstans listát/modellt ad vissza. Ezek persze inkább csak elvi lehetőségek, ténylegesen általában nem ilyenekre van szükségünk.

Eddig két implementáló osztályt hoztunk létre: az egyik az adott helyen lévő fájlból való olvasást biztosítja, a másik egy adatbázisból enged olvasni.

Mivel gyakoribb, hogy listába kell tennünk a beolvasáskor létrehozott objektumokat, ezért ezt a metódust írtuk meg részletesen, a modellt létrehozó metódusban csak hivatkozunk rá.
Az adatfájl elérését nem égetjük be, hiszen ténylegesen a vezérlés (esetünkben most a panel) tudja azt, hogy pontosan hol is van a fájl, vagyis majd a konstruktor paramétereként adjuk át.

```
public class FajlbolInput implements AdatInput {

    private String italEleres;
    private final String CHAR_SET = "UTF-8";

    public FajlbolInput(String italEleres) {
        this.italEleres = italEleres;
    }

    @Override
    public List<Ital> italLista() throws Exception {
        // lehet úgy is, hogy üresen - vagy a figyelmeztető üzenettel hagyjuk
        // valamelyik metódust
        List<Ital> italok = new ArrayList<>();
        // mivel az inputstream is és a scanner is lezárandó objektum, ezért
        // érdemes a try fejében megnyitni, ekkor automatikus a lezárás
        try (
                InputStream ins = this.getClass().getResourceAsStream(italEleres);
                Scanner fajlScanner = new Scanner(ins, CHAR_SET)){

            String sor;
            String[] adatok;
            Ital ital = null;
            while (fajlScanner.hasNextLine()) {
                sor = fajlScanner.nextLine();
                adatok = sor.split(";");
                if (!sor.isEmpty()) {
                    adatok = sor.split(";");
                    if (adatok.length == 3) {
                        ital = new Ital(adatok[0], adatok[1],
                                Integer.parseInt(adatok[2]));
                    }
                    if (adatok.length == 5) {
                        ital = new AlkoholosItal(adatok[0], adatok[1],
                                Integer.parseInt(adatok[2]),
                                adatok[3], Double.parseDouble(adatok[4]));
                    }
                }
                italok.add(ital);
            }
        }
        return italok;
    }
```

```
    @Override
    public DefaultListModel<Ital> italModell() throws Exception {
        // lehet úgy is, hogy üresen - vagy a figyelmeztető üzenettel hagyjuk
        // valamelyik metódust
        DefaultListModel<Ital> italModel = new DefaultListModel<>();
        List<Ital> templist = italLista();
        for (Ital ital : templist) {
            italModel.addElement(ital);
        }
        return  italModel;
    }
}
```

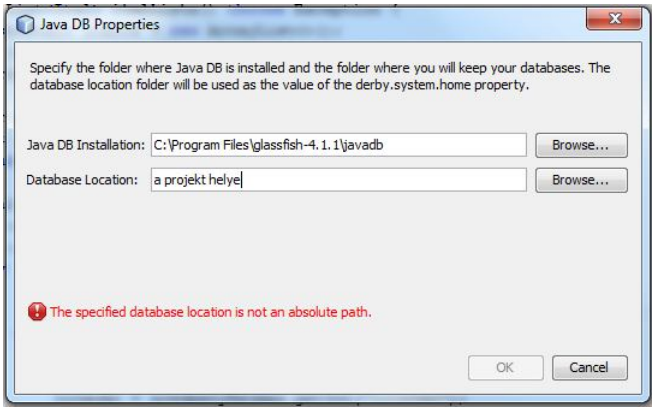
Megjegyzés: Mivel gyakorlaton másoltunk, ezért bekerült a `catch` ág is, amire nincs is szükség, hiszen továbbdobtuk a kivételt. (Ezt is ugyanúgy lehet `catch` ág nélkül, mint ahogy az adatbázis-kezelő osztályban írtuk.)

Egy adatbázis eléréséhez szükségünk van a vele való kapcsolatra. Ezt a Connection osztály egy példánya kezeli. Szó volt róla, hogy arra is figyeljünk, hogy egy későbbi módosítás során az adatbázis átkerül egy másik adatbázisszerverre, vagy esetleg másik adatbázis-kezelőt akarunk majd használni. Ezt megint csak a vezérlés tudja, vagyis a kapcsolatot ott kellene kiépíteni, és a konstruktor paramétereként átadni az inputot megvalósító osztály példányának.

Ha van kapcsolat, megkérjük, hogy hozzon létre egy `Statement` típusú utasításobjektumot, majd ezt az utasításobjektumot kérjük meg arra, hogy hajtsa végre a megadott sql utasítást – esetünkben egy lekérdezést (select * from …). Ennek eredménye egy `ResultSet` típusú eredményhalmaz, amely sorfolytonosan tartalmazza a lekért adatokat. Ezen végigiterálva lekérhetjük az egyes adatokat. (Mivel az adatbázis-kezelő nem pont ugyanolyan formátumban tárolja az adatot, mint amilyenben egy Java programnak kell, ezért azokat át kell konvertálni. Erre szolgálnak a `getString()`, `getInt()`, stb. metódusok.)

Természetesen ahhoz, hogy működjön a beolvasás, szükségünk van az adatbázisra. Ez lehet egy külső adatbázisszerveren, ekkor csak az elérhetőségét kell ismernünk, most azonban mi magunk hoztuk létre ezt az adatbázist. A hogyanját ld. a feladatkiírásban.

Ezt kiegészítendő még egy fontos megjegyzés: a Services \ JavaDb pontra jobb egérgombbal kattintva a Properties menüpont hatására felbukkan a mellékelt ablak:
Itt megadhatjuk, hogy hol van installálva a javadb driver. A legújabb Java verziókból ez hiányzik, ezért a legegyszerűbb, ha letölti a glassfish-4-et, és kicsomagolja valahova. Ezen belül megtalálja a szükséges installációt.



Az adatbázis helyeként pedig célszerű a projekt mappáját megadni, mert akkor mindig együtt lesz a kettő. Ha már korábban létrehozta az adatbázist, akkor természetesen nem kell ismét és

ismét létrehozni a megadott sql fájl alapján, hanem csak itt, a tulajdonság-ablakban kell beállítani az adatbázis helyét, és ha eredetileg jól hozta létre az adatbázist, akkor így meg is találja azt.

Az adatbázist kezelő osztály:

```
public class AdatBazisbolInput implements AdatInput{

    Connection kapcsolat;

    public AdatBazisbolInput(Connection kapcsolat) {
        this.kapcsolat = kapcsolat;
    }

    @Override
    public List<Ital> italLista() throws Exception {
        List<Ital> italok = new ArrayList<>();
        // ezt az sql utasítást kell majd végrehajtani
        String sqlUtasitas = "select * from ITALOK";

        String fajta, vonalkod, marka;
        int literAr;
        double alkoholFok;

        Ital ital;
        // az utasításobjektum és az eredményhalmaz is lezárandó, ezért
        // a try fejében hozzuk létre őket.
        try (Statement utasitasObjektum = kapcsolat.createStatement();
                ResultSet eredmenyHalmaz =
                                    utasitasObjektum.executeQuery(sqlUtasitas)) {
            while (eredmenyHalmaz.next()) {
                fajta = eredmenyHalmaz.getString("fajta");
                vonalkod = eredmenyHalmaz.getString("vonalkod");
                literAr = eredmenyHalmaz.getInt("literar");
                marka = eredmenyHalmaz.getString("marka");
                alkoholFok = eredmenyHalmaz.getDouble("alkoholfok");
                if(marka == null){
                    ital = new Ital(fajta, vonalkod, literAr);
                }
                else ital = new AlkoholosItal(fajta, vonalkod,
                                                literAr, marka, alkoholFok);

                italok.add(ital);
            }
        }
        return italok;
    }
```

Az osztály `ItalModell()` metódusa szó szerint ugyanaz, mint a másik fájl ilyen metódusa. (Felmerülhet a „vád”, hogy ez kódismétlés. Valóban az, és ha garantált, hogy a két osztályt ugyanabban a projektben használjuk, akkor a modellbe pakolást célszerű lenne mondjuk a panelen megírni, de ahogy korábban szó volt róla, ezeket az osztályokat általában nem

ugyanabban a projektben használjuk, illetve nem egyszerre, hanem egy esetleges módosítás során az egyiket kicseréljük a másikra.)

Megjegyzés: Az adattábla oszlopait az oszlop száma alapján is meg lehet adni, de vigyázzon rá, hogy itt a számozás 1-ről indul (hiszen első oszlop). Az adatbázis első oszlopa az egyedi azonosító volt, vagyis a fajta oszlopa a 2-es számú:

```
fajta = eredmenyHalmaz.getString(2);
vonalkod = eredmenyHalmaz.getString(3);
literAr = eredmenyHalmaz.getInt(4);
marka = eredmenyHalmaz.getString(5);
alkoholFok = eredmenyHalmaz.getDouble(6);
```

Ezek után lássuk az alkalmazásukat, és vegye észre, hogy valóban csak annyi az eltérés, hogy másképp példányosítjuk az `AdatInput` típusú változót. (Az eltérő konstruktor miatt nyilván az adatfájl elérését, illetve az adatbázis-kapcsolatot is létre kell hoznunk, de ezek apró eltérések.)

Órán csak szóban beszéltünk róla, itt meg is valósítom azt, hogy csak akkor váljon aktívvá a rendezés menüpont, ha sikeres a beolvasás.
A két menüpont hatása (`InditoFrame` osztály):

```
private void adatbazisbolMenupontActionPerformed(java.awt.event.ActionEvent evt) {
    if (kocsmaPanel1.adatBazisbol()) {
        rendezesMenu.setEnabled(true);
    }
}

private void fixHelyrolMenuPontActionPerformed(java.awt.event.ActionEvent evt) {
    if (kocsmaPanel1.fixFajlbol()) {
        rendezesMenu.setEnabled(true);
    }
}
```

A panel ide vonatkozó metódusai:

```
private final String ITAL_ELERES = "/adatok/arlista.txt";
private DefaultListModel<Ital> italModel;

public boolean fixFajlbol() {
    try {
        AdatInput adatInput = new FajlbolInput(ITAL_ELERES);
        adatBevitel(adatInput);
        return true;
    } catch (Exception ex) {
        Logger.getLogger(KocsmaPanel.class.getName()).log(Level.SEVERE,
                                                          null, ex);
        return false;
    }
}

public boolean adatBazisbol() {
    try (Connection kapcsolat = adatBazisKapcsolat()) {
        AdatInput adatInput = new AdatBazisbolInput(kapcsolat);
        adatBevitel(adatInput);
        return true;
    } catch (Exception ex) {
        Logger.getLogger(KocsmaPanel.class.getName()).log(Level.SEVERE,
                                                          null, ex);
        return false;
    }
}

private Connection adatBazisKapcsolat() throws ClassNotFoundException,
                                               SQLException {
    // az adatbázis driver meghatározása
    Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
    // az adatbázis definiálása
    String url = "jdbc:derby://localhost:1527/PROG3";
    // kapcsolodas az adatbázishoz
    return DriverManager.getConnection(url, "kocsma", "kocsma");
}

private void adatBevitel(AdatInput adatInput) throws Exception {
    italModel = adatInput.italModell();
    lstItallap.setModel(italModel);
}
```

(Ebben a megoldásban PROG3 volt az adatbázis neve ☺.)

Apró megjegyzés: gyakorlaton láttam, hogy néhány embernél semmi sem történt a menüpontra való kattintáskor. Ilyenkor célszerű ellenőrizni, hogy jó eseményhez került-e az input hívása. Legegyszerűbben így ellenőrizheti: tervezési módban kattintson kétszer a menüpontra. Ha ez a source oldalon új metódust generál, akkor biztos, hogy nem ehhez volt rendelve a beolvasás. Ha a megírthoz vezet, akkor persze tovább kell keresni a hiba okát.