Foci EB 1.

A foci EB-n életbe lépett egy korábbi szabályzat feleségek és barátnők számára. Ennek szellemében írjon egy EB-szimulációt.

A szimulációban házaspárok vesznek részt. A programban résztvevő minden egyes **ember**nek van *neve*, és mindegyikükre jellemző a *meccsnézés()*, de teljesen eltérő módon, az viszont közös, hogy ennek során a megnézett meccs (vagyis a metódus paramétere) bekerül a *megnézett meccsek* listájába.

A **férj**ek meccsnézése során a meccsek közben elfogyasztott *sörök száma* növekszik, mégpedig ha jó a meccs, akkor egy, az összes férjre egyformán jellemző *darabszám*mal, ha nem jó, akkor pedig ugyancsak egy egyformán jellemző, de *másik darabszám*mal.

A **feleség**ek esetében a metódus hatására a *szabadidejük mennyisége* növekszik a paraméterben adott meccs hosszával.

A **házaspár**okat egy férj és egy feleség alkotja, és *meccsnézés()*ük során mindkét fél „nézi” a meccset.

Egy **meccs** megadásához két csapat kell. Jellemző rá a *meccshossz()*, amely a minden meccsre egyforma értékű *játékidő* és a *ráadás* összege.

Végezetül egy **csapat** a *nev*ével jellemezhető. Jelenleg csak ennyi érdekes belőle, de nem kizárt, hogy valamikor később még bővíteni lehet majd.

Olvassa be a házaspárok és a csapatok adatait, de most úgy oldja meg a feladatot, hogy bármikor könnyen és gyorsan át lehessen váltani az adatfájlból való olvasásról adatbázisból való olvasásra, és viszont. A program indulásakor írassuk ki a házaspárok névsorát, illetve külön-külön a férjeket is és a feleségeket is. Már induláskor is legyen látható a sörök és szabadidők értéke (nulla).

Szimuláljuk az EB-t, mégpedig úgy, hogy írassuk ki az aktuális meccset – ez úgy áll elő, hogy véletlenszerűen kiválasztjuk a meccshez tartozó két csapatot (figyeljünk rá, hogy egy csapat ne játsszon saját magával, de persze, több meccs is lehet ugyanazon két csapat között). Állítsuk be a meccs ráadás-idejét – ez egy 0 és adott határ közötti véletlen érték, majd azt is, hogy jó-e a meccs – ennek esélye valahány százalék. Ezek után a házaspárok véletlenszerűen „eldöntik”, hogy nézik-e a meccset – valahány százalék az esélye annak, hogy igen. Végül írassuk ki a férjek, feleségek aktuális állapotát.

Ezt addig ismételjük, amíg van újabb meccs (egy kérdésre adott választól függ, hogy van-e vagy sincs).

Megoldásrészletek:

Az alapfeladat megoldását különösebb részletezés nélkül közöljük, hiszen ilyet már nem egyszer csináltunk.

Alaposztályok – setterek, getterek nélkül:

```
public class Csapat {

    private String nev;

    public Csapat(String nev) {
        this.nev = nev;
    }

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

```
public class Meccs {

    private Csapat csapat1;
    private Csapat csapat2;

    private boolean jo;
    private int raadas;

    private static int jatekIdo;

    public Meccs(Csapat csapat1, Csapat csapat2) {
        this.csapat1 = csapat1;
        this.csapat2 = csapat2;
    }

    public int meccsHossz(){
        return jatekIdo + raadas;
    }

    @Override
    public String toString() {
        return csapat1 + " - " + csapat2;
    }
```

```java
public class Ember {
    private String nev;
    private List<Meccs> megnezettMeccsek = new ArrayList<>();

    public Ember(String nev) {
        this.nev = nev;
    }

    public void meccsNezes(Meccs meccs){
        if(!megnezettMeccsek.contains(meccs)){
            megnezettMeccsek.add(meccs);
        }
    }

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

    public List<Meccs> getMegnezettMeccsek() {
        return new ArrayList<>(megnezettMeccsek);
    }
```

Itt kell egy megjegyzés arról, hogy a getter miért nem az eredeti `megnezettMeccsek` listát adja vissza, hanem annak egy másolatát. Az ok az egységbezárás elve. Ha ugyanis az eredetit adnánk vissza, akkor bárki módosítani tudná az eredeti listát, így viszont nem. Természetesen maga a `getMegnezettMeccsek()` lista módosítható (pl. rendezhető), de a getter mindig az eredeti listát adja vissza.

(Egyéb lehetőségként ld.: https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html)

```java
public class Feleseg extends Ember{

    private int szabadIdo;

    public Feleseg(String nev) {
        super(nev);
    }

    @Override
    public void meccsNezes(Meccs meccs) {
        super.meccsNezes(meccs);
        szabadIdo += meccs.meccsHossz();
    }

    @Override
    public String toString() {
        String temp = "";
        if (!super.getNezettMeccsek().isEmpty()) {
            temp = " szabadideje: " + szabadIdo + " perc";
        }
        return super.toString() + temp;
    }
```

```java
public class Ferj extends Ember{

    private int megivottSorokSzama;

    private static int joMeccsSorSzam;
    private static int rosszMeccsSorSzam;

    public Ferj(String nev) {
        super(nev);
    }

    @Override
    public void meccsNezes(Meccs meccs) {
        super.meccsNezes(meccs);
        if(meccs.isJo()) megivottSorokSzama += joMeccsSorSzam;
        else megivottSorokSzama += rosszMeccsSorSzam;
    }

    @Override
    public String toString() {
        String temp = "";
        if(!super.getNezettMeccsek().isEmpty()){
            temp = " " + megivottSorokSzama + " sört ivott";
        }
        return super.toString() + temp;
    }
```

```java
public class Hazaspar implements Comparable<Hazaspar>{

    private Ferj ferj;
    private Feleseg feleseg;

    public Hazaspar(Ferj ferj, Feleseg feleseg) {
        this.ferj = ferj;
        this.feleseg = feleseg;
    }

    public void meccsNezes(Meccs meccs){
        ferj.meccsNezes(meccs);
        feleseg.meccsNezes(meccs);
    }

    @Override
    public String toString() {
        return ferj.getNev() + " - " + feleseg.getNev();
    }

    @Override
    public int compareTo(Hazaspar t) {
        return t.ferj.getSorokSzama() - this.ferj.getSorokSzama();
    }
```

Ne felejtse el tesztelni az osztályokat!

Mivel az adatbevitel hosszabb megbeszélést igényel, ezért most csak jelezzük azt a vezérlő osztályban:

```
public class Main {

    private final int JATEK_IDO = 90;
    private final int JO_MECCS_SOR_DARABSZAM = 2;
    private final int ROSSZ_MECCS_SOR_DARABSZAM = 1;
    private final int MAX_RAADAS = 30;  // legföljebb ennyi a ráadás
    private final double JO_MECCS_SZAZALEK = 0.6;// ilyen százalékban jó a meccs
    private final double NEZES_SZAZALEK = 0.4;  // ekkora az esélye, hogy nézik

    private List<Csapat> csapatok;
    private List<Hazaspar> hazasparok;

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

    private void start() {
        statikusAdatok();
        adatBevitel();
        kiirParok();
        meccsek();
    }

    private void statikusAdatok() {
        Meccs.setJatekIdo(JATEK_IDO);
        Ferj.setJoMeccsSorSzam(JO_MECCS_SOR_DARABSZAM);
        Ferj.setRosszMeccsSorSzam(ROSSZ_MECCS_SOR_DARABSZAM);
    }

    private void adatBevitel() {...9 lines }

    private void kiirParok() {
        System.out.println("házaspárok: ");
        for (Hazaspar hazaspar : hazasparok) {
            System.out.println(hazaspar);
        }
    }

    private void meccsek() {
        Scanner sc = new Scanner(System.in);
        do{
            meccs();
            System.out.print("\nVan még meccs? (i/n)");
        }while(sc.next().equals("i"));
    }
```

```
    public void meccs() {
        // kiválasztunk két véletlen csapatot, arra figyelve, hogy ne legyenek
        // azonosak.
        int veletlenCsapatIndex1 = (int) (Math.random()*csapatok.size());
        int veletlenCsapatIndex2;
        do {
            veletlenCsapatIndex2 = (int) (Math.random() * csapatok.size());
        } while (veletlenCsapatIndex1 == veletlenCsapatIndex2);

        // létrehozunk egy meccset
        Meccs meccs = new Meccs(csapatok.get(veletlenCsapatIndex1),
                csapatok.get(veletlenCsapatIndex2));

        // véletlen értéket generálunk a hosszabbítás számára
        meccs.setRaadas((int) (Math.random() * MAX_RAADAS));

        // ekkora eséllyel jó a meccs
        if (Math.random() < JO_MECCS_SZAZALEK) {
            meccs.setJo(true);
        }
        kiirMeccs(meccs);

        // ekkora eséllyel nézi egy-egy pár a meccset.
        for (Hazaspar hazaspar : hazasparok) {
            if (Math.random() < NEZES_SZAZALEK) {
                hazaspar.meccsNezes(meccs);
            }
        }

        Collections.sort(hazasparok);
        kiirFerjFeleseg();
    }

    private void kiirMeccs(Meccs meccs) {
        System.out.println("\nA meccs: ");
        System.out.println(meccs);
    }

    private void kiirFerjFeleseg() {
        System.out.println("\nFérjek: ");
        for (Hazaspar hazaspar : hazasparok) {
            System.out.println(hazaspar.getFerj());
        }

        System.out.println("\nFeleségek: ");
        for (Hazaspar hazaspar : hazasparok) {
            System.out.println(hazaspar.getFeleseg());
        }
    }
}
```

Beszéljük meg az adatbevitelt. A cél az, hogy könnyedén át tudjuk alakítani a feladatot úgy, hogy adatbázisból olvassa az adatokat, ezért (és persze, azért is, hogy lásson példát interfészre, ill. annak hasznára), most külön csomagba és külön osztályba írjuk az adatbevitelt.

Először egy interfészt hozunk létre, majd ezt implementálja egyszer a fájlból való olvasást végző osztály, egyszer pedig az adatbázisból való olvasást megvalósító. Ha így oldjuk meg a feladatot, akkor a vezérlésben gyakorlatilag alig kell majd valamit változtatni ahhoz, hogy át tudjunk térni az egyik beolvasási módról a másikra.

Az interfészben azt is előírhatjuk, hogy kötelező legyen a kivételkezelés:

```
public interface AdatInput {

    public List<Csapat> csapatBevitelLista() throws Exception;
    public List<Hazaspar> hazasparBevitelLista() throws Exception;
}
```

Az interfészt megvalósító egyik lehetséges osztály:

```
public class FajlBevitel implements AdatInput{

    private String csapatUtvonal;
    private String hazasparUtvonal;
    private String CHAR_SET = "UTF-8";

    public FajlBevitel(String csapatUtvonal, String hazasparUtvonal) {
        this.csapatUtvonal = csapatUtvonal;
        this.hazasparUtvonal = hazasparUtvonal;
    }

    @Override
    public List<Csapat> csapatBevitelLista() throws Exception {
        List<Csapat> csapatok = new ArrayList<>();
        try (InputStream ins = this.getClass().getResourceAsStream(csapatUtvonal);
                Scanner fajlScanner = new Scanner(ins, CHAR_SET)) {

            String sor;
            // Portugália
            while (fajlScanner.hasNextLine()) {
                sor = fajlScanner.nextLine();
                csapatok.add(new Csapat(sor));
            }
        }
        return csapatok;
    }
```

```
    @Override
    public List<Hazaspar> hazasparBevitelLista() throws Exception {
        List<Hazaspar> hazasparok = new ArrayList<>();
        try (InputStream ins = this.getClass().getResourceAsStream(hazasparUtvonal);
                Scanner fajlScanner = new Scanner(ins, CHAR_SET)) {

            String sor;
            String adatok[];
            //István;Ildikó
            while (fajlScanner.hasNextLine()) {
                sor = fajlScanner.nextLine();
                adatok = sor.split(";");
                hazasparok.add(new Hazaspar(new Ferj(adatok[0]),
                                            new Feleseg(adatok[1])));
            }
        }
        return hazasparok;
    }
}
```

Természetesen úgy is lehetne implementálni az interfészt (bár az elvi lehetőségen kívül nem sok értelme volna), hogy a `csapatBevitelLista()`, illetve a `hazasparBevitellista()` metódusok egy-egy konstans listát adnának vissza.

A minket érdeklő másik implementáció az, hogy az adatokat egy adatbázisból olvassuk. Ehhez természetesen kell majd egy adatbázis, amelyben van két adattábla:

CSAPATOK – attribútumai: ID (int) és NEV (varchar)

PAROK – attribútumai: ID (int), FERFINEV (varchar), NOINEV (varchar).

Ez az adatbázis bárhol lehet – külső adatszerveren, vagy lokális gépen, és bármilyen adatbázis-kezelő rendszer lehet, a JDBC biztosítja a hozzájuk való hozzáférést. Bár éles alkalmazásra nem igazán használatos, de tanulásra kiválóan alkalmas a Java saját adatbázis-kezelője, a Derby. Lokális változatának használatával az adatbázist a projekten belül tárolhatjuk, így könnyedén megoldható a kezelése. De ha módja van saját adatbázis-szervert működtetni, akkor természetesen azt is kipróbálhatja. Bármilyen rendszert is használ, az adatbázishoz mindenképpen kapcsolódni kell. Ezt biztosítja a `Connection` típusú `kapcsolat` változó. Ennek segítségével tudunk majd létrehozni egy utasításobjektumot, amely végrehajtatja a megadott SQL utasítást, és visszaadja a várt eredményhalmazt. Az adatok kinyeréséhez ezen kell végigiterálni, mégpedig úgy, hogy az adatbázisnak megfelelő formátumú adatokat Java formátumúra alakítjuk (illetve alakíttatjuk a megfelelő metódusokkal).

A lehetséges implementáció:

```java
public class AdatBazisBevitel implements AdatInput{

    private Connection kapcsolat;

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

    @Override
    public List<Csapat> csapatBevitelLista() throws Exception {
        List<Csapat> csapatok = new ArrayList<>();
        // Ezt az SQL utasítást kell majd végrehajtani.
        String sqlUtasitas = "SELECT * FROM CSAPATOK";

        // Megkérjük a kapcsolatot, hogy hozzon létre egy Statement
        // típusú, utasitasObjektum nevű változót.
        // Az utasitasObjektum hajtatja végre a megadott SQL utasítást.
        // Eredményként egy ResultSet típusú eredményhalmazt kapunk.
        try (Statement utasitasObj = kapcsolat.createStatement();
                ResultSet eredmenyHz = utasitasObj.executeQuery(sqlUtasitas)) {
            String nev;
            // Végigjárjuk az eredményhalmazt, és a kapott adatokból
            // létrehozzuk a csapatok listáját.
            while (eredmenyHz.next()) {
                nev = eredmenyHz.getString("nev");
                csapatok.add(new Csapat(nev));
            }
        }
        return csapatok;
    }

    @Override
    public List<Hazaspar> hazasparBevitelLista() throws Exception {
        List<Hazaspar> hazasparok = new ArrayList<>();
        String sqlUtasitas = "SELECT * FROM PAROK";
        try (Statement utasitasObj = kapcsolat.createStatement();
                ResultSet eredmenyHz = utasitasObj.executeQuery(sqlUtasitas)) {
            String noiNev, ferfiNev;
            while (eredmenyHz.next()) {
                noiNev = eredmenyHz.getString("noiNev");
                ferfiNev = eredmenyHz.getString("ferfinev");
                hazasparok.add(new Hazaspar(new Ferj(ferfiNev),
                                            new Feleseg(noiNev)));
            }
        }
        return hazasparok;
    }
}
```

Lássuk ezek után a vezérlés adatbeolvasó metódusát:

1. **Fájlból** olvassuk az adatokat. Ekkor természetesen meg kell adnunk azt is, hogy hol találjuk őket.

```
private final String CSAPAT_UTVONAL = "/adatok/csapatok.txt";
private final String HAZASPAR_UTVONAL = "/adatok/parok.txt";

private void adatBevitel() {
    try {
        AdatInput adatInput =
                new FajlBevitel(CSAPAT_UTVONAL, HAZASPAR_UTVONAL);
        csapatok = adatInput.csapatBevitelLista();
        hazasparok = adatInput.hazasparBevitelLista();
    } catch (Exception ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Hamarosan látni fogjuk az interfész használatának előnyét: ha át akarunk térni adatbázisból való olvasásra, akkor lényegében csak az `adatInput` változó példányosítását kell megváltoztatnunk, minden más változatlan.

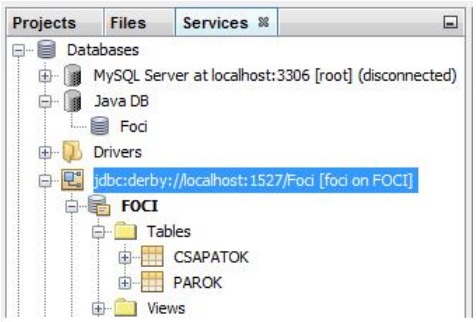
2. **Adatbázisból** olvassuk az adatokat. Ekkor nyilván nincs szükség a két fájl elérési útvonalára, viszont meg kell adnunk azt, hogy milyen fajta adatbázist használunk, és hol találjuk azt, vagyis létre kell majd hoznunk a beolvasó osztály példányosításához szükséges kapcsolat példányt. Ez azonban egy lezárandó (`Closeable`) objektum, ezért a `try` blokk fejében hozzuk létre.

```
private void adatBevitel() {
    // try {
    //     AdatInput adatInput =
    //             new FajlBevitel(CSAPAT_UTVONAL, HAZASPAR_UTVONAL);

    try (Connection kapcsolat = kapcsolodas()) {
        AdatInput adatInput = new AdatBazisBevitel(kapcsolat);

        csapatok = adatInput.csapatBevitelLista();
        hazasparok = adatInput.hazasparBevitelLista();
    } catch (Exception ex) {
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
    }
}

private Connection kapcsolodas() throws ClassNotFoundException,
                                        SQLException {

    // az adatbázis driver meghatározása
    Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
    // az adatbázis definiálása
    String url = "jdbc:derby://localhost:1527/FOCI";
    // kapcsolodas az adatbázishoz
    return DriverManager.getConnection(url, "foci", "foci");
}
```

(Az `adatBevitel()` metódusban azért maradt benne a kommentezett rész, hogy jobban lássa, lényegében valóban csak a példányosítást kellett megváltoztatni.)

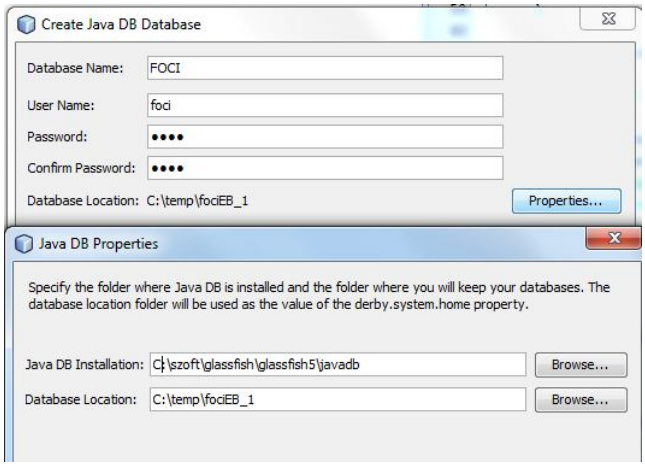
Megjegyzés: Láthatjuk, milyen előnyökkel jár az, ha interfészt definiálunk az osztályok létrehozása előtt. Egy „igazi” öröklődés nagyon sokszor nem az (esetleg absztrakt) ősosztályról indul, hanem egy jól megtervezett interfészről.

Megírtuk tehát a beolvasást, de még nincs adatbázis, és még nem kapcsolódtunk az adatbázis-szerverhez. Ennek beállításához jó segítséget nyújt a https://netbeans.org/kb/docs/ide/java-db.html tutoriál, de itt is lesz róla néhány szó. (Ha nem NetBeans-t használ a feladatok megoldásához, vagy más adatbázisszervert, akkor értelemszerűen annak megfelelően kell létrehozni az adatbázist.)

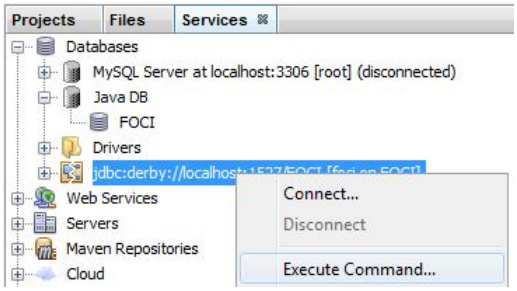


A Services fülön tudjuk létrehozni az adatbázist (Java DB – jobb egérgomb – Create Database):
Ekkor meg kell adnunk az adatbázis nevét (esetünkben FOCI), a felhasználónevet (foci) és a jelszót (most ez is foci).

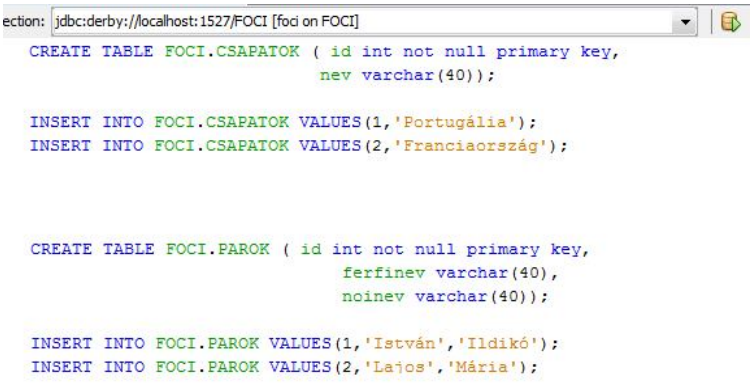
A tulajdonságokat (Properties) is be kell állítanunk, vagyis meg kell adnunk az adatbázist működtető javadb fájl helyét, illetve az adatbázis helyét – ez utóbbit célszerű az aktuális projektbe tenni.



FONTOS: A Java DB installációs helyét elvileg automatikusan felajánlja a rendszer, gyakorlatilag azonban sajnos a NetBeans nem minden Java al-verzióval működik jól együtt, és előfordulhat, hogy a felkínált nem működik. Ekkor külön le kell tölteni a glassfish-t (kis csomagolt állomány), kibontani, és erre hivatkozni.



Ha sikerült létrehozni az adatbázist, akkor a kék csíkkal jelölt soron jobb egérgombbal kattintva elérhetővé válik az Execute Command menüpont. Ennek hatására megnyílik egy szerkesztő-ablak, itt meg tudjuk írni az SQL parancsokat, és végre is tudjuk hajtatni.



A parancsokat most bemásolhatjuk a megadott fájlokból, futtatni ennek az ikonnak a hatására lehet:

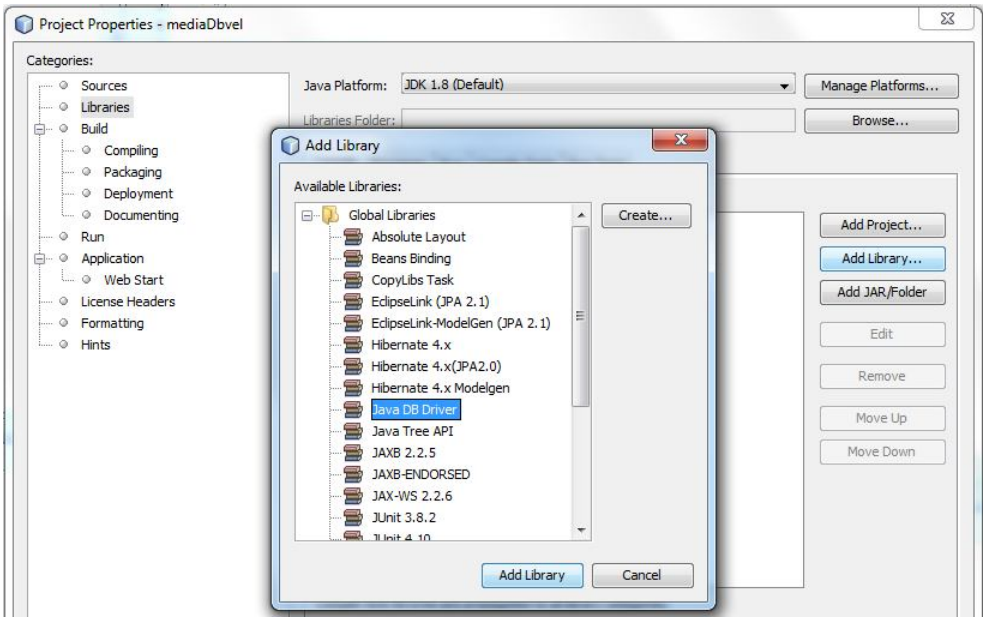
A táblanéven jobb egérgombot nyomva a View Data menüpont hatására táblázatos formában is láthatjuk az adatokat.

Ezek után boldogan futtathatjuk az alkalmazást, ámde még csalódás ér, ezt a hibaüzenetet kapjuk:

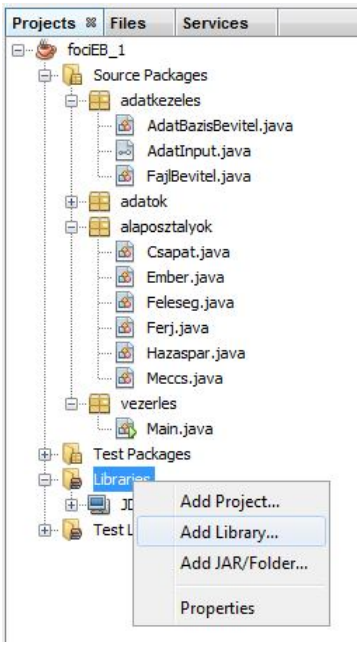
```
java.lang.ClassNotFoundException: org.apache.derby.jdbc.EmbeddedDriver
```

Ilyen hibaüzenet esetén legelső dolgunk, hogy felkeressük google barátunkat. ☺

Egyébként pedig az a baj, hogy nem találja a Derby driver-t, hiszen ez nem tartozik az alapértelmezetten betöltött állományok közé. De egyszerű a megoldás: projektnév – jobb egérgomb – Properties és:



Ugyanez az ablak közvetlenül a projekt fülön is előhívható:



Megjegyzések:

1. Másik gépre rakva, vagy valamikor később ismét futtatva a projektet, nem kell újból létrehozni az adatbázist, elég, ha a megadott útvonalon kapcsolódunk hozzá

(Services fül, Java DB, jobb egérgomb, Connect – természetesen a Properties ablakban helyesen kell megadni az adatbázis elérhetőségét.)

2. Nyilván elég hamar kényelmetlenné válik, ha külső library-kat vagy .jar fájlokat így, „kézzel” kell hozzáadni a projektünkhöz. Ennek automatizálására (és még sok minden másra) szolgál a Maven szoftver – ha ismeri, akkor használja már most, ha nem, akkor később még visszatérünk rá.