Tartalom

III. GYAKORLAT: JAVA DATABASE CONNECTIVITY (JDBC) 1

I. FÜGGELÉK: UNIX ÖSSZEFOGLALÓ A LEGFONTOSABB PARANCSOKRÓL 18

III. FÜGGELÉK: REGULÁRIS KIFEJEZÉSEK 20

III. gyakorlat: Java Database Connectivity (JDBC)[1]

Szerzők: Mátéfi Gergely, Kollár Ádám, Remeli Viktor, Kamarás Roland, Varsányi Márton

1. BEVEZETÉS 1
2. ADATBÁZIS-KEZELÉS KLIENS-SZERVER ARCHITEKTÚRÁBAN 1
3. A JDBC 1.2 API 3
   3.1. *A programozói felület felépítése* 3
   3.2. *Adatbázis kapcsolatok menedzsmentje* 3
   3.3. *SQL utasítások végrehajtása* 4
   3.4. *Eredménytáblák kezelése* 6
   3.5. *Hibakezelés* 7
   3.6. *Tranzakciókezelés* 7
   3.7. *Adatbázis információk* 7
4. AZ ORACLE JDBC MEGHAJTÓI 8
5. EGY PÉLDA WEBSTART ALKALMAZÁSRA 8
   5.1. *Java Web Start technológia* 8
   5.2. *Minta alkalmazás, JavaFX* 9
6. FELHASZNÁLT IRODALOM 16
7. FÜGGELÉK: ORACLE ADATTÍPUSOK ELÉRÉSE JDBC-BŐL 17

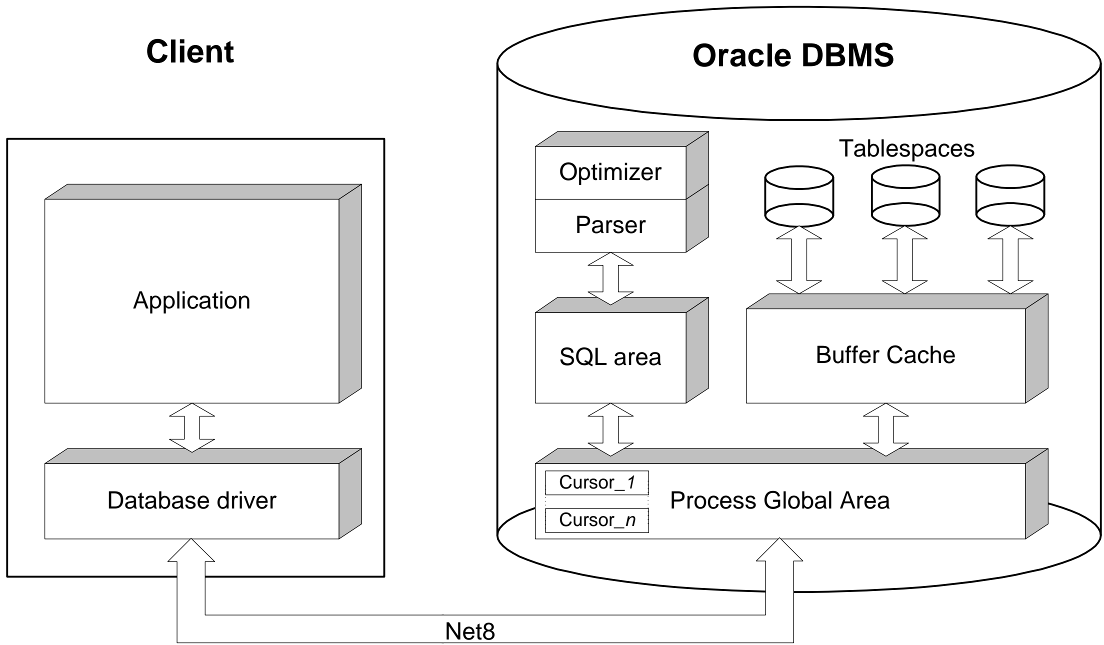
1. Bevezetés

A Java Database Connectivity (JDBC) a Javából történő adatbázis elérés gyártófüggetlen *de facto* szabványa. A JDBC programozói felület (Application Programming Interface, API) Java osztályok és interfészek halmaza, amelyek relációs adatbázisokhoz biztosítanak alacsonyszintű hozzáférést: kapcsolódást, SQL utasítások végrehajtását, a kapott eredmények feldolgozását. Az interfészeket a szállítók saját *meghajtói (driverei)* implementálják. A meghajtóknak – a Java működésének megfelelően – elegendő futási időben rendelkezésre állniuk, így a programfejlesztőnek lehetősége van a Java alkalmazást az adatbázis-kezelő rendszertől (DBMS-től) függetlenül elkészítenie.
Jelen labor célja Java és JDBC környezeten keresztül az adatbázisalapú, kliens-szerver architektúrájú alkalmazások fejlesztésének bemutatása. Az első alfejezetben a kliens-szerver architektúrát mutatjuk be röviden, ezt követően a JDBC legfontosabb nyelvi elemeit foglaljuk össze, végül konkrét példán demonstráljuk a JDBC használatát. A segédlet a laborkörnyezet adottságaihoz igazodva a JDBC 1.2 változatú API bemutatására korlátozódik.

2. Adatbázis-kezelés kliens-szerver architektúrában[2]

Kliens-szerver architektúra mellett a kliensen futó alkalmazás hálózaton keresztül, a gyártó által szállított meghajtó segítségével éri el a DBMS-t. A meghajtó az alkalmazástól függően lehet például C nyelvű könyvtár, ODBC- vagy JDBC- meghajtó.

[1] Ld. még a segédlet végén a reguláris kifejezésekről szóló függeléket is.
[2] Az alfejezet az alapfogalmakat az Oracle RDBMS (jelentősen leegyszerűsített) működésén keresztül mutatja be, maguk a fogalmak azonban nem Oracle-specifikusak



Az adatbázis műveleteket megelőzően a felhasználónak egy ún. *adatbázis-kapcsolatot* (*sessiont*) kell felépítenie, melynek során autentikálja magát a DBMS felé. Oracle rendszerben a felépítés során a DBMS a session számára erőforrásokat allokál: memóriaterületet (Process Global Area, PGA) foglal és kiszolgálófolyamatot (server process) indít.[3]

A session élete során a kliens adatbázis műveleteket kezdeményezhet, melyeket a meghajtó SQL utasításként továbbít a DBMS felé. Az utasítást a DBMS több lépésben dolgozza fel. A feldolgozás kezdetén a kiszolgálófolyamat *memóriaterületet különít el* a PGA-ban: itt tárolódnak a feldolgozással kapcsolatos információk, többek között a lefordított SQL utasítás és az eredményhalmazbeli pillanatnyi pozíció is (ld. lejjebb). Az elkülönített memóriaterület leíróját *kurzornak* (*cursor*), a feldolgozás megkezdését a *kurzor megnyitásának* is nevezik. Egy session egy időben több megnyitott kurzorral is rendelkezhet.

A feldolgozás első lépése az SQL *utasítás elemezése* (*parsing*), melynek során a DBMS lefordítja az utasítást tartalmazó stringet, ezt követi az érintett adatbázis objektumokhoz tartozó *hozzáférési jogosultságok ellenőrzése*. A sikeresen lefordított utasításhoz az Optimizer készíti el az ún. *végrehajtási tervet* (*execution plan*). A végrehajtási terv tartalmazza az utasítás által érintett sorok fizikai leválogatásának lépéseit: mely táblából kiindulva, mely indexek felhasználásával, hogyan történjék a leválogatás. Mivel az elemzés és a végrehajtási terv meghatározás számításigényes művelet, a DBMS gyorsítótárban (*SQL area*) tárolja legutóbbi SQL utasítások végrehajtási tervét.

Egy adatbázisalapú alkalmazás futása során tipikusan néhány, *adott szerkezetű* SQL utasítást használ, de *eltérő paraméterezéssel*. Egy számlázószoftver például rendre ugyanazon adatokat hívja le az ügyfelekről, a lekérdezésekben mindössze az ügyfélazonosító változik. Az SQL nyelv lehetőséget teremt ezen utasítások paraméteres megírására:

```
SELECT NEV, CIM, ADOSZAM FROM UGYFEL WHERE UGYFEL_ID = ?
```

A paraméteres SQL utasítást az adatbázis-kezelő az első feldolgozáskor fordítja le, a későbbi meghívások során már nincs szükség újrafordításra. A gyorsítótár használatát az teszi lehetővé, hogy a feldolgozás során a *paraméterek behelyettesítése csak az elemzést és végrehajtási terv meghatározást követően történik*.

A végrehajtási terv meghatározása és az esetleges behelyettesítések után az SQL utasítás *végrehajtódik*. SELECT típusú lekérdezések esetén a kiválasztott sorok logikailag egy *ered-*

[3] Van lehetőség osztott szerverfolyamatok használatára is.

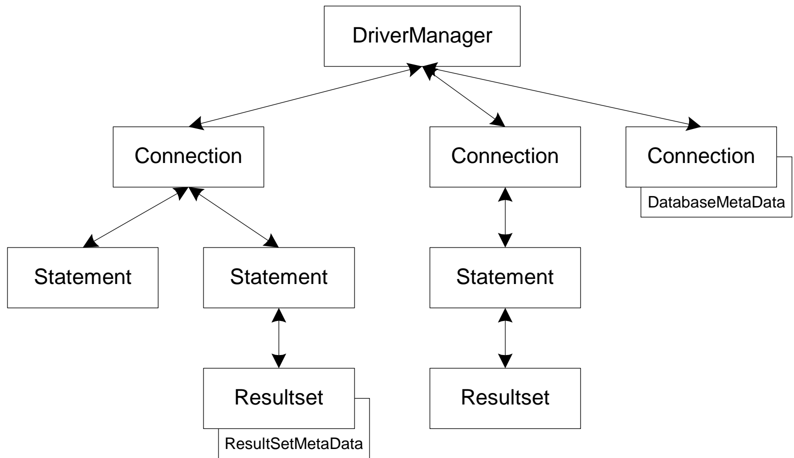
ménytáblát képeznek, melynek sorait a kliens egyenként[4] kérdezheti le az ún. *fetch* művelettel.
Az eredménytábla kiolvasása, illetve a tranzakció befejezése (commit/rollback) után a feldolgozásra elkülönített memóriaterület felszabadul, a *kurzor bezáródik*.

3. A JDBC 1.2 API

3.1. A programozói felület felépítése

A JDBC API Java osztályok és interfészek halmazából áll. A leglényegesebb osztályok és interfészek:

- java.sql.DriverManager osztály az adatbázis URL feloldásáért és új adatbázis kapcsolatok létrehozásáért felelős
- java.sql.Connection interfész egy adott adatbázis kapcsolatot reprezentál
- java.sql.DatabaseMetaData interfészen keresztül az adatbázissal kapcsolatos (meta)információkat lehet lekérdezni
- java.sql.Statement interfész SQL utasítások végrehajtását vezérli
- java.sql.ResultSet interfész egy adott lekérdezés eredményeihez való hozzáférést teszi lehetővé
- java.sql.ResultSetMetaData interfészen keresztül az eredménytábla metainformációi kérdezhetők le



3.2. Adatbázis kapcsolatok menedzsmentje

A JDBC meghajtók menedzsmentjét, új kapcsolatok létrehozását-lebontását a `java.sql.DriverManager` osztály végzi. A DriverManager tagváltozói és metódusai statikusak, így példányosítására nincs szükség az alkalmazásban. Egy új kapcsolat létrehozása a DriverManager-en keresztül egyetlen parancssorral elvégezhető:

```
Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");
```

A `getConnection` függvény első paramétere az adatbázist azonosító URL string, a második és harmadik paramétere a adatbázis felhasználót azonosító név és jelszó. Az URL tartalma adatbázisfüggő, struktúrája konvenció szerint a következő:

```
jdbc:<subprotocol>:<subname>
```

[4] A hatékony működés érdekében az adatbázis meghajtó egy fetch során kötegelten több sort is lehozhat.

ahol a <subprotocol> az adatbázis kapcsolódási mechanizmust azonosítja és a <subname> tag tartalmazza az adott mechanizmussal kapcsolatos paramétereket. Példaképpen a "Fred" által azonosított ODBC adatforráshoz a következő utasítással kapcsolódhatunk:

```
String url = "jdbc:odbc:Fred";
Connection con = DriverManager.getConnection(url, "Fernanda", "J8");
```

Ha a meghajtó által előírt URL már tartalmazza a felhasználói nevet és jelszót, akkor a függvény második és harmadik paramétere elmaradhat. A `getConnection` függvény meghívásakor a DriverManager egyenként lekérdezi a *regisztrált* JDBC meghajtókat, és az *első* olyan meghajtóval, amely képes a megadott URL feloldására, felépíti az adatbázis kapcsolatot. A kívánt műveletek elvégzése után a kapcsolatot a `Connection.close` metódusával kell lezárni. A `close` metódus a `Connection` objektum megsemmisítésekor (garbage collection) automatikusan is meghívódik.
A meghajtókat használatba vételük előtt *be kell tölteni* és *regisztrálni* kell a DriverManager számára. A programozónak általában csak a meghajtóprogram betöltéséről gondoskodnia, a meghajtók a statikus inicializátorukban rendszerint automatikusan regisztráltatják magukat. A betöltés legegyszerűbb módja a `Class.forName` metódus használata, például:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Biztonsági megfontolások miatt *applet* csak olyan meghajtókat használhat, amelyek vagy a lokális gépen helyezkednek el, vagy ugyanarról a címről lettek letöltve, mint az applet kódja. A „megbízhatatlan forrásból" származó appletekkel szemben a „megbízható" applikációkat teljes értékű programként futtatja a JVM, azokra a megkötés nem vonatkozik.

3.3. SQL utasítások végrehajtása

Egyszerű SQL utasítások végrehajtására a `Statement` interfész szolgál. Egy utasítás végrehajtásához az interfészt megvalósító meghajtó osztályt *példányosítani* kell, majd a példány – SQL utasítástól függő - *végrehajtó metódusát* kell meghívni. Egy `Statement` példány az aktív adatbázis-kapcsolatot reprezentáló `Connection` példány `createStatement` metódusával hozható létre:

```
Statement stmt = con.createStatement();
```

A `Statement` osztály háromféle végrehajtó metódussal rendelkezik:
`executeQuery`: a paraméterében megadott SQL lekérdezést végrehajtja, majd az eredménytáblával (`ResultSet`) tér vissza. A metódus lekérdező (SELECT) utasítások végrehajtására használandó.
`executeUpdate`: a paraméterében megadott SQL utasítást végrehajtja, majd a módosított sorok számával tér vissza. Használható mind adatmanipulációs (DML), mind adatdefiníciós (DDL) utasítások végrehajtására. DDL utasítások esetén a visszatérési érték 0.
`execute`: a paraméterében megadott SQL utasítást hajtja végre. Az előző két metódus általánosításának tekinthető. Visszatérési értéke true, ha az utasítás által visszaadott eredmény `ResultSet` típusú, ekkor az a `Statement.getResultSet` metódussal kérdezhető le. Az utasítással módosított sorok számát a `Statement.getUpdateCount` metódus adja vissza.
A következő példában a klasszikus Kávészünet Kft. számlázószoftveréhez hozzuk létre a számlák adatait tartalmazó táblát:

```
int n = stmt.executeUpdate("CREATE TABLE COFFEES ( " +
          "COF_NAME VARCHAR(32)," +
          "SUP_ID   NUMBER(8),  " +
          "PRICE    NUMBER(6,2)," +
          "SALES    NUMBER(4),  " +
          "TOTAL    NUMBER(6,2) )");
```

A vállalkozás beindulása után az alábbi utasítással tudjuk lekérdezni az eddigi vásárlásokat:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM COFFEES");
```

Egy utasítás végrehajtása akkor zárul le, ha az összes visszaadott eredménytábla fel lett dolgozva (minden sora kiolvasásra került). Az utasítás végrehajtása manuálisan is lezárható a `Statement.close` metódussal. Egy `Statement` objektum végrehajtó függvényének újbóli meghívása lezárja ugyanazon objektum korábbi lezáratlan végrehajtását.

A *paraméteres SQL utasítások* kezelése némiképp eltér az egyszerű SQL utasításokétól. A JDBC-ben a `PreparedStatement` interfész reprezentálja a paraméteres SQL utasításokat. Létrehozása az egyszerű `Statement`-hez hasonlóan, a kapcsolatot reprezentáló Connection példány `prepareStatement` metódusával történik, *a paraméteres SQL utasítás megadásával*:

```
PreparedStatement updateSales = con.prepareStatement(
  "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

A `PreparedStatement`-ben tárolt utasítás – kérdőjelekkel jelzett – paraméterei a `setXXX` metóduscsalád segítségével állíthatók be. A végrehajtásra a `Statement` osztálynál már megismert `executeQuery`, `executeUpdate` és `execute` metódusok használhatók, amelyeket itt *argumentum nélkül* kell megadni. A következő kódrészlet a COFFEES táblába történő adatfelvitelt szemlélteti, paraméteres SQL utasítások segítségével:

```
PreparedStatement updateSales;
String updateString =
  "update COFFEES set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
                     "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
  updateSales.setInt(1, salesForWeek[i]);
  updateSales.setString(2, coffees[i]);
  updateSales.executeUpdate();
}
```

A `setXXX` metódusok két argumentumot várnak. Az első argumentum a beállítandó SQL paraméter indexe; a paraméterek az SQL utasításban balról jobbra, 1-gyel kezdődően indexelődnek. A második argumentum a beállítandó érték. A bemeneti paraméterek megadásánál a JDBC *nem végez implicit típuskonverziót*, a programozó felelőssége, hogy az adatbázis-kezelő által várt típust adja meg. Null érték a `PreparedStatement.setNull` metódussal állítható be. Egy beállított paraméter az SQL utasítás többszöri lefuttatásánál is felhasználható.

Tárolt eljárások és függvények meghívására a `CallableStatement` interfész használható. A `CallableStatement` a kapcsolatot reprezentáló `Connection` példány `prepareCall` metódusával példányosítható a korábbiakhoz hasonló módon. A `CallableStatement` interfész a `PreparedStatement` interfész leszármazottja, így a bemeneti (IN) paraméterek a `setXXX` metódusokkal állíthatók. A kimeneti (OUT) paramétereket a végrehajtás előtt a `CallableStatement.registerOutParameter` metódussal, a típus megadásával regisztrálni kell. A végrehajtást követően a `getXXX` metóduscsalád használható a kimeneti paraméterek lekérdezésére, mint a következő példa szemlélteti:

```
CallableStatement stmt = conn.prepareCall("call getTestData(?,?)");
stmt.registerOutParameter(1,java.sql.Types.TINYINT);
stmt.registerOutParameter(2,java.sql.Types.DECIMAL);
stmt.executeUpdate();
```

```
byte x = stmt.getByte(1);
BigDecimal n = stmt.getBigDecimal(2);
```

A `setXXX` metódusokhoz hasonlóan, a `getXXX` metódusok *sem végeznek típuskonverziót*: a programozó felelőssége, hogy az adatbázis által kiadott típusok, a `registerOutParameter` és a `getXXX` metódusok összhangban legyenek.

3.4. Eredménytáblák kezelése

A lekérdezések eredményeihez a `java.sql.ResultSet` osztályon keresztül lehet hozzáférni, melyet a `Statement` interfészek `executeQuery`, illetve `getResultSet` metódusai példányosítanak. A `ResultSet` által reprezentált eredménytáblának mindig az *aktuális* (kurzor által kijelölt) sora érhető el. A kurzor kezdetben mindig az első sor *elé* mutat, a `ResultSet.next` metódussal léptethető a következő sorra.[5] A `next` metódus visszatérési értéke false, ha a kurzor *túlment* az utolsó soron, true egyébként.
Az aktuális sor mezőinek értéke a `getXXX` metóduscsaláddal kérdezhető le.[6] A mezőkre a `getXXX` függvények kétféle módon hivatkozhatnak: oszlopindexszel, illetve az oszlopnevekkel. Az oszlopok az SQL lekérdezésben balról jobbra, 1-gyel kezdődően indexelődnek. Az oszlopnevekkel történő hivatkozás a futásidőben történő leképezés miatt kevésbé hatékony, ellenben kényelmesebb megoldást kínál. A `ResultSet.getXXX` metódusok, szemben a `PreparedStatement` és a `CallableStatement getXXX` függvényeivel, *automatikus típuskonverziót végeznek*. Amennyiben a típuskonverzió nem lehetséges (például a `getInt` függvény meghívása a VARCHAR típusú, "foo" stringet tartalmazó mezőre), `SQLException` kivétel lép fel. Ha a mező *SQL NULL értéket tartalmaz*, akkor a `getXXX` metódus zérus, illetve Java null értéket ad vissza, a `getXXX` függvénytől függően. A mező értékének kiolvasása *után* a `ResultSet.wasNull` metódussal ellenőrizhető, hogy a kapott érték SQL NULL értékből származott-e.[7]
Az eredménytábla lezárásával a programozónak általában nem kell foglalkoznia, mivel ez a `Statement` lezáródásával automatikusan megtörténik. A lezárás ugyanakkor manuálisan is elvégezhető a `ResultSet.close` metódussal.

Az eredménytáblákkal kapcsolatos metainformációkat a `ResultSetMetaData` interfészen keresztül lehet elérni. Az interfészt megvalósító objektumot a `ResultSet.getMetaData` metódus adja vissza. A `ResultSetMetaData getColumnCount` metódusa az eredménytábla oszlopainak számát, a `getColumnName(int column)` az indexszel megadott oszlop elnevezését adja meg.
A következő példa a `ResultSet` és a `ResultSetMetaData` használatát szemlélteti. A lekérdezés első oszlopa integer, a második String, a harmadik bájtokból alkotott tömb típusú:

```
Statement stmt = conn.CreateStatement();
ResultSet r = stmt.executeQuery("SELECT a, b, c FROM table1");
ResultSetMetaData rsmd = r.getMetaData();
for (int j = 1; j <= rsmd.getColumnCount(); j++) {
  System.out.print(rsmd.getColumnName(j));
}
System.out.println;
while (r.next()) {
  // Aktuális sor mezőinek kiíratása
  int i = r.getInt("a");
```

[5] Csak a JDBC 2.0 változatban van lehetőség van a kurzor visszafelé léptetésre és adott sorra történő mozgatására (ha ezt a meghajtó is támogatja)
[6] Felsorolásuk a Függelékben
[7] A mezőérték kiolvasása *előtti* nullitásvizsgálatot nem támogatja minden adatbázis-kezelő rendszer, emiatt maradt ki a JDBC 1.2 API-ból.

```
  String s = r.getString("b");
  byte b[] = r.getBytes("c");
  System.out.println(i + " " + s + " " + b[0]);
}
stmt.close();
```

3.5. Hibakezelés

Ha az adatbázis kapcsolat során bármiféle hiba történik, Java szinten `SQLException` kivétel lép fel. A hiba szövegét az `SQLException.getMessage`, a kódját az `SQLException.getErrorCode`, az X/Open SQLstate konvenció szerinti állapotleírást az `SQLException.getSQLState` metódusok adják vissza.

3.6. Tranzakciókezelés

A `Connection` osztállyal reprezentált adatbázis-kapcsolatok *alapértelmezésben auto-commit módban* vannak. Ez azt jelenti, hogy minden SQL utasítás (`Statement`) egyedi tranzakcióként fut le és végrehajtása után azonnal véglegesítődik (commit). Az alapértelmezés átállítható a `Connection.setAutoCommit(false)` metódussal. Ebben az esetben a tranzakciót programból kell véglegesíteni illetve visszavonni a `Connection.commit` ill. `Connection.rollback` metódusokkal, a következő példának megfelelően:

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
  "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
  "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

3.7. Adatbázis információk

Az adatbázissal kapcsolatos információkhoz (metaadatokhoz) a `DatabaseMetaData` interfészen keresztül lehet hozzáférni. Az interfészt megvalósító osztályt a kapcsolatot reprezentáló `Connection` példány `getMetaData` metódusa adja vissza:

```
DatabaseMetaData dbmd = conn.getMetaData();
```

A `DatabaseMetaData` interfész egyes metódusai a lekérdezett információtól függően egyszerű Java típussal, vagy `ResultSet`tel térnek vissza. Az alábbi táblázat néhány fontosabb metódust sorol fel.

Metódus elnevezése	Visszatérési érték	Leírás
`getDatabaseProductName`	`String`	Adatbázis termék elnevezése
`getDatabaseProductVersion`	`String`	Adatbázis termék verziószáma

`getTables(String catalog, String schemaPattern,`
 	
`String tableNamePattern, String types[])`
	`ResultSet`	A megadott keresési feltételnek eleget tevő táblákat listázza

4. Az Oracle JDBC meghajtói

Az Oracle cég két kliensoldali JDBC meghajtót fejlesztett ki: a *JDBC OCI Drivert* és a *JDBC Thin Drivert*.

A **JDBC OCI Driver** a JDBC metódusokat OCI könyvtári hívásokként implementálja. Az Oracle Call Level Interface (OCI) az Oracle C nyelven megírt standard kliens oldali programozási felülete (adatbázis meghajtója), amely magasabb szintű fejlesztőeszközök számára biztosít hozzáférést az adatbázis-kezelő szolgáltatásaihoz. Egy JDBC-beli metódus (pl. utasításvégrehajtás) meghívásakor a JDBC OCI Driver továbbítja a hívást az OCI réteghez, amelyet az az SQL*Net ill. Net8 protokollon keresztül juttat el az adatbázis-kezelőhöz. A C könyvtári hívások miatt a JDBC OCI Driver platform- és operációs rendszer-függő; a natív kód miatt ugyanakkor hatékonyabb a tisztán Java-ban megírt Thin meghajtónál.

A **JDBC Thin Driver** teljes egészében Java-ban íródott meghajtó. A Thin Driver tartalmazza az SQL*Net/Net8 protokoll egyszerűsített, TCP/IP alapú implementációját; így egy JDBC metódushívást közvetlenül az adatbázis-kezelőhöz továbbít. A tiszta Java implementáció miatt a JDBC Thin Driver platformfüggetlen, így a Java applettel együtt letölthető. Az egyszerűsített implementáció miatt nem minden OCI funkciót biztosít (pl. titkosított kommunikáció, IP-n kívüli protokollok stb. nem támogatottak).

Az Oracle az adatbázisok címzésére – illeszkedve a JDBC konvencióhoz – a következő URL struktúrát használja:

```
jdbc:oracle:drivertype:user/password@host:port:sid
```

ahol a drivertype *oci7*, *oci8* vagy *thin* lehet; a host az adatbázis-szerver DNS-neve, a port a szerveroldali TNS listener portszáma, a sid pedig az adatbázis azonosítója. A felhasználói név és a jelszó a getConnection függvény második és harmadik paraméterében is megadható, ekkor az URL-ből a user/password szakasz elhagyandó.

5. Egy példa WebStart alkalmazásra

5.1. Java Web Start technológia

A WebStart a Java 1.4-től bevezetett, webről történő közvetlen alkalmazástelepítést és indítást könnyítő platformfüggetlen technológia. Egyetlen linkre való kattintással váltja ki a laikus felhasználók számára egyébként nehezen elsajátítható parancssori formulát. Az alkalmazás elindításán kívül továbbá biztosítja, hogy mindig a legfrissebb verzió legyen a kliens gyorsítótárába töltve, és az is fusson (az internetkapcsolat emiatt alap esetben kötelező). A WebStart használatához a Java alkalmazásunkon semmit nem kell változtatni, azt leszámítva, hogy kötelezően JAR csomag(ok)ba kell rendeznünk azt. Ezen kívül egy JNLP (Java Network Launching Protocol) telepítés-leíró állományt kell mellékelnünk és kézzel megszerkesztenünk.

A WebStarttal indított alkalmazás az appletekhez hasonlóan homokozóban (sandbox) fut, azaz olyan futtatókörnyezetben, mely korlátozza a helyi fájlrendszerekhez és a hálózathoz való hozzáférést. Az alkalmazás korlátlan jogokat kaphat azonban, ha minden komponense digitális aláírással rendelkezik, a JNLP-ben kimondottan kéri a plusz jogokat, és a felhasználó az aláíróban, illetve annak hitelesítés-szolgáltatójában megbízván ezeket explicit módon meg is adja (az engedélyezés csak első futáskor kell, később gyorstárazásra kerül). Számunkra ez azért fontos, mert egyébként nem tudnánk a kliensen futó programmal az adatbázishoz csatlakozni (hiszen olyan hálózati erőforrást szeretnénk használni, mely nem egyezik meg a letöltés helyével). A kliens egyéb erőforrásait a homokozóból a JNLP API rétegen keresztül tudjuk elérni (erre a mérésen nem lesz szükség).

5.2. Minta alkalmazás, JavaFX

A labor alkalmával a hallgatók rendelkezésére bocsátott minta alkalmazás – melynek forrása ZIP archívumban összecsomagolva letölthető a http://rapid.eik.bme.hu/~varsanyi.marton/jdbc/lab5jdbc.zip mérések során használt Oracle adatbázis szerverhez, majd lekérdezi és megjeleníti az 'Oktatas' sémában található 'Szemelyek' tábla első 20 rekordját.

Az alkalmazás könyvtárszerkezetének leírását, valamint fordításának és futtatásának menetét részletesen tartalmazza a tárgy JDBC mérésének weboldaláról letölthető hallgatói útmutató. Jelen leírás célja, hogy áttekintést adjon a minta alkalmazás funkcionalitásáról, architektúrájáról, valamint a létrehozása során alkalmazott technikákról. Jelen ismertetőben tehát az alkalmazás felületét, valamint Java forráskódját vesszük szemügyre, mely a fenti címről letölthető csomag *src*, illetve *resources* könyvtáraiban található fájlok tartalmát jelenti.

Az alkalmazás a labor témájának megfelelően Java programnyelven íródott és a JavaFX (https://en.wikipedia.org/wiki/JavaFX) névre hallgató GUI (Graphical User Interface) keretrendszert használja a felület létrehozásához. A JavaFX filozófiájának megfelelően a példa alkalmazás felépítése, architektúrája a klasszikus Model-View-Controller rétegződést követi. Ez kiegészül egy adatelérési réteggel, a Model rétegben található osztályok példányait ez a logikai réteg fogja szolgáltatni. Az *src* könyvtár az alábbi Java package-ket tartalmazza:

- **application** – Az alkalmazás Controller rétegét és a megjelenítéshez szükséges segéodsztályokat tartalmazza**.**
    - **AppMain.java** – Az alkalmazás belépési pontját tartalmazó osztály forráskódja; létrehozza és inicializálja a nézetet
    - **Controller.java** - Az alkalmazás vezérlő rétegét megvalósító osztály forráskódja; példányosítja az adatelérési réteget, feldolgozza a megjelenítési rétegtől érkező kéréseket, melyek kiszolgálásához az adatelérési réteg metódusait hívja, majd az eredményeket átadja a megjelenítési rétegnek.
    - **ComboBoxItem.java –** Egy példa arra, hogy hogyan tudunk tárolni adatokat a ComboBoxban.
- **dal** – Ebben a packageben vannak az adatelérési réteghez tartozó forrásfájlok.
    - **DataAccessLayer.java** – generikus interfész az adatelérési réteg szolgáltatásainak kiajánlására.
    - **ActionResult.java** – Az adatmódosító feladat lehetséges eredményeit tartalmazó enum.
- **dal.impl** – Ennek egyetlen – feladattípustól függő – forrásfájlja tartalmazza az adatelérési osztály szkeletonját.
- **dal.exceptions** - Ebben a package-ben vannak definiálva az adatelérési rétegben használt kivételek.

- **model –** Tartalmaz három osztályt, ami szintén feladattípusonként különböző, ezekkel hordozunk adatot a megjelenítés és az adatelérési réteg között. Található benne továbbá egy Person osztály, aminek segítségével valósul meg a példa SQL-lekérdezés eredményeinek megjelenítése.

A *resources* könyvtár a minta alkalmazásban egyetlen fájlt tartalmaz, mely a **View.fxml** nevet viseli. Ez a fájl írja le XML (Extensible Markup Language) (https://en.wikipedia.org/wiki/XML) nyelven az alkalmazás felületét, a felületen megjelenő vezérlőelemeket és azok kapcsolatát. JavaFX-specifikus fájl.

A minta alkalmazásban az adatbázishoz történő kapcsolódáshoz szükséges felhasználónév-jelszó páros az alkalmazás ablakának felső részén található beviteli mezőkben adható meg, a *Connect* gombra kattintva pedig létrehozza a kapcsolatot az adatbázissal. A kapcsolat státusza a *Connect* gomb mellett jelenik meg.

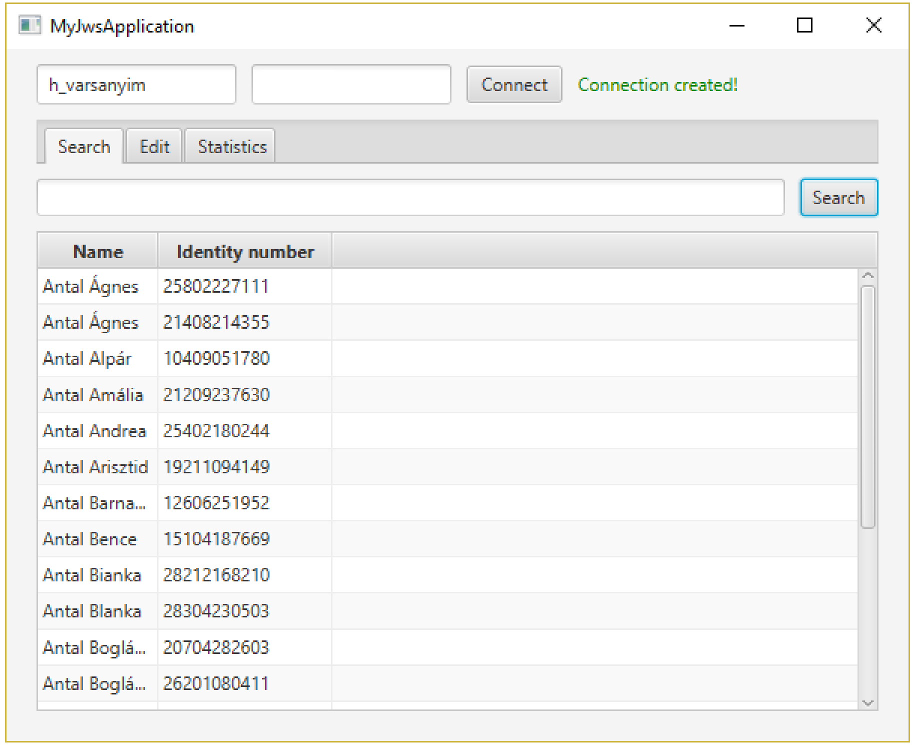
Az alkalmazás három fület tartalmaz, melyek rendre a következő nevet viselik: *Search*, *Edit* és *Statistics*. A fülek és a rajtuk elhelyezett vezérlőelemek szerepe kettős: egyrészről mintaként szolgálnak hasonló felületelemek létrehozásához, másrészt a labor során létrehozandó alkalmazás felületén ezekre az elemekre lesz szükség. A minta alkalmazás így részben meghatározza a gyakorlat során implementálandó alkalmazás felületét, másrészről segítséget is nyújt, hiszen így nagyobb figyelmet szentelhetünk a mérés lényegét jelentő adatbáziskezelés témakörének.

A minta alkalmazás *Search* névre hallgató fülén egy szövegbeviteli mező, egy gomb és egy táblázatelem kapott helyet. A *Search* gomb megnyomására a kezdetben üres táblázatba a már korábban említett lekérdezéshez tartozó rekordok jelennek meg.

Az *Edit* fülön példákat mutatunk címke (label), legördülő menü (dropdown list), szövegbeviteli mező és gomb elhelyezésére a felületen. A legördülő menüben két érték választható: ’Value A’ vagy ’Value B’.

Végül a *Statistics* fülön ismét egy gomb, illetve egy táblázat található, mely az utolsó mérési feladat megoldását hivatott segíteni.

Az alkalmazás felülete látható az alábbi ábrán sikeres csatlakozást követően:



Az alkalmazás forráskódjának tanulmányozása során első lépésként nézzük meg az alkalmazás belépési pontját – a *main* függvényt – is tartalmazó *AppMain.java* fájl tartalmát. A kód a szükséges JavaFX importokat követően az alkalmazás osztály létrehozásával kezdődik. JavaFX esetében ablakos alkalmazásunkat az *Application* osztályból származtathatjuk. Ahogy a kódból is látható, a főosztály neve *AppMain* lesz.

A fájl végefelé található a *main* függvény, mely paraméterként az esetleges parancssori argumentumokat kaphatja. A függvény törzse egyetlen sort tartalmaz, mely példányosítja és elindítja alkalmazásunkat.

Az alkalmazás indulása során elvégzendő tevékenységeket a *start* nevű metódusban kell implementálni, míg a *stop* metódus az alkalmazás leállítása előtt hajtódik végre. A *start* függvény elején az *FXMLLoader* objektum segítségével betöltjük az alkalmazás felületét leíró XML fájlt, nevezetesen a *resources* könyvtárban található *View.fxml* állományt. Ekkor létrejön az alkalmazás felületét képező vezérlőelemek hierarchiája (az XML fájl alapján), melynek gyökérelemére referenciát is kapunk a *viewRoot* változóval.

A JavaFX alkalmazások felületének létrehozásánál a legfelsőbb szintű konténert a *Stage* objektum jelenti. Ezen objektum gyerekeként kell beállítani azt a *Scene* objektumot, ami az ablakban ténylegesen megjelenő elemek konténere. Végül a *Scene* objektum gyerekeként a korábbi *viewRoot* objektumot adhatjuk meg, mely az XML fájlban leírt legfelső szintű felületelem.

A fenti beállítások elvégzését követően a *setTitle* metódushívással megadjuk az ablak nevét, végül megjelenítjük azt. Az előbbi lépések láthatók az alábbi kódrészletben:

```
// Create a loader object and load View and Controller
final FXMLLoader loader = new
FXMLLoader(getClass().getClassLoader().getResource("resources/View.fxml"));
final VBox viewRoot = (VBox) loader.load();
```

```
// Get controller object and initialize it
controller = loader.getController();

// Set scene (and the title of the window) and display it
Scene scene = new Scene(viewRoot);
primaryStage.setScene(scene);
primaryStage.setTitle("MyJwsApplication");
primaryStage.show();
```

A fenti kódrészletben még az is látható, hogy elkérjük az FXML fájl alapján létrehozott felület vezérlő osztályának referenciáját, mely a nézetet hivatott kezelni.

Következő lépésként vizsgáljuk meg az alkalmazás felületét, annak elrendezését, megjelenését leíró FXML fájlt. Az XML fájlok, így esetünkben az FXML fájl is (*View.fxml*) a HTML fájlokhoz hasonlóan ún. tag-ekből épülnek fel. A tageket csúcsos zárójelek közé írjuk, mint azt láthatjuk az alábbi példa esetében is:

```
<VBox fx:controller="application.Controller"
xmlns:fx="http://javafx.com/fxml/1"
      fx:id="rootLayout" alignment="CENTER" spacing="10" prefWidth="600"
      prefHeight="460" minWidth="600" minHeight="460">
```

FXML fájlokban ezek a tagek egy-egy felületelemet írnak le, mint pl. egy gombot, címkét, szövegbeviteli mezőt, vagy egy elrendezést (layout). A tagek (valójában párok) rendelkeznek egy nyitó és egy záró taggel.

Valamennyi tag egyedi névvel rendelkezik, melyet a ’<’ jel után írhatunk. A fenti példa esetében ez a *VBox*-nak felel meg. Ez jelenti az adott felületelem megnevezését. Ezt követően soroljuk fel a tag ún. attribútumait, vagyis azokat a paramétereket, melyek a felületelem egyes beállításainak értékeit adják meg. Valamennyi beállítást egy-egy attribútumnév jelöl, mint pl. a *prefHeight*, *prefWidth* attribútumok a fenti kódrészletben, melyek jelen esetben a *VBox* elem preferált magasságát és szélességét határozzák meg. Az attribútumokhoz értéket *név=érték* formában rendelhetünk, ahogy a példában is látható. Az értékek minden esetben idézőjelek közé írandók.

Az importokon és az alkalmazás elején található speciális *xml* tagen kívül a tageknek van egy záró párjuk is. Egy taget a *</tag_neve>* formában zárhatunk le. Egy tag nyitó és záró tagjén belül további taget/tageket helyezhetünk el, melynek jelentése, hogy a tag által reprezentált felületelembe további felületetelemet ágyazunk. Ez elsősorban a konténer típusú elemek esetén lényeges, de természetesen vannak más esetek is. Amennyiben egy felületelembe nem ágyazunk további elemeket, mint pl. egy gomb esetén, úgy logikusan a nyitó és záró tagek közötti rész üresen marad. A jobb áttekinthetőség érdekében ezt rövidíthetjük úgy, hogy a nyitó és záró taget összevonjuk a következő formában: *<tag_neve esetleges_attribútumok />*.

Az XML fájlok – és így az FXML fájlunk is – kötelezően egy *xml* taggel kezdődik, mely esetünkben az alábbi:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Ebben megadjuk az XML verzióját és a kódolást, mely UTF-8 lesz. Ezt követően szükséges importálni (természetesen az XML formátumnak megfelelően) a felület leírása során

felhasznált elemek osztályát. Erre mutat példát a következő sor, melyben a gomb felületelem (*Button*) osztályát importáljuk:

```
<?import javafx.scene.control.Button?>
```

A felület részeként számos elem megadható, ezek tárházából csak kisebb ízelítőt próbál nyújtani a minta alkalmazás. Az FXML fájlban láthatunk példát gombok, címkék, szövegmezők, fülek, legördülő menü, táblázat stb. elhelyezésére, illetve kapunk példát elrendezések, ún. layout-ok használatára is.

Ilyen layout a korábbi példában szereplő *VBox* felületelem is. Ez egy olyan konténer, mely egymáshoz képest vertikálisan helyezi el a belé ágyazott elemeket. A *VBox*-os példa attribútumaiból továbbá az is látszik, hogy a layout-ban elhelyezett elemek egymástól fixen 10 pixel távolságra kerülnek (lsd.: *spacing* attribútum értéke), valamint, hogy a beágyazott elemek középre igazítva jelennek meg (*alignment* attribútum). Érdemes megnéznünk az FXML fájlban azt is, hogy amennyiben szeretnénk valamekkora méretű üres területet beállítani a konténer elem kerete és a tartalom között, úgy azt a *padding* tag segítségével tehetjük meg, ahol a *padding* tagbe ágyazott *Insets* nevű tag attribútumaiban kell megadnunk a keret (border) alsó, felső, jobb és a bal oldali részénél beállítandó szabad terület mértékét.

A *VBox*-hoz hasonló a *HBox* névre hallgató layout, csak míg előbbi esetben vertikálisan, addig utóbbi esetben horizontálisan kerülnek elhelyezésre a layout-ba ágyazott elemek.

Fontos kiemelni négy speciális attribútumra, melyek közül háromra a *VBox* esetén példa is látható. Az első az *fx:controller* attribútum, mely megadja a felület kezelését, az eseménykezelést ellátó osztály nevét. Ahogy a példában látható, ezt a szerepet – a korábban írtakkal összhangban – a *Controller* nevű osztály tölti be. Ezt az attribútumot egyetlen alkalommal lehet csak megadni az XML alapú felület gyökérelemének tagjében (a mi esetünken a legkülső *VBox* tagben). Az attribútum értékeként azért írtunk *application.Controller*t és nem simán *Controller*t, mert a minta alkalmazás valamennyi osztálya egy *application* névre hallgató csomagon (package) belül található. (Ennek a package-nek a definiálása megtalálható valamennyi Java fájl elején.)

Az *xmlns:fx* attribútum azt a névteret definiálja, melyből a használt tagek nevei származnak. A névtér jelen esetben a Java *fxml* névtér, melynek pontos hivatkozása a következő: *http://javafx.com/fxml/1*. Névteret is csak egy alkalommal adunk meg, a gyökérelem tagjében.

Az *fx:id* attribútum a felületelem egyedi azonosítóját adja meg. Ez az azonosító azért különösen fontos a számunkra, mert ezen az ID-n keresztül tudunk majd a felület kezelését ellátó osztályból hivatkozni a felületelemre.

Végezetül amennyiben eseményeket is szeretnénk kezelni, úgy valamilyen módon eseménykezelőket (eseménykezelő metódusokat) is kell rendelnünk az egyes vezérlőelemekhez. Ennek módja az, hogy a vezérlőelem (pl.: *Button*) *onAction* nevű attribútumában megadjuk az eseménykezelő metódus nevét egy '#' karakterrel az elején. Erre láthatunk példát alább:

```
<!-- Search button -->
<Button fx:id="searchButton" text="Search" onAction="#searchEventHandler"
/>
```

A minta alapján a *searchButton* azonosítójú gomb megnyomásának hatására a nézetet kezelő osztályban található *searchEventHandler* metódus hívódik meg, mely lekezeli az eseményt. A példából az is látszik továbbá, hogy megjegyzéseket is elhelyezhetünk az XML kódban, melyeket a <!-- nyitó és --> záró részek közé kell írnunk.

Az XML fájl leírásának végén megjegyezzük, hogy az XML nyelvvel a hallgatók az 5. (XSQL című) mérés során találkozhatnak újra, melynek keretében mélyrehatóbban ismerkedhetnek azzal.

Leírásunk folytatásaként térjünk át a nézet kezelését megvalósító *Controller* osztályra, mely a *Controller.java* fájlban található. Ennek az osztálynak a konstruktorában példányosítjuk az adatelérési réteget, a következőképpen (a példában a VIDEO feladatsorhoz tartozó osztály szerepel):

```
public Controller() {
	dal = new VideoDal();

}
```

A továbbiakban a *Controller* osztály ennek az adatelérési rétegnek a segítségével fog kapcsolódni az adatbázishoz, és a lekérdező-, illetve adatmódosító műveleteket elvégezni.

Az osztályon belül *@FXML* dekorátorral kell ellátnunk azokat a metódusokat, melyeket hivatkozunk az FXML fájlból. Ilyenek lesznek az eseménykezelő metódusok, mint pl. a *Connect* vagy a *Search* gomb eseménykezelő metódusai. Eseménykezelő metódusra mutat példát az alábbi kódrészlet:

```
@FXML
public void searchEventHandler() {
	//TODO: replace this query with your solution.
	try {
		List<Person> people = dal.sampleQuery();
		searchTable.setItems(FXCollections.observableArrayList(people));
	} catch (NotConnectedException e) {
		e.printStackTrace();
	}

}
```

Az eseménykezelő metódusokban általában el is akarunk érni adott felületelemet, melyen módosítást hajtunk végre (pl. módosítjuk egy szövegbeviteli mező tartalmát). Ehhez az szükségeltetik, hogy rendelkezzünk valamilyen referenciával az érintett felületelemre. A felületen megjelenő egyes vezérlőelemekre, layout-okra úgy tudunk hivatkozni Java kódból, hogy a kód elején létrehozunk a felületelemek ID-jával azonos nevű objektumokat, melyek típusa (osztálya) megegyezik a felületelemek típusával (nevével). Ezen kívül az objektum deklarálása felett el kell helyeznünk a már megimert *@FXML* dekorátort. Ehhez példaként szolgál a következő kódsor:

```
@FXML
private TextField usernameField;
```

A mintában a *usernameField* ID-jú szövegmező elemhez hozunk létre objektumot *TextField* osztály megadása mellett. Természetesen a szükséges JavaFX osztályokat (mint pl. a *TextField* is) importálnunk kell a kód elején.

Fontos még kiemelnünk az *initialize* metódust, mely automatikusan meghívódik a felület felépítését követően, így a felületelemek inicializálását (pl.: szövegmezők törlését) ezen függvényen belül muszáj elvégeznünk. Ennek az az oka, hogy az osztály példányosításakor a felületelemek még NULL értékűek. Az *initalize* metódus lefutásakor azonban már biztosak lehetünk abban, a keretrendszer már hozzákötötte a megfelelő vezérlőt a tagváltozóhoz. Ebben a metódusban érdemes feltöltenünk a ComboBoxokat a megfelelő értékekkel, illetve a táblázatok oszlopaiban az adatkötést elvégeznünk:

```
@Override
public void initialize(URL location, ResourceBundle resources) {
	// TODO: initalize property-value factories
	//EXAMPLE: sampleCombo stores two strings.
	sampleCombo.getItems().add(new ComboBoxItem<String>("Value A", "a"));
	sampleCombo.getItems().add(new ComboBoxItem<String>("Value B", "b"));

	//EXAMPLE: this is how we bind the private variables to a data cell
	nameColumn.setCellValueFactory(new PropertyValueFactory<>("name"));
	identityNumberColumn.setCellValueFactory(
		new PropertyValueFactory<>("identityNumber"));

}
```

Az adatkötés során az történik valójában, hogy megadjuk az adott sorban lévő objektummelyik tagváltozójából olvassa ki a keretrendszer a megjelenítendő értéket. Ez a név esetében *name* tagváltozó, a személyi igazolványszám esetében pedig az *identityNumber*.

Végezetül az adatelérési réteg (**Dal.java, pl: VideoDal.java*) a fejezet elején leírtaknak megfelelően tartalmazza az adatbáziskezeléssel kapcsolatos logikát. Ebben az osztályban történik az adatbázishoz való csatlakozás, illetve a példaként bemutatott lekérdezés megvalósítása. Az adatbázishoz való csatlakozás előtt betöltjük az Oracle JDBC meghajtóját. Ez látható a következő kódrészletben:

```
// Load the specified database driver
Class.forName(driverName);
```

Az adatbázisból adatok lekérdezésére egy példát a *sampleQuery* metódusban láthatunk:

```
@Override
public List<Person> sampleQuery() throws NotConnectedException {
	checkConnected();
	List<Person> result = new ArrayList<>();
	try (Statement stmt = connection.createStatement()) {
		try (ResultSet rset = stmt.executeQuery(
					"SELECT nev, szemelyi_szam FROM OKTATAS.SZEMELYEK "
					+ "ORDER BY NEV "
					+ "OFFSET 0 ROWS FETCH NEXT 20 ROWS ONLY")) {
			while (rset.next()) {
				Person p = new Person(rset.getString("nev"),
							rset.getString("szemelyi_szam"));
				result.add(p);
			}
			return result;
		}
	} catch (SQLException e) {
		e.printStackTrace();
		return null;
```

```
    }
}
```

Az első sorban ellenőrizzük, hogy kapcsolódva vagyunk-e már az adatbázishoz, egy privát metódus segítségével. Létrehozunk egy *Statement* típusú objektumot, majd lefuttatjuk a lekérdezést. Ennek rekordjain végigiterálunk, és mindegyikből létrehozunk egy *Person* objektumot, amit az eredményként visszaadott listába tesszük bele. A labor során ajánlott – a mintakódnak megfelelően – a Java 8-as try-with-resources szerkezet használata. Lásd bővebben: https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html. Az adatbázistól érkező hibákat a kivételkezelő ágakban SQLException-ök formájában kaphatjuk el.

A model packageben lévő osztályok egyszerű POJO (Plain old Java Object) osztályok, a privát tagváltozókat getter-setter metódusokkal érhetjük el.

A fentiekben vázolt JavaFX-es alkalmazás WebStart-os „keretet" kap azáltal, hogy létrehozunk egy WebStart-ot vezérlő konfigurációs JNLP állományt:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="http://rapid.eik.bme.hu/~xxxxxx/jdbc"
href="application.jnlp">
      <information>
            <title>My Java Webstart JDBC Application</title>
            <vendor>Test student</vendor>
            <icon href="logo.png" kind="default" />
      </information>
      <security>
            <all-permissions />
      </security>
      <resources arch="" os="">
            <j2se version="1.7+" />
            <jar href="ojdbc7.jar" />
            <jar href="MySignedApplication.jar" main="true" />
      </resources>
      <application-desc />
</jnlp>
```

A WebStart-ot beágyazó HTML oldal részlete:

```
<a href="application.jnlp">
      <font size="4">My Java Webstart JDBC Application</font><br />
      <img src="logo.png" alt="My Java Webstart JDBC Application" />
</a>
```

6. Felhasznált irodalom

1. *The JDBC API Verson 1.20*, Sun Microsystems Inc.
2. S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner: *JDBC 2.0 API Tutorial and Reference, Second Edition: Universal Data Access for the Java 2 Platform*
3. S. Kahn: *Accessing Oracle from Java*, Oracle Co.
4. Nyékiné et al. (szerk.)*: Java 1.1 útikalauz programozóknak*, ELTE TTK Hallgatói Alapítvány

7. Függelék: Oracle adattípusok elérése JDBC-ből

Java típus	Lekérdező metódus	CHAR	VARCHAR2	NUMBER	DATE
byte	getByte	●	●	●	○
short	getShort	●	●	●	○
int	getInt	●	●	●	○
long	getLong	●	●	●	○
float	getFloat	●	●	●	○
double	getDouble	●	●	●	○
java.Math.BigDecimal	getBigDecimal	●	●	●	○
boolean	getBoolean	●	●	●	○
String	getString	●	●	●	●
java.sql.Date	getDate	○	○	○	●
java.sql.Time	getTime	○	○	○	●
java.sql.TimeStamp	getTimeStamp	○	○	○	●

Jelölések:

○: a getXXX metódus *nem használható* az adott SQL típus elérésére

●: a getXXX metódus *használható* az adott SQL típus elérésére

●: a getXXX metódus *ajánlott* az adott SQL típus elérésére

I. Függelék: UNIX összefoglaló a legfontosabb parancsokról

1. Könyvtárstruktúra

- mkdir <dirname> létrehoz egy könyvtárat.
- rmdir <dirname> törli a könyvtárat.
- cd <dirname> belép a könyvtárba.
- ls <dirname> kilistázza a könyvtár tartalmát. Az ls -l paranccsal részletes listát kapunk. Néhány speciális könyvtár:
    - . aktuális könyvtár,
    - .. szülő könyvtár,
    - / gyökérkönyvtár,
    - ~ a $(HOME) könyvtár – a felhasználó munkakönyvtára.

  Több cd parancs egymás utáni kiadása helyett cd <dirname>/<dirname>/[stb] alakú parancs is kiadható.
- cp <source> <dest> fájlt (fájlokat) másol.
- rm <filename> fájlt töröl.
- mv <source> <dest> fájlt mozgat.
- cat <filename> fájl tartalmának megjelenítése.

2. Jogosultságok

UNIXban a fájlokra három szinten háromfajta jogosultságot lehet beállítani. A szintek:

- owner – az állomány gazdája (u)
- group – az állomány csoportja (g)
- other – a csoporton kívül mindenki más (o)

A fajták (könyvtárra vonatkozó jog zárójelben):

- **r**ead – olvasási (listázási) jog
- **w**rite – írási jog
- e**x**ecute – futtatási (belépési) jog

A gazda és csoport a chown <owner>.<group> <filename> paranccsal állítható, A jogosultságok állítására a chmod <szint(ek)>(+|-)<fajta('k)> <filename> parancs szolgál. Például a foo.php fájlra olvasási jog adása mindenki más számára: chmod o+r foo.php

3. Fájlkezelő

- `mc` Midnight Commander. Hasonló, mint az nc/FAR DOS/Windows alatt. Támogatja többek között az ftp átvitelt, könyvtárak ki-/betömörítését.

4. Szövegszerkesztők

- `pico` szövegeket jó benne szerkeszteni, forrást nem annyira, mert automatikusan sort tör.
- `joe` jó forrás szerkesztésre, beállítható, hogy ne törjön sort, hozzá kell szokni a ^K+H típusú parancsokhoz, vagyis [CTRL]+[K] után egy [H] megnyomása (jelen esetben az a help parancs).
- `mcedit` az mc szövegszerkesztője. Ncurses alapon syntax highlighthingot, azaz kódszínezést biztosít.
- `vi` ősi unixos szövegszerkesztő. Kezdőknek nem ajánlott.

5. Grafikus szövegszerkesztők

- `emacs` sokan dicsérik, mert sokat tud.
- `nedit` kevesebbet tud, de kezdőknek egyszerűbb megtanulni.

6. Egyebek

- `gzip, gunzip` tömörítő/kitömörítő program. Csak egy fájlt kezel, ezért ha több fájlt kell egybe zippelni, akkor azokat előtte egybe kell csomagolni.
- `tar` csomagolóprogram. Becsomagolás: `-cvf <tarfile> <source>`, kicsomagolás: `-xvf <tarfile> <dest>`.
- csomagolás és tömörítés egyben: `tar -zcvf <tgzfile> <sources>`, visszafelé: `tar -zxvf <tgzfile> <dest>`
- `ssh` biztonságos távoli shell. Pl: `ssh -X xy123@ural2.hszk.bme.hu`
- `scp <source> <dest>` biztonságos fájlátvitel.
  Pl: `scp xy123@rapid.eik.bme.hu:php.tgz ./`
- `ping <host>` hostokat teszteli, hogy elérhetőek-e.

A parancsokról bővebb információ a `man <command>` begépelésével tudható meg.

III. Függelék: Reguláris kifejezések

Szerző: Soproni Péter

1. BEVEZETÉS 20
2. ALAPSZABÁLY 20
3. KARAKTEROSZTÁLYOK 21
   3.1. *Egyszerű karakterosztály* 21
   3.2. *Kizárás* 21
   3.3. *Intervallum karakterosztály* 21
   3.4. *Karakterosztályok uniója* 22
   3.5. *Karakterosztályok metszete* 22
4. ELŐRE DEFINIÁLT KARAKTEROSZTÁLYOK 22
5. KARAKTERCSOPORTOK 23
6. CSOPORTOK ISMÉTLÉSE 23
   6.1. *Mohó kiértékelés* 24
   6.2. *Lusta kiértékelés* 24
7. TOVÁBBI HASZNOS FUNKCIÓK 24
8. REGULÁRIS KIFEJEZÉS ALAPÚ DOS TÁMADÁS 25
9. JAVA ÁLTAL BIZTOSÍTOTT REGULÁRIS KIFEJEZÉS API 25
   9.1. *Java.util.regex.Pattern* 25
   9.2. *Java.util.regex.Matcher* 26
10. MELLÉKLET 26
   10.1. *Fordítás, futtatás* 26
   10.2. *Használat* 27
   10.3. *Alkalmazás kódja* 27

1. Bevezetés

Szinte a számítástechnikával egyidős az igény egy olyan nyelv kialakítására, amely egyszerű szűréseket, átalakításokat, ellenőrzéseket tesz lehetővé alapvetően szöveges adatokon, deklaratív szemlélet mellett. Az idők során több nyelv és ezen nyelvek több nyelvjárása alakult ki a különböző szabványosítási kísérletek ellenére is [R1-R2].

Jelenleg a legelterjedtebb a Perl nyelv által bevezetett szintaxis [R2]. Ennek kisebb-nagyobb mértékben átdolgozott változatait szinte az összes modern programozási nyelv támogatja (.NET [R3], Java [R4], PHP [R5]). A továbbiakban mi is ennek a nyelvnek az alapjait ismertetjük, illetve a hozzá Java-ban biztosított API használatába nyújtunk betekintést.

2. Alapszabály

Reguláris kifejezések írása során a cél egy minta megfogalmazása. Az adott mintára illeszkedő vagy éppen nem illeszkedő szövegrészeket keressük. Minden a mintában szereplő karakter, ha nem speciális, egy vele azonos karakter létét írja elő az illeszkedésekben.

A legfontosabb speciális karakterek: *[]().\^+?*{},$-*. Ha ezekre, mint nem speciális karakterre kívánunk illeszkedést biztosítani, úgy eléjük visszaperjelet (backslash) írni, azaz például a '\.' illeszkedik a pontra, a '\\' pedig a visszaperjelre.

Ennek megfelelően, ha egy ismert keltezésére illeszkedő szakaszokat keresünk (azaz például egy olyan szövegrészt, ami egy ismert kibocsátási hellyel kezdődik, és egy dátummezővel záródik), azt a következő reguláris kifejezéssel tehetjük meg:

Reguláris kifejezés:[8] 'Pécs, 1978\. december 07\.'	
Illeszkedésre példa	**Nem illeszkedő példa**
Pécs, 1978. december 07.	Bécs, 1978. december 07.
	Pécs,1978. december 07.
	Pécs,1978, december 07.

Látható, hogy a pontra való illeszkedés érdekében, mivel az egy speciális karakter, egy visszaperjelet kellett elé írni. További fontos észrevétel, hogy itt a szóköznek is jelentése van. A többi karakterhez hasonlóan egy vele azonos írásjel meglétét követeli az illeszkedés fennállásához (ezért nem illeszkedik a második ellenpélda).

3. Karakterosztályok

A legtöbb esetben a keresett szövegrészben egy adott helyen nem egyetlen karakter, hanem karakterek egy halmaza állhat. A reguláris kifejezésekben ennek megfogalmazására létrehozott konstrukció az úgynevezett karakterosztály.
Általános szintaxis: *[<karakterosztály elemei>]*
Az előző példánál maradva, általában nem egy konkrét keltezést keresünk, hanem több lehetőségből egyet. Ha mind Pécset, mind Bécset el tudjuk fogadni, mint helyszínt, akkor kifejezésünk a következőképpen alakul:

Reguláris kifejezés: '[BP]écs, 1978\. december 07\.'	
Illeszkedésre példa	**Nem illeszkedő példa**
Pécs, 1978. december 07.	Budapest, 1978. december 07.
Bécs, 1978. december 07.	Mécs, 1978. december 07.

3.1. Egyszerű karakterosztály

A lehetséges karakterek egyszerű felsorolásával képezzük. Minden felsorolt karakterre illeszkedik, de semmilyen más elemre nem.
Szintaxis: *[<az elfogadott karakterek listája>]*

Reguláris kifejezés: '[PBb]'	
Illeszkedésre példa	**Nem illeszkedő példa**
b	p
P	PP

3.2. Kizárás

Lehetőség van arra, hogy azt fogalmazzuk meg, mit nem fogadunk el az adott helyen.
Szintaxis: *[^<el nem fogadott karakterek>]*

Reguláris kifejezés: '[^PBb]'	
Illeszkedésre példa	**Nem illeszkedő példa**
G	b
V	vv

3.3. Intervallum karakterosztály

Az egyes elfogadott karakterek helyett az elfogadott intervallumokat[9] is megadhatjuk. Ilyen intervallum lehet pl. a számok halmaza 1 és 7 között, vagy az összes kisbetű.
Szintaxis: *[<intervallum alsó széle>-<intervallum felső széle>]*

[8] A reguláris kifejezésekre mutatott példáknál a kifejezéseket határoló aposztrófok csak a kifejezés határait jelölik ki, nem képezik azok részét. A példáknál, ha külön nem jelezzük, a megadott szöveg teljes illeszkedését vizsgáljuk, nem azt, hogy van-e illeszkedő részminta vagy részszöveg.
[9] Az intervallum a két határoló karakter ASCII karakterkódja által meghatározott intervallumba eső karakterkóddal meghatározott karakterek halmaza.

Reguláris kifejezés: '[1-9]'	
Illeszkedésre példa	**Nem illeszkedő példa**
3	0
4	44

3.4. Karakterosztályok uniója

Az egyes karakterosztályok uniója alatt azokat a karaktereket értjük, amelyek valamely eredeti karakterosztályban benne vannak.
Szintaxis: *[<első karakterosztály>|<második karakterosztály>]* vagy *[<első karakterosztály><második karakterosztály>]*[10]

Reguláris kifejezés: '[[1-9]|[ab]]'	
Illeszkedésre példa	**Nem illeszkedő példa**
a	0
7	A

3.5. Karakterosztályok metszete

Az egyes karakterosztályok metszete azon karakterek halmaza amelyek mindkét karakter osztályban megtalálhatók.
Szintaxis: *[<első karakterosztály>&&<második karakterosztály>]*

Reguláris kifejezés: '[a-z&&[^c-f]]'	
Illeszkedésre példa	**Nem illeszkedő példa**
a	D
z	A

4. Előre definiált karakterosztályok

A gyakorlatban a gyakran használt karakterosztályok halmaza meglehetősen szűk (betűk, számok, stb.), ezeknek nagy része a nyelvben beépítve is megtalálható.
Beépített karakterosztályok[11]:

- . - bármilyen karakter
- *d* – decimális karakterek ([0-9])
- *D* – nem decimális karakterek ([^0-9])
- *s* – whitespace karakterek ([\n\t\r\f])
- *S* – nem whitespace karakterek ([^\s])
- *w* – latin ábécé szó karakterei ([a-zA-Z0-9_]), ékezetes betűk nem
- *W* – nem szó karakterek ([^\w])
- *p{Ll}* – bármilyen kisbetű, ideértve minden ékezetest is
- *p{Lu}* – bármilyen nagybetű, ideértve minden ékezetest is
- *p{L}* – bármilyen betű, ideértve minden ékezetest is

Ha a keltezésnél nem vagyunk biztosak az évben, de tudjuk, hogy 1000 utáni, decemberi dátumról van szó, akkor a kifejezésünk a következőképpen is írható:

[10] Unió képzésnél lehetőség van a karakterosztályok leírásának egyszerűsítésére. Azaz az egymásba ágyazott karakterosztályoknál a belső definiáló szögletes zárójelek elhagyhatók. Például a '[[a]|[d-e]]' írható '[ad-e]'-nek is. Ez kizárásra, illetve metszetre nem vonatkozik.
[11] Információk az UNICODE támogatásáról: http://www.regular-expressions.info/unicode.html

Reguláris kifejezés: '[BP]écs\, [1-9]\d\d\d\. december \d\d\.'	
Illeszkedésre példa	**Nem illeszkedő példa**
Pécs, 1978. december 07.	Pécs, 978. december 07.
Pécs, 1978. december 09.	Pécs, 1978. december 7.
Pécs, 2978. december 09.	Mécs, 2978. december 09.

5. Karaktercsoportok

Lehetőség van arra, hogy a mintán belül csoportokat hozzunk létre. A csoport olyan mintarész, melyet egy nyitó és az ahhoz tartozó csukó zárójel határol. A mintán belüli csoportok számozottak. A nullás az egész minta, az n-edik pedig az a csoport melynek nyitózárójele az n-edik nyitózárójel a minta elejéről nézve.

- Reguláris kifejezés: ([BP]écs)\, (([1-9]\d\d\d)\. december \d\d\.)
- Egyes csoportok:

0. csoport: ([BP]écs)\, (([1-9]\d\d\d)\. (december) (\d\d)\.)
1. csoport: ([BP]écs)
2. csoport: (([1-9]\d\d\d)\. december \d\d\.)
3. csoport: ([1-9]\d\d\d)

Az egyes csoportok a reguláris kifejezésen belül hivatkozhatók. Ilyenkor a hivatkozott csoportra illeszkedő szövegrésznek meg kell ismétlődnie a mintában a hivatkozás helyén is.
Szintaxis: *(<minta>)....\<minta azonosító>*

Reguláris kifejezés: '(\w)vs\.\1'	
Illeszkedésre példa	**Nem illeszkedő példa**
1vs.1	avs.b
3vs.3	2vs.1

6. Csoportok ismétlése

Az eddig ismertetettek lehetővé teszik a minta egyes pozícióiban elfogadható karakterek igen pontos leírását. Ellenben arra még nem adnak módszert, hogy a mintában előforduló típus ismétléseket hogyan kezeljük.
Ismétlés megadásának szintaxisa:

- *<karakterosztály vagy karakter csoport>{<n>}* - pontosan 'n'-szeres ismétlés a mintának
- *< karakterosztály vagy karakter csoport>{<n, m>}* - 'n' és 'm' közé esik az ismétlések száma ('n' és 'm' ismétlés megengedett)
- *< karakterosztály vagy karakter csoport>{<n, >}* - legalább 'n' ismétlés
- *< karakterosztály vagy karakter csoport>+* - ekvivalens a {1,}-gyel
- *< karakterosztály vagy karakter csoport>?* - ekvivalens a {0,1}-gyel
- *< karakterosztály vagy karakter csoport>** - ekvivalens a {0,}-gyel

A előző példában várhatóan bármely helységet elfogadjuk (a helységről tételezzük fel, hogy nagybetűvel kezdődik, amelyet legalább egy kisbetű követ). Mindezek alapján a mintánk következőképpen alakul:

Reguláris kifejezés: '\p{Lu}\p{Ll}+, [1-9]\d{3}\. \p{Ll}+ \d{2}\.'	
Illeszkedésre példa	**Nem illeszkedő példa**
Szeged, 1978. december 07.	Szeged, 978. december 07.
Budapest, 1978. december 07.	nappalodott, 978. december 07.

Ezzel azonban még nem definiáltuk, hogy ha ismétléseket tartalmazó minta többféleként is illeszkedhet, akkor melyiket szeretnénk használni. A két legfontosabb kiértékelési mód a mohó és a lusta.

6.1. Mohó kiértékelés

Ilyenkor az ismételhető minta, részminta a lehető legtöbb elemre próbál illeszkedni. Ha ez nem lehetséges, mert így az egész mintának nincs illeszkedése, akkor az utolsó lehetséges karaktert kivéve mindegyikre próbál illeszkedik. És így tovább. Ez a mintaillesztés alap viselkedése.

Reguláris kifejezés: '(.+)(\d{1,2})\.'		
Illesztett szöveg: Pécs, 1978. december 07.		
Illesztett szöveg (0. csoport)	**1. csoport**	**2.csoport**
Pécs, 1978. december 07.	Pécs, 1978. december 0	7

6.2. Lusta kiértékelés

A mohó fordítottja. Elsőként a lehető legkevesebb elemet felhasználva próbálja a részmintákat illeszteni. Csak ha ez nem lehetséges, akkor bővíti az illeszkedés hosszát.
Szintaxis: <ismételt csoport>?

Reguláris kifejezés: '(.+?)(\d{1,2})\.'		
Illesztett szöveg: Pécs, 1978. december 07.		
Illesztett szöveg (0. csoport)	**1. csoport**	**2.csoport**
Pécs, 1978. december 07.	Pécs, 1978. december	07

7. További hasznos funkciók

Bizonyos körülmények között több alapvetően eltérő mintát is el kell fogadnunk. A korábbi keltezéses példánál maradva elképzelhető, hogy a keltezés helye egyszerűen ki van pontozva. Az ilyen problémák megoldására javasolt a minták vagylagos összefűzése.
Szintaxis: *(<első minta>)|(<második minta>)*

<table>
<tr><td colspan="2">Reguláris kifejezés: '(\p{Lu}\p{Ll}+, [1-9][\d]{3}\. \p{Ll}+ [\d]{2}\.)|(\.{10,})'</td></tr>
<tr><th>Illeszkedésre példa</th><th>Nem illeszkedő példa</th></tr>
<tr><td>Szolnok, 1978. december 07.</td><td>reggeledett, 1978. december 07.</td></tr>
<tr><td>......................................</td><td>....</td></tr>
</table>

A keltezés tulajdonsága, függően a használt nyelvtől, hogy csak a sor elején vagy mindig egy sor lezárásaként szerepel. Erre szolgáló beépített módosítók:

- Sor elejére illeszkedik: ^
- Sor végére illeszkedik: *$*

<table>
<tr><td colspan="2">Reguláris kifejezés[12]: '^(\p{Lu}\p{Ll}+, [1-9][\d]{3}\. \p{Ll}+ [\d]{2}\.)|^(\.{10,})'</td></tr>
<tr><th>Illeszkedésre példa</th><th>Nem illeszkedő példa</th></tr>
<tr><td>Szolnok, 1978. december 07.</td><td>Kelt.: 978. december 07.</td></tr>
<tr><td>......................................</td><td>Kelt.: ....</td></tr>
</table>

A Perl reguláris kifejezéseknél lehetőség van tovább speciális opciók bekapcsolására. Ezen opciók közül a gyakorlatban a legfontosabb a kisbetű-nagybetű érzékenység kikapcsolása. Ez az opció annak a csoportnak a hátralevő részéig fejti ki hatását ahol definiálva lett.
Szintaxis: (?i)<reguláris kifejezés>

Reguláris kifejezés: '([A-Z](?i)[a-z]+e)[a-z]'	
Illeszkedésre példa	**Nem illeszkedő példa**
Losangeles	LOSANGELES
LosAngeles	losAngeles
LOSAngelEs	Losangeles

A Perl szintaxis lehetővé teszi kommentek elhelyezését a reguláris kifejezésben[13].
Szintaxis: (?#<komment szövege>)

[12] Itt a problémát, mint részminta keresést vizsgáljuk.
[13] A Java reguláris kifejezés motorja nem támogatja.

Reguláris kifejezés: 'júli(?#ez a reguláris kifejezés júliusra illeszkedik)us'	
Illeszkedésre példa	**Nem illeszkedő példa**
Július	júli(#ez a reguláris kifejezés júliusra illeszkedik)us'

8. Reguláris kifejezés alapú DoS támadás

A reguláris kifejezések, minden hasznuk mellett is, bizonyos körülmények között veszély forrásai lehetnek. Egyes esetekben más sebezhetőségek könnyebb kihasználhatóságát teszik lehetővé [R6], máskor nem megfelelő alkalmazásuk képezi a problémát [R7]. Jelen jegyzet szempontjából az utóbbi eset, annak is a weblapok bemeneti adatinak reguláris kifejezésekkel történő ellenőrzését kihasználó úgynevezett REDoS (R*egular Expression Denial of Service*) támadás bír kiemelt fontossággal.
A REDoS támadás azt az igen elterjedt és helyes gyakorlatot használja ki, hogy az egyes honlapok a bemeneti adataik formátumának ellenőrzésére reguláris kifejezéseket alkalmaznak – ilyen adat lehet például a felhasználó neve vagy éppen a születési éve.
Példaként vizsgáljuk meg a következő kifejezést, melynek célja a megjegyzések ellenőrzése lenne: ’^(\p{L}+\s?)+$’. A kifejezés azokra a szövegekre illeszkedik, amelyek egy vagy több, egymástól whitespace karakterrel elválasztott szóból állnak. Nézzük, hogy a következő szövegekre hányféleképpen illeszthető:

- ’a’: 1
- ’aaaa’: 16
- ’aaaaaaaaaaaaaaaa’: 65536

Látható, hogy a szöveg hosszával nem lineárisan, hanem exponenciálisan növekszik a lehetséges illeszkedések száma. A problémát elsősorban az eset képezi, amikor végül nem találunk illeszkedést, mint a ’aaaa-’. Itt a reguláris kifejezés értelmezőnek az ’aaaa’ mind a 16 lehetséges felosztását ki kell próbálnia mielőtt, kijelenthetné: nincs illeszkedés (elvileg a 16-ból bármelyiknél lehetne szerencséje, azaz illeszkedést lelhetne).
Az ilyen exponenciális robbanást produkáló reguláris kifejezések okozta sebezhetőség alkalmas lehet a kiszolgáló leterhelésére, azaz DoS támadás megvalósítására. Természetesen a támadást jelentősen megnehezíti, hogy minden reguláris kifejezéshez külön ki kell találni egy olyan bemenetet, amely mellett az exponenciális robbanás bekövetkezik.
Veszélyes reguláris kifejezések jellemző felépítése:

- Tartalmaz karaktercsoport ismétlést
- Az ismételt csoporton belül ismétlést (például: ’(a+)+’) vagy átfedő alternatívákat (például: ’(a|aa)+’)

9. Java által biztosított reguláris kifejezés API[14]

A Java 1.4-es változatába kerültek bele a Pattern, illetve a Matcher osztályok.

9.1. Java.util.regex.Pattern[15]

A Pattern osztály felelős a reguláris kifejezések feldolgozásáért, illetve az egyszerű mintailleszkedés ellenőrzésért.
Fontosabb metódusai:

- *public static boolean* ***matches(String reg, CharSequence input):*** Igaz, ha a reg kifejezés illeszkedik a bemenetként megadott teljes szövegre.

- ***public static Pattern compile(String regex)***: A regexp paraméterként kapott kifejezést lefordítja, az alapján készít egy Pattern objektumot, ami a továbbiakban illeszkedés vizsgálatra használható.

[14] Mivel a Java nyelvben a String definíciója során a '\' jel speciális karakter, ezért az előre definiált reguláris kifejezésekben '\\'-ként írandó. Hasonlóan, a dupla visszapert négy egymást követő visszaperjellel lehet leírni, ami az illeszkedésben egy visszaperjel meglétét fogja megkövetelni.
[15] http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html

- ***public String[] split(CharSequence input)***: A korábban megadott reguláris kifejezés illeszkedései közötti szövegrészeket adja vissza.
- ***public Matcher matcher(CharSequence input)***: Az inputra vonatkozó Matcher típusú objektummal tér vissza, ami lehetővé teszi a további, hatékony művelet végzést.

9.2. Java.util.regex.Matcher[16]

A Matcher feladata az egyes illeszkedések lekérdezése, manipulálása a Patternek átadott karaktersorozatban.
Fontosabb metódusai:

- *public int **end**()*: A jelenleg vizsgált illeszkedés utáni első karakter pozícióját adja vissza.
- *public int **end**(int group)*: A group által azonosított karaktercsoport utolsó illeszkedő elemére következő karaktert adja vissza, -1-t ha az adott csoport nem illeszkedett.
- *public String **group**()*: Az illeszkedés szövegét adja vissza.
- *public String **group**(int group)*: A group által jelölt karaktercsoport aktuális illeszkedő szövegét adja vissza.
- *public boolean **find**()*: Megpróbálja megkeresi a minta következő illeszkedését a szövegben. Igazat ad vissza, ha van még illeszkedés, egyébként hamisat.
- *public boolean **lookingAt**()*: Igaz, ha a reguláris kifejezés illeszthető a szövegre vagy annak egy részére.
- *public boolean **matches**()*: Igaz, ha a reguláris kifejezés illeszthető az egész szövegre.
- *public String **replaceAll**(String replacement)*: Lecseréli a reguláris kifejezés összes illeszkedését a replacment-ben megadott kifejezés alapján. A replacment tartalmazhat hivatkozásokat a reguláris kifejezés egyes karaktercsoportjaira.

Ajánlott irodalom:

A1. Jeffrey E. F. Friedl: Reguláris kifejezések mesterfokon

A2. Laura Lemay: Perl mesteri szinten 21 nap alatt

A3. http://java.sun.com/docs/books/tutorial/essential/regex/index.html

Referencia:

R1. http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html

R2. http://www.perl.com/doc/manual/html/pod/perlre.html

R3. http://msdn.microsoft.com/en-us/library/hs600312%28VS.71%29.aspx

R4. http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/package-summary.html

R5. http://hu.php.net/manual/en/book.pcre.php

R6. http://www.ihteam.net/papers/blind-sqli-regexp-attack.pdf

R7. http://msdn.microsoft.com/en-us/magazine/ff646973.aspx

10. Melléklet

A példák megértését, illetve a reguláris kifejezések írását megkönnyítendő, mellékletként egy Java alkalmazást teszünk közre, mely a reguláris kifejezések tesztelésére használható.

10.1. Fordítás, futtatás

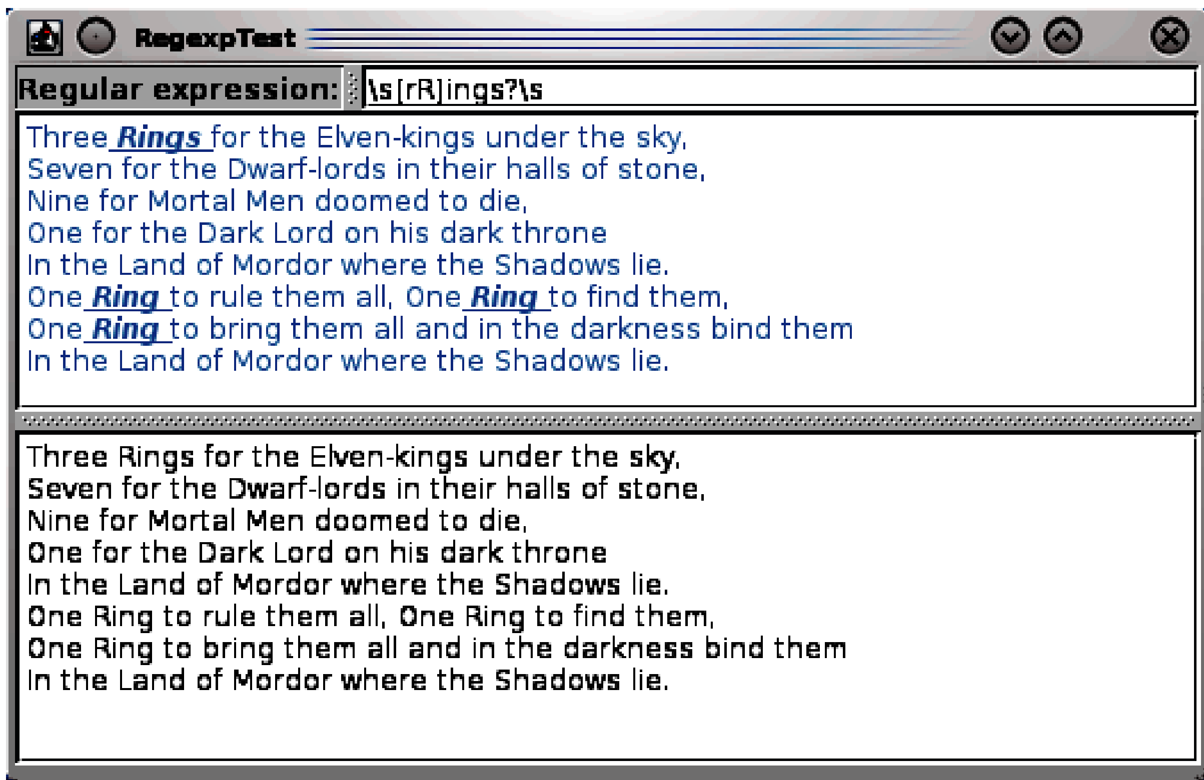
A teszt alkalmazás fordításának, futtatásának lépései:

[16] http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Matcher.html

1. Legalább 1.4-es Java JDK telepítése a számítógépre
2. A melléklet végén található kód kimásolása egy *RegexpTest.java* nevű fájlba
3. A program az előző pontban létrehozott könyvtárban a következő utasítással fordítható: *javac -g RegexpTest.java*
4. Futtatás: *java RegexpTest*

10.2. Használat

Tételezzük fel, hogy szeretnénk megkeresni a Gyűrűk Ura című könyvben megtalálható, a gyűrűk szétosztását leíró versben a 'ring' szó összes előfordulását.
A vizsgálat elvégzéséhez indítsuk el az alkalmazást. Az ablak jobb felső sarkában lévő szöveg mezőbe kell beírni azt a reguláris kifejezést, amelynek az illeszkedéseit keressük (az esetünkben: '\s[rR]ings?\s'). Az alsó mezőbe kerül a szöveg, melyben az illeszkedéseket vizsgáljuk. A szoftver az alsó mező tartalmát automatikusan a felsőbe másolja és ott az illeszkedő elemeket félkövér, dőlt, aláhúzott karakterrel jeleníti meg, lásd 1. ábra. Ezek alapján a 'ring' szó négyszer szerepel a versben.



1. ábra

10.3. Alkalmazás kódja

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;

import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JSplitPane;
import javax.swing.JLabel;
import javax.swing.JTextPane;
import javax.swing.text.SimpleAttributeSet;
```

```java
import javax.swing.text.StyleConstants;
import javax.swing.JScrollPane;

public class RegexpTest extends JFrame {

	private static final long serialVersionUID = 1L;

	// UI components
	private JPanel jContentPane = null;
	private JSplitPane jSplitPane = null;
	private JLabel regexpLabel = null;
	private JTextField regexpTextField = null;
	private JSplitPane splitPane = null;
	private JTextPane sourceTextPane = null;
	private JTextPane matchTextPane = null;
	private JScrollPane upperScrollPane = null;
	private JScrollPane lowerScrollPane = null;

	// Pattern matching objects
	private Pattern pattern = null;
	private Matcher matcher = null;
	private SimpleAttributeSet matchedTextAttributes = null;

	/**
	 * Creats the new pattern object from the input field's content if possible.
Otherwise
	 * pattern is set to null
	 */
	private void generatePattern()
	{
		try
		{
		pattern = Pattern.compile(regexpTextField.getText());
		}
		catch(PatternSyntaxException e)
		{
			pattern = null;
			matcher = null;
		}
	}

	/**
	 * Shows the matches of the pattern in the upper pane
	 */
	private void showMatches()
	{
		matchTextPane.setText(sourceTextPane.getText());
		if (pattern == null)
		{
			return;
		}

		matcher = pattern.matcher(sourceTextPane.getText());

	  while (matcher.find()) {
		matchTextPane.setSelectedTextColor(Color.WHITE);
		matchTextPane.setSelectionStart(matcher.start());
		matchTextPane.setSelectionEnd(matcher.end());
		matchTextPane.setSelectionColor(Color.WHITE);

	matchTextPane.getStyledDocument().setCharacterAttributes(matcher.start(),
				matcher.end() - matcher.start(), matchedTextAttributes,
true);
	  }
	}

	/**
	 * Generates the text style used to mark matches
```

```
	 */
	private void generateMatchedTextAttributes() {
		// Style of the matched text
	    matchedTextAttributes = new SimpleAttributeSet();

matchedTextAttributes.addAttribute(StyleConstants.CharacterConstants.Bold,
Boolean.TRUE);

matchedTextAttributes.addAttribute(StyleConstants.CharacterConstants.Italic,
Boolean.TRUE);

matchedTextAttributes.addAttribute(StyleConstants.CharacterConstants.Underline,
Boolean.TRUE);
	}

	public RegexpTest() {
		super();
		initialize();
		generateMatchedTextAttributes();
	}

	private void initialize() {
		this.setSize(400, 400);
		this.setContentPane(getJContentPane());
		this.setTitle("RegexpTest");
	}

	/**
	 * UI components
	 */

	private JPanel getJContentPane() {
		if (jContentPane == null) {
			jContentPane = new JPanel();
			jContentPane.setLayout(new BorderLayout());
			jContentPane.add(getJSplitPane(), BorderLayout.NORTH);
			jContentPane.add(getSplitPanel(), BorderLayout.CENTER);
		}
		return jContentPane;
	}

	private JSplitPane getJSplitPane() {
		if (jSplitPane == null) {
			regexpLabel = new JLabel();
			regexpLabel.setText("Regular expression:");

			jSplitPane = new JSplitPane();
			jSplitPane.setLeftComponent(regexpLabel);
			jSplitPane.setRightComponent(getRegexpTextField());
			jSplitPane.setEnabled(false);
		}
		return jSplitPane;
	}

	private JTextField getRegexpTextField() {
		if (regexpTextField == null) {
			regexpTextField = new JTextField();
			regexpTextField.setToolTipText("Regular expression");
			regexpTextField.addKeyListener(new java.awt.event.KeyAdapter()
{
				public void keyReleased(java.awt.event.KeyEvent e) {
					generatePattern();
					showMatches();
				}
			});
		}
		return regexpTextField;
	}
```

```
	private JSplitPane getSplitPanel() {
		if (splitPane == null) {
			splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
			splitPane.setTopComponent(getUpperScrollPane());
			splitPane.setBottomComponent(getLowerScrollPane1());

		}
		return splitPane;
	}


	private JTextPane getSourceTextPane() {
		if (sourceTextPane == null) {
			sourceTextPane = new JTextPane();
			sourceTextPane.setToolTipText("Test text");
			sourceTextPane.addKeyListener(new java.awt.event.KeyAdapter() {
				public void keyReleased(java.awt.event.KeyEvent e) {
					showMatches();
				}
			});
		}
		return sourceTextPane;
	}

	private JTextPane getMatchTextPane() {
		if (matchTextPane == null) {
			matchTextPane = new JTextPane();
			matchTextPane.setEnabled(false);
			matchTextPane.setDragEnabled(false);
			matchTextPane.setFocusable(false);
			matchTextPane.setToolTipText("Matched words are in italic, bold 
and underlined");
		}
		return matchTextPane;
	}

	private JScrollPane getUpperScrollPane() {
		if (upperScrollPane == null) {
			upperScrollPane = new JScrollPane();
			upperScrollPane.setViewportView(getMatchTextPane());
		}
		return upperScrollPane;
	}

	private JScrollPane getLowerScrollPane1() {
		if (lowerScrollPane == null) {
			lowerScrollPane = new JScrollPane();
			lowerScrollPane.setViewportView(getSourceTextPane());
		}
		return lowerScrollPane;
	}

	public static void main(String[] args)
	{
		RegexpTest mainclass = new RegexpTest();
		mainclass.setVisible(true);
	}
}
```