Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Adatbázis-kezelés

Sipos Róbert

siposr@hit.bme.hu

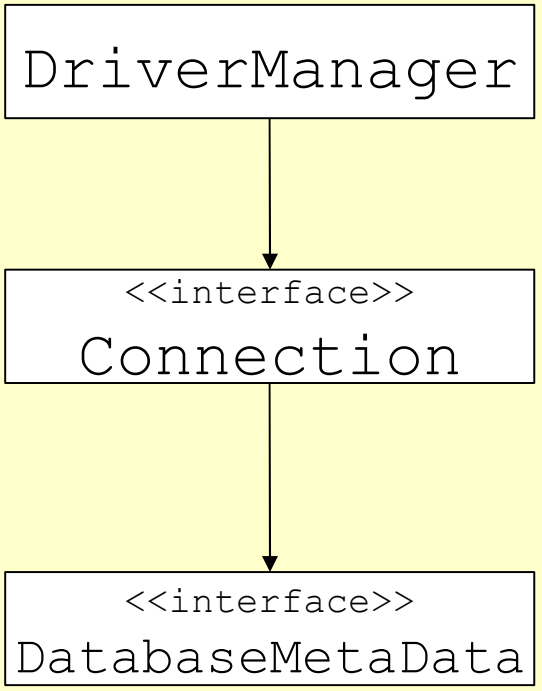
2014. 04. 29.

Bevezetés

- Számos alkalmazás tárolja (perzisztálja) az adatait **relációs adatbázisok**ban
- A Java platform adatbázis elérést támogató interfésze a **JDBC API** (Java DataBase Connectivity)
- **Gyártófüggetlen** adatbázis-kezelést támogató **interfész**, így konkrét adatbázis-rendszertől független program valósítható meg felhasználásával
- A Java program a JDBC segítségével teremt **alacsony-szintű kapcsolatot a relációs adatbázissal**, **SQL utasítások** segítségével végezhet műveleteket az adatbázison, és ezután a kapott eredményeket fel lehet dolgozni általa.

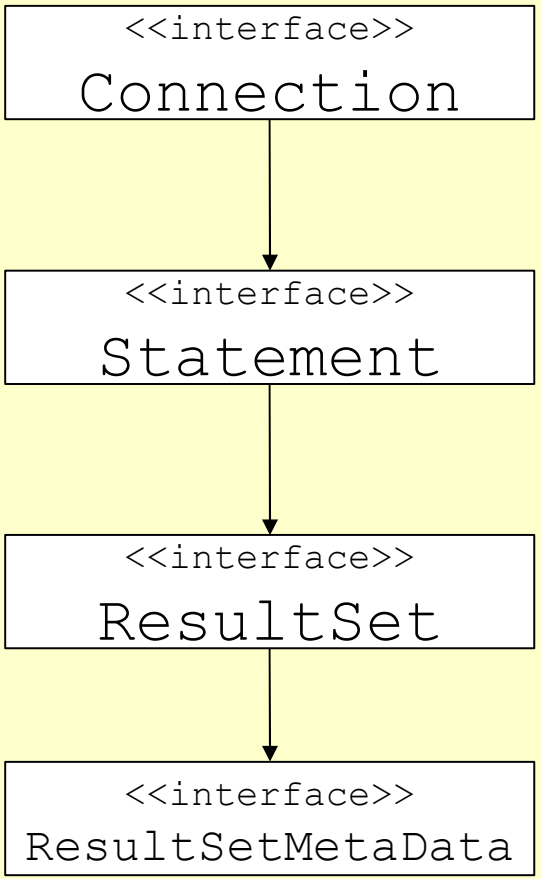
JDBC 1.2 API

- **DriverManager**: az adatbázis URL feloldása, új adatbázis kapcsolatok felépítése
  - ☞ Csak statikus metódusok, **factory** minta
- **Connection**: egy adatbázis kapcsolatot reprezentál
- **DatabaseMetaData**: adatbázis meta-információi (az adatbázis képességei (pl. tranzakciókezelés, adattípusok, tárolt eljárások...))



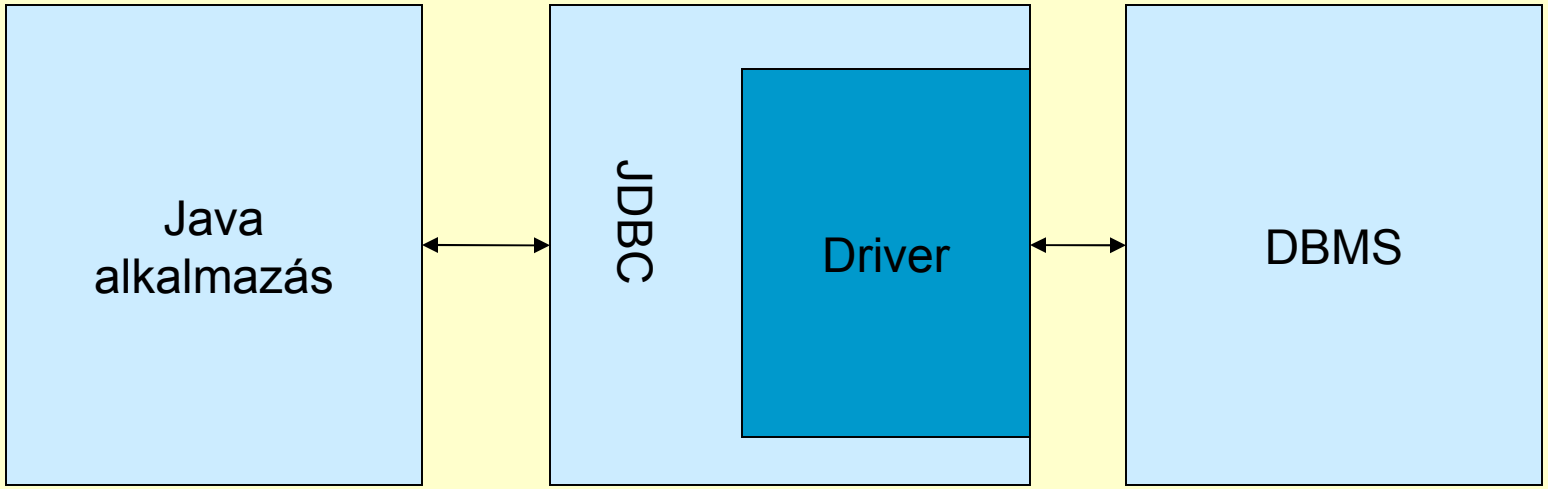
JDBC 1.2 API

- **Statement**: egy SQL utasítás végrehajtása
- **ResultSet**: egy lekérdezés eredménytáblája
- **ResultSetMetaData**: az eredménytábla meta-információi (pl. oszloptípus, pontosság)



Architektúra

- A JDBC API a **java.sql** és a **javax.sql** package-ekben található
- A JDBC-t támogató DBMS gyártóknak egy meghajtóprogramot kell megvalósítaniuk, ami az adatbázis-specifikus részeket valósítja meg



Meghajtóprogramok

- A JDBC az alábbi négyféle meghajtóprogramot definiálja
  - **JDBC-ODBC bridge**: összekötő program, ha egy adatbázishoz csak C nyelvű ODBC (Open DataBase Connectivity) driver létezik
    - A natív kód miatt hatékonyabb, de platformfüggő
  - **JDBC-adatbázis kliens API**: a JDBC hívásokat az adatbáziskezelő kliens API-jára fordítja le, hátránya, hogy csak a platformfüggő kliens programmal együtt képes működni
  - **JDBC-hálózati protokoll**: teljesen Java alapú meghajtóprogram a JDBC hívásokat adatbázisfüggetlen hálózati protokollon keresztül továbbítja egy közbenső szerverhez, melyen egy program értelmezi a parancsokat, és a konkrét DBMS API-jára fordítja le
  - **JDBC-DBMS API**: teljesen Java alapú meghajtóprogram a JDBC hívásokat közvetlenül a DBMS API-jára fordítja le
    - Teljesen platformfüggetlen

Kapcsolatfelépítés

1. a megfelelő meghajtóprogram betöltése (és szükség esetén regisztrálása)
2. kapcsolatobjektum létrehozása (és lezárása)

```
try {
    Class.forName("vendor.jdbc.DriverXYZ");
    // DriverManager.registerDriver(new vendor.jdbc.DriverXYZ());
    Connection con =
     DriverManager.getConnection("jdbc:xyz://localhost/datadase",
     "username", "password");
    // …
    con.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

SQL utasítások végrehajtása, hibakezelés

SQL utasítások végrehajtása

- **Statement** példányosítása a **Connection** példány `createStatement()` metódusával
- Utasítás végrehajtása az adott SQL utasítás átadásával
  - ☞ `executeQuery` metódus: az eredménytáblával (**ResultSet**) tér vissza
    - ➢Lekérdező (SELECT) utasításokhoz
  - ☞ `executeUpdate`: a módosított sorok számával tér vissza
    - ➢Adatmanipulációs (DML) és adatdefiníciós (DDL) utasításokhoz
  - ☞ `execute`: általánosítás, a visszatérési érték `true`, ha van eredménytábla (`getResultSet()`), egyébként `getUpdateCount()`
- Utasítás lezárása a `close()` metódussal (ha az eredménytábla minden sorát kiolvastuk, vagy újra meghívtunk egy végrehajtó függvényt, akkor automatikusan is lezáródik)

SQL adatmódosító műveletek végrehajtása

1. Statement objektum létrehozása
2. SQL utasítás átadása
3. executeUpdate a módosított sorok számát adja vissza

```
try {
    Statement stmt = con.createStatement();
    stmt.executeUpdate("INSERT INTO names VALUES ('Kovács','Attila')");
    stmt.executeUpdate("INSERT INTO names VALUES ('Nagy','László')");
    int n;
    n=stmt.executeUpdate("DELETE FROM names WHERE keresztnev='Nagy'");
} catch (Exception e) {
    e.printStackTrace();
}
```

SQL lekérdező műveletek végrehajtása 1.

1. Statement objektum létrehozása
2. SQL utasítás átadása
3. eredmény kinyerése egy ResultSet objektumból (...)

```
try {
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM names");
} catch (Exception e) {
    e.printStackTrace();
}
```

SQL lekérdező műveletek végrehajtása 2.

- Az eredménytáblának mindig csak az aktuális sora érhető el
  - A kurzor kezdetben az első sor elé mutat
  - A `next()` metódussal léptethető a következő sorra
    - `true`, ha van következő sor, `false`, ha túlszaladt
- Az aktuális sor mezőinek értékének lekérdezésére **getXYZ függvénycsalád**
  - `getByte(), getShort(), getInt(), getLong(), getFloat(), getDouble(), getBigDecimal(), getBoolean(), getString(), …`
  - Hivatkozás oszlopindexszel (balról jobbra, 1-től kezdődően) vagy oszlopnévvel (utóbbi kényelmesebb, de kevésbé hatékony)
  - **Automatikus típuskonverzió**, ha nem lehetséges → **SQLException**
  - SQL NULL érték esetén 0 vagy `null` a visszatérési érték, a NULL ellenőrzésére utólag (!) a `wasNull()` metódus szolgál

SQL lekérdező műveletek végrehajtása 3.

- A ResultSet példányt nem szükséges külön lezárni, a `Statement.close()` lezárja automatikusan azt is (bár manuálisan is megtehető a `ResultSet.close()` hívással)
- Az eredménytábla meta-adatait tartalmazó **ResultSetMetaData** példány a `getMetaData()` metódussal kérhető el
  - `getColumnCount()`: visszaadja az oszlopok számát
  - `getColumnName(int)`: visszaadja az oszlop nevét
  - `getColumnType(int)`: visszaadja az oszlop típusát (→ **java.sql.Types**)
  - ...

SQL lekérdező műveletek végrehajtása 4.

```
try {
    ResultSetMetaData rsmd = rs.getMetaData();
    for (int i = 1; i <= rsmd.getColumnCount; i++) {
        System.out.print(rsmd.getColumnName(i) + "\t");
    } System.out.println();
    while (rs.next()) {
        String vezetekNev = rs.getString("vezeteknev");
        String keresztNev = rs.getString(2);
        System.out.println(vezetekNev + "\t" + keresztNev);
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

Hibakezelés

- Minden adatbázis művelet során fellépő hiba esetén **SQLException** dobódik:
  - `getMessage()`: visszaadja a hiba szövegét (hasonlóan minden kivételhez)
  - `getErrorCode()`: visszaadja a hiba (gyártóspecifikus) kódját
  - `getSQLState()`: X/Open SQL State szerinti hibakód
  - `get/setNextException()`: láncolhatóak is a kivételek
- A hibakódok alapján és esetleg a szöveges üzenetből történő parszolással a lehetséges hibákhoz megfelelő hibakezelő ág készíthető.
  - A felhasználó számára értelmes hibaüzenet készíthető (pl. adott mező kitöltése kötelező, hibás dátumformátum, stb.).
  - Általában nem szerencsés az eredeti üzenetet a felhasználó számára megjeleníteni, mert az alkalmazás mögötti adatbázis-sémának az ismerete biztonsági kockázatot jelent.

Előfordított lekérdezések

Előfordított lekérdezések 1.

- Az előfordított lekérdezéseket a **PreparedStatement** interfész reprezentálja, amely a **Statement**-ből származik
  - ☞ `Connection.prepareStatement()` metódusával példányosítható
- A paraméteres SQL utasításban a változók helyére '?' írandó

```
PreparedStatement updateNames = con.prepareStatement( "UPDATE names
SET keresztnev = ? WHERE vezeteknev LIKE ?");
```

- Ilyenkor a lekérdezés szintaktikai elemzése és a lekérdezési terv elkészítése a példányosításkor történik meg, nem pedig közvetlenül a végrehajtás előtt

Előfordított lekérdezések 2.

- A paraméterek helyére konkrét értékek a **setXYZ** függvénycsalád segítségével illeszthetőek be (úgynevezett *bind*-olás)
  - Az első paraméter az index, azaz a '?' sorszáma (balról jobbra, 1-től kezdődően), a második paraméter pedig a használni kívánt érték
  - NULL érték beszúrása a `setNull()` segítségével
  - Nincsen implicit típuskonverzió
  - A végrehajtó metódusok ezek után argumentum nélkül hívhatóak
  - A beállított értékek többszöri végrehajtáshoz is felhasználhatóak
  - **Csak konstans bindolható** (pl. oszlopnév vagy SQL utasításrészlet nem)

Előfordított lekérdezések 3.

```
try {
    PreparedStatement updateNames = con.prepareStatement("UPDATE
    names SET keresztnev = ? WHERE vezeteknev LIKE ?");
    String[] atKeresztelendok = {"Kovács", "Nagy", "Tóth"};
    updateNames.setString(1, "Tamás");
    for(int i=0; i < atKeresztelendok.length; i++) {
        updateNames.setString(2, atKeresztelendok[i]);
        updateNames.executeUpdate();
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

Előfordított lekérdezések 4.

- A **PreparedStatement** előnyei
  - Biztonság
    - Mivel a nem megbízható forrásból (pl. felhasználó) származó input konstansként kerül felhasználására a már korábban elkészített lekérdezési tervbe, így nincs lehetőség SQL Injection támadás alkalmazására, pl.:

```
statement = ”SELECT * FROM users WHERE name = '” + userName + ”';”
userName = ”' or '1'='1”;
userName = ”a'; DROP TABLE users; --”;
```

Előfordított lekérdezések 5.

- A **PreparedStatement** előnyei
  - ☞ Hatékonyság
    - ➢ Egy lekérdezés többszöri végrehajtása esetén is csak egyszer történik meg a „drága” konstruktor hívás (pl. a lekérdezés szintaktikai elemzése és a végrehajtási terv készítése)
    - ➢ Ehhez természetesen meg kell tartani az elkészített példányt, és nem minden híváskor (pl. gombnyomás) újrapéldányosítani
      - – A paraméterek kicserélhetőek
      - – Lehet lusta inicializálást is alkalmazni, és az első híváskor példányosítani

Tárolt eljárások

Tárolt eljárások 1.

- SQL utasítások egy sorozata, amely egy bizonyos az adatokkal kapcsolatos feladatot foglal egy egységbe
  - Programvezérlő utasításokkal is kiegészített nyelv (pl. ciklusok)
    - Sajnos ez nem szabványos (pl. PL/SQL, Transact-SQL, …)
- Előnyök
  - Egyszerűsödik a kliens kódja (több utasítás helyett egyetlen utasítás)
  - Hatékonyabb
    - A tárolt eljárás kódjának értelmezésére is csak egyszer van szükség
    - Csak a végleges adatokat kell mozgatni a szerver és a kliens között
  - Az adatbázisra vonatkozó logikai kényszerek biztosíthatóak
  - Az üzleti logika elválik a megjelenítési rétegtől, így az könnyebben cserélhető
  - Biztonságosabb: nem kell pl. SELECT/INSERT/UPDATE/DELETE jogokat adni egy felhasználónak, hanem elegendő a rá vonatkozó tárolt eljárásokhoz végrehajtási jogot adni

Tárolt eljárások 2.

- A JDBC API-ban a **CallableStatement** interfész reprezentálja, amely a PreparedStatement leszármazottja
  - Az egyes bemeneti (IN) paraméterek továbbra is a **setXYZ** függvénycsalád segítségével adhatóak meg
  - `Connection.prepareCall()` segítségével példányosítható
- SQL escape szintaxis
  - {call <procedure-name>[<arg1>, <arg2>, …]}
  - { ? = call <procedure-name>[<arg1>, <arg2>, …]}

Tárolt eljárások 3.

```
// A tárolt SQL eljárás
CREATE PROCEDURE getNotlenek AS
    SELECT vezeteknev,keresztnev FROM names WHERE allapot='nőtlen'

// A alkalmazásbeli java kód
CallableStatement callable = con.prepareCall("{call getNotlenek}");
ResultSet result = callable.executeQuery();
```

Tárolt eljárások 4.

- A tárolt eljárásoknak az eredménytáblán kívül lehetnek további kimeneti (OUT) paraméterei is
  - Végrehajtás előtt definiálni kell az OUT paraméterek típusát
    `void registerOutParameter(int index, int sqlType)`
- A végrehajtás után az OUT paraméterek a **getXYZ** metódusokkal nyerhetőek ki
  - Nincs automatikus típuskonverzió
  - NULL értékek ellenőrzésére itt is a `wasNull()` metódus szolgál

Tárolt eljárások 5.

- Az előbbi lehetőségek kombinálhatóak
  - INOUT paraméterek
  - több ResultSet-tel visszatérő eljárások (ilyenkor az `execute()` metódust kell használni, a `getMoreResults()` segítségével lekérdezhető, hogy van-e több, majd a `getResultSet()` használatával sorban elkérhetőek)
  - egy eljárásnak lehet egyszerre mindenféle paramétertípusa és visszaadott többszörös ResultSet-je is (ilyenkor először a ResultSet-eket kell kiolvasni és csak utána a paramétereket)

Tárolt eljárások 6.

```
CREATE OR REPLACE  FUNCTION "MERDEI"."COMPLEX"
(be in number, ki out varchar, be_ki in out varchar )
return varchar
as
    temp varchar(100);
begin
    ki:='ez itt a kimenet';
    temp:='A bemenet=' || be || ' a be_ki=' || be_ki;
    be_ki:=be_ki || '--modositva';
    return temp;
end;
```

Tárolt eljárások 7.

```
try {
   DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
   Connection conn = DriverManager.getConnection
   ("jdbc:oracle:oci:@DBLAB_DB.BME.HU","merdei","password");
   CallableStatement complex = conn.prepareCall
      ("begin ? := complex (?,?,?); end;");

   complex.registerOutParameter(1,Types.CHAR);
   complex.registerOutParameter(3,Types.CHAR);
   complex.registerOutParameter(4,Types.CHAR);
   complex.setInt(2,200);
   complex.setString (4,"proba");
   complex.execute();

   System.out.println("A fuggveny visszateresi
   erteke:"+complex.getString(1));
   System.out.println("Az out parameter:"+complex.getString (3));
   System.out.println("Az inout parameter:"+complex.getString (4));

   complex.close();
   conn.close();

 } catch (Exception e) { ...
```

Tranzakció kezelés

Tranzakció

- Utasítások logikailag egybefüggő sorozata
- Tulajdonságok
  - **Atomi**: oszthatatlan, azaz vagy teljes egészében végrehajtásra, vagy teljesen visszavonásra kerül
  - **Konzisztens**: az adatbázist konzisztens állapotból konzisztens állapotba viszi
  - **Izolált**: a párhuzamosan futó tranzakció egymástól függetlenül hajtódnak végre, mint a soros végrehajtás esetén
  - **Tartós**: a változások a háttértáron is megjelennek

Tranzakciókezelés 1.

- **Connection** objektum létrehozásakor alapértelmezés szerint a kapcsolat auto-commit módban van
  - Minden utasítás önálló tranzakció, a végrehajtás után azonnal véglegesítődik
  - Átállítható a `setAutoCommit(boolean)` metódussal
- Manuális tranzakció kezelés
  - Commit: a Connection objektum `commit()` metódusával
  - Rollback, a tranzakció visszagörgetése: `rollback()`
- Savepointok kezelése
  - `Savepoint setSavepoint(String name)`
  - `void releaseSavepoint(Savepoint savepoint)`
  - `rollback(Savepoint savepoint)`

Tranzakciókezelés 2.

```
try {
    con = DriverManager.getConnection(url,userid,passwd);
    con.setAutoCommit(false);
    Statement stmt = con.createStatement();
    stmt.executeUpdate("UPDATE x SET allapot='nős' WHERE
      vezeteknev='Kovács' AND keresztnev='Béla'");
    stmt.executeUpdate("UPDATE x SET allapot='férjezett' WHERE
      vezeteknev='Nagy' AND keresztnev='Nóra'");
    con.commit();
    con.close();
} catch (SQLException e) {
    if (con != null) {
        try {
            con.rollback();
            System.err.println("Error! Transaction is rolled back.");
        } catch (Exception e2) {
            e.printStackTrace()
        }
    }
}
```

Tranzakciókezelés 3.

- Tranzakció izolációs szintek
  - TRANSACTION_NONE – nincs tranzakciókezelés
  - TRANSACTION_READ_UNCOMMITTED – olvasáskor az aktuális értéket látjuk → dirty read
  - TRANSACTION_READ_COMMITTED – olvasáskor csak a véglegesített tranzakciók hatását látjuk → non-repeatable read
  - TRANSACTION_REPEATABLE_READ – olvasáskor a többi tranzakciótól függetlenül is mindig ugyanazt látjuk az aktuális tranzakció végrehajtása alatt → phantom read
  - TRANSACTION_SERIALIZABLE – a tranzakció végrehajtása alatt az általa olvasott értékeket más tranzakciók nem módosíthatják
  - A Connection objektum `setTransactionIsolation(int level)` metódusa segítségével állítható be.

JDBC 2.0

lehetőségei

Fejlettebb lehetőségek

- Eddig JDBC 1.2
  - Csak olvasható, egy irányban bejárható eredménytáblák (a ResultSet-ben csak a `next` metódussal mozoghattunk)
- JDBC 2.0 újabb szolgáltatásai
  - navigálható eredménytáblák
  - adatmódosító műveletek Java nyelv segítségével SQL helyett (az eredménytáblák segítségével)
  - kötegelt utasításvégrehajtás
  - SQL3 adattípusok használata

Navigálható eredménytáblák 1.

- Eredménytáblák csoportosítása
  - navigálás iránya szerint
    - TYPE_FORWARD_ONLY: csak előre (hagyományos)
    - TYPE_SCROLL_INSENSITIVE: mindkét irányba mozoghat, de nem érzékeli mások módosításait
    - TYPE_SCROLL_SENSITIVE: mindkét irányba mozoghat, és érzékel bármilyen változást
  - módosíthatóság alapján
    - CONCUR_READ_ONLY: csak olvasható
    - CONCUR_UPDATABLE: megváltoztatható

Navigálható eredménytáblák 2.

- Ha valamely igény nem teljesülhet akkor azt egy **SQLWarning** jelzi (pl. ha a lekérdezésben JOIN van, akkor nem módosítható)
- Nem minden adatbáziskezelő-rendszer támogatja az összes eredménytábla típust
- Az adatbáziskezelő képességeiről a **DatabaseMetaData** (a Connection objektum `getMetaData()` metódusával érhető el) objektumból szerezhetünk információt az alábbi metódusokkal:
  - `supportsResultSetType(int type)`
  - `supportsResultSetConcurrency(int type, int concurrency)`
- Az utasítás példányosításánál lehet megadni, hogy milyen tulajdonságú eredménytáblát szeretnénk.

Navigálás az eredménytáblában

- Navigáló metódusok
  - `next(), previous()`
  - `last(), first()`
  - `afterLast(), beforeFirst()`
  - `absolute(int row), relative(int row)`
- Pozíció-lekérdező metódusok
  - `isAfterLast(), isBeforeFirst()`
  - `isFirst(), isLast()`
  - `getRow()`

Adatmódosító műveletek 1.

- Adatmódosítás az aktuális soron az **updateXYZ** metódusokkal (hasonlóan a getter és setter függvénycsaládhoz)
- A megváltoztatott értékek csak az `updateRow()` metódus meghívása után kerülnek be az adatbázisba
  - ☞ Ha előtte elhagyjuk a sort, a változtatások elvesznek
- A változtatások visszavonhatóak a `cancelRowUpdates()` metódussal is

Adatmódosító műveletek 2.

- Aktuális sor törlése a `deleteRow()` metódussal
  - egyes JDBC driverek esetében üres sor marad a törölt helyén, míg másoknál kiesik a törölt sor
- Új sor beszúrása
  - egy speciális területre, az ún. beszúrási pufferre navigálunk `moveToInsertRow()`
  - ilyenkor nincs aktuális sor az eredménytáblában
  - a sor értékeit beállítjuk a setXYZ metódusokkal
  - `insertRow()` metódussal érvényesítjük a beszúrást
  - elhagyjuk a puffert: vagy a `moveToCurrentRow()` metódussal (a pufferre mozgás előtti sorra visz vissza) vagy egy abszolút kurzormozgató metódussal

Változások érzékelése

- az aktuális sor frissítése: `refreshRow` metódus segítségével
- költséges, különösképpen akkor, ha a minden meghívásakor több sort kér le az adatbázistól
- sűrű hívása jelentősen csökkentheti a sebességet
- csak akkor célszerű használni, ha feltétlenül szükséges, hogy a felhasználó a legfrissebb verziót lássa
- TYPE_SCROLL_INSENSITIVE típusú eredménytábla esetén hívása hatástalan

Kötegelt végrehajtás 1.

- sok, kis adatmódosítást végző művelet összevonható, a hatékonyság növelése érdekében
- természetesen csak adatmanipulációs és adatdefiníciós műveletekre értelmezhető
- utasítások gyűjtése az addBatch metódussal
  - Statement esetén egy String paraméterben kell átadni az utasítást
  - PreparedStatement illetve CallableStatement esetén az előzőleg felparaméterezett utasítást veszi fel a pufferbe
- a puffer a clearBatch metódussal kiüríthető
- kötegelt végrehajtás elindítása az executeBatch metódussal

Kötegelt végrehajtás 2.

```
connection.setAutoCommit(false);

PreparedStatement pstmt = connection.prepareStatement("INSERT INTO
nevek VALUES (?,?)");

pstmt.setString(1,"Tóth");

String[] keresztnevek = {"Péter","István","Attila","Tamás"};

for(int i=0; i < keresztnevek.length; i++) {

    pstmt.setString(2,keresztnevek[i]);

    pstmt.addBatch();

}

int[] updateCounters = pstmt.executeBatch();
```

SQL3 adattípusok

- Blob, Clob, Array objektumok
- ezeknek megfelelő setter és getter metódusok
- Stream-ként is kezelhetők
  - ☞ getAsciiStream
  - ☞ getBinaryStream
  - ☞ getCharacterStream

SQL 4

- Driver autoloading
- Rowid
- SQLException subclasses
- Sok apróság