13. JDBC, avagy a Java adatbázis-kezelése

13.1. Mi a JDBC?

A JDBC a Java adatbázis-kezelője. A legtöbb irodalomban a JDBC-t a következőképpen definiálják:

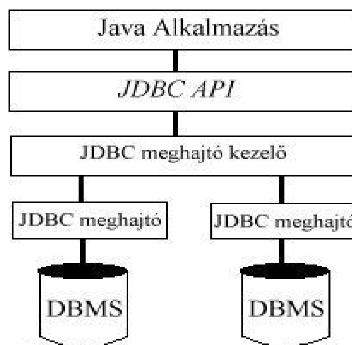
„*A JDBC™ egy Java programozói interfész (API) SQL (Structured Query Language) utasítások végrehajtására, ami egyszerű adatbázis-elérési lehetőségnél többet jelent, hiszen segítségével tetszőleges soralapú adatforrás kezelése (akár adatfájlok feldolgozása is) megoldható.*” [5]

A JDBC két dimenziós. Ennek lényege, hogy elkülöníthessük az alacsony szintű (*low level*) programozást a magas szintű (*high level*) alkalmazási interfésztől.

Az alacsony szintű programozási részt a JDBC meghajtó valósítja meg (JDBC driver), ezáltal a fejlesztőknek lehetőség nyílik különböző adatbázisokhoz kapcsolódni az előre elkészített meghajtók segítségével.

A JDBC a következők figyelembevételével épül fel:

- A *JDBC* fő célja egy adatbázis-független interfész egy „általános SQL adatbázis hozzáférési keretrendszer”
- A programozó egyetlen adatbázis interfészt készít. A JDBC használatával a program gond nélkül képes hozzáférni bármilyen adatforráshoz.



1. ábra JDBC modell

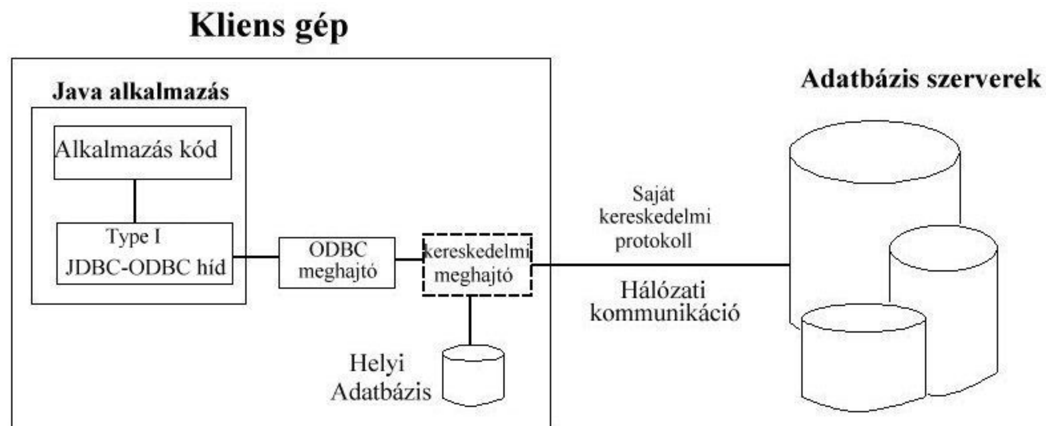
Az ábra a JDBC felépítését mutatja. A meghajtó-kezelő (*DriverManager*) a kapcsolat megnyitásáért felelős a JDBC meghajtón keresztül, amelyet a meghajtó-kezelőnek

regisztrálnia kell még a kapcsolódás előtt. A kapcsolat megkísérlésekor a meghajtó-kezelő kiválasztja a regisztrált meghajtók közül azt, amelyik megfelel az aktuális adatbázisnak. Miután létrejött a kapcsolat, a kommunikáció az adatbázissal közvetlenül a JDBC meghajtón keresztül történik. A meghajtónak implementálni kell mindazon osztályokat, amelyek megvalósítják az adatbázis funkcióit, és ezen merev követelmények biztosítják, hogy a meghajtó minden esetben teljesítse azt, amit elvárunk tőle.

13.2. JDBC meghajtó-programok

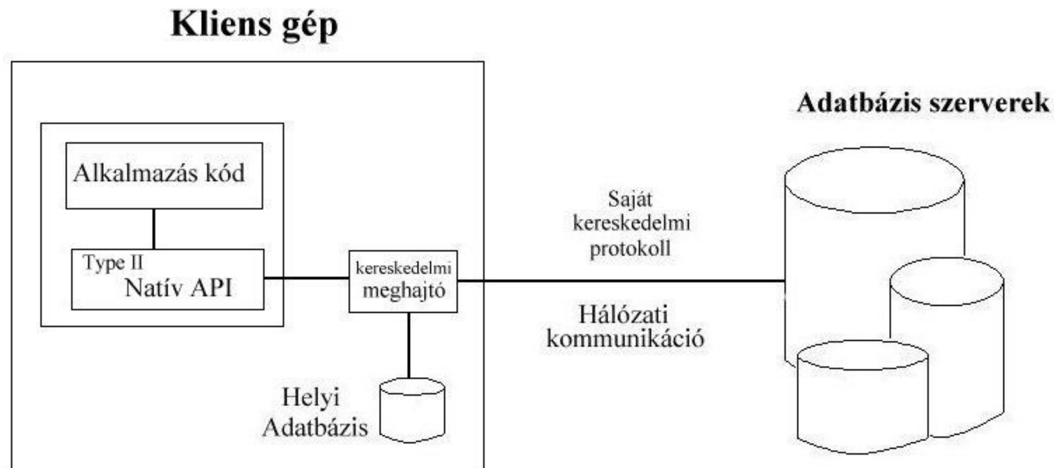
A JDBC meghajtó-programok végzik a JDBC hívások értelmezését az adatbázisok felé. Ezeket a következő típusokba szokás besorolni:

1) *JDBC-ODBC áthidaló-program és ODBC meghajtó-program*: (***Type I***)
   A JDBC hívások kiszolgálása ODBC közbeiktatásával történik.



2. ábra Type I

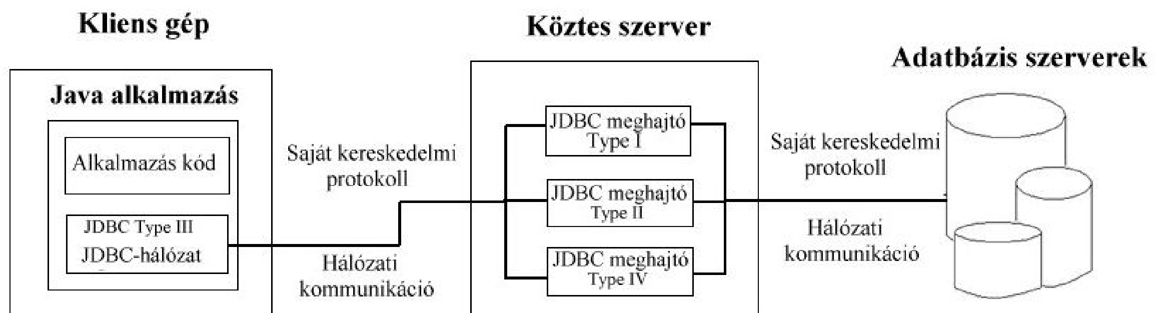
2) *JDBC-natív kliens*-API: (***Type II***)
   A Java meghajtóprogram a JDBC hívásokat közvetlenül átalakítja a megfelelő adatbázis kliens natív API hívásaira. Ebben az esetben minden gépen ott kell lennie a megfelelő adatbázis kliens-API-t megvalósító bináris program(könyvtár)-nak



3. ábra Type II

3) JDBC-hálózati protokoll (***Type III***)

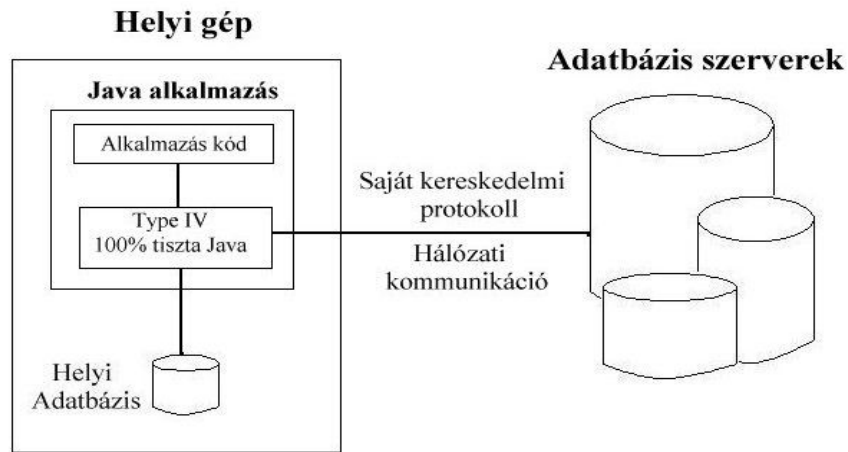
   A tisztán Java-ban írt meghajtóprogram a JDBC hívásokat adatbázis-független protokoll-hívásokká alakítja, amelyeket egy szerverprogram értelmez, és lefordít az adott adatbázis-kezelő API-jának hívásaira. Ebben az esetben a kliens nem közvetlenül az adatbázissal, hanem egy köztes szerverrel kommunikál, és csak ezen szerverprogram tart fenn kapcsolatot az adatbázissal (3-rétegű adatbázis-elérési modell)



4. ábra Type III

4) JDBC-adatbázis protokoll (***Type IV***)

   Szintén tisztán Java-ban írt meghajtóprogram, amely a hívásokat közvetlenül az adatbázis-kezelő nyelvére alakítja át. Itt nincs szükség közbülső szerverprogramra, hiszen a meghajtó-program a hálózaton keresztül közvetlenül az adatbázis-kezelővel kommunikál.



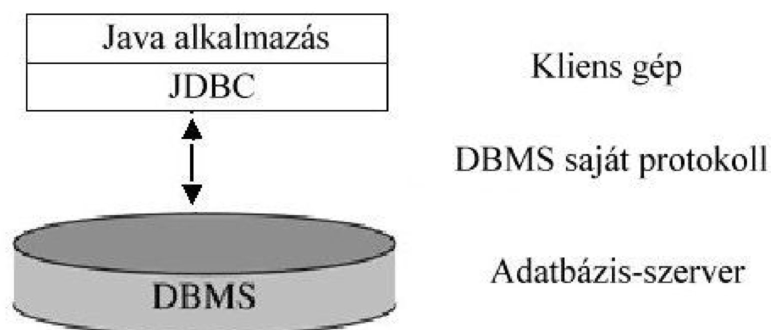
5. ábra Type IV

Az 1) és 2) esetben a Java program elveszíti hordozhatóságát, míg a 3) és 4) esetben programjaink platform-függetlenek maradnak.

13.3. Adatbázis elérési modellek

13.3.1. Kétrétegű adatbázis-elérési modell

A *kétrétegű modellben* a Java applet vagy alkalmazás közvetlenül az adatbázissal kommunikál. Ez megköveteli egy olyan JDBC meghajtó jelenlétét, amely képes kommunikálni az adott adatbázis-kezelővel. A felhasználói kérések kézbesítésre kerülnek az adatbázisnak, vagy egyéb adatforrásnak, az eredmény vagy a kimutatás pedig visszaérkezik a felhasználóhoz.



6. ábra Kétrétegű modell

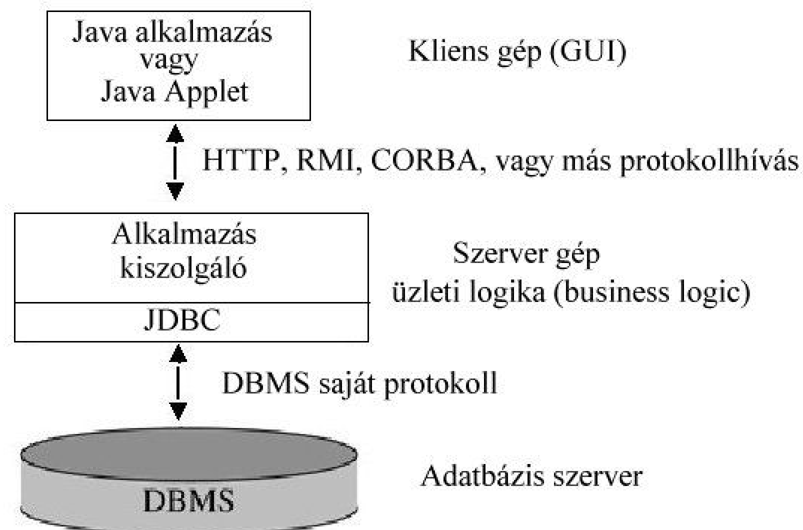
Az adatforrás elhelyezkedhet a helyi gépen, de akár egy másik gépen is, amelyhez a felhasználó a hálózaton keresztül kapcsolódik. Ez a konfiguráció a kliens/szerver modellre utal, melyben a felhasználó gépe a kliens, az adatforrást tároló gép pedig a szerver szerepét

tölti be. A hálózat tetszés szerint lehet intranet –, ami pl. egy vállalat dolgozóit köti össze – vagy akár lehet az Internet is.

13.3.2. Háromrétegű adatbázis-elérési modell

A *háromrétegű modellben* a parancsok továbbítása egy középső, szolgáltató rétegnek (middle tier) történik, majd ez továbbítja a parancsokat az adatforrásnak.

Az adatforrás feldolgozza a parancsokat, majd az eredményt a középső szolgáltató rétegen keresztül elküldi a felhasználónak. A cégek azért tartják vonzónak a háromrétegű modellt, mert a középső, szolgáltató réteg, lehetővé teszi az adatbázishoz való hozzáférés és tranzakció-kezelés felügyeletét. A modell további előnye, hogy megkönnyíti az alkalmazások továbbfejlesztését. Végül, de nem utolsó sorban e modell teljesítmény mutatói a kedvezőbbek.



7. ábra Háromrétegű modell

Régen a középső szolgáltató réteg tipikusan C vagy C++ nyelven íródott, ami nagy teljesítményt biztosított. Azonban a fordítók (compilers) fejlődésével a Java olyan fejlettségi szintet ért el, hogy a Java platform lett a középső réteg irányadó fejlesztő eszköze.

13.4. A JDBC használata

A következő ábrán látható általános struktúra minden adatbázis-kezelést használó Java programra jellemző. A következőkben ezeket a lépéseket részletezzük.

8. ábra JDBC használatának lépései

13.4.1. Meghajtó-programok kezelése

A `java.sql.Driver` interfész és a `java.sql.DriverManager` osztály szolgáltatja az eszközöket a JDBC meghajtók kezelésére. Minden JDBC-nek megfelelő meghajtó-programnak implementálni kell a Driver interfészt. A DriverManager osztállyal végezzük a JDBC meghajtók kezelését. Ennek segítségével regisztrálhatjuk, kiválaszthatjuk, de akár vissza is vonhatjuk a JDBC-nek megfelelő meghajtókat a Java programunkból. Ezek után nyilvánvaló, hogy egyszerre több meghajtó is regisztrálva lehet, és használatkor a meghajtó-menedzser az elérni kívánt adatbázist kezelő meghajtó-programot fogja aktivizálni.

Meghajtó-programok regisztrálása

A JDBC meghajtó-programok valósítják meg a JDBC hívásokat különböző adatbázisok felé. Annak érdekében, hogy a meghajtó-program hozzáférhetővé váljon a Java programban, először regisztrálnunk kell a DriverManager segítségével.

A JDBC meghajtók regisztrálására két alternatíva létezik:

- `Class.forName(String` *`driverName`*`).newInstance()`

A meghajtó-programok regisztrálásának ez az elterjedtebb módja. Azzal, hogy a Class.forName statikus metódus paraméterének egy Stringet kell megadni, nagyban növeli a rugalmasságot.

Példa:

```
String meghajto="sun.jdbc.odbc.JdbcOdbcDriver";
try
{
  Class.forName(meghajto).newInstance();
}catch (ClassNotFoundException e){
  e.printStackTrace();
}
```

- `DriverManager.registerDriver(Driver driverName)`

  Itt egy rendszerparaméter beállításával történik a meghajtóprogram regisztrálása, amelynek hátránya, hogy a rendszerparaméterek inicializálása csak egyszer, a program indításakor történik. További hátránya az is, hogy appletek esetében nem használható, hiszen itt a rendszerparaméterek beállítása nincs engedélyezve.

Példa:

```
try
{
     DriverManager.registerDriver(
            new sun.jdbc.odbc.JdbcOdbcDriver());
}catch (SqlException e){
      e.printStackTrace();
}
```

Meghajtó-programok eltávolítása

Ha bármilyen okból szükségét látjuk az egyes meghajtó-programok eltávolításának, akkor azt gond nélkül megtehetjük a `DriverManager.deregisterDriver()` metódussal. Ez teljesen hasonlóan működik a regisztrálásnál használt `registerDriver()` metódushoz.

13.4.2. Kapcsolattartás az adatbázissal

A program és az adatbázis között a kapcsolatot a `Connection` osztály reprezentálja. Egy program egyszerre több, akár különböző adatbázisokkal is tarthat fenn kapcsolatot.

Adatbázis URL

A JDBC megköveteli egy azonosítási rendszer meglétét, melynek segítségével el tudjuk dönteni, hogy melyik adatbázishoz szeretnénk kapcsolódni. Ezt az adatbázis URL biztosítja. Tehát egy adatbázis URL az elérni kívánt adatbázis azonosítására szolgál. A meghajtó-programunknak fel kell ismernie az URL alapján, hogy rá van szükség a kommunikáció lebonyolítására.

Az általános struktúra a következőképpen néz ki:

jdbc:<*al-protokoll*>:<*adatforrás leírása*>

Az *al-protokollt* a megfelelő meghajtó-program forgalmazója határozza meg. Néhány forgalmazó összetett protokollt használ adatbázis szervereinek elérésére.

Az *adatforrás leírása* a kért adatbázis eléréséhez szükséges adatokat határozza meg. Ennek szintaxisát is a meghajtó-program forgalmazója írja elő.

Példák:

- JDBC-ODBC
  Meghajtóprogram: `sun.jdbc.odbc.JdbcOdbcDriver`
  Adatbázis-URL: `jdbc:odbc:bank`

- JDBC- MS SQL (SQL Uniform Software Team JDBC meghajtója)
  Meghajtóprogram: `net.sourceforge.jtds.jdbc.Driver`
  Adatbázis- URL: `jdbc:jtds:sqlserver://london/SERVER:1433/bank`

Kapcsolat felvétele a `getConnection()` metódussal

A kapcsolat felvételének egyik lehetséges módja a Drivermanager.getConnection() metódus használata. Több előnye is van annak, hogy a kapcsolatfelvételnek ezen módját alkalmazzuk. Az egyik előnye, hogy több regisztrált meghajtó-program esetén a DriverManager ki tudja választani a megfelelő meghajtót az adatbázishoz.

Másik előnye, hogy használatával lehetőség nyílik több kapcsolatfelvételi metódus közül választani. Ezen metódusok a következők:

- `getConnection (String url)`

  Egyetlen paramétere van, az adatbázis URL. Megjegyzendő, hogy a biztonsági információk hiányoznak ebből a metódusból. Ezt akkor érdemes használni, ha az adatbázisunk nem biztosít közvetlen felhasználói hitelesítést. A legtöbb adatbázis esetén azonban ez nem használható, hiszen általában megfelelő jogokkal kell rendelkeznünk ahhoz, hogy hozzáférjünk az adatbázishoz.

- `getConnection (String url, Properties tulajdonsagok)`

  A második lehetőségnél az adatbázis URL mellett egy tulajdonság-objektum is található. Erre akkor lehet szükségünk, ha az adatbázis igényel valamilyen speciális követelményt a kapcsolódáskor.

- `getConnection (String url, String felhasznalo, String jelszo)`

  A harmadik lehetőség a legelterjedtebb, amikor az adatbázis URL mellett a hozzáféréshez szükséges felhasználónevet és jelszót adjuk meg a metódus paramétereként.

A `getConnection()` metódus meghívásakor a `DriverManager` egy érvényes `Connection` objektumot ad vissza. Ha betekintünk a kulisszák mögé, akkor ez úgy történik, hogy a DriverManager végigpróbálja a getConnection() metódus paramétereit az összes regisztrált meghajtó-programon. Az első JDBC meghajtóval kezdi, és ha nem sikerül a kapcsolódás, akkor megpróbálja a következőt, és így tovább. Addig ismétli ezt a folyamatot, amíg meg nem találja azt a meghajtót, amellyel sikeresen kapcsolódni tud az adatbázis URL-ben megadott adatbázishoz.

Ha nem talál megfelelő meghajtót, akkor egy kivétel generálódik (SqlException), amely által az adatbázisra jellemző hibaüzenetet kapunk.

Ha több olyan meghajtó-program van regisztrálva, amely megfelelő lenne a kapcsolat kialakításához, akkor a választás az időben korábban regisztrált meghajtó-programra esik.

Példa:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
DriverManager.getConnection("jdbc:odbc:bankiadatbazis");
```

Kapcsolat lezárása

A kapcsolatot lezáró `close()` metódus felszabadítja az adatbázis-kapcsolat által lefoglalt JDBC erőforrásokat, megszünteti az összes, a kapcsolaton keresztül kiadott utasítást és azok eredményeinek objektumait. Fontos, hogy kapcsolatainkat mindig lezárjuk, ha már nincs rájuk szükségünk, mert ezzel sok kellemetlenségtől kímélhetjük meg magunkat. Ha azonban elfeledkeznénk a lezárásról, a Java „hulladékgyűjtő” szolgáltatása (Garbage Collector) megteszi ezt helyettünk, de sose hagyatkozzunk csak erre.

Érdemes az egész adatbázis használtatot egy `try` blokkba beágyazni, és a kapcsolatot a `finally` ágban lezárni.

13.4.3. SQL utasítások végrehajtása

A korábbiakban láthattuk, hogy milyen lehetőségeink vannak az adatbázisokhoz kapcsolódásra. Most arra keressük a választ, hogy milyen módon lehet SQL utasításokat küldeni az adatbázisnak. Egy adatbázishoz többféleképpen küldhetünk kéréseket. Lehetséges, hogy csak egyszerű lekérdezésre van szükségünk, ami táblákból gyűjt ki adatokat vagy létrehoz táblákat. Vannak azonban bonyolultabb estek is, például amikor egy lekérdezés nem minden paraméterét ismerjük előre.
Az SQL utasítások végrehajtására három interfész áll rendelkezésünkre:

- `Statement`
  Segítségével egy általános célú, egyszerű SQL utasítások hajthatók végre. Akkor használjuk, ha nincs szükségünk paraméterekre a lekérdezésben.
- `PreparedStatement`
  Akkor használjuk, ha terveink szerint többször is szükségünk lehet erre az SQL utasításra. Gyorsabb a végrehajtása, mint az egyszerű statement lekérdezéseknek, mert előre le van fordítva. A `PreparedStatement` interfész elfogad bemenő (input) paramétereket.
- `CallableStatement`
  Akkor használjuk, ha az adatbázis tárolt eljárásaihoz szeretnénk hozzáférni. Az előzőhöz hasonlóan ez az interfész is elfogad bemenő (input) paramétereket.

Statement interfész használata

A statement interfészek hierarchiájának alapját képezi. Segítségével egyszerűen végrehajthatunk bármilyen adatmanipulációs (DML – Data Manipulating Language) és

adatdefiníciós (DDL – Data Defining Language) parancsot, és lehetővé teszi további adatbázis specifikus parancsok létrehozását is.

`Statement` létrehozása

A `statement` objektum egy fennálló kapcsolatot reprezentáló `Connection` objektum `createStatement` metódusával hozható létre.

Példa:

```
Connection kapcsolat = DriverManager.getConnection(
        url, "felhasználó", "jelszó");
Statement utasitas = kapcsolat.createStatement();
```

`Statement` végrehajtása

A Statement objektumot három különböző metódussal is végre lehet hajtatni:

- `executeQuery`
  SELECT utasítások végrehajtására használható. Eredményül egy `ResultSet` objektumot ad vissza.
- `executeUpdate`
  Adatmanipulációs (UPDATE, INSERT, DELETE) és adatdefiníciós (CREATE TABLE, DROP TABLE…stb) SQL utasítások végrehajtására használható. Eredményül a megváltozott adatbázistábla-sorok számát adja vissza.
- `execute`
  Olyan esetekben használandó, ha az utasítás többfajta eredményt ad vissza (pl. tárolt eljárások) vagy nem ismert, hogy milyen eredményt ad vissza.

PreparedStatement interfész használata

Ez az intefész a `Statement` interfész kiterjesztésével jön létre. Két dologban tér el a `Statement`-től:

- Az interfészt megvalósító objektum egy adott SQL utasítást tárol előfordított formában.
- Az SQL interfész tartalmazhat bemenő paramétereket, amelyeket az SQL utasításon belül kérdőjellel jelölünk.

`PreparedStatement` létrehozása

Egy fennálló kapcsolatot reprezentáló `Connection` objektum `prepareStatement` metódusával hozható létre az interfész.

Példa:

```
// A kapcsolat egy fennálló Connection objektum
String SQL = "UPDATE dolgozo SET fizetes=? WHERE nev=?";
PreparedStatement utasitas = kapcsolat.prepareStatement(SQL);
```

`PreparedStatement` végrehajtása

A végrehajtásra a Statement interfésznél ismertetett három metódus használható. Annyi a különbség, hogy itt nem kell paramétert megadni, mivel az elvégzendő SQL utasítás már ismert. Végrehajtáskor azonban minden bemenő paraméternek be kell állítani az aktuális értékét.

Bemenő paraméterek megadása

Egy bemenő paramétert az SQL utasításban mindig a „?” szimbolizál. Az értékeket a `set`*típusnév* metódussal lehet beállítani. Az argumentumok a következők:

- a beállítandó paraméter sorszáma (1-től számozva),
- a beállítandó érték.

CallableStatement interfész használata

Ez az interfész a `PreparedStatement` interfész kiterjesztettje. Tárolt SQL eljárások meghívására lehet használni. Tekintve, hogy a tárolt eljárást az adatbázis tartalmazza, a `CallableStatement` objektum csak ezen eljárást meghívó utasításból áll. Jelen esetben bemenő paramétereken kívül kimenő paramétereket is lehet használni az utasításban, ezek jelölése a bemenő paraméterek jelölésével megegyezik.

Fontos megjegyezni, hogy nem minden adatbázis támogatja a tárolt eljárásokat. Adott adatbázisról a `DatabaseMetadata supportStoredProcedures` metódusával lehet megtudni, hogy támogatja-e a tárolt eljárások használatát. Az adatbázisban található tárolt eljárások lekérdezését szintén a `DatabaseMetadata` egy `getProcedures` nevű metódusával lehet megkapni.

`CallableStatement` létrehozása

Egy fennálló kapcsolatot reprezentáló Connection objektum prepareCall metódusával hozható létre egy CallableStatement interfészt megvalósító objektum.

Az eljáráshívás a következők szerint történhet:

- {call <eljárásnév>(?,?,?, …)}
  Visszatérés nélküli tárolt eljárás meghívása.
- {?=call <eljárásnév> (?,?,?, …)}
  Eredményt visszaadó tárolt eljárás meghívása.

`CallableStatement` végrehajtása

A végrehajtás ugyanúgy történik, mint a `PreparedStatement` esetén.

13.4.4. Eredmények feldolgozása

A `Connection` objektum az adatbázis kapcsolatot reprezentálja, az SQL utasítások végrehajtása a `Statement` objektumok segítségével történik. Ahogy az ábrán is látható volt, az SQL eredményeket valamilyen módon fel kell dolgoznunk. Erre szolgál a `ResultSet` objektum. Ez valójában egy logikai nézet, mely az adatbázis oszlopoknak és soroknak az SQL lekérdezés szerinti megfeleltetése. A `ResultSet` bármennyi sorból és oszlopból állhat. A JDBC meghajtó biztosítja a `ResultSet` osztályt, amely az interfészt implementálja. Amikor egy Statement, PreparedStatement, `CallableStatement` objektum sikeresen végrehajt egy SQL lekérdezést, akkor az egy `ResultSet` objektumot ad vissza.

Az alapértelmezett ResultSet objektum arra ad lehetőséget, hogy megtekintsük azokat az adatokat az adatbázisban, amelyek megfelelnek a lekérdezés feltételeinek. Lehetőség van azonban arra is, hogy olyan ResultSet objektumot hozzunk létre, amelynek segítségével frissíteni tudjuk az éppen megtekintés alatt álló sort, illetve lehetőségünk van új sor beszúrására vagy akár törlésére is. Ezáltal a ResultSet objektumot használhatjuk DML utasítások végrehajtására explicit SQL lekérdezések használata nélkül.

Eredménytáblák létrehozása

Eredménytáblát a `Statement` interfész `executeQuery` és `getResultSet`, valamint a `DatabaseMetadata` interfész különböző információ-lekérdező metódusai adhatnak vissza.

Példa:

```
Satatement parancs;
```

```
…
ResultSet adattomb;
adattomb = parancs.executeQuery(”SELECT * from ugyfel”);
```

Navigálás az eredménytáblában

Annak ellenére, hogy egy `ResultSet`-ben számos sor található, egyszerre csak egyetlen sor, az aktuális sor érhető el. Ezt a sort egy külön SQL kurzor jelöli meg. Kezdetben ez a kurzor az eredménytábla első sora elé mutat. Ha egy másik sorra akarunk hivatkozni, akkor a kurzort a `ResultSet` objektum kurzormozgató metódusainak egyikével a megfelelő helyre kell mozgatni. Ezen metódusok a következők:

- `next`
  A következő sor lesz az aktuális.
- `previous`
  A megelőző sor lesz az aktuális.
- `last`
  A kurzor az utolsó sorra pozicionál.
- `first`
  A kurzor az első sorra pozicionál.
- `afterlast`
  A kurzor az eredménytábla vége után pozicionál.
- `beforefirst`
  Az eredménytábla első sora elé pozicionál.

Adatok kinyerése az eredménytáblából

Az SQL lekérdezés eredményeként kapott adatok JDBC adattípusok. Ahhoz, hogy programunkban használni tudjuk, először át kell alakítani őket Java adattípusokká. A `ResultSet` objektum biztosítja a `get<típusnév>()` metódust, amelynek segítségével elvégezhető a konverzió. Ezen metódussal lehetőségünk van típuskényszerítésre is. Ez azt jelenti, hogy különböző SQL adattípusokat eltérő típusú `get` metódussal olvasunk be. Például egy SQL `int` típust `getString()` metódussal olvassuk be.

Példa:

```
try{
    ResultSet adattomb;
```

```
      adattomb= parancs.executeQuery(
                 ”SELECT vezeteknev from ugyfel”);
      while (adattomb.next()){
            nev = adattomb.getString(”vezeteknev”);
            System.out.println(nev);
      }
}
catch (Exception e){
  System.out.println(”Hiba az eredmény feldolgozása közben”);
}
```