11. Gyakorlat

legendi@inf.elte.hu

2010. április 28.

Kiegészítés

StringBuilder vs. StringBuffer `StringBuffer` szinkronizált, a `StringBuilder` nem, cserébe hatékonyabb (nincs szinkronizációs költség).

Szál indítása Figyeljetek! **Nem** a `run()`, hanem a `start()` függvény használandó erre. Előbbi hatására ugyanúgy szekvenciális programunk lesz.

Deprecated függvények a Thread osztályban Erősen deadlock-prone függvények. Pl. egy szál `synchronized` blokkban van, és suspendelik, akkor nem fogja eleresztени az erőforrásokat, így más nem juthat hozzá - így könnyen deadlock alakulhat ki. Részletes magyarázat, megoldások, workaroundok itt találhatók: `http://java.sun.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`

Párhuzamosság 2.

Feladat Adott $v_1, v_2, ..., v_n$ vektorok, amelyen n szál dolgozik. A program parancssori argumenetunkként kap egy e értéket. Keressük meg az első olyan j indexet, ahol ez a szám megtalálható, vagyis $v_i[j] = e, 1 \leq j \leq n$. *Az egyszerűség kedvéért feltételezzük, hogy minden vektorelem egyedi, valamelyik vektorban megtalálható a keresett e érték, és $n = 2$.*

'Nuff said, let's rock!

1. kísérlet

Indítsunk két szálat. Közös változó a `found`, lokális változó a `v, i`.

```
found = false; i = 0;     // A
while (!found) {          // B
  found = v.get(i) == e; // C
  i++;                    // D
}
```

FAIL Tegyük fel, hogy az egyes szál elindult, az i. elem épp e, C végrehajtása után kapja meg a vezérlést a második szál. A inicializáló utasításával `found` ismét hamis lesz **végtelen ciklus**.

2. kísérlet

Ja, akkor nem a szálakban inicializálok. Kerüljük el, hogy minden szál külö-külön is inicializálja a közös változót, tegyük meg ezt a szálak indítása előtt!

```
found = false // Threadek inditasa elott

i = 0;                    // A
while (!found) {          // B
  found = v.get(i) == e; // C
  i++;                    // D
}
```

FAIL Tegyük fel, hogy az egyes szál C-hez ér, végrehajtja, és épp megtalálja az adott értéket! Így `found` értéke igaz lesz. De! Ha közben a másik szál is C-nél volt, és ezután hajtódik végre, `found` értéke ismét hamis lesz → **végtelen ciklus**.

3. kísérlet

Ouch, tényleg! Csak akkor adjunk új értéket a `found` változónak, ha megtaláltuk az elemet.

```
i = 0;                       // A
while (!found) {             // B
  if (e == v[i]) b = true;  // C
  i++;                       // D
}
```

FAIL Tegyük fel hogy az első szál az első elemében rögtön fel is fedezi az e keresett értéket, és terminál (sérül a feltétlen pártatlan ütemezés elve, a szálak nem dolgoznak szinkronban). Ekkor a második szál soha nem terminál, → **végtelen várakozás**.

4. kísérlet

Jó, akkor ütemezek én! Vezessünk be egy `next` flaget, amely jelölje, hogy melyik szál futhat a `while` ciklusba való belépés után! A feltételhez kötött várakozást `await` szimbólummal jelölve, az első szál definíciója:

```
i = 0;                              // A
while (!found) {                    // B
  await (1 == next) { next = 2; } // C
  if (e == v[i]) b = true;          // D
  i++;                              // E
}
```

valamint a második szál definíciója legyen a következő:

```
j = 0;                              // A
while (!found) {                    // B
  await (2 == next) { next = 1; } // C
  if (e == v[j]) b = true;          // D
  j++;                              // E
}
```

FAIL Tegyük fel, hogy az első szál eljut D végrehajtásáig, majd ezután a második szál is eljut ugyaneddig (`next` értékét 2-re állítva). Tegyük fel, hogy a második szál következő eleme nem a keresett e elem, így a C ponton tovább várakozik. Ha eközben az első szál megtalálja a keresett elemet, és terminál → **holtpont**.

5. kísérlet

Ooooh! És ha terminálásnál is jelzek?! A szálak terminálásánál is figyeljünk a `next` változóra! Az első szál kódját módosítsuk a következőképp:

```
i = 0;                              // A
while (!found) {                    // B
  await (1 == next) { next = 2; } // C
  if (e == v[i]) b = true;          // D
  i++;                              // E
}
next = 2;                           // F
```

a másodikét pedig az alábbi módon:

```
j = 0;                              // A
while (!found) {                    // B
  await (2 == next) { next = 1; } // C
```

```
  if (e == v[j]) b = true;          // D
  j++;                              // E
}
next = 1;                           // F
```

Na ez már menni fog. ☺

Lásd még

Peterson-féle algoritmus kölcsönös kizárás megoldására, vektorértékadás atomicitása nélkül.

Irodalom

1. Brian Goetz et al.: Java Concurrency in Practice, Addison-Wesley Professional, May 19, 2006.
2. Kozma, L. és Varga, L.: A szoftvertechnológia elméleti kérdései, ELTE Eötvös Kiadó, első kiadás 2003, második kiadás 2006.

Kliens-szerver architektúra

A szerveroldali kód:

```
// Raakaszkodas a portra
ServerSocket ss = new ServerSocket( port );
// Fuss, amig...
while (true) {
    // Egy bejovo kapcsolat elkapasa
    Socket newSocket = ss.accept();

    // Kapcsolat kezelese
    // ...
}
```

Feladat

Készítsetek egy többszálú chat szerveralkalmazást, valamint egy klienst hozzá! Ha valaki küld egy üzenetet a szervernek, a szerveralkalmazás broadcastolja azt mindenki másnak is. A szerver a 2442 számú porton figyeljen, és ide csatlakozzanak a kliensek is!

A szerveralkalmazás minden egyes bejövő kapcsolatot külön szállal kezeljen, a váza valahogy így nézzen ki:

```
ServerSocket socket = new ServerSocket(PORT);
while (true) {
    new Handler(socket.accept()).start();
}
```

A kliensek is legyenek többszálú alkalmazások: az egyik szál folyamatosan figyelje, hogy nem jön-e új üzenet a csatornán, miközben a másik szál írjon a csatornára, ha a felhasználó üzenetet írt a konzolra!

Linkek

Socket példa `http://java.sun.com/developer/onlineTraining/Programming/BasicJava2/socket.html`

Java Tutorial, All About Socket fejezete `http://java.sun.com/docs/books/tutorial/networking/sockets/index.html`

Kapcsolat az adatbázissal

Kliens-szerver architektúra kiváló különböző adatbázisok eléréséhez: *Java DataBase Connectivity*. Ez egy szabványos API, `java.sql.*` (*Core API*), `javax.sql.*` (*Extension*) csomagokban definiált osztályok (v.ö. ODBC).

A szolgáltatások három csoportba sorolhatók:

1. Kapcsolódás a DB-hez
2. Utasítások végrehajtása
3. Eredmény lekérdezése

A DB elérhető natívan is (kétrétegű modell), de az API három rétegű: egy absztrakt szint bevezetésével leválaszthatók a DB-specifikus dolgok, a kommunikáció a JDBC-n keresztül történik.

Meghajtóprogramok

Minden kapcsolathoz szükséges a hozzá tartozó meghajtóprogram betöltése, amely kezeli a kapcsolatot, megoldja a hívások értelmezését és kiszolgálását (változó, hogy milyenek vannak, pl. Oracle fizetős, MySQL ingyenes). Minden ilyen osztály a `Driver` interfészt implementálja. A futtatáshoz kell a megfelelő meghajtó jar fájl is, pl.:

```
> java -cp .;lib/mysql.jar MyMySQLTestClass
```

Nem kötelező kézzel betölteni, ha több DB-t is támogat a programunk, használhatjuk a `DriverManager` osztályt, amely megpróbálja betölteni az éppen aktuálisan használt DB-hez a megfelelő meghajtóprogramot.

A kapcsolat a DB-vel a `Connection` osztályon keresztül történik (egyszerre több kapcsolat is fenntartható, ez a DB beállításától függő érték). Ezt egy URL-ben kell megadni, amely a következő formátummal rendelkezik:

```
jdbc:alprotokoll:adatforrás_leírása
```

Ahol:

1. maga a protokoll a **jdbc**
2. az alprotokoll megegyezik a forgalmazó nevével
3. az adatforrás leírása pedig tartalmazhatja a DB szerver címét (*host:port*), az adattábla nevét, és tartalmazhatja a felhasználó nevét, jelszavát

Itt a gyakon egy egyszerű, minimális DB kezelőt, a Derby-t fogjuk használni (*"Java DB is a free, fast, robust, full-featured pure Java database that fits in a 2.5MB JAR file, blah-blah-buzzword-blah-blah)"*.[1] Ehhez a következő osztály dinamikus betöltésére van szükség (ő implementálja a `Driver` interfészt):

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

A driverek specifikáció szerint osztálybetöltéskor egy statikus inicializátor blokkban bejegyzik magukat a `DriverManager` osztályban, így rendelkezésre állnak. A kapcsolat kiépítéséhez a következő URL-t használhatjuk:

```
jdbc:derby:[//host:port//]<dbName>[properties]
```

Ahol a **properties** tartalmazhatja a következő információkat (ezeket ; karakterekkel választhatjuk el):

- `create=true` Megpróbálja létrehozni a DB-t, ha még nincs. Adattáblákat nem csinál.
- `user=userName` DB felhasználó neve.
- `password=userPassword` DB felhasználó jelszava.
- `shutdown=true`

[1] Letölthető innen: `http://developers.sun.com/javadb/`

```
Connection dbConnection = null;
String strUrl = "jdbc:derby:DefaultAddressBook;user=dbuser;password=dbuserpwd";
try {
    dbConnection = DriverManager.getConnection(strUrl);
} catch (SQLException e) {
    e.printStackTrace();
}
```

Másik megoldás (kicsit biztonságosabb), ha property-kbe rakjuk a felhasználó nevét és jelszavát:

```
Connection dbConnection = null;
String strUrl = "jdbc:derby:DefaultAddressBook";

Properties props = new Properties();
props.put("user", "dbuser");
props.put("password", "dbuserpwd");
try {
    dbConnection = DriverManager.getConnection(strUrl, props);
} catch(SQLException sqle) {
    sqle.printStackTrace();
}
```

Hova kerul a DB? A `derby.system.home` system property által beállított érték határozza meg. Ezt vagy kódból lehet beállítani:

```
System.setProperty("derby.system.home", "/tmp");
```

vagy futtatásnál lehet megadni:

```
> java -cp .;lib/mysql.jar -Dderby.system.home="/tmp" MyDerbyTestClass
```

Ezt a kapcsolatot is ugyanúgy le kell zárni, mint a streameket. És nem, nem a finalize() függvényben!! A kapcsolatról rengeteg hasznos információ elkérhető a `getMetaData()` függvényhívással.

Tranzakciókezelés

Tranzakciókezelés támogatott (csak olyan SQL utasítás hajtható végre, amelynek eredményét vagy véglegesítjük a DB-ben (**commit**), vagy visszavonunk minden változtatást (**rollback**)). Ez alapból be van kapcsolva, aki kikapcsolja vagy tudja, hogy mit csinál, vagy vessen magára.

JDBC - Java type mapping

TODO: táblázat ide

Utasítások végrehajtása

Három lehetőség:

1. Statement: egyszerű SQL utasításokhoz. **Gyakon csak ez.**
2. PreparedStatement: bemenő paramétereket tartalmazó, előfordított SQL utasításokhoz.
3. CallableStatement: bemenő, kimenő paramétereket tartalmazó, tárolt eljárások hívásához.

Statement végrehajtása

- `execute(String)`: tetszőleges utasításhoz, pl. tábla létrehozása:

```
String strCreateTable = "CREATE TABLE inventory
(
   id INT PRIMARY KEY,
   product VARCHAR(50),
   quantity INT,
   price DECIMAL
)";

statement = dbConnection.createStatement();
statement.execute(strCreateTable);
```

- `executeQuery(String)`: lekérdezéshez, az eredmény egy `ResultSet` objektum lesz. Pl.:

```
ResultSet rs = statement.executeQuery("SELECT * FROM inventory");
while (rs.next()) {
    String p = rs.getString("product");
    int q = rs.getInt("quantity");
    double d = rs.getDouble("price");
    ...
}
```

- `executeUpdate(String)`: insert, update, delete, és adattedinícios utasításokhoz, az eredmény a módosított sorok száma (vagy 0). Pl.:

```
statement.executeUpdate("DELETE WHERE id=0");
```

Kötegelt végrehajtás

Van rá lehtőség, hogy parancsokat összefogjunk, és egyszerre küldjünk el a szervernek feldolgozásra, így sok kis adatmódosító utasítás gyorsabban lefuthat, mintha külön-külön futtatgatnánk le őket. Pl.:

```
statement.addBatch("Create TABLE ...");
statement.addBatch("INSERT INTO ...");
statement.addBatch("INSERT INTO ...");
statement.addBatch("INSERT INTO ...");
...
statement.executeBatch();
```

Az `executeBatch()` egy tömbbel tér vissza, hogy az egyes utasítások hány sort változtattak a DB-ben (itt $[0, 1, 1, 1, ...]$ lesz).

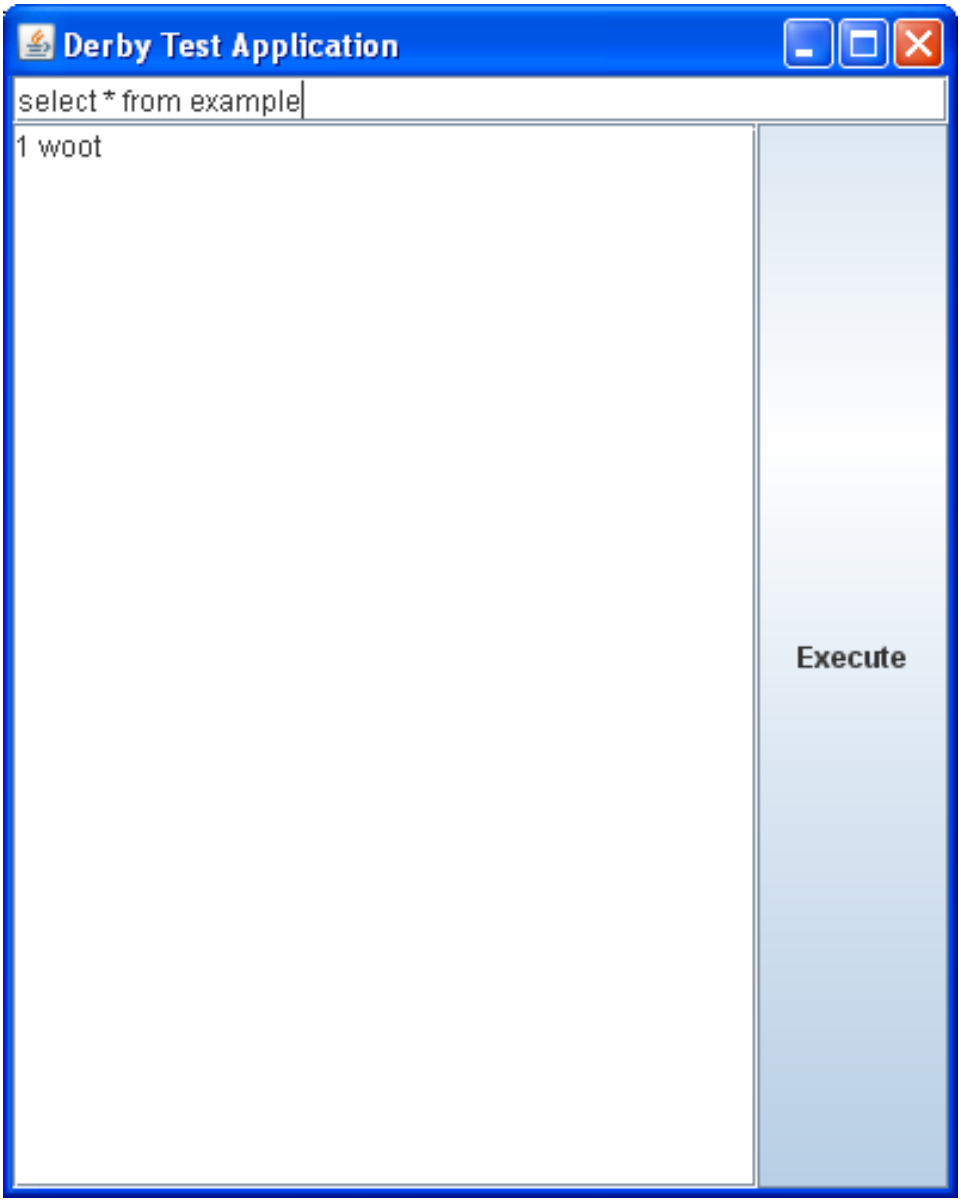
Részletesen

- `http://java.sun.com/docs/books/tutorial/jdbc/index.html`
- `http://www.jdbc-tutorial.com/`
- `http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javadb/`

Feladat

Készítsetek egy egyszerű adatbázis kezelő grafikus felületet, amely az 1. ábrán látható! A program tartalmazzon egy `JTextField` komponenst, ahol a lekérdezést lehet megadni, egy `JTextArea` komponenst, ahol megjeleníti az eredményt, valamint egy gombot, amivel le lehet futtatni a megadott SQL utasítást.

1. A megvalósításhoz használjátok a következő címen elérhető `derby.jar` fájlt: `http://people.inf.elte.hu/legendi/java/res/derby.jar`
2. A program az aktuális könyvtár alá, egy `derby` könyvtárba tegye az adatbázis fájlokat!
3. Egy statikus inicializáló blokkban próbáljuk meg betölteni a szükséges meghajtó osztályt! Ha ez nem megy, terminálljon a program.
4. Az utasítás végrehajtásához használjátok az `execute(String sql)` függvényt!
5. Az eredmény objektum bejárásánál elég, ha az elemeket a `getString(int)` metódussal írjátok ki. Ehhez tudnotok kell, hogy hány oszlop található az eredményben, ezt a `ResultSet#getMetaData()` függvényen keresztül elért objektumtól tudjátok lekérdezni.



1. ábra. A program grafikus szerkezete