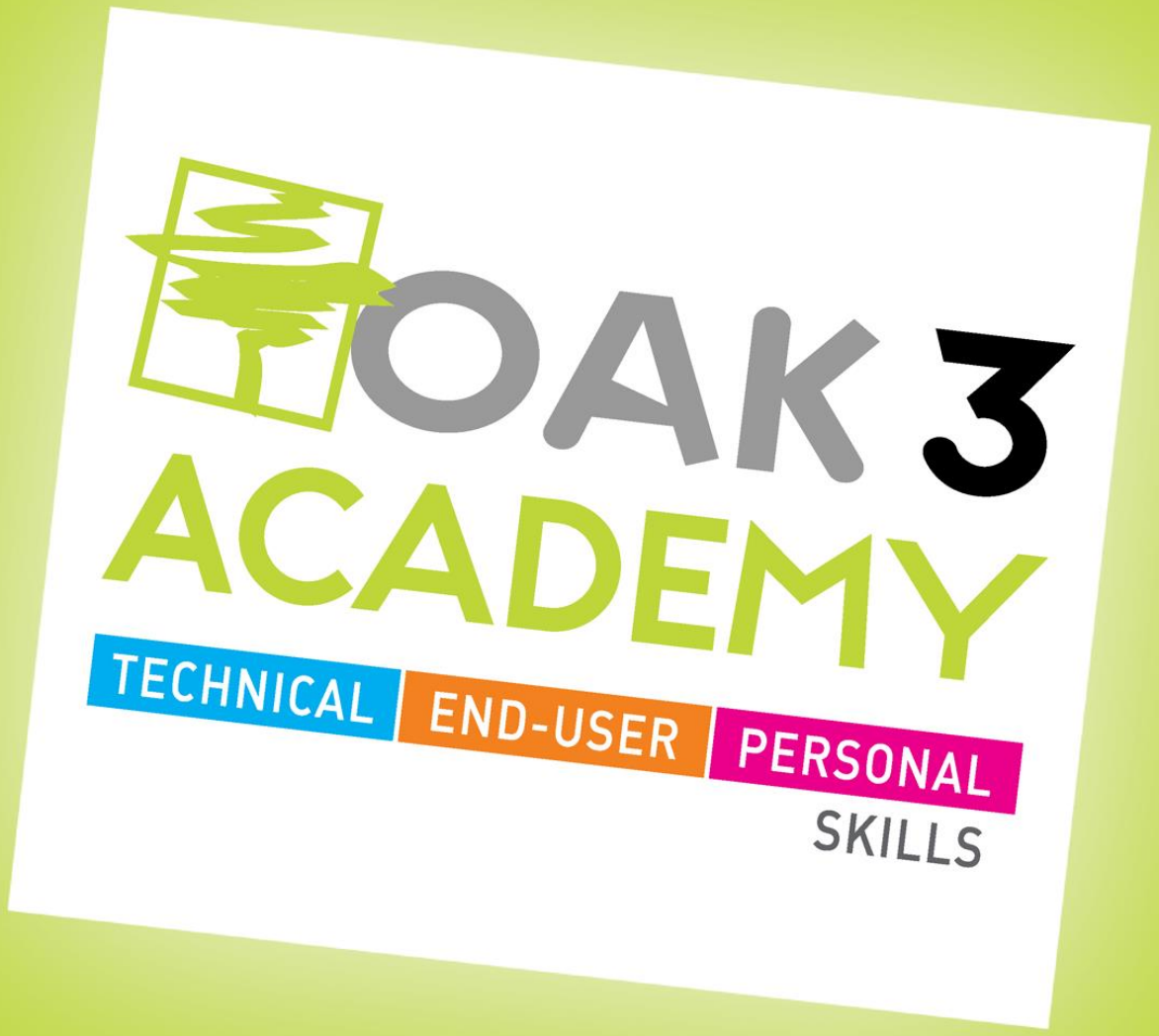
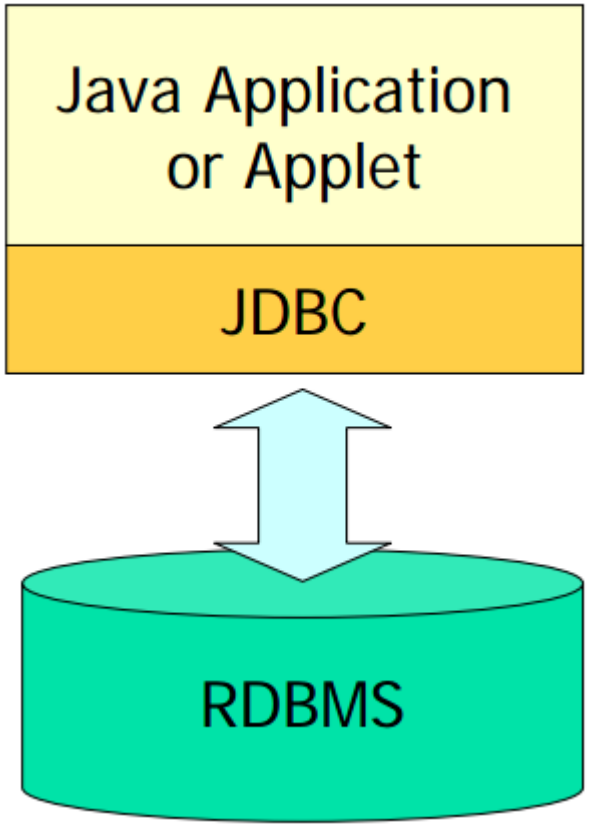


JDBC: Databasetoegang

Inleiding

Wat is JDBC ?

= Java Database Connectivity



- **JDBC**, is een Java API (= application programming interface).
- Een Java-programma kan via JDBC in SQL communiceren met een database.

Benodigdheden

RDMS: Relational Database Managment System (Oracle, MySQL, MS SQL …)

Connector: connectie tussen JDBC en RDMS

IDE - JDK

MySQL en Connector downloaden

De verbinding tussen JDBC en het RDMS gebeurt via een connector die we eerst moeten downloaden of via Maven toevoegen aan onze Dependencies.

7 stappen

De 7 stappen om de database te benaderen

1. **De driver laden** (sinds Java 6 niet meer nodig)
2. **De connectie URL definiëren**
3. **De connectie openen**
    4. **Een Statement object aanmaken**
    5. **Een SQL-query uitvoeren**
    6. **De resultaten verwerken**
7. **De connectie afsluiten**

1. De driver laden

Deze stap mag je vanaf Java 6 overslaan. De driver wordt dan namelijk automatisch geladen.

Zo moest het vroeger:

```
try {
   Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException cnfe) {
   System.out.println("Error: " + cnfe);
}
```

2. De connectie URL definiëren

De connectie URL heeft deze vorm:

jdbc:<DBMS>://<HOSTNAME>:<PORT_NUMBER>/DATABASE_NAME

Wij gebruiken MySQL als DBMS.

String url
=“jdbc:mysql://noelvaes.eu:3306/StudentDB”

3. De connectie openen

We gebruiken hiervoor een **Connection object** en de methode **getConnection** van de class **DriverManager** die beiden in het package **java.sql** zitten:

```
String url = “jdbc:mysql://noelvaes.eu/StudentDB”;
String username   = “student”;
String password   =  “student123”;

Connection con =
DriverManager.getConnection(url, username, password);
```

Hierbij verkrijgt dbConnection een **open verbinding** naar onze database. Username en password zijn optionele parameters.

De connectie openen

OPMERKING:
Stop het openen van de connectie in een “try-with-resources” statement.

```
try (Connection con =
     DriverManager.getConnection(url, username,
password);) {
…

}
catch (SQLException sqle) {
…
}
```

4. Een Statement object aanmaken

Om queries of commandos naar het RDBMS te sturen gebruiken we een Statement object (uit de package java.sql).

We onderscheiden daarbij volgende types:

- gewone **Statement**,
- een **PreparedStatement;**

```
try (Connection con =
      DriverManager.getConnection(url, username, password);
      Statement stmt = con.createStatement( ); ) {



}
catch (Exception ex) {
…
}
```

Een Statement object aanmaken

OPMERKING:
Ook hier weer stop je best het aanmaken van een statement object in een "try-with-resources" statement:

```
try (Connection con =
        DriverManager.getConnection(url, username, password);
        Statement stmt = con.createStatement( ); ) {



}
catch (SQLException sqle) {
…
}
```

5. Een SQL query uitvoeren

3 methods van ons Statement object:

executeQuery()	Wordt gewoonlijk gebruikt om de inhoud van de database te lezen (**SELECT**). Geeft een ResultSet terug.
executeUpdate()	Gewoonlijk gebruikt om de database te wijzigen. (**DROP TABLE of DATABASE, INSERT into TABLE, UPDATE TABLE, DELETE from TABLE**). Het resultaat is van het type int. Deze waarde duidt aan hoeveel rijen er gewijzigd zijn geweest door de query.
execute()	Geeft een boolean terug. True betekent dat het resultaat een ResultSet is. False betekent dat het resultaat een int is, die de waarde aanduidt hoeveel rijen er beïnvloed zijn geweest door de query.

VOORBEELD:

```
String query = “select * from beers”;

...

ResultSet rs = stmt.executeQuery(query);
```

6. De resultaten verwerken

De class Resultset kent vele methods. Zie de java documentatie.

boolean **next()**	Verplaatst de cursor één rij verder. Als er geen rijen meer zijn dan wordt de waarde FALSE teruggegeven, anderzijds de waarde TRUE.
String **getString(**String kolomLabel**)**	Geeft de waarde van een kolom weer, ofwel NULL indien geen waarde gevonden werd.

VOORBEELD:

```
ResultSet rs = stmt.executeQuery(query);

while ( rs.next( ) ) {
        System.out.println( rs.getString( “naam” );
}
```

Resultset

- Bevat de rijen die voldoen aan een select-statement.
- De data is **rij per rij toegankelijk** door gebruik te maken van een cursor die de huidige rij aangeeft. Deze cursor beweegt alleen voorwaarts door middel van de method **next()** en staat aanvankelijk vóór de eerste rij.

Scrollable resultsets

- De data is **rij per rij voorwaards toegankelijk** door middel van de method **next()**.
- Bij **scrollable resultsets** is de data ook **achterwaarts toegankelijk** d.m.v. de method **previous()**.
- en daarnaast ook **direct toegankelijk**:
  - first()
  - last()
  - absolute(*recordnr*)
  - relative(*rows*)

→ Extra parameters meegeven met de method **createStatement()** van de databaseconnection:
een **resultSetType** en een **resultSetConcurrency**.

Scrollable resultsets

Voorbeeld:

```
    ...
statement = databaseConnection.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);

rs = stmt.executeQuery("select * from Beers");

rs.absolute(23); // Spring naar het 23ste record
```

Scrollable resultsets

Mogelijke waarden voor **resultSetType**:

TYPE_FORWARD_ONLY
(m.a.w. geen scrollable resultset)

TYPE_SCROLL_INSENSITIVE
Als er tijdens het scrollen veranderingen gebeuren door andere gebruikers, zijn deze onzichtbaar.

TYPE_SCROLL_SENSITIVE
Wijzigingen van andere gebruikers zijn wél zichtbaar.

Mogelijke waarden voor **resultSetConcurrency**:

CONCUR_READ_ONLY : de resultset is read-only

CONCUR_UPDATABLE : de resultset is updatable

7. De connectie sluiten

Voor de komst van Java 7 → method close().

Vanaf Java 7 → “try-with-resources” statement.

Klassikale demo: Connectie maken

1. MySQL starten → Connectie van Noël Vaes toevoegen
2. Student1DB – Student2DB – Student3DB
3. Execute the following query: select * from Beers

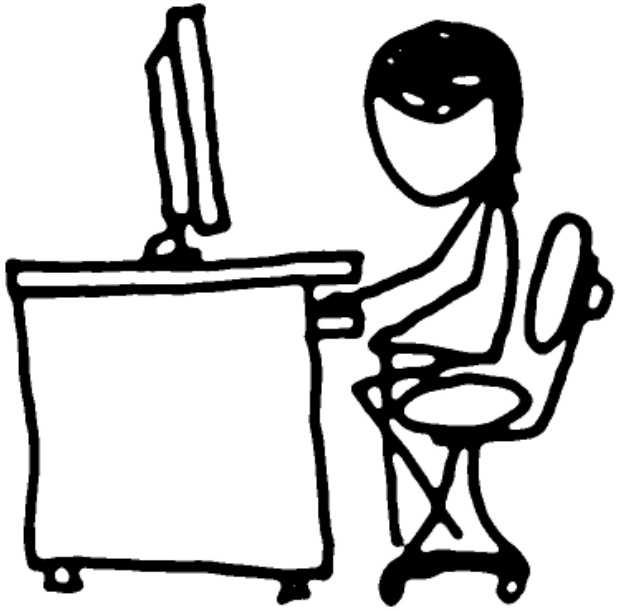
Klassikale demo: Eerste JDBC programma

1. Nieuw Java Project aanmaken: JDBC
2. ConnectorJ toevoegen aan ons project
3. Connectie maken met de database (en uittesten).

Oefeningen

Opdracht 1 samen maken.

Opdracht 2, 3, 4 en 5 zelfstandig maken in de cursus



PreparedStatement

PreparedStatement

WAAROM?
Een gewoon statement dat veelvuldig gebruikt wordt voor gelijkaardige SQL- commando's zal **performance** issues veroorzaken. Dit omdat het statement telkens zal moeten gecompileerd worden door de server alvorens het kan uitgevoerd worden.

→ PreparedStatements: Eenmalig gecompileerd door de server en daarom sneller!

PreparedStatement

VOORBEELD

Parameter

```
PreparedStatement ps = co .prepareStatement( “select *
from planten where kleur = ?”);

ps.setString(1,”Geel”);
ResultSet rs = ps.exe uteQuery();
```

Waarde van de eerste (en hier enige) parameter wordt “Geel”

PreparedStatement

NOG EEN VOORBEELD

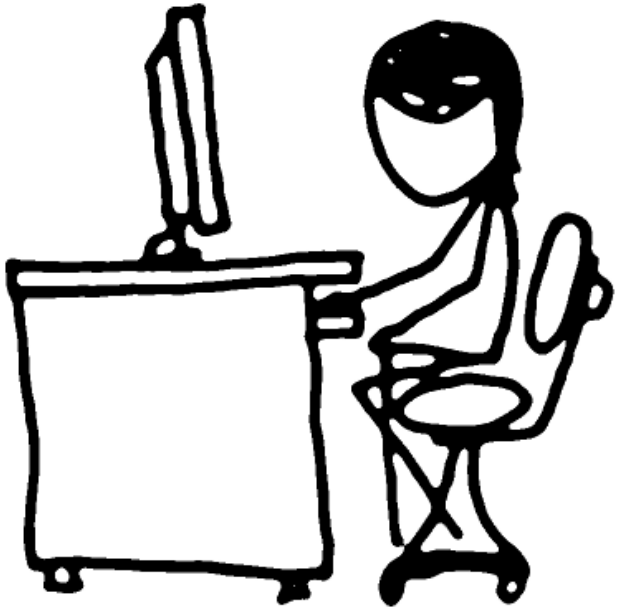
```
ps = con.prepareStatement (
"insert into leveranciers (naam, adres, postnr, woonplaats)”
+ “values (?,?,?,?)" );

ps.setString(1,"Fluitekruid"); ps.setString(2,"Prieelstraat 3");
ps.setString(3,"2323"); ps.setString(4,"Wortel");

aantal = ps.executeUpdate();
```

Oefeningen

Opdracht 6 maken in de cursus



Transactions

Transactions

SQL commando's worden in JDBC **standaard** automatisch en onmiddellijk **elk afzonderlijk** uitgevoerd.(= autocommit)

Transactie = uitvoeren van **meerdere SQL commando's als één geheel**. Mislukt een commando dan worden ze allemaal geannuleerd.

VOORBEELD VAN EEN TRANSACTIE:

Overschrijving rekening1 → rekening2:

stap 1: Verminder rekening1 met het bedrag

stap 2: Vermeerder rekening2 met het bedrag

De auto-commit mode

1. Door eerst de autocommit modus uit te zetten met de methode **setAutoCommit** van de interface Connection. Dan de queries zoals gewoonlijk uitvoeren:

```
…
databaseConnection = DriveManager.getConnection(...);
databaseConnection.setAutoCommit(false);
…
// een statement aanmaken en een “execute” method oproepen
// idem voor (een/de) volgende statement(s)
…
```

2. Tenslotte de “commit” methode aanroepen:

```
...
databaseConnection.commit();
...
```

Een transactie terugdraaien

rollback( )

Methode van de interface Connection om commando's expliciet te annuleren. De transactie wordt teruggedraaid tot de laatste commit.

savepoints

Door binnen een transactie 'savepoints' aan te brengen kan je een transactie deels rollback-en. De savepoint geef je dan als parameter meer bij de method rollback( ).

VOORBEELD

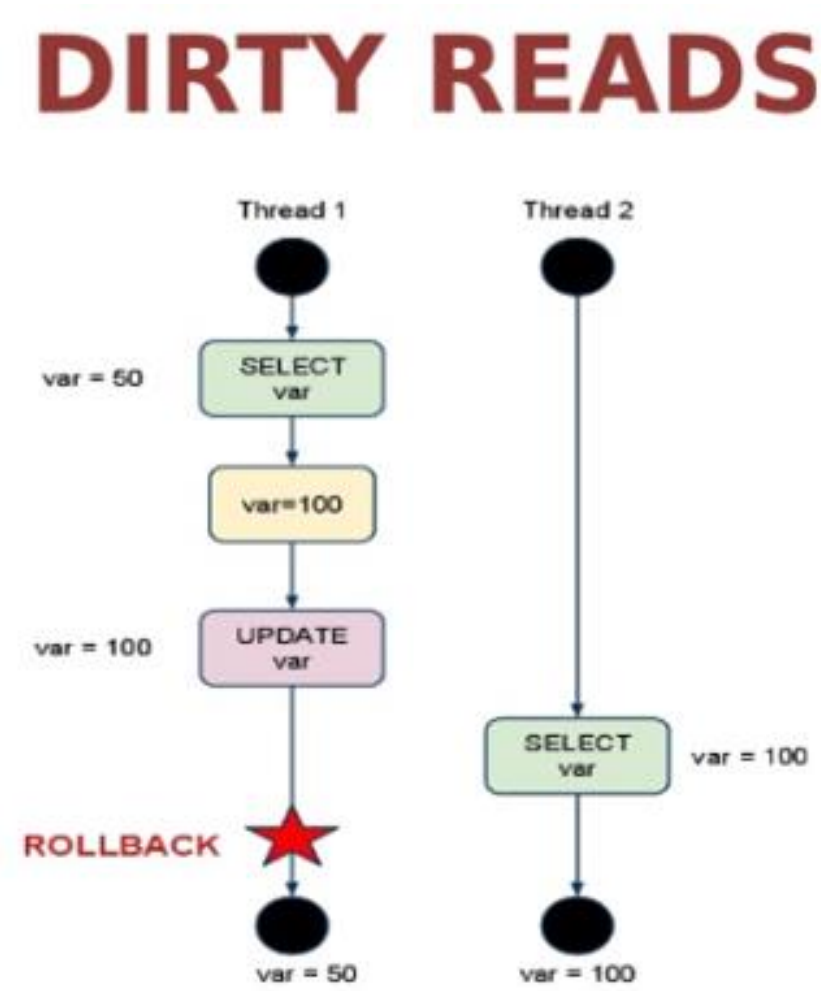
```
Savepoint svpt = con.setSavepoint("Kom_hier");
…
con.rollback(svpt);
```

Transaction: problemen bij rollback

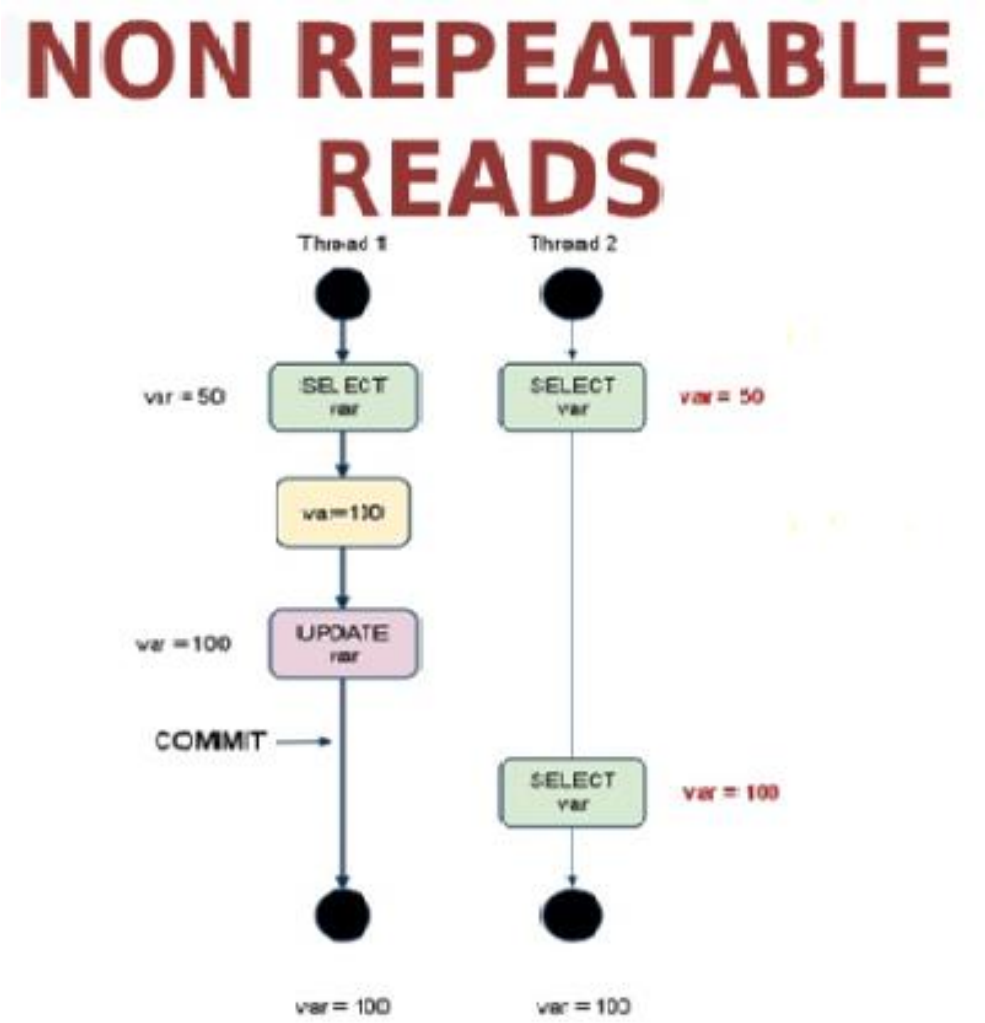
Tijdens de transactie worden de gegevens **vergrendeld**. Na het afsluiten van de transactie (na een commit of rollback) kunnen wijzigingen aangebracht worden. **Lezen is altijd mogelijk**. Dit kan leiden tot volgende mogelijkheden:

Dirty read	Treedt op als een transactie reeds gegevens leest die nog niet gecommit zijn door een andere transactie.
Non-repeateable read	Transactie A leest een rij, transactie B verandert dezelfde rij en vervolgens leest transactie A de rij opnieuw, maar krijgt andere waarden.
Phantom read	Transactie A leest een aantal rijen, transactie B voegt ondertussen een nieuwe rij toe en vervolgens leest transactie A de rijen opnieuw maar krijgt er een extra (spook)rij bij.

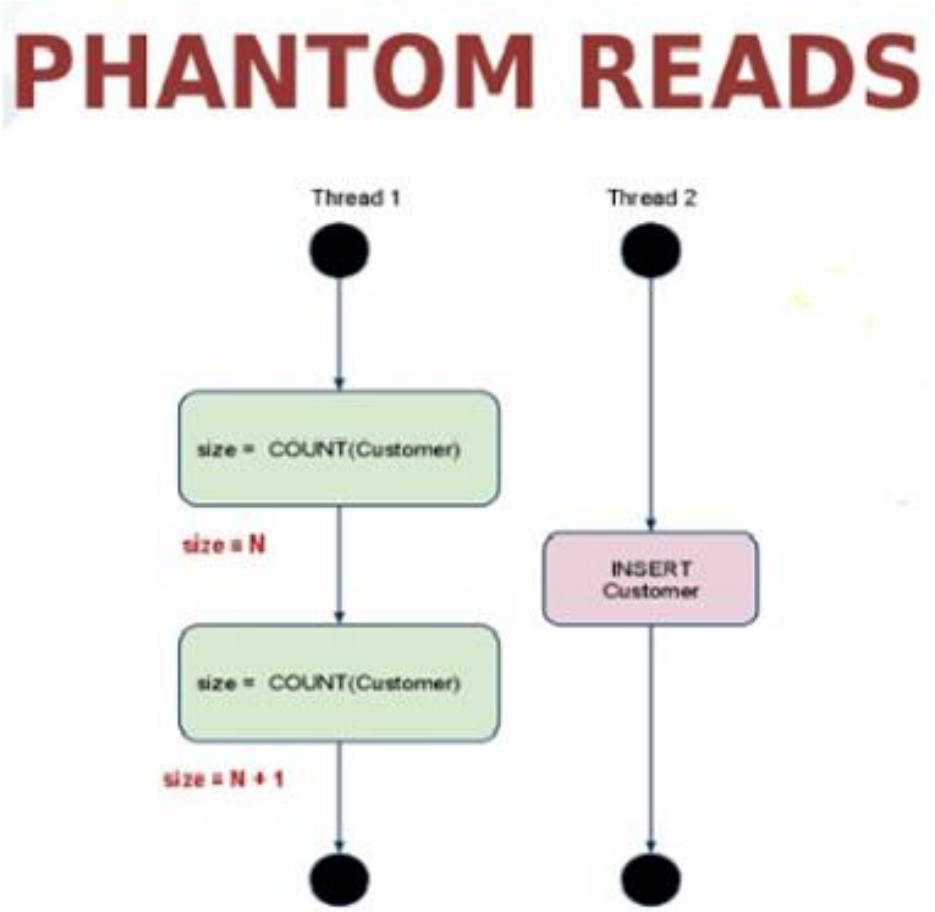
Voorbeeld: dirty read



Voorbeeld: non repeatable read



Voorbeeld: phantom read



Transaction isolation levels

Om dit soort fouten te voorkomen kunnen we records in mindere of meerdere mate „locken". Dit gebeurt door een zogenaamd isolationlevel in te stellen. We onderscheiden **5 isolationlevels** van geen record-locking tot zware record-locking :

Level	Dirty read	Non-repeatable read	Phantom read
TRANSACTION_NONE	**Geen transacties**		
TRANSACTION_READ_UNCOMMITTED	☺	☺	☺
TRANSACTION_READ_COMMITED	✗	☺	☺
TRANSACTION_REPEATABLE_READ	✗	✗	☺
TRANSACTION_SERIALIZABLE	✗	✗	✗

Het isolationlevel wordt ingesteld met de method setTransactionIsolation() van de Connection. Voorbeeld :

```
databaseConnection.setTransactionIsolation(
Connection.TRANSACTION_READ_COMMITTED);
```

Batch updates

Batch updates

= meerdere statements gegroepeerd ter uitvoering doorsturen naar de database (efficiëntie!)

WERKING:

1. **Statement of prepared statement aanmaken**.
2. met de method **addBatch()** SQL-commando's toevoegen. **Opgelet enkel mogelijk bij die een int als resultaat hebben:** inserts, delete commando's, updates,...
3. opstarten met de method **executeBatch()**

→ Gaat door als commando mislukt.

Batch updates

VOORBEELD:

```
con.setAutoCommit( false );
statement = con.createStatement( );

statement.addBatch("insert into…. values(...)");
statement.addBatch("insert into…. values(...)");
statement.addBatch("insert into…. values(...)");
...

// De methode executeBatch stuurt een lijst terug met alle returnwaarden
// van bovenstaande queries in de volgorde van uitvoering:
int [ ] updateCounts = statement.executeBatch( );

System.out.println("Alle " + updateCounts.length + " zijn gelukt");
con.commit( );
```

Oefening: transactions & batch updates

Failliet

Brouwer 1 gaat failliet.

Zijn bieren met een alcohol% vanaf 7.5 worden overgenomen door brouwer 2.

Zijn andere bieren worden overgenomen door brouwer 3.

Schrijf een applicatie die de nodige bewerkingen doet in de database. Op het einde verwijdert de applicatie brouwer 1.

Doe al deze bewerkingen binnen één transactie.

→ Toon hoeveel resultaten er veranderd zijn.

Metadata

ResultsetMetaData

Bevat data over het ResultSet object.
BV: aantal rijen, aantal kolommen, naam van de kolommen enz …

→ **getMetaData()**

Methods van ResultSetMetaData:

- **int getColumnCount()** : geeft ons het aantal kolommen in de resultset.
- **String getColumnName(int kolom)** : geeft de naam van de kolom
- **String getColumnClassName(int kolom)** : geeft het type van de kolom als string, bijvoorbeeld java.lang.String, java.lang.Integer,…
- **int getColumnType(int kolom)** : geeft eveneens het type van de kolom maar als int, bijvoorbeeld 4 voor een Integer, 12 voor een String, 2 voor een BigDecimal, 7 voor een Float, 93 voor een TimeStamp, …

Resultsetmetadata

VOORBEELD:

```
ResultSetMetaData rsmd = resultset.getMetaData();

for (int i=1; i<=rsmd.getColumnCount( ); i++) {
     System.out.println(i + " "
                    + rsmd.getColumnName(i) + " "
                    + rsmd.getColumnType(i) + " "
                    + rsmd.getColumnClassName(i));
}
```

UITVOER :

```
1 LevNr 4 java.lang.Integer
2 Naam 12 java.lang.String
3 Adres 12 java.lang.String
4 PostNr 12 java.lang.String
5 Woonplaats 12 java.lang.String
```

DatabaseMetaData

Waarom?
Bevat data over de database zelf: aanwezige tabellen, versie database en driver …

Hoe?
getMetaData van de interface Connection

Voorbeeld:

```
DatabaseMetaData dMeta = connection.getMetaData();
System.out.println(dMeta.getDriverName());
System.out.println(dMeta.getURL());
System.out.println(dMeta.getUserName());
```

Oefeningen

Opdracht 8, 9 en 10

Grote Objecten

Grote objecten

Opslaan van grote objecten:

- Volledige documenten
- Foto's
- Geluid-en beeldfragmenten

→ Aangepaste SQL datatypes: CLOB (Character Large Object) , BLOB (Binary Large Object) enz…

Demo Logo uploaden

1. Logo downloaden van het Internet
2. Logo in project plaatsen
3. Nieuwe klasse aanmaken: DemoLogo.java
4. Code: Logo in database plaatsen bij een bepaald bier.

DAO

DAO

= Data Access Object

→ Code en configuratie voor de toegang tot de databank.
→ BV: Methodes om elementen toe te voegen, te verwijderen, te updaten … in een tabel

Demo DAO

1. Klasse Beer maken

2. Klasse BeerDao → deze klasse maakt connectie met de databank

3. Klasse BeerClient → Applicatie om alles uit te testen.