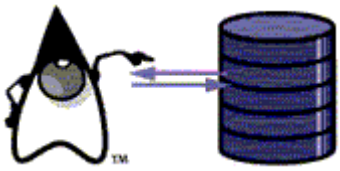
VDAB

samen sterk voor werk

Java

JDBC



Deze cursus is eigendom van VDAB Competentiecentra ©

INHOUD

1 Inleiding

1.1 Doelstelling

Je spreekt met JDBC (Java Database Connectivity) vanuit Java code een relationele database aan.

1.2 Vereiste voorkennis

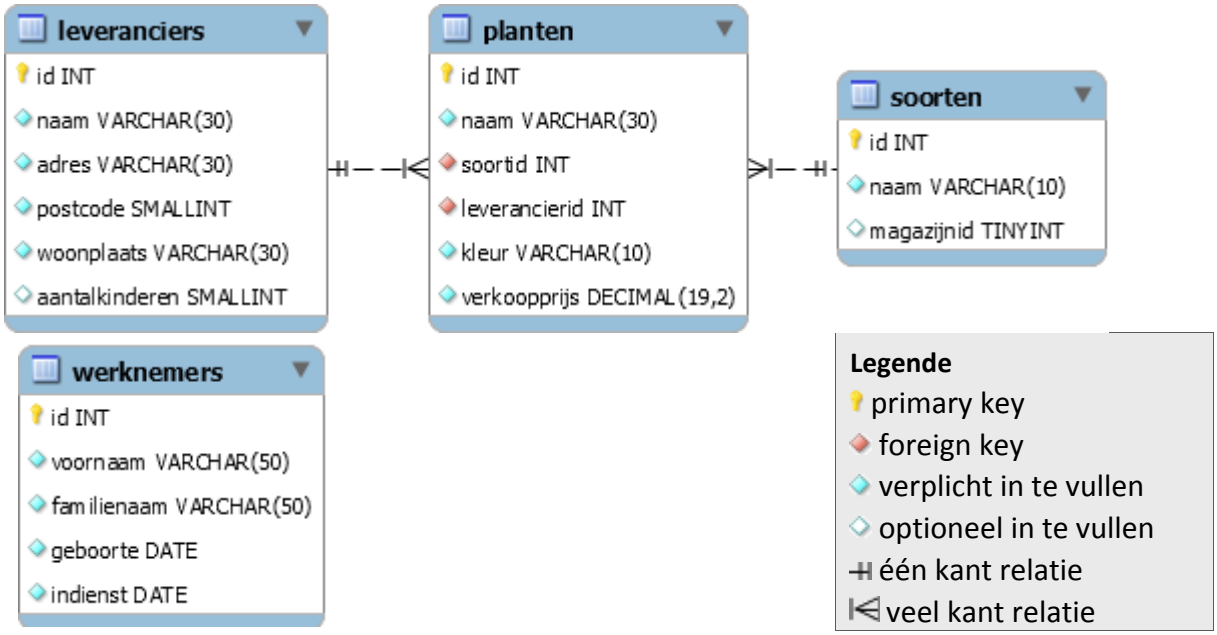
- Java Programming Fundamentals
- SQL

1.3 Nodige software

- een JDK (Java Developer Kit) met versie 7 of hoger
- een MySQL database server
- MySQL Workbench
- NetBeans

1.4 De voorbeelddatabases

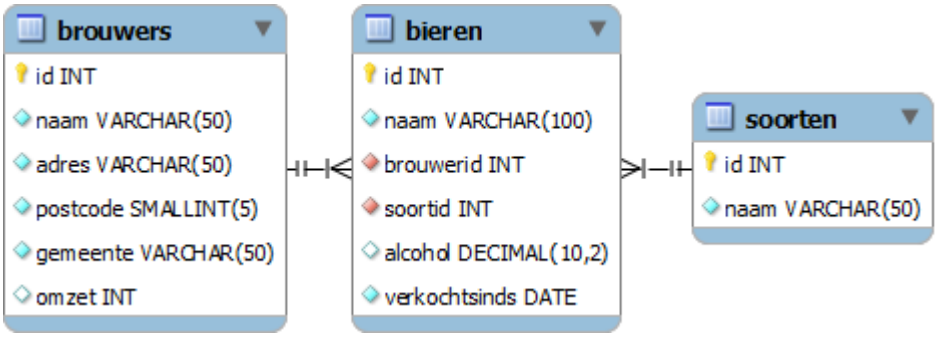
Je gebruikt in de theorie de database `tuincentrum`. Deze heeft volgende structuur



Je maakt deze database met het script `tuincentrum.sql` uit het materiaal bij de cursus

1. Je start de MySQL Workbench en je logt in op de `Local instance`
2. Je kiest in het menu `File` de opdracht `Open SQL Script` en je opent `tuincentrum.sql`
3. Je voert dit script uit met de knop 

Je gebruikt In de taken de database `bieren`. Je maakt deze database met het script `bieren.sql`




Tip: Je kan zelf dit schema maken via de opdracht `Reverse Engineer` in het menu `Database` van de MySQL workbench.

2 JDBC driver - Connection

2.1 JDBC driver

Je kan met JDBC elk relationeel database merk aanspreken (MySQL, PostgreSQL, Oracle, …)

Je hebt per merk een JDBC driver nodig. Dit is een Java library, verpakt als een JAR bestand.
De classes in dit bestand zijn specifiek voor één databasemerk.

Je downloadt de JDBC driver die hoort bij MySQL

1. Je opent in de browser dev.mysql.com/downloads
2. Je kiest `MySQL Connectors`
3. Je kiest `Connector/J`
4. Je kiest `Platform Independent` bij `Select platform`
5. Je kiest `Download` bij `Platform Independent (Architecture Independent),ZIP Archive`
6. Je kiest `No thanks, just start my download.`
7. Je opent het ZIP bestand en de map binnen het ZIP bestand
8. Je extract het JAR bestand en plaatst dit ergens op de computer

2.2 Het project

Je maakt in NetBeans een Java project.

Je voegt aan dit project een verwijzing toe naar de JDBC driver

1. Je klikt met de rechtermuisknop op het project onderdeel `Libraries`
2. Je kiest `Add JAR/Folder`
3. Je duidt het JAR bestand aan dat je in de vorige paragraaf op de computer plaatste

Je ziet in het project onderdeel `Libraries` een vermelding van dit JAR bestand.

Je voegt in elk ander project waarin je een database aanspreekt eenzelfde verwijzing toe.

2.3 Connection

Je hebt een databaseverbinding nodig om vanuit Java code de database aan te spreken.

De interface `java.sql.Connection` stelt zo'n databaseverbinding voor.
Elke JDBC driver bevat een class die deze interface implementeert.

Je verkrijgt een `Connection` object (een object waarvan de class de interface `Connection` implementeert) van de static method `getConnection` van de class `DriverManager`.
Deze mehod zoekt in de JDBC driver de class die de interface `Connection` implementeert, maakt een object van deze class en geeft dit object als returnwaarde terug.

De method `getConnection` heeft drie parameters

- Een JDBC URL. Dit is een String met de naam en de locatie van de te openen database.
  - De String begint altijd met `jdbc:`
  - Bij MySQL komt hierna `mysql://`
  - Bij MySQL komt hierna de netwerknaam van de computer waarop MySQL draait. Als dit de computer is waarop je werkt, is de netwerknaam `localhost`
  - Bij MySQL komt hierna een `/` en de naam van de te openen database, dus dit wordt `jdbc:mysql//localhost/tuincentrum`

  Bij een andere JDBC driver lees je in zijn documentatie de opbouw van de JDBC URL.
- Een gebruikersnaam (gedefinieerd in de database) waarmee je de connectie maakt
- Het paswoord dat bij deze gebruikersnaam hoort

Het volledige programma

```
import java.sql.Connection;                                              (1)
import java.sql.DriverManager;
import java.sql.SQLException;
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "root";
  private static final String PASSWORD = "vdab";
  public static void main(String[] args) {
    Connection connection = null;
    try {
      connection = DriverManager.getConnection(URL, USER, PASSWORD);     (2)
      System.out.println("Connectie geopend");
    } catch (SQLException ex) {                                          (3)
      ex.printStackTrace();
    } finally {
      if (connection != null) {
        try {
          connection.close();                                            (4)
        } catch (SQLException ex) {                                      (5)
          ex.printStackTrace();
        }
      }
    }
  }
}
```

(1) De packages `java.sql` en `javax.sql` bevatten de JDBC classes en interfaces.
Ook andere packages bevatten een interface `Connection`, maar dat is geen JDBC `Connection`. De package `com.mysql.jdbc` bevat ook een interface `Connection`.
Deze stelt een connectie voor naar MySQL, niet naar andere databasemerken.
We gebruiken deze interface niet: zo werkt ons programma met alle merken databases.

(2) Je maakt een verbinding met de database `tuincentrum` op de eigen computer.
Je verbindt met de gebruikersnaam `root` en het bijbehorende paswoord `vdab`

(3) De verbinding is mislukt (redenen: tikfout in de JDBC URL, MySQL is niet gestart, de database `tuincentrum` bestaat niet, verkeerd paswoord, …) JDBC werpt dan een `SQLException`.

(4) Je sluit de verbinding na gebruik. Als je de verbinding niet sluit zal na een tijdje ofwel het programma, ofwel MySQL fouten veroorzaken wegens te veel openstaande verbindingen.

(5) Ook het sluiten van een verbinding kan een exception werpen.

Je kan de applicatie uitproberen.

2.4 Try with resources

Vanaf Java 7 kan je een variabele, waarvan de class de interface `AutoCloseable` implementeert, declareren en initialiseren binnen ronde haakjes bij de `try` opdracht. Dit heet “try-with-resources”.

De compiler voegt dan een `finally` blok toe aan dit `try` blok waarin het object wordt gesloten.
Jij hoeft dit `finally` blok dus niet meer zelf te schrijven.

Gezien de interface `Connection` erft van `AutoCloseable` kan de code in de method `main` korter:

```
try (Connection connection=DriverManager.getConnection(URL,USER,PASSWORD)) {  (1)
  System.out.println("Connectie geopend");
} catch (SQLException ex) {
  ex.printStackTrace();
}                                                                        (2)
```

(1) Je declareert en initialiseert de variabele binnen de ronde haakjes van de `try` opdracht.

(2) De compiler voegt hier een `finally` blok toe waarin `connection` gesloten wordt.

Je kan de applicatie uitproberen.

Tip: Veel applicaties kunnen tegelijk een database aanspreken.
Als ze hun connectie lang openhouden, heeft de database veel connecties tegelijk open.
Dit benadeelt de database performantie. Het is dus belangrijk de connectie
zo kort mogelijk open te houden. Je vraagt gebruikersinvoer bijvoorbeeld
voor het openen van de connectie, niet terwijl de connectie open staat.

2.4.1 TCP/IP poortnummer

Je programma communiceert met MySQL via het TCP/IP protocol.

Op één computer kunnen meerdere programma's het TCP/IP protocol gebruiken.
Elk programma krijgt bij TCP/IP een uniek identificatiegetal: het poortnummer.

- Webservers gebruiken standaard poort nummer 80.
- Mail servers gebruiken standaard poort nummer 25.
- MySQL gebruikt standaard poort nummer 3306.

Als de MySQL waarmee je wil verbinden een ander poort nummer gebruikt dan 3306,
moet je het poort nummer vermelden in de JDBC URL. Bij poort nummer 3307 is de JDBC URL
`jdbc:mysql://localhost:3307/tuincentrum`

2.4.2 De databasegebruiker

Het is een slechte gewoonte om een databaseverbinding te maken met de gebruiker root.
Als een hacker het bijbehorende paswoord ontdekt waarmee de applicatie verbinding
maakt, kan hij in alle databases schade aanrichten: de gebruiker root heeft alle rechten.

Je maakt beter verbinding met een gebruiker die enkel rechten heeft in de database `tuincentrum`.

Je voert eerst volgende opdrachten uit in de `MySQL Workbench`

```
create user cursist identified by 'cursist'
```

Je maakt hiermee een gebruiker met de naam `cursist` en het paswoord `cursist`
Deze gebruiker heeft nog geen rechten.

```
grant all on tuincentrum.* to cursist
```

Je geeft deze gebruiker rechten op alle tables in de database `tuincentrum`.

Je vervangt in de Java code `vdab` en `root` door `cursist`.

Je kan de applicatie terug uitproberen.

2.5 Samenvatting


DriverManager maakt 
Connection naar 
Database

3 Records toevoegen, wijzigen of verwijderen met Statement

De interface `Statement` stelt een SQL statement voor dat je vanuit Java naar de database stuurt.

Elke JDBC driver bevat een class die deze interface implementeert.
De `Connection` method `createStatement` geeft je een `Statement`.

Je voert een `insert`, `update` of `delete` statement uit met de `Statement` method `executeUpdate`.
Je geeft als parameter een `String` mee met het uit te voeren SQL statement.
De method voert dit statement uit en geeft daarna een `int` terug. Deze bevat bij een

- `insert` statement het aantal toegevoegde records
- `update` statement het aantal gewijzigde records
- `delete` statement het aantal verwijderde records

Voorbeeld: je verhoogt de verkoopprijzen van alle planten met 10%

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String UPDATE_SQL =
    "update planten set verkoopprijs = verkoopprijs * 1.1";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement statement = connection.createStatement()) {            (1)
      System.out.println(statement.executeUpdate(UPDATE_SQL));            (2)
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) De `Connection` method `createStatement` geeft je een `Statement` object.
Je moet dit object na gebruik sluiten. `Statement` erft van `AutoCloseable`.
Je maakt het object tussen ronde haakjes bij de `try` opdracht.
De compiler maakt dan de code om het object te sluiten.

(2) Je geeft aan de method `executeUpdate` een SQL statement mee.
De method voert dit SQL statement uit en geeft het aantal aangepaste records terug.

Je kan de applicatie uitproberen en dan met de `MySQL Workbench` de aangepaste prijzen zien.

3.1 Samenvatting

DriverManager — Connection — Statement

Bieren verwijderen: zie takenbundel

4 Records lezen met ResultSet

Je voert met de `Statement` method `executeQuery` een `select` statement uit.
Je geeft als parameter een String met het uit te voeren `select` statement mee.
De method voert dit statement uit en geeft daarna een object terug
dat de interface `ResultSet` implementeert.

`ResultSet` stelt de rijen voor die het resultaat zijn van het `select` statement.

```
select id, naam from leveranciers
where woonplaats = 'kortrijk'
order by id
```

ResultSet

2	Baumgarten
7	Bloem

Je benadert die rijen één per één, door over de rijen te itereren. Je staat initieel voor de eerste rij.
Je plaatst je op een volgende rij met de `ResultSet` method `next`.
Deze geeft `true` terug als er een volgende rij was, of `false` als er geen volgende rij was
(dit is het geval als je op de laatste rij staat en de `next` method uitvoert).

Je staat bij bovenstaand voorbeeld initieel ook voor de eerste rij.

- Je voert de method `next` uit. Die plaatst je op de eerste rij (`Baumgarten`) en geeft `true` terug.
- Je voert de method `next` uit. Die plaatst je op de tweede rij (`Bloem`) en geeft `true` terug.
- Je voert de method `next` uit. Die geeft `false` terug. Je hebt dus alle rijen gelezen.

Deze opeenvolging van opdrachten:

```
while (resultSet.next()) {
  // lees de kolomwaarden in de rij waarop je nu gepositioneerd bent
}
```

Als je op een rij staat, kan je zijn kolomwaarden lezen. Je kan op 2 manieren een kolom aanduiden

- met het volgnummer van de kolom in het `select` statement (nummering vanaf 1)
- met de kolomnaam (`"id"` of `"naam"`)

4.1 Kolommen aanduiden met hun volgnummer

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SELECT_SQL =
    "select id, naam from leveranciers order by id";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(SELECT_SQL)) {       (1)
      while (resultSet.next()) {                                           (2)
        System.out.printf("%4d %s%n", resultSet.getInt(1),                 (3)
          resultSet.getString(2));                                         (4)
      }
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
  }
}
```

(1) De `Statement` method `executeQuery` geeft een `ResultSet` object.
Je moet dit object na gebruik sluiten. `ResultSet` erft van `AutoCloseable`.
Je maakt het object tussen ronde haakjes bij de `try` opdracht.
De compiler maakt dan de code om het object te sluiten.
(2) Je itereert over de `ResultSet` rijen
(3) Je leest met de method `getInt` de int waarde in de eerste kolom (`id`)
(4) Je leest met de method `getString` de `String` waarde in de tweede kolom (`naam`)

Je kan de applicatie uitproberen.

Kolommen aanduiden met hun volgnummer heeft nadelen

- Als je het SQL statement wijzigt of uitbreidt, kunnen de kolomvolgnummers wijzigen.
  Als je het voorbeeld SQL statement wijzigt naar
  `select naam, id from leveranciers order by id`
  moet je ook `getInt(1)` vervangen door `getInt(2)` en `getString(2)` door `getString(1)`
- Als het SQL statement veel kolommen bevat, vermindert de leesbaarheid van de code.
  Bij `getString(13)` moet je visueel in het SQL statement tellen om welke kolom dit gaat.

4.2 Kolommen aanduiden met hun naam

Je vermijdt bovenstaande nadelen door de kolommen aan te duiden met hun naam.

Je wijzigt in het voorbeeldprogramma de opdracht `System.out.printf`

```
System.out.printf("%4d %s%n", resultSet.getInt("id"),                      (1)
  resultSet.getString("naam"));                                            (2)
```

(1) Je leest de `int` waarde in de kolom `id`
(2) Je leest de `String` waarde van de kolom `naam`

Je kan de applicatie uitproberen.

Naast `getInt` en `getString` kan je met volgende `ResultSet` methods een kolomwaarde lezen

- `getByte` een getalwaarde lezen als een `byte`
- `getShort` een getalwaarde lezen als een `short`
- `getLong` een getalwaarde lezen als een `long`
- `getFloat` een getalwaarde lezen als een `float`
- `getDouble` een getalwaarde lezen als een `double`
- `getBigDecimal` een getalwaarde lezen als een `BigDecimal`
  Je gebruikt deze method meestal bij het kolomtype `decimal`
- `getBoolean` een waarde lezen als een `boolean`
  Je gebruikt deze method meestal bij de kolomtypes `boolean` en `bit`
- `getDate` een waarde met een datum of datum+tijd lezen als een `java.util.Date`
  Als de waarde een datum+tijd bevat, lees je enkel de datum
  Je gebruikt deze method meestal bij het kolomtype `date`
  De method geeft je de datum als een `java.sql.Date`. Deze class erft van `java.util.Date`. Gezien `java.util.Date` een handiger class is dan `java.sql.Date`, kan je het resultaat van `getDate` beter onthouden in een `java.util.Date` variabele.
- getTime een waarde met een tijd of datum+tijd lezen als een java.util.Date
  als de waarde een datum+tijd bevat, lees je enkel de tijd
  Je gebruikt deze method meestal bij het kolomtype `time`
- getTimestamp een waarde met een datum+tijd lezen als een Date
  je leest de datum én de tijd
  Je gebruikt deze method meestal bij het kolomtype `datetime`

Als het `select` statement berekende kolommen bevat, geef je deze kolommen een alias.
Je leest dan deze kolomwaarden in Java met die alias. Voorbeeld

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SELECT_SQL =
   "select avg(verkoopprijs) as gemiddelde from planten";                (1)
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(SELECT_SQL)) {
      if (resultSet.next()) {                                            (2)
        System.out.println(resultSet.getBigDecimal("gemiddelde"));       (3)
      }
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) Je geeft de berekening als alias `gemiddelde`
(2) Deze `ResultSet` bevat maximaal één record. Je geeft dit duidelijk aan door een `if` te gebruiken in de plaats van een `while` bij het verwerken van de `ResultSet`.
(3) Je leest de inhoud van de kolom met de alias `gemiddelde`.

Je kan de applicatie uitproberen.

Als het `select` statement kolommen met dezelfde naam leest uit verschillende tables,
geef je die kolommen elk een eigen alias. Je kan ze zo van mekaar onderscheiden in Java code

```
select planten.naam as plantnaam, soorten.naam as soortnaam
from planten inner join soorten on planten.soortid = soorten.id
```

4.3 select *

Het is een slechte gewoonte om in een `select` statement alle kolommen te lezen met *
Voorbeeld `select * from planten order by id`

Wanneer de applicatie in productie gaat, leest dit statement alle kolommen uit de table `planten`:
`id`, `naam`, `soortid`, `leverancierid`, `kleur`, `verkoopprijs`.

Dit geeft op termijn problemen

- Later worden aan de table `planten` kolommen toegevoegd die jouw applicatie niet gebruikt, maar wel andere applicaties. Toch leest je applicatie alle kolommen, ook de kolommen die je applicatie niet nodig heeft. Je applicatie werkt trager en trager, zeker als één van die kolommen veel bytes bevat, zoals een afbeelding.
- Later worden aan de table `planten` kolommen toegevoegd met informatie die slechts enkele applicaties mogen lezen, zoals een kolom met de winst per plant. Je kan in de database rechten geven zodat de gebruiker `root` die kolom wel mag lezen, maar de gebruiker `cursist` niet. Zodra de kolom met deze rechten toegevoegd is, werpt je applicatie een exceptie. Je applicatie vraagt met select * alle kolommen, ook de kolom waarop ze geen rechten heeft.

Je voorkomt deze problemen door in je `select` statement enkel de nodige kolommen te vragen.

4.4 Null values

De kolom `aantalkinderen` in de table `leveranciers` kan `null` values bevatten.

- De kolom bevat `null` als het aantal kinderen van een leverancier onbekend is
- De kolom bevat `0` als een leverancier geen kinderen heeft

De `ResultSet` method `getInt("aantalkinderen")` geeft echter `0`,
zowel als de kolom `0` bevat, maar ook als de kolom `null` bevat.
Je kan zo geen onderscheid maken tussen deze twee mogelijkheden.

Je maakt het onderscheid met de `ResultSet` method `wasNull`.
Als de laatst gelezen kolom `null` bevat geeft de method `true` terug, anders geeft ze `false` terug.

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SELECT_SQL =
   "select naam, aantalkinderen from leveranciers order by naam";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(SELECT_SQL)) {
      while (resultSet.next()) {
        System.out.print(resultSet.getString("naam") + ' ');
        int aantalKinderen = resultSet.getInt("aantalkinderen");          (1)
        System.out.println(resultSet.wasNull()?"onbekend":aantalKinderen); (2)
      }
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) Je leest de waarde uit de kolom `aantalkinderen`.
(2) Je controleert of de laatst gelezen kolom (`aantalkinderen`) `null` bevatte.

Je kan de applicatie uitproberen.

Tip: de `ResultSet` methods `getString`, `getBigDecimal`, `getDate`, `getTime` en `getTimestamp` geven `null` terug als je er een `null` value mee leest.
Je moet bij die methods de controle met `wasNull` dus niet doen.

4.5 Soorten ResultSets

Een `ResultSet` heeft twee eigenschappen

- een eigenschap met de waarde Forward-only of Scrollable
- een eigenschap met de waarde Read-only of Updatable

Een `ResultSet` is standaard

- Forward-only
  Je kan enkel van voor naar achter door de `ResultSet` rijen itereren
- Read-only
  Je kan de `ResultSet` waarden enkel lezen, niet wijzigen

De `ResultSets` die je tot nu maakt zijn standaard `ResultSets`

Je wijzigt deze eigenschappen met extra parameters van de `Statement` method `executeQuery`

- Je kan een scrollable `ResultSet` maken.
  Je kan in zo'n `ResultSet` de rijen van voor naar achter lezen, maar ook van achter naar voor.
  Je kan ook naar een rij met een bepaald volgnummer springen.
  Men gebruikt zelden een scrollable `ResultSet`
  - Sommige JDBC drivers ondersteunen geen scrollable `ResultSets`
  - Scrollable `ResultSets` gebruiken meestal meer RAM dan Forward-Only `ResultSets`
  - Scrollable `ResultSets` zijn meestal trager dan Forward-Only `ResultSets`

  Je gebruikt daarom in deze cursus geen scrollable `ResultSets`
- Je kan een updatable `ResultSet` maken.
  Als je een waarde wijzigt in een Updatable `ResultSet`, wijzigt JDBC die waarde ook in de database. Als je een rij toevoegt aan de `ResultSet`, voegt JDBC ook een record toe aan de database. Als je een rij verwijdert uit de `ResultSet`, verwijdert JDBC ook het bijbehorende record uit de database.
  Men gebruikt zelden updatable ResultSet
  - Sommige JDBC drivers ondersteunen geen Updatable `ResultSets`
  - Je kan van sommige SQL `select` statements geen updatable `ResultSet` maken

  Je gebruikt daarom in deze cursus geen updatable `ResultSets`

4.6 Samenvatting


DriverManager maakt 
Connection maakt 
Statement maakt 
ResultSet

Aantal bieren per brouwer: zie takenbundel

5 Parameters in SQL statements en PreparedStatement

Veel SQL statements bevatten veranderlijke onderdelen. Deze wijzigen bij elke uitvoering.

De gebruiker tikt in een voorbeeldapplicatie een woonplaats. Je toont de leveranciers uit die woonplaats. Deze woonplaats wordt het veranderlijk onderdeel van het SQL statement

De gebruiker tikt	Het SQL statement wordt
Kortrijk	`select naam from leveranciers where woonplaats = 'Kortrijk'`
Menen	`select naam from leveranciers where woonplaats = 'Menen'`

Je maakt een SQL statement met veranderlijke onderdelen in volgende stappen

1. Je stelt elk veranderlijk onderdeel in het SQL statement voor met een **?**
   `select naam from leveranciers where woonplaats = ?`
   Zo'n ? heet een parameter. Als ? tekst voorstelt, moet je rond ? geen quotes schrijven.
   Je kan meerdere veranderlijke onderdelen voorstellen met meerdere vraagtekens.
2. Je geeft dit SQL statement mee aan een `PreparedStatement` object (erft van `Statement`).
3. Je vult de parameter ? met `Kortrijk` met de method `setString(1, "Kortrijk");`
   Dit betekent: vul de 1° parameter met de waarde `Kortrijk`.
   Naast `setString` bestaan ook `setInt`, `setBigDecimal`, …
4. Je voert het `PreparedStatement` uit.

5.1 Voorbeeld

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SELECT_SQL =
    "select naam from leveranciers where woonplaats = ?";
  public static void main(String[] args) {
    try (Scanner scanner = new Scanner(System.in)) {                    (1)
    System.out.print("Woonplaats:");
    String woonplaats = scanner.nextLine();
      try (Connection connection=DriverManager.getConnection(URL, USER, PASSWORD);
        PreparedStatement statement=connection.prepareStatement(SELECT_SQL)) {(2)
        statement.setString(1, woonplaats);                             (3)
        try (ResultSet resultSet = statement.executeQuery()) {          (4)
          while (resultSet.next()) {
            System.out.println(resultSet.getString("naam"));
          }
        }
      } catch (SQLException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

(1) `Scanner` implementeert ook de interface `AutoCloseable`. Tip: je mag een `Scanner(System.in)` slechts één keer openen en sluiten. Zoniet krijg je een exception.
(2) Je geeft aan de `Connection` method `prepareStatement` een SQL statement mee. Je krijgt een `PreparedStatement` object terug.
(3) Je vult de eerste (en hier enige) parameter met de waarde die de gebruiker intikte.
(4) Je voert het SQL statement uit dat je bij (1) meegaf.

Je kan de applicatie uitproberen.

5.2 SQL code injection

Het is een slechte gewoonte om een SQL statement met parameters te vervangen door een SQL statement waarin je stukken SQL concateneert met gebruikersinvoer:

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  public static void main(String[] args) {
    try (Scanner scanner = new Scanner(System.in)) {
      System.out.print("Naam:");
      String naam = scanner.nextLine();
      try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD);
           Statement statement = connection.createStatement()) {
        System.out.println(statement.executeUpdate(
          "update planten set verkoopprijs = verkoopprijs * 1.1 where naam = '"
           + naam + "'"));                                                    (1)
      } catch (SQLException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

(1) Je concateneert stukken SQL met de gebruikersinvoer tot een SQL statement

Als de gebruiker Linde intikt, is er geen probleem: enkel de prijs van Linde wordt gewijzigd.

Een hacker tikt geen Linde, maar `' or ''='`
De database verwerkt dan volgend statement en wijzigt ongewenst alle plantenprijzen !
`update planten set verkoopprijs=verkoopprijs*1.1 where naam='' or ''=''`
Deze hack heet SQL code injection: de hacker tikt SQL code waar jij dit niet verwacht had.

Je vermijdt dit probleem met een PreparedStatement. Dit verhindert intern SQL code injection.

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String UPDATE_SQL =
    "update planten set verkoopprijs = verkoopprijs * 1.1 where naam=?";
  public static void main(String[] args) {
    try (Scanner scanner = new Scanner(System.in)) {
      System.out.print("Naam:");
      String naam = scanner.nextLine();
      try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD);
           PreparedStatement statement = connection.prepareStatement(UPDATE_SQL)) {
        statement.setString(1, naam);
        System.out.println(statement.executeUpdate());
      } catch (SQLException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Bieren van tot alcohol: zie takenbundel

6 Metadata

Metadata zijn data die eigenschappen van andere data beschrijven.

Je kan bij JDBC metadata opvragen over de JDBC driver, de database of een `ResultSet`.

6.1 Metadata over de JDBC driver

Voorbeeld: de naam en het versienummer van de JDBC driver ophalen

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD)) {
      DatabaseMetaData metaData = connection.getMetaData();                (1)
      System.out.printf("%s %d.%d%n", metaData.getDriverName(),           (2)
        metaData.getDriverMajorVersion(),
        metaData.getDriverMinorVersion());
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) De method `getMetaData` geeft metadata over de JDBC driver en over de database
(2) Je leest de naam en de versienummers van de JDBC driver

Je kan de applicatie uitproberen.

6.2 Metadata over de database

Voorbeeld: de naam en het versienummer van de database engine ophalen

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD)) {
      DatabaseMetaData metaData = connection.getMetaData();
      System.out.println(metaData.getDatabaseProductName() + ' ' +
                         metaData.getDatabaseMajorVersion() + ' ' +
                         metaData.getDatabaseMinorVersion());
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

Je kan de applicatie uitproberen.

6.3 Metadata over een ResultSet

Je kan van een `ResultSet` het aantal kolommen vragen, per kolom de naam en het type vragen …

Voorbeeld

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SQL_SELECT =
    "select id, voornaam, indienst from werknemers";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD);
         Statement statement = connection.createStatement()) {
      try (ResultSet resultSet = statement.executeQuery(SQL_SELECT)) {
        ResultSetMetaData metaData = resultSet.getMetaData();              (1)
        for (int index = 1; index <= metaData.getColumnCount(); index++) { (2)
          System.out.printf("%s %s%n", metaData.getColumnName(index),     (3)
            metaData.getColumnTypeName(index));                            (4)
          }
      }
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) De method `getMetaData` geeft metadata over de `ResultSet`
(2) Je leest het aantal kolommen in de `ResultSet`
(3) Je leest de naam van de kolom met een bepaald volgnummer
(4) Je leest het type van de kolom met een bepaald volgnummer

Je kan de applicatie uitproberen.

Opmerking: je hebt zelden metadata nodig

7 Stored procedures en CallableStatement

Een stored procedure is een verzameling van één of meerdere SQL statements,
die onder een naam is opgeslagen in de database.

Je kan vanuit je applicatie met die naam de stored procedure oproepen.
De database voert dan de SQL statements uit die beschreven zijn in de stored procedure.

Een stored procedure kan parameters bevatten.
Je geeft waarden voor deze parameters mee bij het oproepen van de stored procedure.

7.1 Een stored procedure aanmaken en uitproberen

Je maakt een stored procedure met de naam `PlantenMetEenWoord` in de MySQL Workbench.

1. Je dubbelklikt onder SCHEMAS op de database `tuincentrum`.
2. Je klikt met de rechtermuisknop op `Stored Procedures`
   en je kiest `Create Stored Procedure`
3. Je vervolledigt de code, tot ze er als volgt uitziet:

```
create procedure tuincentrum.PlantenMetEenWoord (woord varchar(50))        (1)
begin                                                                      (2)
select naam from planten where naam like woord order by naam;              (3)
end                                                                        (4)
```

(1) Je maakt een stored procedure met de sleutelwoorden `create procedure`
Je tikt daarna de naam van de database waarin je de stored procedure aanmaakt, een punt en de naam van de stored procedure.
Je tikt daarna tussen ronde haakjes één of meerdere parameters, gescheiden door een ,
Je geeft elke parameter een naam en een type

(2) Je begint de stored procedure met het sleutelwoord `begin`

(3) Je gebruikt de waarde in de parameter `woord`
Je sluit elk SQL statement af met een **;**

(4) Je eindigt de stored procedure met het sleutelwoord `end`

Je maakt de stored procedure aan met de knoppen `Apply`, `Apply` en `Finish`

Je kan de stored procedure uitproberen met de MySQL Workbench.
Je tikt de opdracht `call tuincentrum.PlantenMetEenWoord('%bloem%')` en je drukt op

7.2 De stored procedure oproepen vanuit Java code met CallableStatement

Je roept een stored procedure op met een `CallableStatement` object.
`CallableStatement` erft van `PreparedStatement`.

```
// enkele imports
class Main {
  private static final String URL =
    "jdbc:mysql://localhost/tuincentrum?noAccessToProcedureBodies=true";   (1)
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String CALL = "{call PlantenMetEenWoord(?)}";
  public static void main(String[] args) {
    try (Scanner scanner = new Scanner(System.in)) {
      System.out.print("Woord:");
      String woord = scanner.nextLine();
      try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD);
           CallableStatement statement = connection.prepareCall(CALL)) {   (2)
        statement.setString(1, '%' + woord + '%');                         (3)
```

```
        try (ResultSet resultSet = statement.executeQuery()) {          (4)
          while (resultSet.next()) {
            System.out.println(resultSet.getString("naam"));
          }
        }
      } catch (SQLException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

(1) Een MySQL gebruiker kan standaard enkel een stored procedure uitvoeren als hij de metadata van alle databases mag lezen. Dit is gevaarlijk als een hacker deze gebruiker (en zijn paswoord) ontdekt. Een gebruiker kan zonder deze rechten een stored procedure uitvoeren als je in de JDBC URL `noAccessToProcedureBodies` op `true` plaatst.

(2) Je geeft aan de `Connection` method `prepareCall` de naam van een stored procedure mee.
- Je tikt `{call` voor die naam
- Je stelt de stored procedure parameter(s) voor met `?`
- Je sluit af met `}`
  Je krijgt een `CallableStatement` object terug.

(3) Je vult de eerste stored procedure parameter in. Je concateneert `%` voor en na het woord dat de gebruiker intikte. Het `like` onderdeel van het `select` statement in de stored procedure wordt dus `'%bloem%'` als de gebruiker `bloem` intikt.

(4) Je voert de stored procedure uit die je bij (2) vermeldde en je krijgt een `ResultSet` terug.

Je kan de applicatie uitproberen.

Voordelen van een stored procedure
- Je kan een SQL statement gemakkelijker over meerdere regels schrijven in een stored procedure dan in Java.
- Als je syntaxfouten tikt, krijg je al een foutmelding bij het opslaan van de stored procedure.
- Je kan een stored procedure uittesten met de MySQL Workbench.
- Een stored procedure kan veel SQL statements bevatten. De database voert deze sneller uit dan dat ze vanuit een applicatie naar de database gestuurd worden.
- Je kan een stored procedure niet enkel vanuit Java oproepen, maar ook vanuit C#, PHP, …

Nadelen van een stored procedure
- Als je verandert van database merk (bijvoorbeeld van MySQL naar Oracle), moet je alle stored procedures herschrijven op de nieuwe database.
- Je kan in een stored procedure ook variabelen, `if` `else` structuren en iteraties gebruiken. Je kan echter geen object oriëntatie gebruiken. Door die beperking vergroot de kans dat een grote stored procedure minder leesbaar en onderhoudbaar is.
- De sleutelwoorden om in een stored procedure variabelen, `if` `else` structuren en iteraties te schrijven, verschillen per databasemerk. Als je verandert van merk, moet je de sleutelwoorden van het nieuwe merk leren kennen.

De meeste Java ontwikkelaars vinden de nadelen van stored procedures belangrijker dan de voordelen. Ze gebruiken daarom zelden stored procedures.

Bieren van tot alcohol 2: zie takenbundel

8 Transacties

Dit is een **belangrijk** hoofdstuk !

Je stuurt in veel applicatie onderdelen meerdere SQL statements naar de database,
die **al** deze statements moet uitvoeren.

Voorbeeld

- Je verhoogt de prijzen van planten met een prijs vanaf € 100 met 10% én
- je verhoogt de prijzen van planten met een prijs onder € 100 met 5%.

Je moet daartoe twee statements naar de database sturen

```
update planten set verkoopprijs = verkoopprijs * 1.1  where verkoopprijs >= 100
update planten set verkoopprijs = verkoopprijs * 1.05 where verkoopprijs < 100
```

De gebruiker start dit applicatie onderdeel. Dit onderdeel stuurt het eerste `update` statement naar de database. Juist daarna (en voor je het tweede `update` statement naar de database stuurt), valt de computer uit of verliest de applicatie zijn netwerkverbinding met de database.
Het tweede `update` statement wordt dus niet uitgevoerd.

De gebruiker heeft nu een foutieve situatie die hij niet kan herstellen.

- ⛔ Als hij het onderdeel niet meer uitvoert, zijn de plantenprijzen onder € 100 niet aangepast.
- ⛔ Als hij het onderdeel nog eens uitvoert, past hij de plantenprijzen vanaf € 100 nog eens aan.

Je vermijdt deze problemen met een transactie. Dit is een groep SQL statements die de database

- ofwel allemaal uitvoert
- ofwel allemaal ongedaan maakt als een probleem voorkomt.

8.1 De autocommit mode

JDBC werkt standaard in autocommit mode. Hierbij is elk individueel SQL statement één transactie.
Je kan zo meerdere SQL statements niet groeperen in één transactie.
De `Connection` method `setAutoCommit(false)` zet de autocommit mode af.
Alle SQL statements die je vanaf dan uitvoert op die `Connection` behoren tot één transactie.

8.2 Commit en rollback

Nadat je die SQL statements hebt uitgevoerd, roep je de `Connection` method `commit` op.
Deze sluit de transactie af. De database legt nu gegarandeerd alle bewerkingen, die SQL statements hebben uitgevoerd, vast in de database.

Als je de `commit` method niet uitvoert, doet de database automatisch een rollback bij het sluiten van de `Connection`. De database maakt daarbij alle bewerkingen,
die de SQL statements binnen de transactie hebben uitgevoerd, ongedaan.

Je kan ook zelf een rollback activeren met de `Connection` method `rollback`.

8.3 Voorbeeld

Dit is het uitgewerkte voorbeeld zoals eerder op deze pagina beschreven.

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SQL_UPDATE_10_PROCENT =
    "update planten set verkoopprijs=verkoopprijs*1.1 where verkoopprijs>=100";
  private static final String SQL_UPDATE__PROCENT =
    "update planten set verkoopprijs=verkoopprijs*1.05 where verkoopprijs < 100";
```

```
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
      Statement statement = connection.createStatement()) {
      connection.setAutoCommit(false);                                    (1)
      statement.executeUpdate(SQL_UPDATE_10_PROCENT);                     (2)
      statement.executeUpdate(SQL_UPDATE_5_PROCENT);                      (3)
      connection.commit();                                                (4)
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) Je zet de autocommit mode af. Alle SQL statements die je vanaf nu op deze `Connection` uitvoert behoren tot één transactie. Dit geldt voor SQL statements die je uitvoert met een `Statement`, een `PreparedStatement` of een `CallableStatement`

(2) De database voert dit statement uit binnen de transactie die je startte bij (1)

(3) De database voert ook dit statement uit binnen dezelfde transactie die je startte bij (1)

(4) Nadat je beide statements kon uitvoeren, doe je een commit. De database legt dan alle bewerkingen die de SQL statement uitvoerden in de database vast. Als je deze regel niet uitvoert, wegens stroompanne of een exception, doet de database automatisch een rollback: de database maakt de bewerkingen die de SQL statements uitvoerden ongedaan.

Je kan de applicatie uitproberen.

Je ziet een rollback aan het werk met volgende handelingen.
Je vervangt in het tweede SQL statement `update` door **o**`pdate.` Je voert de applicatie opnieuw uit. Omdat het uitvoeren van het tweede SQL statement een exception veroorzaakt, voert je applicatie de `commit` method niet uit. De database doet automatisch een rollback en doet dus de bewerkingen van het eerste SQL statement ongedaan.

Als je in een programma onderdeel *meerdere* `insert` `update` en/of `delete` statements uitvoert, is het essentieel dit te doen in een transactie.

8.4 Samenvatting


DriverManager maakt 
Connection maakt 
Transaction verzamelt 
Statements

Failliet: zie takenbundel

8.5 Isolation level

Het transaction isolation level definieert hoe andere gelijktijdige transacties (van andere gebruikers) je huidige transactie beïnvloeden. Volgende problemen kunnen optreden bij gelijktijdige transacties

- **Dirty read**
  Je transactie leest data die een andere transactie schreef, maar nog niet committe.
  Als die transactie een rollback doet, heeft jouw transactie verkeerde data gelezen.
- **Nonrepeatable read**
  Je transactie leest dezelfde data meerdere keren en krijgt per leesopdracht andere data.
  De oorzaak zijn andere transacties die tussen de leesoperaties van jouw transactie dezelfde data wijzigen. Jouw transactie krijgt geen stabiel beeld van de gelezen data.
- **Phantom read**
  Je transactie leest dezelfde data meerdere keren en krijgt per leesopdracht meer records.
  De oorzaak zijn andere transacties die tussen de leesoperaties van jouw transactie records toevoegen. Jouw transactie krijgt geen stabiel beeld van de gelezen data.

Je vermijdt één of enkele van deze problemen door het isolation level van de transactie in te stellen

↓ Isolation level ↓	Dirty read kan optreden	Nonrepeatable read kan optreden	Phantom read kan optreden	Performantie van dit level
Read uncommitted	Ja	Ja	Ja	Snel
Read committed	Nee	Ja	Ja	Trager
Repeatable read	Nee	Nee	Ja	Nog trager
Serializable	Nee	Nee	Nee	Traagst

Het is aanlokkelijk altijd het isolation level `Serializable` te kiezen:
het lost alle problemen op.
`Serializable` is echter het traagste isolation level: de database vergrendelt (lockt) dan de records die je leest in de transactie tot het einde van die transactie.
Andere gebruikers kunnen in de tijd deze records niet wijzigen of verwijderen.
Je kiest `Serializable` enkel als je in je transactie records leest én daarna in diezelfde transactie andere databasebewerkingen op diezelfde records moet doen.

Je moet dus per applicatie onderdeel analyseren welke problemen (dirty read, ...) de goede werking van dat onderdeel benadelen. Daarna kies je een isolation level dat deze problemen oplost.
Je kiest zelden het isolation level read uncommitted, omdat het geen enkel probleem oplost.

Als je het isolation level niet instelt, krijgt de transactie het default isolation level van de database. Dit kan verschillen per merk database. Bij MySQL is dit Repeatable read.

Je stelt het isolation level in met de `Connection` method `setTransactionIsolation`.
Je doet dit voor je de transactie start.

8.5.1 Voorbeeld

De gebruiker tikt een nieuwe verkoopprijs voor een bepaalde plant.
Deze nieuwe prijs mag maximaal 10% hoger zijn dan de huidige prijs.

Je doet dit met volgende stappen:

1. Je laat de gebruiker de id en de nieuwe prijs van de plant intikken
2. Je leest de aan te passen plant in de database
3. Je controleert of de nieuwe prijs maximaal 10% hoger is dan de huidige prijs
   Je wil niet dat een andere gebruiker op dit moment dezelfde plant wijzigt !
   Je neemt daartoe de SQL statements op in een transactie met het isolation level Serializable.
   De database vergrendelt dan bij punt 2 het gelezen record, tot het einde van de transactie.
4. Als de nieuwe prijs max. 10% hoger is dan de huidige prijs, wijzig je de prijs in de database

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SQL_SELECT =
    "select verkoopprijs from planten where id = ?";
  private static final String SQL_UPDATE =
    " update planten set verkoopprijs = ? where id = ?";
  public static void main(String[] args) {
    try (Scanner scanner = new Scanner(System.in)) {
      System.out.print("Id:");
      int id = scanner.nextInt();
      System.out.print("Verkoopprijs:");
      BigDecimal nieuwePrijs = scanner.nextBigDecimal();
```

```
      try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD);
           PreparedStatement statementSelect =
             connection.prepareStatement(SQL_SELECT);
           PreparedStatement statementUpdate =
             connection.prepareStatement(SQL_UPDATE)) {
        statementSelect.setInt(1, id);
        connection.setTransactionIsolation(
          Connection.TRANSACTION_SERIALIZABLE);                          (1)
        connection.setAutoCommit(false);                                 (2)
        try (ResultSet resultSet = statementSelect.executeQuery()) {     (3)
          if (resultSet.next()) {
            BigDecimal oudePrijs = resultSet.getBigDecimal("verkoopprijs");
            if (nieuwePrijs.compareTo(
                oudePrijs.multiply(BigDecimal.valueOf(1.1))) <= 0) {     (4)
              statementUpdate.setBigDecimal(1, nieuwePrijs);
              statementUpdate.setInt(2, id);
              statementUpdate.executeUpdate();                           (5)
              connection.commit();
            } else {
              System.out.println("Nieuwe verkoopprijs te hoog");
            }
          } else {
            System.out.println("Plant niet gevonden");
          }
        }
      } catch (SQLException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

(1) Je stelt het transaction isolation level in op Serializable
De class Connection bevat een constante per isolation level.

(2) Je start de transactie.

(3) Je leest de plant. Als je die vindt, vergrendelt de database die tot het einde van de transactie.
Op die manier kan niemand anders de verkoopprijs wijzigen terwijl jij de verkoopprijs wijzigt.
De database doet deze vergrendeling omdat het isolation level op Serializable staat.

(4) Je controleert of de nieuwe prijs maximaal 10% boven de oude prijs ligt.

(5) Je wijzigt de verkoopprijs in de database.

Je kan de applicatie uitproberen.

Als je in een programma onderdeel records leest én ze in datzelfde onderdeel wijzigt, start je voor de lees operatie een transactie met isolation level serializable, zodat de gelezen records gelocked staan in dat programma onderdeel.
Je verhindert zo dat andere gebruikers tegelijk dezelfde records wijzigen.

8.5.2 Geoptimaliseerde oplossing

Je vindt hier onder voor hetzelfde voorbeeld een performantere oplossing, die niet het trage isolation level `Serializable` gebruikt en meestal maar één SQL statement naar de database stuurt.

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SQL_UPDATE =                               (1)
    "update planten set verkoopprijs = ? where id = ? and ? <= verkoopprijs*1.1";
```

```
  private static final String SQL_SELECT = "select id from planten where id = ?";
  public static void main(String[] args) {
    try (Scanner scanner = new Scanner(System.in)) {
      System.out.print("Id:");
      int id = scanner.nextInt();
      System.out.print("Verkoopprijs:");
      BigDecimal nieuwePrijs = scanner.nextBigDecimal();
      try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD);
           PreparedStatement statementUpdate =
            connection.prepareStatement(SQL_UPDATE)) {
        statementUpdate.setBigDecimal(1, nieuwePrijs);
        statementUpdate.setInt(2, id);
        statementUpdate.setBigDecimal(3, nieuwePrijs);
        connection.setTransactionIsolation(
          Connection.TRANSACTION_READ_COMMITTED);                          (2)
        connection.setAutoCommit(false);
        if (statementUpdate.executeUpdate() == 0) {                        (3)
          try (PreparedStatement statementSelect =
            connection.prepareStatement(SQL_SELECT)) {
            statementSelect.setInt(1, id);
            try (ResultSet resultSet = statementSelect.executeQuery()) {   (4)
              System.out.println(resultSet.next() ?                        (5)
               "Nieuwe verkoopprijs te hoog" : "Plant niet gevonden");
            }
          }
        }
        connection.commit();
      } catch (SQLException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

(1) Dit statement zal de verkoopprijs van een plant wijzigen, op voorwaarde dat een parameter kleiner of gelijk is aan de verkoopprijs *1.1. Je zal deze parameter invullen met de nieuwe prijs. Het update statement zal tijdens zijn uitvoering de plant dus enkel aanpassen als nieuwe prijs maximaal 10% hoger is dan de huidige prijs.

(2) Je plaatst het isolation level op het laagste niveau dat je voor deze code nodig hebt.

(3) Je voert het update statement uit. Je vraagt daarna het aantal aangepaste records.
Als dit 1 is (meestal het geval) , moet je geen ander statement naar de database sturen.
Als dit 0 is (zelden het geval), zijn er twee mogelijkheden

a. de database bevat geen plant met het ingetikte plantnummer
b. de nieuwe verkoopprijs is te hoog

Je onderzoekt welke mogelijkheid is opgetreden.

(4) Je zoekt de plant met het ingetikte plantnummer.

(5) Als je de plant vindt, was de nieuwe verkoopprijs te hoog

(6) Als je de plant niet vindt, bestond de plant niet

Je kan de applicatie uitproberen.

9 Batch updates

JDBC stuurt een SQL statement als een TCP/IP netwerkpakket naar de database.
De database verwerkt dit statement. Hij stuurt daarna een netwerkpakket met het resultaat van de verwerking terug naar de applicatie. 3 SQL statements zijn dus 6 netwerkpakketten.

Je werkt sneller door meerdere statements in één netwerkpakket (batch) naar de database te sturen. De database voert deze statements na mekaar uit en stuurt daarna 1 netwerkpakket terug met de resultaten van de verwerkingen.
3 SQL statements zijn zo maar 2 netwerkpakketten.

- De `Statement` method `addBatch` voegt een SQL statement toe aan een netwerkpakket, maar verstuurt dit netwerkpakket nog niet naar de database.
- De `Statement` method `executeBatch` stuurt dit netwerkpakket naar de database.
  De database voert de statements uit en stuurt daarna één netwerkpakket terug.
  De returnwaarde van de method `executeBatch` is een array met `int` waarden.
  Elke waarde bevat aantal records dat één van de SQL statements bijwerkte.

Een batch heeft volgende beperkingen

- Hij kan geen `select` statement bevatten.
- De statements moeten ofwel allemaal statements zonder parameter zijn, of moeten meerdere uitvoeringen van één SQL statement met parameters zijn.

9.1 Statements zonder parameter

Je gebruikt een batch in het eerste voorbeeld van het hoofdstuk Transacties.

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SQL_UPDATE1 = "update planten set verkoopprijs = "
    + "verkoopprijs * 1.1 where verkoopprijs >= 100";
  private static final String SQL_UPDATE2 = "update planten set verkoopprijs = "
    + " verkoopprijs * 1.05 where verkoopprijs < 100";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
      Statement statement = connection.createStatement()) {
      connection.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITTED);
      connection.setAutoCommit(false);
      statement.addBatch(SQL_UPDATE1);                                    (1)
      statement.addBatch(SQL_UPDATE2);                                    (2)
      int[] aantalGewijzigdeRecordsPerUpdate = statement.executeBatch();  (3)
      System.out.printf("%d planten met 10 %% verhoogd%n",
        aantalGewijzigdeRecordsPerUpdate[0]);
      System.out.printf("%d planten met 5 %% verhoogd%n",
        aantalGewijzigdeRecordsPerUpdate[1]);
      connection.commit();
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) Je voegt het `Statement` toe aan het netwerkpakket. Je doet dit ook op de volgende regel.
(2) Je stuurt de `Statements` als één netwerkpakket naar de database.
De database voert de `update` statements uit en stuurt daarna één netwerkpakket terug.
Dit pakket bevat een array met per `update` statement het aantal gewijzigde records.

9.2 Meerdere uitvoeringen van één SQL statement met parameter(s)

De gebruiker tikt meerdere soortnamen. Je voegt daarmee records toe aan de table soorten.

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String INSERT_SQL = "insert into soorten(naam) values (?)";
  public static void main(String[] args) {
    List<String> namen = new ArrayList<>();
    try (Scanner scanner = new Scanner(System.in)) {
      System.out.println("Tik soortnamen, tik stop na de laatste naam");
      for (String naam; ! "stop".equalsIgnoreCase(naam = scanner.nextLine());
        namen.add(naam));                                                    (1)
    }
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         PreparedStatement statement =
           connection.prepareStatement(INSERT_SQL)) {                      (2)
      connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
      connection.setAutoCommit(false);
      for (String naam : namen) {
        statement.setString(1, naam);                                        (3)
        statement.addBatch();                                                (4)
      }
      int[] aantalToegevoegdeRecordsPerInsert = statement.executeBatch();  (5)
      connection.commit();
      int aantalToegevoegdeSoorten = 0;
      for (int aantalToegevoegdeRecords : aantalToegevoegdeRecordsPerInsert) {
        aantalToegevoegdeSoorten += aantalToegevoegdeRecords;
      }
      System.out.println(aantalToegevoegdeSoorten);
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) De iteratie declareert één keer een variabele naam. De gebruiker tikt per iteratie een naam. Zolang die naam verschilt van stop voeg je de naam toe aan namen.
NetBeans kan op deze opdracht een onterechte warning geven.

(2) Je maakt het PreparedStatement één keer aan.

(3) Je vult de parameter in het insert statement met één van de ingetikte namen.

(4) Je voegt het PreparedStatement met zijn insert statement en ingevulde parameter toe aan het netwerkpakket.

(5) Je stuurt het PreparedStatement als één netwerkpakket naar de database.
De database voert alle insert statements uit en stuurt daarna één netwerkpakket terug.
Dit pakket bevat een array met per insert statement het aantal toegevoegde records.

Failliet 2: zie takenbundel

10 Autonumber kolommen

Als je een record toevoegt, geef je geen waarde mee voor een autonumber kolom:
de database vult een waarde in. Je kan na het toevoegen deze waarde vragen in 2 stappen

1) Wanneer je het SQL `insert` statement specifieert,
   geef je als tweede parameter de constante `Statement.RETURN_GENERATED_KEYS` mee.
2) Nadat je het `insert` statement uitvoert, voer je op het `Statement` of het `PreparedStatement` de method `getGeneratedKeys` uit. Deze geeft een `ResultSet`. Deze bevat één rij en één kolom. De kolomwaarde is de inhoud van de autonumber kolom in het toegevoegde record.

10.1 Voorbeeld

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String INSERT_SQL = "insert into soorten(naam) values (?)";
  public static void main(String[] args) {
    try (Scanner scanner = new Scanner(System.in)) {
      System.out.print("Naam:");
      String naam = scanner.nextLine();
      try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD);
           PreparedStatement statement = connection.prepareStatement(INSERT_SQL,
             Statement.RETURN_GENERATED_KEYS)) {                          (1)
        statement.setString(1, naam);
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        statement.executeUpdate();                                        (2)
        try (ResultSet resultSet = statement.getGeneratedKeys()) {        (3)
          resultSet.next();                                               (4)
          System.out.println(resultSet.getLong(1));                       (5)
          connection.commit();
        }
      } catch (SQLException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

(1) Je vermeldt `RETURN_GENERATED_KEYS` waar je het `insert` statement specifieert.
(2) Je voert het `insert` statement uit.
(3) De method `getGeneratedKeys` geeft een `ResultSet` met de autonumber waarde.
Je vermeldt ook deze `ResultSet` binnen de ronde haakjes van een `try` blok.
Zo wordt hij automatisch gesloten na gebruik.
(4) Je plaatst je op de eerste `ResultSet` rij met de `next` method.
(5) Je leest de inhoud van de eerste kolom. Je spreekt de kolom aan met zijn volgnummer omdat ze geen naam heeft. De kolom inhoud is de autonumber waarde.

Je kan de applicatie uitproberen.

10.2 Batch updates

Als je na de method `executeBatch` de method `getGeneratedKeys` uitvoert,
krijg je een `ResultSet` met evenveel rijen als je batch `insert` statements bevatte.

Elke rij bevat één kolom met een autonumber waarde die de database aanmaakte
bij het uitvoeren van een `insert` statement.

11 Datums en tijden

Kolommen kunnen van het type `date`, `time` en/of `datetime` zijn.

De SQL standaard is vaag over

- hoe je een datum of tijd letterlijk schrijft in een SQL statement
  De SQL standaard zegt bijvoorbeeld niet in welke volgorde je de dag, maand en het jaar tikt.
  Je schrijft bij elk databasemerk een datum of tijd in een ander formaat
- welke datum functies en tijd functies je kan oproepen in een SQL statement.
  Elk databasemerk heeft zijn eigen datum functies en tijd functies.

Dit maakt het moeilijk om een applicatie te schrijven die werkt met meerdere databasemerken.
JDBC bevat de nodige voorzieningen om deze problemen op te lossen.

11.1 Een datum of tijd letterlijk schrijven in een SQL statement

Je schrijft bij JDBC een datum in een SQL statement als `{d 'yyyy-mm-dd'}`.
Je vervangt hierbij yyyy door het jaar, `mm` door de maand en `dd` door de dag.
Je schrijft 31/1/2001 bijvoorbeeld als `{d '2001-1-31'}`
JDBC geeft deze datum op correcte manier door naar elk databasemerk.

Je schrijft een tijd in een SQL statement als `{t 'hh:mm:ss'}`
Je vervangt hierbij `hh` door het uur, `mm` door de minuten en `ss` door de seconden.
Je schrijft 12:35:00 bijvoorbeeld als `{t '12:35:00'}`

Je kan ook een datum én tijd combineren als `{ts 'yyyy-mm-dd hh:mm:ss'}`
Je schrijft 31/1/2001 12:35:00 bijvoorbeeld als `{ts '2001-1-31 12:35:00'}`

Voorbeeld: een lijst van werknemers die vanaf 2001 in dienst kwamen

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SELECT_SQL = "select indienst,voornaam,familienaam"
    + " from werknemers where indienst >= {d '2001-1-1'} order by indienst";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(SELECT_SQL)) {
      while (resultSet.next()) {
        System.out.printf("%s %s %s%n", resultSet.getDate("indienst"),
          resultSet.getString("voornaam"), resultSet.getString("familienaam"));
      }
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

Je kan de applicatie uitproberen.

11.2 Een datum als parameter

Als de gebruiker een datum intikt, die je nodig hebt in een SQL statement, stel je die datum in het statement voor als een parameter (?) en je gebruikt een `PreparedStatement`

Voorbeeld: een lijst van werknemers die vanaf een ingetikte datum in dienst kwamen

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SELECT_SQL = "select indienst,voornaam,familienaam"
    + " from werknemers where indienst >= ? order by indienst";
  public static void main(String[] args) {
    System.out.print("Datum vanaf (dd/mm/yyyy):");
    try (Scanner scanner = new Scanner(System.in)) {
      SimpleDateFormat dateFormat = new SimpleDateFormat("d/M/y");          (1)
      Date datum = dateFormat.parse(scanner.nextLine());                    (2)
      try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD);
           PreparedStatement statement =connection.prepareStatement(SELECT_SQL)) {
        statement.setDate(1, new java.sql.Date(datum.getTime()));           (3)
        try (ResultSet resultSet = statement.executeQuery()) {
          while (resultSet.next()) {
            System.out.printf("%s %s %s%n", resultSet.getDate("indienst"),
              resultSet.getString("voornaam"),
              resultSet.getString("familienaam"));
          }
        }
      } catch (SQLException ex) {
        ex.printStackTrace();
      }
    } catch (ParseException ex) {
      System.out.println("Verkeerde datum");
    }
  }
}
```

(1) Een `SimpleDateFormat` object converteert een String naar een `java.util.Date`.
De constructor heeft een `String` parameter met de datumopmaak.
`d` betekent dag, `M` betekent maand, `y` betekent jaar.

(2) De `parse` method converteert een `String` naar een `java.util.Date`.
Als de conversie mislukt, werpt Java een `ParseException`, die je onder in de source opvangt.

(3) De method `SetDate` verwacht een `java.sql.Date`. Je converteert de `java.util.Date` naar dit type. `getTime` geeft het aantal milliseconden sinds `1/1/1970` in de `java.util.Date`.
Je geeft dit getal mee aan de `java.sql.Date` constructor.
Je krijgt zo een `java.sql.Date` met dezelfde datum als die in de `java.util.Date`.

Je kan de applicatie uitproberen.

11.3 Datum en tijd functies

JDBC bevat datum en tijd functies die elk databasemerk correct verwerkt. De belangrijkste functies:

- `curdate()` huidige datum
- `curtime()` huidige tijd
- `now()` huidige datum en tijd
- `dayofmonth(eenDatum)` dag in de maand van `eenDatum` (getal tussen 1 en 31)
- `dayofweek(eenDatum)` dag in de week van `eenDatum` (getal tussen 1: zondag en 7)
- `dayofyear(eenDatum)` dag in het jaar van `eenDatum` (getal tussen 1 en 366)
- `month(eenDatum)` maand in `eenDatum` (getal tussen 1 en 12)
- `week(eenDatum)` week van `eenDatum` (getal tussen 1 en 53)
- `year(eenDatum)` jaartal van `eenDatum`
- `hour(eenTijd)` uur van `eenTijd` (getal tussen 0 en 23)
- `minute(eenTijd)` minuten van `eenTijd` (getal tussen 0 en 59)
- `second(eenTijd)` seconden van `eenTijd` (getal tussen 0 en 59)

Je roept in een SQL statement deze functies op met volgende syntax:

```
{fn naamVanDeFunctie(eventueleParameter)}
```

Voorbeeld: een lijst van werknemers die in de huidige maand jarig zijn.
Een werknemer komt in de lijst voor als de maand van zijn geboortedatum `{fn month(geboorte)}` gelijk is aan de maand van de systeemdatum `{fn month({fn curdate()})}`

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SELECT_SQL = "select geboorte,voornaam,familienaam"
    + " from werknemers where {fn month(geboorte)} = "
    + "{fn month({fn curdate()})} order by {fn dayofmonth(geboorte)}";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(SELECT_SQL)) {
      while (resultSet.next()) {
        System.out.printf("%s %s %s%n", resultSet.getDate("geboorte"),
          resultSet.getString("voornaam"), resultSet.getString("familienaam"));
      }
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

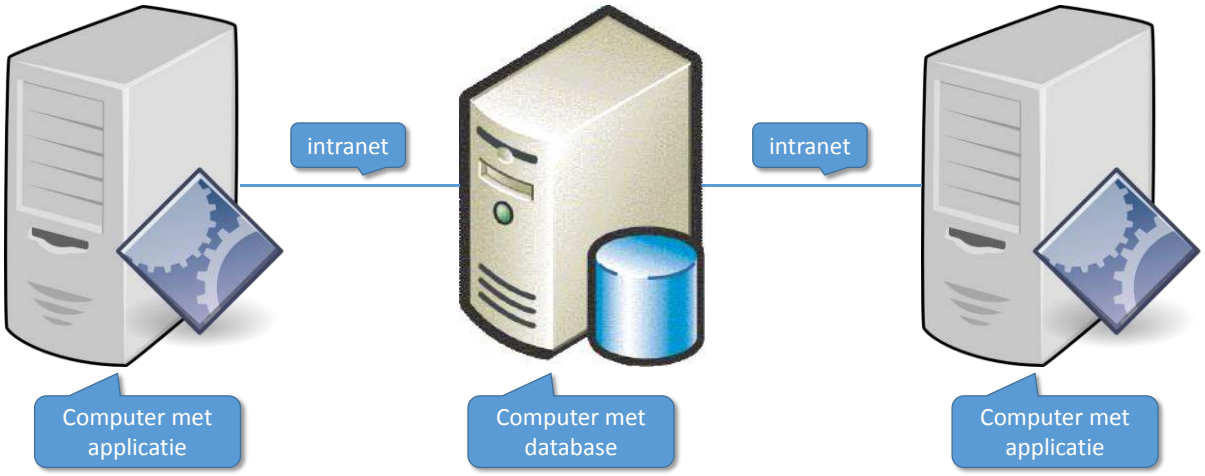
Je kan de applicatie uitproberen.

Bieren van een maand: zie takenbundel

12 De database optimaal aanspreken

De computer waarop de applicatie draait is meestal een andere dan die waarop de database draait. De applicatie communiceert dan over het intranet met de database. Voordelen:

- Je verdeelt het werk: op de ene computer voeren de CPU's de code van de applicatie uit. Op de andere computer voeren de CPU's de code van de databaseserver uit.
- Meerdere applicaties, geïnstalleerd op verschillende computers, kunnen eenzelfde database delen.



Telkens je vanuit je applicatie de database aanspreekt, gebruik je

- het intranet
- en de harddisk (de database server zoekt de data op zijn harddisk)

Als je hierbij rekening houdt dat

- het intranet ongeveer **1.000** keer trager is dan RAM geheugen
- een harddisk ongeveer **100.000** keer trager is dan RAM geheugen

besef je dat de het aanspreken van de database het onderdeel in je code is waar je moet proberen optimaal tewerk te gaan. Daarbij gelden twee vuistregels

- lees enkel de records die je nodig hebt
- lees records, die gerelateerd zijn aan andere gelezen records via joins in je SQL statements

Je leert hieronder de details van deze twee vuistregels.

12.1 Lees enkel de records die je nodig hebt

Als je voor een programma onderdeel alle records uit een table nodig hebt, stuur je een SQL `select` statement naar de database dat alle records leest. Voorbeeld: je hebt alle leveranciers nodig:

```
select id, naam, woonplaats from leveranciers
```

Als je echter maar een deel van de records nodig hebt, is de slechte oplossing

1. alle records uit de database te lezen
2. in Java code de records te filteren die je nodig hebt

Deze oplossing is slecht omdat de database alle records (meer dan nodig) over het netwerk naar de applicatie stuurt. Dit transport van veel bytes over het netwerk maakt de oplossing traag.

Voorbeeld van zo'n slechte oplossing: je wil enkel de leveranciers uit `Wevelgem`

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
```

```
  private static final String SELECT_SQL =
    "select id, naam, woonplaats from leveranciers";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(SELECT_SQL)) {       (1)
      int aantalLeveranciers = 0;
      while (resultSet.next()) {
        if ("Wevelgem".equals(resultSet.getString("woonplaats"))) {       (2)
          ++aantalLeveranciers;
          System.out.printf("%4d %s%n", resultSet.getInt("id"),
            resultSet.getString("naam"));
        }
      }
      System.out.printf("%d leveranciers(s)", aantalLeveranciers);
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) Je leest *alle* leveranciers uit de database
(2) Je filtert de juiste leveranciers in Java code

De goede oplossing is enkel de records te lezen die je nodig hebt via het `where` deel van het select statement:

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SELECT_SQL =                                (1)
    "select id, naam, woonplaats from leveranciers where woonplaats = 'Wevelgem'";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(SELECT_SQL)) {
      int aantalLeveranciers = 0;
      while (resultSet.next()) {                                          (2)
        ++aantalLeveranciers;
        System.out.printf("%4d %s%n", resultSet.getInt("id"),
          resultSet.getString("naam"));
      }
      System.out.printf("%d leveranciers(s)", aantalLeveranciers);
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) Je leest enkel de leveranciers uit `Wevelgem` uit de database
(2) Je moet in je Java code niet meer filteren.

De database stuurt bij deze oplossing enkel de leveranciers uit `Wevelgem` over het netwerk naar de applicatie. De oplossing is snel omdat er minder bytes over het netwerk getransporteerd worden.

Een mogelijk bijkomend voordeel van de goede oplossing is dat als er een index ligt op de kolom `woonplaats`, de database in deze index snel weet welke leveranciers als woonplaats `Wevelgem` hebben, en enkel van die leveranciers de `id` en `naam` effectief moet lezen op de harde schijf.

12.2 Lees records, die gerelateerd zijn aan andere gelezen records via joins in je SQL statements

Je hebt in veel overzichten data uit gerelateerde tables nodig. Je maakt als voorbeeld een overzicht met de namen van rode planten (uit de table `planten`)
en naast iedere naam de bijbehorende leveranciersnaam (uit de gerelateerde table `leveranciers`).

Een verkeerde (want trage) oplossing is

- eerst enkel de planten (nog niet de leveranciers) te lezen met een SQL `select` statement
- daarna over deze gelezen planten te itereren en per plant met een SQL `select` statement de bijbehorende leverancier lezen

Door de overvloed van SQL statements (en de resultaten van die SQL statements) die je over het intranet verstuurt, wordt dit een trage applicatie. De code zou er als volgt uit zien:

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SELECT_SQL_PLANTEN =
    "select naam, leverancierid from planten where kleur ='rood'";
  private static final String SELECT_SQL_LEVERANCIER =
    "select naam from leveranciers where id = ?";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement statementPlanten = connection.createStatement();
         PreparedStatement statementLeverancier =
           connection.prepareStatement(SELECT_SQL_LEVERANCIER);
         ResultSet resultSetPlanten =
           statementPlanten.executeQuery(SELECT_SQL_PLANTEN)) {          (1)
      while (resultSetPlanten.next()) {
        System.out.print(resultSetPlanten.getString("naam"));
        System.out.print(' ');
        statementLeverancier.setLong(1,resultSetPlanten.getLong("leverancierid"));
        try (ResultSet resultSetLeverancier =
               statementLeverancier.executeQuery()){                    (2)
          System.out.println(resultSetLeverancier.next() ?
            resultSetLeverancier.getString("naam") : "leverancier niet gevonden");
        }
      }
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

(1) Je leest één keer enkel de rode planten
(2) Je leest per plant de bijbehorende leverancier

Een performantere oplossing (minder lezen op harddisk en minder netwerkverkeer) is de rode planten én hun bijbehorende leveranciers met één `select` statement te lezen.

Niet enkel is de performantie beter, ook is de code korter:

```
// enkele imports
class Main {
  private static final String URL = "jdbc:mysql://localhost/tuincentrum";
  private static final String USER = "cursist";
  private static final String PASSWORD = "cursist";
  private static final String SELECT_SQL =
    "select p.naam as plantnaam, l.naam as leveranciersnaam " +
    " from planten p inner join leveranciers l on p.leverancierid=l.id" +
    " where kleur ='rood'";
  public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
         Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(SELECT_SQL)) {
      while (resultSet.next()) {
        System.out.printf("%s %s%n", resultSet.getString("plantnaam"),
          resultSet.getString("leveranciersnaam"));
      }
    } catch (SQLException ex) {
      ex.printStackTrace();
    }
  }
}
```

Je kan de applicatie uitproberen.

13 Herhalingsoefeningen

Omzet niet gekend: zie takenbundel

Bieren van een soort zie takenbundel

Omzet leegmaken zie takenbundel

COLOFON

Domeinexpertisemanager	Jean Smits
Moduleverantwoordelijke	
Auteurs	Hans Desmet
Versie	7/10/2015
Codes	Peoplesoftcode: Wettelijk depot:

Omschrijving module-inhoud

Abstract	Doelgroep	Opleiding Java Ontwikkelaar
	Aanpak	Zelfstudie
	Doelstelling	JDBC kunnen gebruiken
Trefwoorden		JDBC
Bronnen/meer info		