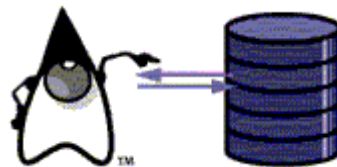




samen sterk voor werk

















Java

JDBC



Deze cursus is eigendom van VDAB Competentiecentra ©

INHOUD

1	Inleiding 	3
1.1	Doelstelling	3
1.2	Vereiste voorkennis	3
1.3	Nodige software	3
1.4	Tijdzone	3
1.5	De voorbeelddatabases	3
2	JDBC driver  - Connection 	5
2.1	JDBC driver 	5
2.2	Het project 	5
2.3	Connection 	6
2.3.1	TCP/IP poortnummer    	7
2.3.2	De databasegebruiker 	7
2.4	Samenvatting	7
3	Statement  : records toevoegen, wijzigen of verwijderen	8
3.1	Samenvatting	8
4	ResultSet  : records lezen	9
4.1	Kolommen aanduiden met hun volgnummer	9
4.2	Kolommen aanduiden met hun naam	10
4.3	select *	11
4.4	Null values	12
4.5	Soorten ResultSets	13
4.6	Samenvatting	13
5	PreparedStatement: parameters in SQL statements 	14
5.1	Voorbeeld	14
5.2	SQL code injection	15
6	Metadata 	16
6.1	Metadata over de JDBC driver	16
6.2	Metadata over de database	16
6.3	Metadata over een ResultSet	17
7	CallableStatement: oproepen van stored procedures 	18
7.1	Een stored procedure aanmaken en uitproberen	18
7.2	De stored procedure oproepen vanuit Java code met CallableStatement	18

8	Transacties 🎮	20
8.1	De autocommit mode	20
8.2	Commit en rollback	20
8.3	Voorbeeld	20
8.4	Samenvatting	21
8.5	Isolation level	22
8.6	Voorbeeld	23
9	Batch updates 🏃	26
9.1	Voorbeeld 1: statements zonder parameter	26
9.2	Voorbeeld 2: meerdere uitvoeringen van een SQL statement met parameter(s)	27
10	Autonumber kolommen 📅	28
10.1	Voorbeeld: een soort toevoegen	28
11	Datums en tijden 🕒	29
11.1	Een datum of tijd letterlijk schrijven in een SQL statement	29
11.2	Een datum als parameter	30
11.3	Datum en tijd functies	31
12	Een record enkel wijzigen als het aan voorwaarden voldoet	32
13	De database optimaal aanspreken 🏃	34
13.1	Lees enkel de records die je nodig hebt	34
13.2	Lees records uit de 1 kant van een relatie via joins in je SQL statements	36
13.3	Maak optimaal gebruik van het in keyword in SQL	37
14	JDBC en object oriëntatie 👥	39
15	Herhalingsoefeningen 🛑	41

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 Tijdzone

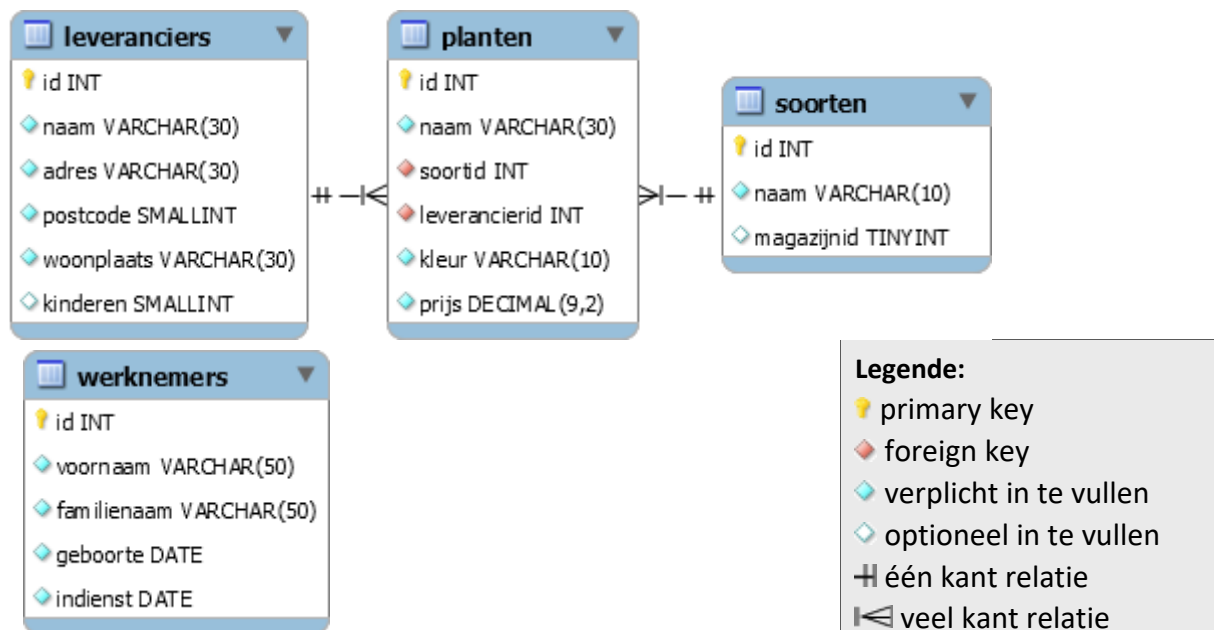
Je moet de tijdzone van je MySQL server instellen om databases aan te spreken met JDBC.

Je doet daartoe volgende stappen:


1. Je start de MySQL Workbench en je logt in op de Local instance.
2. Je kiest links Options File.
3. Je scrolt in de instellingen naar beneden naar het onderdeel International.
4. Je plaatst daarbinnen een vinkje bij default-time-zone.
5. Je tikt daarnaast +01:00 (dit geeft de tijdzone aan waartoe Brussel behoort).
6. Je klikt rechts onder op de knop Apply...
7. Je klikt op de knop Apply.
8. Je kiest links boven in het scherm Startup/Shutdown.
9. Je kiest de knop Stop Server.
10. De MySQL server stopt. Je kiest daarna de knop Start Server.

1.5 De voorbeelddatabases

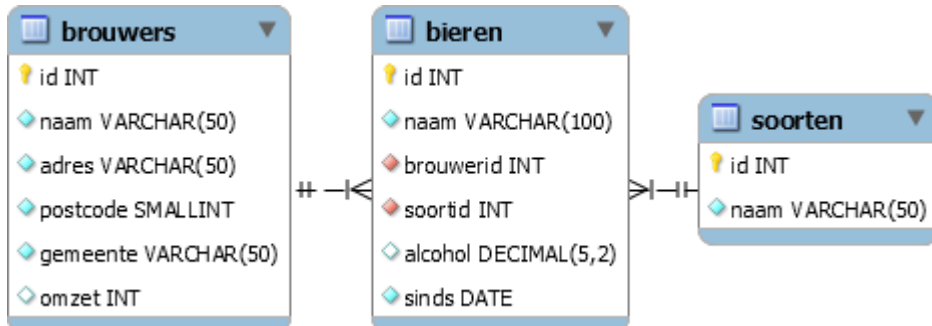
Je gebruikt in de theorie de database tuincentrum. Deze heeft volgende structuur:



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 dit schema zelf maken via de opdracht Reverse Engineer in het menu Database van de MySQL workbench.

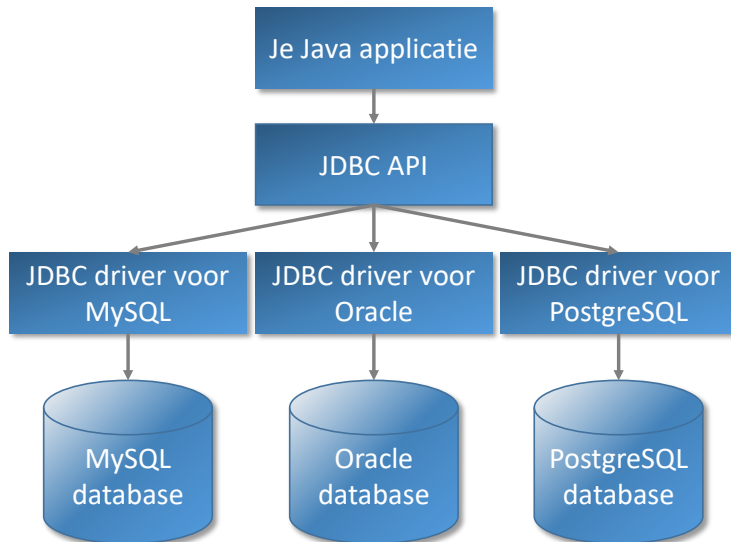
2 JDBC driver - Connection

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:

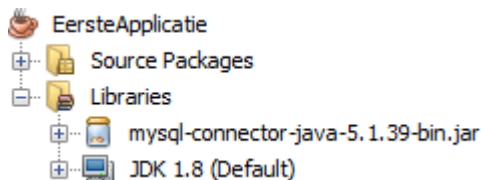
1. Je opent in de browser <http://dev.mysql.com/downloads/connector/j>.
2. Je kiest Platform Independent bij Select platform.
3. Je kiest Download bij Platform Independent (Architecture Independent), ZIP Archive.
4. Je kiest No thanks, just start my download.
5. Je opent het ZIP bestand en de map binnen het ZIP bestand.
6. Je extract het JAR bestand en plaatst dit ergens op de computer.

2.2 Het project

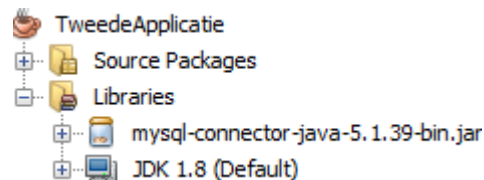
Je maakt in NetBeans een Java project.

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 MySQL database aanspreekt eenzelfde verwijzing toe.



Opmerking: je zal in deze cursus JDBC rechtstreeks aanspreken vanuit `public static void main(String[] args)`. Je kan je zo concentreren op de studie van JDBC zelf. Je zal later in de opleiding JDBC aanspreken vanuit aparte classes.

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 methode 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 netwerknnaam van de computer waarop MySQL draait. Als dit de computer is waarop je werkt, is de netwerknnaam `localhost`.
 - Bij MySQL komt hierna een `/` en de naam van de te openen database, dus dit wordt `jdbc:mysql://localhost/tuincentrum`.
 - Bij MySQL kunnen hierna parameters komen die de connectie verfijnen. Voor de eerste parameter staat een vraagteken. Voor elke volgende parameter staat het `&` teken.

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;           ❶
import java.sql.DriverManager;
import java.sql.SQLException;
class Main {
    private static final String URL =
        "jdbc:mysql://localhost/tuincentrum?useSSL=false"; ❷
    private static final String USER = "root";
    private static final String PASSWORD = "vdab";
    public static void main(String[] args) {
        try (Connection connection =           ❸
            DriverManager.getConnection(URL, USER, PASSWORD)) { ❹
            System.out.println("Connectie geopend");
        } catch (SQLException ex) {           ❺
            ex.printStackTrace(System.err);    ❻
        }
    }
}
```

- (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 geeft in de JDBC URL een parameter `useSSL` mee die je op `false` plaatst. Dit geeft aan dat het dataverkeer tussen je applicatie en de database niet geëncrypteerd wordt. De configuratie van geëncrypteerd dataverkeer is complex en valt buiten het bereik van deze cursus. Moderne versies van MySQL geven een warning als je deze parameter niet meegeeft.

- (3) Connection erft van AutoCloseable. Als je de Connection variabele declareert en initialiseert binnen de ronde haakjes van de try, voegt de compiler zelf een finally blok aan dit try blok toe waarin hij de Connection sluit.
- (4) Je maakt een verbinding met de database tuincentrum op de eigen computer.
Je verbindt met de gebruikersnaam root en het bijbehorende paswoord vdab.
- (5) 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.

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.

2.3.1 TCP/IP poortnummer 3306

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.

- Webserver gebruiken standaard poort nummer 80.
- Mail server 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?useSSL=false>

2.3.2 De databasegebruiker



Het is een slechte gewoonte om een databaseverbinding te maken met de gebruiker root.

Als een hacker het bijbehorende paswoord ontdekt 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 if not exists cursist identified by 'cursist';
use tuincentrum;
grant select on leveranciers to cursist;
grant select, update on planten to cursist;
grant select on werknemers to cursist;
grant select, insert on soorten to cursist;
```

❶

❷

- (1) Je maakt een gebruiker met de naam cursist en het paswoord cursist, tenzij deze gebruiker al bestaat.
- (2) Je geeft deze gebruiker leesrechten op de table leveranciers.

Je geeft de gebruiker niet meer rechten dan nodig. Dit wordt aanzien als 'best practice'.

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

Je kan de applicatie terug uitproberen.

2.4 Samenvatting



DriverManager

maakt



Connection

naar



database.

3 Statement : records toevoegen, wijzigen of verwijderen

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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String UPDATE_PRIJS =
        "update planten set prijs = prijs * 1.1";
    public static void main(String[] args) {
        try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
            Statement statement = connection.createStatement()) {
            System.out.println(statement.executeUpdate(UPDATE_PRIJS));
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

- (1) De Connection method createStatement geeft je een Statement object.
Volgens de JDBC specificatie wordt een Statement automatisch gesloten als je de bijbehorende Connection sluit. Jammer genoeg volgen niet alle JDBC drivers de specificatie. Het is daarom beter een Statement na gebruik te sluiten om de database performantie optimaal te houden. Statement erft van AutoCloseable. Je maakt het Statement daarom ook tussen de ronde haakjes van de try opdracht. De compiler maakt dan zelf code die dit Statement sluit in een finally blok
- (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



Bieren verwijderen: zie takenbundel

4 ResultSet : records lezen

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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT =
        "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)) {
            while (resultSet.next()) {
                System.out.println(resultSet.getInt(1) + " " +
                                   resultSet.getString(2));
            }
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

①
②
③
④

- (1) De Statement method `executeQuery` geeft een `ResultSet` object.
Volgens de JDBC specificatie wordt een `ResultSet` automatisch gesloten als je het bijbehorende Statement sluit. Jammer genoeg volgen niet alle JDBC drivers de specificatie. Het is daarom beter een `ResultSet` na gebruik te sluiten om de database performantie optimaal te houden. `ResultSet` erft van `AutoCloseable`.
Je maakt de `ResultSet` daarom ook tussen de ronde haakjes van de try opdracht.
De compiler maakt dan zelf code die deze `ResultSet` sluit in een `finally` blok.
- (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.println`:

```
System.out.println(resultSet.getInt("id") + " " +  
    resultSet.getString("naam"));
```

①
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.sql.Date`.
Als de waarde een datum+tijd bevat, lees je enkel de datum.
Je gebruikt deze method meestal bij het kolomtype `date`.
Gezien `LocalDate` een handiger class is dan `java.sql.Date`, kan je het resultaat van `getDate` beter direct converteren naar een `LocalDate` met de method `toLocalDate` van `java.sql.Date`.
- `getTime` een waarde met een tijd of datum+tijd lezen als een `java.sql.Time`.
Als de waarde een datum+tijd bevat, lees je enkel de tijd.
Je gebruikt deze method meestal bij het kolomtype `time`.
Gezien `LocalTime` een handiger class is dan `java.sql.time`, kan je het resultaat van `getTime` beter direct converteren naar een `LocalTime` met de method `toLocalTime` van `java.sql.Time`.

- `getTimestamp` een waarde met een datum+tijd lezen als een `java.sql.Timestamp`.
Je leest de datum én de tijd.
Je gebruikt deze method meestal bij het kolomtype `datetime`.
Gezien `LocalDateTime` een handiger class is dan `java.sql.Timestamp`, kan je het resultaat van `getTimestamp` beter direct converteren naar een `LocalDateTime` met de method `toLocalDateTime` van `java.sql.Timestamp`.

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

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

- (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, prijs`.

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 kinderen 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("kinderen") 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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT =
        "select naam, kinderen 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)) {
            while (resultSet.next()) {
                System.out.print(resultSet.getString("naam") + ' ');
                int kinderen = resultSet.getInt("kinderen");
                System.out.println(resultSet.isNull() ? "onbekend" : kinderen);
            }
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

(1) Je leest de waarde uit de kolom kinderen.

(2) Je controleert of de laatst gelezen kolom (kinderen) null bevatte.

Je kan de applicatie uitproberen.



Tip: een SQL statement intikt in een String in Java is onhandiger dan een SQL statement intikken in de MySQL Workbench: keywords krijgen geen kleuren, je krijgt geen popup vensters die je helpen de opdracht te tikken, ... Een oplossing is het SQL statement eerst te tikken en uit te proberen in de MySQL Workbench, en het daarna met copy-paste over te brengen naar Java.



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 een 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.



Je sluit Connections, Statements en ResultSets. Je bent zo zeker dat

- je programma werkt met alle merken databases.
- je programma werkt als het door veel gelijktijdige personen gebruikt wordt.

4.6 Samenvatting



Driver

Manager



Connection.



Connection



Statement.



Statement



Result

Set.



Aantal bieren per brouwer: zie takenbundel

5 PreparedStatement: parameters in SQL statements

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	<code>select naam from leveranciers where woonplaats = 'Kortrijk'</code>
Menen	<code>select naam from leveranciers where woonplaats = 'Menen'</code>

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^o 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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT =
        "select naam from leveranciers where woonplaats = ?";
    public static void main(String[] args) {
        System.out.print("Woonplaats:");
        String woonplaats = new Scanner(System.in).nextLine();
        try (Connection connection=DriverManager.getConnection(URL, USER, PASSWORD);
            PreparedStatement statement = connection.prepareStatement(SELECT)) {
            statement.setString(1, woonplaats);
            try (ResultSet resultSet = statement.executeQuery()) {
                while (resultSet.next()) {
                    System.out.println(resultSet.getString("naam"));
                }
            }
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

- (1) Je geeft aan de Connection method `prepareStatement` een SQL statement mee.
 Je krijgt een PreparedStatement object terug.
- (2) Je vult de eerste (en hier enige) parameter met de waarde die de gebruiker intikte.
- (3) Je voert het SQL statement uit dat je bij (1) meegaf.

Je kan de applicatie uitproberen.



Tip: als je SQL statement eerst wil uitproberen in de MySQL Workbench, moet je daar het `?` vervangen door een voorbeeldwaarde (bvb. 'Kortrijk').

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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    public static void main(String[] args) {
        System.out.print("Naam:");
        String naam = new Scanner(System.in).nextLine();
        try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
            Statement statement = connection.createStatement()) {
            System.out.println(statement.executeUpdate(
                "update planten set prijs = prijs * 1.1 where naam = '" + naam + "'")); ❶
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

(1) Je concateneert stukken SQL met de gebruikersinvoer tot één 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 plantensprijzen !

update planten **set** prijs = prijs * 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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String UPDATE =
        "update planten set prijs = prijs * 1.1 where naam = ?";
    public static void main(String[] args) {
        System.out.print("Naam:");
        String naam = new Scanner(System.in).nextLine();
        try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
            PreparedStatement statement = connection.prepareStatement(UPDATE)) {
            statement.setString(1, naam);
            System.out.println(statement.executeUpdate());
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

Als een hacker nu ' or '=' intikt, wordt geen enkele plant aangepast.



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?useSSL=false";
    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.getDriverName() + ' ' +
                                metaData.getDriverMajorVersion() + ' ' +
                                metaData.getDriverMinorVersion());
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

- (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?useSSL=false";
    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(System.err);
        }
    }
}
```

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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String 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();
            ResultSet resultSet = statement.executeQuery(SELECT)) {
            ResultSetMetaData metaData = resultSet.getMetaData();
            for (int index = 1; index <= metaData.getColumnCount(); index++) {
                System.out.println(metaData.getColumnName(index) + ' ' +
                                   metaData.getColumnTypeName(index));
            }
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

- (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 CallableStatement: oproepen van stored procedures

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))
begin
select naam from planten where naam like woord order by naam;
end
```

①
②
③
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 .

Je geeft de gebruiker cursist het recht om deze stored procedure uit te voeren:

`grant execute on procedure plantenmeteenwoord to cursist`

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?useSSL=false" +
        "&noAccessToProcedureBodies=true";
    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) {
        System.out.print("Woord:");
        String woord = new Scanner(System.in).nextLine();
        try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
            CallableStatement statement = connection.prepareCall(CALL)) {
            statement.setString(1, '%' + woord + '%');
        }
```

①
②
③

```

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

```

- (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 de parameter `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 één per één 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, of **geen enkel** van deze statements mag 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 prijs = prijs * 1.1 where prijs >= 100
```

```
update planten set prijs = prijs * 1.05 where prijs < 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 plantensprijzen onder € 100 niet aangepast.
- ➖ Als hij het onderdeel nog eens uitvoert, past hij de plantensprijzen 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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String UPDATE_PRIJS_10_PROCENT =
        "update planten set prijs = prijs * 1.1 where prijs >= 100";
    private static final String UPDATE_PRIJS_5_PROCENT =
        "update planten set prijs = prijs * 1.05 where prijs < 100";
```

```

public static void main(String[] args) {
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
        Statement statement = connection.createStatement()) {
        connection.setAutoCommit(false);
        statement.executeUpdate(UPDATE_PRIJS_10_PROCENT);
        statement.executeUpdate(UPDATE_PRIJS_5_PROCENT);
        connection.commit();
    } catch (SQLException ex) {
        ex.printStackTrace(System.err);
    }
}

```

- (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 opdate. 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 die jouw transactie leest.
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 Serializable te kiezen: het lost alle problemen op. Serializable is echter het traagste isolation level: de database vergrendelt (lockt) dan records tot het einde van de transactie. Andere gebruikers kunnen in die tijd die records niet wijzigen of verwijderen. Soms verhindert de database ook het toevoegen van records. Je kiest heel zelden het isolation level read uncommitted, omdat het geen enkel probleem oplost.

Je hebt de problemen nonrepeatable read en phantom read enkel voor als je in één transactie *dezelfde* records *meer dan één keer* leest. Als je slim programmeert heb je daar geen behoefte aan. Je leest de records één keer in je transactie en je onthoudt de gelezen data in het interne geheugen. Als je deze data gedurende de transactie *nog eens* nodig hebt, lees je ze niet opnieuw uit de database, maar uit het interne geheugen (waar je ze onthouden hebt na de eerste lees operatie).

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.

We zullen daarom het isolation level bij iedere transactie expliciet instellen.

Je stelt het isolation level in met de Connection method `setTransactionIsolation`.

Je doet dit voor je de transactie start:

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

In het vervolg van de cursus gebruiken we altijd een transactie, zelfs als we maar één statement naar de database sturen. We doen dit om het isolation level op het gewenste niveau te plaatsen.

8.6 Voorbeeld

De gebruiker tikt een soortnaam. Je voegt daarmee een record toe aan de table soorten.

De table soorten bevat een unieke index op de kolom naam.

De table kan dus geen twee records met dezelfde waarde in de kolom naam bevatten.

Als je dit toch probeert, krijg je een fout. Je wil verhinderen dat de gebruiker met een exception geconfronteerd wordt.

Je gebruikt bij een eerste oplossing het transaction isolation level Serializable:

```
// enkele imports
class Main {
    private static final String URL =
        "jdbc:mysql://localhost/tuincentrum?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT = "select id from soorten where naam=?";
    private static final String INSERT = "insert into soorten(naam) values (?)";
    public static void main(String[] args) {
        System.out.print("Soortnaam:");
        String soortNaam = new Scanner(System.in).nextLine();
        try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
            PreparedStatement statementSelect = connection.prepareStatement(SELECT);
            PreparedStatement statementInsert = connection.prepareStatement(INSERT)) {
            statementSelect.setString(1, soortNaam);
            connection.setTransactionIsolation(
                Connection.TRANSACTION_SERIALIZABLE);
            connection.setAutoCommit(false);
            try (ResultSet resultSet = statementSelect.executeQuery()) {
                if (resultSet.next()) {
                    System.out.println("Soort met deze naam bestaat al");
                } else {
                    statementInsert.setString(1, soortNaam);
                    statementInsert.executeUpdate();
                    connection.commit();
                }
            }
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```


(1) Je plaatst het isolation level op Serializable.


(2) Je zoekt of een record met de ingetikte soortnaam reeds in de database voorkomt.

Serializable verhindert vanaf nu tot het einde van de transactie dat andere gebruikers een record toevoegen met dezelfde soortnaam.

Je kan het programma uitproberen.

Je kan de werking van het isolation level Serializable op volgende manier zien:

1. Je plaatst een breakpoint op de regel `if (resultSet.next()) {`
Je doet dit door te klikken in de grijze marge voor deze regel. De achtergrond van deze regel wordt roze. Als je straks het programma uitvoert in debug mode, zal de uitvoering van het programma pauzeren juist voor deze regel uitgevoerd wordt.
2. Je start het programma in debug modus met de knop  in de toolbar.
3. Je tikt in het tabblad Output (beneden in NetBeans) de soortnaam test en je drukt Enter.
4. De uitvoering van je programma pauzeert op het breakpoint.
De regel van het breakpoint heeft dan een groene achtergrond.

5. Je laat NetBeans open staan en je schakelt over naar de MySQL WorkBench.
6. Je voert daar volgend statement uit: **insert into** soorten(naam) **values** ('test')
7. Je ziet onder in de MySQL WorkBench naast dit statement Running...
De MySQL WorkBench kan je statement nog niet uitvoeren: hij wordt tegengehouden door de transactie met isolation level Serializable die in uitvoering is in je programma in NetBeans.
8. Je schakelt terug naar NetBeans.
9. Je laat je programma verder lopen met de knop  in de toolbar.
10. Je programma voegt het record toe, commit de transactie en komt ten einde.
11. Je schakelt terug naar de MySQL WorkBench.
Omdat de transactie in je programma in NetBeans afgelopen is, kon de MySQL WorkBench proberen het insert statement uit te voeren. Dit mislukt omdat je programma in NetBeans reeds een record met dezelfde naam heeft toegevoegd. Je ziet dit onder in de MySQL WorkBench naast je statement: Error Code: 1062. Duplicate entry 'test' for key 'naam'

Deze oplossing heeft volgende nadelen:

- ➔ Hij gebruikt Serializable, wat veel inspanning vraagt aan de database.
- ➔ Hij stuurt altijd twee SQL statements (één select en één insert statement) naar de database.

Jeverwijdt een breakpoint door te klikken op het roze vierkant in de marge van de breakpoint.

Je gebruikt bij een tweede oplossing het isolation level Read Committed:

```
// enkele imports
class Main {
    private static final String URL =
        "jdbc:mysql://localhost/tuincentrum?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT = "select id from soorten where naam=?";
    private static final String INSERT = "insert into soorten(naam) values (?)";
    public static void main(String[] args) {
        System.out.print("Soortnaam:");
        String soortNaam = new Scanner(System.in).nextLine();
        try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
            PreparedStatement statementSelect = connection.prepareStatement(SELECT);
            PreparedStatement statementInsert = connection.prepareStatement(INSERT)) {
            statementInsert.setString(1, soortNaam);
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
            try {
                statementInsert.executeUpdate();
                connection.commit();
            } catch (SQLException ex) {
                statementSelect.setString(1, soortNaam);
                try (ResultSet resultSet = statementSelect.executeQuery()) {
                    if (resultSet.next()) {
                        System.out.println("Soort met deze naam bestaat al");
                    } else {
                        ex.printStackTrace(System.err);
                    }
                }
            }
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

- (1) Je probeert een record met de ingetikte soortnaam toe te voegen.
Dit lukt als er nog geen record met deze soortnaam bestaat.
- (2) Het toevoegen is mislukt. De eerste mogelijke reden is dat er reeds een record met deze soortnaam bestaat. Andere redenen kunnen zijn: database uitgevallen, de gebruiker heeft geen rechten om records toe te voegen, ...
- (3) Om te weten of het toevoegen mislukte wegens de eerste reden zoek je of reeds een record voorkomt met de ingetikte soortnaam.
- (4) Als dit het geval is, toont je een gebruikersvriendelijke foutmelding aan de gebruiker.

Je kan het programma uitvoeren.

Deze oplossing heeft volgende voordelen:

- Hij gebruikt Read committed, wat minder inspanning vraagt aan de database.
- Hij stuurt meestal maar één SQL statements (een insert statement) naar de database.

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 het aantal records dat één van de SQL statements bijwerkte.

Een batch kan geen select statement bevatten, wel insert, update en delete statements.

9.1 Voorbeeld 1: statements zonder parameter

Het programma verhoogt de plantensprijzen met 10 % als die nu een prijs hebben vanaf 100.

Het verhoogt de prijzen met 5 % als die nu een prijs hebben kleiner dan 100.

// enkele imports

```
class Main {
    private static final String URL =
        "jdbc:mysql://localhost/tuincentrum?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String UPDATE_PRIJS_10_PROCENT =
        "update planten set prijs = prijs * 1.1 where prijs >= 100";
    private static final String UPDATE_PRIJS_5_PROCENT =
        "update planten set prijs = prijs * 1.05 where prijs < 100";
    public static void main(String[] args) {
        try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
            Statement statement = connection.createStatement()) {
            statement.addBatch(UPDATE_PRIJS_10_PROCENT);
            statement.addBatch(UPDATE_PRIJS_5_PROCENT);
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
            int[] aantalGewijzigdeRecordsPerUpdate = statement.executeBatch();
            System.out.println(aantalGewijzigdeRecordsPerUpdate[0] +
                "planten met 10 % verhoogd");
            System.out.println(aantalGewijzigdeRecordsPerUpdate[1] +
                "planten met 5 % verhoogd");
            connection.commit();
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

- Je voegt het Statement toe aan het netwerkpakket. Je doet dit ook op de volgende regel.
- 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 Voorbeeld 2: meerdere uitvoeringen van een SQL statement met parameter(s)

Het programma verhoogt de plantensprijzen met een ingetikt % als die nu een prijs hebben vanaf 100. Het verhoogt de prijzen met een ander ingetikt % als die nu een prijs hebben kleiner dan 100.

```
// enkele imports
class Main {
private static final String URL =
    "jdbc:mysql://localhost/tuincentrum?useSSL=false";
private static final String USER = "cursist";
private static final String PASSWORD = "cursist";
private static final String UPDATE = "update planten" +
    "set prijs = prijs * (1 + ? / 100) where prijs between ? and ?";
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("% voor prijzen kleiner dan 100:");
    BigDecimal percentageKleinerDan100 = scanner.nextBigDecimal();
    System.out.print("% voor prijzen vanaf 100:");
    BigDecimal percentageVanaf100 = scanner.nextBigDecimal();
    try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
        PreparedStatement statement = connection.prepareStatement(UPDATE)) {
        statement.setBigDecimal(1, percentageKleinerDan100);
        statement.setBigDecimal(2, BigDecimal.ZERO);
        statement.setBigDecimal(3, BigDecimal.valueOf(99.99));
        statement.addBatch();
        statement.setBigDecimal(1, percentageVanaf100);
        statement.setBigDecimal(2, BigDecimal.valueOf(100));
        statement.setBigDecimal(3, BigDecimal.valueOf(999.999));
        statement.addBatch();
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        int[] aantalGewijzigdeRecordsPerUpdate = statement.executeBatch();
        System.out.println(aantalGewijzigdeRecordsPerUpdate[0] +
            "planten verhoogd met" + percentageKleinerDan100 + "%");
        System.out.println(aantalGewijzigdeRecordsPerUpdate[1] +
            "planten verhoogd met" + percentageVanaf100 + "%");
        connection.commit();
    } catch (SQLException ex) {
        ex.printStackTrace(System.err);
    }
}
```

Je hebt nu twee technologieën gezien die beide omgaan met een *groep* SQL statements:

- Transactie: zorgt er voor dat ofwel alle statements in de groep worden uitgevoerd, ofwel de handelingen van die statements wordt ongedaan gemaakt.
- Batch updates: sturen de statements in de groep in één keer naar de database, wat de performantie verhoogt.



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 specificeert, 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: een soort toevoegen

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

- (1) Je vermeldt `RETURN_GENERATED_KEYS` waar je het insert statement specificeert.
- (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.

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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT = "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()) {
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
            try (ResultSet resultSet = statement.executeQuery(SELECT)) {
                while (resultSet.next()) {
                    System.out.println(resultSet.getDate("indienst") + " " +
                        resultSet.getString("voornaam") + " " +
                        resultSet.getString("familienaam"));
                }
            }
            connection.commit();
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT = "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):");
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("d/M/y"); ❶
        LocalDate datum = LocalDate.parse(new Scanner(System.in).nextLine(), ❷
            formatter);
        try (Connection connection = DriverManager.getConnection(URL,USER,PASSWORD);
            PreparedStatement statement = connection.prepareStatement(SELECT)) {
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            statement.setDate(1, java.sql.Date.valueOf(datum)); ❸
            connection.setAutoCommit(false);
            try (ResultSet resultSet = statement.executeQuery()) {
                while (resultSet.next()) {
                    System.out.println(resultSet.getDate("indienst") + " " +
                        resultSet.getString("voornaam") + " " +
                        resultSet.getString("familienaam"));
                }
            }
            connection.commit();
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

- (1) Een DateTimeFormatter object helpt een String te converteren naar een LocalDate. 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 LocalDate. De eerste parameter is de te converteren String. De tweede parameter is een DateTimeFormatter object met de datumopmaak. Als de conversie mislukt, werpt Java een DateTimeParseException, die je onder in de source opvangt.
- (3) De method SetDate verwacht een java.sql.Date. Je converteert de LocalDate naar dit type met de static method valueOf. De parameter is een LocalDate. De returnwaarde is een java.sql.Date.

Je kan de applicatie uitproberen.

11.3 Datum en tijd functies

JDBC bevat datum en tijd functies die elk databasemerken 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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT =
        "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()) {
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
            try (ResultSet resultSet = statement.executeQuery(SELECT)) {
                while (resultSet.next()) {
                    System.out.println(resultSet.getDate("geboorte") + " " +
                        resultSet.getString("voornaam") + " " +
                        resultSet.getString("familienaam"));
                }
            }
            connection.commit();
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

Je kan de applicatie uitproberen.



Bieren van een maand: zie takenbundel

12 Een record enkel wijzigen als het aan voorwaarden voldoet

Voorbeeld: Je kan de prijs van een plant maximaal tot de helft verminderen. Stappen:

1. Je vraagt het nummer van de plant.
2. Je vraagt de nieuwe prijs.
3. Je leest de plant uit de database.
4. Als de nieuwe prijs \geq huidige prijs / 2, wijzig je de prijs in de database.

In code:

```
// enkele imports
class Main {
    private static final String URL =
        "jdbc:mysql://localhost/tuincentrum?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT =
        "select prijs from planten where id = ? for update";
    private static final String UPDATE = "update planten set prijs=? where id=?";
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Nummer plant:");
        long id = scanner.nextLong();
        System.out.print("Nieuwe prijs:");
        BigDecimal nieuwePrijs = scanner.nextBigDecimal();
        try (Connection connection=DriverManager.getConnection(URL, USER, PASSWORD);
            PreparedStatement statementSelect = connection.prepareStatement(SELECT);
            PreparedStatement statementUpdate= connection.prepareStatement(UPDATE)) {
            statementSelect.setLong(1, id);
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
            try (ResultSet resultSet = statementSelect.executeQuery()) {
                if (resultSet.next()) {
                    if (nieuwePrijs.compareTo(resultSet.getBigDecimal("prijs")
                        .multiply(BigDecimal.valueOf(0.5))) < 0) {
                        System.out.println("Nieuwe prijs te laag");
                    } else {
                        statementUpdate.setBigDecimal(1, nieuwePrijs);
                        statementUpdate.setLong(2, id);
                        statementUpdate.executeUpdate();
                        connection.commit();
                    }
                } else {
                    System.out.println("Plant niet gevonden");
                }
            }
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

- (1) `select ... for update` leest een record én vergrendelt dit record tot het einde van de transactie. Andere gebruikers mogen in die periode het record niet wijzigen of verwijderen.

Je kan het programma uitproberen.



Opmerking: MySQL Server versie 8 bevat momenteel een bug zodat je een `select ... for update` enkel kan uitvoeren als gebruiker **root**, niet als gebruiker **cursist**.

Geoptimaliseerde oplossing, waarbij meestal slechts één statement (een update statement) naar de database gestuurd wordt. Stappen:

1. Je vraagt het nummer van de plant.
2. Je vraagt de nieuwe prijs.
3. Als het nummer bijvoorbeeld 1 is en de nieuwe prijs is 50, voer je volgende SQL opdracht uit:
`update planten set prijs = ? where id = ? and ? > prijs / 2`
4. Je controleert het aantal aangepaste records.
 - a. Als dit gelijk is aan één is het programma OK verlopen.
 - b. Anders zijn er twee mogelijkheden:
 - i. De plant bestaat niet
 - ii. De nieuwe prijs is kleiner dan de huidige prijs / 2.

Om te weten welke van deze mogelijkheden zijn opgetreden, zoek je de plant in de database.
 Als je die vindt, is de tweede mogelijkheid opgetreden.

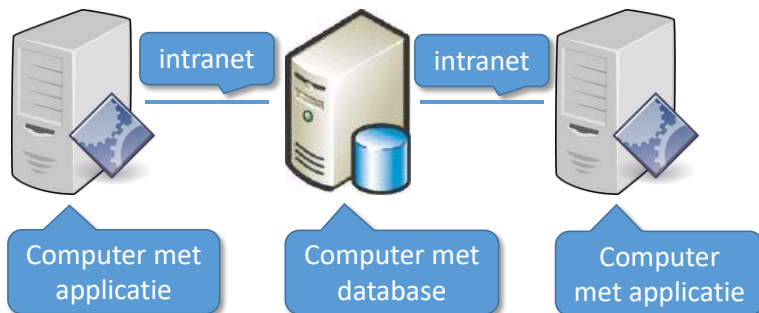
```
//enkele imports
class Main {
    private static final String URL =
        "jdbc:mysql://localhost/tuincentrum?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT = "select id from planten where id = ?";
    private static final String UPDATE =
        "update planten " + "set prijs = ? where id = ? and ? > prijs / 2";
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Nummer plant:");
        long id = scanner.nextLong();
        System.out.print("Nieuwe prijs:");
        BigDecimal nieuwePrijs = scanner.nextBigDecimal();
        try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
            PreparedStatement statementUpdate = connection.prepareStatement(UPDATE);
            PreparedStatement statementSelect = connection.prepareStatement(SELECT)) {
            statementUpdate.setBigDecimal(1, nieuwePrijs);
            statementUpdate.setLong(2, id);
            statementUpdate.setBigDecimal(3, nieuwePrijs);
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
            int aantalAangepasteRecords = statementUpdate.executeUpdate();
            if (aantalAangepasteRecords == 0) {
                statementSelect.setLong(1, id);
                try (ResultSet resultSet = statementSelect.executeQuery()) {
                    if (resultSet.next()) {
                        System.out.println("Nieuwe prijs te laag");
                    } else {
                        System.out.println("Plant niet gevonden");
                    }
                }
            }
            connection.commit();
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

Je kan het programma uitproberen.

13 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

beseft je dat de het aanspreken van de database het onderdeel in je code is waar je moet proberen optimaal tewerk te gaan.

Je leert hieronder enkele belangrijke tips.

13.1 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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT =
        "select id, naam, woonplaats from leveranciers";
    public static void main(String[] args) {
        try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
            Statement statement = connection.createStatement();) {
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
```

```

    try (ResultSet resultSet=statement.executeQuery(SELECT)) {
        int aantalLeveranciers = 0;
        while (resultSet.next()) {
            if ("Wevelgem".equals(resultSet.getString("woonplaats"))) {
                ++aantalLeveranciers;
                System.out.println(resultSet.getInt("id") + " " +
                    resultSet.getString("naam"));
            }
        }
        System.out.println(aantalLeveranciers + " leveranciers(s)");
    }
    connection.commit();
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
}
}

```

- (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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT =
        "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()) {
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
            try (ResultSet resultSet = statement.executeQuery(SELECT)) {
                int aantalLeveranciers = 0;
                while (resultSet.next()) {
                    ++aantalLeveranciers;
                    System.out.println(resultSet.getInt("id") + " " +
                        resultSet.getString("naam"));
                }
                System.out.println(aantalLeveranciers + " leveranciers(s)");
            }
            connection.commit();
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}

```

- (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.

13.2 Lees records uit de 1 kant van een relatie via joins in je SQL statements

Je hebt in veel overzichten data uit gerelateerde tables nodig.

Als je data leest uit de n kant van een relatie en je hebt de bijbehorende data nodig uit de 1 kant van de relatie geldt de tip uit dit hoofdstuk.

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).

Je leest dus data uit de n kant van de relatie (planten)

en je hebt de bijbehorende data nodig uit de 1 kant van de relatie (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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT_RODE_PLANTEN =
        "select naam, leverancierid from planten where kleur = 'rood'";
    private static final String SELECT_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_LEVERANCIER)) {
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
            try (ResultSet resultSetPlanten =
                statementPlanten.executeQuery(SELECT_RODE_PLANTEN)) {
                while (resultSetPlanten.next()) {
                    System.out.print(resultSetPlanten.getString("naam") + ' ');
                    statementLeverancier.setLong(
                        1, resultSetPlanten.getLong("leverancierid"));
                    try (ResultSet resultSetLeverancier =
                        statementLeverancier.executeQuery()) {
                        System.out.println(resultSetLeverancier.next() ?
                            resultSetLeverancier.getString("naam"):
                            "leverancier niet gevonden");
                    }
                }
            }
            connection.commit();
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

- (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?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT =
        "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()) {
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
            try (ResultSet resultSet = statement.executeQuery(SELECT)) {
                while (resultSet.next()) {
                    System.out.println(resultSet.getString("plantnaam") + " " +
                        resultSet.getString("leveranciersnaam"));
                }
            }
            connection.commit();
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

Je kan de applicatie uitproberen.

13.3 Maak optimaal gebruik van het in keyword in SQL

Voorbeeld: de gebruiker wil de planten 3, 7 en 9 zien.

Je kan dit oplossen door 3 select statements naar de database te sturen:

- `select ... from planten where id = 3`
- `select ... from planten where id = 7`
- `select ... from planten where id = 9`

Je kan dit veel performanter oplossen door 1 select statement naar de database te sturen:

```
select ... from planten where id in (3, 7, 9)
```

Voorbeeldprogramma: de gebruiker tikt plantnummers tot hij 0 tikt.

Jij toont daarna de nummer en namen van die planten.

```
// enkele imports
class Main {
    private static final String URL =
        "jdbc:mysql://localhost/tuincentrum?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String BEGIN_SELECT =
        "select id,naam from planten where id in (";
    public static void main(String[] args) {
        System.out.println("Tik plantnummers, eindig met 0");
        Set<Long> nummers = new LinkedHashSet<>();
```



```

Scanner scanner = new Scanner(System.in);
for (long nummer; (nummer = scanner.nextLong()) != 0;) {
    nummers.add(nummer);
}
StringBuilder select = new StringBuilder(BEGIN_SELECT);
for (int teller = 0; teller != nummers.size(); teller++) {
    select.append("?,");
}
select.setCharAt(select.length() - 1, ');');
try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
    PreparedStatement statement =
        connection.prepareStatement(select.toString())) {
    int index = 1;
    for (long nummer : nummers) {
        statement.setLong(index++, nummer);
    }
    connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
    connection.setAutoCommit(false);
    try (ResultSet resultSet = statement.executeQuery()) {
        while (resultSet.next()) {
            System.out.println(resultSet.getLong("id") + ":" +
                resultSet.getString("naam"));
        }
    }
    connection.commit();
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
}
}

```

Je kan deze optimalisatie ook gebruiken als de gebruiker de planten 3, 7 en 9 wil verwijderen.

Zonder optimalisatie:

- `delete from planten where id = 3`
- `delete from planten where id = 7`
- `delete from planten where id = 9`

Met optimalisatie:

`delete from planten where id in (3, 7, 9)`

Je kan deze optimalisatie ook gebruiken als de gebruiker de prijs van planten 3, 7 en 9 met 10 % wil opslaan.

Zonder optimalisatie:

- `update planten set prijs = prijs * 1.1 where id = 3`
- `update planten set prijs = prijs * 1.1 where id = 7`
- `update planten set prijs = prijs * 1.1 where id = 9`

Met optimalisatie:

`update planten set prijs = prijs * 1.1 where id in (3, 7, 9)`

14 JDBC en object oriëntatie

Je stelt de gegevens uit de werkelijkheid voor als records in tables in de database.

De records in de table werknemers stellen bijvoorbeeld de werknemers van de firma voor.

Bij object oriëntatie maak je ook een class Werknemer.

Objecten van deze class stellen dezelfde werknemers van de firma voor.

Je leert in dit hoofdstuk hoe je de twee voorstellingen (records enerzijds, objecten anderzijds) samen gebruikt. Je zal de records lezen, op basis van deze records Werknemer objecten maken en die daarna afbeelden. Je toont JARIG bij een werknemer die vandaag jarig is.

Dit is gebaseerd op een method isJarig in de class Werknemer.

Je voegt een class Werknemer toe:

```
class Werknemer {
    private long id;
    private String voornaam;
    private String familienaam;
    private LocalDate geboorte;
    private LocalDate inDienst;
    public Werknemer(long id, String voornaam, String familienaam,
        LocalDate geboorte, LocalDate inDienst) {
        this.id = id;
        this.voornaam = voornaam;
        this.familienaam = familienaam;
        this.geboorte = geboorte;
        this.inDienst = inDienst;
    }
    @Override
    public String toString() {
        return voornaam + ' ' + familienaam;
    }
    public boolean isJarig() {
        LocalDate vandaag = LocalDate.now();
        return geboorte.getMonth() == vandaag.getMonth()
            && geboorte.getDayOfMonth() == vandaag.getDayOfMonth();
    }
}
```

(1) Je maakt een LocalDate object met de datum van vandaag.

(2) Een werknemer is vandaag jarig als zijn geboortemaand gelijk is aan de maand van vandaag én zijn geboortedag gelijk is aan de dag van vandaag.

Je gebruikt dit in de Main class:

```
class Main {
    private static final String URL =
        "jdbc:mysql://localhost/tuincentrum?useSSL=false";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    private static final String SELECT = "select id, voornaam, familienaam," +
        "geboorte, indienst from werknemers ";
    public static void main(String[] args) {
        try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD);
            Statement statement = connection.createStatement()) {
            connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
            connection.setAutoCommit(false);
```

```
try (ResultSet resultSet = statement.executeQuery(SELECT)) {
    while (resultSet.next()) {
        Werknemer werknemer = new Werknemer(resultSet.getLong("id"),
            resultSet.getString("voornaam"), resultSet.getString("familienaam"),
            resultSet.getDate("geboorte").toLocalDate(),
            resultSet.getDate("indienst").toLocalDate());
        System.out.print(werknemer);
        if (werknemer.isJarig()) {
            System.out.print(" JARIG");
        }
        System.out.println();
    }
}
connection.commit();
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
}
```

15 Herhalings**o**efeningen



Omzet niet gekend: zie takenbundel



Bieren van een soort: zie takenbundel



Omzet leegmaken: zie takenbundel

COLOFON

Domeinexpertisemanager	Jean Smits
Moduleverantwoordelijke	
Auteurs	Hans Desmet
Versie	4/9/2018
Codes	Peoplesoftcode: Wettelijk depot:

Omschrijving module-inhoud

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