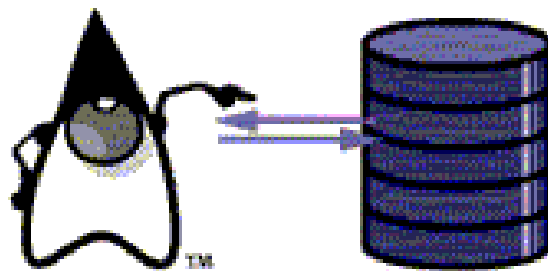




JDBC



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	Tuincentrum	4
2	Driver	5
2.1	Het project.....	5
3	Connection	6
3.1	Connection kort openhouden	7
3.2	TCP/IP poortnummer	7
3.3	Databasegebruiker	7
3.4	Samenvatting.....	7
4	Repository	8
5	PreparedStatement	9
5.1	Samenvatting.....	9
6	ResultSet	10
6.1	Kolomvolgnummers	10
6.2	Kolomnamen	11
6.3	select *	12
6.4	Werkelijkheid	13
6.5	Soorten ResultSets	14
6.6	Samenvatting.....	14
7	SQL statements met parameters	15
7.1	Voorbeeld	15
7.2	SQL code injection	16
8	Een record lezen aan de hand van zijn id	17
9	Stored procedure	18
9.1	Stored procedure maken	18
9.2	Stored prodecure testen	18
9.3	CallableStatement	18
10	Transactie.....	20

10.1	Autocommit mode	20
10.2	Commit en rollback.....	20
10.3	Voorbeeld	20
10.4	Samenvatting	21
11	Isolation level	22
11.1	Voorbeeld	23
12	Autonumber kolom	26
12.1	Voorbeeld	26
13	Datums en tijden	27
13.1	Letterlijke datum of tijd in een SQL statement	27
13.2	Datum als parameter	28
13.3	Functies.....	28
14	Lock.....	30
15	Database optimaal aanspreken	33
15.1	Enkel de records lezen die je nodig hebt.....	33
15.2	Het SQL keyword in.....	34
15.3	Join	36
15.4	Batch update.....	37
16	Herhalings oefeningen	39

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


- JDK (Java Developer Kit) (minstens versie 8).
- MySQL database server (minstens versie **8.0.14**)
- MySQL Workbench.
- IntelliJ.

1.4 Tijdzone

Je wil de database aanspreken met JDBC.

Je moet dan de tijdzone van je MySQL server instellen.

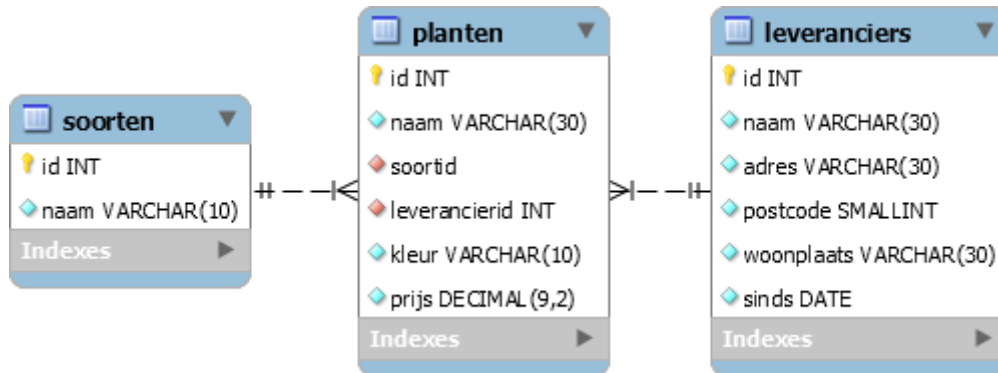
Je doet dit met volgende stappen:

1. Je opent in een browser <https://dev.mysql.com/downloads/timezones.html>.
2. Je downloadt daar `timezone_2019c_posix_sql.zip`.
3. Je haalt uit dit ZIP bestand `timezone_posix.sql`.
4. Je start de MySQL Workbench.
5. Je logt in op de Local instance.
6. Je kiest in het menu File de opdracht Open SQL Script.
7. Je opent `timezone_posix.sql`.
8. Je voegt vooraan in het script een opdracht toe: `use mysql;`
9. Je voert dit script uit met de knop . Dit kan een tijdje duren.
Met dit script moet je een tijdzone niet uitdrukken als een getal ten opzichte van de GMT (Greenwich Mean Time) (bijvoorbeeld `+01:00`).
Je kan een tijdzone uitdrukken met een naam (bijvoorbeeld Europe/Brussels).
10. Je kiest links het tabblad Administration.
11. Je kiest in dit tabblad Startup/Shutdown.
12. Je kiest de knop Stop Server.
13. De MySQL server stopt. Het is mogelijk dat hij daarna zelf herstart.
Anders kies je de knop Start Server.
14. Je kiest in het tabblad Administration voor Options File.
15. Je scrollt in de instellingen naar beneden naar het onderdeel International.
16. Je plaatst daarbinnen een vinkje bij default-time-zone.
17. Je tikt daarnaast Europe/Brussels
18. Je klikt rechts onder op de knop **Apply...**
19. Je klikt op de knop Apply.
20. Je kiest in het tabblad Administration voor Startup/Shutdown.
21. Je kiest de knop Stop Server.
22. De MySQL server stopt. Het is mogelijk dat hij daarna zelf herstart.
Anders kies je de knop Start Server.

1.5 Tuincentrum

Je gebruikt in de theorie de database tuincentrum.

Je maakt die in de MySQL Workbench met het script tuincentrum.sql.

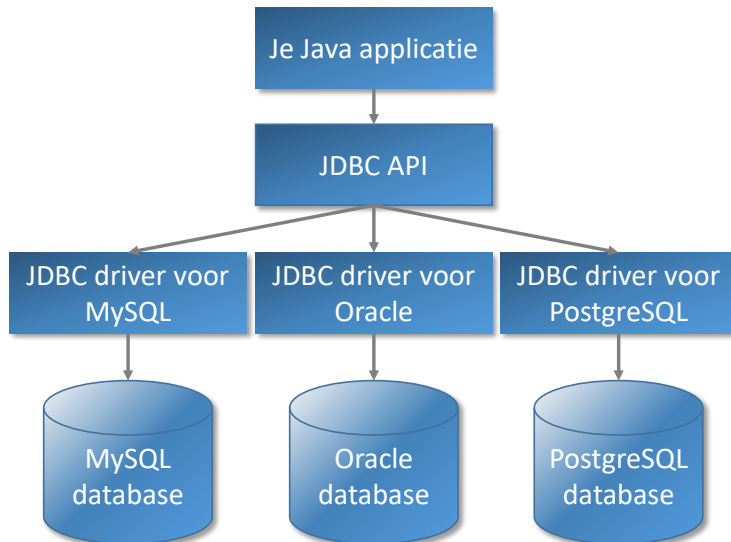


2 Driver

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 database merk:



Je downloadt de JDBC driver die hoort bij MySQL:

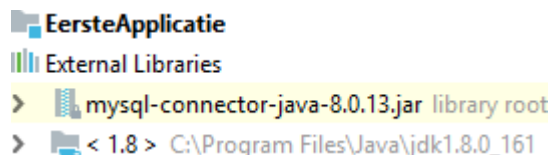
1. Je opent in de browser <http://dev.mysql.com/downloads/connector/j>.
2. Je kiest Platform Independent bij Select platform.
3. Je kiest Download bij Platform Independent (Architecture Independent), ZIP Archive.
4. Je kiest No thanks, just start my download.
5. Je opent het ZIP bestand.
6. Je opent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

2.1 Het project

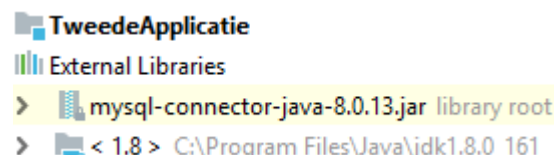
Je maakt in IntelliJ een Java console project.

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

1. Je kiest in het menu File de opdracht Project Structure.
2. Je kiest links Libraries.
3. Je kiest in het midden boven op +.
4. Je kiest Java.
5. Je duidt het JAR bestand aan dat je in de vorige paragraaf op de computer plaatste.



IntelliJ vermeldt dit JAR bestand in het project onderdeel External Libraries.



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

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 Connection implementeert.

Je maakt het programma in een package `be.vdab`:

```
package be.vdab;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
class Main {
    private static final String URL =
        "jdbc:mysql://localhost/tuincentrum?useSSL=false&allowPublicKeyRetrieval=true" +
        "&serverTimezone=Europe/Brussels";
    private static final String USER = "root";
    private static final String PASSWORD = "vdab";
    public static void main(String[] args) {
        try (Connection connection =
            DriverManager.getConnection(URL, USER, PASSWORD)) {
            System.out.println("Connectie geopend");
        } catch (SQLException ex) {
            ex.printStackTrace(System.err);
        }
    }
}
```

(1) De packages `java.sql` en `javax.sql` bevatten de JDBC classes en interfaces.

Andere packages bevatten ook een interface `Connection`. Dat is geen JDBC `Connection`.

(2) Dit is een JDBC URL. Die beschrijft de naam en de locatie van de te openen database.

De String begint altijd met `jdbc:`. Bij MySQL komt hierna `mysql://`.

Hierna komt de netwerknnaam van de computer waarop MySQL draait.

Als dit de computer is waarop je werkt, is de netwerknnaam `localhost`.

Hierna komt een `/` en de naam van de te openen database: `tuincentrum`.

Hierna kunnen parameters komen die de connectie verfijnen.

Voor de 1^o parameter staat `?`. Voor de volgende parameters staat `&`.

De 1^o parameter noemt `useSSL` en bevat `false`.

Dit geeft aan dat het dataverkeer tussen je applicatie en de database niet geëncrypteerd is.

De encryptie configuratie is complex en valt buiten het bereik van de cursus.

Je plaatst bij niet-geëncrypteerde data ook de parameter `allowPublicKeyRetrieval` op `true`.

De parameter `serverTimezone` bevat de tijdzone van je MySQL server.

(3) `Connection` erft van `AutoCloseable`. Je declareert de `Connection` variabele daarom binnen de ronde haakjes van een `try` blok. De compiler voegt zelf een `finally` blok aan dit `try` blok.

Dit `finally` blok bevat code die de `Connection` sluit.

(4) De static method `getConnection` van de class `DriverManager` geeft je een `Connection`.

`DriverManager` is zo een implementatie van het factory design pattern.

Je geeft 3 parameters mee aan `getConnection`:

a. De JDBC URL.

b. Een database gebruikersnaam waarmee je de connectie maakt: `root`.

c. Het paswoord dat bij die gebruikersnaam hoort.

(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 uitvoeren.

3.1 Connection kort openhouden

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.

3.2 TCP/IP poortnummer

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

Op één computer kunnen meerdere programma's het TCP/IP-protocol gebruiken.

Elk programma krijgt bij TCP/IP een uniek identificatie getal: het poortnummer.

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

Als de MySQL een ander poort nummer gebruikt dan 3306, vermeld je het nummer in de JDBC URL.

Bij poort nummer 3307 is de JDBC URL: `jdbc:mysql://localhost:3307/tuincentrum...`

3.3 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 volgende opdrachten uit in de MySQL Workbench:

create user if not exists cursist identified by 'cursist';

❶

use tuincentrum;

grant select on leveranciers to cursist;

❷

grant select, update on planten to cursist;

grant select, insert on soorten to cursist;

(1) Je maakt een gebruiker met de naam cursist en het paswoord cursist, tenzij die gebruiker al bestaat.

(2) Je geeft de gebruiker leesrechten op de table leveranciers.

Je geeft de gebruiker niet meer rechten dan nodig. Dit is een 'best practice'.

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

Je kan de applicatie terug uitvoeren.

3.4 Samenvatting



DriverManager

maakt



Connection

naar



database.

4 Repository

“Single responsibility” is een “best practice” bij programmeren.

Het houdt in dat een class slechts *één* verantwoordelijkheid heeft.

Je programmeert daarom de *code waarmee je de database aanspreekt best niet in de class Main.*

Als je dit wel doet heeft de class Main *meerdere* verantwoordelijkheden:

- De database aanspreken
- Interactie doen met de gebruiker: data tonen en data opvragen.

De class Main wordt dan groot, moeilijk leesbaar en moeilijk onderhoudbaar.

Je programmeert de code waarmee je de database aanspreekt in een aparte class.

Je roept die class op vanuit de Main.

Men noemt een class die de database aanspreekt een *repository* class.

Je maakt een *repository class per gegevenstype in de werkelijkheid die je automatiseert.*

De werkelijkheid bevat in ons voorbeeld drie gegevenstypes: brouwers, bieren en soorten.

Je hebt dus de classes LeveranciersRepository, PlantenRepository en SoortenRepository.

- LeveranciersRepository bevat code die de database table leveranciers aanspreekt.
- PlantenRepository bevat code die de database table planten aanspreekt.
- SoortenRepository bevat code die de database table soorten aanspreekt.

Alle classes gebruiken dezelfde database. Ze hebben dus *connecties nodig met dezelfde JDBC URL, dezelfde gebruikersnaam en hetzelfde paswoord.* Deze informatie herhalen in elk van de classes is verkeerd. Als de informatie wijzigt, moet je *meerdere* classes wijzigen.

Je plaatst de informatie *daarom één keer* in een base class waarvan de repository classes erven.

Je maakt de base class in een package `be.vdab.repositories`:

```
package be.vdab.repositories;
// enkele imports uit de package java.sql
abstract class AbstractRepository {
    private static final String URL = "jdbc:mysql://localhost/tuincentrum?" +
        "useSSL=false&allowPublicKeyRetrieval=true&serverTimezone=Europe/Brussels";
    private static final String USER = "cursist";
    private static final String PASSWORD = "cursist";
    protected Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASSWORD);
    }
}
```

- (1) Je maakt de class abstract: ze dient enkel als base class voor je repository classes. Het is niet de bedoeling dat je van de class een object kan maken. Je geeft de class geen public visibility. De class heeft dan package visibility. Ze is enkel zichtbaar in andere classes van dezelfde package: `be.vdab.repositories`.
- (2) Je zal deze method oproepen in de repository classes, telkens je een Connection nodig hebt.



Repository: zie takenbundel

5 PreparedStatement

De interface `PreparedStatement` stelt een SQL statement voor dat je naar de database stuurt.

Elke JDBC driver bevat een class die deze interface implementeert.

De Connection method `prepareStatement` geeft je een `PreparedStatement`.

Je voert een `insert`, `update` of `delete` statement uit met de `PreparedStatement` 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. Die `int` 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 %.

Je maakt de class `PlantenRepository`:

```
package be.vdab.repositories;

public class PlantenRepository extends AbstractRepository {
    public int verhoogAllePrijzenMet10Procent() throws SQLException {
        String sql = "update planten set prijs = prijs * 1.1";
        try (Connection connection = super.getConnection();           ❶
            PreparedStatement statement = connection.prepareStatement(sql)) { ❷
            return statement.executeUpdate();                             ❸
        }
    }
}
```

(1) Je vraagt een `Connection` aan de method `getConnection` in je base class `AbstractRepository`.

(2) De Connection method `prepareStatement` geeft je een `PreparedStatement` object.

`PreparedStatement` erft van `AutoCloseable`.

Je maakt het `PreparedStatement` daarom ook tussen () van de try opdracht.

De compiler maakt dan code die dit `PreparedStatement` sluit in een `finally` blok.

Je geeft als parameter een SQL statement mee.

(3) De method `executeUpdate` voert het SQL statement bij ❷ uit en geeft het aantal aangepaste records terug.

Je roept de method op in de class `Main`. Je wijzigt de code in de method `main`:

```
PlantenRepository repository = new PlantenRepository();
try {
    System.out.print(repository.verhoogAllePrijzenMet10Procent());
    System.out.println(" planten aangepast.");
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je bekijkt met de MySQL Workbench de prijzen van de planten.

Je voert de applicatie uit.

Je bekijkt daarna met de MySQL Workbench de aangepaste prijzen.

5.1 Samenvatting



Connection maakt `PreparedStatement`.



Bieren verwijderen: zie takenbundel

6 ResultSet

Je voert met de `PreparedStatement` 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 van het type `ResultSet`.

De interface `ResultSet` stelt de rijen voor die het resultaat zijn van het select statement:

SQL statement:

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

Rijen:

2	Baumgarten
7	Bloem

Je leest de 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`.

`next` 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.

De opeenvolging van opdrachten:

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

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

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

6.1 Kolomvolgnummers

Je maakt de class `LeveranciersRepository`. Die bevat een method `findAllNamen`.

De method geeft de namen van alle leveranciers terug:

```
package be.vdab.repositories;
// enkele imports

public class LeveranciersRepository {
    public List<String> findAllNamen() throws SQLException {
        String sql = "select naam from leveranciers";
        try (Connection connection = super.getConnection();
            PreparedStatement statement = connection.prepareStatement(sql);
            ResultSet result = statement.executeQuery()) {
            List<String> namen = new ArrayList<>();
            while (result.next()) {
                namen.add(result.getString(1));
            }
            return namen;
        }
    }
}
```

- (1) De `Statement` method `executeQuery` geeft een `ResultSet` object. `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 de `ResultSet` sluit in een `finally` blok. Je geeft als parameter het select statement mee.
- (2) Je itereert over de rijen in de `ResultSet`.
- (3) Je leest de `String` waarde in de eerste kolom van het select statement.

Je roept de method op in de class Main. Je wijzigt de code in de method main:

```
LeveranciersRepository repository = new LeveranciersRepository();
try {
    for (String naam : repository.findAllNamen()) {
        System.out.println(naam);
    }
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je kan de applicatie uitvoeren.

6.2 Kolomnamen

Kolommen aanduiden met hun volgnummer heeft nadelen:

- ⊖ Als je het SQL statement wijzigt of uitbreidt, kunnen de volgnummers wijzigen.
Als je het SQL statement wijzigt naar `select id, naam from leveranciers` moet je `getString(1)` door `getString(2)`
- ⊖ Als het SQL statement veel kolommen bevat, is de code niet leesbaar.
Bij `getString(13)` moet je visueel in het SQL statement tellen om welke kolom dit gaat.

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

Je wijzigt in `LeveranciersRepository` `getString(1)` naar `getString("naam")`.

Je kan de applicatie uitvoeren.

`ResultSet` bevat naast `getString` andere methods om een kolom waarde te lezen:

- `getInt` een getalwaarde lezen als een `int`.
- `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 kolom type `decimal`.
- `getBoolean` een waarde lezen als een `boolean`.
Je gebruikt deze method meestal bij de kolom types `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 kolom type `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 kolom type `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 kolom type `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** die kolommen **een alias met as**.

Je leest die kolom waarden in Java met die alias. Voorbeeld: Je maakt in LeveranciersRepository een method findAantal. Die geeft de het aantal leveranciers terug:

```
public int findAantal() throws SQLException {
    String sql = "select count(*) as aantal from leveranciers";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql);
        ResultSet result = statement.executeQuery()) {
        result.next();
        return result.getInt("aantal");
    }
}
```

❶

❷

❸

- (1) Je geeft de berekening de alias aantal.
- (2) Het select statement bij ❶ geeft altijd één record. Je plaatst je op dit record.
- (3) Je leest de inhoud van de kolom met de alias aantal.

Je roept de method op in de class Main. Je wijzigt de code in het try blok in de method main:

```
System.out.print("Aantal leveranciers:");
System.out.println(repository.findAantal());
```

Je kan de applicatie uitvoeren.

6.3 select *

Het is een **slechte gewoonte** om in een select statement alle kolommen te lezen met * :

select * from soorten

Als de applicatie in productie gaat, leest die *alle* kolommen uit de table soorten: id en naam.

Dit geeft op termijn problemen:

- ❌ Later worden aan de table kolommen toegevoegd. Jouw applicatie gebruikt die niet, maar andere applicaties wel. Toch leest je applicatie met select* ... *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 kolommen toegevoegd met informatie die slechts enkele applicaties mogen lezen, zoals een kolom met een winstpercentage. Je kan in de database rechten geven zodat de gebruiker root die kolom mag lezen, de gebruiker cursist niet. Zodra de kolom met die 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 de problemen door in je select statement enkel de nodige kolommen te vragen:

select id, naam **from** soorten



Een SQL statement tikken in een String in Java is niet zo **handig** als een SQL statement tikken in de MySQL Workbench: keywords krijgen geen kleuren, je krijgt geen popup vensters die je helpen de opdracht te tikken, ...
Oplossing: je **tikt het SQL statement eerst in de MySQL Workbench en je test het daar.**
Je brengt het daarna met copy-paste over naar je Java code.

6.4 Werkelijkheid

Je applicatie werkt met Leveranciers, Planten, en Soorten.

Bij object oriëntatie stel je ze voor met de classes Leverancier, Plant en Soort.

Je maakt als voorbeeld de class Leverancier. Je maakt die in een package `be.vdab.domain`.

Het woord `domain` wordt regelmatig gebruikt als synoniem voor de te automatiseren werkelijkheid.

```
package be.vdab.domain;
public class Leverancier {
    private static final DateTimeFormatter FORMATTER =
        DateTimeFormatter.ofPattern("dd/MM/yyyy");
    private final long id;
    private final String naam;
    private final String adres;
    private final int postcode;
    private final String woonplaats;
    private final LocalDate sinds;
    public Leverancier(long id, String naam, String adres, int postcode,
        String woonplaats, LocalDate sinds) {
        this.id = id;
        this.naam = naam;
        this.adres = adres;
        this.postcode = postcode;
        this.woonplaats = woonplaats;
        this.sinds = sinds;
    }
    @Override
    public String toString() {
        return id + ":" + naam + " (" + woonplaats + ") " + sinds.format(FORMATTER);
    }
}
```

Je maakt in `LeveranciersRepository` een method `findAll`. Die geeft alle leveranciers terug:

```
public List<Leverancier> findAll() throws SQLException {
    String sql = "select id,naam,adres,postcode,woonplaats,sinds from leveranciers";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql);
        ResultSet result = statement.executeQuery()) {
        List<Leverancier> leveranciers = new ArrayList<>();
        while (result.next()) {
            leveranciers.add(resultNaarLeverancier(result));
        }
        return leveranciers;
    }
}

private Leverancier resultNaarLeverancier(ResultSet result) throws SQLException {
    return new Leverancier(result.getLong("id"), result.getString("naam"),
        result.getString("adres"), result.getInt("postcode"),
        result.getString("woonplaats"), result.getDate("sinds").toLocalDate());
}
```

(1) Je maakt per rij uit de `ResultSet` een `Leverancier` object met de hulp van je method `resultNaarLeverancier` (bij ②) en je voegt die `Leverancier` toe aan de lijst.

Je roept de method op in de class `Main`. Je wijzigt de code in het `try` blok in de method `main`:

```
for (Leverancier leverancier : repository.findAll()) {
    System.out.println(leverancier);
}
```

Je kan de applicatie uitvoeren.

6.5 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 `ResultSet`s die je tot nu maakt zijn standaard `ResultSet`s

Je wijzigt de 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 `ResultSet`s.
- Scrollable `ResultSet`s gebruiken meestal meer RAM dan Forward-Only `ResultSet`s.
- Scrollable `ResultSet`s zijn meestal trager dan Forward-Only `ResultSet`s.

Je gebruikt daarom in deze cursus geen scrollable `ResultSet`s.

- 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 `ResultSet`s.
- Je kan van sommige SQL select statements geen updatable `ResultSet` maken.

Je gebruikt daarom in deze cursus geen updatable `ResultSet`s.

6.6 Samenvatting



`PreparedStatement` maakt `ResultSet`.



Gemiddelde: zie takenbundel

7 SQL statements met parameters

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

De gebruiker tikt bijvoorbeeld een woonplaats. Je toont de leveranciers uit die woonplaats.

De 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, schrijf je rond ? geen quotes.
 Je kan meerdere veranderlijke onderdelen voorstellen met meerdere vraagtekens.
2. Je geeft dit SQL statement mee aan een PreparedStatement object.
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.

7.1 Voorbeeld

Je maakt in `LeveranciersRepository` een method `findByWoonplaats`.

Die geeft de leveranciers uit één woonplaats terug:

```
public List<Leverancier> findByWoonplaats(String woonplaats) throws SQLException {
    String sql =
        "select id,naam,adres,postcode,woonplaats from leveranciers where woonplaats=?";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        statement.setString(1, woonplaats);
        try (ResultSet result = statement.executeQuery()) {
            List<Leverancier> leveranciers = new ArrayList<>();
            while (result.next()) {
                leveranciers.add(resultNaarLeverancier(result));
            }
            return leveranciers;
        }
    }
}
```

- (1) Je vult de eerste (en hier enige) parameter met de te zoeken woonplaats.

Je roept de method op in de class `Main`. Je wijzigt de code in de method `main`:

```
System.out.print("Woonplaats:");
Scanner scanner=new Scanner(System.in);
String woonplaats=scanner.nextLine();
LeveranciersRepository repository = new LeveranciersRepository();
try {
    for (Leverancier leverancier : repository.findByWoonplaats(woonplaats)) {
        System.out.println(leverancier);
    }
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je kan de applicatie uitvoeren.



Als je SQL statement eerst uitprobeert in de MySQL Workbench, vervang je daar het ? door een voorbeeld waarde (bvb. 'Kortrijk').

7.2 SQL code injection

Je mag een SQL statement met parameters niet vervangen door een SQL statement waarin je stukken SQL concateneert met gebruikersinvoer. Dan misbruiken hackers je programma.

Je ziet dit met een nieuwe method in PlantenRepository:

```
public int verhoogPrijzenMet10ProcentByNaam(String naam) throws SQLException {
    String sql =
        "update planten set prijs = prijs * 1.1 where naam = '" + naam + "'";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        return statement.executeUpdate();
    }
}
```

(1) Je concateneert stukken SQL met gebruikersinvoer (de gebruiker tikt de naam van de plant straks in de class Main) tot één SQL statement.

Je roept de method op in de class Main. Je wijzigt de code in de method main:

```
System.out.print("Naam:");
Scanner scanner = new Scanner(System.in);
String naam = scanner.nextLine();
PlantenRepository repository = new PlantenRepository();
try {
    System.out.print(repository.verhoogPrijzenMet10ProcentByNaam(naam));
    System.out.println(" plant(en) aangepast.");
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je voert het programma uit als een gewone gebruiker: Je tikt de naam Linde.

Er is geen probleem: je programma wijzigt één plant.

Je voert het programma uit als hacker: Je tikt ' or ''='

Er is een groot probleem: je programma wijzigt *alle* planten! Dit komt omdat je nu volgend SQL statement uitvoert, waarbij het where onderdeel bij alle records true teruggeeft, omdat de voorwaarde na or altijd true teruggeeft:

```
update planten set prijs = prijs * 1.1 where naam=' or ''='
```

Dit heet SQL code injection: de hacker tikt SQL code waar jij dit niet verwacht had.

Je verhindert SQL code injection met een SQL statement met een parameter.

Je wijzigt in PlantenRepository de code in de method verhoogPrijzenMet10ProcentByNaam

```
String sql = "update planten set prijs = prijs * 1.1 where naam =?";
try (Connection connection = super.getConnection();
    PreparedStatement statement = connection.prepareStatement(sql)) {
    statement.setString(1, naam);
    return statement.executeUpdate(sql);
}
```

Als een hacker nu ' or '=' intikt, wijzigt je applicatie geen enkele plant.



Naast PreparedStatement bestaat ook Statement.
Statement ondersteunt geen parameters in SQL statements
en is meestal trager dan PreparedStatement.



Van tot: zie takenbundel

8 Een record lezen aan de hand van zijn id

LeveranciersRepository bevat nu volgende method die data *lezen*:

findAllNamen, findAantal, findAll en findByIdWoonplaats.

- findAantal verwerkt een ResultSet die *altijd* één rij bevat: het aantal leveranciers.
findAantal voert op de ResultSet één keer de next method uit en plaatst zich zo op die rij.
- findAllNamen, findAll en findByIdWoonplaats itereren met while over de ResultSet.
Dit is omdat de ResultSet nul tot een ongekend aantal rijen kan bevatten.

Als je een record zoekt op zijn id (primary key) bevat de ResultSet ofwel nul rijen (als je het record niet gevonden hebt), of één rij (als je het record gevonden hebt).
Mooie code itereert dan niet met while over de ResultSet, maar gebruikt een if.

Je voegt een method toe aan LeveranciersRepository om dit te proberen:

```
public Optional<Leverancier> findById(long id) throws SQLException {
    String sql =
        "select id,naam,adres,postcode,woonplaats,sinds from leveranciers where id = ?";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        statement.setLong(1, id);
        try (ResultSet result = statement.executeQuery()) {
            if (result.next()) {
                return Optional.of(resultNaarLeverancier(result));
            }
            return Optional.empty();
        }
    }
}
```

- (1) De method zoekt een leverancier waarvan je de id kent.
De method geeft een Optional terug.
De Optional is "empty" als je de leverancier *niet* vindt.
De Optional bevat een Leverancier object als je de leverancier *wel* vindt.
- (2) Je probeert je op de rij van de ResultSet te plaatsen.
- (3) Je zet de rij om naar een Leverancier object. Je vult met dit object een Optional object.
- (4) De ResultSet is leeg. Je geeft een "lege" Optional terug.

Je roept de method op in de class Main. Je wijzigt de code in de method main:

```
System.out.print("id:");
Scanner scanner = new Scanner(System.in);
long id = scanner.nextLong();
LeveranciersRepository repository = new LeveranciersRepository();
try {
    Optional<Leverancier> optionalLeverancier = repository.findById(id);
    if(optionalLeverancier.isPresent()) {
        Leverancier leverancier = optionalLeverancier.get();
        System.out.println(leverancier);
    } else {
        System.out.println("Leverancier niet gevonden.");
    }
}
catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je kan de applicatie uitvoeren.



Id: zie takenbundel

9 Stored procedure

Een stored procedure is een verzameling 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 in de stored procedure uit.

Een stored procedure kan parameters bevatten.

Je geeft waarden voor de parameters mee bij het oproepen van de stored procedure.

9.1 Stored procedure maken

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:

```
create procedure tuincentrum.PlantNamenMetEenWoord (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 begin.
- (3) Je gebruikt de waarde in de parameter woord. Je tikt na elk SQL statement een ;
- (4) Je eindigt de stored procedure met end.

Je maakt de stored procedure met de knoppen Apply, Apply en Finish.

9.2 Stored procedure testen

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

```
grant execute on procedure PlantNamenMetEenWoord to cursist
```

Je kan de stored procedure uitvoeren met de MySQL Workbench.

```
call tuincentrum.PlantNamenMetEenWoord('%bloem%')
```

9.3 CallableStatement

Je roept in je Java code een stored procedure op met een CallableStatement object. CallableStatement erft van PreparedStatement.

Je voegt in AbstractRepository het volgende toe achter in de de variabele URL:

```
&noAccessToProcedureBodies=true
```

①

- (1) Je plaatst bij MySQL in de JDBC URL de parameter noAccessToProcedureBodies op true om een stored procedure aan te spreken. Zoniet krijg je een exception.

Je voegt een method toe aan PlantenRepository:

```
public List<String> findNamenByWoord(String woord) throws SQLException {
    String call = "{call PlantNamenMetEenWoord(?)}";
    try (Connection connection = super.getConnection();
        CallableStatement statement = connection.prepareCall(call)) {
        statement.setString(1, '%' + woord + '%');
        try (ResultSet result = statement.executeQuery()) {
            List<String> namen = new ArrayList<>();
```

①
②
③
④

```

        while (result.next()) {
            namen.add(result.getString("naam"));
        }
        return namen;
    }
}
}

```

- (1) Je geeft deze String bij ❷ mee bij het aanspreken van de stored procedure.
 - Je tikt {call voor de naam van de stored procedure.
 - Je stelt de stored procedure parameter(s) voor met ?
 - Je sluit af met }.
- (2) Je geeft aan de Connection method prepareCall de string bij ❶ mee. 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. Als de gebruiker bloem tikt word het like onderdeel van het select statement in de stored procedure dus '%bloem%'.
 - Je concateneert % voor en na het woord dat de gebruiker intikte.
 - Als de gebruiker bloem tikt word het like onderdeel van het select statement in de stored procedure dus '%bloem%'.
- (4) Je voert de stored procedure uit. Je krijgt een ResultSet terug.

Je roept de method op in de class Main. Je wijzigt de code in de method main:

```

System.out.print("Woord:");
Scanner scanner = new Scanner(System.in);
String woord = scanner.nextLine();
PlantenRepository repository = new PlantenRepository();
try {
    for (String naam:repository.findNamenByWoord(woord)) {
        System.out.println(naam);
    }
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}

```

Je kan de applicatie uitvoeren.

Voordelen van een stored procedure:

- ⊕ Je schrijft een SQL statement gemakkelijker over meerdere regels 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 ze sneller uit dan dat je ze vanuit een applicatie één per één naar de database stuurt.
- ⊕ 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 database merk. Als je verandert van merk, moet je de sleutelwoorden van het nieuwe merk leren kennen.

De meeste Java ontwikkelaars vinden de nadelen belangrijker dan de voordelen.

Ze gebruiken daarom zelden stored procedures.



Stored procedure: zie takenbundel

10 Transactie

Je stuurt in veel applicatie onderdelen meerdere SQL statements naar de database, die **al** de statements moet uitvoeren, of **geen enkel** van de 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 stuurt daartoe twee statements naar de database:

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

De gebruiker start de applicatie. Je 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 planten prijzen onder € 100 niet aangepast.
- ➔ Als hij het onderdeel nog eens uitvoert, past hij de planten prijzen vanaf € 100 nog eens aan.

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

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

10.1 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.

10.2 Commit en rollback

Nadat je die SQL statements uitvoert roep je de Connection method `commit` op. `commit` 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`.

10.3 Voorbeeld

Dit is het uitgewerkte voorbeeld zoals eerder beschreven.

Je voegt een method toe aan `PlantenRepository`:

```
public void verhoogPrijzenBovenEnOnder100€() throws SQLException {
    String sqlVanaf100 = "update planten set prijs = prijs*1.1 where prijs >= 100";
    String sqlTot100 = "update planten set prijs = prijs*1.05 where prijs < 100";
    try (Connection connection = super.getConnection();
        PreparedStatement statementVanaf100 = connection.prepareStatement(sqlVanaf100);
        PreparedStatement statementTot100 = connection.prepareStatement(sqlTot100)) {
        connection.setAutoCommit(false);
        statementVanaf100.executeUpdate();
        statementTot100.executeUpdate();
        connection.commit();
    }
}
```

- (1) Je zet de autocommit mode af.
Alle SQL statements die je vanaf nu op de Connection uitvoert behoren tot één transactie.
- (2) De database voert dit statement uit binnen de transactie die je startte bij ❶.
- (3) De database voert ook dit statement uit binnen dezelfde transactie die je startte bij ❶.
- (4) Nadat je beide statements kon uitvoeren, doe je een commit. De database legt 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 roept de method op in de class Main. Je wijzigt de code in het try blok in de method main:

```
repository.verhoogPrijzenBovenEnOnder100€();
```

Je bekijkt met de MySQL Workbench de prijzen van de planten.

Je voert de applicatie uit.

Je bekijkt daarna met de MySQL Workbench de aangepaste prijzen.

Je ziet een rollback aan het werk met volgende handelingen.

Je vervangt in het tweede SQL statement update door `update`. 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.

10.4 Samenvatting



Connection

maakt



Transaction

verzamelt



Statements



Failliet: zie takenbundel

11 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 je transactie dezelfde data wijzigen die jouw transactie leest. Je 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 de 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 de 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.

11.1 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 exception. Je wil niet dat de gebruiker die exception ziet.

Oplossing 1: het transaction isolation level Serializable:

Je maakt een package `be.vdab.exceptions`. Je maakt daarin de class `SoortBestaatAlException`:

```
package be.vdab.exceptions;
public class SoortBestaatAlException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

Je maakt in de package `be.vdab.repositories` de class `SoortenRepository`:

```
package be.vdab.repositories;
// enkele imports
public class SoortenRepository {
    public void create(String naam) throws SQLException {
        String select = "select id from soorten where naam = ?";
        String insert = "insert into soorten(naam) values (?)";
        try (Connection connection = super.getConnection();
            PreparedStatement statementSelect = connection.prepareStatement(select)) {
            statementSelect.setString(1, naam);
            connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE); ❶
            connection.setAutoCommit(false);
            try (ResultSet result = statementSelect.executeQuery()) {
                if (result.next()) {
                    connection.commit(); ❷
                    throw new SoortBestaatAlException(); ❸
                } else {
                    try (PreparedStatement statementInsert = connection.prepareStatement(insert)) {
                        statementInsert.setString(1, naam);
                        statementInsert.executeUpdate();
                        connection.commit();
                    }
                }
            }
        }
    }
}
```

- (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.
- (3) Het record bestaat al. Je commit de transactie. Dit lijkt op onnodig: je hebt geen record toegevoegd. Sommige databases vergrendelen tijdens een transactie records die je leest, zodat andere gebruikers deze records niet kunnen wijzigen terwijl jouw applicatie die records gebruikt. Door de transactie te committen ontgrendel je die records zo snel mogelijk.

Je roept de method op in de class Main. Je wijzigt de code in de method main:

```
System.out.print("Naam:");
Scanner scanner = new Scanner(System.in);
String naam = scanner.nextLine();
SoortenRepository repository = new SoortenRepository();
try {
    repository.create(naam);
    System.out.println("Soort toegevoegd.");
}
```

```

} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
catch (SoortBestaatAlException ex) {
    System.out.println("Soort bestaat al.");
}

```

Je kan het programma uitvoeren.

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

1. Je plaatst een breakpoint op de regel `if (resultSet.next()) {`
Je doet dit door te klikken in de grijze marge voor de regel. Je ziet een rode bol in de marge.
Je debugt straks de code IntelliJ zal de uitvoering juist voor de regel pauzeren.
2. Je kiest in het menu Run de opdracht Debug 'Main'.
3. Je tikt in het tabblad Console (beneden in IntelliJ) 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 blauwe achtergrond.
5. Je laat IntelliJ 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 IntelliJ.
8. Je schakelt terug naar IntelliJ.
9. Je laat je programma verder lopen met de knop  links onder.
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 IntelliJ afgelopen is, kon de MySQL WorkBench proberen het insert statement uit te voeren.
Dit mislukt omdat je programma in IntelliJ reeds een record met dezelfde naam toevoegde.
Je ziet dit onder in de MySQL WorkBench naast je statement:
Error Code: 1062. Duplicate entry 'test' for key 'naam'

De oplossing heeft 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.

Oplossing 2: het isolation level Read Committed.

Je wijzigt in SoortenRepository de code in de method create:

```

String select = "select id from soorten where naam=?";
String insert = "insert into soorten(naam) values (?)";
try (Connection connection = super.getConnection();
    PreparedStatement statementInsert = connection.prepareStatement(insert)) {
    statementInsert.setString(1, naam);
    connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
    connection.setAutoCommit(false);
    try {
        statementInsert.executeUpdate(); // ❶
        connection.commit();
    } catch (SQLException ex) { // ❷
        try (PreparedStatement statementSelect = connection.prepareStatement(select)) {
            statementSelect.setString(1, naam);
            try (ResultSet result = statementSelect.executeQuery()) { // ❸
                if (result.next()) {
                    connection.commit();
                    throw new SoortBestaatAlException();
                }
            }
        }
    }
}

```

```
        connection.commit();  
        throw ex;  
    }  
}  
}
```

- (1) Je probeert een record met de ingetikte soortnaam toe te voegen.
Dit lukt als er nog geen record met die soortnaam bestaat.
- (2) Het toevoegen is mislukt. Eerste mogelijke reden: er bestaat reeds een record met die soortnaam. Andere redenen : database is uitgevallen, syntax fout in de SQL, ...
- (3) Om te weten of het toevoegen mislukte wegens de eerste reden zoek je of reeds een record voorkomt met de ingetikte soortnaam.

Je kan het programma uitvoeren.

De oplossing heeft 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.



Isolation: zie takenbundel

12 Autonumber kolom

Als je een record toevoegt, geef je geen waarde mee voor een autonumber kolom:

de database vult zelf een waarde in. Je kan na het toevoegen de waarde vragen in twee 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 `PreparedStatement` de method `getGeneratedKeys` uit. Die geeft een `ResultSet`. Die bevat één rij en één kolom. De kolom waarde is de inhoud van de autonumber kolom in het toegevoegde record.

12.1 Voorbeeld

Je wijzigt in `SoortenRepository` de method `create`.

Die geeft het nummer van de toegevoegde soort terug.

```
public long create(String naam) throws SQLException {
    String select = "select id from soorten where naam=?";
    String insert = "insert into soorten(naam) values (?)";
    try (Connection connection = super.getConnection();
        PreparedStatement statementInsert = connection.prepareStatement(insert,
            PreparedStatement.RETURN_GENERATED_KEYS)) { ❶
        statementInsert.setString(1, naam);
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        try {
            statementInsert.executeUpdate();
            try (ResultSet result = statementInsert.getGeneratedKeys()) { ❷
                result.next(); ❸
                long nieuweId = result.getLong(1); ❹
                connection.commit();
                return nieuweId;
            }
        } catch (SQLException ex) {
            // de code hier blijft dezelfde als vroeger
        }
    }
}
```

- (1) Je vermeldt `RETURN_GENERATED_KEYS` waar je het insert statement specificeert.
- (2) De method `getGeneratedKeys` geeft een `ResultSet` met de autonumber waarde.
- (3) Je plaatst je op de eerste `ResultSet` rij met de `next` method.
- (4) Je leest de inhoud van de eerste kolom. Je spreekt de kolom aan met zijn volgnummer omdat de kolom geen naam heeft. De kolom inhoud is de autonumber waarde.

Je roept de method op in de class `Main`. Je wijzigt de code in de method `main`:

```
System.out.print("Naam:");
Scanner scanner = new Scanner(System.in);
String naam = scanner.nextLine();
SoortenRepository repository = new SoortenRepository();
try {
    long nieuweId = repository.create(naam);
    System.out.println("Soort toegevoegd. Het nummer is " + nieuweId);
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
catch (SoortBestaatAlException ex) {
    System.out.println("Soort bestaat al.");
}
```

Je kan de applicatie uitvoeren.

13 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 database merk een datum of tijd in een ander formaat
- welke datum functies en tijd functies je kan oproepen in een SQL statement.
Elk database merk heeft zijn eigen datum functies en tijd functies.

Dit maakt het moeilijk om een applicatie te schrijven die werkt met meerdere database merken.

JDBC bevat de nodige voorzieningen om de problemen op te lossen.

13.1 Letterlijke datum of tijd 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 de datum op correcte manier door naar elk database merk.

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: de lijst van leveranciers die vanaf het jaar 2000 leverancier zijn.

Je voegt een method toe aan LeveranciersRepository:

```
public List<Leverancier> findBySinds2000() throws SQLException {
    String sql = "select id,naam,adres,postcode,woonplaats,sinds from leveranciers" +
        "where sinds >= {d '2000-01-01'}";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        try (ResultSet result = statement.executeQuery()) {
            List<Leverancier> leveranciers = new ArrayList<>();
            while (result.next()) {
                leveranciers.add(resultNaarLeverancier(result));
            }
            connection.commit();
            return leveranciers;
        }
    }
}
```

Je roept de method op in de class Main. Je wijzigt de code in de method main:

```
LeveranciersRepository repository = new LeveranciersRepository();
try {
    for (Leverancier leverancier : repository.findBySinds2000()) {
        System.out.println(leverancier);
    }
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je kan de applicatie uitvoeren.

13.2 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 (?).

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

Je voegt een method toe aan LeveranciersRepository:

```
public List<Leverancier> findBySindsVanaf(LocalDate datum) throws SQLException {
    String sql = "select id,naam,adres,postcode,woonplaats,sinds from leveranciers" +
        "where sinds >= ?";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        statement.setDate(1, java.sql.Date.valueOf(datum));
        try (ResultSet result = statement.executeQuery()) {
            List<Leverancier> leveranciers = new ArrayList<>();
            while (result.next()) {
                leveranciers.add(resultNaarLeverancier(result));
            }
            connection.commit();
            return leveranciers;
        }
    }
}
```

- (1) 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 roept de method op in de class Main. Je wijzigt de code in de method main:

```
System.out.print("Datum vanaf (dd/mm/yyyy):");
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("d/M/y");
Scanner scanner = new Scanner(System.in);
LocalDate datum = LocalDate.parse(scanner.nextLine(), formatter);
LeveranciersRepository repository = new LeveranciersRepository();
try {
    for (Leverancier leverancier : repository.findBySindsVanaf(datum)) {
        System.out.println(leverancier);
    }
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je kan de applicatie uitvoeren.

13.3 Functies

JDBC bevat datum en tijd functies die elk database merk 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 die functies op met volgende syntax:

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

Voorbeeld: een lijst van leveranciers die in het jaar 2000 leverancier werden.

Je voegt een method toe aan `LeveranciersRepository`:

```
public List<Leverancier> findLeverancierGewordenInHetJaar2000() throws SQLException {
    String sql = "select id,naam,adres,postcode,woonplaats,sinds from leveranciers" +
        "where {fn year(sinds)} =2000";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        try (ResultSet result = statement.executeQuery()) {
            List<Leverancier> leveranciers = new ArrayList<>();
            while (result.next()) {
                leveranciers.add(resultNaarLeverancier(result));
            }
            connection.commit();
            return leveranciers;
        }
    }
}
```

Je roept de method op in de class `Main`. Je wijzigt de code in de method `main`:

`LeveranciersRepository repository = new LeveranciersRepository();`

```
try {
    for (Leverancier leverancier: repository.findLeverancierGewordenInHetJaar2000()) {
        System.out.println(leverancier);
    }
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je kan de applicatie uitvoeren.



Bieren van een maand: zie takenbundel

14 Lock

Als je een record leest, kan je dit record vergrendelen (locken) tot het einde van je transactie. Andere gebruikers kunnen dit record niet wijzigen of verwijderen terwijl jij het vergrendelt.

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

Je maakt een class PlantNietGevondenException:

```
package be.vdab.exceptions;
public class PlantNietGevondenException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

Je maakt een class PrijsTeLaagException:

```
package be.vdab.exceptions;
public class PrijsTeLaagException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

Je voegt een method toe aan PlantenRepository:

```
public void verlaagPrijs(long id, BigDecimal nieuwePrijs) throws SQLException {
    String sqlSelect = "select prijs from planten where id = ? for update"; ❶
    try (Connection connection = super.getConnection();
        PreparedStatement statementSelect = connection.prepareStatement(sqlSelect)) {
        statementSelect.setLong(1, id);
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        try (ResultSet result = statementSelect.executeQuery()) { ❷
            if (result.next()) { ❸
                BigDecimal oudePrijs = result.getBigDecimal("prijs");
                BigDecimal minimumNieuwePrijs = oudePrijs.divide(BigDecimal.valueOf(2),
                    2, RoundingMode.HALF_UP);
                if (nieuwePrijs.compareTo(minimumNieuwePrijs) >= 0) { ❹
                    String sqlUpdate = "update planten set prijs = ? where id = ?";
                    try (PreparedStatement statementUpdate =
                        connection.prepareStatement(sqlUpdate)) {
                        statementUpdate.setBigDecimal(1, nieuwePrijs);
                        statementUpdate.setLong(2, id);
                        statementUpdate.executeUpdate(); ❺
                        connection.commit();
                        return;
                    }
                }
            }
            connection.rollback();
            throw new PrijsTeLaagException();
        }
        connection.rollback();
        throw new PlantNietGevondenException();
    }
}
```

- (1) Je definieert met `for update` dat je het gelezen record lockt tot het einde van de transactie.
- (2) Je zoekt de plant. Als je ze gevonden hebt, wordt ze door de database gelocked. Andere gebruikers kunnen tot het einde van je transactie de plant niet wijzigen. Je verhindert zo dat meerdere gebruikers *tegelijk* de prijs aanpassen.
- (3) Je controleert of je de plant gevonden hebt.
- (4) Je controleert of de nieuwe prijs minstens de helft van de gelezen prijs is.
- (5) Je wijzigt de prijs.

Je roept de method op in de class Main. Je wijzigt de code in de method main:

```
Scanner scanner = new Scanner(System.in);
System.out.print("Nummer plant:");
long id = scanner.nextLong();
System.out.print("Nieuwe prijs:");
BigDecimal nieuwePrijs = scanner.nextBigDecimal();
PlantenRepository repository = new PlantenRepository();
try {
    repository.verlaagPrijsTotMaximumHelft(id, nieuwePrijs);
    System.out.println("Prijs aangepast");
} catch (PlantNietGevondenException ex) {
    System.out.println("Plant niet gevonden.");
} catch (PrijsTeLaagException ex) {
    System.out.println("Prijs te laag.");
}
catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je kan het programma uitvoeren.

De oplossing stuurt twee SQL statements naar de database.

Er is een betere oplossing waarbij je meestal slechts één SQL statement naar de database stuurt:

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 = 50 where id = 1 and 50 > prijs / 2`
 Je wijzigt het record dus enkel als 50 (de nieuwe prijs) groter is dan de huidige prijs in het record gedeeld door 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 die mogelijkheden zijn opgetreden, zoek je de plant in de database.
 Als je die vindt, is de tweede mogelijkheid opgetreden.

Je wijzigt op die manier de method verlaagPrijs in PlantenRepository:

```
public void verlaagPrijs(long id, BigDecimal nieuwePrijs) throws SQLException {
    String sqlUpdate = "update planten set prijs = ? where id = ? and ? > prijs/2";
    try (Connection connection = super.getConnection();
        PreparedStatement statementUpdate = connection.prepareStatement(sqlUpdate)) {
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        statementUpdate.setBigDecimal(1, nieuwePrijs);
        statementUpdate.setLong(2, id);
        statementUpdate.setBigDecimal(3, nieuwePrijs);
        int aantalAangepast = statementUpdate.executeUpdate();
        if (aantalAangepast == 1) {
            connection.commit();
            return;
        }
    }
```

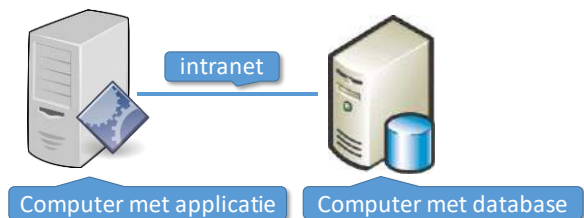
```
String sqlSelect = "select count(*) as aantal from planten where id = ?";
try (PreparedStatement statemenSelect=connection.prepareStatement(sqlSelect)){
    statemenSelect.setLong(1, id);
    try (ResultSet result = statemenSelect.executeQuery()) {
        result.next();
        if (result.getLong("aantal") == 0) {
            connection.rollback();
            throw new PlantNietGevondenException();
        }
        connection.rollback();
        throw new PrijsTeLaagException();
    }
}
}
```

Je kan het programma uitvoeren.

15 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, draaiend op verschillende computers, kunnen dezelfde database delen.



Telkens je vanuit je applicatie de database aanspreekt, gebruik je

- ⊖ het intranet. Het intranet is ongeveer 1.000 keer trager dan RAM geheugen.
- ⊖ de harddisk (de database server zoekt de data op zijn harddisk)
Een harddisk ongeveer 100.000 keer trager is dan RAM geheugen.

Het aanspreken van de database is dus het onderdeel in je code dat je best optimaliseert. Je leert hieronder belangrijke tips.

15.1 Enkel de records lezen 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
```

Volgende oplossing is slecht als je maar *een deel* van de records nodig hebt.

Voorbeeld: je hebt slechts de leveranciers uit één woonplaats nodig.

1. alle records uit de database lezen (`select id, naam, woonplaats from leveranciers`)
2. in Java code de records filteren die je nodig hebt

Dit is slecht: de database leest *alle* leveranciers van de harde schijf en stuurt *alle* leveranciers over het netwerk naar je applicatie. Die itereert daarna over *alle* leveranciers om slechts enkele te tonen.

Voorbeeld van zo'n slechte oplossing: je wil enkel de leveranciers uit één gemeente

```

public List<Leverancier> findByWoonplaats(String woonplaats) throws SQLException {
    String sql = "select id,naam,adres,postcode,woonplaats,sinds from leveranciers";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        try (ResultSet result = statement.executeQuery()) {
            List<Leverancier> leveranciers = new ArrayList<>();
            while (result.next()) {
                if ("Wevelgem".equals(result.getString("woonplaats"))) {
                    leveranciers.add(resultNaarLeverancier(result));
                }
            }
            return leveranciers;
        }
        connection.commit();
    }
}
  
```

(1) Je leest *alle* records uit de database.

(2) Je neemt enkel de gelezen records met een woonplaats Wevelgem op in het resultaat.

Je kan met de MySQL Workbench zien dat de oplossing in de database zelf traag is.

Je tikt de opdracht **select** id, naam, woonplaats **from** leveranciers en je klikt op .

Je ziet een overzicht met de stappen die MySQL intern doet om deze query uit te voeren.

Je ziet **Full Table Scan**. Je laat er de muisaanwijzer op rusten. Je ziet een detailvenster.

Je ziet daarin Cost Hint: Very High. Dit betekent dat als de table leveranciers honderden records bevat (wat normaal is), de query traag wordt.

Goede oplossing: je leest enkel de records die je nodig hebt via where deel in het select statement:

select id, naam, woonplaats **from** leveranciers where **woonplaats=wevelgem'**

De database stuurt enkel de leveranciers uit Wevelgem over het netwerk naar de applicatie.

Die itereert enkel over de leveranciers uit Wevelgem. Zo is de oplossing veel sneller.

Je vindt de oplossing al in de bestaande method `findByWoonplaats` in `LeveranciesRepository`.

Je kan de oplossing nog wat optimaliseren met transactiebeheer:


```
public List<Leverancier> findByWoonplaats(String woonplaats) throws SQLException {
    String sql =
        "select id,naam,adres,postcode,woonplaats,sinds from leveranciers where woonplaats=?";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        statement.setString(1, woonplaats);
        try (ResultSet result = statement.executeQuery()) {
            List<Leverancier> leveranciers = new ArrayList<>();
            while (result.next()) {
                leveranciers.add(resultNaarLeverancier(result));
            }
            connection.commit();
            return leveranciers;
        }
    }
}
```

De goede oplossing kan een extra voordeel hebben. Als er een index ligt op de kolom woonplaats, leest de database in die index snel welke leveranciers als woonplaats Wevelgem hebben.

De database moet enkel van die leveranciers de id en naam lezen uit de table.

Dit is het geval in de huidige database.

Je ziet met de MySQL Workbench dat deze oplossing in de database zelf sneller is. Je tikt de opdracht

select id, naam, woonplaats **from** leveranciers where **woonplaats='wevelgem'** en je kiest .

Je ziet een overzicht met de stappen die MySQL intern doet om deze query uit te voeren.

Je ziet **Non-Unique Key Lookup**. Je laat er de muisaanwijzer op rusten. Je ziet een detailvenster.

Je ziet daarin Cost Hint: Low-Medium. Dit betekent dat MySQL de query snel kan uitvoeren

15.2 Het SQL keyword in

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

Trage oplossing: drie 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

Goede oplossing: slechts één select statement naar de database te sturen:

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

Voorbeeld: de gebruiker tikt enkele plantnummers. Jij toont de namen van die planten.

Je voegt een method toe aan PlantenRepository:

```
public List<String> findNamenByIds(Set<Long> ids) throws SQLException {
    StringBuilder sql = new StringBuilder("select naam from planten where id in (");
    for (int teller = 0; teller != ids.size(); teller++) {
        sql.append("?,");
    }
    sql.setCharAt(sql.length() - 1, ')');
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql.toString())) {
        int index = 1;
        for (long id : ids) {
            statement.setLong(index++, id);
        }
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        try (ResultSet result = statement.executeQuery()) {
            List<String> namen=new ArrayList<>();
            while (result.next()) {
                namen.add(result.getString("naam"));
            }
            connection.commit();
            return namen;
        }
    }
}
```

Je roept de method op in de class Main. Je wijzigt de code in de method main:

```
Set<Long> ids = new HashSet<>();
Scanner scanner = new Scanner(System.in);
System.out.print("Nummer plant (0 om te stoppen):");
long id = scanner.nextInt();
while (id != 0) {
    ids.add(id);
    System.out.print("Nummer plant (0 om te stoppen):");
    id = scanner.nextInt();
}
PlantenRepository repository = new PlantenRepository();
try {
    for (String naam : repository.findNamenByIds(ids)) {
        System.out.println(naam);
    }
}
catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je kan het programma uitvoeren.

Je kan de 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)`

De optimalisatie is ook bruikbaar 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)`

15.3 Join

Je hebt in veel overzichten data uit gerelateerde tables nodig.

Als je data leest uit de n kant van een relatie en je hebt ook 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:
`select naam, leverancierid from planten where kleur = 'rood'`
 Dit geeft 12 planten. De 1° plant heeft de leverancierid 6, de 2° plant heeft leverancierid 8, ...
- ➔ daarna over de gelezen planten te itereren en per plant met een SQL select statement de bijbehorende leverancier lezen:
`select naam from leveranciers where id = 6`
`select naam from leveranciers where id = 6`
 ...

Dit wordt traag door de overvloed van SQL statements die je uitvoert.

De goede oplossing is de rode planten én hun leveranciers met één select statement lezen.

Je gebruikt hierbij in je select statement een join tussen de table planten en de table leveranciers.

Je maakt een package `be.vdab.dto`. DTO is een afkorting voor Data Transfer Object. Een DTO stelt geen data voor uit de werkelijkheid, zoals Leverancier, maar die je toch leest uit de database.

Je maakt in die package de class `PlantNaamEnLeveranciersNaam`:

```
package be.vdab.dto;
public class PlantNaamEnLeveranciersNaam {
    private final String plantNaam;
    private final String leveranciersNaam;
    // je maakt een geparametriseerde constructor en getters
}
```

Je voegt een method toe aan `PlantenRepository`:

```
public List<PlantNaamEnLeveranciersNaam> findRodePlantenEnHunLeveranciers() throws
SQLException {
    String sql = "select planten.naam as plantnaam, " +
        "leveranciers.naam as leveranciersnaam " +
        "from planten inner join leveranciers on planten.leverancierid = leveranciers.id" +
        "where kleur = 'rood'";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        try (ResultSet result = statement.executeQuery()) {
            List<PlantNaamEnLeveranciersNaam> list = new ArrayList<>();
            while (result.next()) {
                list.add(new PlantNaamEnLeveranciersNaam(result.getString("plantnaam"),
                    result.getString("leveranciersnaam")));
            }
            connection.commit();
            return list;
        }
    }
}
```

Je roept de method op in de class Main. Je wijzigt de code in de method main:

```
PlantenRepository repository=new PlantenRepository();
try {
    for (PlantNaamEnLeveranciersNaam naamDuo :
        repository.findRodePlantenEnHunLeveranciers()) {
        System.out.println(naamDuo.getPlantNaam() + " - " +
            naamDuo.getLeveranciersNaam());
    }
}
catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je kan de applicatie uitvoeren.

15.4 Batch update

Je wil een SQL statement soms *meerdere keren* uitvoeren, waarbij je bij elke uitvoering andere waarden meegeeft voor de ? tekens in het SQL statement.

Voorbeeld: de volgende method in SoortenRepository krijgt een verzameling soortnamen als parameter binnen. De method voegt per soortnaam een record toe aan de table soorten:

```
public void create(List<String> namen) throws SQLException {
    String sql = "insert into soorten(naam) values (?)";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        for (String naam : namen) {
            statement.setString(1, naam);
            statement.executeUpdate();
        }
        connection.commit();
    }
}
```

- (1) De parameter namen bevat de namen van de toe te voegen soorten.
- (2) Je itereert over de namen.
- (3) Je vult met de naam het ? teken in het insert statement.
- (4) Je voert het insert statement uit.

Je roept de method op in de class Main. Je wijzigt de code in de method main:

```
List<String> namen = new ArrayList<>();
Scanner scanner = new Scanner(System.in);
System.out.print("Naam (stop met STOP):");
String naam = scanner.nextLine();
while (!"STOP".equals(naam)) {
    namen.add(naam);
    System.out.print("Naam (stop met STOP):");
    naam = scanner.nextLine();
}
try {
    SoortenRepository repository = new SoortenRepository();
    repository.create(namen);
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Je kan de applicatie uitvoeren.

Je voert het insert statement bij ❹ meerdere keren uit. Je applicatie stuurt het insert statement als een netwerkpakket naar de computer waarop MySQL draait. Bij zeven toe te voegen soorten verstuurt je applicatie dus zeven netwerkpakketten. Je kan dit optimaliseren met batch updates. Daarbij verstuurt je *meerdere* SQL statements in *één* netwerkpakket. Een beperking is dat dit enkel insert, update of delete statements kunnen zijn, geen select statements.

Je wijzigt de opdracht bij ❹ naar: `statement.addBatch();`

Dit voegt het insert statement toe aan een netwerkpakket, maar verstuurt dit pakket nog niet.

Je voegt voor `connection.commit();` volgende opdracht toe: `statement.executeBatch();`

Dit stuurt het netwerkpakket met de insert statements naar de database.

De database voert de statements uit.

Je kan de applicatie uitvoeren.

Misschien wil je de automatisch gegenereerde id's van de toegevoegde records weten.

Je wijzigt dan de method create als volgt:

```
public List<Long> create(List<String> namen) throws SQLException {           ❶
    String sql = "insert into soorten(naam) values (?)";
    try (Connection connection = super.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql,
            PreparedStatement.RETURN_GENERATED_KEYS)) {                       ❷
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        for (String naam : namen) {
            statement.setString(1, naam);
            statement.addBatch();
        }
        statement.executeBatch();
        List<Long> gegenereerdeIds = new ArrayList<>();
        try (ResultSet result = statement.getGeneratedKeys()) {               ❸
            while (result.next()) {
                gegenereerdeIds.add(result.getLong(1));                       ❹
            }
        }
        connection.commit();
        return gegenereerdeIds;
    }
}
```

(1) De method zal een verzameling met de gegenereerde id's teruggeven.

(2) Je vermeldt RETURN_GENERATED_KEYS waar je het insert statement specificeert.

(3) De method getGeneratedKeys geeft een ResultSet met de gegenereerde id's.

(4) Je leest een gegenereerde id en je voegt hem toe aan de verzameling gegenereerde id's.

Je wijzigt in de class Main de opdracht `repository.create(namen);` naar:

```
List<Long> gegenereerdeIds = repository.create(namen);
for (long id: gegenereerdeIds) {
    System.out.println(id);
}
```

Je kan de applicatie uitvoeren.



Aantal bieren per brouwer: zie takenbundel

16 Herhalings oefeningen



Bieren van een soort: zie takenbundel



Omzet leegmaken: zie takenbundel



Gezin: zie takenbundel



Erfenis: zie takenbundel

COLOFON

Domeinexpertisemanager	Jean Smits
Moduleverantwoordelijke	
Auteurs	Hans Desmet
Versie	27/1/2020
Codes	Peoplesoftcode: Wettelijk depot:

Omschrijving module-inhoud

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