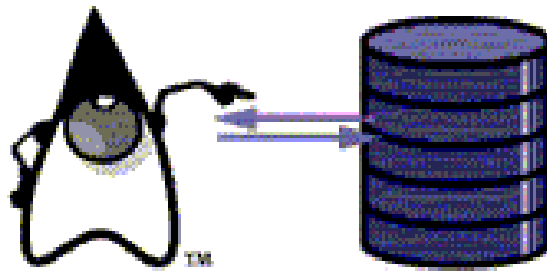




JDBC



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	3
2	Driver	4
2.1	Project	4
3	Connection	5
3.1	Connection kort openhouden	5
3.2	TCP/IP poortnummer	6
3.3	Gebruiker.....	6
3.4	Samenvatting.....	6
4	Repository	7
4.1	Single responsibility.....	7
4.2	AbstractRepository.....	7
5	PreparedStatement	8
5.1	Samenvatting.....	8
6	ResultSet	9
6.1	Kolomvolgnummers	9
6.2	Kolomnamen	10
6.3	Andere methods.....	10
6.4	Berekende kolommen	11
6.5	select *.....	11
6.6	Werkelijkheid	12
6.7	Soorten ResultSets	13
6.8	Samenvatting.....	13
7	SQL statements met parameters.....	14
7.1	Voorbeeld	14
7.2	SQL code injection	15
8	Een record lezen aan de hand van zijn id.....	16
9	Stored procedure	17

9.1	Maken	17
9.2	Testen.....	17
9.3	CallableStatement.....	17
9.4	Voordelen en nadelen	18
10	Transactie.....	19
10.1	Voorbeeld bij een bank.....	19
10.2	Voorbeeld bij een bestelling	19
10.3	Voorbeeld in de database tuincentrum.....	19
10.4	Autocommit mode	19
10.5	Commit en rollback.....	20
10.6	Voorbeeld	20
11	Isolation level	21
11.1	Voorbeeld	22
11.2	ACID.....	24
12	Autonumber kolom	25
12.1	Voorbeeld	25
13	Datums en tijden	26
13.1	Letterlijke datum of tijd in een SQL statement	26
13.2	Datum als parameter	27
13.3	Functies.....	27
14	Lock.....	29
14.1	Voorbeeld	29
14.2	Betere oplossing voor dit voorbeeld	31
15	Database optimaal aanspreken	32
15.1	Enkel de records lezen die je nodig hebt.....	32
15.2	Het SQL keyword in.....	34
15.3	Join	35
15.4	Batch update.....	36
16	Checklist	38
17	Herhalingsoefeningen	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
- Maven

1.3 Nodige software

- JDK (Java Developer Kit) (minstens versie 11)
- MySQL database server (minstens versie 8.0.20)
- MySQL Workbench (minstens versie 8.0.20)
- IntelliJ

1.4 Tijdzone

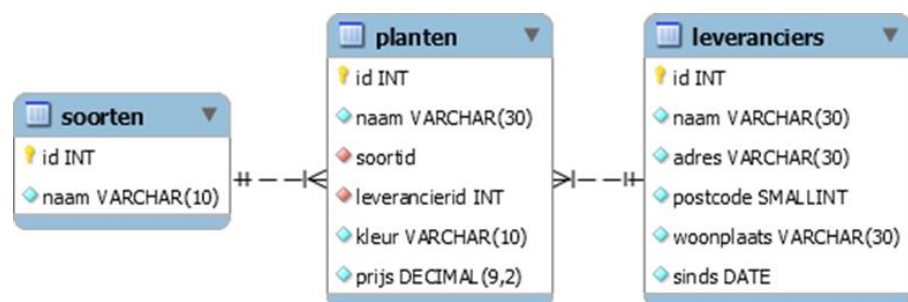
Je wil MySQL aanspreken met JDBC. Je moet dan de tijdzone van je MySQL server instellen:

1. Open in een browser <https://dev.mysql.com/downloads/timezones.html>.
2. Download daar `timezone_2020a_posix_sql.zip` (onder The other set is for 5.7+)
3. Open de folder in dit ZIP bestand en haal uit die folder `timezone_posix.sql`.
4. Start de MySQL Workbench.
5. Log in op de Local instance.
6. Kies in het menu File de opdracht Open SQL Script.
7. Open `timezone_posix.sql`.
8. Voeg vooraan in het script een opdracht toe: `use mysql;`
9. Voer dit script uit met de knop ⚡. Dit kan een tijdje duren.
Je moet met dit script 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. Start het programma NotePad.
11. Kies in het menu File de opdracht Open.
12. Wijzig rechts onderaan Text Documents (*.txt) naar All Files (*.*)
13. Open in de folder `C:\ProgramData\MySQL\MySQL Server 8.0` het bestand `my.ini`
14. Zoek in dit bestand de tekst `[mysqld]`
15. Typ op een regel daar onder
`default-time-zone='Europe/Brussels'`
16. Bewaar het bestand.
17. Herstart je computer.

1.5 Tuincentrum

Je gebruikt in de theorie de database `tuincentrum`.

Maak die in de MySQL Workbench met het script `tuincentrum.sql`.



2 Driver

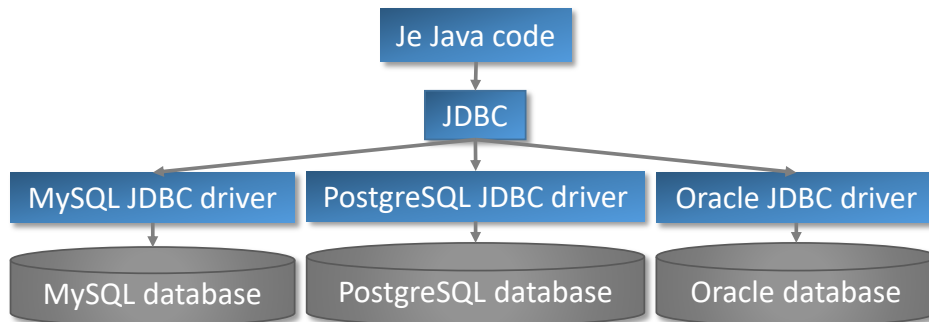
JDBC is een onderdeel van de standaard Java libraries.

Je spreekt met JDBC een relationele database aan:

- MySQL populair, open source.
- PostgreSQL populair, open source.
- Oracle commercieel.
- ...

Je hebt per merk een JDBC driver nodig. Dit is een Java library, verpakt als een JAR bestand.

De classes in de library zijn specifiek voor één database merk:



2.1 Project

Maak een Maven project in IntelliJ.

1. Kies het menu File, New, Project.
2. Kies links Maven.
3. Kies Next.
4. Typ `be.vdab.jdbc` bij `GroupId`.
5. Typ `theorie` bij `ArtifactId`.
6. Kies Next.
7. Kies Finish.

Voeg in `pom.xml` de dependency voor de JDBC driver voor MySQL toe, onder `<version>`.

Je geeft ook aan dat je Java 11 gebruikt en de sources uitgedrukt zijn in UTF-8:

```

<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
    <version>8.0.22</version>
  </dependency>
</dependencies>
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
  
```

1

- (1) Je plaatst de scope op `runtime`. De driver is dan beschikbaar tijdens het uitvoeren van je programma, maar niet tijdens het compileren van je programma.
JDBC gebruikt **intern** de JDBC driver. Jij roept de driver nooit zelf rechtstreeks op.
Je verhindert met de `runtime` scope dat je dit per ongeluk toch zou doen.
Je maakt nu een programma dat met **alle** merken databases kan samenwerken, niet enkel met MySQL.

Klik rechts op het verticale tabblad Maven. Klik daar op .

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.

Maak 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";
    private static final String USER = "root";
    private static final String PASSWORD = "vervangDitDoorHetPaswoordVanRoot";
    public static void main(String[] args) {
        try (var 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 netwerknaam van de computer waarop MySQL draait.
Als dit de computer is waarop je werkt, is de netwerknaam `localhost`.
Hierna komt een `/` en de naam van de te openen database: `tuincentrum`.
Bij een database van het merk PostgreSQL zou de String een beetje anders zijn:
`jdbc:postgresql://localhost/tuincentrum`
- (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.
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.
- (4) De verbinding is mislukt (redenen: typfout in de JDBC URL, MySQL is niet gestart, de database `tuincentrum` bestaat niet, verkeerd paswoord, ...). JDBC werpt dan een `SQLException`.

Voer de applicatie uit.

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 TCP/IP.

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

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

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

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

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

3.3 Gebruiker



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.

Voer 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, insert on soorten to cursist;
```

❶

❷

(1) Je maakt een gebruiker met de naam cursist en het paswoord cursist, als die gebruiker nog niet 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'.

Vervang in de Java code vdab en root door cursist.

Voer de applicatie terug uit.

3.4 Samenvatting



DriverManager

maakt



Connection

naar



database.

4 Repository

4.1 Single responsibility

“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*.

Een synoniem voor repository is DAO (Data Access Object).

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

De werkelijkheid bevat in ons voorbeeld drie gegevenstypes: leveranciers, planten en soorten.

Je hebt dus de classes LeverancierRepository, PlantRepository en SoortRepository.

- LeverancierRepository bevat code die de database table leveranciers aanspreekt.
- PlantRepository bevat code die de database table planten aanspreekt.
- SoortRepository bevat code die de database table soorten aanspreekt.

4.2 AbstractRepository

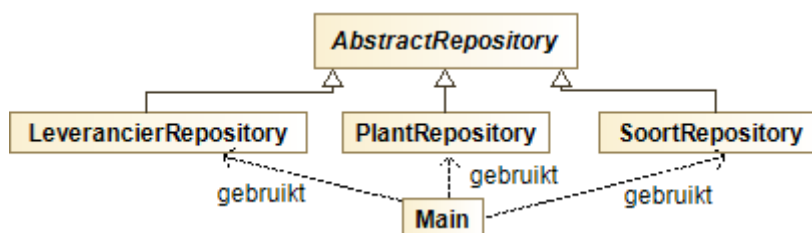
Alle classes gebruiken dezelfde database. Ze hebben dus connecties nodig met dezelfde JDBC URL, dezelfde gebruikersnaam en hetzelfde paswoord. Deze informatie herhalen in elk van de classes is verkeerd. Als de informatie wijzigt, moet je *meerdere* classes wijzigen.

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

Maak 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";
    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 taken

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 preparedStatement 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 %.

Maak de class PlantRepository:

```
package be.vdab.repositories;

public class PlantRepository extends AbstractRepository {
    public int verhoogPrijzenMet10Procent() throws SQLException {
        try (var connection = super.getConnection();           ❶
            var statement = connection.prepareStatement(        ❷
                "update planten set prijs = prijs * 1.1")) {    ❸
            return statement.executeUpdate();                    ❹
        }
    }
}
```

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

(2) De Connection method preparedStatement geeft je een PreparedStatement.

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.

(3) Je geeft als parameter een SQL statement mee.

(4) executeUpdate voert het statement uit en geeft het aantal aangepaste records terug.

Roep de method op in de class Main. Wijzig de code in de method main:

```
var repository = new PlantRepository();
try {
    System.out.print(repository.verhoogPrijzenMet10Procent());
    System.out.println(" planten aangepast.");
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Bekijk met de MySQL Workbench de prijzen van de planten.

Voer de applicatie uit.

Bekijk daarna met de MySQL Workbench de aangepaste prijzen.

5.1 Samenvatting



Connection maakt PreparedStatement.



Bieren verwijderen

6 ResultSet

Je voert met de PreparedStatement method `executeQuery` een select statement uit.
 Je geeft als parameter een String met het select statement mee.
 De method voert het statement uit en geeft daarna een object terug van het type `ResultSet`.

`ResultSet` bevat de rijen die het resultaat zijn van het select statement:

SQL statement:

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

ResultSet:

2	Baumgarten
7	Bloem

Je leest de rijen één per één, door over de rijen te itereren.

Je ziet de stappen in onderstaand overzicht. → geeft aan waar je staat in de `ResultSet`.

Je staat initieel voor de eerste rij:

→

2	Baumgarten
7	Bloem

Je probeert naar de volgende rij te gaan met `resultSet.next()`.

Dit geeft `true` en plaatst je op de eerste rij. Je kan de kolommen van die rij lezen.

→

2	Baumgarten
7	Bloem

Je probeert daarna naar de volgende rij te gaan met `resultSet.next()`.

Dit geeft `true` en plaatst je op de tweede rij. Je kan de kolommen van die rij lezen.

→

2	Baumgarten
7	Bloem

Je probeert daarna naar de volgende rij te gaan met `resultSet.next()`.

Dit geeft `false`. Dit betekent dat je alle rijen gelezen hebt.

De opeenvolging van opdrachten:

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

6.1 Kolomvolgnummers

Maak de class `LeverancierRepository`. Die bevat een method `findAllNamen`.

Die geeft de namen van alle leveranciers.

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

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

❶
❷
❸

- (1) `executeQuery` geeft je een `ResultSet`. De parameter is het select statement.
Je moet een `ResultSet` niet zelf sluiten. Wanneer je het `PreparedStatement` sluit, sluit die de `ResultSet`.
- (2) Je itereert over de rijen in de `ResultSet`.
- (3) Je leest de `String` waarde in de eerste kolom van het select statement.

Roep de method op in de class `Main`. Wijzig de code in de method `main`:

```
var repository = new LeverancierRepository();
try {
    repository.findAllNamen().forEach(System.out::println);
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Voer de applicatie uit.

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 over welke kolom dit gaat.

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

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

Je kan de applicatie uitvoeren.

6.3 Andere methods

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

- `getInt` een getal lezen als een `int`.
- `getByte` een getal lezen als een `byte`.
- `getShort` een getal lezen als een `short`.
- `getLong` een getal lezen als een `long`.
- `getFloat` een getal lezen als een `float`.
- `getDouble` een getal 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`.
- `getObject` een waarde lezen en zelf het type aangeven.
Een date kolom: `getObject("eenDateKolom", LocalDate.class)`
Een time kolom: `getObject("eenTimeKolom", LocalTime.class)`
Een datetime kolom:
`getObject("eenDateTimeKolom", LocalDateTime.class)`

6.4 Berekenende kolommen

Het select statement kan berekenende kolommen bevatten.

Je geeft die kolommen een alias met as.

Je leest die kolom waarden in Java met die alias.

Voorbeeld: maak in LeverancierRepository een method findAantal.

Die geeft de het aantal leveranciers.

```
public int findAantal() throws SQLException {
    try (var connection = super.getConnection();
        var statement = connection.prepareStatement(
            "select count(*) as aantal from leveranciers")) {
        var 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. Zoals bij elke ResultSet sta je initieel *niet op* dit eerste record, maar *voor* dit record. Je gaat met next naar dit eerste record.

(3) Je leest de inhoud van de kolom met de alias aantal.

Roep 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());
```

Voer de applicatie uit.

6.5 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 typen in een String in Java is minder handig als een SQL statement typen in de MySQL Workbench: keywords krijgen geen kleuren, je krijgt geen popup vensters die je helpen de opdracht te typen, ...
Oplossing: Typ het SQL statement eerst in de MySQL Workbench. Test het daar.
Breng het daarna met copy-paste over naar je Java code.

6.6 Werkelijkheid

Je applicatie werkt met Leveranciers, Planten, en Soorten.

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

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

Het woord domain betekent: 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);
    }
}
```

Maak in `LeverancierRepository` een method `findAll`. Die geeft alle leveranciers.

```
public List<Leverancier> findAll() throws SQLException {
    var sql = "select id,naam,adres,postcode,woonplaats,sinds from leveranciers";
    try (var connection = super.getConnection();
        var statement = connection.prepareStatement(sql)) {
        var leveranciers = new ArrayList<Leverancier>();
        var result = statement.executeQuery();
        while (result.next()) {
            leveranciers.add(naarLeverancier(result));
        }
        return leveranciers;
    }
}
private Leverancier naarLeverancier(ResultSet result) throws SQLException {❷
    return new Leverancier(result.getLong("id"), result.getString("naam"),
        result.getString("adres"), result.getInt("postcode"),
        result.getString("woonplaats"), result.getObject("sinds", LocalDate.class));
}
```

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

Roep de method op in de class `Main`. Je wijzigt de code in het `try` blok in de method `main`:
`repository.findAll().forEach(System.out::println);`

Voer de applicatie uit.

6.7 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 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 ResultSets.
- Scrollable ResultSets gebruiken meestal meer RAM dan Forward-Only ResultSets
- Scrollable ResultSets zijn meestal trager dan Forward-Only ResultSets.

- Je kan een updatable ResultSet maken.

Als je een waarde wijzigt in zo'n 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.

Men gebruikt zelden een updatable ResultSet:

- Sommige JDBC drivers ondersteunen geen Updatable ResultSets.
- Je kan van sommige SQL select statements geen updatable ResultSet maken.

6.8 Samenvatting



PreparedStatement maakt ResultSet.



Gemiddelde

7 SQL statements met parameters

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

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

De woonplaats wordt het veranderlijk onderdeel van het SQL statement

De gebruiker typt	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. Stel 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. Geef dit SQL statement mee aan een `PreparedStatement` object.
3. Vul 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. Voer het `PreparedStatement` uit.

7.1 Voorbeeld

Maak in `LeverancierRepository` een method `findByWoonplaats`.

Die geeft de leveranciers uit één woonplaats.

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

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

Roep de method op in de class `Main`. Wijzig de code in de method `main`:

```
System.out.print("Woonplaats:");
var scanner = new Scanner(System.in);
var woonplaats = scanner.nextLine();
var repository = new LeverancierRepository();
try {
    repository.findByWoonplaats(woonplaats).forEach(System.out::println);
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Voer de applicatie uit.



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.

Maak een nieuwe method in PlantRepository:

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

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

Roep de method op in de class Main. Wijzig de code in de method main:

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

Voer het programma uit als een gewone gebruiker: Typ de naam Linde.

Er is geen probleem: je programma wijzigt één plant.

Voer het programma uit als hacker: typ ' or ''='

Er is een groot probleem: je programma wijzigt *alle* planten!

Dit komt omdat je nu volgend SQL statement uitvoert

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

(1) Het where deel geeft bij alle records true terug, omdat de voorwaarde na or altijd true is.

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

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

Wijzig in PlantRepository de code in de method verhoogPrijzenMet10ProcentByNaam

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

Als een hacker nu 'or '=' typt, 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

8 Een record lezen aan de hand van zijn id

LeverancierRepository bevat nu volgende method die data *lezen*:

findAllNamen, findAantal, findAll en findByWoonplaats.

- 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 findByWoonplaats 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.

Voeg een method toe aan LeverancierRepository om dit te proberen:

```
public Optional<Leverancier> findById(long id) throws SQLException { ❶
    var sql =
        "select id,naam,adres,postcode,woonplaats,sinds from leveranciers where id = ?";
    try (var connection = super.getConnection());
        var statement = connection.prepareStatement(sql)) {
        statement.setLong(1, id);
        var result = statement.executeQuery();
        return result.next() ? Optional.of(naarLeverancier(result)) ❷
            : Optional.empty(); ❸
    }
}
```

- (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 naar de rij van de ResultSet te gaan.
Als dit lukt maak je een Leverancier object op basis van de rij.
Je vult met dit object een Optional object.
- (3) De ResultSet is leeg. Je geeft een "lege" Optional terug.

Roep de method op in de class Main. Wijzig de code in de method main:

```
System.out.print("id:");
var scanner = new Scanner(System.in);
var id = scanner.nextLong();
var repository = new LeverancierRepository();
try {
    repository.findById(id)
        .ifPresentOrElse(System.out::println,
            () -> System.out.println("Niet gevonden"));
}
catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Voer de applicatie uit.



Id

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 Maken

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

1. Dubbelklik onder SCHEMAS op de database `tuincentrum`.
2. Klik met de rechtermuisknop op `Stored Procedures`. Kies `Create Stored Procedure`.
3. Wijzig de code:

```
create procedure tuincentrum.PlantNamenMetEenWoord (woord varchar(50)) ❶
begin ❷
select naam from planten where naam like woord order by naam; ❸
end ❹
```

- (1) Je maakt een stored procedure met de sleutelwoorden `create procedure`.
Je typt daarna de naam van de database waarin je de stored procedure aanmaakt.
Je typt daarna een punt en de naam van de stored procedure.
Je typt 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 typt na elk SQL statement een ;
- (4) Je eindigt de stored procedure met `end`.

Maak de stored procedure met de knoppen `Apply`, `Apply` en `Finish`.

9.2 Testen

Geef de gebruiker `cursist` het recht om de stored procedure uit te voeren:

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

Voer de stored procedure uit 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.

Voeg een method toe aan `PlantRepository`:

```
public List<String> findNamenByWoord(String woord) throws SQLException {
    try (var connection = super.getConnection();
        var statement = connection.prepareCall("{call PlantNamenMetEenWoord(?)}") { ❶
        statement.setString(1, '%' + woord + '%'); ❷
        var namen = new ArrayList<String>();
        var result = statement.executeQuery(); ❸
        while (result.next()) {
            namen.add(result.getString("naam"));
        }
        return namen;
    }
}
```

- (1) Je geeft aan de Connection method `prepareCall` een String mee. Je typt deze als volgt:
 - a. Je typt `{call` voor de naam van de stored procedure.
 - b. Je stelt de stored procedure parameter(s) voor met `?`
 - c. Je sluit af met `}`.`prepareCall` geeft je een `CallableStatement` object.
- (2) Je vult de eerste stored procedure parameter in.
 Je concateneert `%` voor en na het woord dat de gebruiker typte.
 Als de gebruiker `bloem` typte wordt het like onderdeel van het select statement in de stored procedure dus `'%bloem%'`.
- (3) Je voert de stored procedure uit. Je krijgt een `ResultSet` terug.

Roep de method op in de class Main. Wijzig de code in de method main:

```
System.out.print("Woord:");
var scanner = new Scanner(System.in);
var woord = scanner.nextLine();
var repository = new PlantRepository();
try {
    repository.findNamenByWoord(woord).forEach(System.out::println);
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Voer de applicatie uit.

9.4 Voordelen en nadelen

Voordelen van een stored procedure:

- + Je schrijft een SQL statement gemakkelijker over meerdere regels dan in Java.
- + Als je syntaxfouten typt, 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

10 Transactie

Een database transactie zorgt er voor dat SQL statements

- ofwel **allemaal** uitgevoerd worden
- ofwel **allemaal ongedaan** gemaakt worden

10.1 Voorbeeld bij een bank

Je schrijft 10 € over van rekening 1 naar rekening 2.

Het saldo van rekening 1 moet daarbij verlagen met 10 €.

Het saldo van rekening 2 moet daarbij verhogen met 10 €.

Je stuurt daartoe twee statements naar de database:

```
update rekeningen set saldo = saldo - 10 where id = 1
```

```
update rekeningen set saldo = saldo + 10 where id = 2
```

Als de database enkel het 1° statement uitvoert (omdat de database of de applicatie uitvalt na het uitvoeren van het 1° statement, maar voor het uitvoeren van het 2° statement) is de overschrijving verkeerd: het geld is verdwenen van rekening 1, maar niet aangekomen bij rekening 2.

De database moet ofwel beide statements uitvoeren, of (bij problemen) de wijzigingen van beide statements ongedaan maken. Dit gebeurt als je de statements verzamelt in een transactie.

10.2 Voorbeeld bij een bestelling

Je verwijdert bestelling 6. Je moet het record verwijderen in de table bestellingen.

Je moet ook de detaillijnen van bestelling 6 verwijderen in de table bestelDetailLijnen

```
delete from bestellingen where id = 6
```

```
delete from bestelDetailLijnen where id = 6
```

Als de database enkel het eerste statement uitvoert,

bevat de table bestelDetailLijnen records van een bestelling die niet meer bestaat.

Dit geeft verkeerde resultaten als je statistieken baseert op die table.

De database moet ofwel beide statements uitvoeren, of (bij problemen) de wijzigingen van beide statements ongedaan maken. Dit gebeurt als je de statements verzamelt in een transactie.

10.3 Voorbeeld in de database tuincentrum

- 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
```

Als enkel het 1° statement wordt uitgevoerd

heeft de gebruiker een foutieve situatie 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 door de statements te verzamelen in een transactie.

10.4 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.

10.5 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`.

10.6 Voorbeeld

Dit is het uitgewerkte voorbeeld zoals eerder beschreven.

Voeg een method toe aan `PlantRepository`:

```
public void verhoogPrijzenBovenEnOnder100€() throws SQLException {
    var sqlVanaf100 = "update planten set prijs = prijs*1.1 where prijs >= 100";
    var sqlTot100 = "update planten set prijs = prijs*1.05 where prijs < 100";
    try (var connection = super.getConnection();
        var statementVanaf100 = connection.prepareStatement(sqlVanaf100);
        var 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.

Roep de method op in de class `Main`. Je wijzigt de code in het `try` blok in de method `main`:
`repository.verhoogPrijzenBovenEnOnder100€();`

Bekijk met de MySQL Workbench de prijzen van de planten.

Voer de applicatie uit.

Bekijk 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 `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.



Failliet

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. 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 typt 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:

Maak een package be.vdab.exceptions. Maak daarin de class SoortBestaatAlException:

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

Maak in de package be.vdab.repositories. de class SoortRepository:

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

(1) Je plaatst het isolation level op Serializable.

(2) Je zoekt of een record met de getypte 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.

Roep de method op in de class Main. Wijzig de code in de method main:

```
System.out.print("Naam:");
var scanner = new Scanner(System.in);
var naam = scanner.nextLine();
var repository = new SoortRepository();
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.");
}
```

Voer het programma uit.

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

1. Plaats een breakpoint op de regel `if (result.next()) {`
Klik daartoe 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. Kies in het menu Run de opdracht Debug 'Main'.
3. Typ in het tabblad Console (beneden in IntelliJ) de soortnaam test. Druk Enter.
4. De uitvoering van je programma pauzeert op het breakpoint.
De regel van het breakpoint heeft dan een blauwe achtergrond.
5. Laat IntelliJ open staan. Schakel over naar de MySQL WorkBench.
6. Voer daar volgend statement uit: `insert into soorten(naam) values ('test')`
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.
7. Schakel terug naar IntelliJ.
8. Laat je programma verder lopen met de knop  links onder.
Je programma voegt het record toe, commit de transactie en komt ten einde.
9. Schakel 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.

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

Oplossing 2: het isolation level Read Committed.

Wijzig in SoortRepository de code in de method create:

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

- (1) Je probeert een record met de getypte 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 getypte soortnaam.

Voer het programma uit.

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.

11.2 ACID

Men zegt dat een transactie ACID eigenschappen heeft. ACID is een afkorting voor:

- **Atomic**
De database voert de transactie volledig uit (commit) of doet die volledig teniet (rollback).
- **Consistent**
Zowel na een commit als na een rollback is de database correct. Voorbeeld: als een kolom verplicht in te vullen is, is dit zo na een rollback of een commit ,bij alle records.
- **Isolated**
Transacties die de database tegelijk uitvoert hebben geen zicht in elkaars tussenresultaten. Je regelt dit met het isolation level.
- **Durable**
Een voltooide transactie blijft voltooid (zelfs na een systeem crash).



Isolation

12 Autonummer kolom

Als je een record toevoegt, geef je geen waarde mee voor een autonummer 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 autonummer kolom in het toegevoegde record.

12.1 Voorbeeld

Wijzig in `SoortRepository` de method `create`. Die geeft het nummer van de nieuwe soort.

```
public long create(String naam) throws SQLException {
    try (var connection = super.getConnection();
        var statementInsert = connection.prepareStatement(
            "insert into soorten(naam) values (?)",
            PreparedStatement.RETURN_GENERATED_KEYS)) { ❶
        statementInsert.setString(1, naam);
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        try {
            statementInsert.executeUpdate();
            var result = statementInsert.getGeneratedKeys(); ❷
            result.next(); ❸
            var 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) `getGeneratedKeys` geeft een `ResultSet` met de autonummer 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 autonummer waarde.

Roep de method op in de class `Main`. Wijzig de code in de method `main`:

```
System.out.print("Naam:");
var scanner = new Scanner(System.in);
var naam = scanner.nextLine();
var repository = new SoortRepository();
try {
    var 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.");
}
```

Voer de applicatie uit.

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 typt.
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.

Voeg een method toe aan LeverancierRepository:

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

Roept de method op in de class Main. Wijzig de code in de method main:

```
var repository = new LeverancierRepository();
try {
    repository.findBySinds2000().forEach(System.out::println);
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Voer de applicatie uit.

13.2 Datum als parameter

Als de gebruiker een datum typt, 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 getypte datum in dienst kwamen.

Voeg een method toe aan LeverancierRepository:

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

(1) Er bestaat geen setLocalDate method. Je gebruikt dan de setObject method.

Roep de method op in de class Main. Wijzig de code in de method main:

```
System.out.print("Datum vanaf (dag/maand/jaar):");
var formatter = DateTimeFormatter.ofPattern("d/M/y");
var scanner = new Scanner(System.in);
var datum = LocalDate.parse(scanner.nextLine(), formatter);
var repository = new LeverancierRepository();
try {
    repository.findBySindsVanaf(datum).forEach(System.out::println);
} catch (SQLException ex) {
    ex.printStackTrace(System.err);
}
```

Voer de applicatie uit.

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.

Voeg een method toe aan LeverancierRepository:

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

Roep de method op in de class Main. Wijzig de code in de method main:

```
var repository = new LeverancierRepository();  
try {  
    repository.findLeverancierGewordenInHetJaar2000().forEach(System.out::println);  
} catch (SQLException ex) {  
    ex.printStackTrace(System.err);  
}
```

Voer de applicatie uit.



Bieren van een maand

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.

14.1 Voorbeeld

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

Maak een class PlantNietGevondenException:

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

Maak een class PrijsTeLaagException:

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

Voeg een method toe aan PlantRepository:

```
public void verlaagPrijs(long id, BigDecimal nieuwePrijs) throws SQLException {
    var sqlSelect = "select prijs from planten where id = ? for update";
    try (var connection = super.getConnection();
        var statementSelect = connection.prepareStatement(sqlSelect)) {
        statementSelect.setLong(1, id);
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        var result = statementSelect.executeQuery();
        if (result.next()) {
            var oudePrijs = result.getBigDecimal("prijs");
            var minimumNieuwePrijs = oudePrijs.divide(BigDecimal.valueOf(2),
                2, RoundingMode.HALF_UP);
            if (nieuwePrijs.compareTo(minimumNieuwePrijs) >= 0) {
                var sqlUpdate = "update planten set prijs = ? where id = ?";
                try (var statementUpdate = connection.prepareStatement(sqlUpdate)) {
                    statementUpdate.setBigDecimal(1, nieuwePrijs);
                    statementUpdate.setLong(2, id);
                    statementUpdate.executeUpdate();
                    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.

Roep de method op in de class Main. Wijzig de code in de method main:

```
var scanner = new Scanner(System.in);
System.out.print("Nummer plant:");
var id = scanner.nextLong();
System.out.print("Nieuwe prijs:");
var nieuwePrijs = scanner.nextBigDecimal();
var repository = new PlantRepository();
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);
}
```

Voer het het programma uit

14.2 Betere oplossing voor dit voorbeeld

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. Vraag het nummer van de plant.
2. Vraag 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. Controleer 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.

Wijzig op die manier de method `verlaagPrijs` in `PlantRepository`:

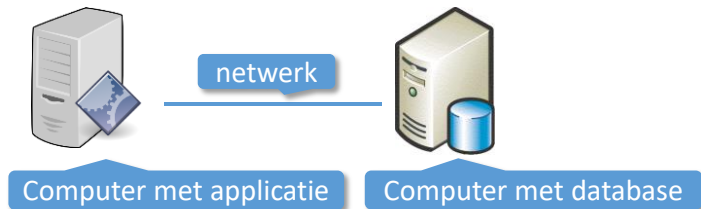
```
public void verlaagPrijs(long id, BigDecimal nieuwePrijs) throws SQLException {
    var sqlUpdate = "update planten set prijs = ? where id = ? and ? > prijs/2";
    try (var connection = super.getConnection();
        var statementUpdate = connection.prepareStatement(sqlUpdate)) {
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        statementUpdate.setBigDecimal(1, nieuwePrijs);
        statementUpdate.setLong(2, id);
        statementUpdate.setBigDecimal(3, nieuwePrijs);
        var aantalAangepast = statementUpdate.executeUpdate();
        if (aantalAangepast == 1) {
            connection.commit();
            return;
        }
        var sqlSelect = "select count(*) as aantal from planten where id = ?";
        try (var statementSelect = connection.prepareStatement(sqlSelect)){
            statementSelect.setLong(1, id);
            var result = statementSelect.executeQuery();
            result.next();
            if (result.getLong("aantal") == 0) {
                connection.rollback();
                throw new PlantNietGevondenException();
            }
            connection.rollback();
            throw new PrijsTeLaagException();
        }
    }
}
```

Voer het programma uit.

15 Database optimaal aanspreken

In productie is de computer met je applicatie meestal een andere dan die met de database. De applicatie communiceert dan over het netwerk 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, op verschillende computers, kunnen dezelfde database delen.



Het vertraagt echter de database toegang. Telkens je applicatie de database aanspreekt, gebruik je het netwerk. Dit is ongeveer **1.000** keer trager dan RAM geheugen.

De database zelf gebruikt de harddisk. Een harddisk is ongeveer **100.000** keer trager is dan RAM.

Het aanspreken van de database is dus het onderdeel in je Java code dat je best optimaliseert.

Je krijgt op de computer waarop je de applicatie ontwikkelt een verkeerd (te positief beeld) van de snelheid waarmee je de database aanspreekt:

- Je programma en de database bevinden zich op dezelfde computer. Je programma spreekt de database dus aan zonder de vertraging van het intranet netwerk. De vertraging is er wel in productie. Dan bevindt de database zich op een andere computer.
- Je bent de enige gebruiker van de database. In productie zijn er veel database gebruikers, waardoor de database trager werkt.
- Je tables bevatten maximaal enkele tientallen of honderden records. In productie bevatten sommige tables duizenden of miljoenen records. Dit maakt sommige queries traag.

Laat je dus niet misleiden door de snelheid waarmee je de database aanspreekt op je ontwikkel PC !

Je leert in dit hoofdstuk technieken waarmee je de database performant aanspreekt.

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 {
    var sql = "select id,naam,adres,postcode,woonplaats,sinds from leveranciers";
    try (var connection = super.getConnection();
        var statement = connection.prepareStatement(sql)) {
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        var leveranciers = new ArrayList<Leverancier>();
        var result = statement.executeQuery();
    }
}
  
```

```

        while (result.next()) {
            if ("Wevelgem".equals(result.getString("woonplaats"))) {
                leveranciers.add(naarLeverancier(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.

Typ `select id, naam, woonplaats from leveranciers`. Kies .

Je ziet een overzicht met de stappen die MySQL intern doet bij het uitvoeren van deze query.

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 `LeverancierRepository`.

Je kan de oplossing nog wat optimaliseren met transactiebeheer:

```

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

`select id, naam, woonplaats from leveranciers where woonplaats='wevelgem'`. Kies .

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 typt enkele plantnummers. Jij toont de namen van die planten.

Voeg een method toe aan PlantRepository:

```
public List<String> findNamenByIds(Set<Long> ids) throws SQLException {
    if (ids.isEmpty()) {
        return List.of();
    }
    var sql = "select naam from planten where id in (" +
        "?,".repeat(ids.size()- 1) +
        "?)";
    try (var connection = super.getConnection();
        var statement = connection.prepareStatement(sql)) {
        int index = 1;
        for (var id : ids) {
            statement.setLong(index++, id);
        }
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        var namen = new ArrayList<String>();
        var result = statement.executeQuery();
        while (result.next()) {
            namen.add(result.getString("naam"));
        }
        connection.commit();
        return namen;
    }
}
```

❶

❷

❸

❹

- (1) Als de lijst met te zoeken id's leeg is geef je een onmiddellijk een lege lijst met namen terug.
- (2) Je maakt het begin van het select statement.
- (3) Je voegt een ? en een , toe per te zoeken id, behalve voor de laatste id.
- (4) Je voegt een ? voor de laatste id en je sluit het statement af met een).

Roep de method op in de class Main. Wijzig de code in de method main:

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

Voer het programma uit.

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

Zonder optimalisatie:

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

Met optimalisatie: `delete from planten where id in (3, 7)`

De optimalisatie is ook bruikbaar als de gebruiker de prijs van planten 3 en 7 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`

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

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 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^o plant heeft de leverancierid 6, de 2^o 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 = 8`
...

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`.

Maak 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.

Maak in die package de class `PlantNaamEnLeveranciersNaam`:

```
package be.vdab.dto;
public class PlantNaamEnLeveranciersNaam {
    private final String plantNaam;
    private final String leveranciersNaam;
    // Maak een geparametriseerde constructor en een toString die
    // plantNaam, gevolgd door -, gevolgd door LeverancierNaam teruggeeft
}
```

Voeg een method toe aan `PlantRepository`:

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

```

        while (result.next()) {
            list.add(new PlantNaamEnLeveranciersNaam(result.getString("plantnaam"),
                result.getString("leveranciersnaam")));
        }
        connection.commit();
        return list;
    }
}

```

Roep de method op in de class Main. Wijzig de code in de method main:

```

var repository = new PlantRepository();
try {
    repository.findRodePlantenEnHunLeveranciers().forEach(System.out::println);
}
catch (SQLException ex) {
    ex.printStackTrace(System.err);
}

```

Voer de applicatie uit.

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 SoortRepository 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 {
    try (var connection = super.getConnection();
        var statement = connection.prepareStatement(
            "insert into soorten(naam) values (?)")) {
        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.

Roep de method op in de class Main. Wijzig de code in de method main:

```

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

```

Voer de applicatie uit.

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 wijzig 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.

Voer de applicatie uit.

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

Wijzig de method create als volgt:

```
public List<Long> create(List<String> namen) throws SQLException {           ❶
    try (var connection = super.getConnection();
        var statement = connection.prepareStatement(
            "insert into soorten(naam) values (?)",
            PreparedStatement.RETURN_GENERATED_KEYS)) {                       ❷
        connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        connection.setAutoCommit(false);
        for (String naam : namen) {
            statement.setString(1, naam);
            statement.addBatch();
        }
        var gegenereerdeIds = new ArrayList<Long>();
        statement.executeBatch();
        var 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 geven.

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

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

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

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

```
var gegenereerdeIds = repository.create(namen);
gegenereerdeIds.forEach(System.out::println);
```

Voer de applicatie uit.



Aantal bieren per brouwer

16 Checklist

Wanneer je programma klaar is, gebruik je onderstaande checklist.

Je controleert zo of je JDBC op de beste manier gebruikt.

Ik maak alle Connections en PreparedStatements in try(...) .	✓
Ik gebruik altijd transacties.	✓
Ik gebruik in elke transactie het beste isolation level.	✓
De gebruiker typt geen gegevens <i>terwijl</i> een Connection open staat.	✓
Ik gebruik geen * in select statements (select * from ...).	✓
Ik geef veranderlijke gegevens in SQL statements aan met ?.	✓
Ik lees enkel alle records als ik echt, echt, echt alle records nodig heb.	✓
Als ik een record lees en daarna ook wijzig, lock ik het record bij het lezen, zodat andere gebruikers dit record niet op hetzelfde moment kunnen wijzigen of verwijderen (select ... for update)	✓
Als ik een record toevoeg, houd ik er rekening mee dat een andere gebruiker een record met dezelfde informatie toevoegt. Ik krijg een exception als ik dan een record toevoeg met een primary key waarde die al bestaat. Ik vang die exception op en toon een foutmelding.	✓
Als ik in een transactie records toevoeg, wijzig of verwijder, doe ik slechts een commit als ik alle gewenste records kon toevoegen, wijzigen of verwijderen.	✓

17 Herhalingsoefeningen



Bieren van een soort, Omzet leegmaken, Gezin, Erfenis

COLOFON

Domeinexpertisemanager	Jean Smits
Moduleverantwoordelijke	
Auteurs	Hans Desmet
Versie	6/1/2021
Codes	Peoplesoftcode: Wettelijk depot:

Omschrijving module-inhoud

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