

# 8 Persistentie

## Serialization, JDBC en DAO

Programmeren 2 – Java

2017 - 2018

**KdG** Karel de Grote  
Hogeschool

Kris Behiels

Jan De Rijke

Mark Goovaerts

# Programmeren 2 - Java

---

1. Herhaling en Collections
2. Generics en documenteren
3. Annotations en Reflection
4. Testen en logging
5. Design patterns (deel 1)
6. Design patterns (deel 2)
7. Lambda's en streams
- 8. Persistentie (JDBC)**
9. XML en JSON
10. Threads
11. Synchronization
12. Concurrency



# Persistentie

---



- Probleem:
  - Objecten verdwijnen na afsluiten programma
- Oplossingen:
  - Data wegschrijven naar een tekst- of een binaire **stream** (zie Java 1)
  - Data in een **database** opslaan (JDBC)
  - Objecten wegschrijven = **serialization**
  - Objecten parsen naar een dataformaat zoals **XML** of **JSON** (zie volgende week)

# Agenda

---

## 1. Serialization

- Serialization / deserialization
- Voorbeeld
- Pro en contra



## 2. JDBC

- JDBC driver
- Datatypes in SQL versus Java
- JDBC met Java:
  - Connection, Statement, ResultSet
  - PreparedStatement

## 3. DAO

- DAO pattern
- Object relational mapping

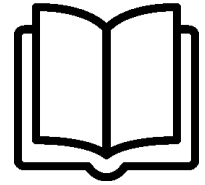


---

# Serialization

## Ondersteunend materiaal

---



- Java How to Program Tutorial  
Introduction to Object Serialization  
(Deitel):

[http://www.deitel.com/articles/java\\_tutorials/20050923/IntroductionToObjectSerialization.html](http://www.deitel.com/articles/java_tutorials/20050923/IntroductionToObjectSerialization.html)

# Object serialization

---



- Object **serialization** is een mechanisme waarbij een objectstructuur in het geheugen geconverteerd wordt naar een sequentie van bytes met daarin:
  - de data van het object
  - informatie over interne structuur en datatypes
  - bevat de klasse (.class bestand) zelf niet
- Object **deserialization**: het terug inlezen en aanmaken van de objectstructuur mét data in het geheugen.

# Serialization: welke tools?

## Lege interface!

The serialization interface has no methods or fields and serves only to identify the semantics of being **serializable**. Implementing the interface **marks** the class as "*okay to serialize*".

- Interface **Serializable**

- **Serialization:**

de klasse **ObjectOutputStream**

```
-public final void writeObject(Object object)  
    throws IOException
```

- **Deserialization:**

de klasse **ObjectInputStream**

```
-public final Object readObject()  
    throws IOException, ClassNotFoundException
```



# Serialization

---

- Om een object met `writeObject` te kunnen wegschrijven moet er aan het volgende voldaan zijn:
  - ✓ de klasse moet **public** zijn;
  - ✓ de klasse moet de **Serializable** interface implementeren;
  - ✓ Recursief: als een attribuut tot een klasse behoort moet die klasse ook **Serializable** zijn
- Attributen die **niet** moeten opgeslagen worden laat je voorafgaan door het keyword **transient**

# Serialization: voorbeeld (1)

```
public class Persoon implements Serializable {
```

```
    private String naam;
```

```
    private String adres;
```

```
    private long rijksRegisterNummer;
```

```
    private transient double loon;
```

Anders:  
**NotSerializableException**

**transient**, want we willen  
loon **niet** serialiseren

```
    public Persoon(String naam, String adres,  
                    long rijksRegisterNummer, double loon) {
```

```
        this.naam = naam;
```

```
        this.adres = adres;
```

```
        this.rijksRegisterNummer = rijksRegisterNummer;
```

```
        this.loon = loon;
```

```
    }
```

```
// getters, setters, ...
```

## Serialization: voorbeeld (2)

```
import java.io.*;
public class SerializePersoon {
    public static void main(String[] args) {
        Persoon persoon = new Persoon("Joske Vermeulen",
            "Kuiperskaai 16, 9000 Gent", 93051822361L, 3521.87);
        try(FileOutputStream fileOut
            = new FileOutputStream("persoon.ser");
            ObjectOutputStream out
            = new ObjectOutputStream(fileOut)) {
            out.writeObject(persoon);
            //out.close();
            //fileOut.close();
            System.out.println("opgeslagen in 'persoon.ser'");
        } catch (IOException ex) {
            //foutafhandeling...
        }
    }
}
```

← Uit de package `java.io`

← try with resources!

← Niet nodig door try with resources!

↑ Java conventie: we gebruiken de extensie `.ser`

# Serialization: voorbeeld (3)

```
public class DeserializePersoon {  
    public static void main(String[] args) {  
        try (FileInputStream fileIn=new FileInputStream("persoon.ser");  
            ObjectInputStream in = new ObjectInputStream(fileIn);) {  
            Persoon persoon = (Persoon) in.readObject();  
            System.out.println("Deserialize Persoon...");  
            System.out.println("Naam: " + persoon.getNaam());  
            System.out.println("Adres: " + persoon.getAdres());  
            System.out.println("RRN:" +  
                persoon.getRijksRegisterNummer());  
            System.out.println("Loon: " + persoon.getLoon());  
  
            //in.close(); //Niet nodig door try with resources  
            //fileIn.close(); //Niet nodig door try with resources  
        } catch (IOException | ClassNotFoundException ex) {  
            //foutafhandeling...  
        }  
    }  
}
```

Loon: 0,0 (want transient)

Deserialize Persoon...  
Naam: Joske Vermeulen  
Adres: Kuiperskaai 16, 9000 Gent  
RRN: 93051822361  
Loon: 0.0

# Serialization: pro en contra

---



- Pro:

- **Platform onafhankelijk** door JVM; dus een object kan geserialiseerd worden op het ene platform en gedeserialiseerd op een compleet verschillend platform



- Contra:

- Taalafhankelijk: lezen en schrijven in java
- De **structuur** van de klasse mag achteraf niet meer wijzigen!
- Alle gebruikte objecten moeten Serializable zijn
- Altijd volledige object terug inlezen  
→ performantie!

*Voorbeeld: StudentenMap bevat een TreeMap met honderden Student objecten. Om één daarvan te benaderen moet het hele StudentenMap object ingeladen worden*

# Opdrachten

---

- Groeiproject

- module 7  
(deel 1 en 2: "Serialization")



- Opdrachten op BB

- Serialization opgave



# Agenda

---

## 1. Serialization

- Serialization / deserialization
- Voorbeeld
- Pro en contra

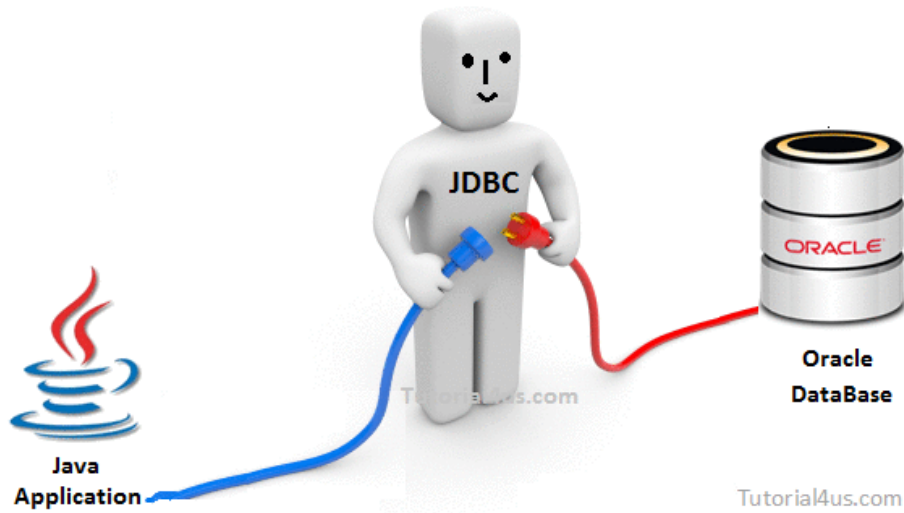


## 2. JDBC

- JDBC driver
- Datatypes in SQL versus Java
- JDBC met Java:
  - Connection, Statement, ResultSet
  - PreparedStatement

## 3. DAO

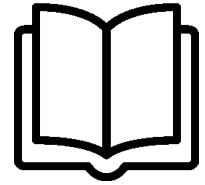
- DAO pattern
- Object relational mapping



---

# JDBC





- Syllabus E-book: "Accessing Databases with JDBC" p.167 ev (Java How to Program, Tenth Edition)
- luv2code video tutorial
  - bekijk les 1-5
  - Gebruikt MySQL en Eclipse

# JDBC

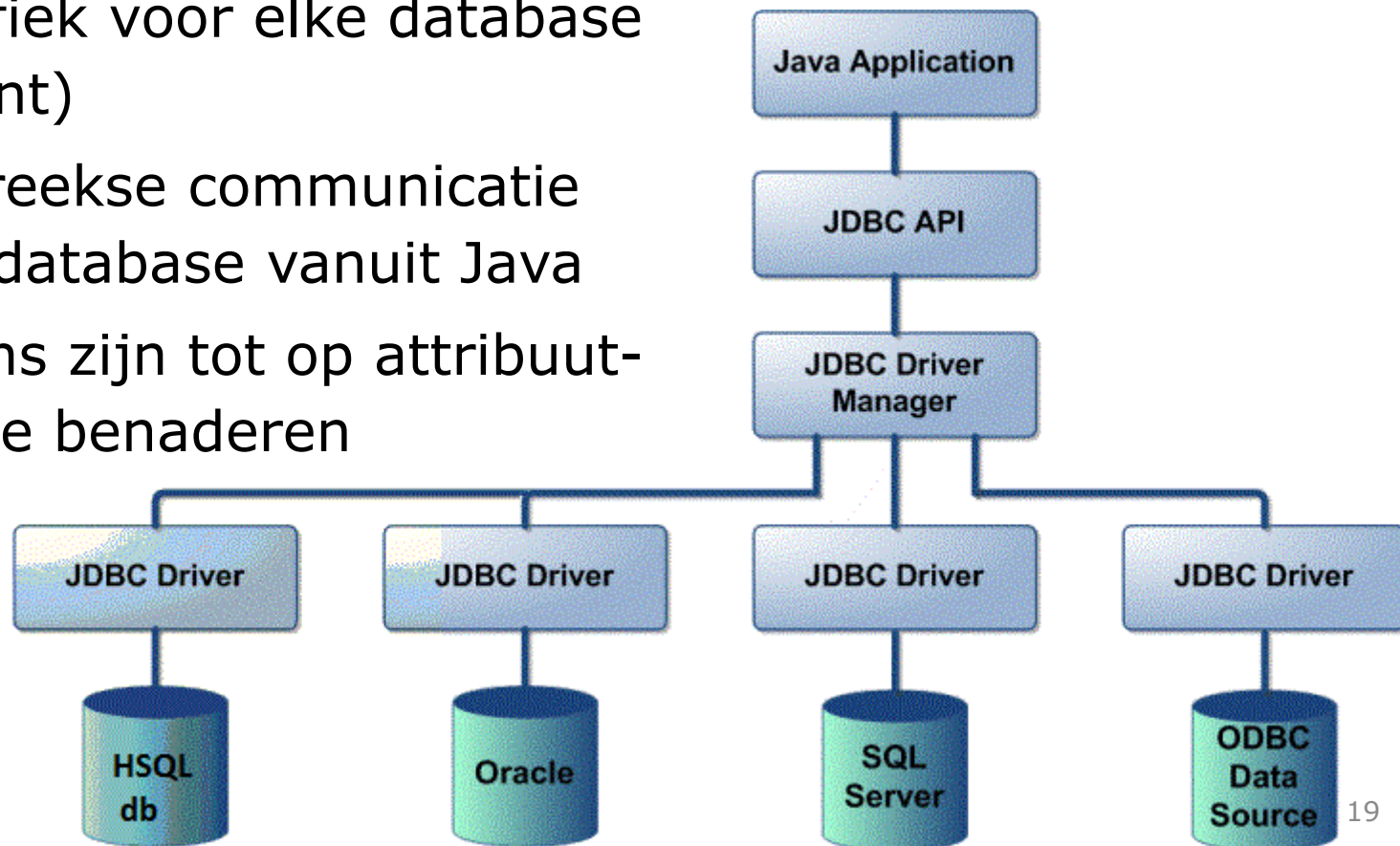
---

- JDBC is vergelijkbaar met Microsoft ODBC
- API voor communicatie tussen Java-programma <-> relationele database
- Onafhankelijk van de database via **Driver**
- Toegang via SQL queries
- Volledig in Java geïntegreerd
  - package: `java.sql.*`
  - JDK < 9 bevat relationele databank: JavaDB (= Apache Derby)

# JDBC driver

- JDBC driver:

- Verzorgt de koppeling met de database
- Is specifiek voor elke database (fabrikant)
- Rechtstreekse communicatie met de database vanuit Java
- Gegevens zijn tot op attribuut-niveau te benaderen



# HSQldb

---

- Waarom **HSQldb**?
  - Lightweight 100% Java SQL Database Engine
  - klein en erg snel
  - goede integratie met java
  - eenvoudig te gebruiken (geen users, server, ...)
  - database van OpenOffice
  - ondersteunt standaard SQL syntax
- **hsqldb\_2.4.0.jar** downloaden en instellen via *File > Project Structure*
  - Staat ook op BB
- zie <http://hsqldb.org/>

# Datatypes in SQL vs Java

SQL	JDBC/Java
VARCHAR	java.lang.String
CHAR	java.lang.String
LONGVARCHAR	java.lang.String
BIT	boolean
NUMERIC	java.math.BigDecimal
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	float
DOUBLE	double
DATE	java.sql.Date
TIME	java.sql.Time

OPGELET:  
converteren naar  
LocalDate / LocalTime  
(zie verder in slides)

## Aanpassingen basisklasse:

---

- Voorzie een extra primary key-attribuut van het type int:

```
public class Student {  
    private int id; ← Primary Key  
    private String naam;  
    private double score;  
    // ...  
}
```

- Hou er rekening mee dat veel databanken autonummering voorzien en dus zélf de PK-waarde kiezen (beginnend vanaf 0)

# Architectuur JDBC

---

Eenvoudige structuur, Klassen uit packages `java.sql` en `javax.sql`

Belangrijkste klassen:

- Connection**

- via DriverManager

- Statement**

- uitvoering van SQL-query op server

- ResultSet**

- soort collection als resultaat van een SELECT-query

# Databank aanspreken in Java

---

## Stappen:

### Registreer JDBC-driver

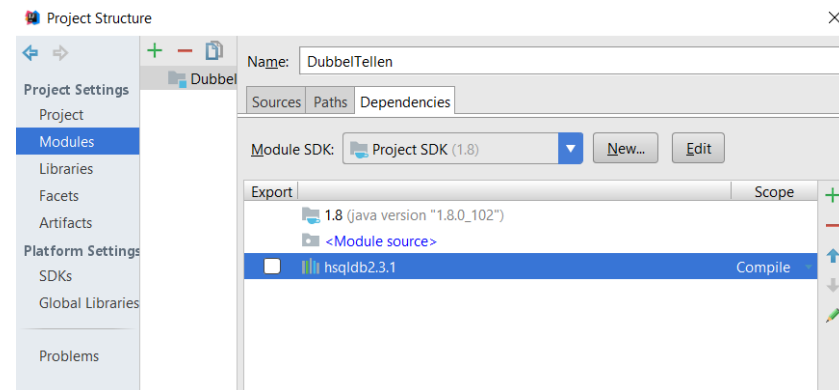
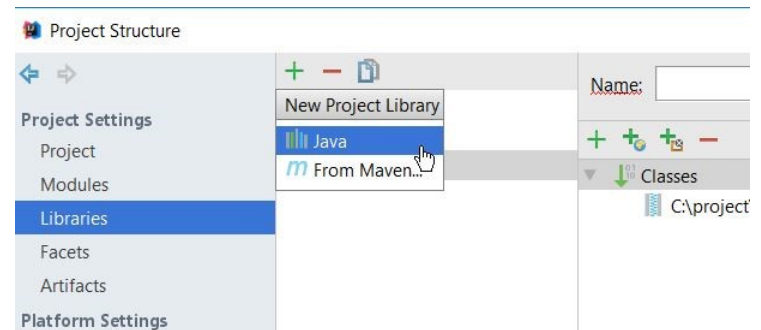
- Zorg dat de driver als library in je project toegevoegd is
- In java 6 moest je de jdbc driver klasse laden via `Class.forName` vb: `Class.forName ("com.mysql.jdbc.driver") ;`

1. Maak een verbinding met de databank (**Connection**)
2. Maak een **Statement**
3. **execute** SQL
4. Verwerk de opgehaalde data (**ResultSet**)
5. Verbreek de verbinding met de databank



# Stap 0: Voeg de driver bibliotheek toe

- Installeer de hsqldb jar in de lib map van je project
- File>Project Structure (`<ALT><CTRL><SHIFT>S`)
  - Project Settings > Libraries
    - Voeg lib/hsqldb\_*versienr*.jar toe
- Je kan de toegevoegde bibliotheek zien via
  - Project Settings > Modules
    - Dependencies



# Stap 1: Maak een verbinding

- Vraag een **Connection**-object op:  
gebruik **URL**, (**username** en **password**)

```
try {  
    Connection connection =  
        DriverManager.getConnection(  
            "jdbc:driver:protocol:databaseName",  
            "username", "password");  
}  
catch (SQLException e) {  
    System.err.println("Fatal error: cannot create connection");  
    System.exit(1);  
}
```

Voorbeeld voor hsqldb: "jdbc:hsqldb:file:dbData/demo", "sa", "sa"



# Stap 1: URL `jdbc:driver:protocol:databaseName`

- De connectie **URL** wordt beschreven in de documentatie van je databank

Voorbeeld voor `hsqldb`: `"jdbc:hsqldb:file:dbData/demo", "sa", "sa"`

Benader DB als bestand

**pathName** waar database zich bevindt

- Een bestand ondersteunt maar één gelijktijdige verbinding
- Andere HSQLDB mogelijkheden
  - `jdbc:hsqldb:mem:memdemo` //in memory DB, schrijft niet naar bestand
  - `jdbc:hsqldb:hsqldb://localhost/demo` //server mode
- URL voorbeelden voor andere databanken:
  - `Jdbc:oracle:thin:@//localhost:1521/demo`
  - `Jdbc:derby://localhost:1527/demo`
  - `jdbc:mysql://localhost:3306/demo`

## Stap 2: Maak een statement

---

Maak een **Statement** via het Connection-object:

```
Statement statement = null;
try {
    statement = connection.createStatement();
} catch (SQLException e) {
    System.err.println("Error: cannot create statement")
}
```

## Stap 3: Voer SQL uit via het statement

Voer het Statement uit via één van de drie **execute** methoden:

```
try {  
    boolean status = statement.execute (  
        "CREATE TABLE studenten...");  
  
    int rowsAffected = statement.executeUpdate (  
        "INSERT INTO studenten VALUES (...)");  
  
    ResultSet resultSet = statement.executeQuery (  
        "SELECT * FROM studenten WHERE ...");  
  
    catch (SQLException e) {  
        System.err.println("Cannot execute statement");  
    }  
}
```

**boolean**  
geeft aan of  
een ResultSet  
kan verkregen  
worden

geeft het  
**aantal**  
records dat  
gewijzigd  
werd

geeft een  
**ResultSet**  
terug met  
geselecteerde  
records

## Stap 4: Verwerk het resultaat

- Itereer door de records in de **ResultSet** en haal elke veldwaarde op:

```
try {  
    List<Student> myList = new ArrayList<>();  
  
    ResultSet resultSet = statement.executeQuery(  
        "SELECT * FROM studenten");  
  
    while (resultSet.next()) {  
        Student student = new Student(  
            resultSet.getInt("id"),  
            resultSet.getString("naam"),  
            resultSet.getDouble("score"));  
        myList.add(student);  
    }  
    System.out.println("opgehaalde data:");  
    myList.forEach(System.out::println);  
  
} catch (SQLException e) {  
    System.err.println("Cannot read ResultSet");  
}
```

bij elke **next** springt hij naar het volgende record

Voor elk datatype een specifieke **getXXX**-methode + naam van het veld in de database

## Stap 5: Sluit statement en connectie

```
finally {  
    try {  
        if (resultSet != null) resultSet.close();  
        if (statement != null) statement.close();  
        if (connection != null) connection.close();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```

Als je **try with resources** gebruikt, is expliciete close niet nodig



# Connections, Statements en ResultSets

---

- Een Connection is **duur**: je maakt een verbinding met de databank
- Je kan op een Connection meerdere statements aanmaken en tegelijk gebruiken. Deze statements lopen wel over dezelfde verbinding, en delen dus bandbreedte. Een statement is **goedkoop**.
- Je kan op een statement meerdere queries *na elkaar* uitvoeren
  - Niet tegelijkertijd. Als je een nieuwe query uitvoert op eenzelfde Statement, wordt de resultSet van de vorige query ongeldig



# Belangrijke opmerking 1: single quotes



- De SELECT-query is omsloten door `""`. Daarin omsluit je String-literals door `'`

– Voorbeelden:

```
"SELECT * FROM klanten WHERE naam = 'Peeters'"
```

```
"SELECT * FROM klanten WHERE naam = '" + klant.getNaam() + "'"
```

```
"INSERT INTO klanten(naam,score) VALUES ('" + klant.getNaam()  
+ "', '" + klant.getEmail() + "'"
```

Let op voor single quotes, komma's, haakjes!

**TIP:** doe een sout ter controle!

## Belangrijke opmerking 2: Enum



- Enum-waarden kan je omzetten naar String:

- Wegschrijven Enum: via methode **name()**

- Voorbeeld:

```
"INSERT INTO klanten(naam, geslacht) VALUES ("  
+ "'" + klant.getNaam() + "',"  
+ "'" + klant.getGeslacht().name() + "'" )"
```

single quotes!

- Inlezen Enum: via methode **valueOf()**

- Voorbeeld:

```
Geslacht geslacht =
```

```
Geslacht.valueOf(resultSet.getString("geslacht")) ;
```

## Belangrijke opmerking 3: LocalDate



- Wegschrijven: omzetten LocalDate → java.sql.Date:

`Date.valueOf(myLocalDate)`

Ook rond de datumwaarde zet je **single quotes**!

- Voorbeeld:

```
"INSERT INTO klanten(naam,geboorte) VALUES ('"  
+ klant.getNaam() + "', '"  
+ Date.valueOf(klant.getDatum()) + "')" "
```

- Inlezen: omzetten java.sql.Date → LocalDate:

`mySqlDate.toLocalDate()`

- Voorbeeld:

```
LocalDate myLocalDate =  
    resultSet.getDate("datum").toLocalDate();
```

# PreparedStatement



- SQL wordt meegegeven bij **PreparedStatement** creatie, niet bij execute
  - SQL wordt voorgecompileerd door de databank
  - PreparedStatement kunnen opnieuw uitgevoerd worden met andere parameters

```
try {  
    String sql = "INSERT INTO klanten(naam,email) VALUES(?,?)",  
    PreparedStatement prep = connection.prepareStatement(sql);  
  
    prep.setString(1,klant1.getNaam()); // naam vervangt 1e vraagteken  
    prep.setString(2,klant1.getEmail ()); // mail vervangt 2e vraagteken  
    int count = prep.executeUpdate();  
  
    prep.setString(1,klant2.getNaam());  
    prep.setString(2,klant2.getEmail ());  
    count += prep.executeUpdate();  
  
} catch (SQLException e) {  
    System.err.println("Error: cannot create klant");  
}
```

Opgelet: nummering begint bij 1

Nog een klant:

- zelfde SQL
- andere parameters

# PreparedStatement

---



- Leesbaardere SQL
  - Geen concatenatie, maar **?** als plaatshouder voor positionele parameters
  - Makkelijker formattering: geen quotes rond Strings en Dates, niet opletten met komma's in floating point nummers...
- Precompilatie
  - Sneller bij heruitvoering
  - Veiliger (beschermd tegen SQL injection attacks)

# Primary Key

- Als je een record toevoegt, kiest HSQLdb zelf een PK. Als je daarna de waarde van de PK wil weten, geef je een extra parameter mee aan het statement:

```
int count = statement.executeUpdate(  
    "INSERT INTO klanten(naam,email) VALUES ('"  
        + klant.getNaam()  
        + "', '" + klant.getEmail + "')"  
    , Statement.RETURN_GENERATED_KEYS);  
ResultSet resultSet = stm.getGeneratedKeys();  
if (resultSet.next()) {  
    int id = resultSet.getInt(1);  
    klant.setId(id);  
}
```

Extra parameter

Vraag PK waarde op

Kolom nr (1<sup>e</sup> kolom) in resultSet

Nu pas weet je de **id** en kan je de setter oproepen

- Bij een PreparedStatement kan je de parameter meegeven bij creatie:

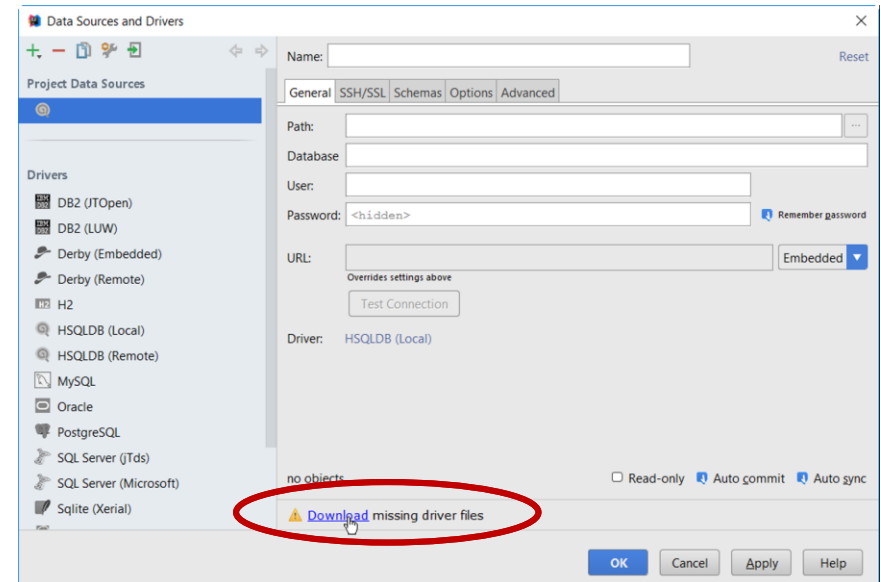
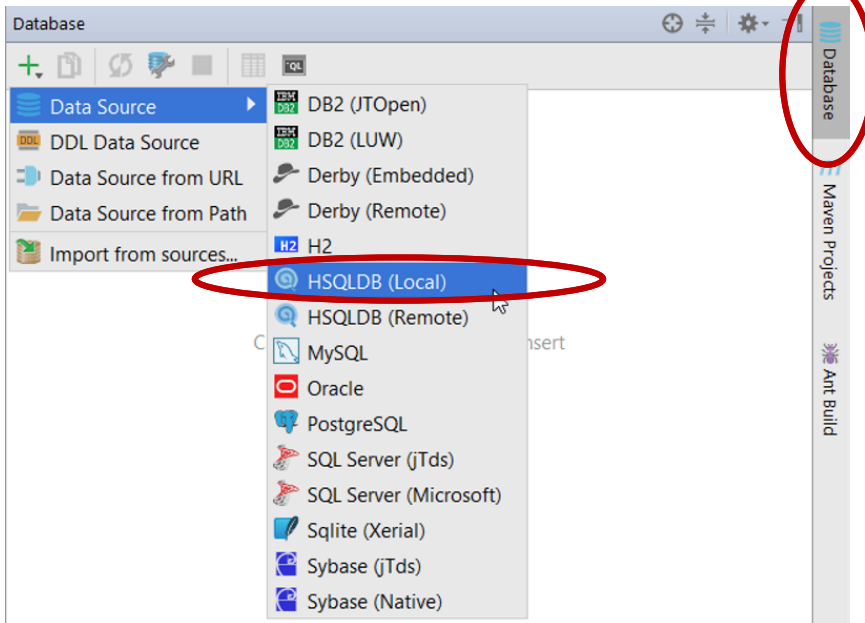
```
PreparedStatement prep = connection.prepareStatement(sql,  
    Statement.RETURN_GENERATED_KEYS);
```

# Databank ondersteuning



- Database tool window

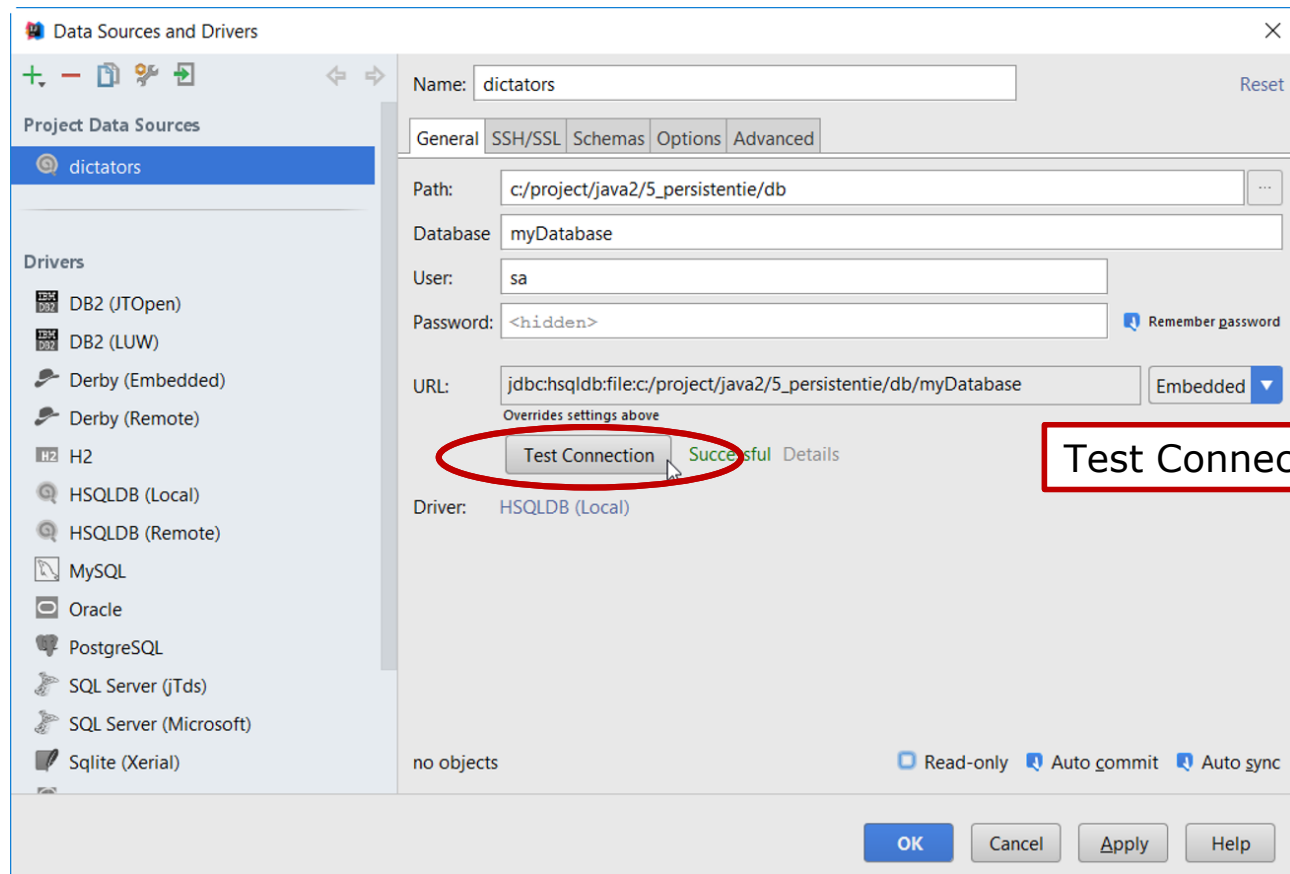
- Vraag driver download indien nodig



# Databank ondersteuning



- Connecteer
  - Als je op de driver link klikt kan je de door intellij gedownloadde jars eventueel vervangen door een andere versie op je systeem





# Databank ondersteuning



- Via het Database venster heb je allerlei tools ter beschikking om de database te bekijken of te veranderen

SQL editor



Table editor




Console view



The screenshot shows the IntelliJ IDEA Database Tools window. On the left is the DDL editor showing the CREATE TABLE DICTATORTABLE statement. In the center is the table viewer showing 13 rows of data. On the right is the schema browser showing the PUBLIC schema and the DICTATORTABLE table. A red circle highlights the toolbar in the schema browser, which contains icons for various database operations. Red arrows point from the text in the top right to the corresponding icons in the toolbar.

ID	NAAM	GESLACHT	GEBORTE	LAND	REGIME	DUUR	
1	Nicolae Ceausescu	MAN	1918-01-26	Roemenië	Despotisme	24	80
2	Jozef Stalin	MAN	1878-12-18	Rusland	Stalinisme	25	91
3	Idi Amin	MAN	1925-01-04	Oeganda	Kannibalisme	7	94.
4	Adolf Hitler	MAN	1889-04-20	Duitsland	Nazisme	12	100
5	Augusto Pinochet	MAN	1915-11-25	Chili	Militaire dictatuur	17	30
6	Mao Tse-tung	MAN	1893-12-26	China	Communisme	27	80
7	Ferdinand Marcos	MAN	1917-09-11	Filipijnen	Kleptocratie	21	50
8	Kim II Sung	MAN	1912-04-15	Noord-Korea	Stalinisme	46	80
9	Laurent-Désiré Kabila	MAN	1939-11-27	Congo	Dictatuur	4	82
10	Benito Mussolini	MAN	1883-07-29	Italië	Fascisme	21	94
11	Pol Pot	MAN	1925-05-19	Cambodja	Militaire dictatuur	4	88
12	Francisco Franco	MAN	1892-12-04	Spanje	Militaire dictatuur	36	50
13	Mobutu Sese-Seko	MAN	1930-10-14	Congo	Kleptocratie	32	60



Terwijl je met de bestands-URL geconnecteerd bent via IntelliJ, kan je NIET via JDBC naar hetzelfde bestand connecteren. (Klik dus in het database venster eerst op de  knop.)

# Databank ondersteuning



- Als de connectie naar de databank open is kan de IDE voor embedded SQL completion in Java zorgen

```
stm.execute( sql: "SELECT * from DICTATORS WHERE |")  
catch (SQLException e) {  
    e.printStackTrace();  
}
```

SLACHTOFFERS	(PUBLIC.PUBLIC.DICTAT...	DOUBLE(64, 0)
DUUR	(PUBLIC.PUBLIC.DICTATORS)	INTEGER
GEBOORTE	(PUBLIC.PUBLIC.DICTATORS)	DATE
GESLACHT	(PUBLIC.PUBLIC.DICTATORS)	VARCHAR(5)
ID	(PUBLIC.PUBLIC.DICTATORS)	INTEGER

- Connectie URL voor JDBC: `jdbc:hsqldb:sql://localhost/myDatabase`
  - default poort 9001
  - server laat meerdere simultane connecties toe
- Eerst HSQLDB als een server starten

➤ `java -classpath lib/hsqldb2.4.0.jar org.hsqldb.server.Server  
--database.0 file:data\myDatabase --dbname.0 myDatabase`

Databank bestand

Databank naam in connectie URL

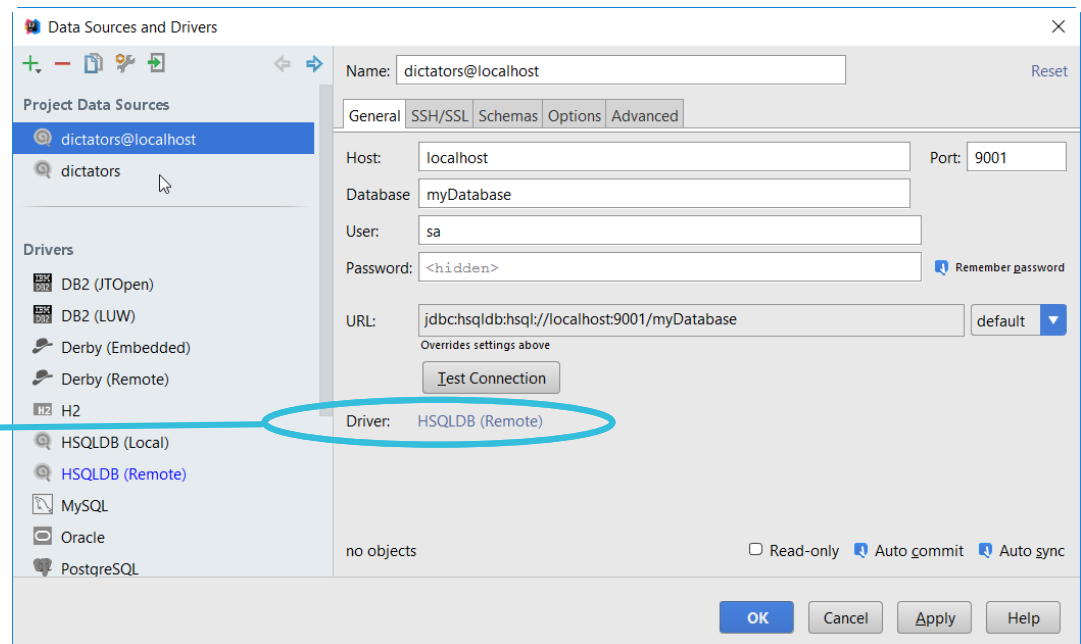
- in voorbeelden staat een hsqldb.bat met dit commando



Vanuit IntelliJ Connecteren nu naar HSQLDB server

- je kan database tools gebruiken terwijl je een JDBC connectie uit je programma hebt


hier kan je driver (jar)  
aanpassen naar laatste  
versie



## Korte opdracht

---



- Neem de voorbeeldcode 2\_JDBC erbij.
- Voer volgende SQL-bewerkingen uit via code:
  - Verwijder alle records waarvan de score hoger is dan 60 ("**DELETE FROM...** ")
  - Verander de naam van het record met id=2 in je eigen naam ("**UPDATE...** ")
  - Bekijk nadien de data via de Data editor. Vergeet niet eerst "*Reload page*" te doen via: 

# Agenda

---

## 1. Serialization

- Serialization / deserialization
- Voorbeeld
- Pro en contra

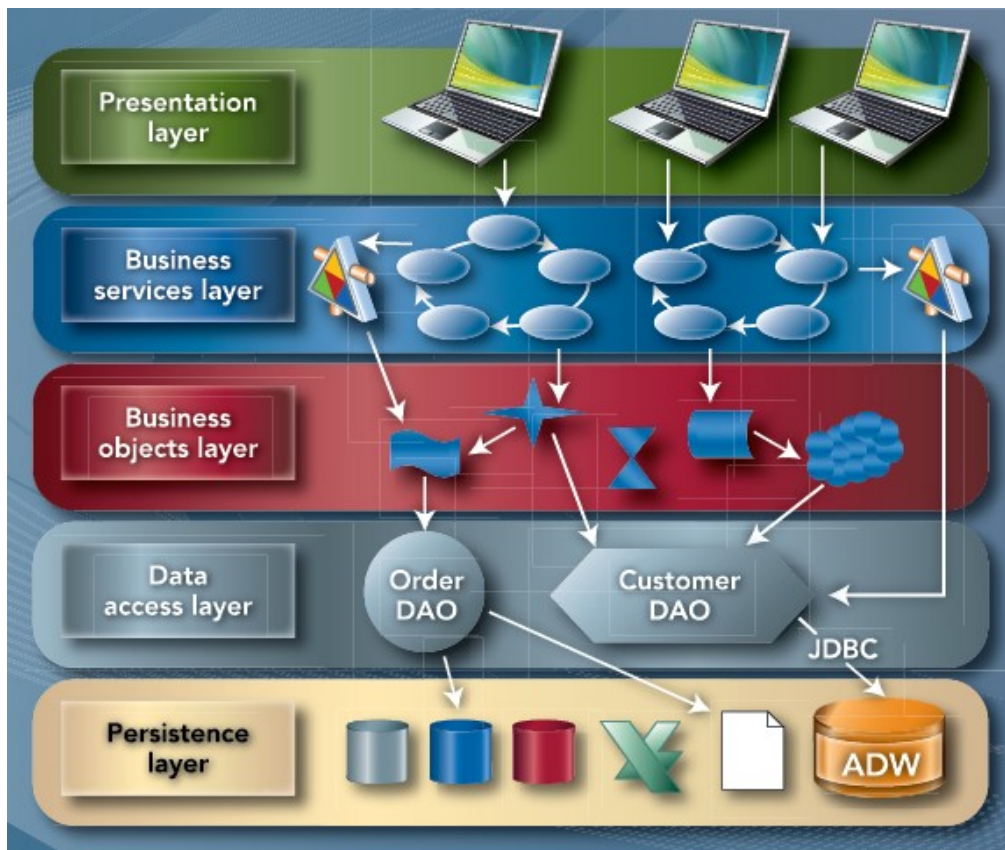


## 2. JDBC

- JDBC driver
- Datatypes in SQL versus Java
- JDBC met Java:
  - Connection, Statement, ResultSet
  - PreparedStatement

## 3. DAO

- DAO pattern
- Object relational mapping



---

# DAO - pattern



# Probleem

---

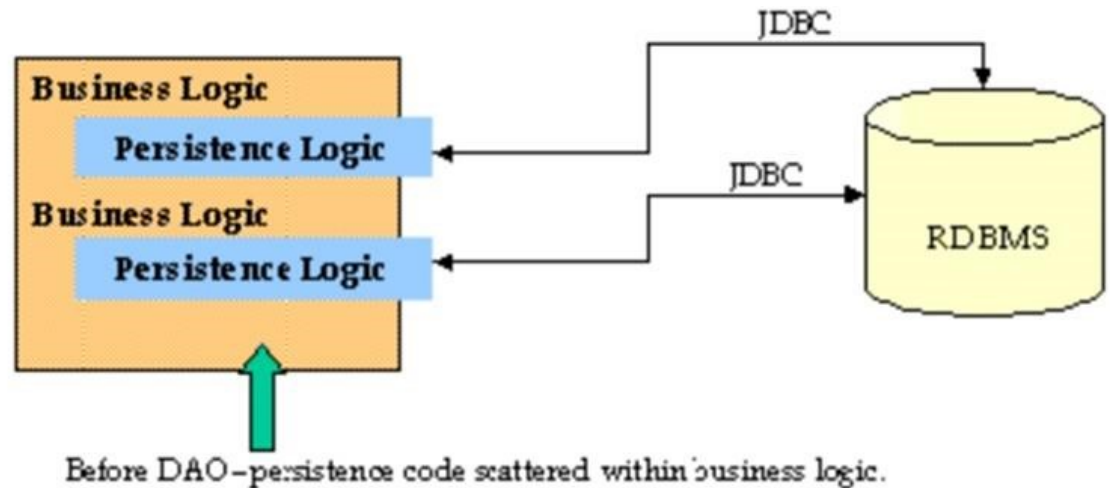
- Java
  - Objectgeoriënteerd
- Database
  - Meestal relationeel model (primary & foreign keys)
- Communicatie tussen Java en database?



# Het DAO pattern

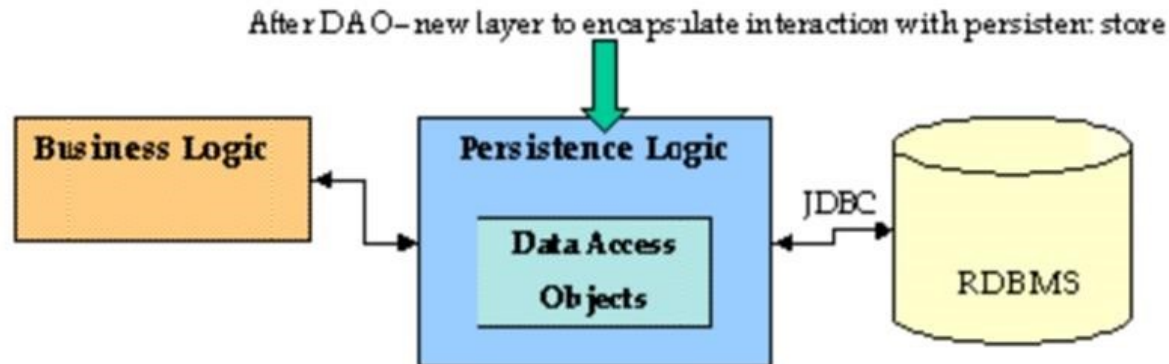
- Probleem:

**Business logica** en **persistentie logica** zijn met elkaar verweven. Zeker bij grote applicaties zorgt dit voor complexe en moeilijk onderhoudbare code!

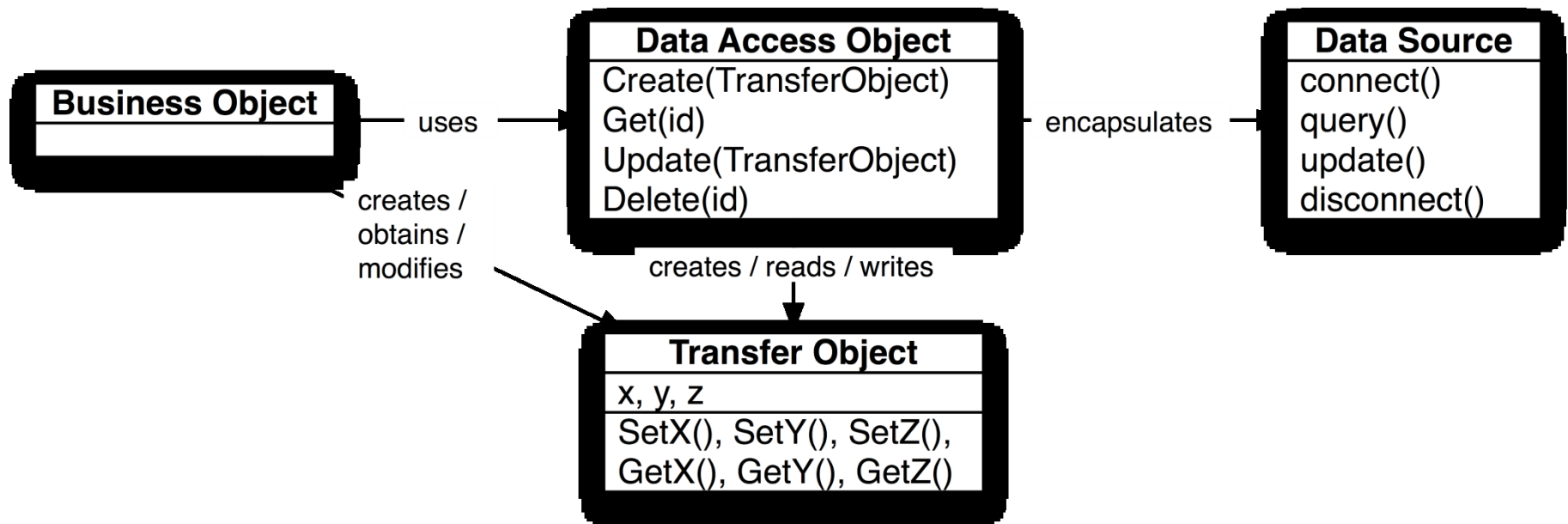


- Oplossing:

Persistentie logica vormt een **aparte laag**. Best voor elke database-tabel een apart **Data Access Object**!



# DAO-pattern: klassediagram



Meer weten?



<http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>

# DAO: verantwoordelijkheden

---

- **Business Object**

- is de **data client**, die bedrijfsregels, berekeningen uitvoert, ...

- **Data Access Object (DAO)**

- vormt een abstracte tussenlaag tussen het Business Object en de data source
- Ook wel Repository genoemd

- **Data Source**

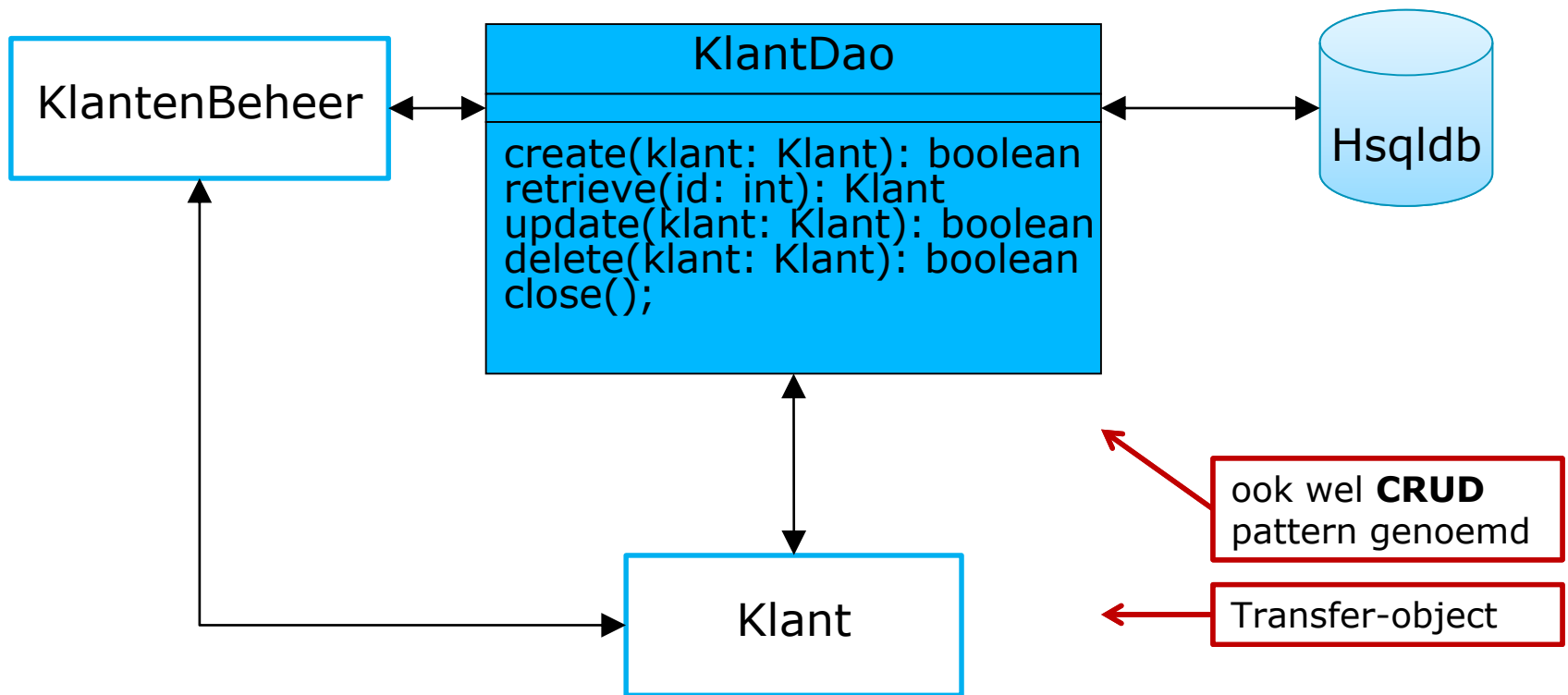
- biedt **CRUD-services** aan het Business Object
- kan opslag zijn zoals RDBMS, OODBMS, XML repository, flat file system...
- kan ook een ander systeem zijn (legacy/mainframe), service (B2B service), of een soort van repository (LDAP).

- **Transfer Object**

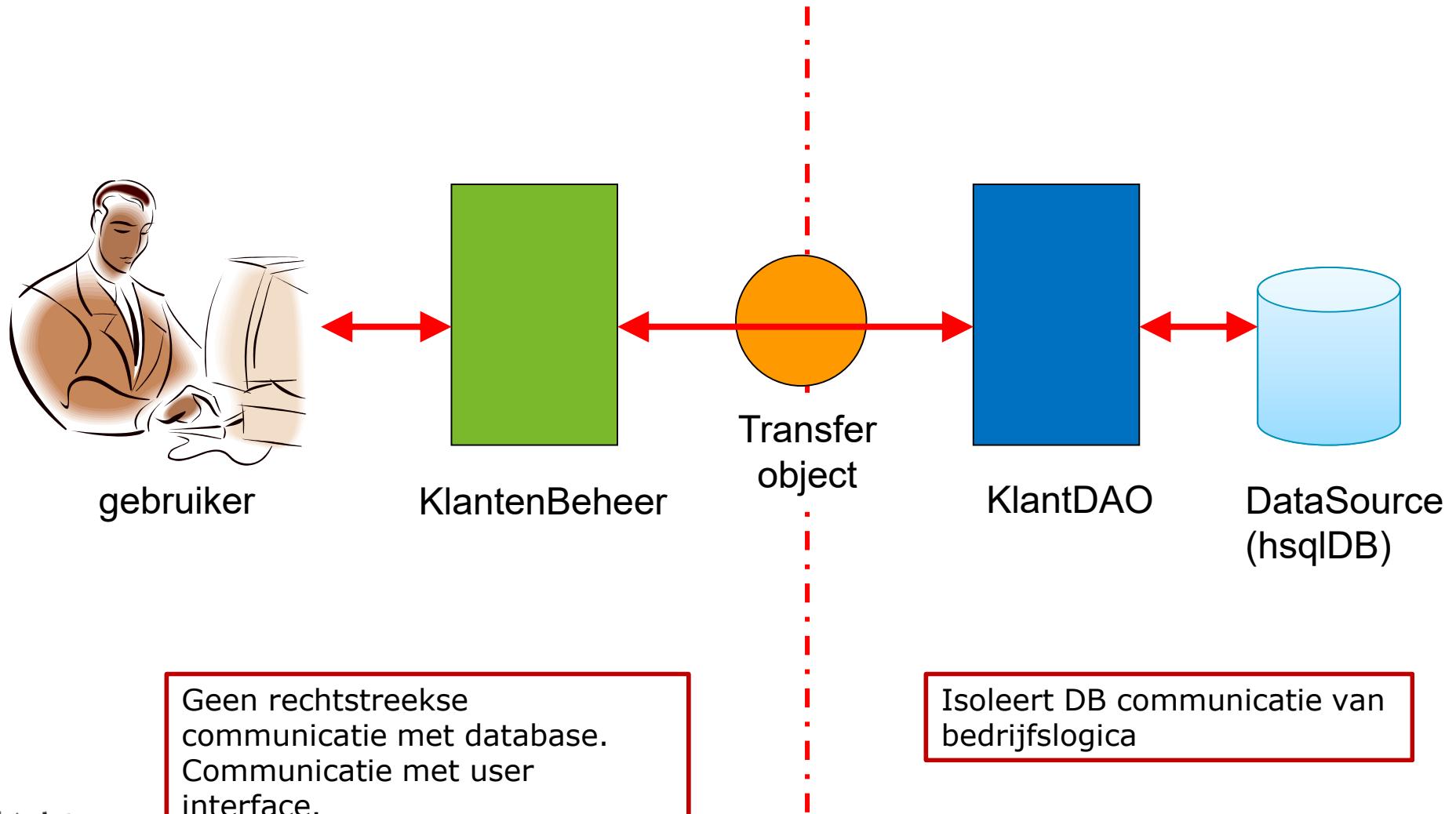
- **data drager**, zowel om data te passeren als parameter (vanuit Business Layer), als om data te retourneren (vanuit DataAccess Layer)

# DAO pattern: voorbeeld

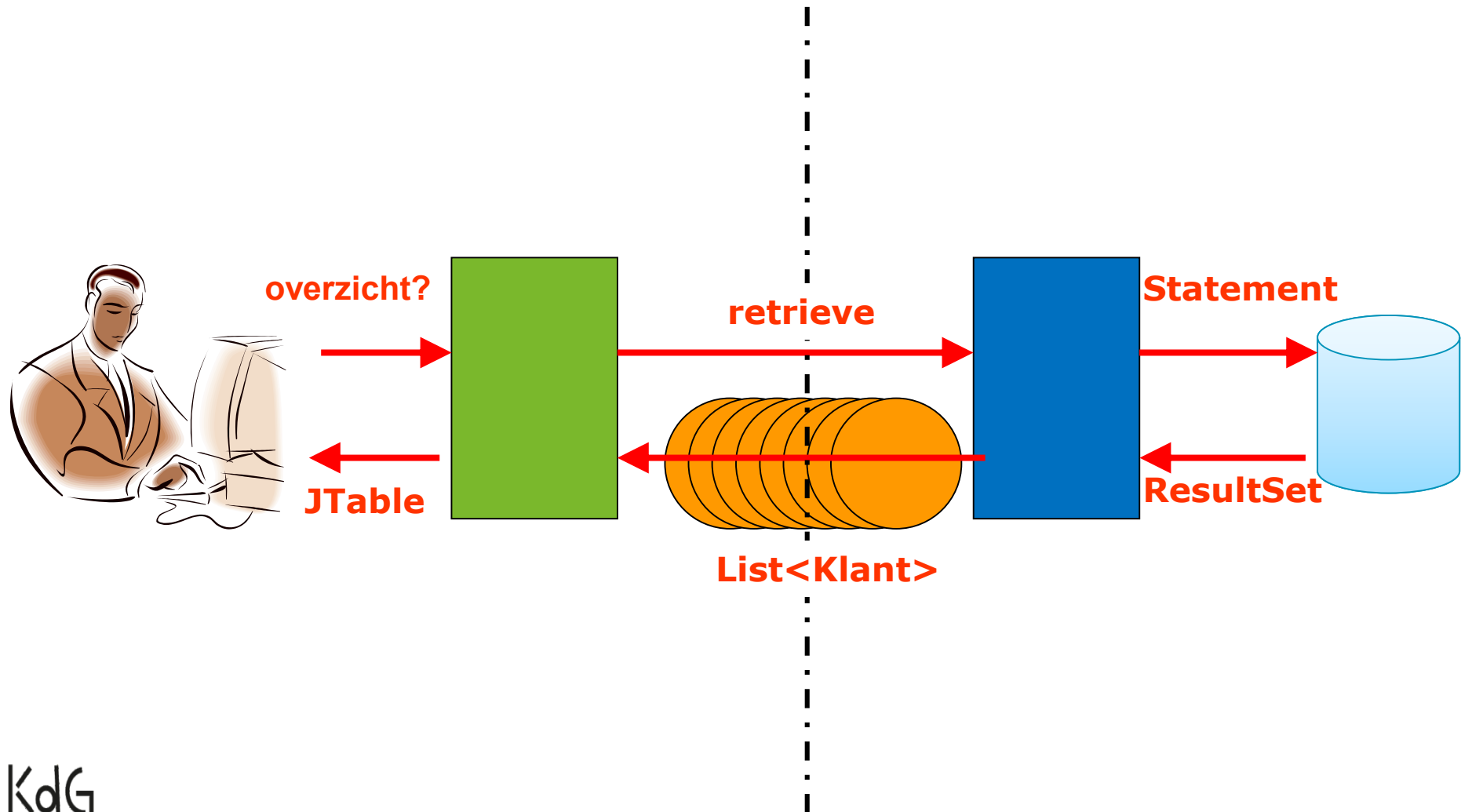
- Door de band een DAO per transfer object (of tabel)



# Gelaagde structuur

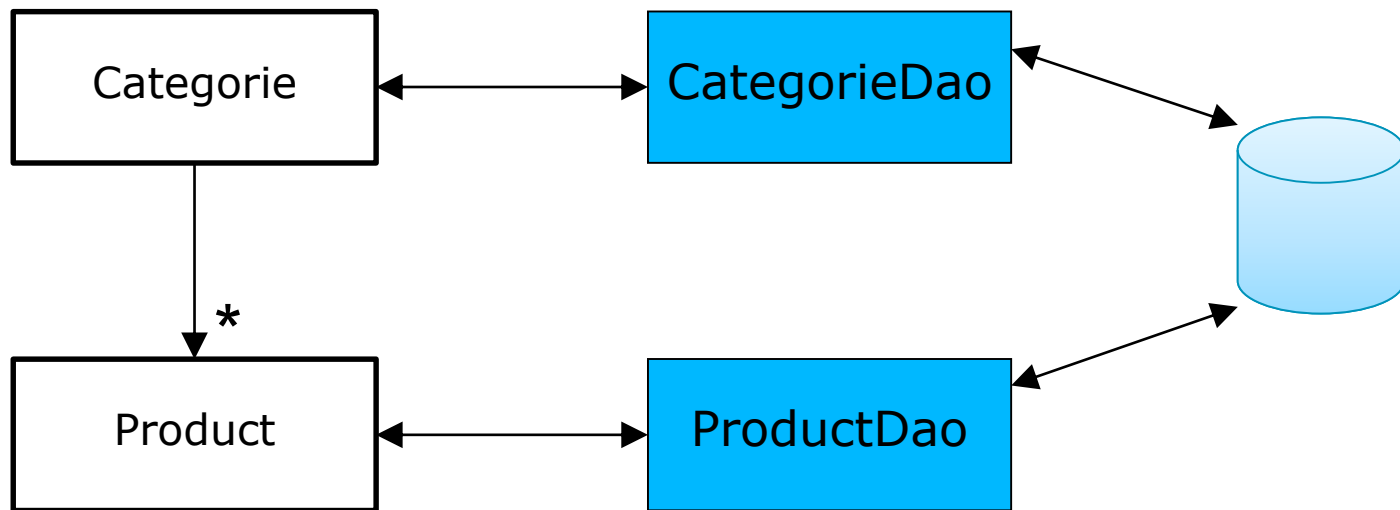


# Voorbeeld: klantenoverzicht



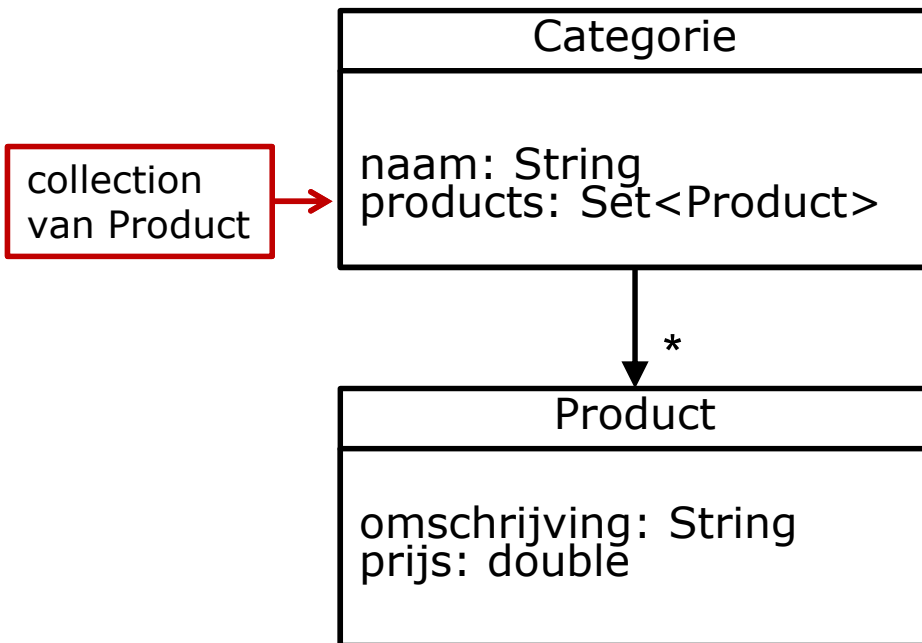
# Object-relational mapping

- Probleemsituatie:
  - Producten worden gegroepeerd per categorie.
  - DUS: **1 op veel relatie** tussen Categorie en Product
  - Hoe lossen we dit op?

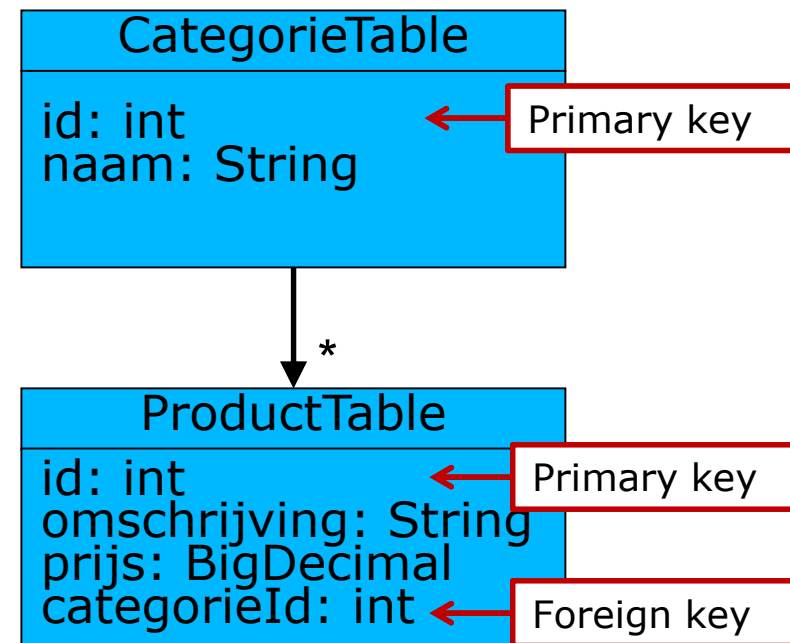


# Object-relational mapping

Puur OO  
(in Java):



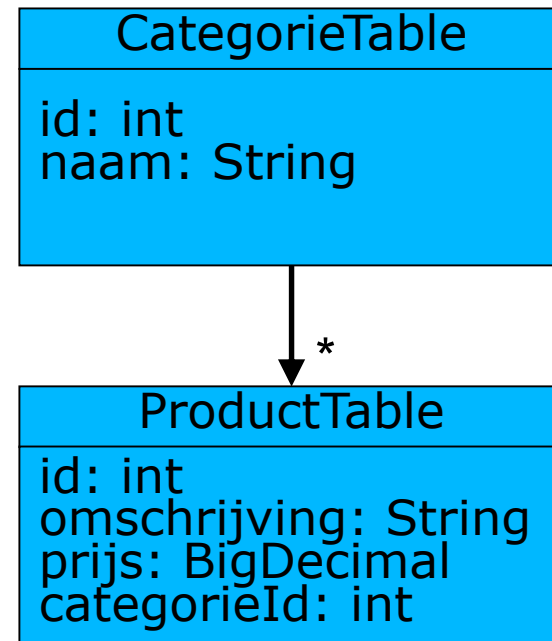
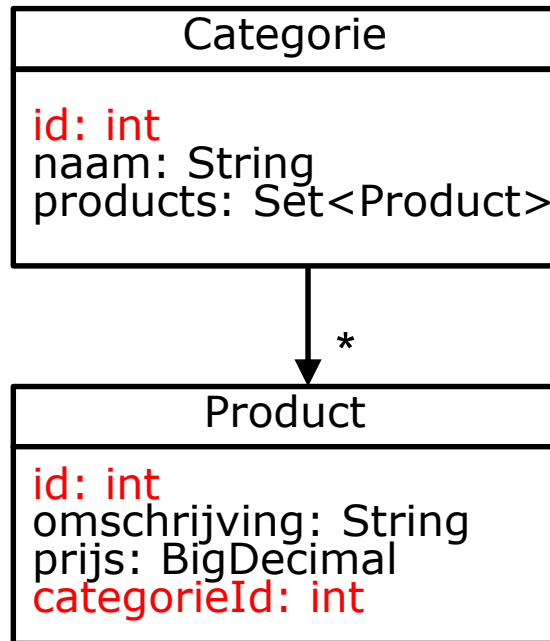
Puur relationeel  
(in database):





# Object-relational mapping

Hybride  
(aanpassing in Java):



# Object-relational mapping

---

- Samengevat:
  - voeg id toe aan de klassen
  - Voeg eventueel foreign key toe aan **Product**
  - Laat de DAO de set creëren in **Categorie**
- Wat gebeurt er bij:  
`categorieDao.retrieve(...)`  
→ is dit efficiënt?



Programmeren 3-Java: Hibernate  
en Java Server Faces  
.Net Entity Framework

# Gebruik van connecties

---

- Als je met een databank bestand werkt, zoals in eerdere slides, kan je maar één connectie tegelijk openen (in dezelfde of een andere toepassing).

Typisch patroon:

– maakConnectie

- maakStatement

- doe uw ding met het statement

- sluitStatement

– sluitConnectie (zet het in de finally)

– Je kan try-with-resources gebruiken om te sluiten

- Bestudeer:
  - Merk op dat de klasse `DatabaseConnection` en `GebruikersDAO` singleton zijn gemaakt
  - In `GebruikersDAO` wordt *try with resources* gebruikt. Daardoor worden gecreëerde `Statement`-objecten automatisch gesloten.
  - In `GebruikersDAO` wordt `PreparedStatement` gebruikt.
  - De view klassen zijn uitgewerkt in swing, maar dat is irrelevant voor het voorbeeld.
  - In `Gebruikers.java` vind je login gegevens

# Opdrachten

---

- Groeiproject
  - module 7  
(deel 3: "JDBC")
- Opdrachten op BB
  - Laptop opgave
  - DAO opgave 1
  - DAO opgave 2
- Zelftest op BB

