JDBC: Java Database Connectivity

Prof. Dr. Inga Saatz

Nov 03, 2023

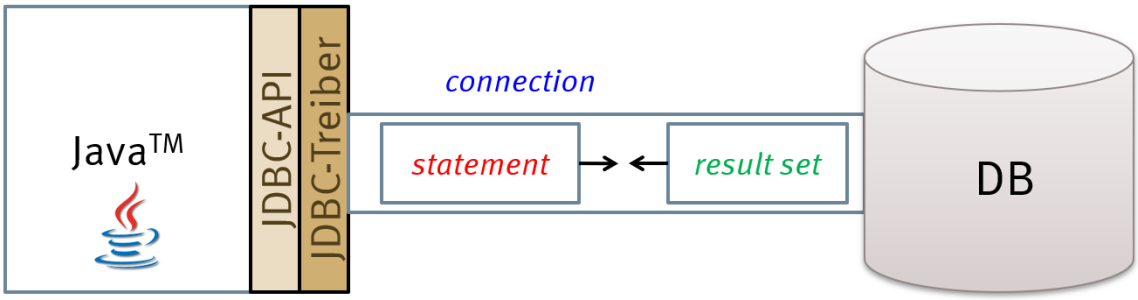
Note: Nach dieser Vorlesung sollten Sie:

- Die benötigten Treiber zur Herstellung einer Datenbankverbindung mit JDBC in ein Java-Projekt einbinden.
- Eine Datenbankverbindung aus Java heraus mit der Datenbank Oracle herstellen.
- Anfragen an die Datenbank senden, Antwort des DBMS auswerten und benutzerfreundliche Fehlermeldungen ausgeben.
- Gefahr von Code-Injections erkennen und diese durch die Verwendung von PreparedStatements verhindern.
- Die Drei-Schichten Architektur einer Datenbankanwendung entwerfen und implementieren.

Das Java-Package `JDBC`

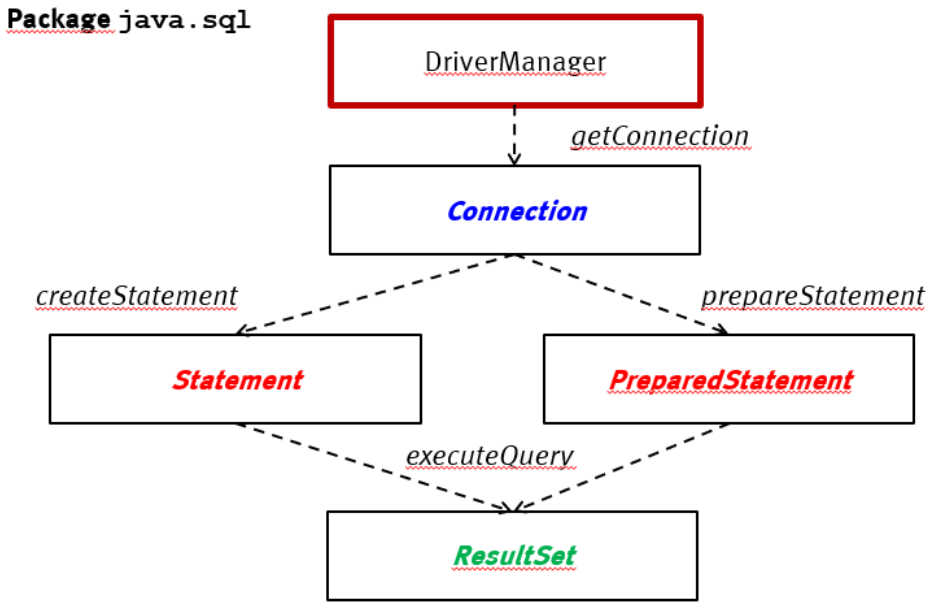
Hierzu gibt es das Video JDBC-API

Java Database Connectivity (JDBC) JDBC ist eine Programmierschnittstelle für Datenbank-Zugriffe in Java. Um SQL-Anweisungen auf einer Datenbank auszuführen, müssen zusätzlich ein Treibern eingebunden werden. Der Treiber enthält die Implementierung der JDBC-API für die verwendete Datenbank.



Attention: Vorgehensweise

1. Einbindung eines JDBC-Treibers (*Anhang 1*)
2. *Aufbau der Datenbankverbindung*
3. *Kommunikation mit der Datenbank* (Statements senden und ResultSets auswerten)
4. Datenbankverbindung schließen und die Ressurcen wieder freigeben



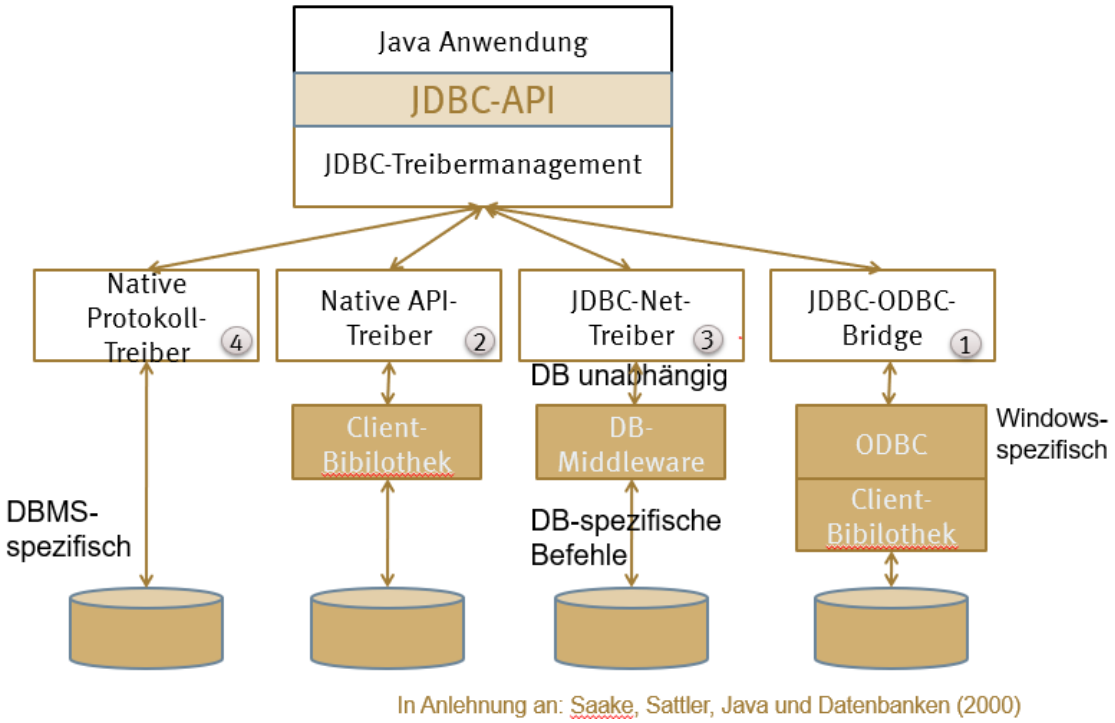
Tip: Erläuterung Die Klasse **DriverManager** bietet eine statische Methode `getConnection`, um ein Connection-Objekt zu instanziieren.
Die Klasse **Connection** bietet Methoden an, um

- Verbindung zur Datenbank herzustellen und zu schließen,
- die Verbindungseigenschaften abzufragen,
- SQL-Anweisungen zu kapseln und an das DBMS zu senden,
- auf den Katalog (Data Dictionary) zuzugreifen sowie
- Transaktionen zu steuern (commit, rollback).

Für Datenbankanfragen werden in den Klassen **Statement**, **PreparedStatement**, **PrepareCall** und **NativeSQL** gekapselt. **Statement** und **PreparedStatement** werden für Anfragen und Änderungsoperationen genutzt. **PrepareCall** wird zum Aufruf von gespeicherten Prozeduren verwendet. **NativeSQL** wird zur Ausführung von herstellerspezifischen Befehlen verwendet, beispielsweise wenn die Befehle nicht im JDBC-API vorhanden sind, beispielsweise der SQL-Befehl `DESCRIBE`.

JDBC-Treiber

Es gibt vier verschiedene JDBC-Treiberarten, welche die JDBC-Schnittstelle implementieren:



Typ 1

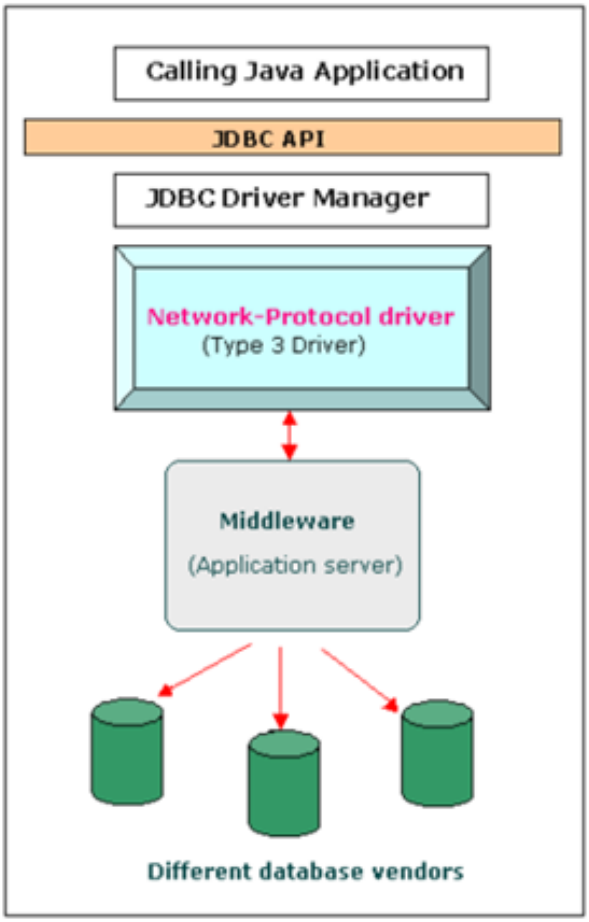
Ein **Typ-1**-Treiber wird verwendet, wenn es zu der Datenbank einen ODBC-Treiber, jedoch keinen eigenständigen JDBC-Treiber gibt. Der JDBC-ODBC-Bridge-Treiber wandelt JDBC- in (Windows-spzeifische) ODBC-Anfragen um.
Typ-1-Treiber werden verwendet, wenn es zu der Datenbank keinen eigenständigen JDBC-Treiber gibt.

Typ 2

Ein **Typ-2**-Treiber kommuniziert über eine plattformspezifische Programmbibliothek auf dem Client mit dem Datenbankserver.
Beispiel: Oracle JDBC OCI client-side driver:
Über Java-Methoden (=native API-Treiber) werden C-Methoden einer C Library (=Client-Bibliothek) aufgerufen.

Typ 3

Mittels des **Typ-3**-Treibers werden die JDBC-API-Befehle in generische DBMS-Befehle übersetzt und an einen Middleware-Treiber auf einem Anwendungsserver übertragen. Der Anwendungsserver transformiert die Befehle in den DBMS-spezifischen SQL-Dialekt. Verwendung: Beispielsweise bei der Kombintion verschiedener Datenbanken



Typ 4

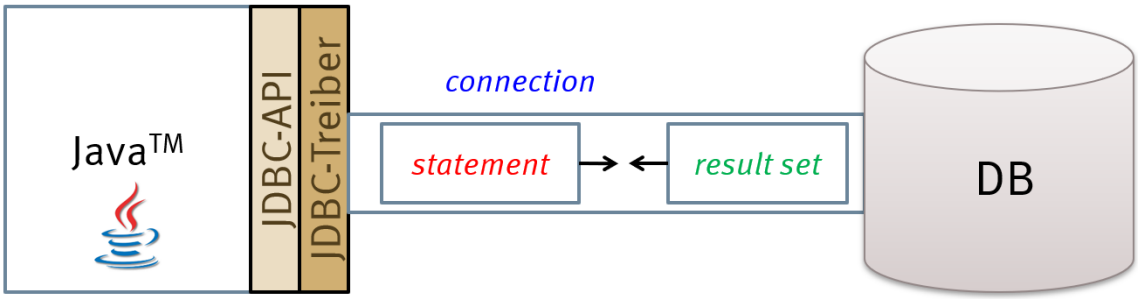
Beim **Typ-4**-Treiber werden die JDBC-API-Befehle direkt in DBMS-Befehle des jeweiligen Datenbankservers übersetzt. Dieses bietet eine höhere Performanz als die Verwendung eines Typ-3-Treibers, ist aber weniger flexibel.
Beispiele:
MySQL Connector/J (reine JAVA-Implementierung)
Oracle Thin clientside driver (reine JAVA-Implementierung)

```
//%jars lib/ojdbc11.jar
%maven com.oracle.database.jdbc:ojdbc11:23.3.0.23.09
```

Aufbau einer Datenbankverbindung

Hierzu gibt es ein Video: JDBC-Verbindung

Was ist eine Connection?



Grundprinzip

Schritte

- Aufbau der Verbindung (*connection*) zur DB
- Senden der SQL-Anweisung (*statement*)
- Verarbeiten der Anfrageergebnisse (result set)
- Schließen der Verbindung (*connection*) zur DB

(abhängige Statement- und ResultSet-Objekte werden mit geschlossen)

Sequenzdiagramm

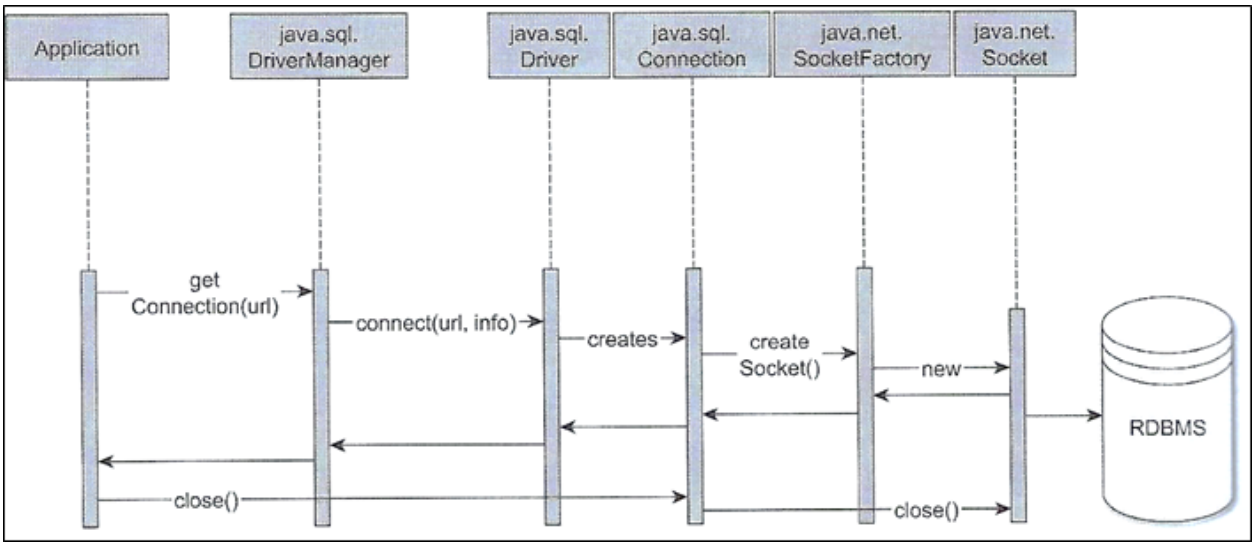
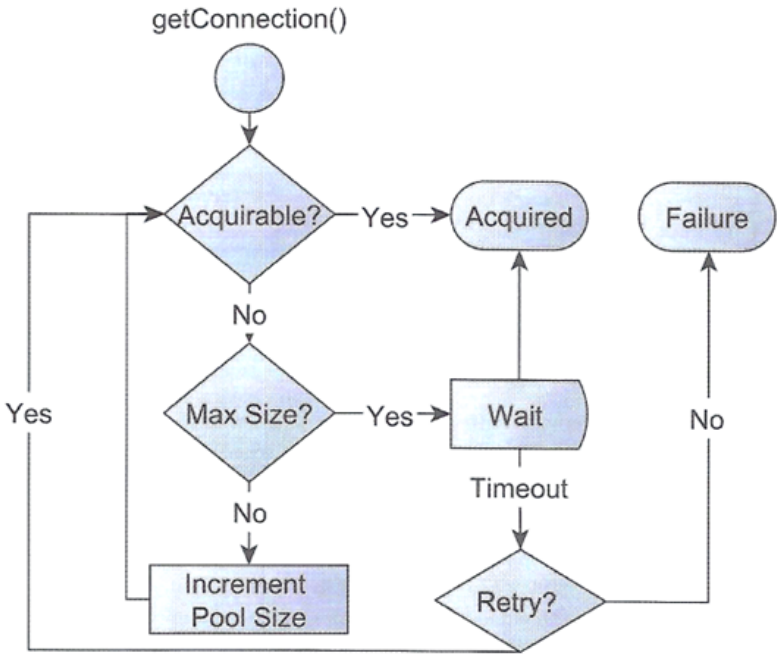


Abb. aus V. Mihalcea, High-Performance Java Persistence, S.14

Pooled Connection Eine **Pooled Connection** ist eine wiederverwendbare Datenbankverbindung. Sie wird beim Aufruf von close() nicht physisch geschlossen, sondern lediglich freigegeben und bleibt für nachfolgende Client-Zugriffe erhalten. In einem **Connection Pool** werden diese gespeichert. Alle Verbindungen im Connection Pool sind dabei mit derselben Datenbank verbunden.



Note: Beispiel: Auswirkung auf die Laufzeit

Connection Pooling durch den Oracle-JDBC-Treiber

Programmlauf	1. Verbindung	bei Wiederverwendung (Mittelwert aus N=10)
1.	404 ms	16 ms
2.	431 ms	16 ms
3.	488 ms	20 ms
4.	403 ms	16 ms
5.	536 ms	15 ms

Anm: Ändern der max. Anzahl von Connections auf 200 (Oracle Express Ed.):
ALTER SYSTEM SET processes=200 scope=spfile;

Aufbau einer Datenbankverbindung

Herstellung der Verbindung mit der Praktikumsdatenbank:

```
    import java.sql.*;
    Connection con;
    String url="jdbc:oracle:thin:@172.22.160.22:1521:xe";
    con = DriverManager.getConnection(url,<user>,<password>); //User-Credentials␣
↪anpassen
    System.out.println(con);
```

Wird dieser Code ausgeführt, so sollte die Objekt-Id des Connection-Objekts ausgegeben werden, z.B. `oracle.jdbc.driver.T4CConnection@6517c39d`.

```
// 2. Herstellung der Verbindung mit der Datenbank
    import java.sql.*;
    Connection con;
    String url="jdbc:oracle:thin:@172.22.160.22:1521:xe";
    con = DriverManager.getConnection(url,"C##FBPOOL2","oracle"); //User-Credentials␣
↪anpassen
    System.out.println(con);
```

```
oracle.jdbc.driver.T4CConnection@1f2e3f61
```

Hint: SQLException `No suitable driver found` Die SQLException `No suitable driver found` signalisiert, dass der Treiber noch nicht richtig eingebunden ist.
Tipp:

- Ist der JDBC-Treiber eingebunden? vgl. *hier*
- Werden im Code die Treiber geladen mit der *`%jars`* oder *`%maven`* Direktive?
- Bleibt die Fehlermeldung, wenn der Kernel neu gestartet und der Output gelöscht wird?
- Wenn mit `%jars` gearbeitet wird:
  - Ist die jar-Datei im Dateisystem vorhanden?
  - Stimmt der angegebene Dateipfad, d.h. wird die jar-Datei im Dateisystem gefunden? Dies kann mit dem folgenden Code geprüft werden (Ausführen in einer Codezelle):

    ```
    List<String> added= %jars lib/ojdbc11.jar
    System.out.println(added);
    ```

Mehr zu Exceptions finden Sie *hier*

Am Ende wird die Verbindung wieder geschlossen, um Ressourcen freizugeben.

```
con.close();
```

Note: Fehlerbehandlung
Typischerweise werden JDBC-Befehle in einem try-catch-Block verwendet, um auftretende Fehler zu behandeln (vgl. *Anhang 2*):

```
try{
     // JDBC-Befehle
}catch(SQLException ex){
```

(continues on next page)

(continued from previous page)

```
    // Behandlung der Fehler, z.B. Loggen, Signalisieren des Fehlers, ...
}finally{
    // optionaler Codeblock, der immer durchlaufen wird
    // Hier könnte bspw. die Connection geschlossen werden, wenn das Programm␣
 ↪beendet wird.
}
```

Alternative Syntax mit automatischer Ressourcenfreigabe (ab Java 7)
Der **try-with-resources** Befehl ist ein try-Block, in dem eine oder mehrere Ressourcen deklariert werden, die nach Abarbeitung des try-catch-Blocks automatisch freigegeben werden.
Beispiel:

```
try (Connection con= DriverManager.getConnection(url, user, password)){
  Statement stmt = con.createStatement();
  ...
  con.close();
}catch(SQLException ex){
  ...
}
```

Danger:

Vorsicht!

Die User-Credentials nicht als plain-Text im Code angeben.

Tip: Best Practice User Credentials Sinnvoll ist es, dass die Credentials in einer separaten Datei serverseitig zu speichern (und **nicht** im Code fest zu verdrahten).
Alternativen:

- Passwörter über die Konsole abfragen und URL aus Variablen zusammensetzen
- Verbindungsdaten in Konfigurationsdateien separat speichern (Name der dns.ini Datei in den Konfigurationsparametern hinterlegen)
- Verbindungsdaten als Environment-Variable speichern (z.B. in der Docker-Datei).

Aber:

- Bei einer Eingabe der Credentials über eine (Web-)Anwendung kann zwar das Passwort durch die Webanwendung verschlüsselt werden. Allerdings können Angreifer ggf. das verschlüsselte Passwort mithören und direkt an die Datenbank per JDBC senden.
- Sicherer ist es, den Nutzern **keinen** direkten Zugriff auf die Datenbank zu geben sondern stattdessen die Datenbank durch eine API zu kapseln. Siehe *Drei Schichten Architektur*.

Ausführung von Statements

Direkte Ausführung von CRUD-Operationen

- Der SQL-Befehl wird direkt als Zeichenkette an das DBMS gesendet und ausgeführt.
- Die Ausführung des SQL-Befehls erfolgt mit der Methode `Statement.executeUpdate()`. `CRUD`-Befehle, also `CREATE`, `INSERT`, `UPDATE` und `DELETE`, liefern nur eine Anzahl der geänderten Tupel und **keine** Ergebnismenge zurück.

Beispiel

Füge die neue Kundin Agnes Neumann aus Dortmund in die Datenbank ein.

Java

```
String   sqlString=  "Insert Into Kunde (Kundennummer, Anrede, Nachname, Vorname, Ort)
 ↪"
                  +   "VALUES (25, 'Frau', 'Neumann', 'Agnes', 'Dortmund')";
Statement stmt = con.createStatement();
int anzahl = stmt.executeUpdate(sqlString);
stmt.close();
```

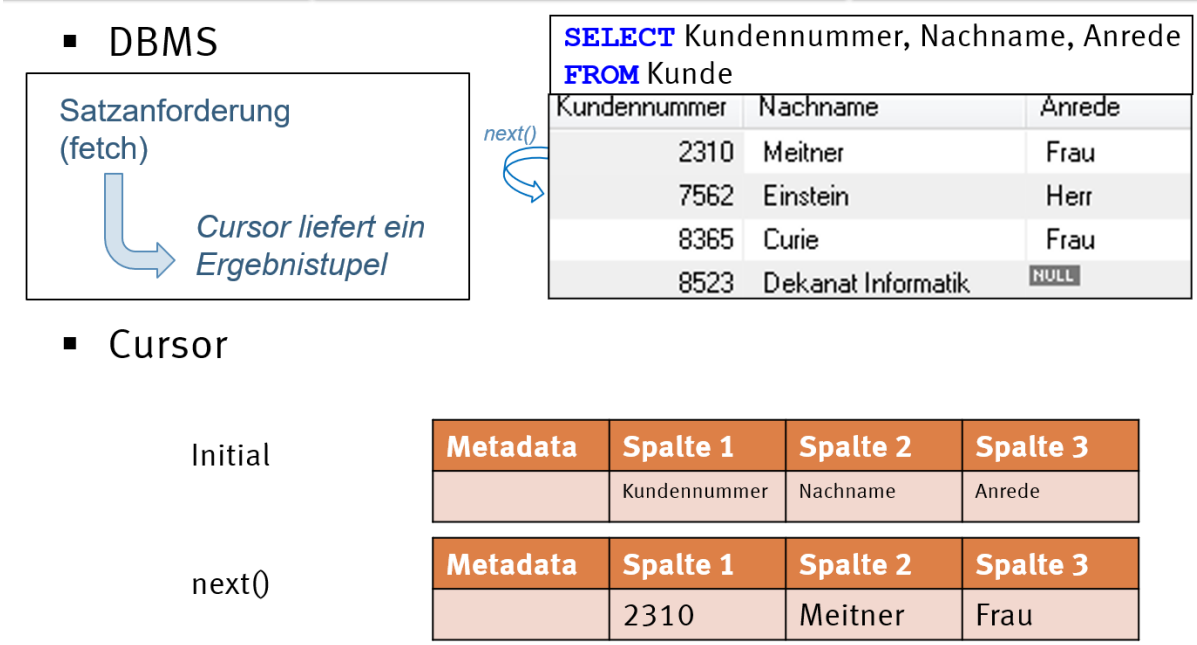
Beispiel

Die Kundin Agnes Neumann ändert ihr Geschlecht und heißt jetzt Andreas.

Direkte Ausführung von SQL-Anfragen

Erläuterung hierzu im Video JDBC-API

- SQL-Anfragen liefern eine Ergebnismenge zurück.
- Das (mengenwertige) ResultSet wird von der Datenbank auf den Client übertragen.



- Im Javaprogramm wird das ResultSet in einer Schleife zeilenweise ausgewertet.
  Das ResultSet ist mit einem Cursor (=Zeiger auf ein Tupel in der Datenbank) verbunden.

Code-Beispiel:

```
void leseKunde(String sqlString) throws SQLException{
    System.out.println("SQL-Anfrage: "+sqlString);
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(sqlString);
    while (rs.next()){
            String knr  = rs.getString(1);
            String name = rs.getString(2);
            String ort  = rs. getString(3);
            System.out.println(knr + " :  "+name + "  , "+ ort);
    }
    stmt.close();
}
```

Attention: Behandlung von NULL-Werten

Anrede	Name	Geburtsdatum
Herr	Einstein	1879-03-14
NULL	Dekanat Informatik	NULL

- Überprüfen auf NULL-Werte

```
String columnValue = rs.getString(1);
If (rs.wasNull()){
        // Nullwert behandeln
        // (Bezogen auf den letzten ausgelesenen Wert)
}
```

- Überprüfung auf Null-Werte erforderlich bei
  - getXXX-Methoden
    - getString(),
    - getDate(),
    - ...
  - getInt(), getDouble(), ... (NULL-Werte liefern 0)
  - getBoolean() (NULL-Werte liefern false)

Erläuterung hierzu im Video: JDBC-Datentypen

Note: Abfragen von SQLWarnings Mit der Methode ResultSet.getWarnings() kann abgefragt werden, ob eine Warnung bei der Ausführung einer Abfrage aufgetreten ist. Alternativ kann dies auch im try-catch-Block erfolgen. Im Video JDBC-Datentypen gibt es dazu ein Beispiel.

Note: Wetere ResultSet-Methoden Häufig verwendete Methoden:

- next()
- gettype(nr)
- gettype(name)
- wasNull()
- getMetaData()

Erweiterungen zur Navigation innerhalb des ResultSets

- absolute(nr)
- afterLast()
- last()
- first()
- beforeFirst()
- previous()

- isAfterLast()
- isLast()

Anfälligkeit für SQL-Injections

Einfallstore sind beispielsweise:

- Eingabefelder, über die andere SQL-Befehle ausgeführt werden.
- Fehlermeldungen des DBMS/Webservers, welche Infomationen über die Datenbank und den Server preisgeben.
- Laufzeitmessungen zu dem Systemverhalten nach Testeingaben, um bspw. den Aufbau von Tabellen zu ermitteln (Beispiel vgl. Blind intrusion).

Fallbeispiel:

```
void einenKundenAnzeigen() throws SQLException{
    Scanner s = new Scanner(System.in);
    System.out.println("Bitte geben Sie die Kundennummer ein:");
    String eingabe = s.nextLine();  // Lesen der Eingabe über die Konsole
    leseKunde("SELECT kundennummer, nachname, ort FROM Kunde WHERE kundennummer=
↪"+eingabe);
}
```

Erwartete Nutzung

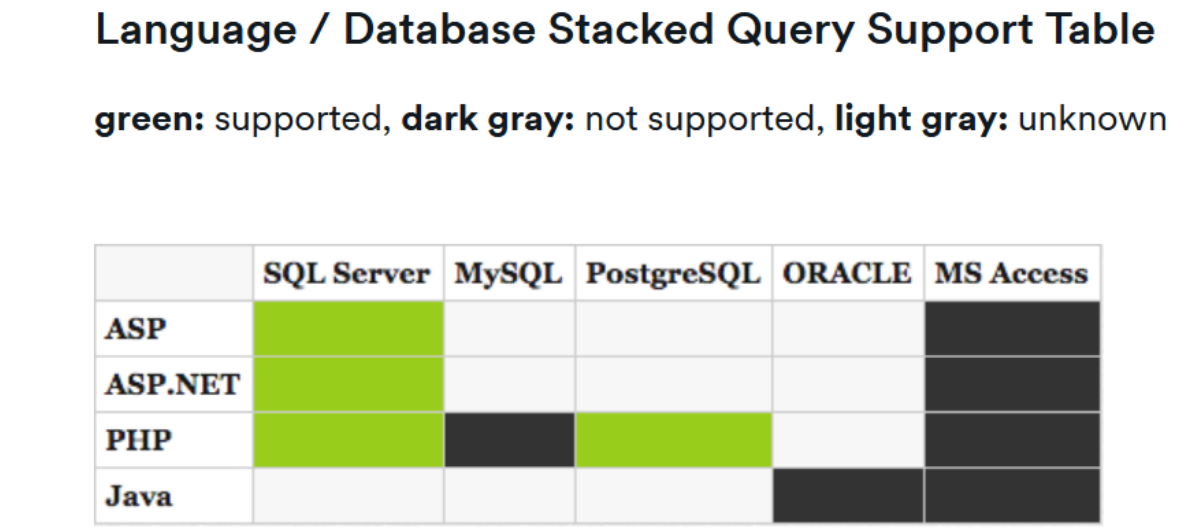
Zeige einen vorhandenen Kunden an mit bekannter Kundennummer. `einenKundenAnzeigen()`

```
einenKundenAnzeigen();
```

Attention: SQL-Injections **SQL-Injections** sind immer dann möglich, wenn benutzerkontrollierte Daten ungefiltert in SQL-Abfragen eingebaut werden. Dadurch können beliebige Befehle innerhalb der Datenbank ausgeführt werden, beispielsweise können Angreifer Nutzerrechte und Daten auslesen und ändern.

Hint: Stacking Query Injection
Das hintereinander Ausführen von durch ; getrennte Befehle wird als Stacking Query Injection bezechnet.
Beispiel: `'-1; ALTER USER system IDENTIFIED BY changedpassword --`

Der JDBC Treiber führt nur jeweils einen Befehl aus, daher führt dieser Versuch mit JDBC zu einem Syntaxfehler. In anderen Programmiersprache ist dies jedoch teilweise möglich.



Quelle: Cheatsheet

Schutz vor SQL-Injection

Einen absoluten Schutz vor SQL-Injection gibt es nicht.
Als Software-Entwickler sollten zwei Aspekte fokussiert werden:

1. Nutzung der Klasse *Prepared Statement* anstelle der Klasse *Statement* in JDBC.
2. Nur verständliche Fehlermeldungen zurückgeben und nicht den gesamten Stacktrace. (vgl. *Exceptions*).
3. Kapselung der Datenbank durch eine API (vgl. *Drei-Schichten-Architektur*)

Prepared Statements

Prepared Statement Ein prepared Statement ist ein parametrisiertes Statement, bei dem der Platzhalter '?' für Attributwerte verwendet wird.
Die '?' werden durchgezählt und die Platzhalter anschließend mit Werten belegt.

- Nutzung von **Prepared Statements** (=parametrisierten Templates) für die auszuführenden SQL-Befehle
- Ein **Prepared Statement** wird in zwei Schritten an das DBMS gesendet:
  1. Schritt: Senden eines Statement-Templates, welches durch das DBMS vorcompiliert wird.
  2. Schritt: Senden des mit Werten belegten Templates an das DBMS.
- Vorteile:
  - Schnellere Ausführung mehrerer gleichartiger Statements nacheinander

```
------------ ERGEBNIS ---------------
JDBC             1. Ausführng: 612 ms
JDBC PS          1. Ausführng: 123 ms
```

  - Schutz vor SQL-Injections

Beispiel

Füge den Kunden ‘Müller KG’ als Geschäftskunden in Dortmund ein.

Note: Einfügen von NULL-Werten Einfügen von NULL-Werten erfolgt mit der Methode `setNull`:

```
setNull(int <parameterindex>, int <java.sql.Types.Typ>)
```

Eine Übersicht über die java.sql.Types-Konstanten für die SQL-Typen finden Sie in der Java-Dokumentation.

Beispiel

Der Kunder Andreas Neumann ändert sein Geschlecht und heißt jetzt wieder Agnes.

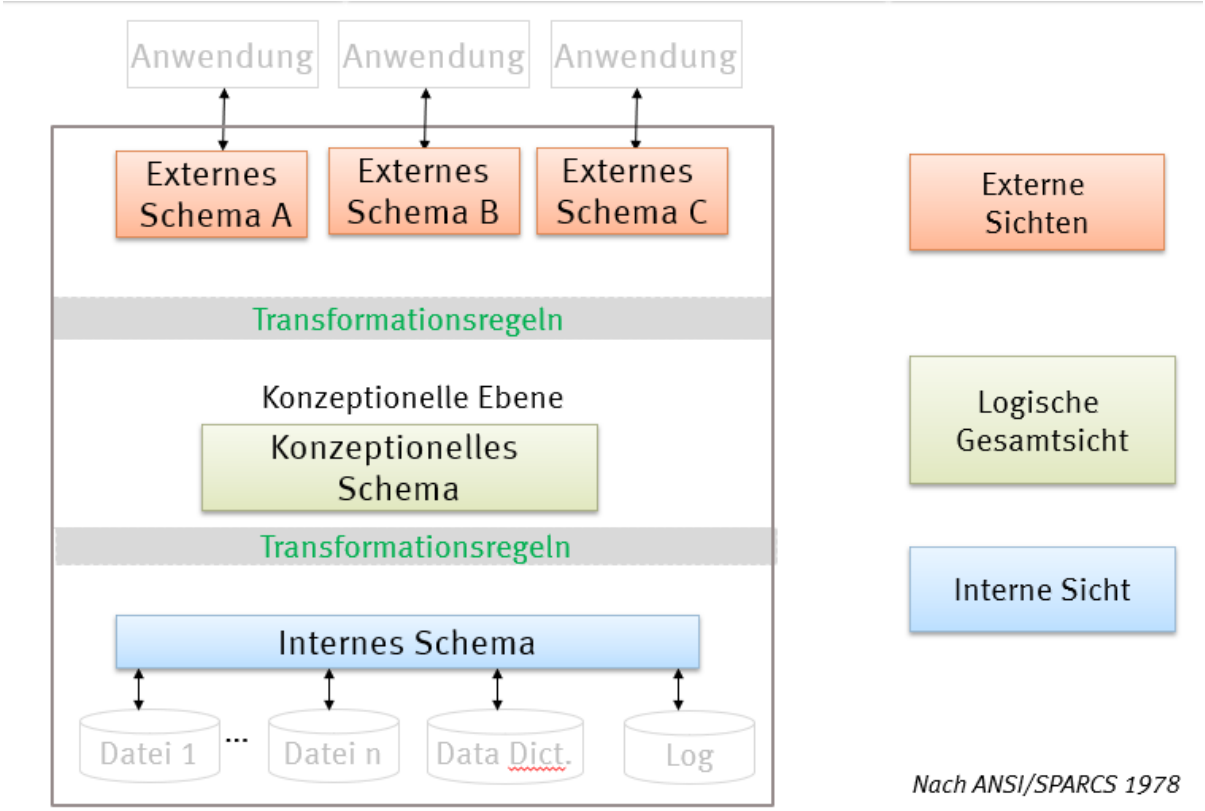
```

```

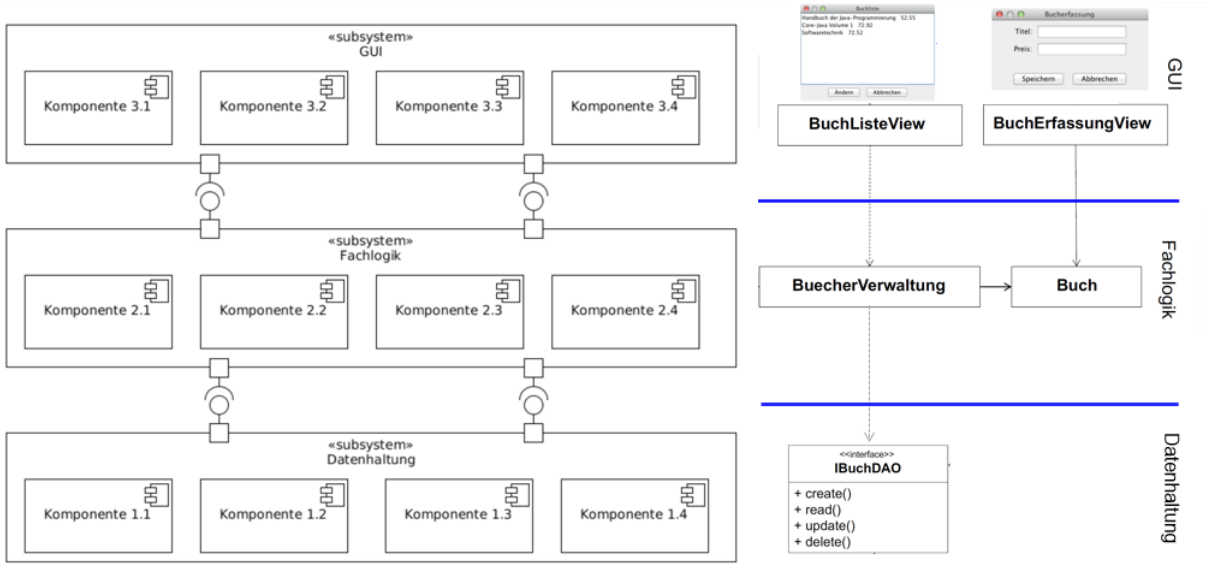
Drei-Schichten Architektur

Hierzu gibt es ein Video in ILIAS.

Der typische Aufbau einer Datenbank basiert auf der Drei-Schichten Architektur.



Diese Aufteilung hat sich bewährt, so dass die Drei-Schichten Architektur auch in der Softwareentwicklung verwendet wird.



Abbildungen aus: PK1 Prof. Wiesmann

Note: Vorteil der Drei-Schichten-Architektur Durch die Drei-Schichten-Architektur wird die Anwendung modularisiert. Damit ist sie besser wartbar, da Komponenten besser ausgetauscht und unabhängig voneinander geändert werden können. Bei der **strikten** Drei-Schichten Architektur können nur zwei benachbarte Schichten miteinander kommunizieren.

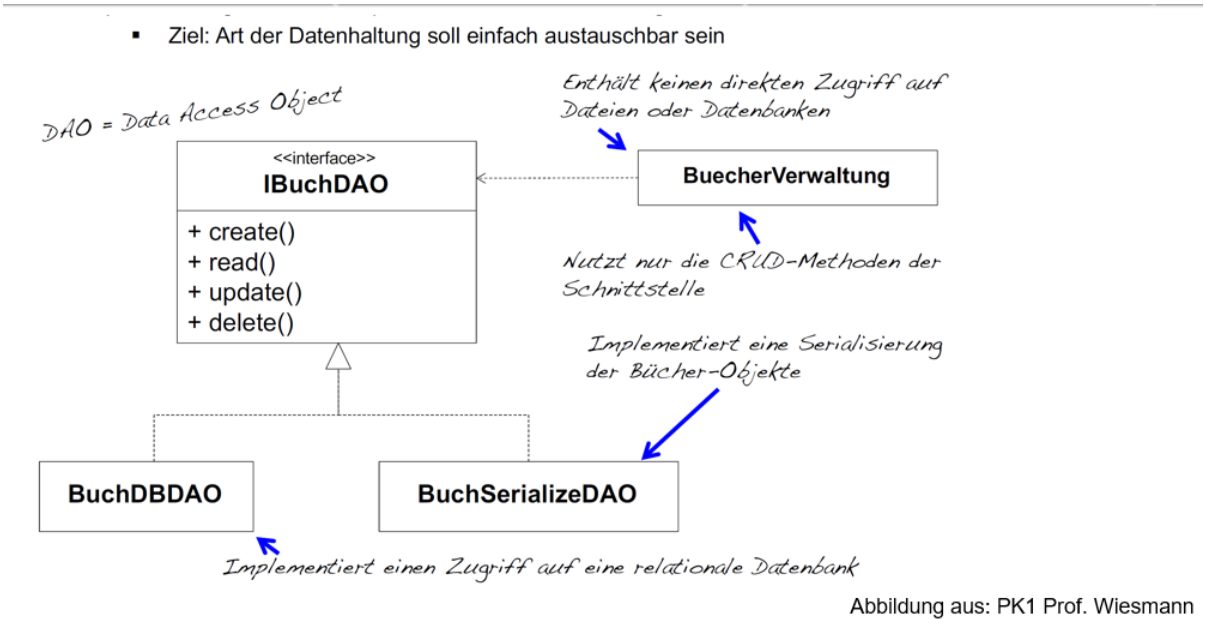


Abbildung aus: PK1 Prof. Wiesmann

Bespielprogramm

Beispielprogramm in ILIAS: eShop.zip
Erläuterungsvideo zum Beispielprogramm in ILIAS

Selbsttest

Die folgenden Fragen sollten Sie jetzt beantworten können:

- Wie wird mit JDBC eine Verbindung zur Datenbank aufgebaut?
- Was ist ein ResultSet?
- Wie können SQL-Injections verhindert werden?
- Wieso ist eine Drei-Schichten Architektur hilfreich, um die Datenbank vor unberechtigen Zugriff zu schützen?

Aufgabe 1

a) Wie kann man durch SQL-Injection herausfinden, wieviele Spalten die Methode `einenKundenAnzeigen()` ausliest?
b) Führen Sie die Methode `einenKundenAnzeigen()` aus, um alle Tabellen eines anderen Nutzers anzuzeigen.
c) Verhindern Sie durch ein geeignetes Umschreiben der Methode, dass diese Angriffe erfolgreich sind.

Selbsttestaufgabe 2

Was ist an dem Beispielprojekt (eShop) noch zu verbessern?

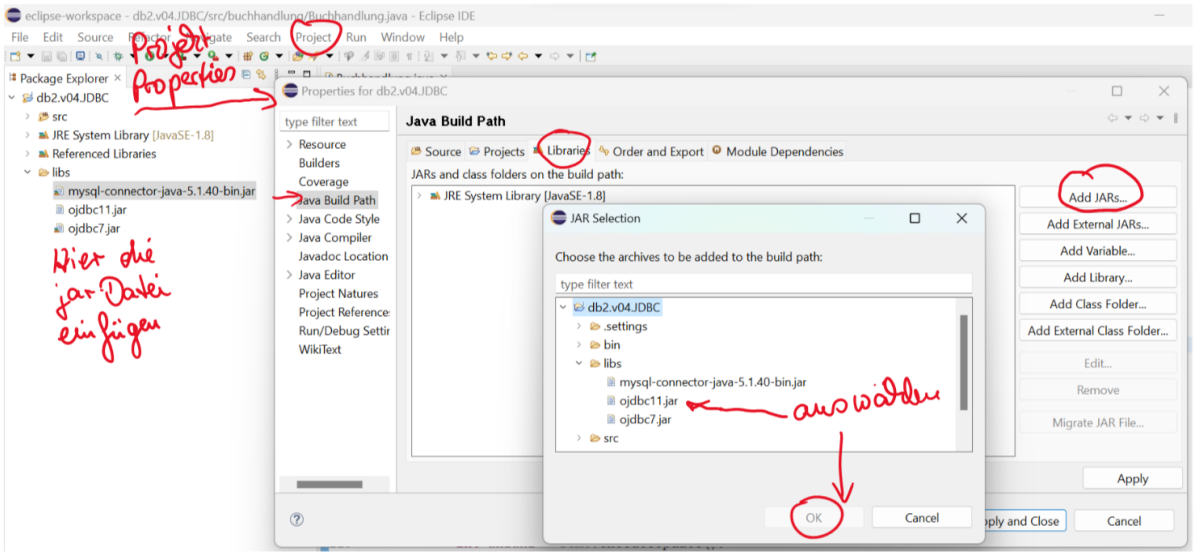
Anhang 1: Einbindung eines JDBC-Treibers

Der Oracle Thin JDBC Treiber `ojdbc11.jar` kann für aktuelle Java-Versionen verwendet werden. Hierbei handelt es sich um einen in Java implementierten Typ 4 JDBC-Treiber.

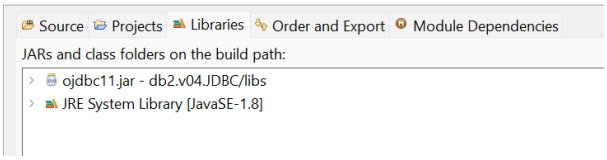
Einbinden des JDBC-Treibers in einem Java-Projekt (in eclipse)

Der zu der verwendeten Java-Version kompatiblen Treiber kann im ILIAS-Kurs oder von der Webseite des Herstellers Oracle herunter geladen werden. Alternativ kann auch das Maven-Repository verwendet werden.

Note: Einbinden des JDBC-Treibers



1. Einfügen des Treibers (jar-Datei) in das Dateisystem des Projekts, bspw. in einem Unterordner “libs”.
2. Öffnen der Projekteinstellungen (Project Properties), erreichbar über das Menü oder per rechtem Mausklick.
3. Navigieren zu “Java Buildpath” und darin den Tab “Libraries”.
4. “Add Jar” auswählen und im Projktverzeichnis die jar-Datei auswählen und mit “ok” bestätigen.
5. Danach sollten die JDBC-Treiberbibliothek im JavaBuild-Path auftauchen:



Einbinden des JDBC-Treibers in ein Jupyter-Notebook (Java-Kernel)

Das Einbinden eines JDBC-Treiber kann über das maven-Repository oder über eine vom Datenbankhersteller bereitgestellte Treiber-Library (jar-Datei) erfolgen.
Einen JDBC-Treiber von Oracle kann mit dem magischen Befehl %maven eingebunden werden:

```
%maven com.oracle.database.jdbc:ojdbc11:23.3.0.23.09
```

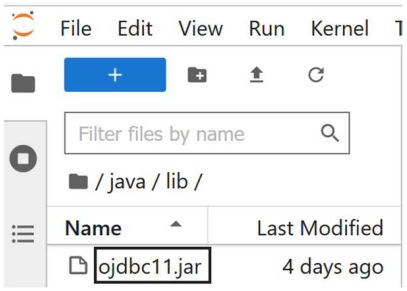
Hint: Erläuterung zur Nutzung des Maven-Repository Das **Maven**-Repository kann genutzt werden, um den Treiber einzubinden. Dort finden Sie entsprechende Versionsangaben zu den verfügbaren Treibern, beispielsweise:

```
<!-- https://mvnrepository.com/artifact/com.oracle.database.jdbc/ojdbc11 -->
<dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc11</artifactId>
    <version>23.3.0.23.09</version>
</dependency>
```

Die Maven-Koordinaten können erfasst werden mit dem magischen Befehl `%maven <groupId>:<artifactId>:<version>`. Also beispielsweise:
`%maven com.oracle.database.jdbc:ojdbc11:23.3.0.23.09`

Einbindung von lokalen Treiberbibliotheken (jar-Dateien)

Hint: Lokale jar-Datei Einfügen der JDBC-Treiber-Library (jar-Datei) in das **lokale** Verzeichnis des Notebooks



Anschließend kann der Treiber im Java-Notebook eingebunden werden mit dem magischen Befehl `%jars <pfad+dateiname>`, also beispielsweise `%jars lib/ojdbc11.jar`.

Anhang 2: Exceptionhandling in Java

Die Klasse `Exception` Das Auftreten von Fehlern wird in Java durch Objekte der Klasse `Exception` signalisiert. Eine `Exception` kapselt alle Informationen zu dem aufgetretenen Fehler, bspw. den Text der Fehlermeldung.
Die `Exception` **muss** *entweder* in der Methode, in der sie auftritt, gefangen (`catch`) und behandelt werden *oder* an die aufrufende Methode weitergereicht zur Fehlerbehandlung weitergereicht (`throws`) werden.

Fallbeispiel

- Es soll eine Kundennummer über die Konsole eingelesen und als Zahl ausgegeben werden.
- Auftretende Fehlermeldungen sollen für Nutzer verständlich sein.

JAVA Exception Handling

- Eine Exception (Fehlermeldungen) kann direkt in der Java-Methode ausgewertet werden. Dazu wird der Codeabschnitt in einen try-catch-Block eingebettet.
- Zum Fangen der Exception werden die vorhandenen catch-Blöcke nacheinander abgearbeitet, bis die Exception gefangen wurde.

```
try{
   // Code, der eine Fehlermeldung auslösen kann
   ...
}catch(e NumberFormatException){
   // Fangen und Verarbeitung von NumberFormatExceptions (Spezialisierung der Klasse
 ↪Exception)
}catch(e Exception){
   // Fange die allgemeine Fehlermeldung und gebe diese auf der Konsole aus
   e.printMessage();    // Ausgeben des Fehlertextes
   e.printStackTrace(); // Ausgabe der aufgerufenen Methoden mit Angabe der Codezeile
}
```

Werfen und Weitergabe von Exceptions

- Soll die Fehlerverarbeitung nicht in der aktuellen Methode durchgeführt werden, muss die Methodensignatur um die Klausel `throws Exception` ergänzt werden.
- Dann muss sich die aufrufende Methode um die Behandlung des Fehlers kümmern. ``
  Eine Exception wird geworfen durch den Befehl `throw`.

```
void kundenMenu() throws Exception{
  ...
  try{
    // Code, bei dem eine Exception auftreten kann
  }catch(e Exception){
    // Umschreiben der Fehlermeldung, so dass dies von Nutzern verstanden wird
    throw new Exception("Nutzerfreundliche Fehlermeldung");
  }
}
```

Erläuterung hierzu im Video: JDBC-Datentypen

SQLException

SQL-Fehlermeldungen sind mit einem herstellerabhängigen Error-Code versehen. Zusätzlich werden Fehlermeldungen der Datenbank nach standardisierten Fehlerklassen als SQLSTATE zurück gegeben. SQLSTATE ist ein CHAR(5)-Variable. Die ersten zwei Byte bezeichnen die Fehlerklasse, die nächsten drei die jeweilige Unterklasse.

```
try{
     // CODE mit JDBC-Befehlen
}catch(e SQLException){
    System.err.println("Message: " +e.getMessage());
    System.err.println("SQLState: "+e.getSQLState());
    System.err.println("Error Code: " +e.getErrorCode());
}finally{
    // Der finally-Block wird immer durchlaufen
    // Hier können Ressourcen geschlossen werden
}
```

SQLWarning

Warnungen werden durch die Klasse `SQLWarning` gekapselt, die entsprechend den SQLExceptions behandelt werden können.

```
try{
    // Code, der SQLWarning werfen kann
}catch(e SQLWarning){
    System.err.println("Message: " +e.getMessage());
    System.err.println("SQLState: "+e.getSQLState());
    System.err.println("Error Code: " +e.getErrorCode());
}
```