



UNIT 0x0E
ODBC/JDBC

Motivation

Ziel

Bislang wurden SQL-Befehle per IDE, z.B. DataGrip, im Terminal oder Web-basiert abgesetzt. Wie kommuniziert aber die IDE oder das Terminal mit dem DBMS?

- Wir wollen Datenbankbefehle, z.B. SQL-Kommandos, programmatisch absetzen
- und auch die Ergebnisse verarbeiten.

Ansätze/Ideen

- Einbettung von SQL in Code
- Zugriff auf das DBMS über spezielle Bibliotheken/APIs (Open Database Connectivity ODBC bzw. Java Database Connectivity JDBC)

Q&A

- Wie sieht die Datenübertragung aus?
- Welche Systeme werden unterstützt?
- Welche Sprachen werden benötigt?

Embedded SQL

IDEE

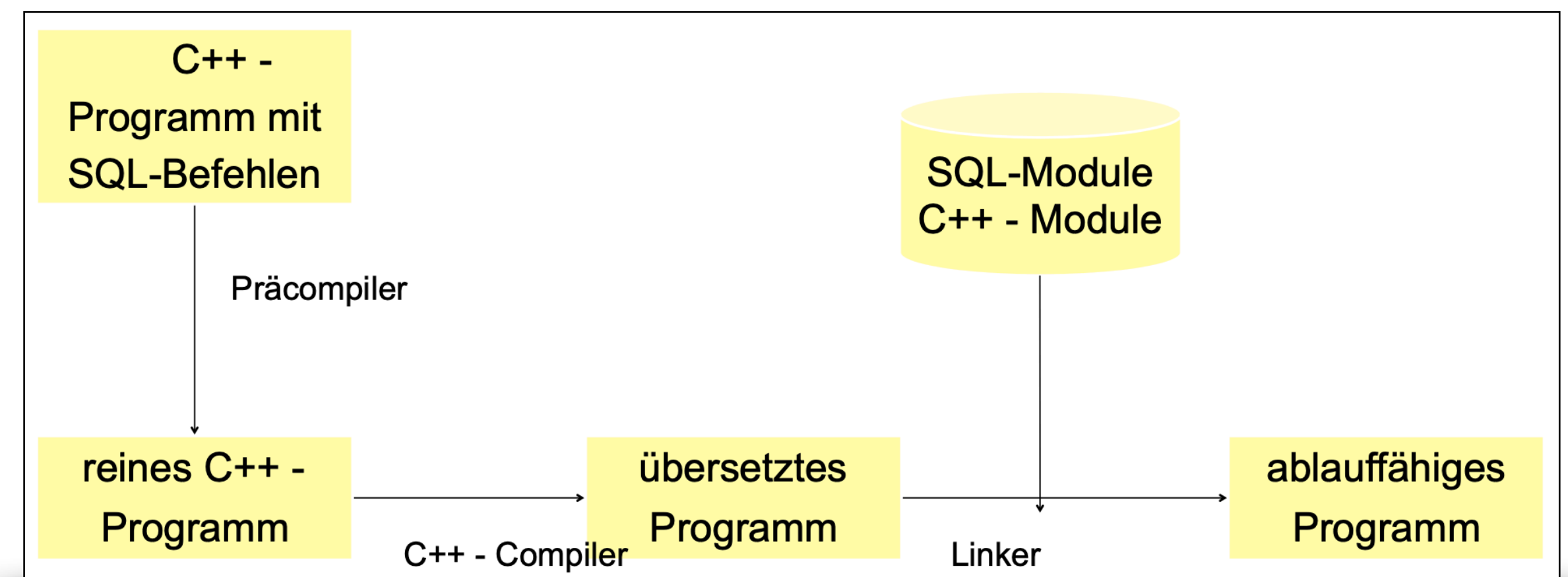
Einbettung von SQL-Anweisungen als eigenständiges Sprachkonstrukt in eine Programmiersprache

- Abgrenzung der SQL-Elemente durch spez. Präfix

```
EXEC SQL <sql-statement> <terminator>  
#sql { <sql-statement> };
```

Nicht-Java, oder
Java

- Umsetzung der Kommandos durch
 - spezielle Tools zur Compilezeit (Präcompiler) und/oder
 - Bibliotheken zur Laufzeit (Linker).



Embedded SQL

Beispiel

- Variablen aus dem Programm werden verwendet.
- Typische Aufgaben, als SQL-Befehl in das Programm eingebettet (hier C):
 - Daten abfragen (SELECT),
 - Daten anlegen (INSERT),
 - Verbindung aufbauen (CONNECT),
 - Fehlerbehandlung (SQLERROR).

```
/* connect to Oracle */  
EXEC SQL CONNECT :userid;  
printf("Connected.\n");
```

```
/* handle errors */  
EXEC SQL WHENEVER SQLERROR do sql_error("Oracle error");
```

```
EXEC SQL SELECT ename, job, sal + 2000  
INTO :emp_name, :job_title, :salary  
FROM emp  
WHERE empno = :emp_number;
```

```
int    emp_number;  
char    temp[20];  
VARCHAR emp_name[20];  
  
/* get values for input host variables */  
printf("Employee number? ");  
gets(temp);  
emp_number = atoi(temp);  
printf("Employee name? ");  
gets(emp_name.arr);  
emp_name.len = strlen(emp_name.arr);  
  
EXEC SQL INSERT INTO EMP (EMPNO, ENAME)  
VALUES (:emp_number, :emp_name);
```

Embedded SQL

Anmerkungen

- Nebenstehend zwei Ausschnitte aus aktuellen Dokumentationen Oracle und IBM DB2, d.h. hier wird dieser Ansatz durchaus gepflegt.
- Erfordert weiteres Tool (Präcompiler) in der Entwicklungspipeline inkl. aller (DBMS-)Abhängigkeiten und Fehlerquellen.
- Sprachen C, Fortran, Cobol, REXX häufig in älteren Systemen verwendet (ohne Wertung).
- Weitere Details hierzu in den Docs.

+ **Changes in This Release for Pro*C/C++ Programmer's Guide**

– **Part I Introduction and Concepts**

- + **1 Introduction**
- + **2 Precompiler Concepts**
- + **3 Database Concepts**
- + **4 Datatypes and Host Variables**
- + **5 Advanced Topics**
- + **6 Embedded SQL**
- + **7 Embedded PL/SQL**
- + **8 Host Arrays**
- + **9 Handling Runtime Errors**
- + **10 Precompiler Options**
- + **11 Multithreaded Applications**

Oracle Doc.

The Db2 development environment

ADO.NET

OLE DB

Snapshot
CLI and ODBC

DB2CI

Embedded SQL

Embedding SQL statements in a host language

Embedded SQL statements in C and C++ applications

Embedded SQL statements in FORTRAN applications

Embedded SQL statements in COBOL applications

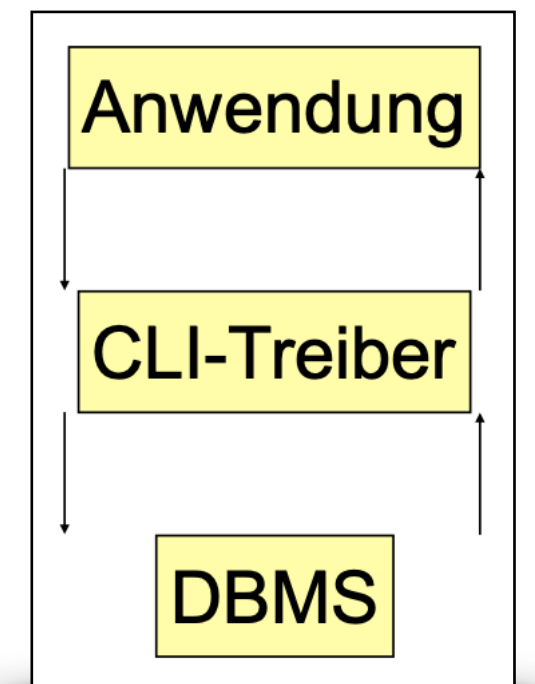
Embedded SQL statements in REXX applications

IBM DB2 Doc.

Call Level Interface (CLI)

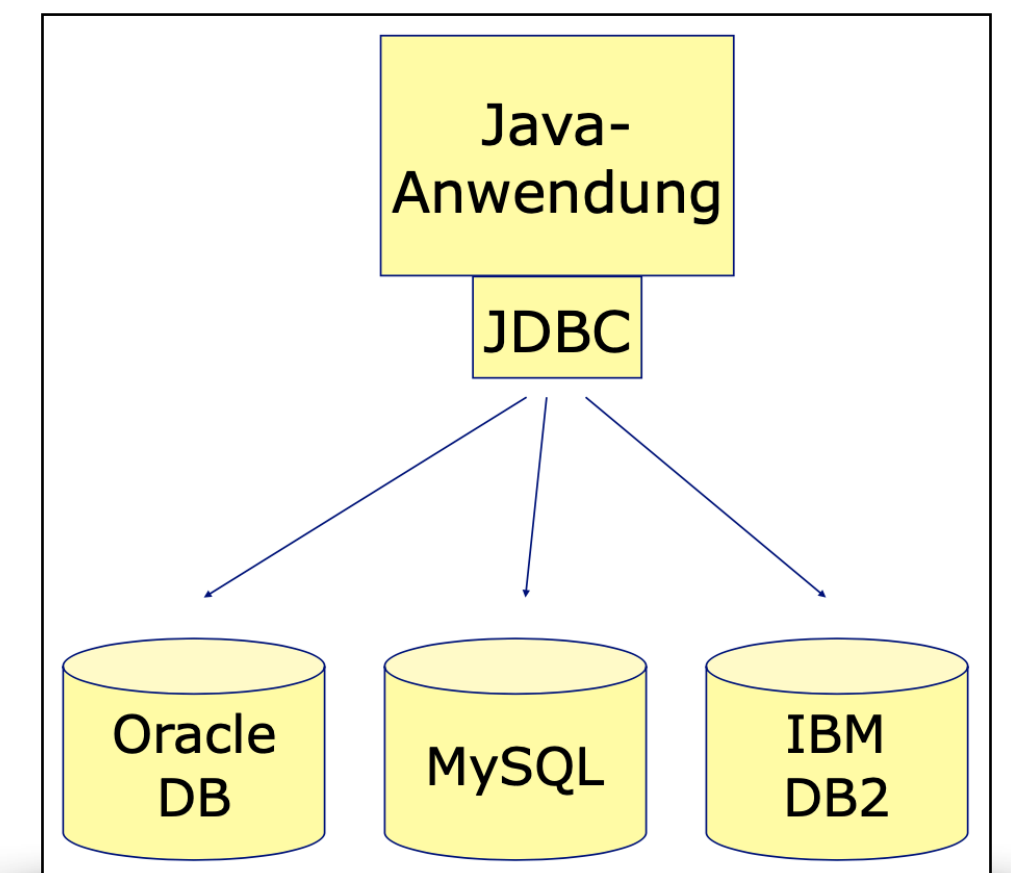
Idee

- Anwendungen benutzen ein sprachkonformes API: Call Level Interface (CLI).
- SQL-Anweisungen werden in CLI-Calls als Zeichenkette eingebettet.



Vorteile

- Integration in Programmiersprache, Bibliotheksnutzung, kein Präcompiler.
- Offene Schnittstelle (DBMS-unabhängig, d.h. durch Austausch der Treiber können DBMS verschiedener Hersteller angesprochen werden), z.B. Open DataBase Connectivity (ODBC), Java DataBase Connectivity (JDBC).
- Weit verbreitet in der Praxis.



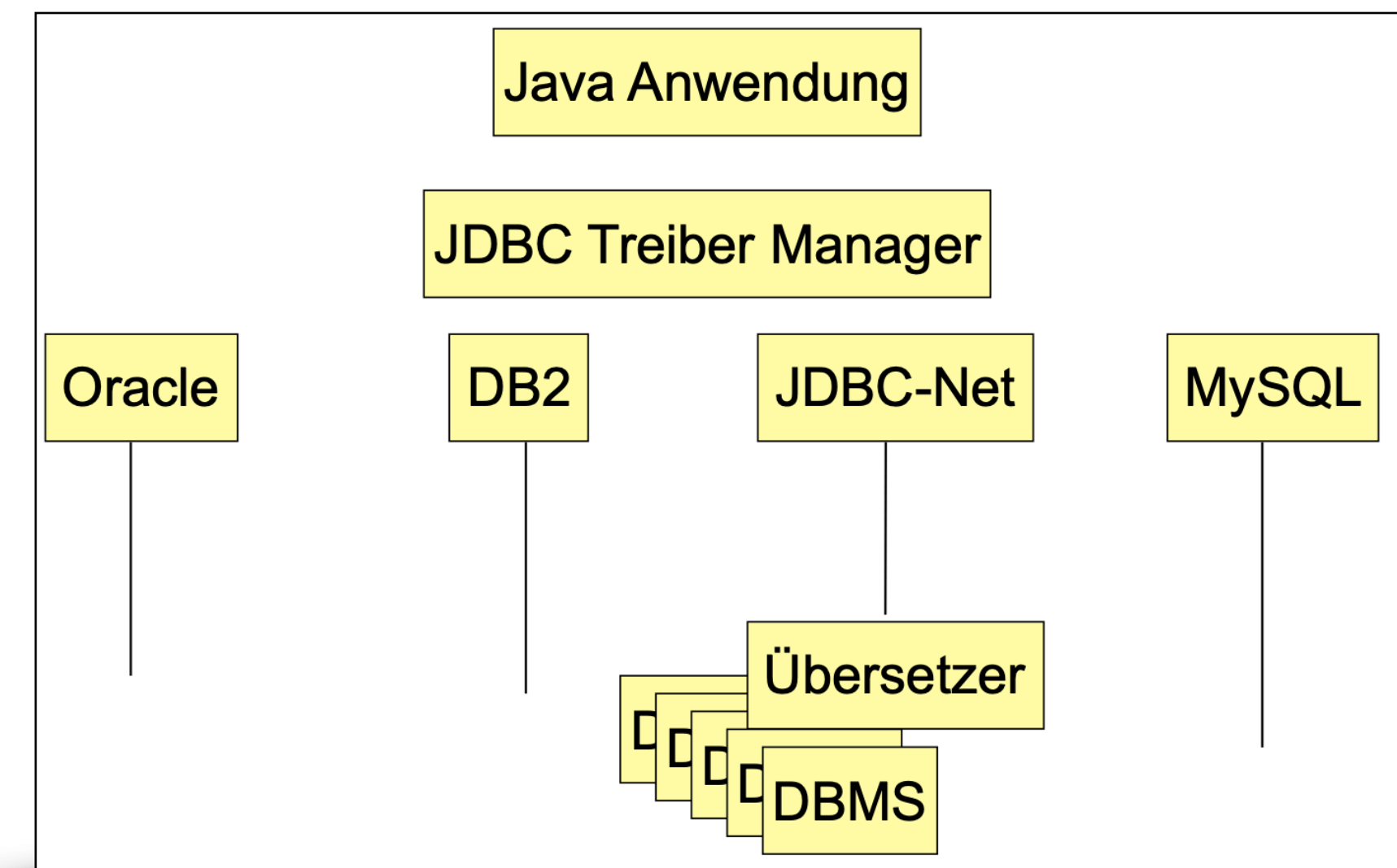
Nachteile

- Ggf. schlechtere Performance wg. dynamischer SQL-Interpretation (zur Laufzeit).

Java DataBase Connectivity (JDBC)

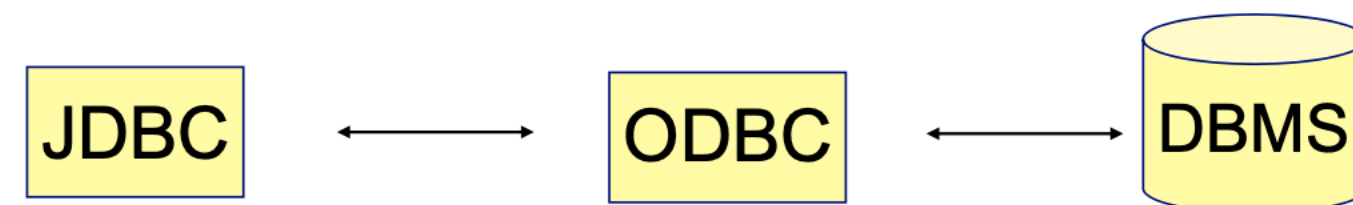
Hintergrund

- Allgemeine Datenbank-Zugriffsschnittstelle für SQL.
- Übertragung der JAVA-Portabilität auf Datenbanken, d.h. JDBC ermöglicht einen SQL-basierten, plattform-unabhängigen Zugriff auf verschiedene relationale DBMS.
- Teil von Java 5 (Package java.sql).
- Basiert auf X/Open SQL CLI (wie auch ODBC).
- Übersichtlicher und einfacher benutzbar als ODBC.
- Unterstützt ANSI SQL92- Basisfunktionalität.
- Treiber sind vom DBMS-Hersteller als Bibliothek (z.B. jar-File) oder Artefakt (z.B. via maven mvnRepository) zu bekommen.

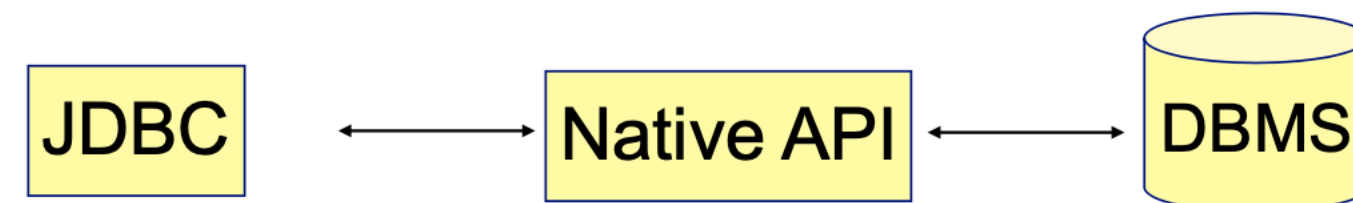


JDBC-Treiber-Implementierung

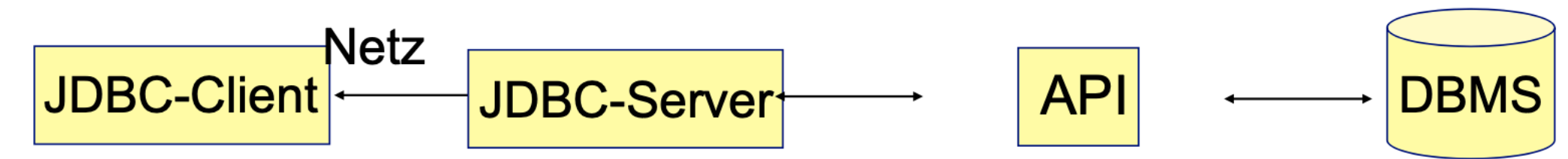
- **JDBC-ODBC-Bridge** (Typ 1)
 - Nutzen vorhandener ODBC-Treiber



- **Native-API-Treiber** (Typ 2)
 - Basiert auf Bibliotheken des DBMS-Herstellers



- **JDBC-Net-Treiber** (Typ 3)
 - Aufteilung Treiber in JDBC-Client und JDBC-Server



- **Native-Protokoll-Treiber** (Typ 4)
- Direkte Übersetzung der JDBC-Aufrufe in DBS-Aufrufe
 - Client direkt mit Datenbankserver verbunden
 - Nachteilig sind die i.A. proprietären DBS-Protokolle

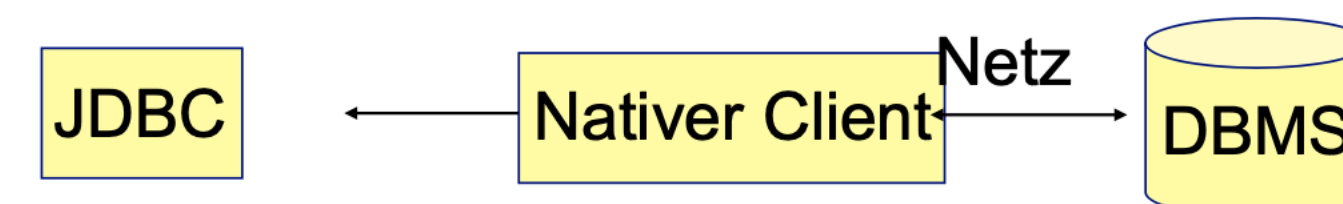


Abbildung SQL- auf Java-Datentypen

SQL-Datentyp	Java-Datentyp
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	<i>java.sql.BigDecimal</i>
DECIMAL	<i>java.sql.BigDecimal</i>
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONVARBINARY	byte[]
DATE	<i>java.sql.Date</i>
TIME	<i>java.sql.Time</i>
TIMESTAMP	<i>java.sql.Timestamp</i>

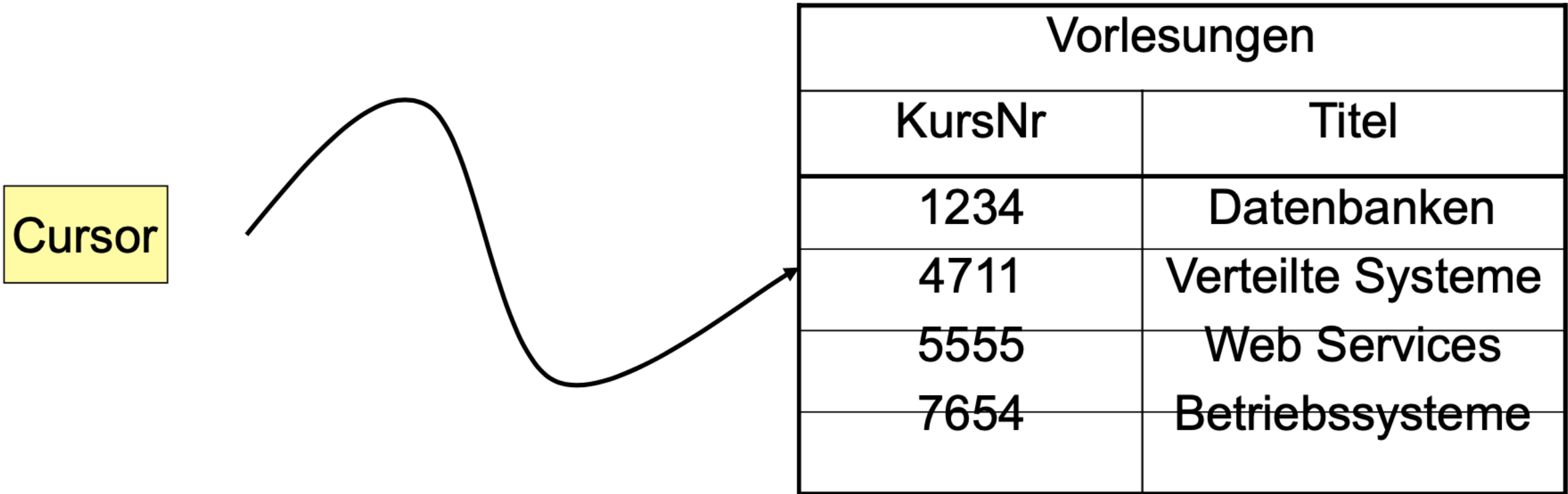
Interaktion mit dem DBMS

Prinzipielle Funktionsweise

- Datenbanktreiber laden (implizit oder explizit).
- Aufbau einer Verbindung (Connection) zur Datenbank.
- Erstellen und Ausführen von SQL-Anweisungen sowie Verarbeitung der Ergebnisse im Kontext der Verbindung
 - Cursor-Konzept notwendig – folgt.
- Alle Verbindungen (Connection, Statement, ResultSet) immer schließen (close).
- Programmbeispiele folgen ebenfalls.

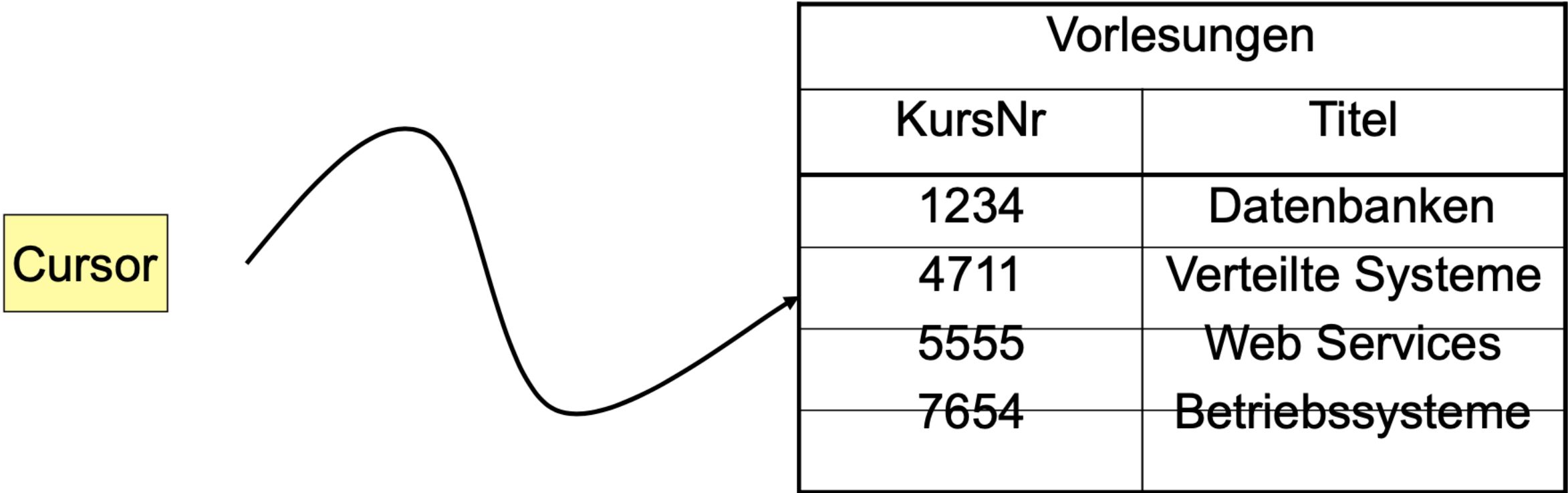
Einschub Cursor-Konzept

- Problem:**
 - Fehlanpassung („impedance mismatch“) zwischen Datenmodell des DBMS und der Anwendung, da normale Programmiersprachen einen Typ „Relation“ nicht anbieten
- Aufgabe:**
 - Abbildung von Tupelmengen auf die Variablen der Programmiersprache
- Lösung:**
 - Cursor-Konzept; sukzessives traversieren der Tupel einer Relation mittels des Iterator-Entwurfsmuster



Einschub Cursor-Konzept

- Durch das Iterator-Muster kann man sequentiell auf die Elemente einer Datenstruktur bzw. eines zusammengesetzten Objekts zugreifen, ohne dabei ihre interne Struktur offen zu legen



Beispiel Java/Kotlin

Projekt

- Mix aus Java und Kotlin.
- Projekt: db_connector_XXX

Ablauf

- Verbindung aufbauen,
- alle Kunden ausgeben,
- Kunde einfügen, alle ausgeben,
- Kunde löschen, alle ausgeben,
- Metadaten Tabelle
Kunde ausgeben,
- Verbindung schliessen.

```
fun main() {  
    try {  
        connectToMatseMhistOf(  
            user = "root",  
            password = "root",  
            port = 3310  
        ).use { connection -> connection.apply { this: Connection  
            println("step 1: select all in Kunde")  
            showAllCustomers()  
  
            val kunde = "FH Aachen"  
            println("step 2: insert kunde '$kunde')  
            insertCustomer(name = kunde, discount = 1.0)  
            showAllCustomers()  
  
            println("step 3: delete kunde '$kunde')  
            deleteCustomer(name = kunde)  
            showAllCustomers()  
  
            println("step 4: metadata table 'kunde')  
            showMetaDataCustomers()  
        } }  
    } catch (e: SQLException) {  
        println("SQL-Error: $e")  
    } catch (e: Exception) {  
        println("Error: $e")  
    }  
}
```


Beispiel Java/Kotlin

Verbindung aufbauen (Kotlin)

```
fun connectToMatseMhistOf(user: String, password: String, port: Int) =  
    DriverManager.getConnection(  
        "jdbc:mysql://localhost:$port/matse_mhist?useSSL=false&allowPublicKeyRetrieval=true",  
        user,  
        password  
    )!!
```

- Aufruf über DriverManager
- Zusammenstellung eines Connection-Strings; genauer Aufbau variiert je Verbindungstyp und DBMS (hier jdbc:mysql), URL, Port (hier localhost, port) und Parametern (hier useSSL etc.).
- Bei Erfolg ist Rückgabe eine Connection, sonst wird eine SQLException geworfen.
- Sieht in Java (fast) genau so aus.

Beispiel Java/Kotlin

Alle Kunden ausgeben (Kotlin)

- SQL-Befehl zusammenstellen (sql),
- Statement aus SQL-Befehl erstellen und ausführen,
- über Ergebnismenge/ResultSet nacheinander iterieren, die Werte der Spalten holen und ausgeben,
- das Statement wird implizit geschlossen (wg. use).
- In Java wird üblicherweise explizit ein Statement- und ein ResultSet-Objekt angelegt, evtl. mittels try-with-resources.

```
fun selectAllInKunde(connection: Connection) {  
    val sql = "SELECT id,name,rabatt_prozent FROM kunde"  
    connection.createStatement().apply { //statement ->  
        executeQuery(sql).use { resultSet ->  
            while (resultSet.next()) {  
                val id = resultSet.getInt("id")  
                val name = resultSet.getString("name")  
                val rabatt = resultSet.getDouble("rabatt_prozent")  
                println(" id:$id, name:'$name', rabatt:$rabatt")  
            }  
        }  
    }  
}
```

Java DataBase Connectivity (JDBC)

Einschub Statements

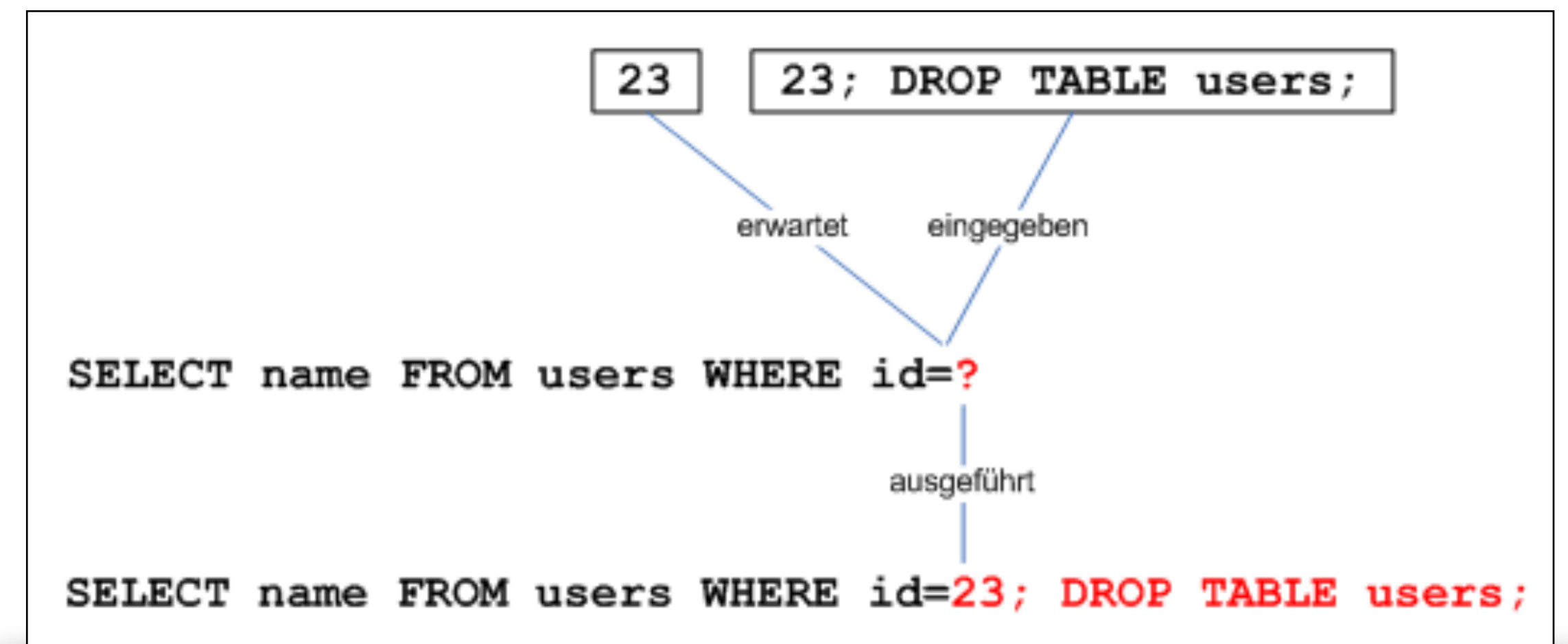
JDBC unterstützt drei Statement-Arten:

- **Statement:** Statische SQL-Anweisung, die von JDBC nicht verändert, sondern einfach weiter gereicht wird.
- **PreparedStatement:** Erlaubt die Formulierung von vorbereiteten SQL-Anweisungen, bei denen später bestimmte Attribute über Parameter belegt werden. Hierdurch können häufig verwendete, nur leicht variierende SQL- Anweisungen bereits vom DBMS übersetzt werden. Eine spätere Ausführung findet zumeist auf der Basis des schon erstellten Zugriffsplan statt. Gleichzeitig wird so ein Angriff via SQL-Injection verhindert.
- **CallableStatement:** Erlaubt das Ausführen einer in der Datenbank hinterlegten Prozedur (Stored Procedure) – hier nicht weiter relevant.

Java DataBase Connectivity (JDBC)

Einschub SQL-Injection

- Angriffsart, bei der durch mangelnde Validierung der Eingabedaten das Einschleusen fremden SQL-Codes in eine Datenbankabfrage möglich ist.
- Wichtiges Problem für Web-basierte Anwendungen, die SQL-Statements generieren.
- Erfolgt durch Modifizierung von übergebenden Parametern.
- Eine dynamisch generierte SQL-Query wird mit fremden Code „infiziert“.
Basiert auf der Idee, das eigentliche Kommando abubrechen und dafür als Parameter ein vollständiges weiteres SQL-Kommando abzusetzen.



Java DataBase Connectivity (JDBC)

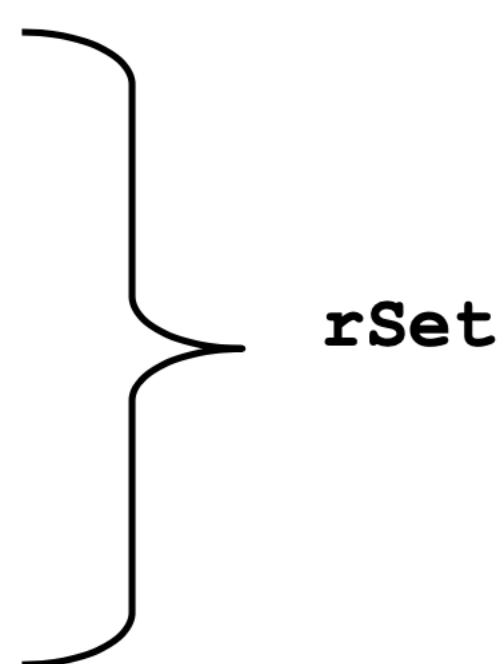
ResultSet

- Nehmen Ergebnisse auf bzw. stellen Methoden zur Verfügung, mit denen diese abgerufen werden sowie Methoden zur Positionierung des Cursors.
- Was beim Positionieren des Cursors möglich ist, hängt am Statements-Objekt, der Anfrage selber und dem DBMS. Hier gibt es verschiedene Typen, z.B.
 - `ResultSet.TYPE_FORWARD_ONLY` (nur vorwärts),
 - `ResultSet.TYPE_SCROLL_INSENSITIVE` (alle Richtungen)
 - `ResultSet.CONCUR_UPDATABLE` (modifizierbar).
- Seit JDBC 2.0 können die in ResultSets gespeicherten Daten auch modifiziert werden.
- Achtung: Bevor Werte verarbeitet werden können, muss die next-Methode aufgerufen werden. Ist der Rückgabewert `TRUE`, so liegt ein Ergebnis vor.

Java DataBase Connectivity (JDBC)

ResultSets

Vorlesungen	
KursNr	Titel
1234	Datenbanken
4711	Verteilte Systeme
5555	Web Services
7654	Betriebssysteme



- Schließlich kann das **ResultSet**-Objekt mittels des Cursor-Prinzips verarbeitet werden

```
while (resultSet.next()) {  
    val id = resultSet.getInt("id")  
    val name = resultSet.getString("name")  
    val rabatt = resultSet.getDouble("rabatt_prozent")  
    println(" id:$id, name:'$name', rabatt:$rabatt")  
}
```

Beispiel Java/Kotlin

Kunde einfügen (Java)

```
static void insertCustomer(Connection connection, String name, Double discount) throws  
    // hier ist das PreparedStatement, man achte auf den Zusatz '(?,?)'  
    // - auf die id wird verzichtet, da die Tabelle Kunde autoincrement id besitzt  
    String sql = "INSERT INTO kunde (`name`, `rabatt_prozent`) VALUES (?,?);";  
    PreparedStatement statement = connection.prepareStatement(sql);  
    statement.setString(parameterIndex: 1, name);  
    statement.setDouble(parameterIndex: 2, discount);  
    statement.execute();  
}
```

- SQL-Befehl als PreparedStatement zusammenstellen. Hierbei werden die '?' durch explizit anzugebende Werte ersetzt.

Beispiel Java/Kotlin

Kunde löschen (Java)

```
public static void deleteCustomer(Connection connection, String name) throws
// hier ist das PreparedStatement, in einem try-with-resources
String sql = "DELETE FROM kunde WHERE `name`=?";
try (PreparedStatement statement = connection.prepareStatement(sql)) {
    statement.setString(parameterIndex: 1, name);
    statement.execute();
}
}
```

- Wie zuvor SQL-Befehl als PreparedStatement zusammenstellen. Hierbei werden wieder die '?' durch explizit anzugebende Werte ersetzt.

Beispiel Java/Kotlin

Metadaten Tabelle Kunde ausgeben (Kotlin)

```
fun showMetaDataCustomers(connection: Connection) {  
    connection.metaData.getColumns(null, null, "kunde", "%").use { set ->  
        while (set.next()) {  
            val column = set.getString("COLUMN_NAME")  
            val dataType = set.getString("DATA_TYPE")  
            println("    column:$column, type-enum:$dataType")  
        }  
    }  
}
```

- Wie beim select wird hier das ResultSet ausgegeben. Es enthält zeilenweise die Spalteninformationen, z.B. den Namen und den Datentyp.
- Auch hier wird das ResultSet wieder geschlossen.

Beispiel Java/Kotlin

Meanwhile in DataGrip

- Nach dem Einfügen war der neue Datensatz in der Tabelle Kunde auch in einer externen Sicht auf die Tabelle zu sehen (DataGrip).

id	name	rabatt_prozent
2	Bayer	1.00
26	BMW	1.00
27	Commerzbank	0.00
28	Daimler	1.00
29	Deutsche Bank	0.00
30	Sparkasse	10.00
31	Börse	0.00
32	E.ON	1.00
33	Infinion	2.00
34	Lufthansa	2.00
35	RWE	1.00
36	SAP	2.00

id	name	rabatt_prozent
2	Bayer	1.00
26	BMW	1.00
27	Commerzbank	0.00
28	Daimler	1.00
29	Deutsche Bank	0.00
30	Sparkasse	10.00
31	Börse	0.00
32	E.ON	1.00
33	Infinion	2.00
34	Lufthansa	2.00
35	RWE	1.00
36	SAP	2.00
51	FH Aachen	1.00

matse_mhist

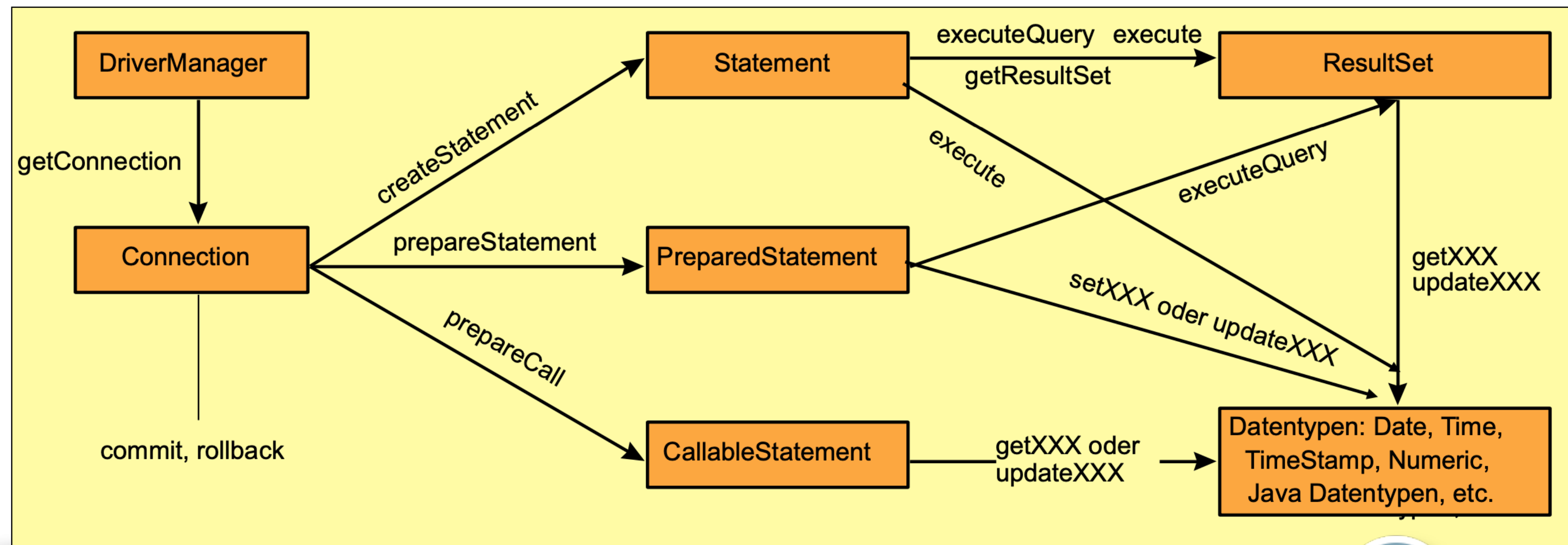
Java DataBase Connectivity (JDBC)

Ergänzung

- Zur Realisierung von Transaktionen gibt es Befehle der Connection zum commit und rollback. Standardmäßig wird ein Auto-Commit bei jedem update durchgeführt. Dies gilt auch für Änderungen mittels ResultSets.
- Die Fehlerbehandlung wird in JDBC ausschließlich über Exceptions, genauer SQLExceptions, abgewickelt.
- Außer den Exceptions gibt es noch die Möglichkeit, Warnungen abzufangen. Diese stoppen nicht die Ausführung einer Anweisung, sie alarmieren nur den Benutzer, dass die Abwicklung nicht wie geplant ablaufen könnte. Sie werden von Connections, Statements und ResultSets über getWarnings bereitgestellt.

Java DataBase Connectivity (JDBC)

Zusammenfassung



Das kann hier nur ein grober Überblick über grundsätzliche Zusammenhänge sein. Details entnehme man bitte der Dokumentation!