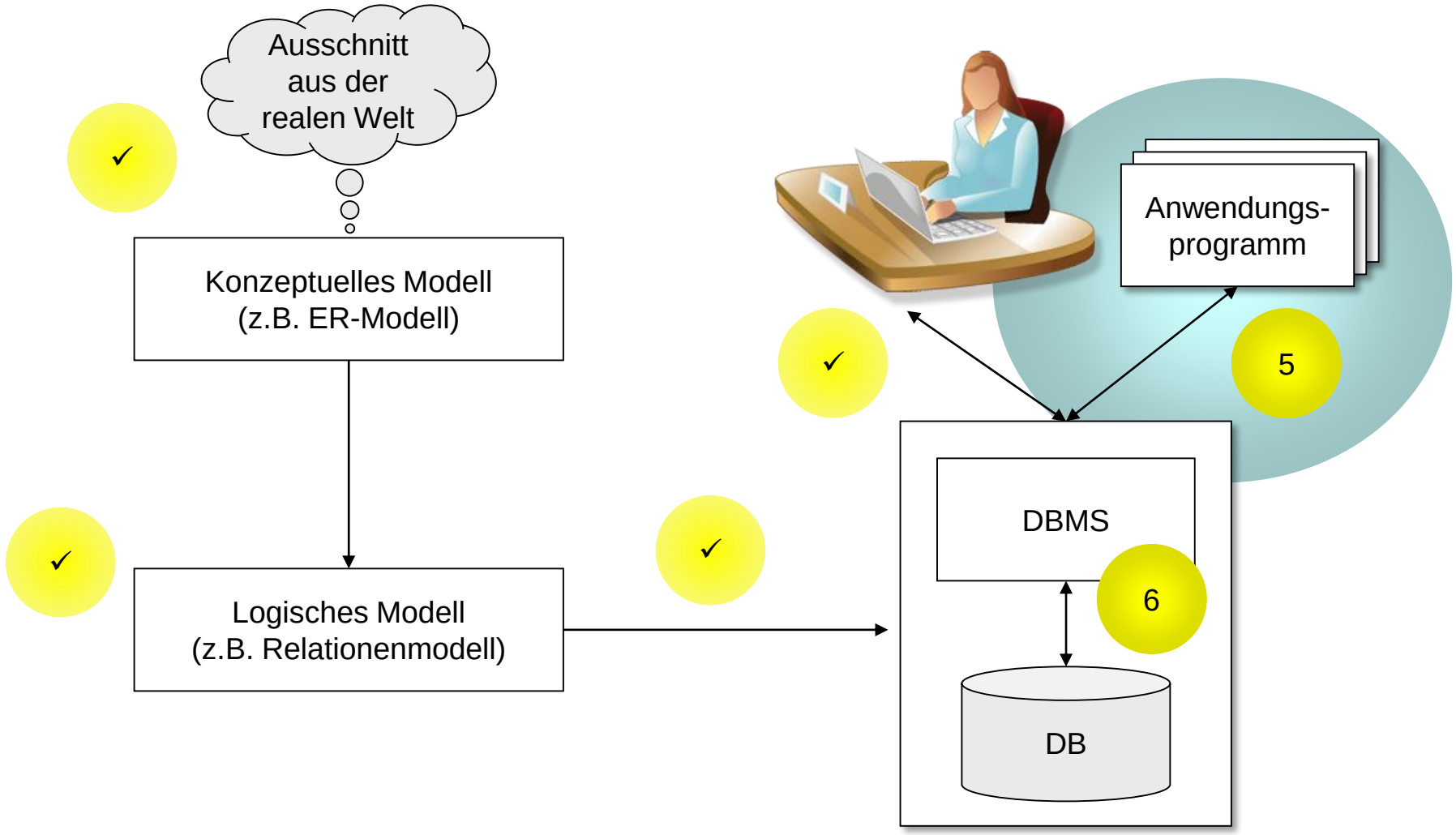
Datenbanken 1

Kapitel 5:

– Datenbank-Anwendungsprogrammierung –

```
create or replace trigger keinePreissenkung
before update of ProdPreis on Produkt
for each row
when (:old.ProdPreis is not null)
begin
    if :new.ProdPreis < :old.ProdPreis then
            :new.ProdPreis := :old.ProdPreis;
        dbms_output.put_line ('Preissenkung nicht erlaubt!');
    end if;
end;
```

Vorlesung Datenbanken 1



Datenbank-Anwendungsprogrammierung

Motivation

- Bisher: DDL- und DML-Befehle, welche ad hoc ausgeführt werden können.

- Aber: In der Realität werden insbesondere DML-Zugriffe meistens aus Anwendungsprogrammen heraus benötigt.

Datenbank-Anwendungsprogrammierung

Inhalt des Kapitels

- Grundprinzipien
- SQL-Zugriff über APIs (u.a. JDBC)
- Einbettung von SQL
- Prozedurale SQL-Erweiterungen
- Integrität und Trigger

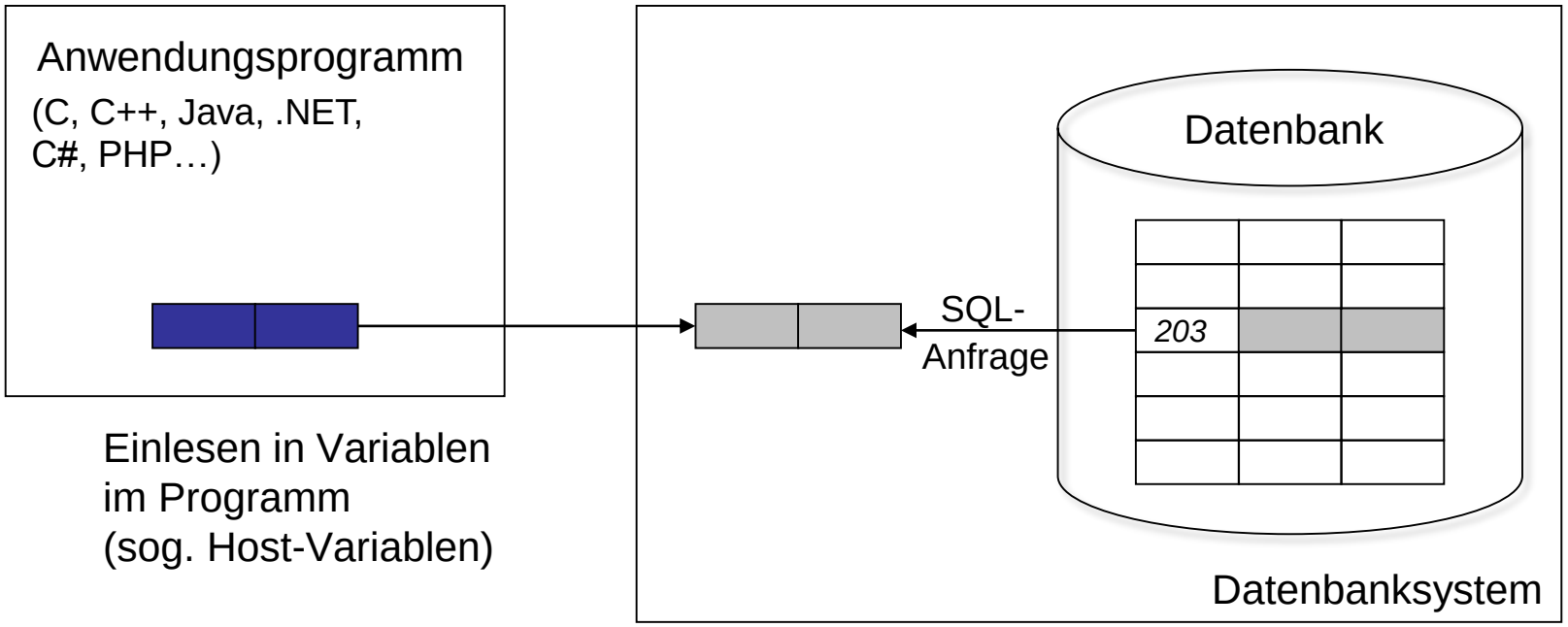
Lernziele

- Kennen und Bewerten verschiedener Varianten der Anwendungsprogrammierung für Datenbanken
- Anwenden der prozeduralen SQL-Erweiterungen
- Umsetzung von Integritätsbedingungen mit Triggern

Grundprinzipien

- **Wichtige Unterscheidung:**
  - Fall 1: Ergebnis der Anfrage **höchstens 1 Tupel**
  - Fall 2: Ergebnis der Anfrage **mehrere Tupel**

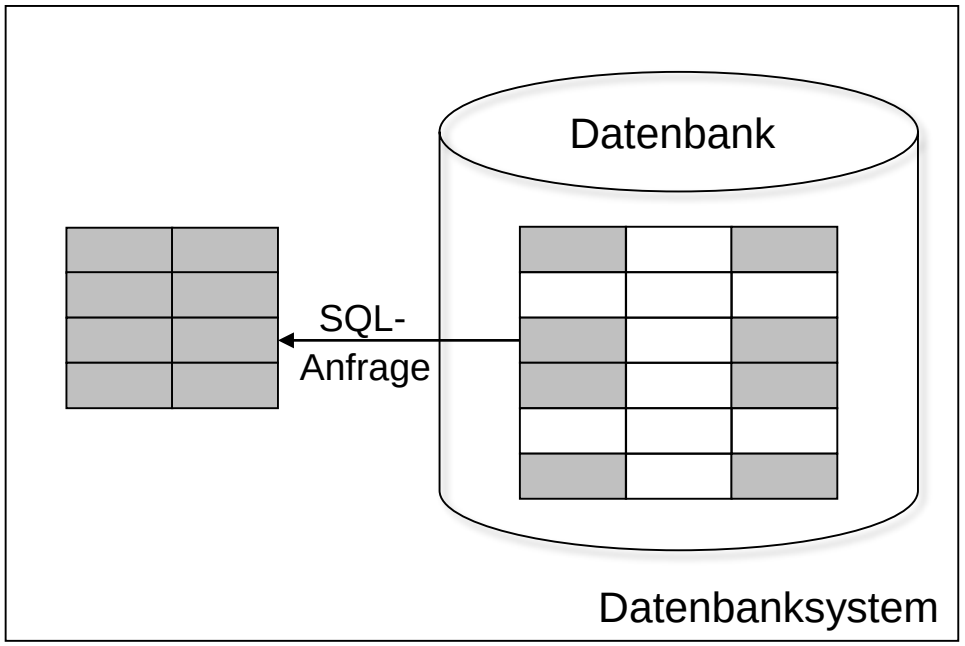
- **Fall 1: Ergebnis der Anfrage höchstens 1 Tupel**
  - Beispiel: *Ausgabe von Preis und Namen des Produkts mit der Produktnummer 203.*



Grundprinzipien

- **Fall 2: Ergebnis der Anfrage mehrere Tupel**
  - Beispiel: *Gib für alle Produkte, die teurer als 100 Euro sind, die Produktnummer und den Preis aus.*

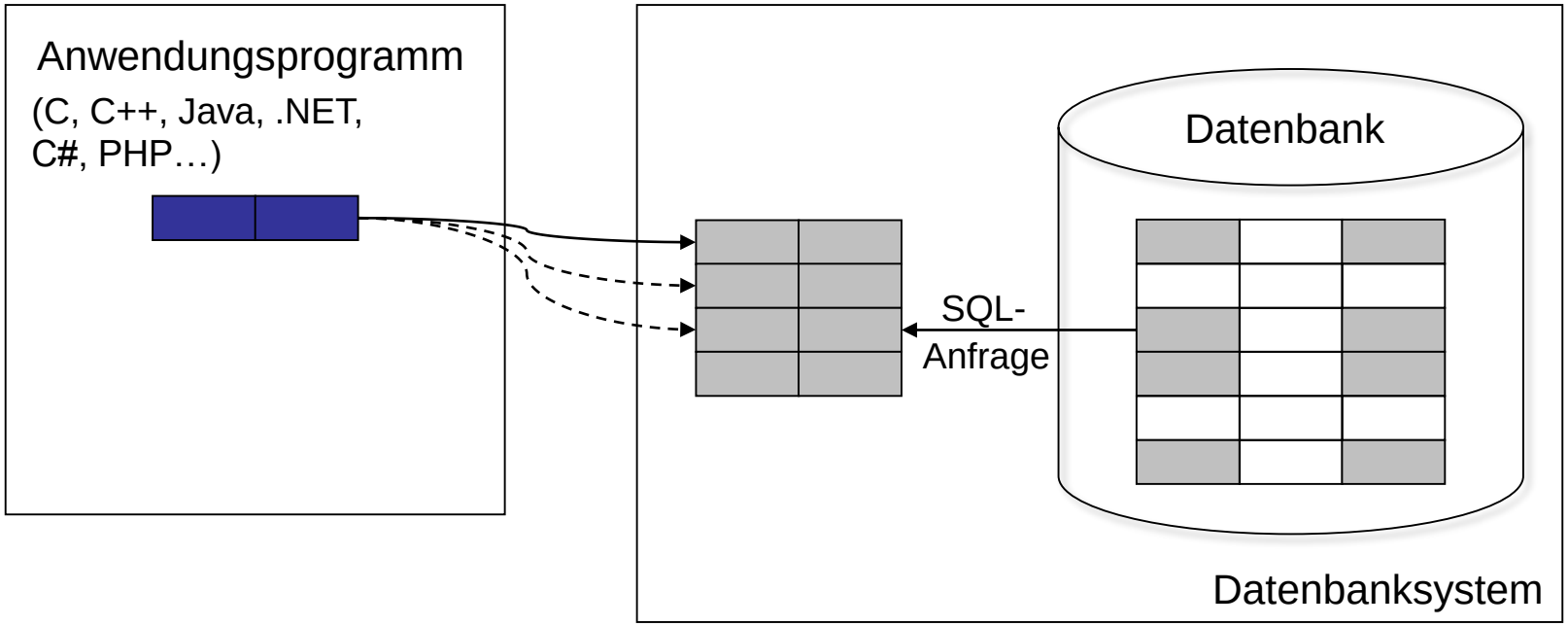
- Problem: Unterschiedliche Datenstrukturkonzepte:
  - (imperative) Programmiersprachen: Tupel
  - SQL: Relation, also **Menge** von Tupeln
  - ⇨ *Impedance mismatch*



Grundprinzipien: Cursor-Konzept

Cursor-Konzept

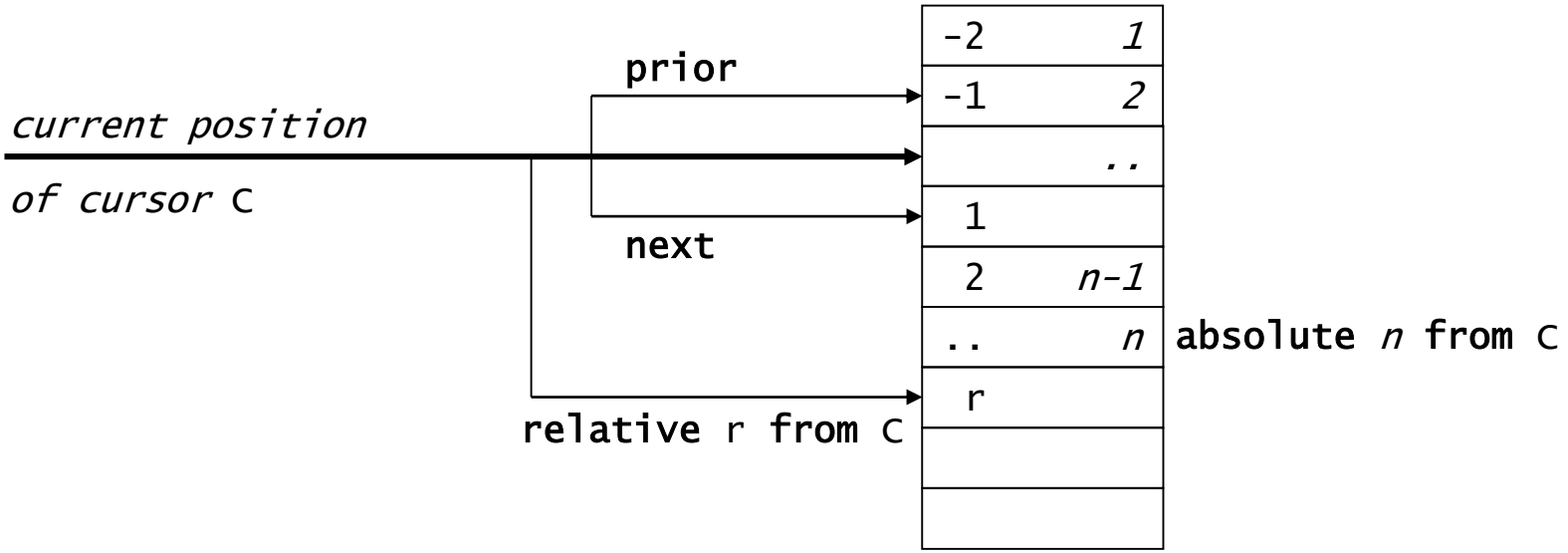
= Abstrakte Sicht auf eine Relation, realisiert als Liste



- Anfrageergebnisse werden sequentiell abgearbeitet!

Cursor-Konzept – SQL92

- SQL92 bietet neben dem einfachen Weitersetzen des Cursors (**fetch**) wesentlich erweiterte Möglichkeiten:



Datenbank-Anwendungsprogrammierung

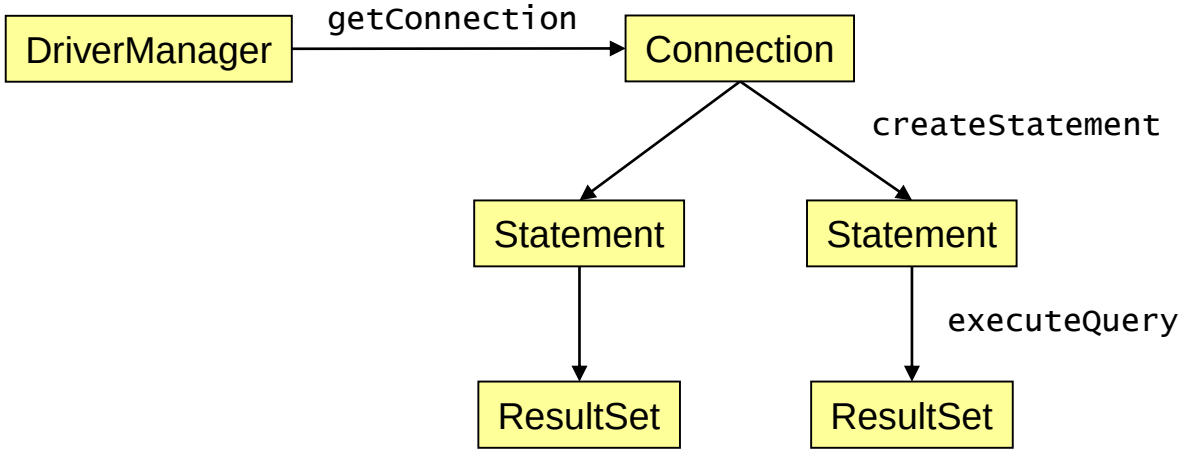
- ✓ Grundprinzipien
- **SQL-Zugriff über APIs**
- **Einbettung von SQL**
- Prozedurale SQL-Erweiterungen
- Integrität und Trigger

SQL-Zugriff über APIs

- SQL/CLI (Call Level Interface)
  - Prozedurale Datenbankschnittstelle für C, C++, Ada, Fortran, Pascal …
  - X/Open bzw. ISO-Standard
  - Datenbankhersteller implementieren diesen Standard und bieten spezifische CLIs an (Oracle CI, DB2 CLI etc.)

- Alternative: **Datenbanksystemunabhängige** Application Programming Interfaces (APIs)
  - Open Database Connectivity (ODBC)
  - Für Java: **Java Database Connectivity (JDBC)**
  - Wichtige Eigenschaften von ODBC/JDBC
    - Sehr weite Verbreitung/Unterstützung
    - Dynamischer Zugriff auf (beliebige) Datenquellen
    - Bei Einhaltung des SQL-Standards ist datenbanksystem-unabhängige Programmierung und sogar Zugriff aus einem Programm auf mehrere (unterschiedliche) DBMS möglich

JDBC – Architektur (grob)



Java Klassen und Interfaces im Paket `java.sql`

- **`java.sql.DriverManager`**
  Laden der Treiber und Aufbau der Verbindung zur Datenbank
- **`java.sql.Connection`**
  ein Connection-Objekt repräsentiert eine Datenbankverbindung
- **`java.sql.Statement`**
  Objekt für die Ausführung von SQL-Anweisungen
- **`java.sql.ResultSet`**
  Objekt für die Repräsentation der Ergebnis-Relation der Abfrage

JDBC – Ablauf

1. Aufbau einer Verbindung zur Datenbank

- Angabe der Verbindungsinformationen
- Auswahl und Laden des Treibers

2. Senden einer SQL-Anweisung

- Definition der Anweisung
- Belegung von Parametern

3. Verarbeiten der Anfrageergebnisse

- Navigation über Ergebnisrelation
- Zugriff auf Spalten

JDBC – Aufbau einer Verbindung

- Treiber laden

```
// Treiber laden
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

- Verbindung herstellen

```
// Aufbau der Verbindung
Connection con;
String url  = "jdbc:odbc:MyOracleDB";
con = DriverManager.getConnection(url, "user", "passwort");
```

  - JDBC-URL spezifiziert
    - Datenquelle/Datenbank
    - Verbindungsmechanismus (Protokoll, Server und Port)

JDBC – Anfrageausführung

- Anweisungsobjekt (Statement) erzeugen

```
// Anweisungsobjekt (Statement) erzeugen
Statement stmt = con.createStatement ();
```

- Anweisung ausführen

```
// Aufbau und Ausführung der Anfrage
String retQuery  = "select ProdNr, ProdName from Produkt";
ResultSet rs = stmt.executeQuery ( retQuery );
```

Für Retrieval-Operationen (`select`):
`executeQuery`

Für Einfüge- (`insert`), Update- (`update`), Lösch- (`delete`) und DDL-Operationen (`create table, drop table`):
`executeUpdate`

```
String insQuery  = "insert into Produkt values (50071, '2Hi4U', 129)";
Statement stmt = con.createStatement ();
int numberRows = stmt.executeUpdate ( insQuery );
```

JDBC – Auswertung des Anfrageergebnis

- Auswertung des ResultSet:
  - Navigation auf der Ergebnismenge mit einem **Cursor**
  - Zugriff auf Spalten des ResultSet mit **get**<*type*> Methoden

```
Statement stmt = con.createStatement ();
String retQuery  = "select ProdNr, ProdName from Produkt";
ResultSet rs = stmt.executeQuery ( retQuery );
String name;
int number;
System.out.println("Produkt-Nr Produkt-Name");
try {
     while (rs.next ())
     {
          number = rs.getInt(1);
          name = rs.getString("ProdName");
          System.out.println (number + " " + name);
     }
     rs.close();
}
catch (SQLException exc) {
     // Fehlerbehandlung
     System.out.println("SQL-Fehler: " + exc);
}
```

Cursor steht am Anfang VOR dem ersten Tupel

Spalten sind von 1 bis n nummeriert;
Anfrage über Spaltenindex oder Spaltenname

JDBC – Nachteile bei der Verwendung der Klasse Statement

- Bei mehrfacher Ausführung wird das Statement immer wieder neu übersetzt und kompiliert – schlecht für Performance ☹
- Gefahr von SQL-Injection ☹
  - Beispiel für SQL-Injection:

```
String sql = "select * from account where username='"
             + username + "' and password='" + password + "'";

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(sql);

if (rs.next()) {
     Boolean loggedIn = true;
     System.out.println("Successfully logged in");
} else {
     System.out.println("Username and/or password not recognized");
}
```

  - Eingabe von *username* und *passwort:* abc' **OR** '1' = '1 führt zu:

```
select * from account where username='abc' OR '1' = '1'
                            and password='abc' OR '1' = '1'
```

JDBC – Vorübersetzung von Anfragen: Klasse PreparedStatement

Statt `Statement` sollte `PreparedStatement` verwendet werden, wenn
- Anfragen wiederholt ausgeführt werden oder
- dynamische Benutzereingabe enthalten.

```
String insQuery  = "insert into Produkt values (?, ?, ?)";
PreparedStatement stmt = con.prepareStatement ( insQuery );

int numberRows;

while (...) {
     // Wiederholtes Setzen der Werte nach Eingabe durch Nutzer
     stmt.setInt ( 1, 50071)
     stmt.setString ( 2, "2Hi4U");
     stmt.setInt ( 3, 129);
     numberRows = stmt.executeUpdate ( );
}
```

JDBC – PreparedStatement

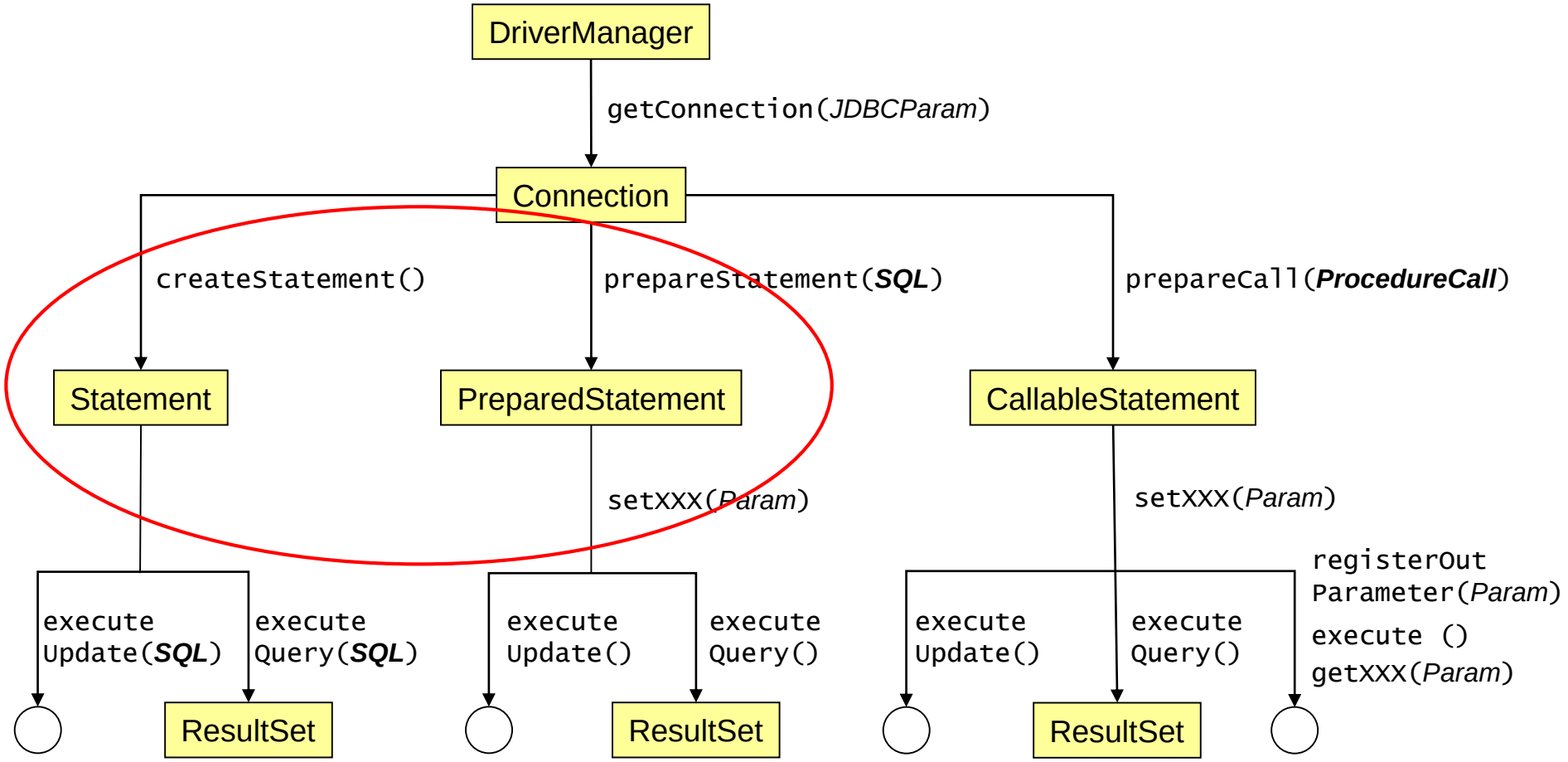
- Die Verwendung von PreparedStatement schützt außerdem vor SQL-Injection ☺

```
String sql = "select * from account where username = ? and password = ?";

PreparedStatement pStmt = con.prepareStatement(sql);

pStmt.setString(1, username);
pStmt.setString(2, password);

ResultSet rs = pstmt.executeQuery();

if (rs.next()) {
     Boolean loggedIn = true;
     System.out.println("Successfully logged in");
} else {
     System.out.println("Username and/or password not recognized");
}
```

  - Eingabe von *username* und *passwort:* abc' OR '1' = '1 führt zu:

```
select * from account where username = " abc' OR '1' = '1' "
                            and password = " abc' OR '1' = '1' "
```

JDBC – Klassen

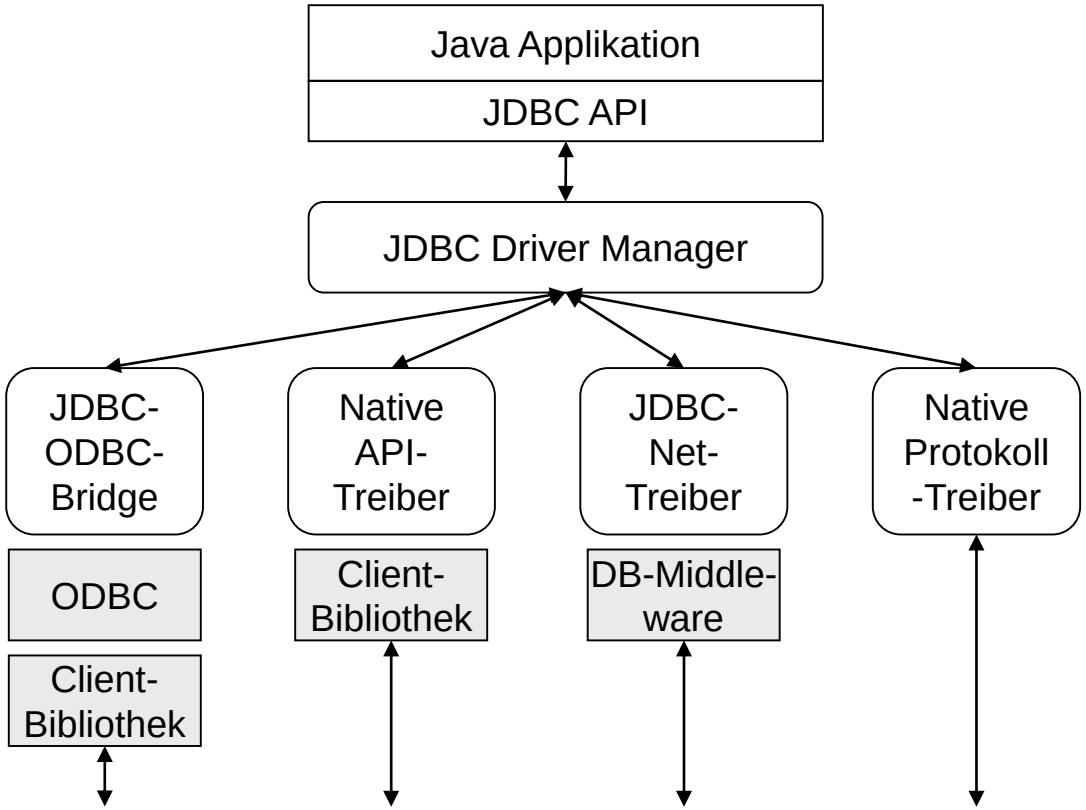


JDBC – Metadaten

- Häufig sehr hilfreich: Abfrage der Metadaten des ResultSets (d.h. der Struktur des Anfrageergebnisses)
- Mehr als 20 verschiedene Methoden – u.a.
  - Abfrage der Spaltennamen (`getColumnName`)
  - der DBMS-spezifischen Datentypen der Spalten (`getColumnTypeName`)
  - der SQL-Datentypen der Spalten (`getColumnType`)

```
...
String stmtString = "select * from " + tabName";
String cName;
String dType;

rs = stmt.executeQuery(stmtString);

ResultSetMetaData rsmdt = rs.getMetaData();
int numberColumns = rsmdt.getColumnCount();
int i;
for (i = 1; i <= numberColumns; i++) {
     String cName = rsmdt.getColumnName(i);
     String dType = rsmdt.getColumnTypeName(i);
     System.out.println(" Name: " + cName + " Datentyp: " + dType);
};
```

JDBC – Treibertypen



- Typ 1: JDBC-ODBC-Bridge - nutzt ODBC-Treiber
- Typ 2: Native-API-Treiber - nutzt DBMS-spezifische APIs
- Typ 3: JDBC-Net-Treiber - nutzt DBMS-unabhängige Middleware
- Typ 4: Native Protokoll-Treiber - nutzt DBMS-spezifisches Protokoll

Verwendung von JDBC-Treibern

Beispiele für Oracle:

- **JDBC-ODBC-Treiber**
  - vorab ODBC-Datenquelle mit Namen **`MyOracleDB`** für Datenbank **`DBPrak`** definiert

```
// JDBC-ODBC-Treiber
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Connection con;
String url  = "jdbc:odbc:MyOracleDB";
con = DriverManager.getConnection(url, "user", "passwort");
```

- **JDBC-Treiber Type 4**
  - Oracle-JDBC-Treiber `ojdbc14.jar` in Build-Path einbinden (als Library o.ä.)

```
// JDBC Type 4 Treiber (JDBC 3.0)
Class.forName("oracle.jdbc.OracleDriver");

Connection con;
String url  = "jdbc:oracle:thin:@//localhost:1521/DBPrak";
con = DriverManager.getConnection(url, "user", "passwort");
```

Eigenschaften SQL über API (z.B. JDBC)

Vorteile

- Ermöglicht (weitgehend) datenbanksystemunabhängige Programmierung
- Flexibel (SQL-Statements können dynamisch zur Laufzeit festgelegt werden)

Nachteile

- Keine Syntax- und Typüberprüfung zur Übersetzungszeit
- Keine Vorübersetzung im Programm möglich (schlecht für Performance)

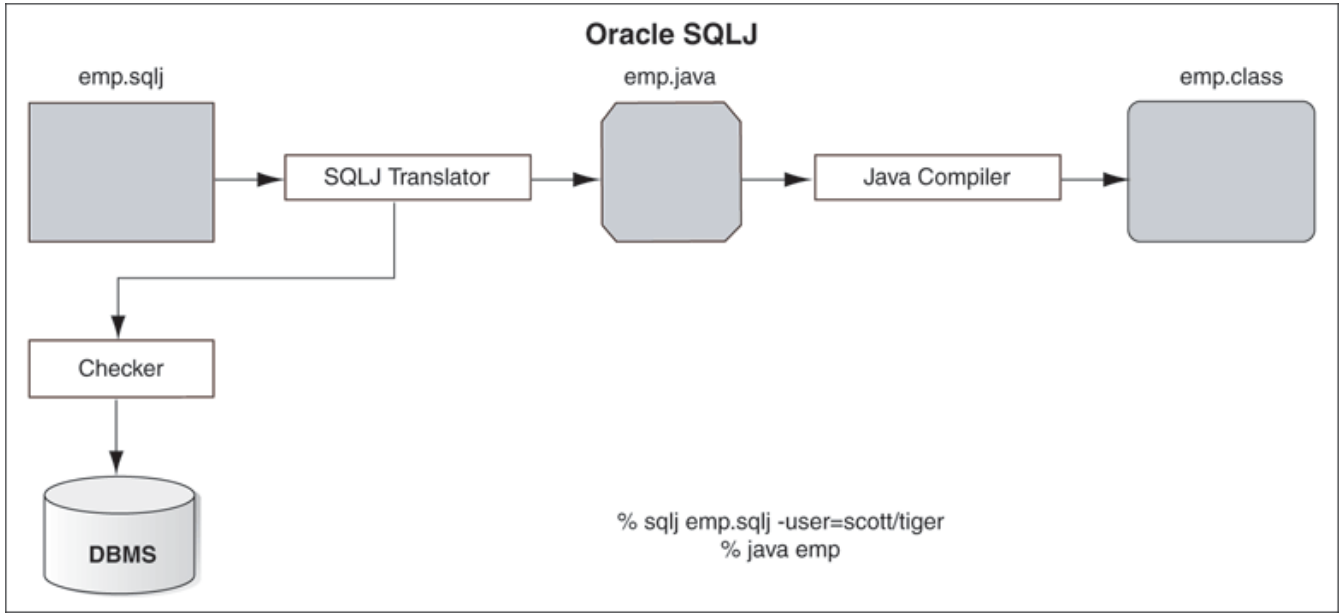
⇨ Alternative:
Einbettung von SQL in Programmiersprache (*Embedded SQL*)

Datenbank-Anwendungsprogrammierung

- ✓ Grundprinzipien
- ✓ SQL-Zugriff über APIs
- Einbettung von SQL
- Prozedurale SQL-Erweiterungen
- Integrität und Trigger

Einbettung von SQL / Embedded SQL

- **Prinzip**
  - Vorübersetzer-Prinzip (Pre-Compiler/Translator)
  - In Programmcode (z.B. C, C++, Java, …) integrierte SQL-Anweisungen werden durch den Pre-Compiler übersetzt, d.h. in Prozeduraufrufe der Wirtssprache übersetzt
  - Anschließend Übersetzung durch Compiler



Quelle: Oracle Database Online Documentation 11g: 2015

Embedded SQL für Java

SQL/OLB (früher: **SQLJ**) **= „Embedded SQL für Java“**

- 1997 vorgeschlagen von IBM, Oracle, Sybase, Informix und Sun; Bestandteil von SQL:1999 – nochmals überarbeiten in SQL:2000 und SQL:2002
- Direkte Einbettung von SQL-Anweisungen in Java Code (= Embedded SQL für Java)
- SQLJ-Anweisungen werden mittels Pre-Compiler (Translator) in Java-Code übersetzt

Spezifikation besteht aus

- ISO/IEC 9075-10:2000, *Information technology—Database languages—SQL—Part 10: Object Language Bindings (SQL/OLB)*
- ISO/IEC 9075-13:2002, *Information technology—Database languages—SQL—Part 13: SQL Routines and Types Using the Java Programming Language (SQL/JRT)*.

Embedded SQL für Java
SQL/OLB (SQLJ)

- Eingebettete SQL-Anweisungen werden durch **`#sql`** gekennzeichnet:
- **Ergebnis-Tupel** können mit einem **`Iterator`** durchlaufen werden (vgl. **Cursor**)

```
...
try {
     #sql iterator PrdItr (int nummer, int preis);
     #sql PrdItr = { select ProdNr, ProdPreis from Produkt };

     while (PrdItr.next ())
     {
        System.out.println (PrdItr.nummer() + " " + PrdItr.preis());
     }
     PrdItr.close();
}
catch (SQLException exc) {
     // Fehlerbehandlung
     System.out.println("SQL-Fehler: " + exc);
}
```

Deklaration des Iterators (Cursors)

Instanziierung des Iterators (Cursors)

- Bereits beim Kompilieren kann die SQL-Syntax überprüft werden!
- Bereits beim Kompilieren kann ein Anfrageplan in der Datenbank berechnet und abgelegt!

JDBC vs. SQLJ

```
Statement stmt = con.createStatement ();
String retQuery  = "select ProdNr, ProdName
from Produkt";
...

ResultSet rs = stmt.executeQuery(retQuery);
while (rs.next ())
{
  i = rs.getInt("ProdName");
  s = rs.getString("ProdName");
  System.out.println(i + " " + s);
}
rs.close();
...
stmt.close();
```

```
#sql iterator PrdItr (int nummer, int
preis);
#sql PrdItr = {select ProdNr, ProdPreis
from Produkt};
...

while (PrdItr.next ())
{
   i = PrdItr.nummer();
   p = PrdItr.preis();
   System.out.println(i + " " + s);
}
PrdItr.close();
```

SQL über API vs. Eingebettetes SQL

SQL über API (z.B. JDBC)

Vorteile

- Flexibel (SQL-Statements können dynamisch zur Laufzeit festgelegt werden)

Nachteile

- Keine Syntax- und Typüberprüfung zur Übersetzungszeit
- Keine Vorübersetzung möglich (schlecht für Performance)

Eingebettetes SQL (z.B. SQL/OLB = SQLJ)

Vorteile

- Syntax- und Typüberprüfung zur Übersetzungszeit
- Vorübersetzung möglich (gut für Performance)

Nachteile

- Geringe Flexibilität (Operationen müssen zur Übersetzungszeit feststehen)

Datenbank-Anwendungsprogrammierung

- ✓ Grundprinzipien
- ✓ SQL-Zugriff über APIs
- ✓ Einbettung von SQL
- • Prozedurale SQL-Erweiterungen
- • Integrität und Trigger

Eingebettetes SQL: Limitationen

- Abarbeitungskontrolle zwischen Anwendungsprogramm (Wirtssprache) und DBMS wird auf sehr kleinen Granularitäten hin- und hergeschaltet
- Anfrageoptimierung durch das DBSM nur für einzelne Anweisung möglich (kritisch für die Performance der Programme!)

- **Lösungsansatz**
  - Erweiterung von SQL um Ablaufkonstrukte wie Sequenz, bedingte Ausführung und Schleifen:

  ⇨ **Prozedurale SQL-Erweiterungen**
    - Proprietär schon früh von DBMS-Herstellern angeboten (z.B. Oracle: PL/SQL)
    - Standardisierung erst in SQL-99 eingeführt

Prozedurales SQL – Aufbau

```
declare
     <Datendeklaration>
begin
     <PL/SQL-Code>
     exception
     <PL/SQL-Code>
end;
/
```

```
declare
     cursor CurProd is
          select ProdNr, ProdPreis
          from Produkt;

     ProdRecord CurProd%rowtype;
```

Prozedurales SQL – Operationale Konstrukte

```
if <Bedingung> then
     <PL/SQL-Anweisungen>;
  [ else
     <PL/SQL-Anweisungen>; ]
end if;
```

```
for <IndexVariable> in <EndlicherBereich>
loop
     <PL/SQL-Anweisungen>;
end loop;
```

```
while <Bedingungsvariable>
loop
     <PL/SQL-Anweisungen>;
end loop;
```

Prozedurales SQL – Cursor-Iteration

```
declare
     cursor CurProd is
         select ProdNr, ProdPreis
         from Produkt;
     ProdRecord CurProd%rowtype;
begin
     open CurProd;
     fetch CurProd into ProdRecord;
     while CurProd%found loop
          dbms_output.put_line (ProdRecord.ProdNr || ProdRecord.ProdPreis);
          fetch CurProd into ProdRecord;
     end loop;
     close CurProd;
end;
```

- Für die sequentielle Abarbeitung eines Cursors existiert eine verkürzte Schreibweise, ohne explizites Öffnen, zeilenweise Lesen und Schließen des Cursor (`open, fetch, close`):

```
...
begin
     for ProdRecord in CurProd loop
          dbms_output.put_line (ProdRecord.ProdNr || ProdRecord.ProdPreis);
     end loop;
end;
```

Prozedurales SQL – Beispiel mit Update

```
declare
     cursor CurProd is
         select ProdNr, ProdPreis
         from Produkt
         for update of ProdPreis;
     ProdRecord CurProd%rowtype;
begin
     for ProdRecord in CurProd loop
         dbms_output.put_line (ProdRecord.ProdNr || ProdRecord.ProdPreis);
         if ProdRecord.ProdPreis < 100 then
             update Produkt
             set ProdPreis = ProdPreis*1.1
             where current of CurProd;
         else
             update Produkt
             set ProdPreis = ProdPreis*1.2
             where current of CurProd;
         end if;
     end loop;
end;
/
```

Prozedurales SQL ohne Cursor

- **Spezialfall: Ergebnis der Anfrage höchstens 1 Tupel**
  - ⇨ Kein Cursor notwendig (und dann auch nicht sinnvoll!)
  - ⇨ Einlesen der Ergebnisse in Host-Variablen mit **`into`**

Beispiel: *Ausgabe des Preises für das Produkt mit der Produktnummer 203.*

```
declare
    preis  Produkt.ProdPreis%type;

begin
    select ProdPreis into preis
    from Produkt
    where ProdNr = 203;

    dbms_output.put_line('Preis: ' || preis);
end;
/
```

Funktionen und Prozeduren

- Nahe liegende Erweiterung zur Strukturierung und Modularisierung: Definition von *Funktionen* und *Prozeduren*

```
function FunktionsName
     ( Param1 ParamTyp1, ..., ParamN ParamTypN ) return ErgebnisTyp
     is
     < Deklarationsteil o h n e Schlüsselwort declare >
     begin
          <PL/SQL-Anweisungen mit return-Anweisung>;
     end;
/
```

```
procedure ProzedurName ( Param1 [ in | out | in out ] ParamTyp1, ... )
     is
     < Deklarationsteil o h n e Schlüsselwort declare >
     begin
          <PL/SQL-Anweisungen>;
     end;
/
```

Stored Procedures

- Funktionen und Prozeduren können auch dauerhaft auf dem DB-Server gespeichert werden:

```
create [ or replace ] function FunktionsName
     ( Param1 ParamTyp1, ..., ParamN ParamTypN ) return ErgebnisTyp
     is
     < Deklarationsteil o h n e Schlüsselwort declare >
     begin
          <PL/SQL-Anweisungen mit return-Anweisung>;
     end;
/
```

```
create [ or replace ] procedure ProzedurName
     ( Param1 [ in | out | in out ] ParamTyp1, ... )
     is
     < Deklarationsteil o h n e Schlüsselwort declare >
     begin
          <PL/SQL-Anweisungen>;
     end;
/
```

Stored Procedures – Beispiel

```
create or replace procedure PreisUpdate (preis1 in decimal, preis2 in decimal)
is
cursor CurProd is
     select ProdNr, ProdPreis
          from Produkt
          for update of ProdPreis;
     ProdRecord CurProd%rowtype;
begin
     for ProdRecord in CurProd loop
          dbms_output.put_line (ProdRecord.ProdNr || ProdRecord.ProdPreis);
          if ProdRecord.ProdPreis < 100 then
               update Produkt
               set ProdPreis = ProdPreis*preis1
               where current of CurProd;
          else
               update Produkt
               set ProdPreis = ProdPreis*preis2
               where current of CurProd;
          end if;
     end loop;
end;
/
```

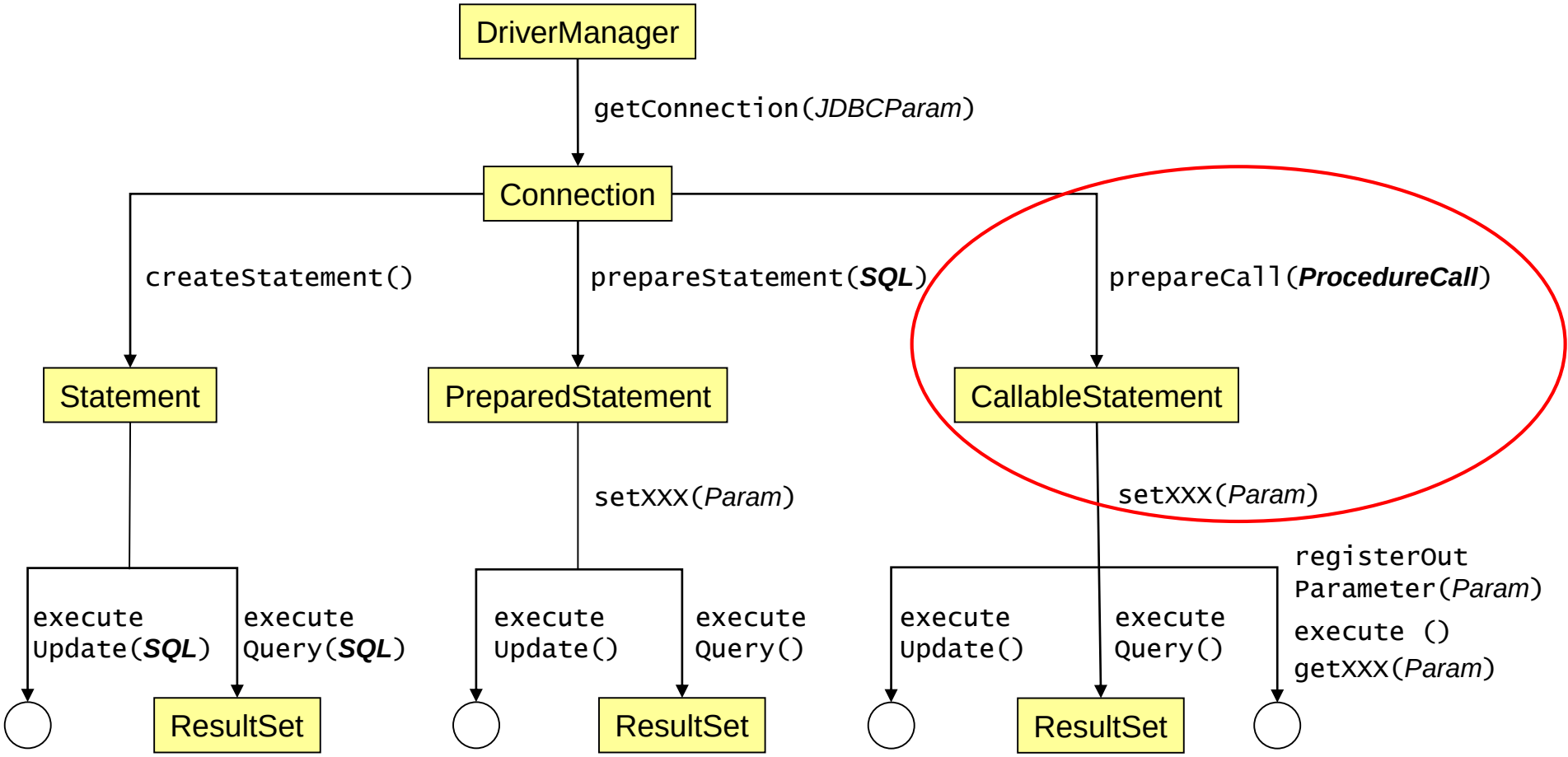
Aufruf im Programm:

```
PreisUpdate (1.1, 1.2);
```

bzw. interaktiv:

```
execute PreisUpdate (1.1, 1.2);
```

Stored Procedures können auch aus Programmen heraus aufgerufen werden



Übung zu *Stored Procedures*

- Vorbemerkung: In Stored Procedures können nicht nur DML- sondern auch **DDL-Befehle** ausgeführt werden; diese müssen mit dem Schlüsselwort **execute immediate** eingeleitet werden:

```
...
   execute immediate 'drop table verlag';
...
```

- *Übung: Schreiben Sie eine Stored Procedure, die alle Tabellen einer Datenbank löscht.*

Stored Procedures – Bewertung

- Funktionen und Prozeduren sind ein wichtiges und bewährtes Strukturierungsmittel für Anwendungen (darüber hinaus können mehrere Funktionen und Prozeduren in Pakete gruppiert werden).

⇨ Vermeidung von Redundanz – relevante Funktionalität kann für verschiedene Anwendungen durch das DBMS bereitgestellt werden.

- Funktionen und Prozeduren sind in der Sprache des DBMS geschrieben sind und können deshalb durch dieses (unabhängig von der Wirtssprache) optimiert werden!

- Ausführung von Stored Procedures vollständig unter Kontrolle des DBMS – geringere Netzwerkbelastung in Client-/Server-Architekturen.

- Stored Procedures können zur Integritätssicherung eingesetzt werden (Stichwort: Trigger - siehe nächster Abschnitt).

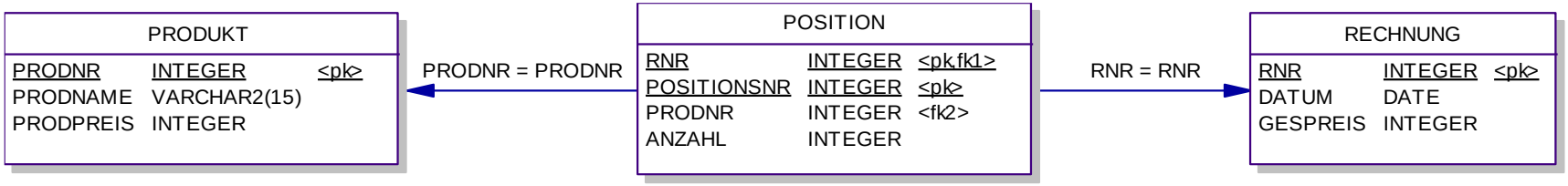
Datenbank-Anwendungsprogrammierung

- ✓ Grundprinzipien
- ✓ SQL-Zugriff über APIs
- ✓ Einbettung von SQL
- ✓ Prozedurale SQL-Erweiterungen
- • Integrität und Trigger

Semantische Integritätsbedingungen

- Semantische Integritätsbedingungen = Integritätsbedingungen, die aus der Modellierung der „Miniwelt“ abgeleitet werden

- Beispiele
  - *Für jedes Produkt muss es eine eindeutige Produktnummer geben.*
  - *Nur existierende Produkte dürfen in einer Rechnungsposition auftauchen.*
  - *Für jede Rechnung muss ein Datum eingetragen sein.*
  - *Der Produktpreis beträgt mindestens 10 Euro.*
  - *Bei einer Veränderung des Preises ist eine Preissenkung nicht zulässig.*



Umsetzung in SQL

- *Für jedes Produkt muss es eine eindeutige Produktnummer geben.*

```
create table Produkt
    ( ProdNr   integer    constraint PK_Produkt primary key,
              ... )
```

Eindeutigkeit für Nicht-Primärschlüsselattribute:

```
ProdName varchar(15) constraint U_Produkt unique,
```

- *Nur existierende Produkte dürfen in einer Rechnungsposition auftauchen.*

```
create table Position
    ( RNr           integer
      PositionsNr   integer,
      ProdNr        integer,
      ...
      constraint PK_Pos primary key ( RNr, PositionsNr ),
      constraint FK_Pos_Prod foreign key ( ProdNr )
                                  references Produkt (ProdNr),
      ... )
```

Umsetzung in SQL

- *Für jede Rechnung muss ein Datum eingetragen sein.*

```
create table Rechnung
    ( RNr integer constraint PK_Rechnung primary key,
      Datum date not null,
      ... )
```

- *Der Produktpreis beträgt mindestens 10 Euro.*

```
create table Produkt
    ( ProdNr    integer  constraint PK_Produkt primary key,
      ProdPreis integer  constraint Min_Preis check (ProdPreis >= 10);
                ... )
```

- *Bei einer Veränderung des Preises ist eine Preissenkung nicht zulässig.*

Umsetzung in SQL

- *Bei einer Veränderung des Preises ist eine Preissenkung nicht zulässig.*
  - ⇨ Dynamische Integritätsbedingung
  - ⇨ können durch Trigger realisiert werden

- Ein Trigger ist eine Folge benutzerdefinierter Anweisungen, die automatisch beim Vorliegen bestimmter Bedingungen (u.a. Einfügen, Update, Löschen) vom DBMS gestartet werden.

- Trigger wurden leider noch nicht in SQL-92 sondern erst in SQL-99 standardisiert

⇨ Teilweise unterschiedliche Syntax in verschiedenen DBMS ☹

Trigger – Beispiel

- *Bei einer Veränderung des Preises ist eine Preissenkung nicht zulässig.*

```
-- Definition eines Trigger-Namens
create or replace trigger keinePreissenkung

-- Ausführungszeitpunkt
before update of ProdPreis on Produkt

-- einmal für jede betroffene Zeile ausführen
for each row

-- einschränkende Bedingung
when (:old.ProdPreis is not null)
begin

    -- Zugriff auf alten bzw. neuen Wert mit :new bzw. :old
    if :new.ProdPreis < :old.ProdPreis then
            :new.ProdPreis := :old.ProdPreis;

        dbms_output.put_line ('Preissenkung nicht erlaubt!');
    end if;
end;
/
```

Aufbau eines Triggers

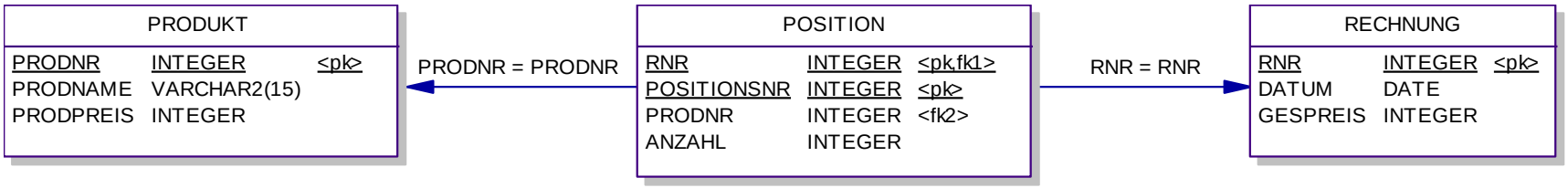
```
create trigger <Triggername>                 Name des Triggers
before | after                              Ausführungszeitpunkt

[ insert | delete |                          Spezifikation
  update [ of <spalte1, ...> ]]              der auslösenden
[ or insert | … ]                            Ereignisse
on <Tabellenname>

[ for each row ]                             Ausführung für alle betroffenen Zeilen
                                             (sonst nur einmal für ganze Tabelle)
[ when <Bedingung> ]                         einschränkende Bedingung
[ declare <Deklarationen> ]                  ggf. Variablendeklarationen
begin
   <Proz. SQL-Anweisungen>                   Anweisungen

     :old.<spalte>                           alter Zustand des Tupel (bei delete
                                             und update)
     :new.<spalte>                           neuer Zustand des Tupel (bei insert
                                             und update)
end;
/
```

Trigger-Beispiel mit Fehlerbehandlung

- *Eingabe oder Änderung eines Verkaufsdatensatzes soll nur in einem bestimmten Zeitfenster möglich sein:*

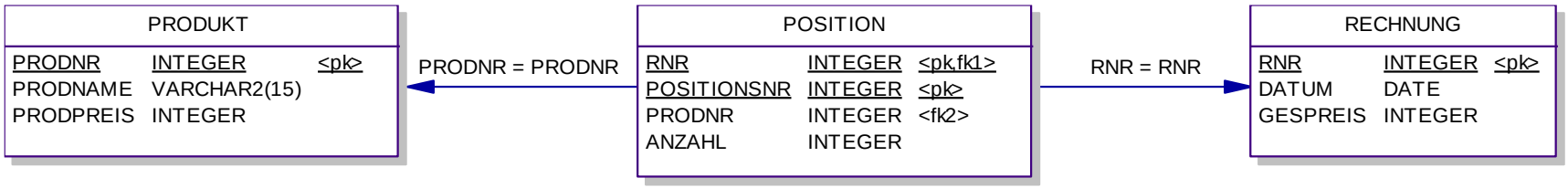
```
create or replace trigger verkauf_aktualisierung
before insert or update on verkauf

-- Deklaration der Exception
declare
   Eingabe_nicht_zulaessig exception;

begin
   if to_char(sysdate, 'HH:MM') not between '08:00' and '17:00'
      then raise Eingabe_nicht_zulaessig;
   end if;

-- Behandlung der Exception
exception
    when Eingabe_nicht_zulaessig then raise_application_error
    (-20001, 'Nur zwischen 8:00 und 17:00 Uhr Daten eingeben! ');
end;
```

Trigger-Beispiel – 1(3)



- *Beim Hinzufügen einer Rechnungsposition soll der Gesamtpreis der Rechnung aktualisiert werden.*

Trigger-Beispiel – 2(3)



1. Lösungsvariante: Berechnung innerhalb des Triggers

```
create or replace trigger PositionInsertTrigger

after insert on Position
for each row
declare
    aktPreis    Produkt.ProdPreis%type;
begin
    select ProdPreis into aktPreis
    from Produkt
    where ProdNr = :new.ProdNr;

    update Rechnung
    set GesPreis = GesPreis + aktPreis * :new.Anzahl
    where RNr = :new.RNr;
    dbms_output.put_line('Aktualisierung ausgeführt');
end;
/
```

Trigger-Beispiel – 3(3)

2. Lösungsvariante: Berechnung in separater Stored Procedure:

```
create or replace procedure setGesPreis
      (aktRNr in integer, aktPNr in integer, aktAnzahl in integer)
is
  aktPreis  Produkt.ProdPreis%type;
begin
  select ProdPreis into aktPreis
  from Produkt
  where ProdNr = aktPNr;

  update Rechnung
  set GesPreis = GesPreis + aktPreis*aktAnzahl
  where RNr = aktRNr;
end;
/
```

Wiederverwendung:

Wie sieht der entsprechende `delete`-Trigger bzw. `update`-Trigger aus?

```
create or replace trigger PositionInsertTrigger
after insert on Position
for each row
begin
  setGesPreis (:new.RNr, :new.ProdNr, :new.Anzahl);
end;
/
```

Übung zu Triggern

- *Beim Einfügen einer Rechnung soll automatisch eine Rechnungsnummer generiert werden. Diese soll um 1 größer sein, als die letzte vergebene Rechnungsnummer.*

  Allgemeiner Anwendungsfall:
  Erzeugen eines künstlichen, sequentiellen Primärschlüssels.

Datenbank-Anwendungsprogrammierung

- ✓ Grundprinzipien
- ✓ SQL-Zugriff über APIs
- ✓ Einbettung von SQL
- ✓ Prozedurale SQL-Erweiterungen
- ✓ Integrität und Trigger

Vorlesung Datenbanken