

# 4.4 SQL - Programmiersprachen-Anbindung

- 4.1 SQL – DDL und DML
- 4.2 SQL-Anfragen 1
- 4.3 SQL-Anfragen 2
- 4.4 Programmiersprachen-Anbindung
  - JDBC
  - Embedded SQL

# SQL und Programmiersprachen

- SQL ist Datenbanksprache, keine Programmiersprache
  - DML nicht Turing-vollständig
- Möglichkeiten der Verknüpfung von SQL mit Programmiermechanismen
  - Erweiterung von SQL um Konstrukte von Programmiersprachen  
s. PL/SQL in Kap. 4.2
  - Integration in eine bestehende Programmiersprache
- Impedance mismatch (semantische Lücke):  
Nicht-Zusammenpassen von Programmiersprachen und Datenbankkonzepten
  - Keine Mengenoperatoren in Programmiersprachen
  - Keine komplexeren Typkonstrukte (Records, Listen) in DB-Modellen
  - Keine NULL-Werte in Datentypen von Programmiersprachen

# Formen der Einbindung von SQL

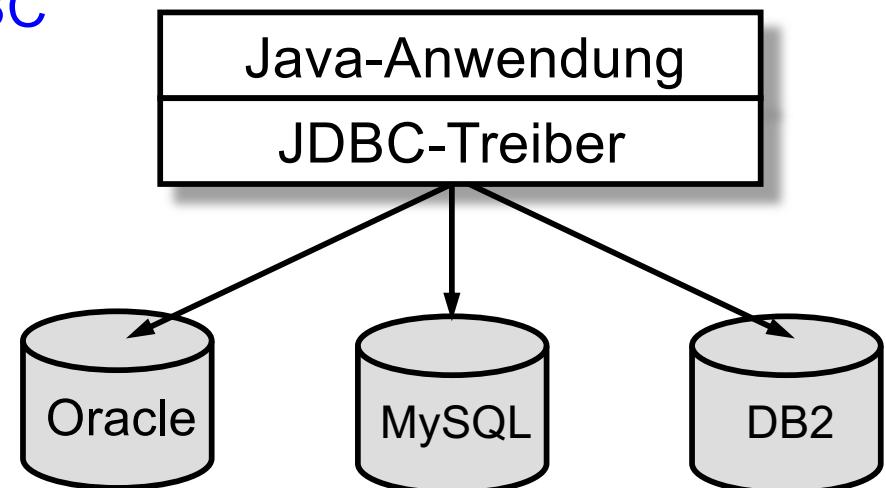
- Statische Einbindung
  - Anfragen sind zur Übersetzungszeit der Programme bekannt
  - Prüfung und Optimierung zur Übersetzungszeit
  - Häufig Verwendung eines Precompilers
- Dynamische Einbindung
  - Anfragen (zumindest einige) werden erst zur Laufzeit von Programmen erstellt
  - Übergabe der Anfrage als textueller Wert einer Variablen
  - Größere Flexibilität
  - Prüfung und Optimierung erst zur Laufzeit

# Beispiele für Einbindung von SQL

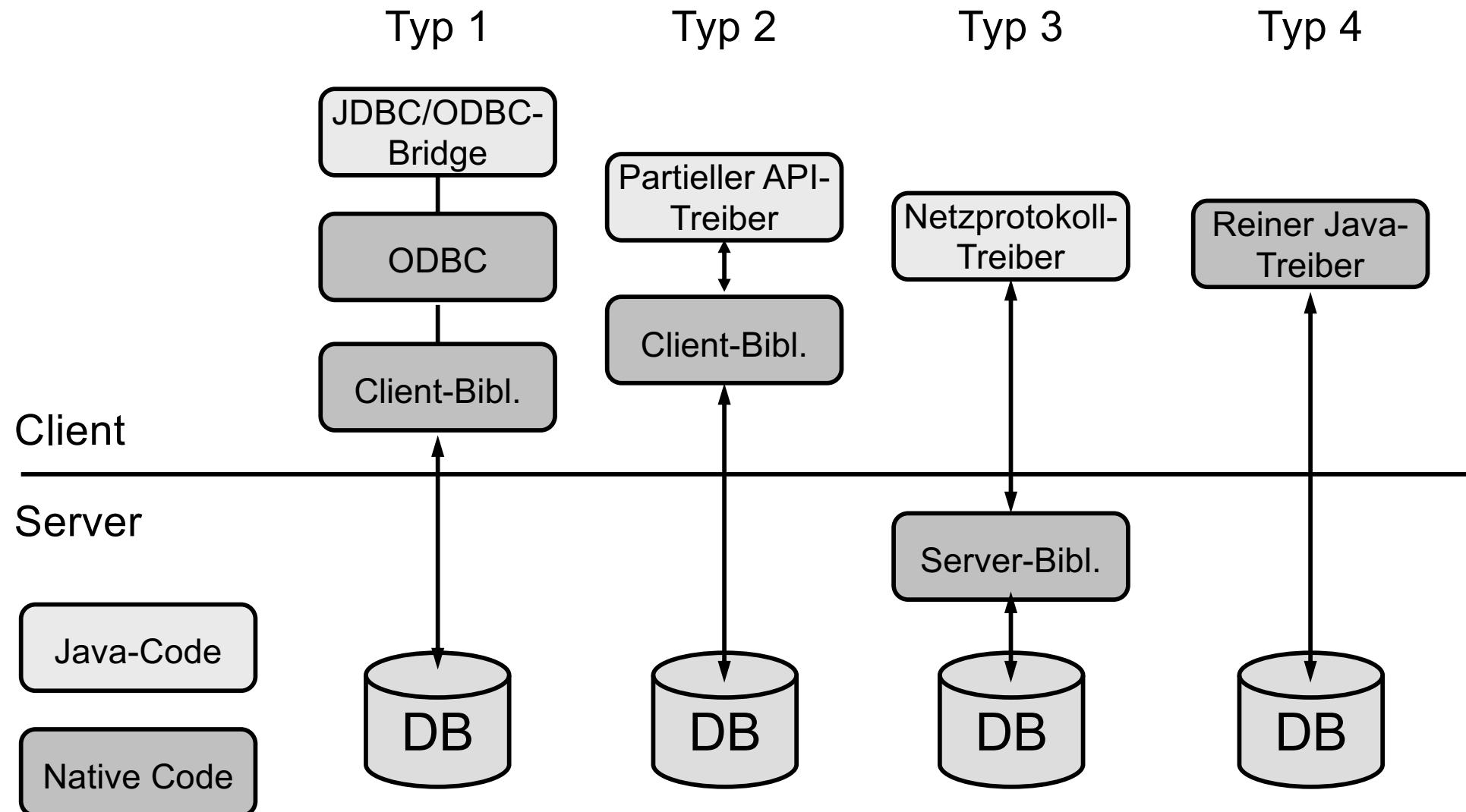
- ODBC
  - Standardisierte Schnittstelle namens Open Database Connectivity
- Embedded SQL (ESQL)
  - Statische Einbettung von SQL mit Hilfe eines Precompilers
  - Dynamische Einbettung für komplexere Suchanfragen
  - SQL92-Standard
- JDBC
  - Dynamische Einbettung von SQL in Java
- SQLJ
  - Statische Einbettung von SQL in Java
- Zahlreiche weitere Einbindungen
  - LINQ (Language Integrated Query): in .NET-Sprachen wie Visual Basic oder C#

# JDBC als Standard für DB-Schnittstellen

- JDBC (Java Database Connectivity)
  - Objektorientiertes DB-Interface für Java, entwickelt von SUN
  - Ziel: standardisierter Zugriff auf beliebiges DBMS
  - Bei DBMS-Wechsel muss nur anderer Treiber eingebunden werden
  - Unterstützung von allen bedeutenden DBMS-Herstellern
- Übersichtlicher und einfacher benutzbar als ODBC
- Literatur
  - R.M. Mennor: Expert Oracle JDBC Programming, Apress, 2005 als E-Book in HTWG-Bibliothek



# JDBC Treibertypen



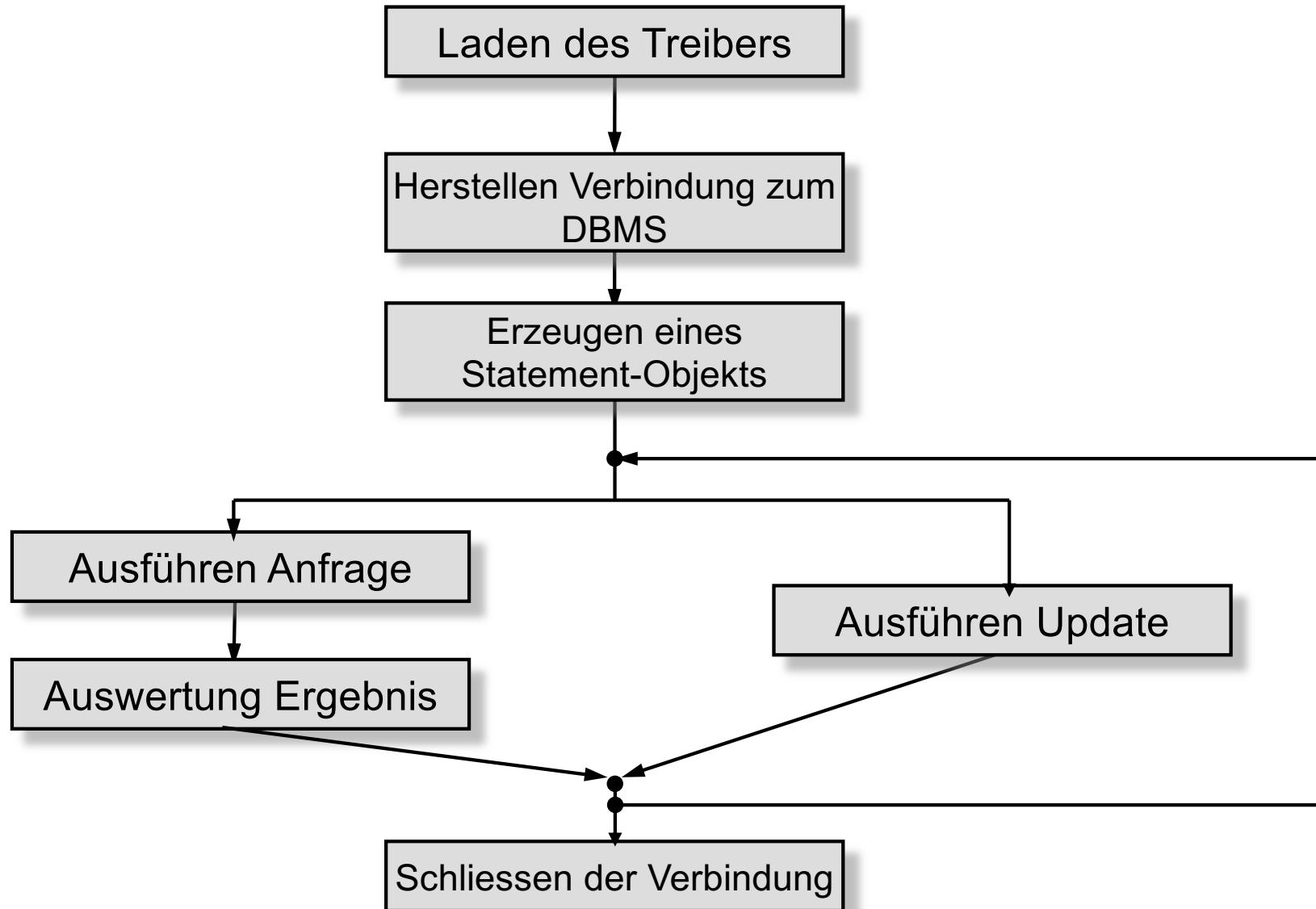
# JDBC Treiber Typen

- Typ 1
  - Mapping auf anderes API, wie z.B. ODBC
  - Einschränkungen, z.B. in Portabilität
- Typ 2
  - Implementierung teilweise in Java, teilweise in native code
  - „Thick drivers“
  - Eingeschränkte Portabilität
  - Oracle's OCI (Oracle Call Interface) client-side driver

# JDBC Treiber Typen

- Typ 3
  - Java-Treiber
  - Kommunikation mit Middleware Server über datenbankunabhängiges Protokoll
  - Middleware Server übersetzt auf datenbankabhängiges Protokoll
- Typ 4
  - Java-Treiber
  - Kommunikation direkt mit Datenbank
  - Keine Software auf Clientseite
  - Geringere Flexibilität
  - „Thin driver“
  - Für Übungen empfohlen: ojdbc8.jar

# Phasen eines JDBC-Datenbankzugriffs



# JDBC-Programm

## Unvollständige Auszüge

```
import java.sql.*;  
  
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());  
  
String url= "jdbc:oracle:thin:@oracle19c.in.htwg-  
konstanz.de:1521:ora19c";  
  
Connection conn = DriverManager.getConnection( url, "name", "passwd");  
Statement stmt = conn.createStatement();  
  
ResultSet rs = stmt.executeQuery(  
        "SELECT name, gehalt FROM pers WHERE anr = 'K55'");  
  
while (rs.next()) {  
    String r1 = rs.getString("name");  
    Int r2 = rs.getInt("gehalt");  
    system.out.println(r1 + " verdient " + r2);  
}  
  
rs.close(); stmt.close(); conn.close();
```

Include SQL

Treiber laden

Verbindung zur DB herstellen

Statement erzeugen

Anweisung absenden

Ergebnisse verarbeiten

# JDBC-API

- Package `java.sql`
- `Connection`
  - Interface zu einer spezifischen Datenbank
  - Verbindungsauflaufbau ohne Benutzer und Passwort

```
static Connection getConnection (String url)
```
  - Verbindungsauflaufbau mit Benutzer und Passwort

```
static Connection getConnection  
(String url, String user, String password)
```
- `Statement`
  - Ausführung von SQL-Statements
  - Statement s = conn.createStatement();

# JDBC-API

- **Klasse Statement – Suchen**

- ```
public ResultSet executeQuery(String sql)
throws SQLException;
```

- **Klasse Statement – Updates**

- ```
public int executeUpdate(String sql)
throws SQLException;
```
  - Rückgabewert: Row count für insert, update und delete-Statements, bzw. 0 falls Statement keine Rückgabe besitzt

- **ResultSet – Ergebnis einer SQL-Suche**

- Zugriff über Spaltenname oder Index
  - ```
ResultSet rs = s.executeQuery("sqlAusdruck");
```
  - ```
String r1 = rs.getString("name"); // Spaltenname
```
  - ```
String r1 = rs.getString(1); // Index
```

# JDBC-API

- Methoden der Klasse SQLException
  - getMessage() : Fehlermeldung mit ORA-Nummer
  - getErrorCode() : ORA-Nummer
  - getSQLState() : Code für SQL-Zustand
  - printStackTrace() : Exception Stack Trace
- Transaktionen
  - conn.commit();
  - conn.rollback();
  - Isolation Level in Klasse Connection:

```
public void setTransactionIsolation( int level )
throws SQLException;
```

# Erweiterte Möglichkeiten von JDBC

- Abfrage Meta-Informationen (Data Dictionary)
  - Methode `Connection.getMetaData`
- Änderbare Ergebnismengen
  - Werte innerhalb einer Ergebnismenge können verändert werden
- Scrollbare Ergebnismengen
  - Cursor kann vorwärts und rückwärts positioniert werden
  - Absolute Positionierung
  - Cursor-Position kann ermittelt werden
- Transaktionsbehandlung
  - Commit und Rollback von Transaktionen

# Scrollbar Cursor in JDBC 2.0

- Festlegen der Eigenschaften im Statement

```
- Statement stmt =  
    conn.createStatement(int resultSetType,  
                        int resultSetConcurrency);
```

- Parameter ResultSetType

| resultSetType                     | Bedeutung                                                                       |
|-----------------------------------|---------------------------------------------------------------------------------|
| ResultSet.TYPE_FORWARD_ONLY       | ResultSet kann nur einmal vom ersten bis zum letzten Element durchlaufen werden |
| ResultSet.TYPE_SCROLL_INSENSITIVE | ResultSet ist scrollbar, simultane Änderungen anderer Nutzer bleiben verborgen  |
| ResultSet.TYPE_SCROLL_SENSITIVE   | ResultSet ist scrollbar, Änderungen anderer Nutzer schlagen auf Werte durch     |

# Scrollbar Cursor in JDBC 2.0

- **Parameter** ResultSetConcurrency

| resultSetConcurrency       | Bedeutung                            |
|----------------------------|--------------------------------------|
| ResultSet.CONCUR_READ_ONLY | Ergebnis kann nicht verändert werden |
| ResultSet.CONCUR_UPDATABLE | Ergebnis kann editiert werden        |

# JDBC-Programm mit scrollbarem Cursor

```
Statement stmt =  
    conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
                        ResultSet.CONCUR_UPDATABLE) ;  
  
ResultSet rs =  
    stmt.executeQuery("SELECT * FROM pers");  
  
rs.absolute(5);                      // Cursor auf Position 5 setzen  
rs.first();                          // Cursor auf erste Position setzen  
rs.next();                           // Cursor nächste Position setzen  
rs.previous();                      // Cursor vorherige Position setzen  
rs.updateDouble("Gehalt", 40000);     // Wert ändern  
  
conn.commit();                       // Commit  
conn.rollback();                     // Rollback  
...  
...
```

# JDBC – weitere Methoden

```
...  
  
boolean b;  
  
int pos = rs.getRow()          // Ermittelt Cursor Position  
  
rs.first()                    // Bewegt Cursor auf erstes Element  
rs.beforeFirst()               // Bewegt Cursor vor erstes Element  
                               // ergibt true bei leerem ResultSet  
                               // praktisch bei folgender Schleife  
                               // while(rs.next()) {...}  
  
rs.last()                     // Bewegt Cursor auf letztes Element  
rs.afterLast()                // Bewegt Cursor auf Ende ResultSet  
  
b = rs.isFirst()               // Test ob Cursor auf erstem Element  
b = rs.isBeforeFirst();        // Test ob Cursor vor erstem Element  
  
b = rs.isLast();               // Test ob Cursor auf letztem Element  
b = rs.isAfterLast();          // Test ob Cursor nach letztem Element
```

# Prepared Statements in JDBC

- Prepared Statement
  - „Vorbereitete Anweisung“
  - Wird vorübersetzt, daher Geschwindigkeitsvorteil
  - Enthält noch keine Parameterwerte
  - Übergabe durch Platzhalter
  - Verhindern von SQL-Injections

```
PreparedStatement ps = Connection.prepareStatement(  
    "SELECT gehalt, beruf FROM pers WHERE name=?");  
  
ps.setString(1, persname);  
  
ResultSet rs = ps.executeQuery(); .
```

# Parameter binding

- SQL-Query ohne Parameter binding
  - Erzeugung des Strings eventuell durch Konkatenation (mit „+“)

```
query = "SELECT * FROM PERS WHERE aname = 'Entwicklung'  
        AND ort = 'Konstanz'";
```

- SQL-Query mit Parameter binding
  - Nachträgliches Binden der variablen Anteile

```
PreparedStatement ps = conn.prepareStatement("SELECT * FROM  
PERS WHERE aname = ? AND ort = ?";  
  
...  
c.setString(1, "Entwicklung");  
c.setString(2, "Konstanz");
```

# JDBC und Large Objects

- Locator
  - Logischer Pointer auf die SQL LOBs statt direkter Daten
- Zugriff von JDBC auf LOBs
  - Zugriff auf LOBs durch Locator
  - Klassen `oracle.sql.BLOB` und `oracle.sql.CLOB`
  - Zugriff auf Locator durch `ResultSet.getObject()`

```
ResultSet rs =
    stmt.executeQuery
        ("SELECT blob_col, clob_col FROM lob_table");
while (rs.next())
{
    oracle.sql.Blob blob = (BLOB)rs.getObject(1);
    oracle.sql.Clob clob = (CLOB)rs.getObject(2);
    ...
}
```

# JDBC und Large Objects

- Lesen und Schreiben von BLOB und CLOB-Daten
  - Verwendung von Methoden zum Lesen von LOBs in Input Stream bzw. Schreiben von Output Stream in LOBs
  - Lesen aus einem BLOB: `getBinaryStream()`
  - Schreiben in ein BLOB: `getBinaryOutputStream()`
  - Lesen aus einem CLOB: `getCharacterStream()`
  - Schreiben in ein CLOB: `getCharacterOutputStream()`

```
java.io.OutputStream outstream;  
  
byte[] data = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
  
// write the array of binary data to a BLOB  
outstream = ((BLOB)my_blob).getBinaryOutputStream();  
outstream.write(data);  
...
```

Quelle: Oracle Database Online Documentation

# JDBC und Large Objects

- Wichtig
  - Die “Write“-Methoden schreiben direkt in die Datenbank, wenn man auf den OutputStream schreibt
  - Kein Update/Commit notwendig um die Daten zu schreiben

```
java.io.Writer writer

// read data into a character array
char[] data = {'0','1','2','3','4','5','6','7','8','9'};

// write the array of character data to a CLOB
writer = ((CLOB)my_clob).getCharacterOutputStream();
writer.write(data);
writer.flush();
writer.close();
...
```

Quelle: Oracle Database Online Documentation

# JDBC und Large Objects

- Weitere Methode von `java.sql.Blob`
  - `byte[] getBytes(long pos, int length)`
- Weitere Methode von `java.sql.Clob`
  - `String getSubString(long pos, int length)`

Quelle: Oracle Database Online Documentation

# Statische vs. dynamische Einbindung

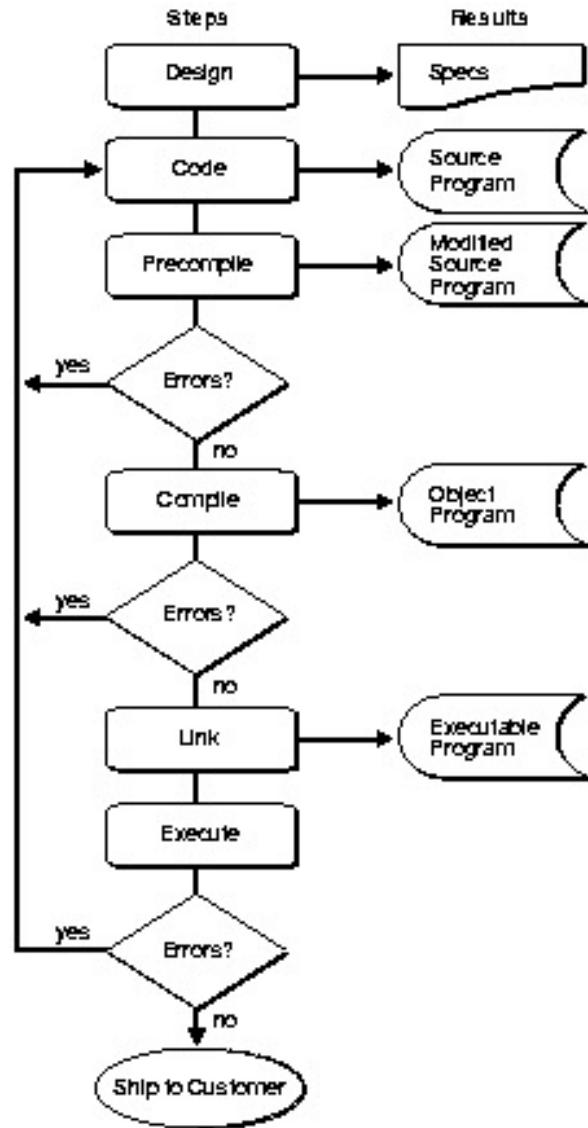
| Statische Einbindung                                                                      | Dynamische Einbindung                                |
|-------------------------------------------------------------------------------------------|------------------------------------------------------|
| - Anfragen müssen zum Übersetzungszeitpunkt bereits bekannt sein                          | + Erstellung von Anfragen zur Laufzeit<br>+ flexibel |
| + leichter lesbar und verständlich                                                        | - schwerer lesbar                                    |
| + Syntaktische und semantische Korrektheit wird bereits zum Übersetzungszeitpunkt geprüft |                                                      |
| + Performancevorteil, da Optimierung von Anfragen bereits zum Übersetzungszeitpunkt       |                                                      |

## 4.3 SQL

### Programmiersprachen-Anbindung

- Grundlagen
- Fortgeschrittene Konzepte
- Programmiersprachen-Anbindung
  - JDBC
  - Embedded SQL

# Embedded SQL Anwendungsentwicklung



Quelle: Oracle9i Database Online Documentation

# Embedded SQL

## Ein einfaches Beispielprogramm

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL WHENEVER SQLERROR GOTO fehler;

cout << "Geben Sie das neue Gehalt an: ";
cin >> gehalt;

EXEC SQL UPDATE pers
    SET Gehalt = :gehalt
    WHERE pnr = 1234;

EXEC SQL COMMIT WORK; //Sichern seit letztem Commit

return 0;

fehler:
cout << "Fehler beim Update der Relation Personal\n";
EXEC SQL ROLLBACK WORK; //Rücknahme seit letztem Commit
```

# Embedded SQL (ESQL)

- Host-Variable
  - Variable der Wirtssprache, die auch in SQL-Anweisungen verwendet werden kann
  - Gesonderte Deklaration

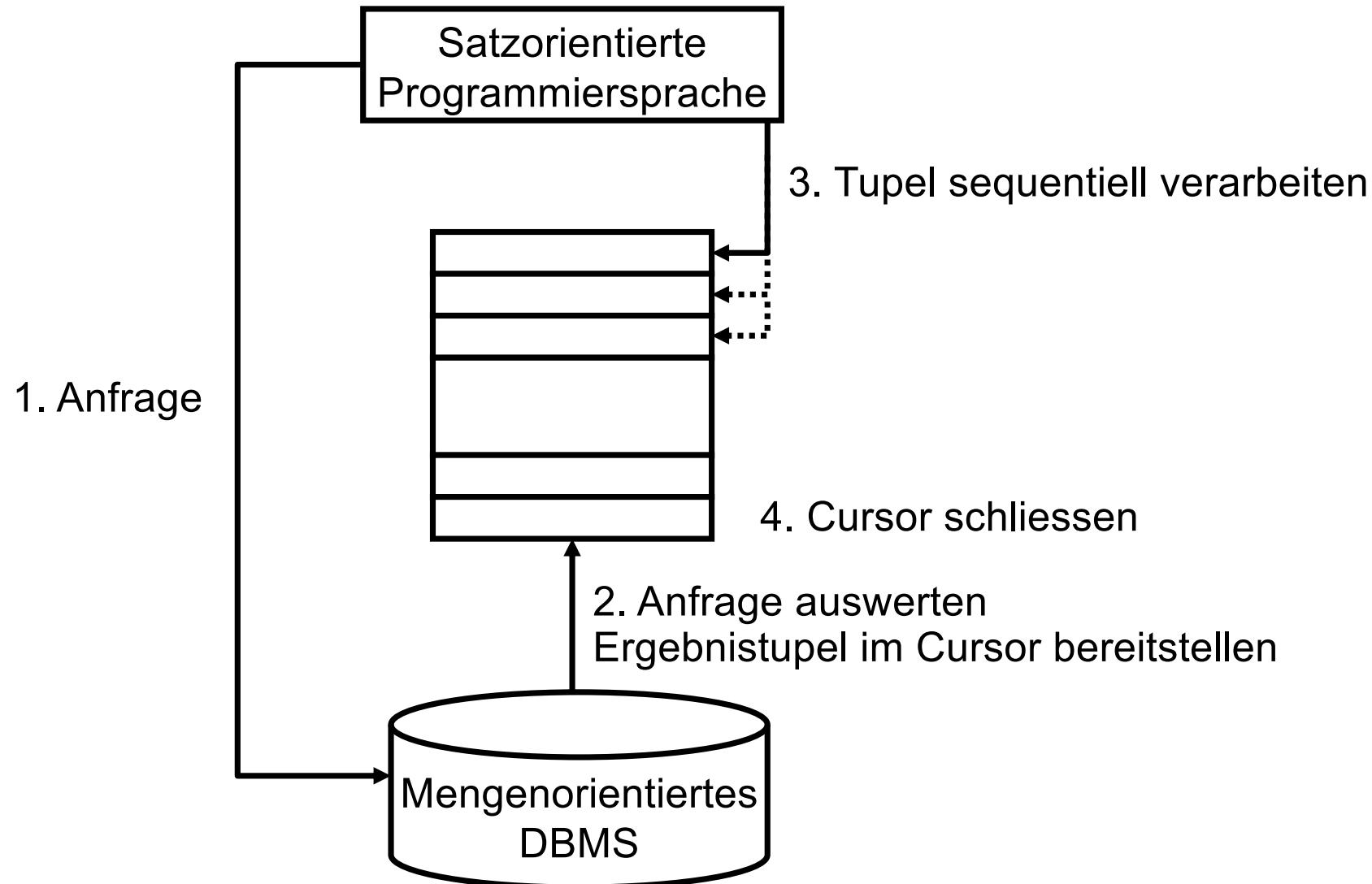
```
EXEC SQL BEGIN DECLARE SECTION
...
EXEC SQL END DECLARE SECTION;
```
  - Verwendung in SQL :Variablenname
- Variablen zur Speicherung von Fehlercodes
  - SQLCODE: Fehlercode, Fehler falls >0
  - SQLSTATE: Genormter Statuscode, der Erfolg oder Fehlermeldung eines Befehls angibt

# Embedded SQL

## Einführung Cursor

- Kopieren des Ergebnisses einer Anfrage nur möglich, wenn höchstens ein Tupel zurückgeliefert wird
  - ```
EXEC SQL SELECT AVG(gehalt)
    INTO :avggehalt
    FROM pers;
```
- Einführung eines Cursors
  - Ein Cursor ist eine (Zeiger-) Variable, welche explizit für jeden SELECT-Befehl, dessen Antworttabelle mit dem Cursor durchlaufen werden soll, deklariert werden muß
- Ein Cursor wird benötigt
  - wenn ein SQL-Befehl mehr als ein Tupel als Antwort liefert
- Ein Cursor wird nicht benötigt
  - bei INSERT-, UPDATE, DELETE-Befehlen
  - wenn höchstens ein Tupel als Antwort geliefert wird

# Cursor-Konzept



# Cursor-Konzept

- Cursor-Deklaration
  - EXEC SQL DECLARE cursor-name CURSOR FOR query  
[FOR {READ ONLY | UPDATE [OF SPALTE] } ]
- Cursor-Eröffnung
  - EXEC SQL OPEN cursor-name
- Übergabe der Tupelwerte an Host-Variable
  - EXEC SQL FETCH cursor-name INTO :var1 [, :var2]
- Update eines aktuellen Tupels
  - EXEC SQL UPDATE table-name SET column-name = expr1  
WHERE CURRENT OF cursor-name
- Löschen eines aktuellen Tupels
  - EXEC SQL DELETE FROM table-name  
WHERE CURRENT OF cursor-name
- Schließen des Cursors
  - EXEC SQL CLOSE cursor-name

# Embedded SQL

## Beispielprogramm 2

```
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR uid[20];
  VARCHAR pwd[20];
  VARCHAR ort[20];
  VARCHAR name[20];
  VARCHAR beruf[20];
EXEC SQL END DECLARE SECTION;

strcpy(ort.arr, "Konstanz");

EXEC SQL INCLUDE sqlca;
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;

EXEC SQL DECLARE C1 CURSOR FOR
  SELECT pers.name, pers.beruf
  FROM pers, abt
  WHERE pers.anr = abt.anr AND ort = :ort;
EXEC SQL OPEN C1;

EXEC SQL WHENEVER NOT FOUND DO break;
for (;;)
{
  EXEC SQL FETCH C1 INTO :name, :beruf;
  printf ("%s %s\n", name.arr, beruf.arr);
}

EXEC SQL CLOSE C1;
```

# Embedded SQL

## Vergleich alternativer Programmierstile

```
...  
EXEC SQL INCLUDE sqlca;  
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;  
  
EXEC SQL DECLARE C1 CURSOR FOR  
  SELECT pers.name, pers.beruf  
  FROM pers, abt  
  WHERE pers.anr = abt.anr AND ort = :ort;  
EXEC SQL OPEN C1;  
  
EXEC SQL WHENEVER NOT FOUND DO break;  
for (;;) {  
  EXEC SQL FETCH C1 INTO :name, :beruf;  
  printf ("%s %s\n", name.arr, beruf.arr);  
}  
...
```

Alternative 1

```
...  
EXEC SQL INCLUDE sqlca;  
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;  
  
EXEC SQL DECLARE C1 CURSOR FOR  
  SELECT pers.name, pers.beruf, pers.ort  
  FROM pers, abt  
  WHERE pers.anr = abt.anr;  
EXEC SQL OPEN C1;  
  
EXEC SQL WHENEVER NOT FOUND DO break;  
for (;;) {  
  EXEC SQL FETCH C1 INTO :name, :beruf, :ort;  
  if (strcmp(ort.arr, "Konstanz"))  
    printf ("%s %s\n",  
           name.arr, beruf.arr);  
}  
...
```

Alternative 2

- Unterschiedliche Programmierstile
  - Test auf Ort wird einmal innerhalb SQL, einmal ausserhalb durchgeführt
  - Welche Alternative ist besser ???

# Weitere Ansätze

- LINQ
  - Komponente von Microsofts .NET-Framework

```
var query = from article in this.Articles  
            where article.Name.StartsWith("A")  
            orderby article.ID  
            select article;  
  
foreach (var article in query) {  
    Console.WriteLine (article.Name );  
}
```

- Hibernate
  - Object-Relational Mapping (OR-Mapping, oder kurz ORM)
  - Mapping Java Klassen auf Tabellen
- JDO (Java Data Objects)
  - Framework zur persistenten Speicherung von Java-Objekten