# 5. Anwendungsprogrammierung

- Anwendungen werden typischerweise in imperativen Sprachen entwickelt (C, Cobol, Fortran, Java, C++…)
  - reichhaltige Funktionalität bei der "gewöhnlichen" Programmierung
  - keine Unterstützung von Persistenz (Datenbankprogrammierung)
  - flexibles Datenmodell:
    basiert auf der Verarbeitung von einzelnen Datensätzen
- Datenbankanfragesprachen (SQL) sind deskriptiv
  - wenig Funktionalität für die "alltägliche" Programmierung:
    Einfache Aufgaben wie z. B. die "Berechne den Flächeninhalt eines beliebigen Polygons" ist in SQL nicht möglich.
  - sehr komfortabel für die Datenbankprogrammierung
  - restriktives Datenmodell:
    basiert auf der Verarbeitung von Mengen (Relationen)

## Frage:

- Wie kann die Programmierung von Datenbankaufgaben mit der von "gewöhnlichen" Aufgaben verbunden werden (ohne dabei die Vorteile von SQL aufzugeben)?

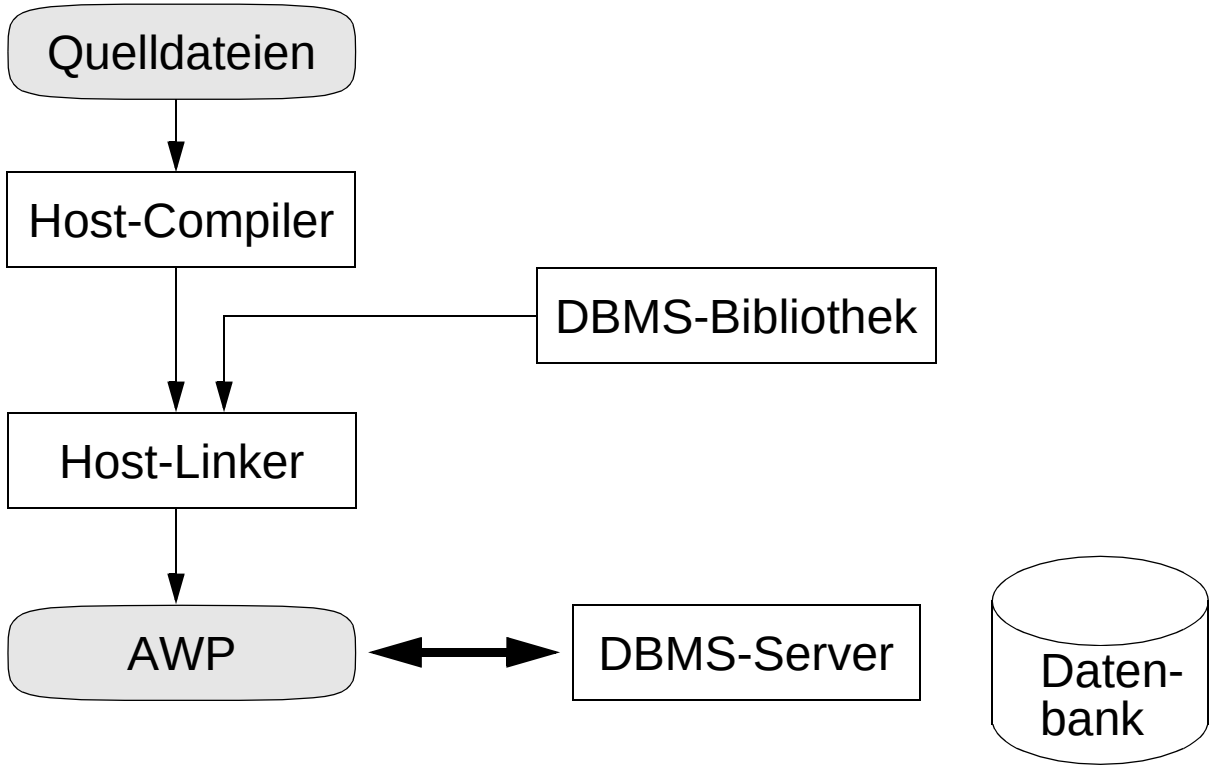
# Kopplungsmöglichkeiten

## Evolutionäre Ansätze:

- CALL-Schnittstelle
  - Bereitstellung von Bibliotheken
- Einbettung mit Vorübersetzer
  - statisch: Struktur der SQL-Befehle ist bereits vorgegeben
  - dynamisch: beliebige SQL-Befehle
- Spracherweiterungen
  - von SQL
  - von einer imperativen Programmiersprache (z. B. PASCAL)
- Skriptsprachen
  - BASIC-ähnliche Sprachen ohne Typkonzept
  - einfache Anbindung an Window- und Graphikumgebung

# 5.1 Prozedurale CALL-Schnittstelle

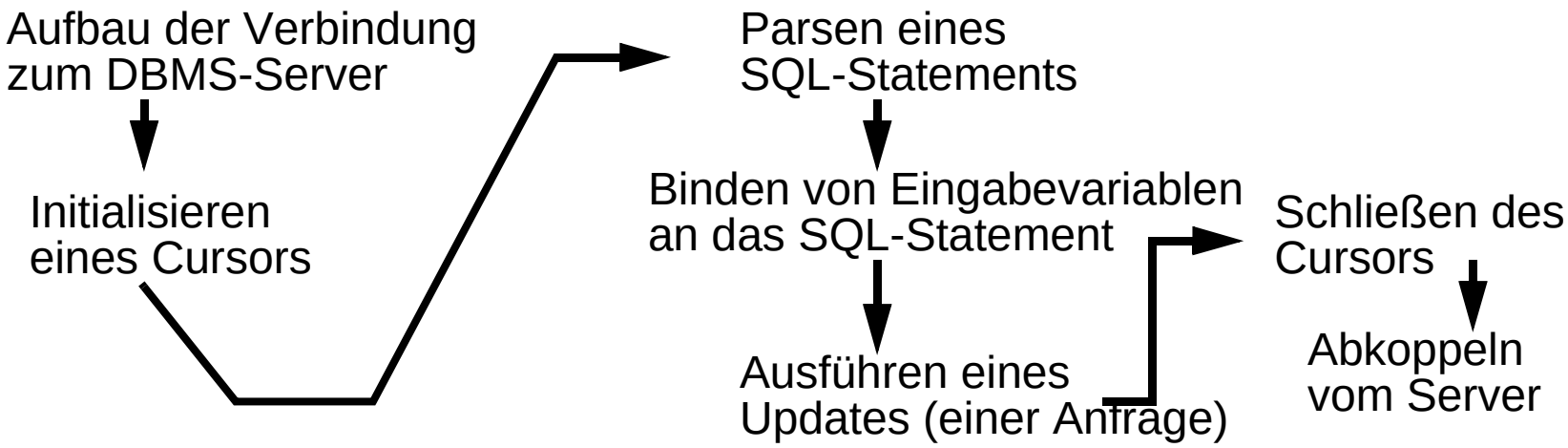
- basiert auf der Verwendung einer Bibliothek
  - im Fall von Oracle: Oracle Call Interface (OCI)
  - im Fall von SQL Server (ODBC)
  - …



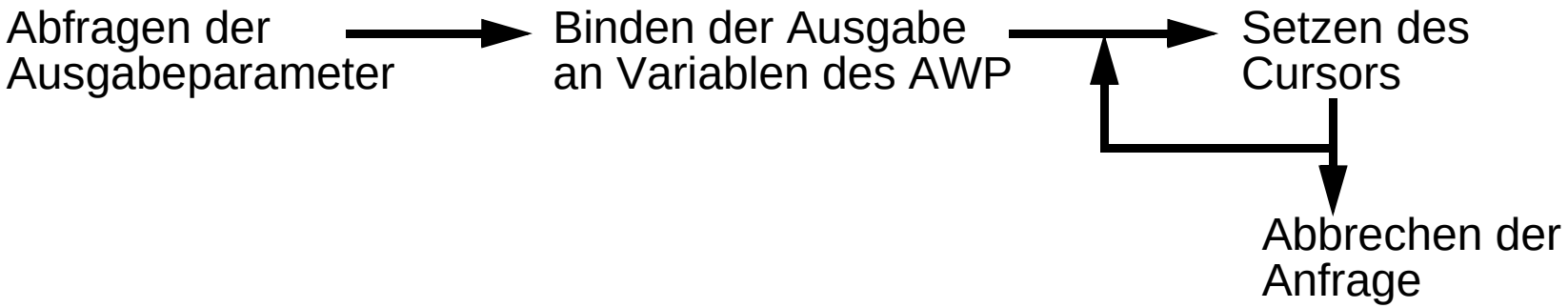
# Komponenten der CALL-Schnittstelle

- Gemeinsame vom AWP und dem Datenbank-Server genutzte Datenstrukturen
  - zum Aufbau der Kommunikation
  - zur Bearbeitung einer Anfrage
- **Cursor**: Im AWP benutzte Datenstruktur zum Zugriff auf Relationen der DB.
- SQL-Anfragen werden im AWP als String repräsentiert.
  - AWP muss einen erheblichen Teil der Typüberprüfung übernehmen.
- Binden der Variablen aus dem AWP an die Datenstrukturen des DBMS-Servers

## Ablauf eines AWP:



# Ausführen einer Anfrage (im Detail):




# Nachteil bei der Benutzung der CALL-Schnittstelle:

- komplizierte Programmierung
  - Abbildung zwischen den Objekten des AWP und den Objekten der Datenbank
- fehleranfällig
  - Typsystem der Programmiersprache und des Datenbanksystems sind unterschiedlich

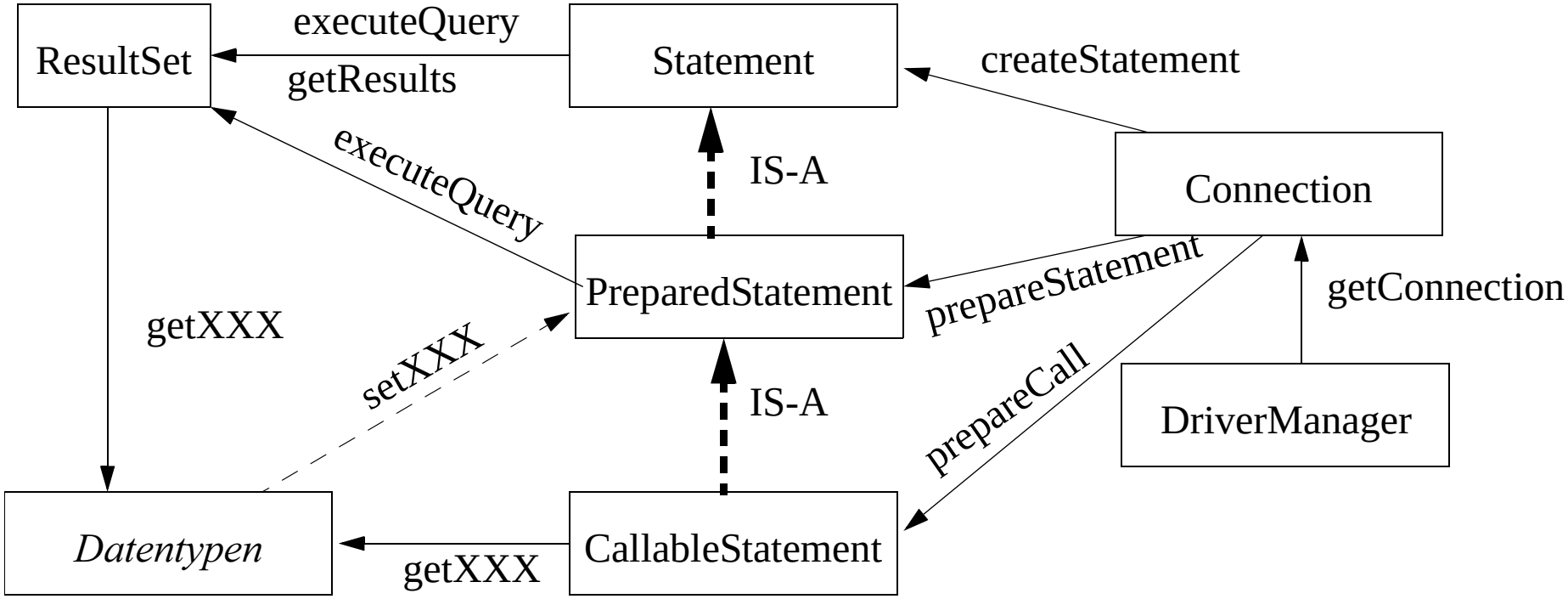
# Vorteile

- hohe Flexibilität
  - Generierung und Kompilieren von Anfragen zur Laufzeit des AWP

# 5.1.1 JDBC

- JDBC (**J**ava **D**ata**b**ase **C**onnectivity) ist eine Java API für die Programmierung relationaler DBMS
  - Anfragen werden im Java-Anwendungsprogramm als uninterpretierte Zeichenketten an das DBMS gegeben.
  - Resultate werden als Objekte einer Klasse *ResultSet* vom DBMS an das AWP geschickt.
- JDBC bietet eine einheitliche Schnittstelle für verschiedene DBMS
  - ähnlich zu ODBC (**O**pen **D**ata**B**ase **C**onnectivity), aber plattformunabhängig
  - JDBC ist unter dem Paket java.sql integraler Bestandteil der Standard API von Java
- Unterstützung des Client-Server Konzepts
  - DBMS läuft logisch auf einem anderen Rechner als das AWP
- Unterstützung wichtiger Konzepte
  - Verschiedene Typen von Anfragen (statische Anfragen, parameterisierbare Anfragen, Aufruf von gespeicherten Prozeduren)
  - Änderungsoperationen auf Relationen und dem Schema
  - Zugriff auf Metadaten

# Wichtige Klassen bei JDBC




- Die Namen in den Rechtecken entsprechen den Namen der verfügbaren Schnittstellen
- Die gerichteten durchgezogenen Kanten zeigen, wie man ein Objekt einer Klasse (auf die eine Kante gerichtet ist) mit Hilfe der anderen Klasse erzeugt.

# Client/Server Kopplung in JDBC

## Aufbau einer Verbindung

- Als erstes muss zunächst die entsprechende Treiberklasse angemeldet werden.

  Class.forName("oracle.jdbc.driver.OracleDriver") ;

- Danach wird ein Connection Objekt erzeugt.

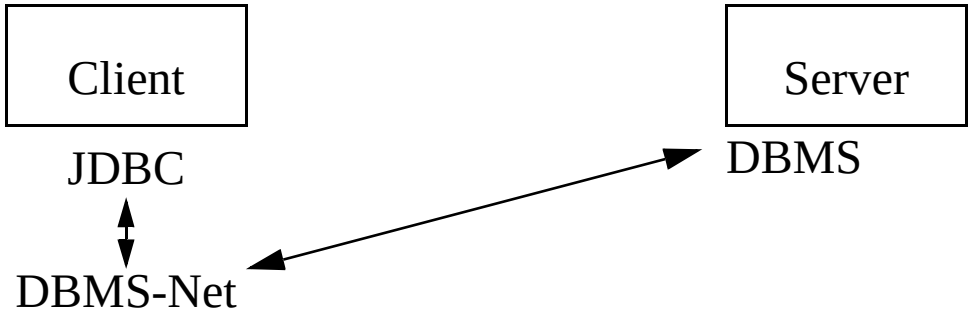
  Connection con = DriverManager.getConnection ("jdbc:oracle:thin:@venus.mathema-tik.uni-marburg.de:1521:Init_DB", "scott", "tiger");

  – Erste Zeichenkette entspricht einer URL zur Datenbank
  – Zweite Zeichenkette ist der Benutzername
  – Dritte Zeichenkette ist das Passwort

## 4 verschiedene Treiberklassen bei JDBC

1. Native API-Treiber
2. JDBC-ODBC Bridge
3. JDBC-Middleware
4. Natives Protokoll

# Native API-Treiber




Spezielle Netzsoftware des DBMS wird auf dem Client installiert. Methoden in JDBC werden nativ auf die Methoden der darunter liegenden Netzsoftware abgebildet.

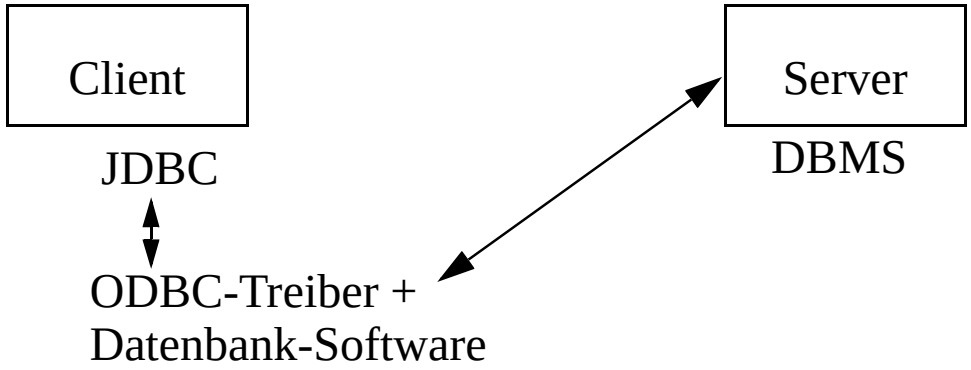
## Nachteile

- Installation der Netzsoftware auf allen Clients
  - keine Unabhängigkeit vom Datenbanksystem
  - Keine Nutzung im Internet

## Vorteil

- relativ gute Performance

# JDBC-ODBC Bridge




ODBC-Treiber und ggf. weitere Datenbank-Software sind auf dem Client verfügbar.
Methoden von JDBC werden auf die Funktionalität von ODBC abgebildet.

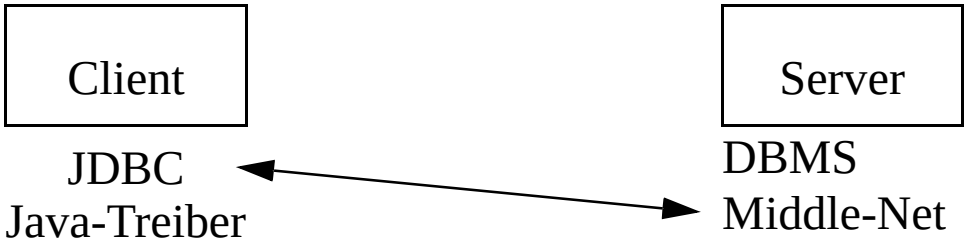
## Vorteil

- Unabhängigkeit von speziellen Datenbanksystemen, das ODBC sehr weit verbreitet ist.

## Nachteil

- Installation von ODBC + Datenbanktreiber auf allen Clients
  - keine Anbindung im Internet
- Anfragen und Ergebnisse müssen über die ODBC-Brücke verschickt werden.
  - schlechte Performance

# JDBC-Middleware




Ähnlich zur ersten Lösung, aber die Netzsoftware ist durch Nutzung einer Middleware unabhängig vom spezifischen DBMS.

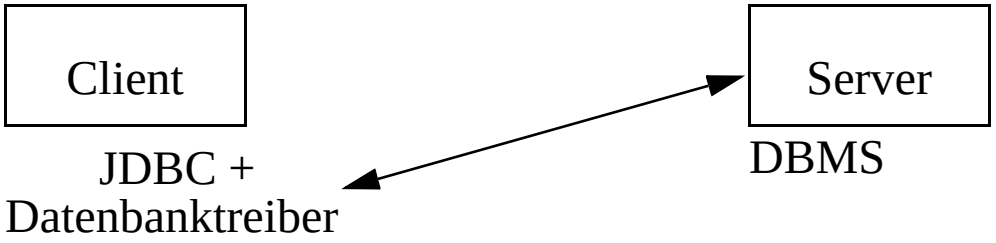
## Vorteil

- Keine Installation von spezieller Software am Client
  - Einfacher (und kleiner) Java-Treiber wird bei Bedarf geladen
- Unabhängigkeit vom DBMS
- Optimierung auf dem Server (wie z. B. Pufferung von Ergebnissen)

## Nachteil

- Alle Ergebnisse werden über die Middleware nach außen gegeben.

# Natives Protokoll




Ein speziell in Java implementierte Netzwerkverbindung übermittelt die Anforderungen des Clients an einen Server. Beim Server wird dann die Anforderung direkt an das DBMS weitergereicht.

## Vorteile

- Software auf dem Client ist in Java geschrieben und kann bei Bedarf geladen werden.
- Direkte Interaktion mit dem DBMS (Vermeidung einer Zwischeninstanz)
  - gute Performance

## Nachteil

- Für jedes DBMS benötigt man einen eigenen Treiber.

# Treibermanagement von JDBC

## Treibermanager

- Vermittlungsschicht zwischen Anwendung und dem eigentlichen Datenbanktreiber
- Es gibt nur einen Treibermanager pro Programm

  => nur statische Methoden in der Klasse **DriverManager**
- Unterstützung folgender Operationen
  - Registrieren und Laden von Treibern
  - Herstellen einer Verbindung zur Datenbank
  - Konfiguration der Verbindung
- Laden eines Treibers
  - Explizites Laden im AWP über einen Aufruf der Methode Class.forName
  - Über die Systemeigenschaft jdbc.drivers der JVM

    ```
    java -Djdbc.drivers=foo.bas.Driver:wom.sql.Driver JDBCDemo1
    ```

    Es wird das Programm JDBCDemo1 gestartet und in der Kommandozeile der verwendete Treiber mitgeteilt.
  - Beim Laden eines Treibers, registriert der Treiber sich beim Treibermanager (durch Aufruf der Methode *registerDriver* der Klasse DriverManager).

# Treiber

- Wichtig für die Praxis sind Methoden zum Abgreifen von Daten über die Version des Treibers und seinen Leistungsumfang.
  - Z. B. liefert die Methode *jdbcCompliant* der Schnittstelle Driver nur dann true, wenn der Treiber SQL92 Entry Level unterstützt und den JDBC-Test erfolgreich bestanden hat.

# Aufbau einer Datenbankverbindung

- erfolgt über den Aufruf einer Methode mit Namen *getConnection* aus der Klasse DriverManager.
  - Alle diese Methoden erwarten eine URL als Parameter, genauer gesagt eine JDBC-URL.
- Aufbau einer JDBC-URL
  jdbc:<subprotocol>:<subname>

# Weitere Funktionalität

- Setzen einer maximalen Wartezeit zur Anmeldung am Server
- Protokollierung von Fehlermeldungen

# 5.1.2 Datenverbindungen bei JDBC

- Eine Verbindung wird in JDBC durch ein Objekt der Klasse Connection repräsentiert.
  - getConnection liefert als Ergebnis ein solches Objekt zurück.
- Unterstützte Funktionalität
  - Senden von SQL-Anweisungen an den Datenbankserver
  - Steuerung von Transaktionen
  - Empfang von Resultaten vom Server
  - Abgreifen von Metainformation über die Datenbank

# Transaktionen

- Eine Transaktion ist eine Folge von SQL-Befehlen, die atomar verarbeitet werden.
  - Durch den Befehl **commit** wird die Transaktion abgeschlossen und die Änderungen sind dann danach garantiert in der Datenbank.
  - Durch den Befehl **rollback** wird die Transaktion wieder zurückgefahren, d. h. alle bisherigen Operationen in der Transaktion sind unwirksam.
- Eine Transaktion wird implizit mit dem ersten Befehl (nach commit und rollback) gestartet.
- Objektmethoden
  - *commit*() und *rollback*() sind Methoden mit der bekannten Semantik
  - void *setAutoCommit*(boolean enable)
    Ist AutoCommit gesetzt, ist jeder Befehl eine Transaktion. Ansonsten muss die Transaktion explizit durch commit bzw. rollback abgeschlossen werden.
  - Methode *setTransactionIsolation* erlaubt das Einstellen eines sogenannten Transaktionslevels. (Siehe auch Kapitel Transaktionen)

# 5.1.3 Anfragen

- In java.sql wird zwischen drei Anfragetypen unterschieden
  - Statement
    Einfache Anfragen ohne Parameterisierung
  - PreparedStatement
    Vorübersetzte Anfrage, die Eingabeparameter unterstützt.
  - CallableStatement
    Aufruf von gespeicherten Prozeduren, die auf dem DBMS hinterlegt sind.
- Zur Ausführung von allgemeinen SQL-Anfragen stehen folgende execute-Methoden zur Verfügung:
  - ResultSet *executeQuery*(String sql)
    Ausführung einer Anfrage mit SELECT-Klausel, wobei das Ergebnis ein Objekt der Klasse ResultSet ist.
  - int *executeUpdate*(String sql)
    Dadurch können Änderungsoperationen (Einfügen neuer Tupel, Löschen, …) unterstützt werden. Als Ergebnis wird die Anzahl der involvierten Tupel geliefert. Weiterhin können auch Schemaänderungen formuliert werden (z. B. create table, …). Dann ist das Ergebnis stets 0.
  - boolean *execute*(String sql)
    Diese Methode wird dann aufgerufen, wenn es nicht bekannt ist, ob die SQL-Anweisung eine Anfrage ist (Rückgabewert true) oder eine Änderungsoperation

(false). Auch kann diese Methode verwendet werden, wenn eine Anweisung mehr als ein Resultat besitzt.

# Inkompatibilität von SQL-Dialekten

– Ein primäres Ziel von JDBC ist die Unabhängigkeit des AWP vom zugrunde liegenden DBMS. Leider gibt es eine Vielzahl von Unterschieden bei den verschiedenen SQL-Dialekten.

Outer-Join in SQL92 Standard

```
select Emp.name, Proj.name
from Emp left outer join Proj on Emp.pid = Proj.id
```

Outer-Join in SQL von Oracle

```
select Emp.name, Proj.name
from Emp, Proj
where Emp.pid = Proj.id(+)
```

– Zur Überwindung solcher Abhängigkeiten gibt es in JDBC eine **Escape-Klausel**. Diese Klausel wird vom Treiber erkannt und dann in den spezifischen Dialekt des DBMS transformiert. Eine Escape-Klausel besteht aus einem Schlüsselwort und einer Menge von Parametern.
{<keyword> …}

Beispiel: Outer-Join mit Escape-Klausel

Outer-Join mit Escape-Klausel

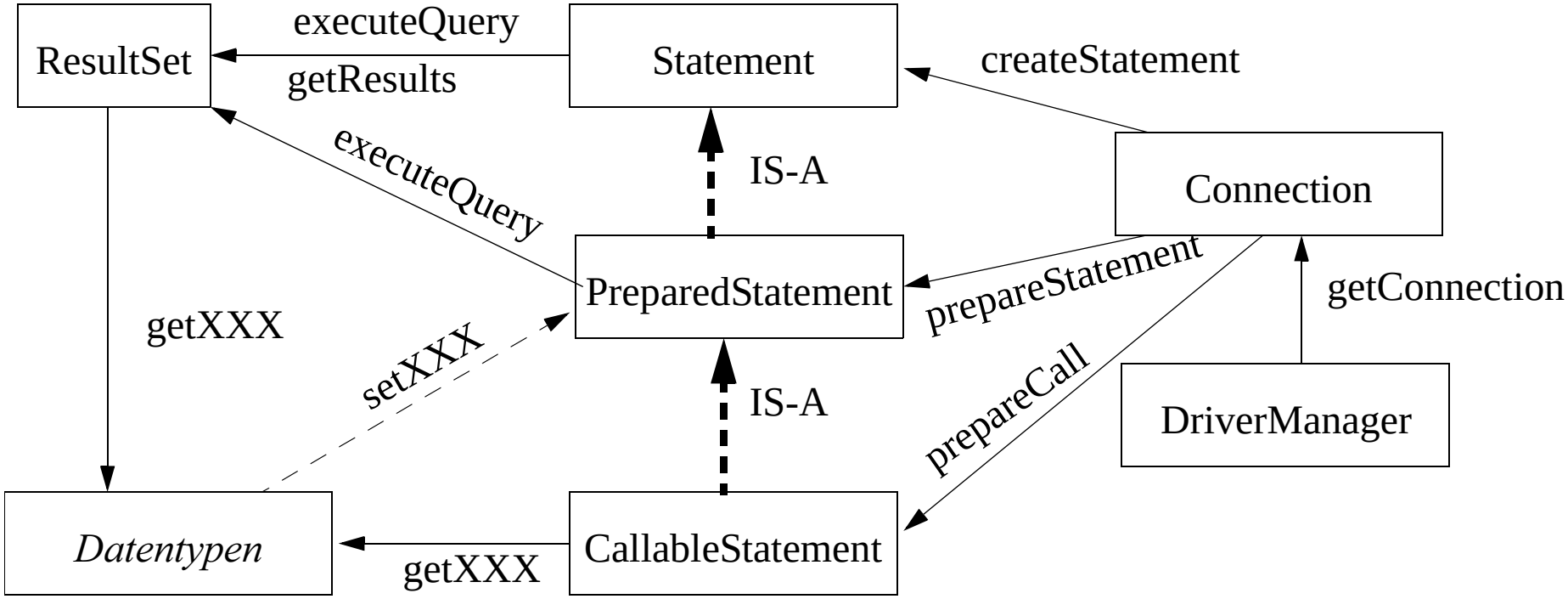
```
select Emp.name, Proj.name
from {oj Emp left outer join Proj
on Emp.pid = Proj.id}
```

- Der Funktionsumfang von Escape-Klauseln hängt stark vom jeweiligen Treiber ab!
- Ein bekanntes Beispiel für Inkompatibilitäten ist die Schreibweise für Datumskonstanten. Mit Escape-Klausel kann folgende Schreibweise für den 6.1.2003 benutzt werden: {d ´2003-01-6´}

## Ergebnisgröße und Laufzeiten von Anfragen

- Mit der Methode *setMaxRows* kann die maximal erwünschte Anzahl von Ergebnissen spezifiziert werden und mit *setQueryTimeout* die maximal akzeptierte Ausführungszeit.

# Wichtige Klassen bei JDBC




- Die Namen in den Rechtecken entsprechen den Namen der verfügbaren Schnittstellen
- Die gerichteten durchgezogenen Kanten zeigen, wie man ein Objekt einer Klasse (auf die eine Kante gerichtet ist) mit Hilfe der anderen Klasse erzeugt.

# PreparedStatement

- Ein PreparedStatement ist eine Spezialisierung eines Statements, wobei die SQL-Anweisung beim Erzeugen des Statements (bis auf Parameter) vorliegen muss.
  - Dieses Statement wird bereits bei der Erzeugung übersetzt und kann ohne Neuübersetzung beliebig oft ausgeführt werden.
  - Eingabeparameter sind erlaubt, die erst bei Ausführung mit Werten belegt werden.
- Bei der Definition eines PrepapredStatement werden die Eingabeparameter jeweils durch ein Fragezeichen markiert.
  - PreparedStatement stmt = con.prepareStatement("select x, y from Points where x < ? and x > ?");
- Vor der Ausführung einer Anweisung müssen die Parameter durch set-Methoden gesetzt werden. Für jeden Typ gibt es eine spezielle Methode.
  - stmt.setInt(1, 20); stmt.setInt(2, 10);
    // Erste Parameter der set-Methoden ist die Stelle des Parameters in der
    // Anweisung.
- Zur Übergabe von NULL-Werten und großen Objekten gibt es spezielle Methoden:
  - void setNull(int stelle, int jdbcType);
    // Der zweite Parameter identifiziert ein Typ, siehe java.sql.Types
  - void setAsciiStream(int stelle, java.io.InputStream s, int len);

# Ergebnisse von Anfragen

- Anfragen mit Select-Klausel liefern als Ergebnis ein Objekt der Klasse ResultSet.
  - Repräsentation einer Menge von Tupeln mittels eines Cursors.
- Methode *next*() liefert beim erfolgreichen weitersetzen des Cursors true, wobei zu Beginn der Cursor vor dem ersten Tupel steht (Ein Aufruf von next ist deshalb vor dem ersten Zugriff zwingend erforderlich).
- Zugriff auf die Spalten erfolgt über get-Methoden, wobei es für jeden Typ int, double, … jeweils eine solche Methode gibt.
  - Parameter der get-Methode ist die Position der Spalte in der Relation.
  - Eine Methode *findColumn*(String str) liefert zu einem Attributnamen die Position.
  - Eine weitere Besonderheit von JDBC ist die Methode *getObject*, die zu dem SQL-Typ einen passenden Java-Objekttyp generiert.
- Null-Werte erfordern wiederum eine Sonderbehandlung.
  - boolean *wasNull*()
    überprüft, ob das zuletzt abgefragte Attribut tatsächlich ein Nullwert war.
  - Diese Methode muss nur in folgenden Fällen aufgerufen werden.
    a) Wenn die get-Methode den Wert null liefert.
    b) Wenn *getBoolean* den Wert false liefert.
    c) Wenn *getInt* und *getDouble* den Wert 0 liefert.

# 5.1.4 Metadaten

- Durch Ausnutzung der Metadaten der zugrunde liegenden Datenbank können Anwendungen erheblich flexibler gestaltet werden.
- JDBC bietet folgende Schnittstellen für Metadaten an
  - *ResultSetMetaData* zur Beschreibung der Relation, die das Objekt der Klasse ResultSet verwendet.
  - *DatabaseMetaData* zur Beschreibung der eigentlichen Datenbank

## ResultSetMetaData

- ist insbesondere dann von Vorteil, wenn der Benutzer nicht genau weiß, welche der get-Methoden bei einem Objekt rs der Klasse ResultSet tatsächlich angewendet werden sollen.
- Abhilfe schafft dann der Aufruf rs.getMetaData(), der zu rs die Metadaten liefert.
- Damit kann z. B. die Anzahl der Spalten, der Typ und Name einer Spalte abgefragt werden.
- Beispiel:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM Personal");
ResultSetMetaData rsm = rs.getMetaData();
for (int i = 1; i <= rsm.getColumnCount(); i++) {
        int ct = rsm.getColumnType(i);
```

```
        String cn = rsm.getColumnName(i);
        String ctn = rsm.getColumnTypeName(i);
        System.out.println("Column #" + i + ": " + cn + " of type " + ctn +
                                " (JDBC type: " + ct + ")");
    }
```

# DatabaseMetaData

- Dient zur Bereitstellung von Information über
  - das Schema der Datenbank
  - die Eigenschaften des DBMS
- Funktionalität
  - Methoden zur Beschreibung von Datenbankobjekten, wie z. B. *getTables*, *getColumns*, …
  - Methoden zum Abfragen wichtiger Systemkonstanten, wie z. B. *getMaxRowSize*, *getMaxConnection*, …
  - Methoden zum Abfragen von Features des DBMS, wie z. B. *supportsUnion*, *supportsANSI92FullSQL*, …
  - Methoden zu Informationen über die Datenbank, wie z. B. *getURL*, *getSQLKeyWords*, …

# 5.1.5 Ausnahmebehandlung

- Wie üblich in Java werden ungewöhnliche Situationen durch Verwendung des Exception-Konzepts behandelt. Hierfür gibt es eine eigene Klasse SQLException, auf die wir hier nur verweisen wollen.
- Zusätzlich können “unkritische” Fehler auch als Warnungen nach außen gegeben werden.
  - Warnungen sind Objekte von der Klasse SQLWarning bzw. von SQLWarning abgeleiteten Klassen.
  - Beispiel: Klasse *DataTruncation*
    Eine solche Warnung wird beim Lesen von Daten gegeben, wenn diese nicht vollständig eingelesen werden konnten.

# 5.1.6 Abbildung von SQL-Typen in Java

- Da das Typsystem der verschiedenen DBMS recht unterschiedlich ist, bietet JDBC zur Überwindung der Inkompatibilitäten in der Klasse java.sql.Types einige generische Typbezeichner an.
  - z. B. VARCHAR, LONGVARBINARY, INTEGER, DOUBLE, DATE, TIME
- Diese Typen (repräsentiert als ganzzahlige Konstanten) sind dann dort einzusetzen, wo SQL-Typen anzugeben sind. Die Umsetzung in die spezifischen Typen des zugrunde liegenden DBMS erfolgt automatisch über den Treiber.
- Ähnliche Vorgehensweise steckt hinter den Aufrufen der getXXX- und setXXX-Methoden.
  - Implizit erfolgt hierbei eine Konvertierung von einem Java-Typ in einen JDBC-Typ und damit durch den Treiber in einen Typ des zugrunde liegenden DBMS.
- Darüber hinaus ist zu beachten, dass die Datums- und Zeittypen von Java aus dem Package java.sql stammen.
  - Diese sind im Wesentlichen Wrapper-Klassen für die aus dem Package java.util.

# 5.1.7 Erweiterungen: JDBC 2 und JDBC 3

## Entwicklung von JDBC

- Die bisherige Beschreibung entspricht im Wesentlichen dem ursprünglichen JDBC-Standard, der aber bereits 1998 wesentlich erweitert wurde (JDBC 2).
- Derzeit unterstützen viele Treiber bereits diesen erweiterten Befehlsumfang, wobei ein noch weitergehender Standard (JDBC 3) bereits verabschiedet wurde und derzeit an JDBC 4 gearbeitet wird.

## Überblick der Erweiterungen

- Scrollen auf Objekten der Klasse ResultSet, Unterstützung von Änderungsoperationen
- Unterstützung von Typen aus SQL99
- Batch-Updates
- Weitere optionale Änderungen im Package javax.sql

# Erweiterungen von ResultSet

- Flexible Positionierung in einem ResultSet
  - vorwärts
  - rückwärts
  - direkt
- Dies hört sich zunächst relativ unspektakulär an, hat aber erhebliche Auswirkungen. Es werden jetzt mehre Typen von ResultSet angeboten
  - forward-only
    Das entspricht der zuvor beschriebenen Funktionalität von ResultSet.
  - scroll-insensitiv
    Dieser Typ unterstützt das durchlaufen in beliebigen Richtungen, wobei sich Änderungen auf den Datenbestand nicht auswirken.
  - scroll-sensitiv
    Dieser Typ unterstützt ebenfalls beliebiges Navigieren, aber Änderungen von anderen Transaktionen werden dabei sichtbar.
- Darüber hinaus wird noch unterschieden zwischen
  - read-only: keine Änderungen auf den Daten zulässig.
  - updatable: Änderungen, Löschen und Einfügen wird unterstützt.

- **Erzeugen** von scrollbaren Objekten der Klasse ResultSet erfolgt über eine neue Methode, die zusätzlich zwei Parameter umfasst:
  - resultSetType (vom Typ int) spezifiziert den jeweiligen Typ
  - resultSetConcurrency (vom Typ int) gibt an, ob das Objekt nur lesbar oder auch änderbar ist.

  Beispiel:

```
Statement stmt = con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
                                      ResultSet.CONCUR_READ_ONLY);
ResultSet srs = stmt.executeQuery("SELECT Name FROM Personal");
```

- Dabei ist zu beachten, dass einige Treiber nicht die komplette Funktionalität unterstützen. Um sicher zu gehen, muss man auf die Methode
  boolean supportsResultSetType(int resultSetType)
  aus der Klasse java.sql.DatabaseMetaData zurückgreifen.

# Scrollen

- Initiales Positionieren eines ResultSet
  - void *beforeFirst*() // Cursor ist vor dem ersten Element
  - void *afterLast*() // Cursor ist hinter dem letzten Element
  - boolean *first*() // Cursor ist auf dem ersten Element
  - boolean *last*() // Cursor ist auf dem letzten Element
- Navigieren auf einem ResultSet
  - Neben der Methode boolean *next*() gibt es noch die Methode
    boolean *previous*() // Setzt den Cursor auf das Vorgängerelement.
  - Darüber hinaus besteht die Möglichkeit den Cursor direkt auf eine absolute Adresse
    boolean *absolute*(int row)
    und auf eine Adresse relativ zum aktuellen Tupel
    boolean *relative*(int row)
    zu setzen. Dabei ist zu beachten, dass das erste Element die Position 1 besitzt.
- Abfragen von der Cursorposition
  - Hierzu kann man direkt *int getRow()* aufrufen.
  - Zusätzlich gibt es noch Methoden *boolean isBeforeFirst()*, *boolean isAfterLast()*, *boolean isFirst()*, *boolean isLast()*.

# Änderungsoperationen

- Bei Änderungsoperationen auf ResultSet ergibt sich die gleiche Problematik wie bei Sichten (Views), dass diese nämlich nur sehr eingeschränkt möglich sind.
  - Die Anfrage beinhaltet nur eine Relation und keine Verbundoperation.
  - Der Primärschlüssel ist Bestandteil der Ergebnisrelation.
  - Alle Attribute, die nicht NULL-Werte zulassen und keine Defaultwerte besitzen, sind ebenfalls Bestandteil der Ergebnisrelation.
- Änderungen (Update eines bestehenden Tupels)
  - Diese Operation bezieht sich stets auf das aktuelle Tupel des ResultSet.
  - Spaltenwerte können über die updateXXX-Methoden geändert werden.
    rs.*updateDouble*(“price”, 9.99);
  - Nach den Änderungen auf einem Tupel muss noch die Methode
    rs.*updateRow*();
    gerufen werden, um die Änderung tatsächlich in die Datenbank zu übertragen.
- Einfügen eines neuen Tupels
  - Jedes ResultSet hat eine Insert-Row-Zeile. Diese kann nach dem Update-Verfahren vorbereitet und danach eingefügt werden.
  - Zuerst muss man an die richtige Stelle positioniert werden
    rs.*moveToInsertRow*();

- Danach können über die updateXXX-Methoden die Attributwerte des neuen Tupels eingegeben werden.
- Schließlich muss noch durch den Aufruf
  rs.*insertRow*();
  das neue Tupel übertragen werden.
- Ein Aufruf der Methode *moveToCurrentRow* sorgt dafür, dass der Cursor auf der vorherigen Position steht.

- Löschen eines Tupels
  - Zunächst muss auf das zu löschende Tupel navigiert werden. Danach kann durch Aufruf der Methode
    rs.*deleteRow*()
    das Tupel gelöscht werden.

# Sichtbarkeit von Änderungen

- Damit kann man steuern, ab welchem Zeitpunkt eine Änderung im AWP tatsächlich sichtbar wird.
  - sofort nach der Änderung
  - am Ende der Transaktion

# Techniken zur Leistungssteigerung

## Problem

- Tupelweiser Zugriff auf ResultSet erfordert bei jedem next-Aufruf Kommunikation mit dem DBMS-Server.
- Änderungsoperationen können nur einzeln an das DBMS übergeben werden.

## Prefetching von Resultaten

- Statt nur das nächste Tupel werden die nächsten k Tupel vom DBMS-Server angefordert. Durch Aufruf der Methode void *setFetchSize*(int rows) wird die Anzahl der Ergebnisse pro Anforderung spezifiziert.

## Batch-Updates

- Bündelung von mehreren Änderungsoperationen, die gemeinsam an die zugrunde liegende Datenbank geschickt werden.
- Hierzu gibt es folgende Methoden
  - void *addBatch*(String sql); // Fügt eine neue Änderungsoperationen zum Batch
  - int[] *executeBatch*(); // Führt einen Batch aus, wobei als Ergebnis die // Anzahl der betroffenen Tupel geliefert wird.

# Beispiel

- Wir betrachten eine Relation Emp(Name,Gehalt).
- Sourcecode

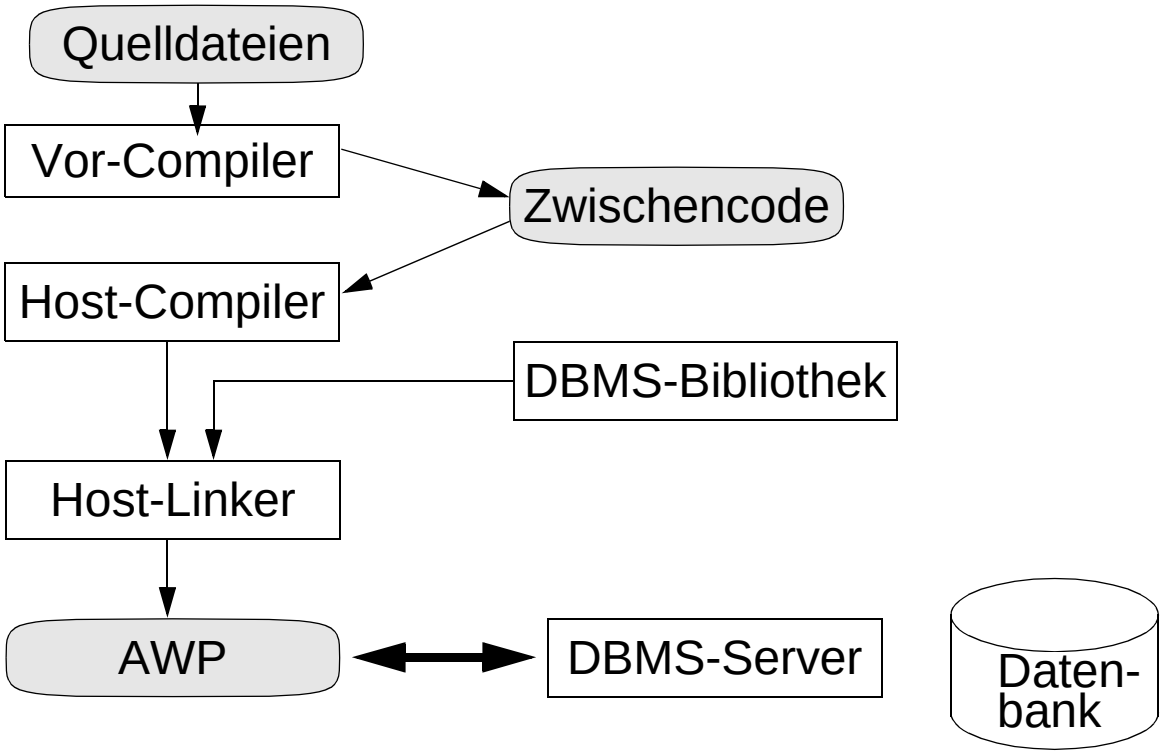
```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO Emp " + "VALUES('Müller', 3000)");
stmt.addBatch("INSERT INTO Emp " + "VALUES('Schmidt', 4000)");
stmt.addBatch("INSERT INTO Emp " + "VALUES('Becker', 3500)");
stmt.addBatch("INSERT INTO Emp " + "VALUES('Bauer', 900)");
int [] updateCounts = stmt.executeBatch();
con.commit();
con.setAutoCommit(true);
```

# 5.1.8 Zusammenfassung

- JDBC hat sich als relativ leicht zu nutzende Call-Schnittstelle für die Erstellung von AWP erwiesen.
  - Anfragen werden in Objekte der Klasse Statement gepackt
  - Resultat einer Anfrage steht als Objekt der Klasse ResultSet zur Verfügung
- Flexible Programmierung durch Unterstützung von Metadaten
- JDBC wurde in den letzten Jahren um weitere Funktionalität erweitert.
  - Insbesondere zur flexibleren und effektiveren Nutzung von ResultSet.
  - Unterstützung des neuen SQL-Standards (bisher noch nicht behandelt)
  - Weitere Erweiterungen sind optional und sind in dem Package javax.sql zu finden.
- Nachteile von JDBC
  - Abbildung der Typen zwischen Java und dem JDBC-Treiber (indirekt also dem DBMS) muss vom Benutzer vorgenommen werden.
  - Viele Fehler treten erst zur Laufzeit auf.
  - So genannte JDBC-konforme Treiber unterstützen nur einen relativ geringen Befehlsumfang. Damit sind oft Applikationen abhängig von dem jeweiligen Treiber (alternativ könnten auch die Eigenschaften eines Treibers erst erfragt werden, was aber die Programmierung erheblich erschwert).

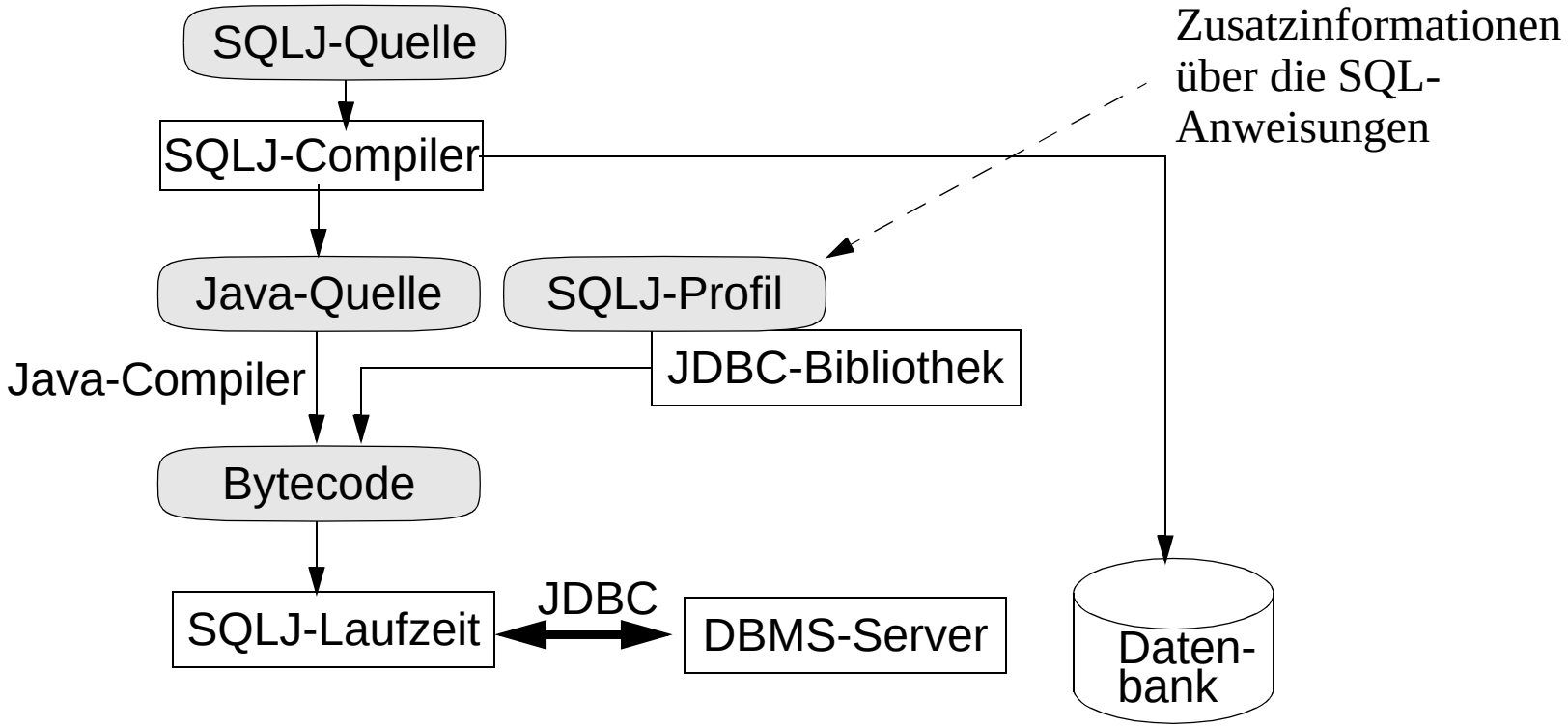
# 5.2 Embedded SQL (eSQL, SQLJ)

- basiert auf der Verwendung eines Vorübersetzers
- Statische Festlegung der Datenbankoperationen zur Übersetzungszeit
  - Typenüberprüfung zwischen AWP und Datenbank durch den Vorübersetzer
  - einfache Übertragung von Daten aus der Datenbank ins AWP
- Verwendung des Cursor-Prinzips zum Durchlaufen von Relationen



# SQLJ

- Embedded SQL wurde zunächst für die Programmiersprache C entwickelt.
- Auf Grund der großen Akzeptanz von Java und JDBC haben führende Datenbankhersteller (IBM, Oracle, Sybase) einen entsprechenden Ansatz für Java unter dem Namen SQLJ entwickelt.
- SQLJ nutzt dabei typischerweise JDBC zur Codegenerierung.



# Syntaktische Kennung von Datenbankoperationen im SQLJ-AWP:

- #sql {*SQL-Befehl*}
- Ein SQL-Befehl bezieht sich auf die Objekte der Datenbank. Eine Ausnahme sind die sogenannten **Hostvariablen**, die zur Datenübertragung zwischen der Datenbank und dem AWP genutzt werden.
  - Eine Hostvariable wird wie eine gewöhnliche Variable von Java deklariert und genutzt.
  - Eine Hostvariable kann auch in einem SQL-Kommando genutzt werden, wobei dann dem Variablennamen ein ":" vorgestellt ist.
- Man kann auch Hostvariablen dazu nutzen, Ergebnisse einer Anfrage aufzunehmen, wobei dann nur ein Ergebnis geliefert werden darf.
  - select-Klausel wird um das Schlüsselwort *into* gefolgt von Hostvariablen ergänzt.

Beispiele

- #sql {select A, B from R where B > :x}

  Hier wird der Wert der Variable x in den SQL-Befehl eingesetzt.
- #sql {select A, B into :a, :b from R where Id = 7}
  - Das Ergebnis wird nun in die Hostvariablen a und b übertragen (Ann.: Id ist Schlüsselkandidat).
  - Die Typen der Hostvariablen müssen zu den SQL-Typen kompatibel sein.

# Öffnen einer Datenbankverbindung

- Ähnlich wie bei JDBC braucht man in SQLJ einen Bezug zu einer bestehenden Datenbank. Aus diesem Grund kann nun ein Kontext definiert werden:

  #sql context verbinden

  Danach kann nun *verbinden* wie eine Klasse benutzt, die insbesondere folgenden Konstruktor besitzt:

  verbinden verbindungsObject = new verbinden("jdbc:oracle:thin:@venus.mathematik.uni-marburg.de:1521:Init_DB", "scott", "tiger");

  Dieses Kontextobjekt kann nun bei einem sql-Befehl noch zusätzlich angegeben werden:

  #sql (verbindungsObject) {select A, B from R where B > :x}

- Wird bei einem sql-Befehl kein Kontextobjekt angegeben, so wird ein Defaultkontextobjekt genutzt, der zuvor aber erzeugt werden muss:
  - DefaultContext ctx = new DefaultContext(DriverManager.getConnection(…));

# Anfrageformulierung mit Iteratoren

- Für SQL-Anweisungen, die mehr als eine Antwort liefern, können nun Iteratoren (Cursor) definiert werden. Es wird unterschieden zwischen positionsbezogenen und namenbezogenen Iteratoren.
- Positionsbezogene Iteratoren (Beispiel):
  - Deklaration:

    ```
    #sql public iterator Pos (String, int);
    ```

    Damit wird ein Iteratortyp *Pos* mit zwei Komponenten definiert.
  - Danach kann man eine Variable des Typs deklarieren:

    ```
    Pos x;
    ```
  - und ein sql-Befehl an diese Variable binden:

    ```
    #sql x = {select A, B from R where B > 10};
    ```
  - Der Zugriff auf die Ergebnismenge erfolgt dann i. A. in einer Schleife:

    ```
    while (true) {
        #sql {fetch :x into :a, :b}; // impliziter Aufruf von next
        if (x.endFetch()) break;
        System.out.println(a + " verdient " + b + " €");
    }
    ```

- Freigabe der Ressourcen:
  x.close();

- Namenbezogene Iteratoren
  - Deklaration:
    #sql public iterator Name (String A, int B);
  - Deklaration einer Variablen
    Name y;
  - Anbinden an einen SQL-Befehl:
    #sql y = {select B, A, C from R where B > 10};
    Die Ergebnisrelation muss alle im Attribut deklarierten Attribute enthalten!
  - Zugriff auf die Ergebnismenge:
    while (y.next())
    System.out.println(y.A() + " verdient " + y.B() + " DM");
    Man beachte hierbei, dass der Zugriff auf die Werte über Methodenaufrufe erfolgt, wobei der Name der Methode dem des Attributs entspricht.

    Mit der Methode *next* wird auf das nächste Tupel zugegriffen.

# Mengenwertige Änderungs- und Löschoperationen

- Solche Operationen verwenden auch Iteratoren, wobei die zu ändernde (löschende) Datenmenge an den Iterator gebunden wird.
- Danach können folgendermaßen die Änderungen vorgenommen werden.

```
#sql public iterator Name implements sqlj.runtime.ForUpdate (String A, int B)
// Hierbei muss der Iterator die oben genannte Schnittstelle implementieren.
…
Name y;
…
#sql y = {select A, B from R where B > 10};
…
while (y.next())
        #sql {update R set B = B + 10 where current of :y}
```

Damit wird nun der derzeit angesprochene Datensatz der Menge geändert.

- Zusätzliche Eigenschaften eines Iterators können bei der Deklaration durch eine with-Klausel gesetzt werden.
  - #sql public iterator Name implements sqlj.runtime.ForUpdate (String A, int B) with (sensitivity=SENSITIVE);

# 5.3 Prozedurale Erweiterung von SQL

- Neben der Einbindung der SQL-Funktionalität in einer imperativen Programmiersprache, besteht auch die Möglichkeit, SQL um prozedurale Konzepte zu erweitern.
  - Dies ist bereits als Anhang zum SQL92-Standard unter dem Namen SQL/PSM (Persistent Stored Modules) berücksichtigt.
  - Oracle hat unter den Namen PL/SQL eine solche Erweiterung bereits frühzeitig definiert, die im industriellen Umfeld zur Entwicklung von AWPs genutzt wird.
  - Oracle und andere Anbieter (IBM) bieten inzwischen an, in Java codierte Prozeduren im DBS ablaufen zu lassen.
- Primäres Ziel der prozeduralen Erweiterung von SQL
  - Bündelung von mehreren SQL.Befehlen unter einer aufrufbaren Einheit, die in der Datenbank persistent abgespeichert wird.
- Vorteile zu bisherigen Ansätzen
  - Leistungsverbesserung, da die Kommunikation zwischen Client und Server erheblich reduziert werden kann.
  - Zentrale Verwaltung von gemeinsam genutzter Funktionalität
  - Ausführung der Prozeduren ist unabhängig von jeweiligen Client.

# Vorgehensweise

- Implementierung einer Prozedur oder Funktion
- Installation der Prozedur auf dem Server
- Registrierung der Prozedur auf dem Server
  - Hierzu benutzt man die SQL-Operationen create procedure bzw. create function.
- Aufruf der Prozedur vom Client
  - Dies geschieht indirekt über das DBMS.

# 5.3.1 PL/SQL

- PL/SQL ist eine von Oracle entwickelte Erweiterung von SQL, die primär zur Erstellung von gespeicherten Prozeduren entwickelt wurde.
  - Syntax orientiert sich an die Programmiersprache Ada
- PL/SQL bietet insbesondere die Möglichkeit Prozeduren zu deklarieren und diese in SQL aufzurufen.

## Vorteile gegenüber einer Hostsprache

- homogene Anbindung der imperativen Konzepte an SQL
- Typkonvertierungen entfallen
- plattformunabhängige Ausführung

## Nachteil:

- Imperative Konzepte sind für eine vollständige Entwicklung von AWP nicht ausreichend.

# PL/SQL Block

- Ein PL/SQL-Block besteht aus drei Teilen:
  - einem optionalen Deklarationsteil, wo Variablen und Objekte deklariert werden,
  - einem ausführbaren Teil, in dem der Zustand der Variablen manipuliert wird,
  - einem optionalen Ausnahmebehandlungsteil, in dem Ausnahmen und Fehler zur Laufzeit behandelt werden.
- Definition eines Blocks:

[<header> is]

[declare <declarations>]

begin

<statements>

[exception <exceptions>]

end

# Deklarationsteil

- Typdeklarationen
  - Neben den SQL-Typen, können Variablen PL/SQL Typen wie boolean haben.
  - PL/SQL unterstützt insbesondere auch die Definition von Record-Typen:

    type PersonTyp is record (Name varchar(50), Salary int);
  - Weiterhin erlaubt PL/SQL auch noch indirekt über Relationen und Variablen erzeugte Typen.
- Variablendeklarationen
  - Eine Variable wird ähnlich wie bei SQL durch Nachstellen des Typs deklariert:

    ang PersonTyp;
  - Eine Besonderheit ist insbesondere, dass die Datentypen der Relationen bei der Variablendeklaration benutzt werden können:

    ang Personal%rowtype;
  - Entsprechend wird durch

    ang2 ang%type;

    eine Variable vom Typ der Variable *ang* deklariert.
  - Ähnlich wie in Pascal erstreckt sich der *Gültigkeitsbereich einer Variablen* auf den lokalen Block und alle Blöcke, die in diesem enthalten sind.

- Die Grundlegende Operation für Variablen ist die Zuweisung. Das in PL/SQL verwendete Symbol ist :=.

- Cursordeklarationen

  Ein Cursor unterstützt die sequentielle Verarbeitung einer Datenmenge.

  - konstanter Cursor
    cursor AlleAng is
    select *
    from Personal;
  - parametrisierter Cursor
    cursor interessantePersonen(von int, bis int) is
    select *
    from Personal
    where Lohn > von and Lohn < bis;

- Cursor und ihre Attribute
  Gewisse Eigenschaften von Cursorn können zur Laufzeit durch Anhängen eines Suffix abgefragt werden:
  - %found: Der letzte Fetch-Befehl auf dem Cursor war erfolgreich.
  - %isopen: Der Cursor ist geöffnet
  - %rowcount: Anzahl der Tupel des Cursors

# Kontrollstrukturen

## Imperative Ablaufsteuerung

- bedingte Anweisung
  if <Bedingung> then <PL/SQL-Anweisung> end if;
- for-Schleife
  for <IndexVariable> in <Bereich>
  loop
  <PL/SQL-Anweisung>
  end loop;

## Verarbeitung eines Cursors

- Öffnen eines Cursors
  open AlleAng;
  open interessantePersonen(1000, 2000);
- Verarbeitung einer Antwortmenge
  - spezielles Schleifenkonstrukt für einen Cursor
    for ang in AlleAng
    loop
    <PL/SQL-Anweisung>
    end loop;

# Prozeduren und Funktionen

- Eine Prozedur ist ein Block, der mit einem Namen versehen ist und optional über eine Parameterleiste verfügt.
- Eine Funktion liefert durch den Befehl return stets ein Ergebnis.

```
function totalSalary(int von, int bis) return int is
begin
    declare p Personal%rowtype;
    int total;
    open interessantePersonen(von, bis);
    …
    return total;
end
```

- Die Parameter können mit drei verschiedenen Optionen versehen sein: in, out, in out

```
procedure run (par1 in Typ1, par2 out Typ2, par3 in out Typ3, …) is
    <PL/SQL-Anweisungen mit Zuweisungen an die OUT-Parameter>
```

# Gespeicherte Prozeduren

- Mittels des Befehls create procedure können Prozeduren im Datenbanksystem in übersetzter Form abgespeichert werden.
  - Dies hat insbesondere den Vorteil gegenüber dem bisherigen Ansatz von PL/SQL, dass die Anweisungen nicht mehr übersetzt werden müssen.
- Die Deklaration einer Prozedur (Funktion) folgt dem bereites vorher erläuterten Muster

## Cursor-Variablen

- Es ist oft günstig die Ergebnisse einer gespeicherten Prozedur durch Cursor-Variablen an das aufrufende PL/SQL-Programm zu geben.
- Eine Cursor-Variable ist eine Referenz auf eine Liste von Datensätzen.
- Es wird zwischen folgenden Typen von Cursor-Variablen unterschieden:
  - starker Typ:
    type personenCurTyp is ref cursor Personal%rowtype;
  - schwacher Typ:
    type allCurTyp is ref cursor;

  Die Variablendeklaration wird wie üblich vorgenommen.
- Zum Zeitpunkt der Deklaration hat die Cursor-Variable noch keinen Bezug zu einer Anfrage.

- Anbindung einer Cursor-Variable an eine Anfrage
  - erfolgt erst beim Öffnen eines Cursors:
    open personenCurTyp for
    select * from Personal where Lohn > 1000;
- typische Verwendung
  - Öffnen einer Cursor-Variablen in der gespeicherte Prozedur/Funktion
  - Übergeben des Cursors an das AWP, wo dann die Datensätze verarbeitet werden.

Trotz der vielen Vorteile, die Cursor-Variablen bieten, gibt es derzeit bei der Nutzung in Oracle noch sehr viele Einschränkungen:

- Eine Cursor-Variable darf nicht im Update-Modus geöffnet werden.
- Der Typ *ref cursor* gibt es nur in PL/SQL, aber nicht in SQL.

# Beispiel 1

- Erhöhe das Gehalt der Angestellten um 10%, die bisher unter dem Durchschnitt verdient haben. Gib das aktuelle Durchschnittsgehalt, falls dieses 50000 überschreitet.

```
declare
    dGehalt number;
begin
    select avg(Lohn) into dGehalt from Personal;
    update Personal set Lohn = Lohn*1.1 where Lohn < dGehalt;
    select avg(Lohn) into dGehalt from Personal;
    if dGehalt > 50000 then
        dbms_output.put_line("Durchschnitt: " || dGehalt);
    end if;
    commit;
exception
    when others then dbms_output_line("Fehler beim Update");
    rollback;
end;
```

# Beispiel 2

- Berechne die Gehaltserhöhung der Angestellten in Abhängigkeit ihres bisherigen Gehalts.

```
declare
    cursor angCursor is select Lohn from Personal for update of Lohn;
    angNrinteger;
    angGehalt Personal.Gehalt%type;
begin
    open angCursor;
    fetch angCursor into angGehalt;
    while angCursor%found
        if angGehalt > 60000
            update Personal set Lohn = Lohn*1.1 where current of angCursor;
        elsif angGehalt > 50000
            update Personal set Lohn = Lohn*1.15 where current of angCursor;
        else
            update Personal set Lohn = Lohn*1.2 where current of angCursor;
```

```
            end if;
            fetch angCursor into angGehalt;
        end loop;
end;
```

# Gespeicherte Prozeduren in JDBC

- Gespeicherte Prozeduren lassen sich mittels JDBC aufrufen, indem ein Objekt der Schnittstelle *CallableStatement* erzeugt wird.
- Der Aufruf einer gespeicherten Prozedur erfolgt über die Methode *prepareCall*, die als Parameter eine Zeichenkette in Escape-Schreibweise hat.

  CallableStatement cstmt = con.prepareCall(“{call TestProc(?,?)}”);

  Parameter werden dabei wie in JDBC üblich mit dem Symbol ? gekennzeichnet.
- Danach müssen die Werte der Eingabeparameter mit setXXX-Methoden gesetzt werden.

  cstmt.setString(1,’Schneider’);

  cstmt.setDouble(2,42.0);
- Für die Ausgabeparameter muss vor der Ausführung noch der JDBC-Typ festgelegt werden (Ann.: 1. Parameter ist IN und der 2. Parameter InOut):

  cstmt.registerOutParameter(1,java.sql.Types.VARCHAR);

  cstmt.registerOutParameter(2,java.sql.Types.FLOAT);
- Die Prozedur kann dann mit einem Aufruf von execute aufgerufen werden.
- Danach können die Ausgabeparameter mit getXXX-Methoden ausgelesen werden.

  double res = cstmt.getDouble(2);

# Gespeicherte Funktionen in SQL

- Gespeicherte Funktionen können innerhalb von SQL deklariert und auch aufgerufen werden. Aber es gelten folgende Einschränkungen:
  - Die Funktion darf keine Gruppierungsoperationen enthalten.
  - Alle Datentypen der Eingabe und der Ausgabe müssen dem Datenbanksystem bekannt sein (also keine PL/SQL-Typen)
- Beispiel einer gespeicherten Funktion:
  create function simple(x in int) return int as begin return x / 101; end simple;
- Beispiel des Aufrufs einer Funktion in einer SQL-Anfrage:
  select Name, simple(Lohn) from Personal;

# 5.3.2 Trigger: Ein Anwendungsfall für gespeicherte Prozeduren

- Ein Trigger ist eine gespeicherte Prozedur, die bei Erfüllung gewisser Kriterien oder als Seiteneffekt einer Änderungsoperation in der Datenbank vom DBMS implizit ausgeführt wird.
  - Eine Ausführung erfolgt immer dann, wenn ein bestimmtes Ereignis erfüllt ist.
- Trigger werden insbesondere zur Wahrung der Datenkonsistenz (dynamische Integritätsbedingungen) genutzt. Trigger gehen deshalb über eine reine Überprüfung des Datenbankzustands hinaus.
  - Ein Trigger kann z. B. dafür sorgen, dass Statistiken aktualisiert werden oder abgeleitete Spalten berechnet werden.
- Ein Trigger besteht aus
  - einem Kopf, in dem Vorbedingungen zur Ausführung formuliert sind,
  - und einem PL/SQL-Block.
- Trigger sind nicht im SQL92 Standard, haben aber eine hohe praktische Relevanz.

# Einführendes Beispiel

- Im Folgenden betrachten wir eine Relation Professoren, die als Attribut den Rang des Professors besitzt (siehe Kemper/Eickler). Durch den folgenden Trigger wird verhindert, dass der Rang von Professoren durch einen Update niedriger wird.

```
create trigger keineDegradierung
before update on Professoren
for each row
when (:old.Rang is not null)
begin
    if :old.Rang = "C3" and :new.Rang = "C2" then :new.Rang := "C3"; end if;
    if :old.Rang = "C4" then :new.Rang := "C4"; end if;
    if :new.Rang is null then :new.Rang := :old.Rang end if;
end;
```

# Kopf eines Triggers

- Anlegen bzw. Verändern eines neuen bzw. bestehenden Triggers erfolgt über
  - create trigger <name> bzw. replace trigger <name>
- Trigger-Ereignis
  - Jedem Ereignis wird der Zeitpunkt vorangestellt, wann der Trigger ausgelöst werden soll.

    before | after
  - Beim Ereignis wird unterschieden zwischen Ändern, Einfügen und Löschen.

    update [of [<Spalte1,…,Spalten>] on <Relationennamen>
    insert on <Relationennamen>
    delete on <Relationennamen>

    Es können auch gleichzeitig mehrere Ereignisse spezifiziert werden. In diesem Fall kann im Rumpf des Triggers durch

    if updating [('<Spalte>')´] then
    if inserting then
    if deleting then

    zwischen den einzelnen Ereignissen unterschieden werden.
- Trigger-Typ

  Es wird unterschieden zwischen einen *befehlsorientierten Trigger*, bei dem der Trigger genau einmal ausgeführt wird und einem *zeilenorientierten Trigger*. Letzteres erfordert

zusätzlich folgende Zeile:

for each row

Bei einem zeilenorientierten Trigger wird der Trigger für jedes geänderte Tupel einmal aufgerufen. Im Rumpf hat man dann die Möglichkeit, das alte Tupel über die Variable :old und das neue Tupel über :new anzusprechen. Ein anderer Zugriff innerhalb des Rumpfs ist aber nicht mehr möglich.

- Trigger-Restriktion
  when <Prädikat>
  - Hier können Bedingungen formuliert werden, welche die Ausführung des Rumpfs auslösen.
  - Bei einem zeilenorientierten Tupel kann man sich über :new und :old auf den neuen bzw. alten Tupel der Relation beziehen.
- Trigger-Rumpf
  - Besteht aus einem PL/SQL-Block mit den obengenannten Modifikationen

# Beispiele

- Protokollierung der Änderungen am Attribut Lohn einer Relation Personal

```
create trigger LogGehalt
before update on Personal
for each row
when (:old > 1333)
begin insert into LogRel values(:old.Lohn, :new.Lohn, sysdate) end;
```

- Zurücksetzen einer Änderung

```
create or replace trigger CheckGehalt
before update on Personal
for each row
when (:new.Lohn > 1500)
begin :new.Lohn := :old.Lohn end;
```

## Probleme bei Triggern

- Anwender muss die Widerspruchsfreiheit der Trigger kontrollieren
- Vermeidung von Aufrufzyklen bei Triggern
- Terminierung
- Trigger nur dann zur Formulierung von Integritätsbedingungen nutzen, wenn dies nicht mit anderen Mitteln möglich ist.

# 5.3.3 Java und Gespeicherte Prozeduren

- Im Zuge des Erfolges der Programmiersprache Java im Umfeld von DBMS bieten derzeit einige DBMS (DB2, Oracle) die Möglichkeit, gespeicherte Prozeduren in Java zu implementieren.
- Die Vorgehensweise ist sehr ähnlich zu der von PL/SQL:
  - Erstellen einer Quelldatei
    ```
    public class Hello {
            public static String world() { return “Hello World”;}
    }
    ```
  - Übersetzen der Quelldatei auf dem Client
    ```
    javac Hello.java
    ```
  - Übergeben der class-Datei an das DBMS
    ```
    loadjava -user scott/tiger Hello.class
    ```
  - Erzeugen einer SQL-Schnittstelle für die Prozedur
    ```
    create function HelloWorld return varchar as
    language java name ’Hello.world() return java.lang.String’;
    ```
  - Ausführen der gespeicherten Prozedur
    ```
    variable myString varchar[20];
    call HELLOWORLD() into :myString;
    print myString;
    ```

# Fazit

- PL/SQL ist eine interessante Erweiterung von SQL, die schon vor Java eine plattformunabhängige Programmierung angeboten hatte.
- SQL wird insbesondere durch gespeicherte Funktionen mächtiger.
- PL/SQL wurde in jüngster Zeit, um so genannte Table Functions erweitert. Diese unterstützen Cursor, die wie eine Relation in der from-Klausel von SQL genutzt werden können.
  - Dabei wird sogar eine bedarfsgesteuerte Ausführung solcher Cursor unterstützt, d.h. die Tupel werden auf Anforderung nach und nach erzeugt.
- Obwohl nahezu jedes relationale DBMS gespeicherte Prozeduren anbietet und es einen Standard SQL/PSM gibt, sind die Lösungen in DBMS derzeit noch recht unterschiedlich.

# 5.4 Weitere Kopplungsmöglichkeiten

## PASCAL/R: eine Erweiterung von Pascal für relationale Datenbanken

- Erweiterung des Typsystems von PASCAL:
  - Datentyp RELATION entspricht “SET OF TUPLE”
  - Operationen der relationalen Algebra
  - Iteratoren (wie Cursor) für die Verarbeitung von Relationen

## Skriptsprachen

- populärer Ansatz für die Erstellung von AWPs (z. B. Visual Basic)
- Nachteil
  - keine strenge Typisierung
  - keine Standards (schlechte Wartung der Applikationssoftware)
  - Durchmischung von verschiedenen Konzepten aus dem Bereich Datenbanken, imperativen Programmierung, Benutzerschnittstellen und Regeln
- Vorteil:
  - schnelle Erstellung von AWP mit graphischen Benutzeroberflächen
  - komfortable Entwicklungsumgebung