1. DB-Anwendungsprogrammierung

- Einleitung
- Eingebettetes SQL
  - statisches SQL / Cursor-Konzept
  - Sitzungskontrolle, Transaktionskontrolle: Isolation Level
  - dynamisches SQL
- Gespeicherte Prozeduren (Stored Procedures)
  - prozedurale Spracherweiterungen von SQL
  - gespeicherte Prozeduren mit Java
- JDBC und SQLJ
  - wesentliche JDBC-Operationen, Transaktionskontrolle
  - Problem der SQL Injection
  - Nutzung von Stored Procedures
  - SQLJ, Iteratorkonzept
- Web-Anbindung
  - CGI, Java Server Pages
  - PHP

Kopplung Programmiersprache – DBS/SQL

- Zwei wesentliche Kriterien
  - Embedded SQL vs. Call-Level-Interface (CLI)
  - Statisches vs. Dynamisches SQL
- Einbettung von SQL (Embedded SQL)
  - Spracherweiterung um spezielle DB-Befehle (EXEC SQL ...)
  - Vorübersetzer (Prä-Compiler) wandelt DB-Aufrufe in Prozeduraufrufe um
- Call-Schnittstelle (CLI)
  - DB-Funktionen werden durch Bibliothek von Prozeduren realisiert
  - Anwendung enthält lediglich Prozeduraufrufe
  - weniger komfortable Programmierung als mit Embedded SQL
- Statisches SQL: Anweisungen müssen zur Übersetzungszeit feststehen
  - Optimierung zur Übersetzungszeit ermöglicht hohe Effizienz (Performance)
- Dynamisches SQL: Konstruktion von SQL-Anweisungen zur Laufzeit

Kopplung Programmiersprache – DBS

Einbettung von SQL in Wirtssprache (Embedded SQL)

- statisches SQL
- dynamisches SQL

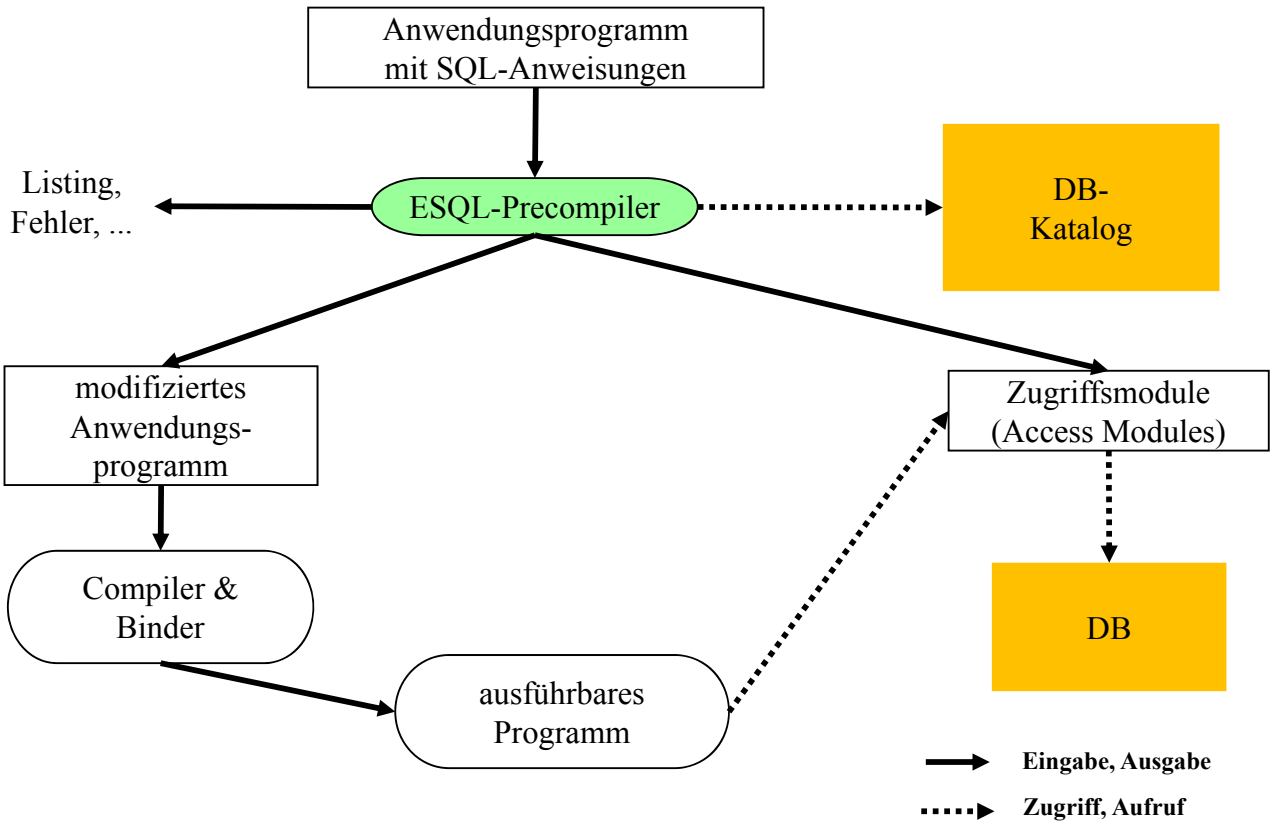
Call-Schnittstelle *Call Level Interface (CLI)*

- dynamisches SQL

Spracherweiterungen (bzw. neue Sprache)

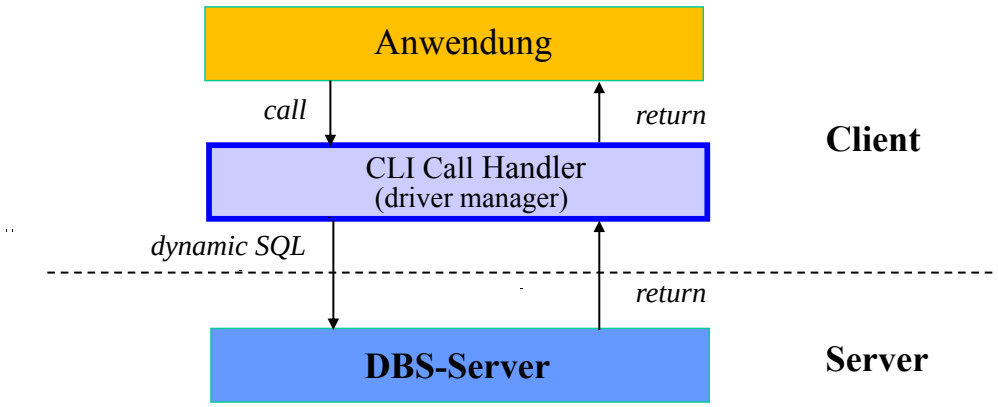
- *SQL-Erweiterungen: Stored Procedures / Kontrollanweisungen*
- *Datenbankprogrammiersprachen*
- *persistente Programmiersprachen*

Verarbeitung von ESQL-Programmen



Call-Level-Interface

- alternative Möglichkeit zum Aufruf von SQL-Befehlen innerhalb von Anwendungsprogrammen: direkte Aufrufe von Prozeduren/Funktionen einer standardisierten Bibliothek (API)
- Hauptvorteil: keine Präkompilierung von Anwendungen
  - Anwendungen mit SQL-Aufrufen brauchen nicht im Source-Code bereitgestellt zu werden
  - wichtig zur Realisierung von kommerzieller Anwendungs-Software bzw. Tools
- Einsatz v. a. in Client/Server-Umgebungen



Call-Level-Interface (2)

- Unterschiede in der SQL-Programmierung zu eingebettetem SQL
  - CLI impliziert i.a. dynamisches SQL (Optimierung zur Laufzeit)
  - komplexere Programmierung
  - explizite Anweisungen zur Datenabbildung zwischen DBS und Programmvariablen
- SQL-Standardisierung des CLI stark an ODBC angelehnt
- CLI für Java-Anwendungen: JDBC

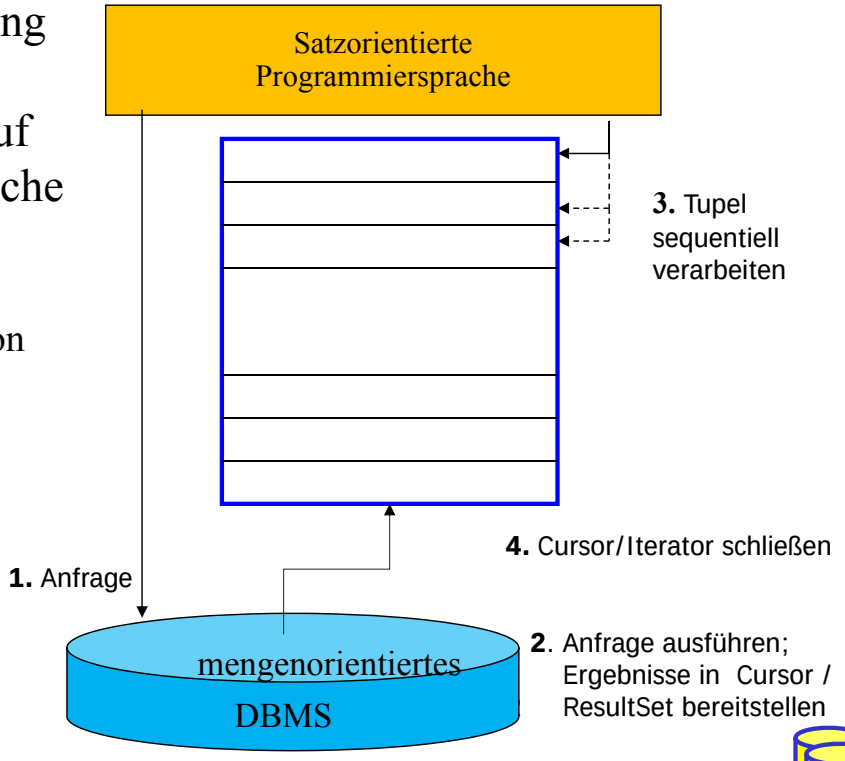
Statisches SQL: Beispiel für C

```
exec sql include sqlca; /* SQL Communication Area */
main ()
{
exec sql begin declare section;
   char  X[8];
   int   GSum;
exec sql end declare section;
exec sql connect to dbname;
exec sql insert into PERS(PNR,PNAME,GEHALT) values (4711,'Ernie', 32000);
exec sql insert into PERS(PNR,PNAME,GEHALT) values (4712,'Bert',  38000);
printf("ANR ? "); scanf(" %s", X);
exec sql select sum (GEHALT) into :GSum from PERS where ANR = :X;
printf("Gehaltssumme: %d\n", GSum)
exec sql commit work;
exec sql disconnect;
}
```

- Anmerkungen
  - eingebettete SQL-Anweisungen werden durch "EXEC SQL" eingeleitet und spezielles Symbol (hier ";") beendet, um Compiler Unterscheidung von anderen Anweisungen zu ermöglichen
  - Verwendung von AP-Variablen in SQL-Anweisungen verlangt Deklaration innerhalb eines "declare section"-Blocks sowie Angabe des Präfix ":" innerhalb von SQL-Anweisungen
  - Werteabbildung mit Typanpassung durch INTO-Klausel bei SELECT
  - Kommunikationsbereich SQLCA (Rückgabe von Statusanzeigern u. ä.)

Mengenorientierte Anfragen

- Queries mit nur einem Ergebnissatz
  - einfache Übernahme der Ergebnisse in Programmvariable (SELECT attr INTO :var)
- Kernproblem bei SQL-Einbettung in Programmiersprachen: Abbildung von Tupelmengen auf Variablen der Programmiersprache
  - Nutzung von Cursor/Iteratoren bzw. Result-Sets zur satzweisen Bereitstellung und Abarbeitung von DBS-Ergebnismengen



Cursor-Konzept in Embedded SQL

- Cursor ist ein Iterator, der einer Anfrage (Relation) zugeordnet wird und mit dessen Hilfe die Tupeln des Ergebnismenge einzeln (one tuple at a time) im Programm bereitgestellt werden
  - Trennung von Query-Spezifikation (Cursor-Deklaration) und Bereitstellung/Verarbeitung von Tupeln im Query-Ergebnis
- Operationen auf einen Cursor C1
  - DECLARE C1 CURSOR FOR table-exp
  - OPEN C1
  - FETCH C1 INTO VAR1, VAR2, . . ., VARn
  - CLOSE C1
- Anbindung einer SQL-Anweisung an die Wirtssprachen-Umgebung
  - Übergabe der Werte eines Tupels mit Hilfe der INTO-Klausel bei FETCH
    => INTO target-commalist (Variablenliste d. Wirtsprogramms)
  - Anpassung der Datentypen (Konversion)
- kein Cursor erforderlich für Select-Anweisungen, die nur einen Ergebnissatz liefern (SELECT INTO)

Cursor-Konzept (2)

- Beispielprogramm in C (vereinfacht)

```
...
   exec sql begin declare section;
        char  X[50];
        char  Y[8];
        double G;
   exec sql end declare section;
   exec sql declare c1 cursor  for
            select NAME, GEHALT from PERS where ANR = :Y;
   printf("ANR ? "); scanf(" %s", Y);
   exec  sql  open  C1;
   while  (sqlcode == ok) {
            exec  sql  fetch  C1  into  :X, :G;
            printf("%s\n", X)}
   exec  sql  close  C1;
...
```

- DECLARE C1 . . . ordnet der Anfrage einen Cursor C1 zu
- OPEN C1 bindet die Werte der Eingabevariablen
- Systemvariable SQLCODE zur Übergabe von Fehlermeldungen (Teil von SQLCA)

Verwaltung von Verbindungen

- Zugriff auf DB erfordert i.a. zunächst, eine Verbindung herzustellen, v.a. in Client/Server-Umgebungen
  - Aufbau der Verbindung mit CONNECT, Abbau mit DISCONNECT
  - jeder Verbindung ist eine Session zugeordnet
  - Anwendung kann Verbindungen (Sessions) zu mehreren Datenbanken offenhalten
  - Umschalten der "aktiven" Verbindung durch SET CONNECTION

```
CONNECT TO   target [AS connect-name] [USER user-name]
```

```
SET CONNECTION   { connect-name | DEFAULT }
```

```
DISCONNECT   { CURRENT | connect-name | ALL }
```

SQL-Isolationsstufen

- 4 Konsistenzebenen (Isolation Level) zur Synchronisation von Transaktionen
  - Default ist Serialisierbarkeit (serializable)
  - Lost-Update muß generell vermieden werden
  - READ UNCOMMITTED für Änderungstransaktionen unzulässig

Isolation Level	Anomalie		
	Dirty Read	Non-Repeatable Read	Phantome
Read Uncommitted	+	+	+
Read Committed	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

- SQL-Anweisung zum Setzen der Konsistenzebene (Embedded SQL)

```
SET TRANSACTION [READ WRITE | READ ONLY] ISOLATION LEVEL <level>
```

Beispiel: exec sql SET TRANSACTON Read Only ISOLATION LEVEL Read Committed

Dynamisches SQL

- dynamisches SQL: Festlegung von SQL-Anweisungen zur Laufzeit
  -> Query-Optimierung i.a. erst zur Laufzeit möglich
- SQL-Anweisungen werden vom Compiler wie Zeichenketten behandelt
  - Dekaration DECLARE STATEMENT
  - Anweisungen enthalten SQL-Parameter (?) statt Programmvariablen
- 2 Varianten: Prepare-and-Execute bzw. Execute Immediate

```
exec sql begin declare section;
   char  Anweisung[256], X[6];
exec sql end declare section;
exec sql declare SQLanw statement;
Anweisung = "DELETE FROM PERS WHERE  ANR = ? AND ORT = ?"; /*bzw. Einlesen
exec sql prepare SQLanw from :Anweisung;
exec sql execute SQLanw using
scanf(" %s", X);
exec sql execute SQLanw using
```

Dynamisches SQL (2)

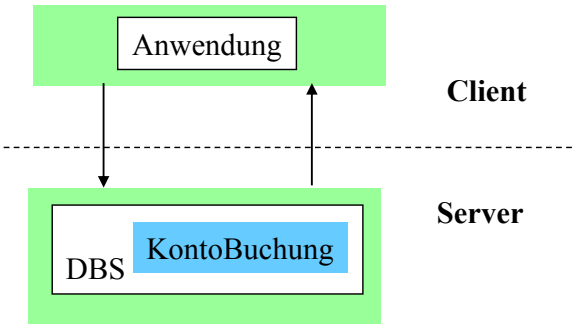
- Variante ohne Vorbereitung: EXECUTE IMMEDIATE
- Beispiel

```
scanf(" %s", Anweisung);
exec sql execute immediate :Anweisung;
```

- Maximale Flexibilität, jedoch potentiell geringe Performance
  - kann für einmalige Query-Ausführung ausreichen

Gespeicherte Prozeduren (Stored Procedures)

- Prozeduren werden durch DBS gespeichert und verwaltet
  - benutzerdefinierte Prozeduren oder Systemprozeduren
  - Programmierung der Prozeduren in SQL oder allgemeiner Programmiersprache



- Vorteile:
  - als gemeinsamer Code für verschiedene Anwendungsprogramme wiederverwendbar
  - Anzahl der Zugriffe des Anwendungsprogramms auf die DB werden reduziert
  - Performance-Vorteile v.a. in Client-Server-Umgebungen
  - höherer Grad der Isolation der Anwendung von DB wird erreicht
- Nachteile ?

Persistente SQL-Module (PSM)

- SQL-Prozeduren erfordern Spracherweiterungen gegenüber SQL1992
  - u.a. allgemeine Kontrollanweisungen IF, WHILE, etc.
  - herstellerspezifische Festlegungen bereits seit 1987 (Sybase Transact-SQL)
  - PSM: seit SQL99 standardisiert
- Routinen: Prozeduren und Funktionen
  - Routinen sind Schema-Objekte (wie Tabellen etc.) und werden im Katalog aufgenommen (beim DBS/Server)
  - geschrieben in SQL (SQL routine) oder in externer Programmiersprache (C, Java, FORTRAN, ...) -> 2 Sprachen / Typsysteme
- zusätzliche DDL-Anweisungen
  - CREATE PROCEDURE
  - DROP PROCEDURE
  - CREATE FUNCTION
  - DROP FUNCTION

PSM: SQL-Routinen

- **SQL-Routinen**: in SQL geschriebene Prozeduren/Funktionen
  - Deklarationen lokaler Variablen etc. innerhalb der Routinen
  - Nutzung zusätzlicher Kontrollanweisungen: Zuweisung, Blockbildung, IF, LOOP, etc.
  - Exception Handling (SIGNAL, RESIGNAL)
  - integrierte Programmierumgebung
  - keine Typkonversionen
- Beispiel

```
CREATE PROCEDURE KontoBuchung
    (IN konto INTEGER, IN betrag DECIMAL (15,2));
          BEGIN  DECLARE C1 CURSOR FOR ...;

                 UPDATE account
                 SET balance = balance + betrag
                 WHERE account_# = konto;
          ...
END;
```

- Prozeduren werden über CALL-Anweisung aufgerufen:

```
exec sql CALL KontoBuchung  (:account_#, :balance);
```

PSM: SQL-Routinen (2)

- Beispiel einer SQL-Funktion:

```
CREATE FUNCTION vermoegen (kunr INTEGER)
                RETURNS DECIMAL (15,2);

 BEGIN     DECLARE vm INTEGER;
           SELECT sum (balance) INTO vm
           FROM   account
           WHERE account_owner = kunr;
           RETURN vm;
 END;
```

- Aufruf persistenter Funktionen (SQL und externe) in SQL-Anweisungen wie Built-in-Funktionen

```
SELECT *
FROM kunde
WHERE vermoegen (KNR) > 100000.00
```

- Prozedur- und Funktionsaufrufe können rekursiv sein

Prozedurale Spracherweiterungen: Kontrollanweisungen

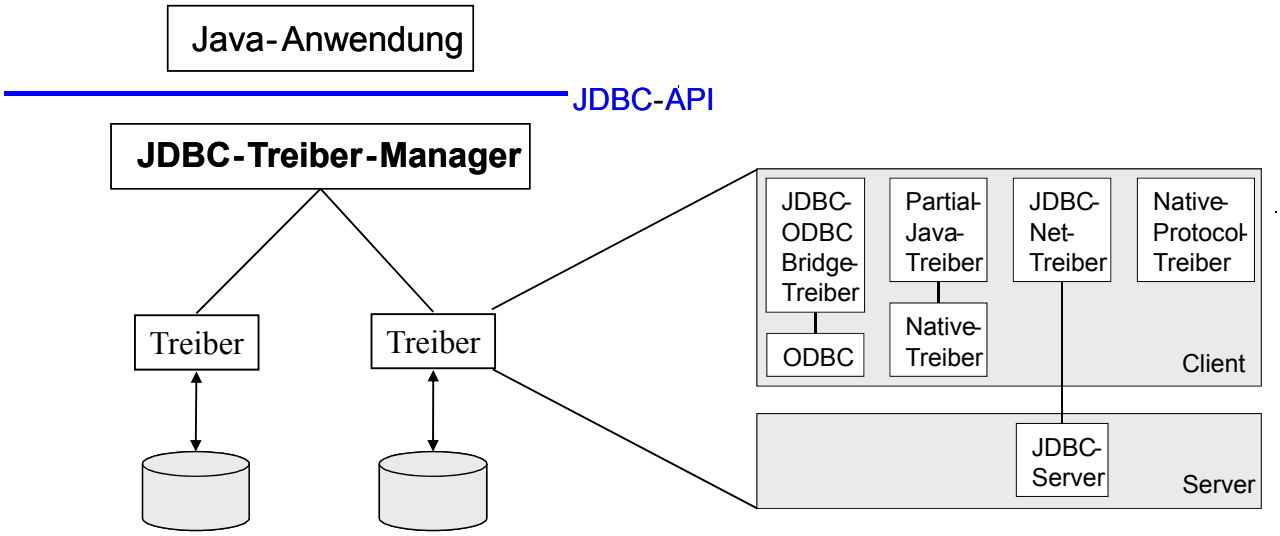
Compound Statement	BEGIN ... END;
SQL-Variablendeklaration	DECLARE var type;
If-Anweisung	IF condition THEN ... ELSE ... :
Case-Anweisung	CASE expression WHEN x THEN ... WHEN ... :
Loop-Anweisung	WHILE i < 100 LOOP ... END LOOP;
For-Anweisung	FOR result AS ... DO ... END FOR;
Leave-Anweisung	LEAVE ...;
Prozeduraufruf	CALL procedure_x (1, 2, 3);
Zuweisung	SET x = “abc”;
Return-Anweisung	RETURN x;
Signal/Resignal	SIGNAL division_by_zero;

PSM Beispiel

```
outer: BEGIN
  DECLARE account INTEGER DEFAULT 0;
  DECLARE balance DECIMAL (15,2);
  DECLARE no_money EXCEPTION FOR SQLSTATE VALUE 'xxxxx';
  DECLARE DB_inconsistent EXCEPTION FOR SQLSTATE VALUE 'yyyyy';

  SELECT account_#, balance INTO account, balance FROM accounts ...;
  IF (balance - 10) < 0 THEN SIGNAL no_money;
  BEGIN ATOMIC
    DECLARE cursor1 SCROLL CURSOR ...;
    DECLARE balance DECIMAL (15,2);
    SET balance = outer.balance - 10;
    UPDATE accounts SET balance = balance WHERE account_# = account;
    INSERT INTO account_history VALUES (account, CURRENT_DATE, 'W', balance); .....
  END;
 EXCEPTION
    WHEN no_money THEN
    CASE (SELECT account_type FROM accounts WHERE account_# = account)
          WHEN 'VIP' THEN INSERT INTO send_letter ....
          WHEN 'NON-VIP' THEN INSERT INTO blocked_accounts ...
    ELSE SIGNAL DB_inconsistent;

    WHEN DB_inconsistent THEN
          BEGIN .... END;

    END;
```

JDBC (Java Database Connectivity)

- Standardschnittstelle für den Zugriff auf SQL-Datenbanken unter Java
- basiert auf dem SQL/CLI (call-level-interface)
- Grobarchitektur



- durch Auswahl eines anderen JDBC-Treibers kann ein Java-Programm ohne Neuübersetzung auf ein anderes Datenbanksystem zugreifen

JDBC: Grundlegende Vorgehensweise

- Schritt 1: Verbindung aufbauen

```
import java.sql.*;
        ...
Class.forName ( "COM.ibm.db2.jdbc.net.DB2Driver");
Connection con =
        DriverManager.getConnection ("jdbc:db2://host:6789/myDB", "login","pw");
```

- Schritt 2: Erzeugen eines SQL-Statement-Objekts

```
Statement stmt = con.createStatement();
```

- Schritt 3: Statement-Ausführung

```
ResultSet rs = stmt.executeQuery ("SELECT matrikel FROM student");
```

- Schritt 4: Iterative Abarbeitung der Ergebnisdatensätze

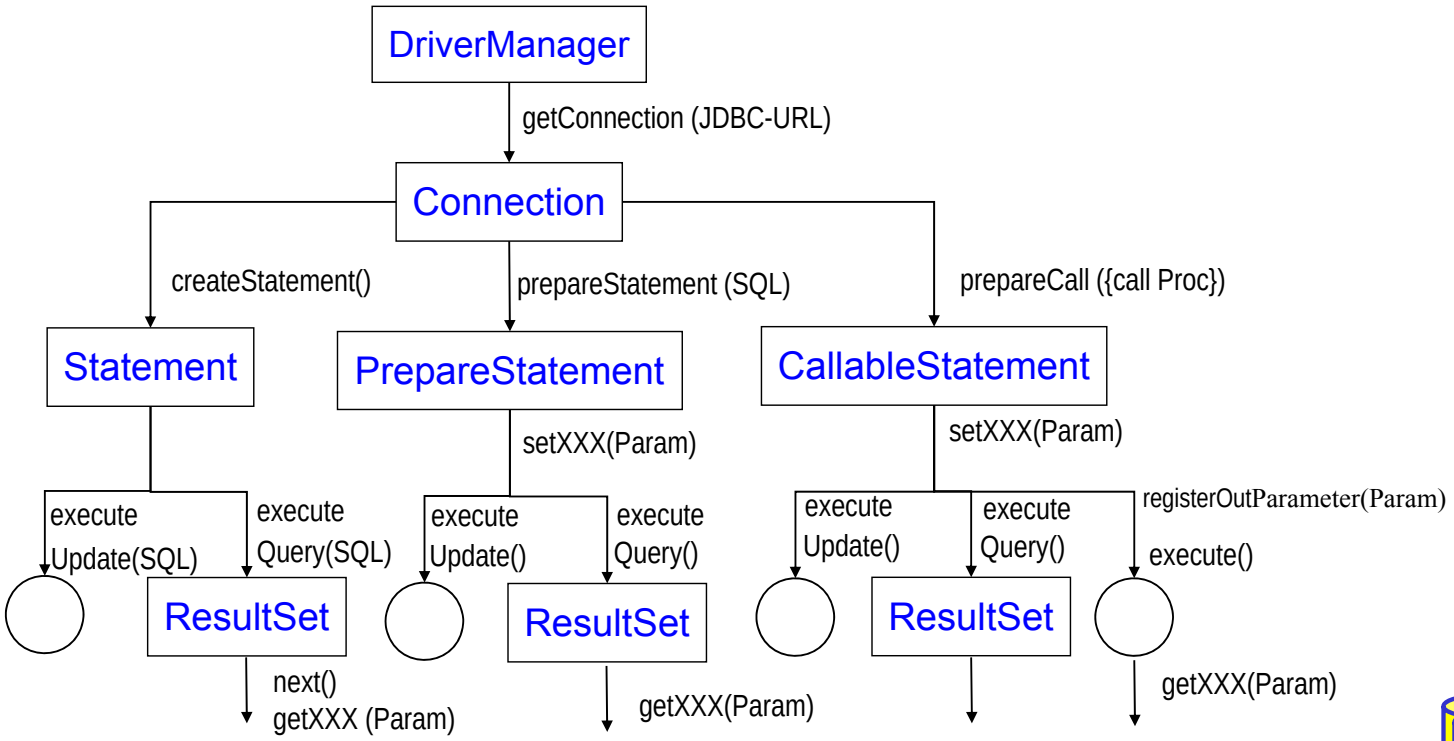
```
while (rs.next())
        System.out.println ("Matrikelnummer: " + rs.getString("matrikel"));
rs.close()
```

- Schritt 5: Schließen der Datenbankverbindung

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

JDBC-Klassen

- streng typisierte objekt-orientierte API
- Aufrufbeziehungen (Ausschnitt)



JDBC: Beispiel 1

- Beispiel: Füge alle Matrikelnummern aus Tabelle ‘Student’ in eine Tabelle ‘Statistik’ ein

```
import java.sql.*;
 ...
public void copyStudents() {
 try {
   Class.forName („COM.ibm.db2.jdbc.net.DB2Driver“);// lade JDBC-Treiber (zur Laufzeit)
 } catch (ClassNotFoundException e) {// Fehlerbehandlung}
 try {
   String url = "jdbc:db2://host:6789/myDB"// spezifiziert JDBC-Treiber, Verbindungsdaten
   Connection con = DriverManager.getConnection(url, "login", "password");
   Statement stmt = con.createStatement();// Ausführen von Queries mit Statement-Objekt
   PreparedStatement pStmt = con.prepareStatement("INSERT INTO statistik (matrikel)
                                                   VALUES (?)");
                             // Prepared-Stmts für wiederholte Ausführung

   ResultSet rs = stmt.executeQuery("SELECT matrikel FROM student");// führe Query aus

   while (rs.next()) {                  // lese die Ergebnisdatensätze aus
      String matrikel = rs.getString(1);  // lese aktuellen Ergebnisdatensatz
      pStmt.setString (1, matrikel);      // setze Parameter der Insert-Anweisung
      pStmt.executeUpdate();              // führe Insert-Operation aus
   }
   con.close();
 } catch (SQLException e) { // Fehlerbehandlung}
}
```

Beispiel 2: Gehaltsänderung

- Verwendung eines Eingabeparameters

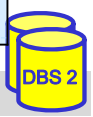
```
import java.sql.*;
 ...
public void main (string[] args){
 try {
   Class.forName („COM.ibm.db2.jdbc.net.DB2Driver“);// lade JDBC-Treiber (zur Laufzeit)
 } catch (ClassNotFoundException e) {// Fehlerbehandlung}
 try {
   String url = "jdbc:db2://host:6789/myDB2"// spezifiziert JDBC-Treiber, Verbindungsdaten
   Connection con = DriverManager.getConnection(url, "login", "password");

   Statement stmt = con.createStatement();

   boolean success = stmt.execute ("UPDATE PERS SET Gehalt=Gehalt*2.0 WHERE PNR="+args[0])

   con.close();
 } catch (SQLException e) { // Fehlerbehandlung}
}
```

args[0]=„35"

PNR	NAME	GEHALT
34	Mey	32000
35	Schultz	42500
37	Abel	41000

→

PNR	NAME	GEHALT
34	Mey	32000
35	Schultz	**85000**
37	Abel	41000



Gefahr einer SQL INJECTION

- Verwendung eines Eingabeparameters

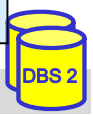
```
import java.sql.*;
 ...
public void main (string[] args){
 try {
   Class.forName („COM.ibm.db2.jdbc.net.DB2Driver“);// lade JDBC-Treiber (zur Laufzeit)
 } catch (ClassNotFoundException e) {// Fehlerbehandlung}
 try {
   String url = "jdbc:db2://host:6789/myDB2"// spezifiziert JDBC-Treiber, Verbindungsdaten
   Connection con = DriverManager.getConnection(url, "login", "password");

   Statement stmt = con.createStatement();

   boolean success = stmt.execute ("UPDATE PERS SET Gehalt=Gehalt*2.0 WHERE PNR="+args[0])

   con.close();
 } catch (SQLException e) { // Fehlerbehandlung}
}
```

args[0]=
„35 OR Gehalt<100000"

PNR	NAME	GEHALT
34	Mey	32000
35	Schultz	42500
37	Abel	41000

→

PNR	NAME	GEHALT
34	Mey	
35	Schultz	
37	Abel	



SQL INJECTION (2)

- Nutzung von Prepared-Statement ermöglicht Abhilfe

```
import java.sql.*;
 ...
public void main (string[] args){
 try {
   Class.forName („COM.ibm.db2.jdbc.net.DB2Driver“);// lade JDBC-Treiber (zur Laufzeit)
 } catch (ClassNotFoundException e) {// Fehlerbehandlung}
 try {
   String url = "jdbc:db2://host:6789/myDB2"// spezifiziert JDBC-Treiber, Verbindungsdaten
   Connection con = DriverManager.getConnection(url, "login", "password");

         PreparedStatement pStmt = con.prepareStatement("UPDATE PERS
                                                  SET Gehalt=Gehalt*2.0 WHERE PNR=?"
         pStmt.setInt (1, 35);    // setze Parameter der Update-Anweisung
         pStmt.executeUpdate();

         pStmt.setString (1, args[0]);
         pStmt.executeUpdate();

   con.close();
 } catch (SQLException e) { // Fehlerbehandlung}
}
```

args[0]= „35 OR Gehalt<100000"

JDBC: Transaktionskontrolle

- **Transaktionskontrolle** durch Methodenaufrufe der Klasse `Connection`
  - `setAutoCommit`: Ein-/Abschalten des Autocommit-Modus (jedes Statement ist eigene Transaktion)
  - `setReadOnly`: Festlegung ob lesende oder ändernde Transaktion
  - `setTransactionIsolation`: Festlegung der Synchronisationsanforderungen (None, Read Uncommitted, Read Committed, Repeatable Read, Serializable)
  - `commit` bzw. `rollback`: erfolgreiches Transaktionsende bzw. Transaktionsabbruch
- Beispiel

```
try {
   con.setAutoCommit (false);
   // einige Änderungsbefehle, z.B. Inserts
   con.commit ();
} catch (SQLException e) {
   try { con.rollback (); } catch (SQLException e2) {}
} finally {
   try { con.setAutoCommit (true); } catch (SQLException e3) {}
}
```

Gespeicherte Prozeduren in Java

- Erstellen einer Stored Procedure (z. B. als Java-Methode)

```
public static void kontoBuchung(int konto,
                        java.math.BigDecimal betrag,
                           java.math.BigDecimal[] kontostandNeu)
 throws SQLException {
   Connection con = DriverManager.getConnection
                        ("jdbc:default:connection");
                              // Nutzung der aktuellen Verbindung
   PreparedStatement pStmt1 = con.prepareStatement(
          "UPDATE account SET balance = balance + ?
                WHERE account_# = ?");
   pStmt1.setBigDecimal( 1, betrag);
   pStmt1.setInt( 2, konto);
   pStmt1.executeUpdate();
   PreparedStatement pStmt2 = con.prepareStatement(
          "SELECT balance FROM account WHERE account_# = ?");
   pStmt2.setInt( 1, konto);
   ResultSet rs = pStmt2.executeQuery();
   if (rs.next()) kontostandNeu[0] = rs.getBigDecimal(1);
   pStmt1.close(); pStmt2.close(); con.close();
   return;
}
```

Gespeicherte Prozeduren in Java (2)

- Deklaration der Prozedur im Datenbanksystem mittels SQL

```
CREATE PROCEDURE KontoBuchung(   IN konto INTEGER,
                                 IN betrag DECIMAL (15,2),
                              OUT kontostandNeu DECIMAL (15,2))
 LANGUAGE java
 PARAMETER STYLE java
 EXTERNAL NAME 'myjar:KontoClass.kontoBuchung'
                // Java-Archiv myjar enthält Methode
```

Gespeicherte Prozeduren in Java (3)

- Aufruf einer Stored Procedure in Java

```
public void ueberweisung(Connection con, int konto1, int konto2,
java.math.BigDecimal betrag)
throws SQLException {
  con.setAutoCommit (false);
  CallableStatement cStmt = con.prepareCall("{call KontoBuchung (?, ?, ?)}");
  cStmt.registerOutParameter(3, java.sql.Types.DECIMAL);
  cStmt.setInt(1, konto1);
  cStmt.setBigDecimal(2, betrag.negate());
  cStmt.executeUpdate();
  java.math.BigDecimal newBetrag = cStmt.getBigDecimal(3);
  System.out.println("Neuer Betrag: " + konto1 + " " + newBetrag.toString());
  cStmt.setInt(1, konto2);
  cStmt.setBigDecimal(2, betrag);
  cStmt.executeUpdate();
  newBetrag = cStmt.getBigDecimal(3);
  System.out.println("Neuer Betrag: " + konto2 + " " + newBetrag.toString());
  cStmt.close();
  con.commit ();
  return;
}
```


SQLJ

- Eingebettetes SQL (Embedded SQL) für Java
- direkte Einbettung von SQL-Anweisungen in Java-Code
  - Präprozessor um SQLJ-Programme in Java-Quelltext zu transformieren
- Vorteile
  - Syntax- und Typprüfung zur Übersetzungszeit
  - Vor-Übersetzung (Performance)
  - einfacher/kompakter als JDBC
  - streng typisierte Iteratoren (Cursor-Konzept)

SQLJ (2)

- eingebettete SQL-Anweisungen: #sql [[<context>]] { <SQL-Anweisung> }
  - beginnen mit **#sql** und können mehrere Zeilen umfassen
  - können Variablen der Programmiersprache (:x) bzw. Ausdrücke (:y + :z) enthalten
  - können Default-Verbindung oder explizite Verbindung verwenden
- Vergleich SQLJ – JDBC (1-Tupel-Select)

SQLJ

```
#sql [con]{ SELECT name INTO :name
            FROM student WHERE matrikel = :mat};
```

JDBC

```
java.sql.PreparedStatement ps =
      con.prepareStatement („SELECT name “+
            „FROM student WHERE matrikel = ?“);
ps.setString (1, mat);
java.sql.ResultSet rs = ps.executeQuery();
rs.next()
name= rs.getString(1);
rs.close;
```

SQLJ (3)

- Iteratoren zur Realisierung eines Cursor-Konzepts
  - eigene Iterator-Klassen
  - benannte Iteratoren: Zugriff auf Spalten des Ergebnisses über Methode mit dem Spaltennamen
  - Positionsiteratoren: Iteratordefinition nur mit Datentypen; Ergebnisabruf mit *FETCH*-Anweisung ; *endFetch()* zeigt an, ob Ende der Ergebnismenge erreicht
- Vorgehensweise (benannte Iteratoren)
  1. Definition Iterator-Klasse

```
#sql public iterator IK (String a1, String a2);
```

  2. Zuweisung mengenwertiges Select-Ergebnis an Iterator-Objekt

```
IK io;
#sql io = { SELECT a1, a2 FROM …};
```

  3. Satzweiser Abruf der Ergebnisse

```
while   io.next() {  …;
     String s1 = io.a1();
     String s2 = io.a2(); … }
```

  4. Schließen Iterator-Objekt

```
io.close();
```

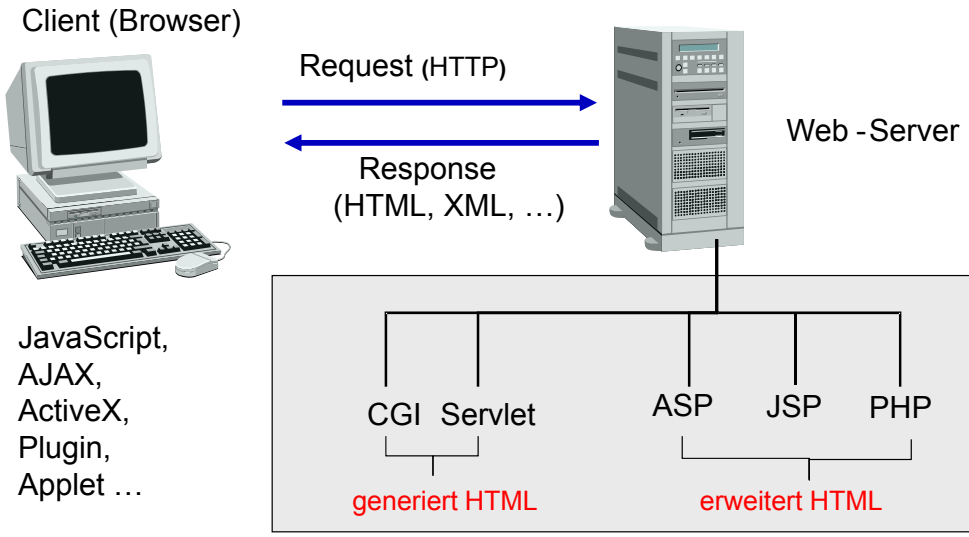
SQLJ (4)

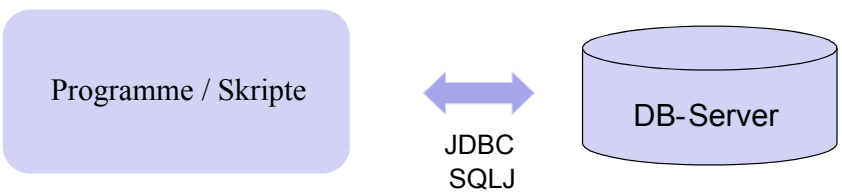
- Beispiel: Füge alle Matrikelnummern aus Tabelle ‘Student’ in eine Tabelle ‘Statistik’ ein.

```
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
...
public void copyStudents() {
String drvClass= „COM.ibm.db2.jdbc.net.DB2Driver“;
   try {
    Class.forName(drvClass);
} catch (ClassNotFoundException e) { // errorlog }
try {
   String url = “jdbc:db2://host:6789/myDB”
   Connection con = DriverManager.getConnection
                        (url, “login”, “password”);
   // erzeuge einen Verbindungskontext
   // (ein Kontext pro Datenbankverbindung)
   DefaultContext ctx = new DefaultContext(con);
   // definiere Kontext als Standard-Kontext
   DefaultContext.setDefaultContext(ctx);
```

```
   // 1. deklariere Klasse für benannten Iterator
   #sql public iterator MatrikelIter (String matrikel);

   // 2. erzeuge Iterator-Objekt;
   //     Zuordnung und Aufruf der SQL-Anfrage
   MatrikelIter mIter;
   #sql mIter = { SELECT matrikel FROM student };

   // 3. navigiere über Ergebnis, Abruf Ergebniswert
   while (mIter.next()) {
    #sql {INSERT INTO statistik (matrikel)
          VALUES ( mIter.matrikel()) };
    }
   // 4. Schliesse Iterator
   mIter.close();
   } catch (SQLException e) { // errorlog }
}
```

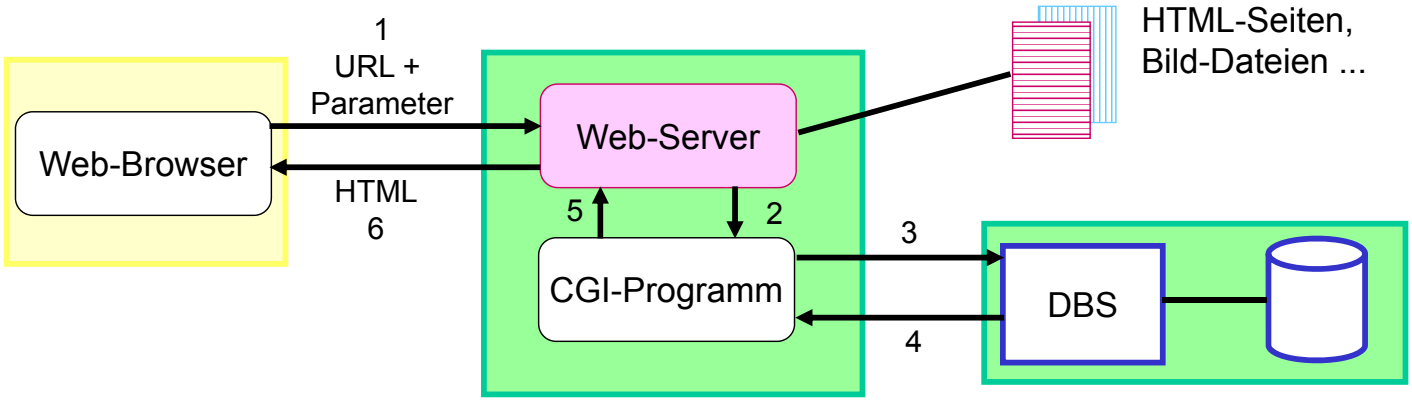

Web-Anbindung von Datenbanken





Server-seitige Anbindung: CGI-Kopplung

- CGI: Common Gateway Interface
  - plattformunabhängige Schnittstelle zwischen Web-Server (HTTP-Server) und externen Anwendungen
  - wird von jedem Web-Server unterstützt



- CGI-Programme (z.B. realisiert in Perl, PHP, Python, Ruby, Shell-Skripte)
  - erhalten Benutzereingaben (aus HTML-Formularen) vom Web-Server als Parameter
  - können beliebige Berechnungen vornehmen und auf Datenbanken zugreifen
  - Ergebnisse werden als dynamisch erzeugte HTML-Seiten an Client geschickt

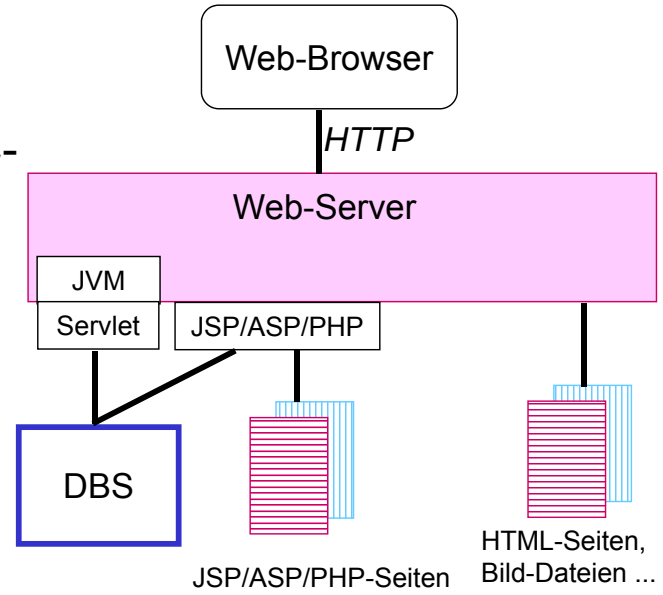
CGI-Kopplung (2)

- CGI-Programme generieren HTML-Ausgabe
- aufwendige / umständliche Programmierung
- mögliche Performance-Probleme
  - Eingabefehler werden erst im CGI-Programm erkannt
  - für jede Interaktion erneutes Starten des CGI-Programms
  - für jede Programmaktivierung erneuter Aufbau der DB-Verbindung

```
#!/bin/perl
use Msql;
# Seitenkopf ausgeben:
print"Content-type: text/html\n\n";
# [...]
# Verbindung mit dem DB-Server herstellen:
$testdb = Msql->connect;
$testdb->selectdb("INFBIBLIOTHEK");
# DB-Anfrage
$q = $testdb->query
     ("select Autor, Titel  from ...");
# Resultat ausgeben:
print"<TABLE BORDER=1>\n"; print"<TR>\n
  <TH>Autor<TH>Titel</TR>";
$rows = $q -> numrows;
while ($rows>0) {
      @sqlrow = $q->fetchrow;
      print      "<tr><td>",@sqlrow[0],
                 "</td><td>",
                 @sqlrow[1],</td></ tr>\n";
       $rows--; }
print"</TABLE>\n";
# Seitenende ausgeben
```

Server-seitige Web-Anbindung: weitere Ansätze

- Einsatz von Java-Servlets
  - herstellerunabhängige Erweiterung von Web-Servern (Java Servlet-API)
  - Integration einer Java Virtual Machine (JVM) im Web-Server -> Servlet-Container
- server-seitige Erweiterung von HTML-Seiten um Skript-/Programmlogik
  - Java Server Pages
  - Active Server Pages (Microsoft)
  - PHP-Anweisungen



Java Server Pages (JSP)

- Entwurf von dynamischen HTML-Seiten mittels HTML-Templates und XML-artiger Tags
- Trennung Layout vs. Applikationslogik durch Verwendung von Java-Beans

JSP-Seite:

```
<HTML>
<BODY>
 <jsp:useBean id="EmpData" class="FetchEmpDataBean“ scope="session">
 <jsp:setProperty name="EmpData“„ property="empNumber" value="1" />
 </jsp:useBean>
 <H1>Employee #1</H1>
 <B>Name:</B> <%=EmpData.getName()%><BR>
 <B>Address:</B> <%=EmpData.getAddress()%><BR>
 <B>City/State/Zip:</B>
 <%=EmpData.getCity()%>,
 <%=EmpData.getState()%>
 <%=EmpData.getZip()%>
</BODY>
</HTML>
```

Employee #1

Name: Jaime Husmillo
Address: 2040 Westlake N
City/State/Zip: Seattle, WA 98109

JSP (2)

Bean:

```
class FetchEmpDataBean {
 private String name, address, city, state, zip;
 private int empNumber = -1;

 public void setEmpNumber(int nr) {
  empNumber = nr;
  try {
   Connection con = DriverManager.getConnection("jdbc:db2:myDB","login","pwd");
   Statement stmt = con.createStatement();
   ResultSet rs = stmt.executeQuery ("SELECT * FROM Employees WHERE EmployeeID=" + nr);
   if (rs.next()) {
    name = rs.getString ("Name");
    address=rs.getString("Address");
    city = rs.getString ("City");
    state=rs.getString("State");
     zip=rs.getString("ZipCode");
   }
   rs.close();
  stmt.close();
  con.close();
  } catch (SQLException e) { //...}
 }
 public String getName() { return name; }
 public String getAddress() { return address; } ...
```

PHP (PHP: Hypertext Preprocessor)

- Open Source Skriptsprache zur Einbettung in HTML-Seiten
  - angelehnt an C, Java und Perl
  - besonders effizient bei der Erzeugung und Auswertung von HTML-Formularen
- Prinzip
  - PHP Server generiert dynamisch HTML
  - Übergabe an den Web-Server → Weiterleitung an Web-Browser
  - Web-Browser interpretiert HTML-Code vom Web-Server
- Beispiel

```
<?php echo "<b>Diesen Text bitte fett darstellen !</b>"; ?>
```

generiert →

```
<b>Diesen Text bitte fett darstellen !</b>
```

interpretiert →

Diesen Text bitte fett darstellen !

PHP – Variablen und Arrays

- Variablen und Datentypen
  - Variablennamen beginnen mit **$:** *$name*, *$address*, *$city*, *$zip_code*
  - Keine Deklaration von Variablen
  - Variablentyp wird automatisch zugewiesen / angepasst

    $name = “Ernie“; (String)
    $zip_code = 04103; (Integer → Ganze Zahl)
    $price = 1.99; (Double → Gleitkomma Zahl)

- Arrays
  - Indexiert: Zugriff auf Inhalt über numerischen Index

    $zahlen = array(1, 2, 3, 4);
    $zahlen[3] = 0;
  - Assoziativ: Zugriff über Stringschlüssel

    $address[“street“] = “Johannisgasse 26“;
    $address[“zip_code“] = 04103;
    $address[“city“] = “Leipzig“;
  - Kombination möglich (mehrdimensionale Arrays), z.B. *$addresses*[4][“zip_code“]

PHP – DB Anbindung

- Grundlegendes Vorgehen
  1. Aufbau einer Verbindung zum Datenbank-Server
  2. Auswahl einer Datenbank
  3. Interaktion mit Datenbank über SQL
  4. Verarbeitung und Präsentation von Ergebnissen

- DB-Anbindungsmöglichkeiten
  - Spezielle Module je nach DBS: MySQL, MS SQL, PostgreSQL, ...
  - Nutzung erweiterter Bibliotheken/Module für DBS-unabhängigen Zugriff
    - einheitliche Schnittstelle für unterschiedliche DBS
    - Beispiele: **PDO**, PEAR::DB
    - Vorteil: DBS kann ausgetauscht werden, Implementierung bleibt

PHP – DB Anbindung (Beispiele)

- MySQL

```
<?php
        $con = mysqli_connect("host", "user", "password");
        mysqli_select_db($con, "myDB");

        $result = mysqli_query($con, "SELECT * FROM Employees WHERE EmployeeID = 1");
        ...
?>
```

- MS SQL

```
<?php
        $con = mssql_connect("host", "login", "password");
        mssql_select_db("myDB", $con);

        $result = mssql_query("SELECT * FROM Employees WHERE EmployeeID = 1", $con);
        ...
?>
```

PHP – DB Anbindung (Beispiele 2)

- MySQL / MS SQL via PEAR::DB

```
<?php
        require_once("DB.php");

        $module = "mysql";    //$module = "mssql";
        $con = DB::connect("$module://user:password@host/myDB");

        $result = $con->query( "SELECT * FROM Employees WHERE EmployeeID = 1");
        ...
?>
```

- MySQL / MS SQL via PDO

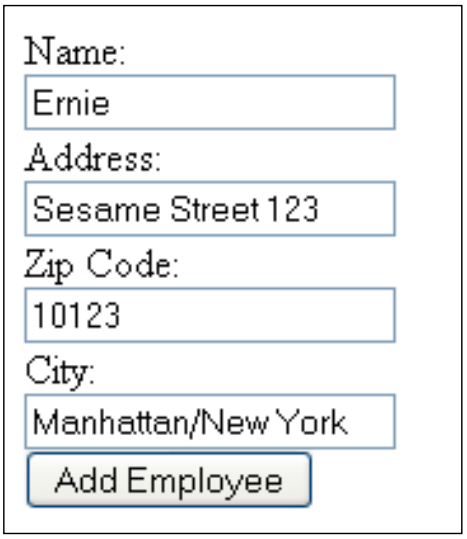
```
<?php
        $module = "mysql";    //$module = "mssql";
        $con = new PDO("$module:host=myHost; dbname=myDB", "user", "password");

        $result = $con->query("SELECT * FROM Employees WHERE EmployeeID = 1");
        ...
?>
```

PHP – Beispiel-Anwendung

- Beispiel: Webbasierte Verwaltung von Mitarbeitern
- 1. Anwendungsfall: Einfügen eines neuen Mitarbeiters

Eingabemaske

```
<form method="post" action="insert_employee.php">
    Name:<br/>
    <input type="text" name="full_name"/><br/>

    Address:<br/>
    <input type="text" name="address"/><br/>

    Zip Code:<br/>
    <input type="text" name="zip_code"/><br/>

    City:<br/>
    <input type="text" name="city"/><br/>

    <input type="submit" name="add_employee" value="Add employee"/>
</form>
```



PHP – Beispiel-Webanwendung (2)

- Verarbeitung der Daten in PHP
  - Variablen aus dem Formular sind über globales assoziatives PHP-Array $_POST verfügbar: <input type='text' name='city'/> +
    → $_POST['city'] enthält Wert „Manhattan/New York"

  - Verbindung zum DB-Server erstellen
  - Anwendungsfall abarbeiten

```
<?php
  $con = new PDO("$mysql:host=myHost; dbname=myDB", "user", "password");


  if(isset($_POST["add_employee"])) {
    $full_name= $_POST["full_name"];
    $address = $_POST['address"];
    $zip_code = $_POST["zip_code"];
    $city= $_POST["city"];

    $sql = "INSERT INTO Employees(Name, Address, ZipCode, City)
              VALUES($full_name, $address, $zip_code, $city)";

    $result = $con->exec($sql);
    if ($result == 0 ) {
        echo "Error adding new employee !";
    } else {
        echo "New employee sucessfully inserted !";
    }
  }
?>
```

PHP – Beispiel-Webanwendung (3)

- 2. Anwendungsfall: Auflisten aller Mitarbeiter (mit Prepared Statement)

```
<html>
  <body>
<?php
    $con = new PDO("$mysql:host=myHost; dbname=myDB", "user", "password");

    $sql= "SELECT * FROM Employees WHERE CITY LIKE ?";
    $stmt= con->prepare($sql);
    $stmt->execute(array("%New York%"));

    $rows= $stmt->fetchAll(PDO::FETCH_ASSOC);
    foreach($rows as $row) {
?>
        <h2>Employee # <?php echo $row["EmployeeID"]?></h2><br/><br/>
        <b>Name:</b>   <?php echo $row["Name"]?> <br/>
        <b>Address:</b> <?php echo $row ["Address"]?> <br/>
        <b>ZIP/City:</b><?php echo $row["ZipCode"] . "," . $row["City"]?><br/><br/>
<?php
    }
?>
  </body>
</html>
```

Employee #4711

Name: Ernie
Address: Sesame Street 123
ZIP/City: 10123 Manhattan/New York

Employee #4712

Name: Bert
Address: Sesame Street 125
ZIP/City: 10123 Manhattan/New York

Vergleich JSP - PHP

- beides sind serverseitige Skriptsprachen zur Einbindung in HTML-Seiten
  - Seiten müssen gelesen und interpretiert werden
- JSP
  - Java-basiert, plattformunabhängig,
  - Nutzung von JDBC für einheitlichen DB-Zugriff
  - unterstützt Trennung von Layout und Programmlogik (Auslagerung in Beans möglich)
  - großer Ressourcenbedarf für Java-Laufzeitumgebung
- PHP
  - einfache Programmierung durch typfreie Variablen und dynamische Arraystrukturen, fehlertolerant, Automatismen zur Verarbeitung von Formularfeldern
  - viele Module z. B. für Bezahldienste, XML-Verarbeitung
  - PHP-Nachteile: unterstütze DB-Funktionalität abhängig von jeweiligem DBS; umfangreiche Programmlogik muss als externes Modul (meist in C, C++) realisiert werden

Zusammenfassung

- statisches (eingebettetes) SQL
  - hohe Effizienz, relativ einfache Programmierung
  - begrenzte Flexibilität (Aufbau aller SQL-Befehle muss zur Übersetzungszeit festliegen)
  - SQLJ: eingebettetes SQL für Java
- Cursor-Konzept zur satzweisen Verarbeitung von Datenmengen
  - Operationen: DECLARE CURSOR, OPEN, FETCH, CLOSE
  - SQLJ: analoges Iteratorkonzept
- Call-Level-Interface (z.B. ODBC, JDBC)
  - erfordert keinen Präcompiler
  - Verwendung von dynamischem SQL
  - häufiger Einsatz
- Stored Procedures: Performance-Gewinn durch reduzierte Häufigkeit von DBS-Aufrufen
  - SQL-Standardisierung: Persistent Storage Modules (PSM)
  - umfassende prozedurale Spracherweiterungen von SQL

Zusammenfassung (2)

- JDBC: Standardansatz für DB-Zugriff mit Java
  - Prepared Statements schützen gegen SQL Injection
- viele Möglichkeiten zur Web-Anbindung von Datenbanken bzw. DB-Anwendungsprogrammen
- CGI; standardisiert, aber veraltet
  - keine Unterstützung zur Trennung von Layout und Programmlogik
- Einbettung von Programmcode in HTML-Seiten: JP, ASP, PHP …
- PHP: flexibler und leichtgewichtiger Ansatz
- größere (Unternehmens-) Anwendungen
  - Applikations-Server
  - JSP, Enterprise Java Beans, DB-Zugriff über JDBC / SQLJ