universität wien

Lektion 6

Datenbank Programmierung

Inhalt

Kopplungsarten zwischen Datenbank- und Programmiersprachen

- Erweiterung der Datenbanksprache um Programmierkonstrukte
  z.B. PL/SQL (4GL)
- Erweiterung von Programmiersprachen um Datenbankkonstrukte: Persistente Programmiersprachen oder auch ooDB
  z.B. Persistent Java
- Datenbankzugriff über API aus einer Programmiersprache
  z.B. ODBC, JDBC
- Einbettung der Datenbanksprache in eine Programmiersprache (C, Pascal, Java/SQLJ)
  z.B. Embedded SQL

Unser Ziel: Zugriff auf Datenbanken aus

1. API aus Objektorientierten Programmiersprachen (Java, C++) ➔ **JDBC**
2. Markup-Sprachen (HTML) ➔ **PHP**

Motivation

Eigenschaften

- SQL-Befehle an das Datenbanksystem richten
- Datenaustausch zwischen Programm und Datenbanksystem
- Typischerweise
  - ESQL bei prozeduralen Programmiersprachen (ESQL-Precompiler)
  - API bei objektorientierten Programmiersprachen
  - Skriptsprachen bei Markup-Sprachen

Limitierungen von SQL

- Nicht-deklarative Aktivitäten, wie Drucken, Report erstellen, Benutzerinteraktion, oder Unterstützung einer graphischen Benutzerschnittstelle

Lösung

- Formulieren der Anwendungsalgorithmen
- Benutzerschnittstelle in der klassischen Programmiersprache
- Datenspeicherung und –retrieval in SQL

JDBC – Java Database Connectivity

API für Low-Level-Datenbankzugriff

- Interface für den (entfernten) Datenbankzugriff von Java-Programmen aus
- Teil des SDK (java.sql.*)

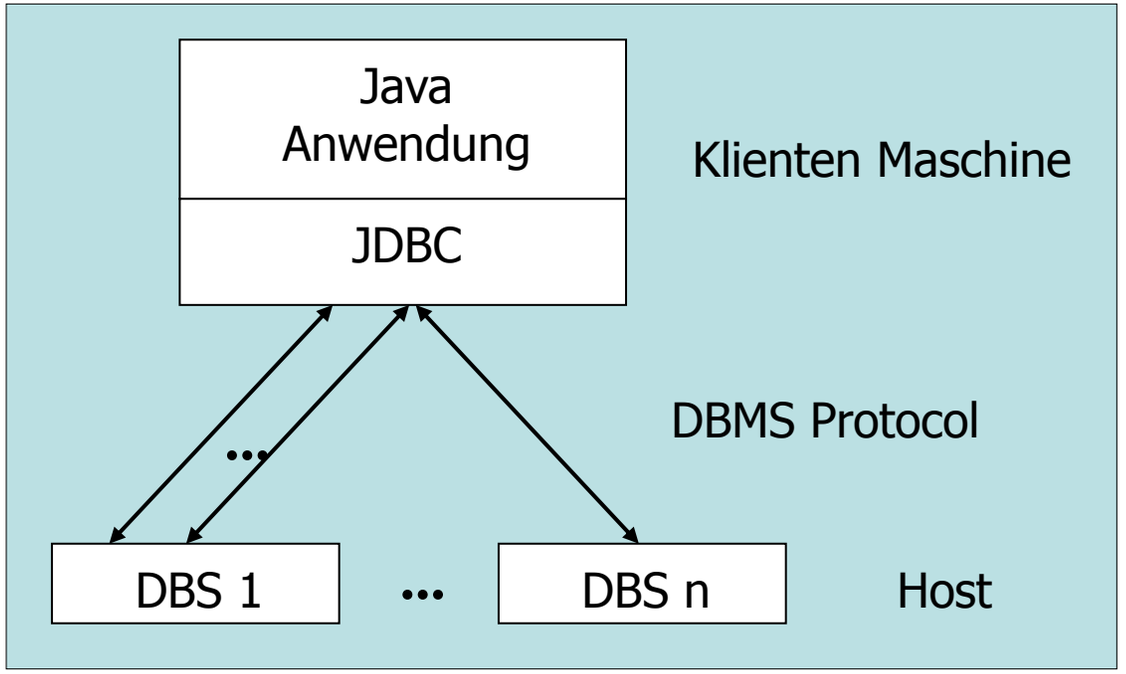
Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden

setzt die Idee von ODBC (Open DataBase Connectivity; ein 1992 entwickelter Standard zum Zugriff auf Datenbanken aus Programmiersprachen) auf Java um

gemeinsame Grundlage ist der X/Open SQL CLI (Call Level Interface) Standard

Alle JDBC-Anwendungen (Java Database Connectivity) durchlaufen die folgenden Schritte:

1. Laden eines JDBC-Treibers
2. Herstellung der Verbindungen zu Datenbanken
3. Anfragen erstellen
4. Anfragen ausführen
5. Ergebnisse abfragen
6. Verbindungen zu Datenbanken schließen



Zwei-Schichten Architektur

JDBC - Beispiel

Aus der Tabelle **konto** sollen alle Konten aufgelistet werden

```
import java.sql.*;
import oracle.jdbc.driver.*;

public class Select {
  public static void main(String args[]) {
    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      String dburl = "jdbc:oracle:thin:@vier.cs.univie.ac.at:1521:o10g ";
      Connection con = DriverManager.getConnection(dburl, "schiki", "pwd");

      Statement stmt = con.createStatement();
      String skontostand = "SELECT * FROM konto";
      ResultSet rs = stmt.executeQuery(skontostand);

      while (rs.next()) {;
        String konr = rs.getString("konr");
        System.out.println(konr + " " + rs.getInt("kontostand"));
      }

      rs.close();
      stmt.close();
      con.close();
    } catch( Exception e ) {System.err.println(e.getMessage());};
  }
}
```

Treiber laden

```
// load the JDBC driver
try {
  Class.forName("oracle.jdbc.driver.OracleDriver");
} catch (Exception e) {
  System.out.println("Failed to load JDBC driver.");
  return;
}
```

Treiber wird explizit mit **`Class.forName("DBTreiber")`** geladen

- Der Treiber registriert sich beim Laden beim Treiber-Manager

Treiber-Manager ist ein Objekt der Klasse *DriverManager* mit den Aufgaben

- JDBC-Treiber verwalten
- Log-Informationen bei Bedarf ausgeben
- Herstellung von Datenbankverbindungen

Verschiedene Treiber-Typen (DBMS-unabhängige, DBMS-spezifische)

In aktuellen Java-Versionen ist dieser Aufruf nicht mehr notwendig

DB-Verbindung herstellen

```
// get a connection
try {
  Connection con;
  String url = "jdbc:oracle:thin:@vier.cs.univie.ac.at:1521:o10g";
  con = DriverManager.getConnection(url, "schiki","pwd");
} catch (Exception e) {
  System.err.println("problems connecting");
}
```

Objekt der Klasse *Connection* repräsentiert eine Verbindung zu einem DBMS

wird erzeugt durch

```
Connection con =
  DriverManger.getConnection(<jdbcurl>,"name","passwd");
```

mehrere Verbindungen zu einem/mehreren DBMS möglich

DB-Verbindung herstellen (1)

JDBC-URL **<jdbcurl>**

bezeichnet die DB, zu der die Verbindung aufgebaut werden soll

sieht ähnlich aus wie ein URL

hängt vom Treiber ab <jdbcurl>::=jdbc:<Subprotokoll>:<Subname>

<Subprotokoll> bezeichnet den Treibernamen (z.B.: **`oracle:thin`**) bzw. den Verbindungsmechanismus (z.B.: **`odbc`**).

<Subname> bezeichnet die DB (hängt von <Subprotokoll> ab).

Schreibweise für <Subname>, falls er eine Netzwerkadresse enthält:
@<Rechnername>:<Port>:<Subsubname>

Beispiel

```
String url =
 "jdbc:oracle:thin:@vier.cs.univie.ac.at:1521:o10g";
```

Host

Port

DB-Name

Subprotokoll

Subname

Anfrage erstellen

```
// create a statement
  try {
    Statement stmt = con.createStatement();
  } catch (Exception e) {
    System.err.println("problems creating a statement");
  }
```

Objekt der Klasse *Statement* repräsentiert eine SQL-Anweisung

SQL-Anweisung kann DDL-, DML- oder Query-Statement sein

Anweisung existiert im Kontext der DB-Verbindung

wird erzeugt durch `Statement stmt = con.createStatement();`

wird geschlossen durch `stmt.close();`

Anfrage ausführen

```
// execute a statement
  try {
    String skontostand = "SELECT * FROM konto";
    ResultSet rs = stmt.executeQuery(skontostand);
  } catch (Exception e) {
    System.err.println("problems executing a statement");
  }
```

Ausführen einer Select-Anw. durch **`stmt.executeQuery(<sql>);`**

- **`<sql>`** stellt das Select-Statement dar

das Select-Statement wird über die DB-Verbindung an das DBMS gestellt und bearbeitet

das Ergebnis der Select-Anweisung (Ergebnisrelation) wird vom DBMS über die DB-Verbindung an das Programm zurückgeliefert

Ergebnis abfragen

```
ResultSet rs = stmt.executeQuery(skontostand);
rs.next();
String konr = rs.getString("konr");
System.out.println(konr);
```

das Ergebnis einer Select-Anweisung repräsentiert ein Objekt der Klasse *ResultSet*, das die Methode **`executeQuery(<sql>)`** der Klasse *Statement* als Returnwert liefert

Ergebnis-Tupel wird erreicht durch **`rs.next()`**

ein Wert aus dem aktuellen Ergebnis wird abgefragt mit

`rs.getXXX(<Spaltennummer>)` oder

`rs.getXXX(<Spaltenname>)`

`XXX` ist der jeweilige Java-Typ der zum SQL-Typ kompatibel sein sollte.

Zuordnung Java-Typen/SQL-Typen

Kompatibilität

SQL type	Java type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Verwendung bei getXXX

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x
getBytes														X	X	x			
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimestamp											x	x	x				x		X
getAsciiStream											x	x	X	x	x	x			
getUnicodeStream											x	x	X	x	x	x			
getBinaryStream														x	x	X			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Ein "X" bedeutet, dass die **getXXX**-Methode für den jeweiligen Typ gültig ist.
Ein "**X**" bedeutet, dass die **getXXX**-Methode für den jeweiligen Typ empfohlen ist.

DB-Verbindung schließen

```
// close a connection
try {
  rs.close();
  stmt.close();
  con.close();
} catch (Exception e) {
  System.err.println("problems disconnecting");
}
```

Eine DB-Verbindung wird geschlossen durch **`con.close()`**

Ressourcen (z. B. Objekte der Klassen *Statement* und *ResultSet*), die von *con* verwendet werden, werden freigegeben

Vermeidung von Resource-Leaks durch try-with-resources-Statement in aktuellen Java-Versionen

Cursorkonzept

Problem:

- SQL – mengenorientiert, 3GL – tupelorientiert
- in 3GL sind keine Operatoren für Relationen vorhanden, d.h., eine Relation muss Tupel für Tupel mit Hilfe einer Schleife durchgegangen werden

Lösung: **Cursorkonzept**

- Ein Cursor ist eine Datenstruktur zum Zwecke des Granularitätsausgleiches zwischen SQL und Wirtssprache

Beispiel

```
rs = stmt.executeQuery("SELECT * FROM konto");
while(rs.next()) {
  String konr = rs.getString(1);
  int kontostand = rs.getInt("kontostand");
  System.out.println(konr + " " + kontostand);
}
rs.close();
```

Cursorkonzept (1)

Die Tupel-Ergebnismenge einer Select-Anweisung ist in einem *ResultSet*-Objekt (*rs*) abgelegt

Jedes *ResultSet*-Objekt realisiert eine **Cursor**("Zeiger")-Struktur, mit der auf die einzelnen Tupel zugegriffen werden kann

Ein Cursor ist solange gültig, bis das *ResultSet*- oder das zugehörige *Statement*-Objekt "geschlossen" wird

- der Cursor steht nach der Instanziierung vor dem ersten Satz (Tupel, Zeile)
- der nächste Satz wird erreicht durch **`rs.next()`**

Die Methode **`next()`** liefert als Rückgabewert einen boolschen Wert:

- **`true`**: weiterer Satz existiert
- **`false`**: keine weiterer Satz existiert

ein Wert aus dem aktuellen Satz wird abgefragt mit

- **`rs.getXXX(<Spaltennummer>)`** oder
- **`rs.getXXX(<Spaltenname>)`**
- **`XXX`** ist der jeweilige Java-Typ der zum SQL-Typ passen sollte

Behandlung von Null-Werten

Problem

- Der SQL-Wert "NULL" ist in Java nicht vorhanden
- Wenn der Wert der zuletzt gelesenen Spalte den Wert "NULL" (SQL) hat - also nicht definiert ist, wird keine Ausnahme erzeugt
- Statt dessen wird die entsprechende Java-Repräsentation zurückgegeben

Java-"null"	bei den Methoden getString, getBigDecimal, getBytes, getDate, getTime, getTimestamp, getAsciiStream, getUnicodeStream, getBinaryStream, getObject
0	bei den Methoden getByte, getShort, getInt, getLong, getFloat, getDouble
False	bei der Methode getBoolean

Lösung

- Aufruf von **`rs.getXXX(...);`**
- durch anschließenden Aufruf von **`boolean rc = rs.wasNull();`** feststellen, ob der übertragene Wert "NULL" (SQL) war
- Rückgabewert ist **`true`** wenn der Wert "NULL" (SQL) ist, ansonsten **`false`**

Precompilierte Anweisungen

Problem

- die durch die Klasse **`statement`** erzeugten Anweisungen werden jedes Mal vom DBMS überprüft und optimiert
- die mehrfache Ausführung von Anweisungen wird damit ineffektiv - Netzwerk und DBMS
  - (z.B. alle Bestellungen des heutigen Tages ermitteln)

Lösung

- Anweisung wird an das DBMS übertragen und dort für die Ausführung vorbereitet (Parsen, Integritätsprüfung, Optimierung, Zugriffsplan-Erstellung,...) und gespeichert
- die (beliebige) Ausführung dieser vorbereiteten (precompilierten) Anweisungen wird durch den Client angestoßen

Precompilierte Anweisungen (1)

```
String strSql = "select BNr from Bestellung " +
                " where TO_CHAR(BestDatum, 'DD-MM-YY') = " +
                "       TO_CHAR(SYSDATE, 'DD-MM-YY')";
PreparedStatement prepStmt = con.prepareStatement(strSql);
...
prepStmt.executeQuery();
...
prepStmt.executeQuery();
...
```

Verwendung der Klasse PreparedStatement statt Statement

Erzeugen durch

`PreparedStatement prepStmt = con.prepareStatement(<sql>);`

Ausführen des precompilierten Select durch **`prepStmt.executeQuery();`**

Methode **`executeQuery()`** hat keine Parameter

Host-Variable in SQL-Anweisungen

Problem

Anweisungen, die sich in der Ausprägung von Werten unterscheiden, werden jedes Mal vom DBMS überprüft und optimiert (precompilieren ist keine Lösung)

die mehrfache Ausführung von ähnlichen Anweisungen wird damit ineffektiv; z.B. mehrere Tupel in eine Tabelle einfügen oder die Klappe eines Mitarbeiters abfragen

Beispiel

```
INSERT INTO kunde VALUES ('Meier', 'Hauptstraße', 'Wien');
INSERT INTO kunde VALUES ('Müller', 'Ring', 'Graz');
INSERT INTO kunde VALUES ('Huber', 'Nebengasse', 'Linz');
```

Lösung

Referenzierung von Host-Variablen in precompilierten SQL-Anweisung, wo eine Konstante stehen darf

Host-Variable (1)

```
String strSql = "SELECT * FROM konto WHERE kontostand > ?";
PreparedStatement prepStmt = con.prepareStatement(strSql);
...
prepStmt.setInteger(1,1000);
prepStmt.executeQuery();
...
prepStmt.setInteger(1,2000);
prepStmt.executeQuery();
```

Parameter setzen

- SQL-Anweisung enthält "?" als Platzhalter für ("IN-")Parameter
- Parameter für jedes "?" i.a. setzen mit
  `prepStmt.setXXX(<Position>,<Wert>);`
  (**`xxx`** ist der passende Datentyp des Parameters)
- Parameter müssen gesetzt sein, bevor die Anweisung ausgeführt wird
- alle Parameter zurücksetzen mit **`prepStmt.clearParameters();`**
- SQL-"NULL"-Wert wird gesetzt mit **`setNull(<Position>,<SQLTyp>)`**

Host-Variable (2)

Typ-Verwendung bei setXXX

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
String	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
java.math.BigDecimal	x	x	x	x	x	x	x	x	x	x	x	x	x						
Boolean	x	x	x	x	x	x	x	x	x	x	x	x	x						
Integer	x	x	x	x	x	x	x	x	x	x	x	x	x						
Long	x	x	x	x	x	x	x	x	x	x	x	x	x						
Float	x	x	x	x	x	x	x	x	x	x	x	x	x						
Double	x	x	x	x	x	x	x	x	x	x	x	x	x						
byte[]														x	x	x			
java.sql.Date											x	x	x				x		x
java.sql.Time											x	x	x					x	
java.sql.Time- stamp											x	x	x				x	x	x

Ein "X" bedeutet, dass die **setXXX**-Methode für den jeweiligen Typ gültig ist.

DDL/DML-Statements ausführen

```
// create table
String ssql = "CREATE TABLE Projekt (pnr NUMBER, bez VARCHAR(32));";
stmt.executeUpdate(ssql);

// insert row into table
stmt.executeUpdate("INSERT INTO Projekt VALUES (68, 'DB');");

// update rows
ssql = "UPDATE Projekt SET bez = 'SW2 Datenbanken' WHERE bez = 'DB'";
int count = stmt.executeUpdate(ssql);
```

Ausführung der Anweisung durch **`int count = stmt.executeUpdate(<sql>)`**

Parameter **`<sql>`** kann eine DML-Anweisung (**`INSERT`**, **`DELETE`**, **`UPDATE`**) oder eine DDL-Anweisung (**`CREATE`**, **`ALTER`**, **`DROP TABLE`**, ...) sein

im Gegensatz zu einer Select-Anweisung liefert ein DDL/DML-Statement (Änderungs-Statement) keine Ergebnis-Menge zurück

Rückgabewert ist bei DML-Anweisungen die Anzahl der geänderten Tupel, sonst 0

Transaktionen

Eine Folge von Anweisungen kann als Transaktion ausgeführt oder zurückgenommen werden

Eine neue DB-Verbindung befindet sich im "Auto-Commit"-Modus, d.h. jede Anweisung wird als Transaktion behandelt und bestätigt

Wechsel in den "Transaktions"-Modus mit

```
con.setAutoCommit(false);
```

Durchführung der Änderungen mit

```
con.commit();
```

Rücknahme mit

```
con.rollback();
```

Durch

```
con.setTransactionIsolation(int level);
```

kann festgelegt werden, dass z.B. geänderte aber noch nicht bestätigte Werte bei parallelen Select-Anweisungen angezeigt werden, wenn dies vom DBMS unterstützt wird

Ausnahmebehandlung

SQLException

JDBC-Befehle können Ausnahmen vom Typ SQLException auslösen

Behandlung mit **`try { ... } catch (...) {...}`**

gibt Fehlerinformationen des DBMS weiter, evtl. eine Kette von Fehlermeldungen

typische Behandlung:

```
catch (SQLException esql) {
    while (esql != null) {
        System.out.println(esql.toString());
        System.out.println("SQL-State: " + esql.getSQLState());
        System.out.println("ErrorCode: " + esql.getErrorCode());
        esql = esql.getNextException();
    }
}
```

SQLWarning

liefert Warnungen des DBMS

Abfrage nicht in **`try...catch`** sondern durch

```
SQLWarning warn = con.getWarnings();
```

Durchlauf und Ausgabe wie oben

JDBC - Zusammenfassung

JDBC (Java Database Connectivity)

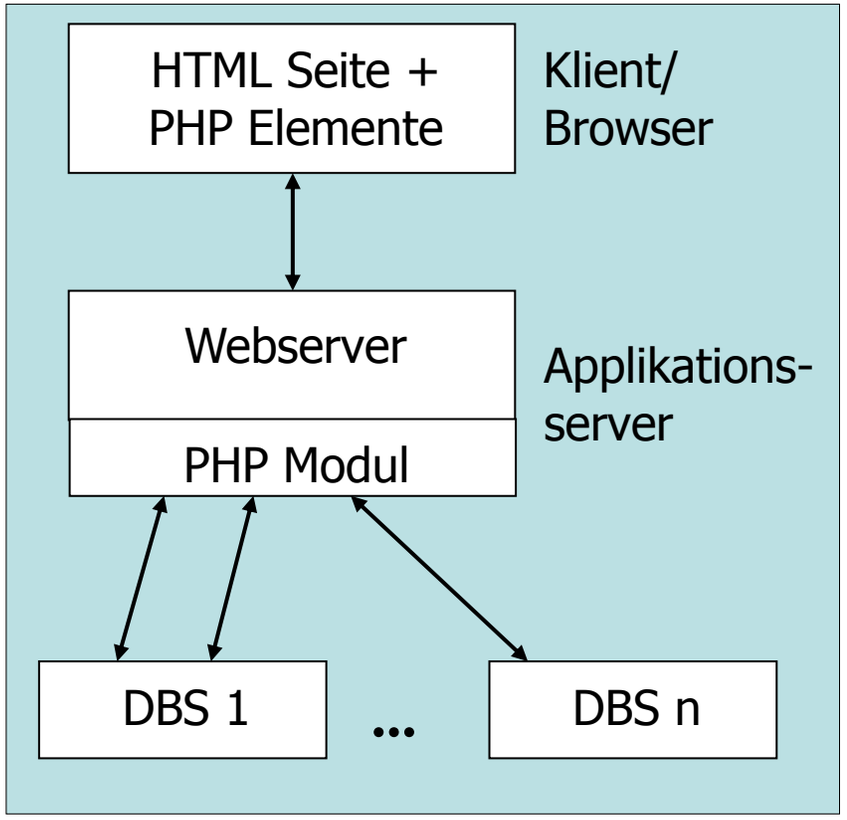
- Ist ein Java Application Programming Interface (API)
- Paket *java.sql*
- enthält Klassen, Interfaces und Methoden, um SQL-Anweisungen an ein Datenbanksystem (DBS) zu senden
- plattform- und datenbankunabhängige Low-Level-Schnittstelle
- einfach zu verwenden (wenige Klasse, Parametrisierung)

Nachteile

- kein Mapping von Tabellen zu Java-Klassen möglich
- Embedded SQL Syntax fehlt

PHP: Hypertext Preprocessor

- Server-seitige Skriptsprache
  - Wird interpretiert, nicht kompiliert
- Freie Software, http://www.php.net
  - es existiert kein Standard, nur defacto Standard durch The PHP Group
- Wurde ursprünglich entwickelt, um Webseiten dynamisch zu erstellen
- PHP Sprachelemente werden in eine HTML Seite eingebettet und werden vom Web Server durch ein spezifisches PHP Modul interpretiert
  - erlauben machtvolle Operationen aus HTML heraus
- gute Eignung für Datenbankzugriffe
  - PHP & MySQL gewann den “Database of the Year”-Preis auf der Web98



Drei-Schichten Architektur

PHP Beispiel – HTML Seite mit eingebetteten PHP Elementen

universität wien

```
<?php
  $DBUSER = 'schiki'; $DBPASS = 'pwd'; $DBDATABASE = 'vier_o10g';
  $conn = oci_connect($DBUSER, $DBPASS, $DBDATABASE);
  if (!$conn) exit;
?>
<html> <head> </head> <body> <h1>Liste aller erfassten Konten</h1>
<?php
  $sql = "SELECT * FROM konto";
  $stmt = oci_parse($conn, $sql);
  oci_execute($stmt);
?>
<table border='1'><thead><tr><th>Kontonummer</th><th>Kontostand</th></tr></thead>  <tbody>
<?php
  while ($row = oci_fetch_assoc($stmt)) {
    echo "<tr>";
    echo "<td>" . $row['konr'] . "</td>";
    echo "<td>" . $row['kontostand'] . "</td>";
    echo "</tr>";
  }
?>
</tbody> </table> <div>Insgesamt <?php echo oci_num_rows($stmt); ?> Konten erfasst!</div>
<?php
  oci_free_statement($stmt);
  oci_close($conn);
?>
</body> </html>
```

DB-Verbindung herstellen

```
<?php
   $DBUSER = 'schiki';
   $DBPASS = 'pwd';
   $DBDATABASE = 'vier_o10g';
   $conn = oci_connect($DBUSER, $DBPASS, $DBDATABASE);
   if (!$conn) exit;
?>
```

Eine Verbindung zur Datenbank wird durch Verwendung der Funktion **`oci_connect`** unter Bekanntgabe des Datenbankbenutzers, dessen Passwort und der entsprechenden Datenbank aufgebaut

Das **`if`**-Statement bricht den Seitenaufbau ab, falls keine Verbindung zur Datenbank hergestellt werden kann.

HTML Datei erstellen

```
<html>
<head> </head>
<body>
<h1>Liste aller erfassten Konten</h1>
```

Kopf der HTML Seite, die später die Ergebnisse der Datenbankabfrage im Browser des Klienten darstellt

Ausführung der SQL Anweisung

```
<?php
   $sql = "SELECT * FROM konto";
   $stmt = oci_parse($conn, $sql);
   oci_execute($stmt);
?>
```

Hier wird das SQL Statement formuliert und mittels **`oci_parse`** Funktion "geparsed" und somit auf eine korrekte Syntax kontrolliert.

Mittels **`oci_execute`** wird das Statement (hier ein **`SELECT`**) ausgeführt.

HTML Tabellenkopf

```
<table border='1'>
   <thead>
     <tr>
       <th>Kontonummer</th>
       <th>Kontostand</th>
     </tr>
   </thead>
   <tbody>
```

Erstellung einer Tabelle mit Spaltenüberschriften zur Ausgabe der Ergebnisse

Abarbeitung der Ergebnisse

```
<?php
  while ($row = oci_fetch_assoc($stmt)) {
    echo "<tr>";
    echo "<td>" . $row['konr'] . "</td>";
    echo "<td>" . $row['kontostand'] . "</td>";
    echo "</tr>";
  }
?>
```

Ab hier beginnt nun die Auswertung der zurückgelieferten Ergebniszeilen

Mittels **`oci_fetch_assoc`** wird jeweils eine Zeile des Ergebnisses ausgelesen und über die Spaltennamen (assoziativ) auf die Werte der entsprechenden Attribute zugegriffen

Die Ausgabe der Werte erfolgt mittels **`echo`**

Ausgabe der Anzahl der Ergebnisse

```
</tbody>
 </table>
   <div>Insgesamt
     <?php echo oci_num_rows($stmt); ?>
     Konten erfasst!
   </div>
```

Ausgabe der Anzahl der zurückgegebenen Zeilen mittels

`oci_num_rows`

Freigabe der Ressourcen und Abmelden

```
<?php
   oci_free_statement($stmt);
   oci_close($conn);
?>
```

Ganz zum Schluss sollte jedes durchgeführte Statement wieder mittels **`oci_free_statement`** freigegeben werden

Die Datenbankverbindung **`$conn`** wird mit **`oci_close`** beendet

```
</body>
</html>
```

Abschließend die End-Tags unseres HTML Dokuments