2. DB-Anwendungsprogrammierung
(Teil 2)

- JDBC
  - wesentliche JDBC-Operationen, Transaktionskontrolle
  - Problem der SQL Injection
  - Nutzung von Stored Procedures
- SQLJ
  - Iteratorkonzept
- Web-Anbindung
  - CGI
  - Java Server Pages
  - PHP

JDBC (Java Database Connectivity)

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

Java-Anwendung

JDBC-API

JDBC-Treiber-Manager

Treiber

Treiber

JDBC-ODBC Bridge-Treiber

ODBC

Partial-Java-Treiber

Native-Treiber

JDBC-Net-Treiber

Native-Protocol-Treiber

Client

JDBC-Server

Server

- 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)

DriverManager

getConnection (JDBC-URL)

Connection

createStatement()

prepareStatement (SQL)

prepareCall ({call Proc})

Statement

PrepareStatement

CallableStatement

setXXX(Param)

setXXX(Param)

execute Update(SQL)

execute Query(SQL)

execute Update()

execute Query()

execute Update()

execute Query()

registerOutParameter(Param)

execute()

ResultSet

ResultSet

ResultSet

next()
getXXX (Param)

getXXX(Param)

getXXX(Param)

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

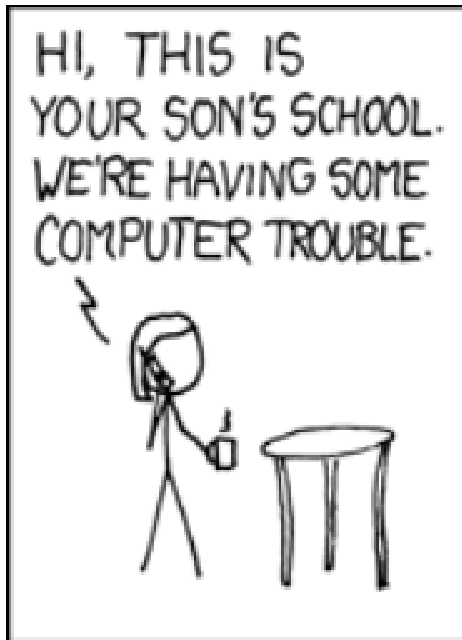
PNR	NAME	GEHALT
34	Mey	
35	Schultz	
37	Abel	

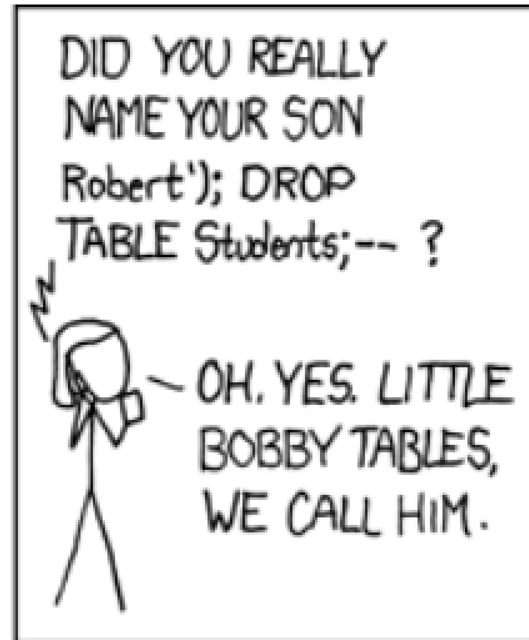
SQL INJECTION (2)

- 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"





URL: https://xkcd.com/327/

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

Client (Browser)

Request (HTTP)

Response (HTML, XML, …)

Web -Server

JavaScript,
AJAX,
ActiveX,
Plugin,
Applet …

CGI Servlet

generiert HTML

ASP JSP PHP

erweitert HTML

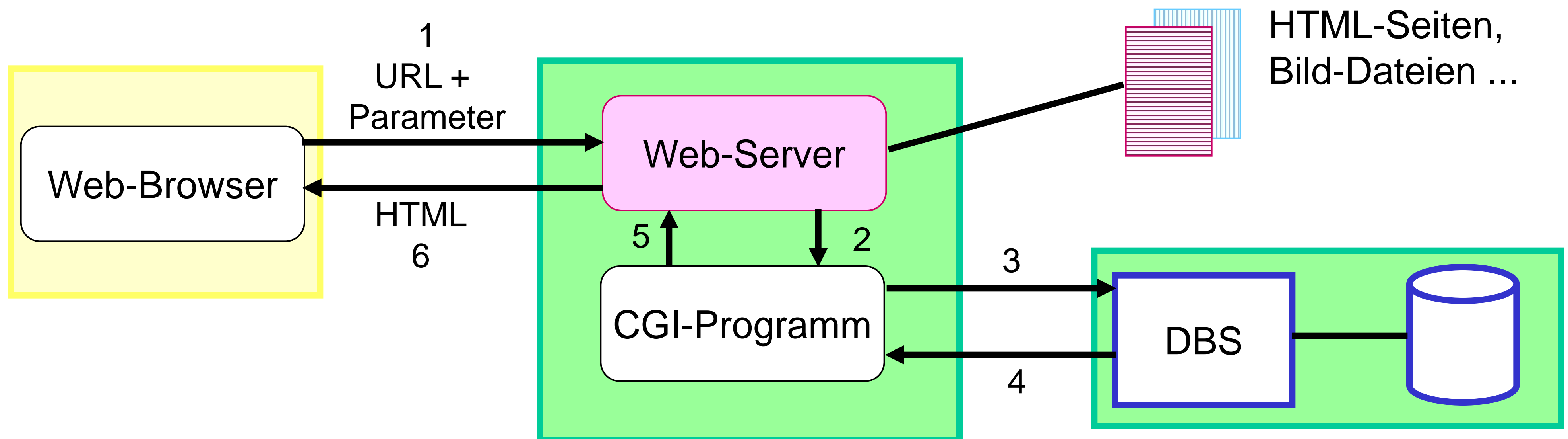
Programme / Skripte

JDBC
SQLJ

DB-Server

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

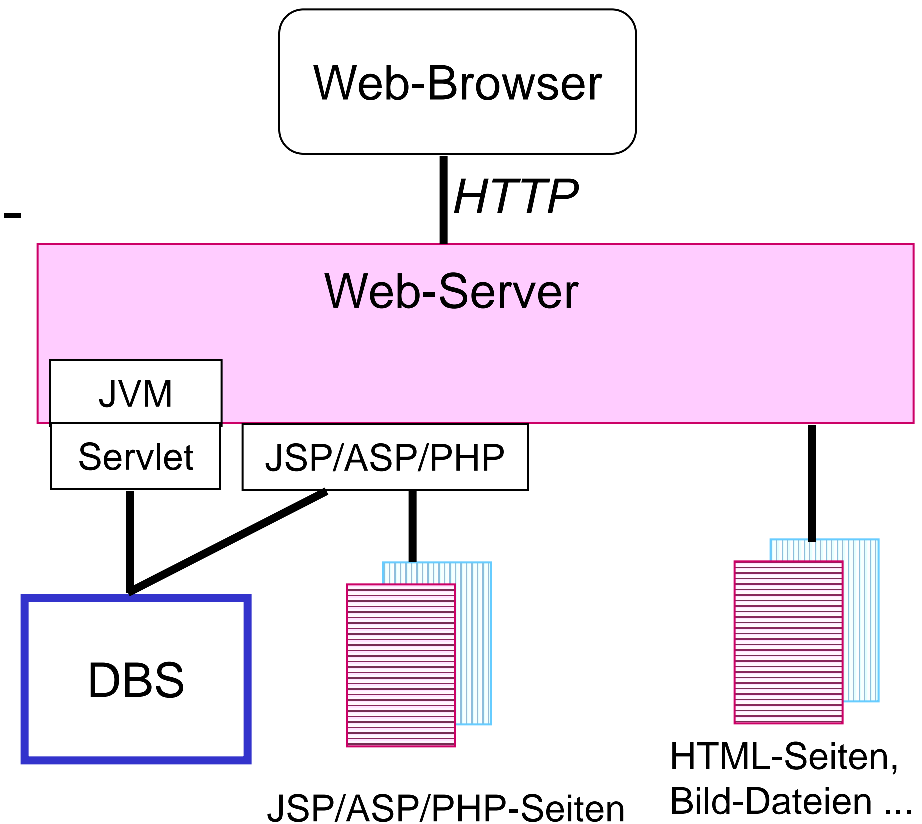
CGI-Kopplung (2)

- 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 →

Diesen Text bitte fett darstellen !

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>
```

Name:
Ernie
Address:
Sesame Street 123
Zip Code:
10123
City:
Manhattan/New York
Add Employee

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‘/> + [Manhattan/New York]
    → $_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

- JDBC: Standardansatz für DB-Zugriff mit Java
  - Call-Level-Interface (erfordert keinen Präcompiler)
  - Verwendung von dynamischem SQL
  - 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: JSP, ASP, PHP …
- PHP: flexibler und leichtgewichtiger Ansatz
- größere (Unternehmens-) Anwendungen
  - Applikations-Server
  - JSP, Enterprise Java Beans, DB-Zugriff über JDBC / SQLJ