6.3.2 Java und relationale Datenbanken – JDBC und weiterführende Konzepte

JDBC
Java Database Connectivity
In Analogie zu ODBC (MS) ist JDBC eine Schnittstelle zu relationalen Datenbanken, die es erlaubt, mit Hilfe geeigneter Interfaces, die die Datenbankkommunikation kapseln, DBMS-unabhängig auf relationale Datenbanken zuzugreifen.

SQLJ – Java Embedded SQL
Embedded SQL erlaubt die direkte Verwendung von SQL- Anweisungen in Java. Hierbei werden die SQL-Anweisungen im Java-Quelltext durch das Schlüsselwort **#sql** gekennzeichnet.

ORM
Objektrelationales-Mapping
Ansätze zum ORM finden sich in der *Java Data Objects-Spezification (JDO), dem Data Access Object Pattern* oder in Frameworks zur Unterstützung des ORM (z.B. *Hibernate*). Die Spezifikation *JSR220* definiert den Standard für das ORM in Java.

JSP - Java Server Pages
JSF – Java Server Faces
Java Servlets und JSP bzw. JSF werden im Bereich web-basierter Applikationen eingesetzt (View). Der Zugriff auf die DB zur Generierung **dynamischer Web-Seiten** erfolgt dann i.d.R. durch DataAccessBeans (Model).

DataAccessBeans
DataAccessBeans kapseln den Zugriff auf relationale DB objektorientiert.

Das J2EE-Framework
Die J2EE-Technologie verwendet so genannte **Enterprise Java Beans (EJB),** um Controls, Businesslogik und Business-Daten zu kapseln. Hierbei repräsentieren die sogenannten **EntityBeans** die Businessobjekte einer Persistenzschicht.

6.3.2 SQLJ - Java Embedded SQL (1)

Zur Erinnerung ...

Embedded SQL ist ein integrierter Bestandteil des SQL-Standards. Er definiert SQL-Sprachkomponenten, die es ermöglichen,

- **statische SQL-Anweisungen** direkt in den Quelltext einer höheren Programmiersprache einzufügen, und

- Spaltenwerte aus Tabellenstrukturen der Datenbank Pogrammvariablen der höheren Programmiersprache (**Hostvariablen**) zuzuordnen (**Cursorkonzept**).

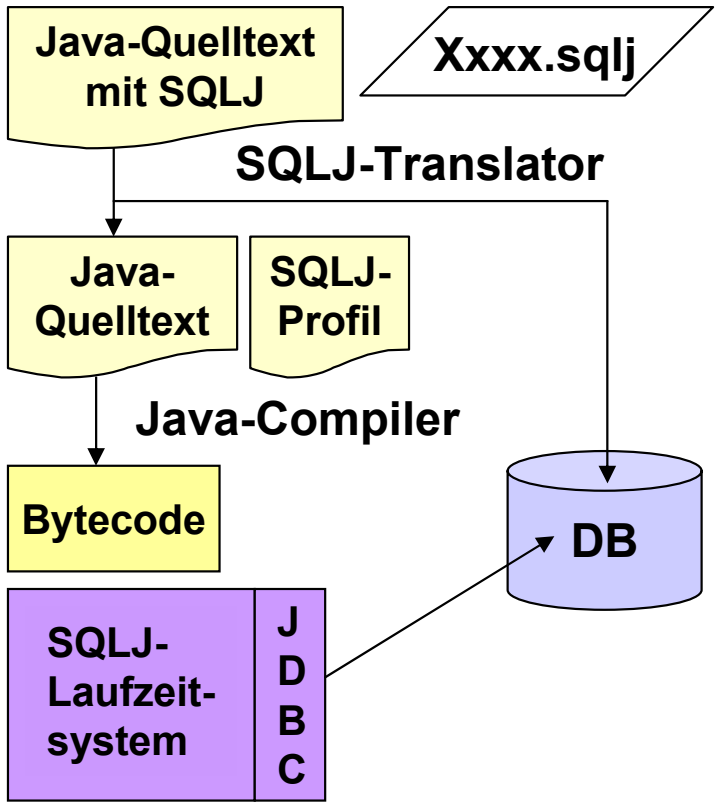
In höheren Programmiersprachen, die compiliert werden, wandelt i.d.R. ein Precompiler die SQL-Anweisungen mit Hilfe spezieller Bibliotheken vor der Compilierung in Anweisungen der entsprechenden Programmiersprache um.

6.3.2 SQLJ - Java Embedded SQL (2)

Bettet man SQL in Java ein, so übernimmt die Rolle des Precompilers der so genannte **SQLJ-Translator**, der aus einem SQLJ-Programm ein Java-Programm und so genannte SQLJ-Profile erzeugt, die Informationen über die eingebetteten SQL-Operationen enthalten.
Aufruf: **sqlj Xxxx.sqlj**

Der **Translator** führt auch SQL-Syntaxprüfungen und Schema-Prüfungen für die entsprechenden Datenbank aus.

Der **Java-Compiler** generiert anschließend aus dem Java-Programm den entsprechenden Java-Bytecode.



6.3.2 SQLJ - Java Embedded SQL (3)

- Das **SQLJ-Laufzeitsystem** besteht aus einer Menge von Java-Klassen, die im Java-Paket **sqlj.runtime** zur Verfügung gestellt werden und den Datenbankzugriff über JDBC realisieren.

- Analog zum direkten Zugriff über JDBC muss ein geeigneter **Datenbanktreiber** geladen werden und darüber hinaus eine **Default-Verbindung** definiert werden, die für alle SQL-Anweisungen genutzt wird:

```
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;

String driverClass =
      "sun.jdbc.odbc.JdbcOdbcDriver";

try {
   Class.forName(driverClass);
}
catch (ClassNotFoundException exc) {...}
try {
   Connection con =
   DriverManager.getConnection
   ("jdbc:odbc:MyJdbc");

   DefaultContext ctx =
      new DefaultContext(con);
   DefaultContext.setDefaultContext(ctx);

... }
```

6.3.2 SQLJ - Java Embedded SQL (4)

- Die **SQL-Anweisungen** stehen in geschweiften Klammern und werden von einem Semikolon abgeschlossen. Sie sind durch das Schlüsselwort **#sql** gekennzeichnet, das der SQL-Anweisung vorangestellt wird:

```
...
#sql {INSERT INTO teil VALUES ('MB_777','Motorrad')};
...
```

- Hostvariablen werden innerhalb einer SQL-Anweisung von SQL-Strukturelementen dadurch unterschieden, dass ihren Bezeichnern ein **Doppelpunkt** vorangestellt wird. Sie können so z.B. als Filter im Zusammenhang mit WHERE-Klauseln benutzt werden. Liefert eine SELECT-Operation **nicht mehr als eine einzige Ergebniszeile** (z.B. bei Filter mit Wert einer Primary Key-Spalte), so kann man die Spaltenwerte der Ergebniszeile direkt mit der **INTO-Klausel** entsprechenden Hostvariablen zuweisen:

```
String findBez(String tNummer) throws SQLException {
   String teileBez;
   #sql { SELECT tbez INTO :teileBez FROM teil WHERE tnr = :tNummer };
   return teileBez;
}
```

6.3.2 SQLJ - Java Embedded SQL (5)

- Erwartet man als Ergebnis einer SELECT-Operation dagegen eine **ungewisse Anzahl von Ergebniszeilen**, so wird die Ergebnismenge mit Hilfe eines **Cursors** gelesen, der die Struktur der Ergebnistabelle kennt und diese zeilenweise lesen und Hostvariablen entsprechenden Typs zuordnen kann.

- In SQLJ entspricht ein Cursor einem **Iterator-Objekt** (vgl. auch die Objekte vom Typ ResultSet in diesem Abschnitt). Im Rahmen dieses Skripts werden nur so genannte „*benannte Iteratoren*“ vorgestellt, d.h. Zugriff auf Spaltenwerte erfolgt über Spaltennamen. Zu darüber hinausgehenden Konzepten vgl. die Literatur.

- Die Fehlerbehandlung erfolgt - ebenfalls in Analogie zu JDBC - über try- & catch-Klauseln, die Exceptions der Klasse **SQLException** abfangen.

6.3.2 SQLJ - Java Embedded SQL (6)

- Jeder Iterator zum Lesen mehrzeiliger Ergebnismengen aus SELECT-Operationen muss zunächst in einer SQLJ-Klausel **deklariert** werden:

```
#sql public iterator TeilIter (String tNr, String tBez);
```

- Der SQLJ-Translator generiert auf Basis der Iterator-Deklaration eine Klasse. Vom Typ dieser Klasse muss dann in Java eine **Variable** definiert werden:

```
TeilIter iter;
```

- Dieser Variablen kann nun das Ergebnis einer SELECT-Operation zugewiesen werden. Damit wird implizit das Iteratorobjekt „geöffnet“:

```
#sql iter = { SELECT * FROM teil} ;
```

- In Analogie zum JDBC-ResultSet kann nun mit der next()-Methode über die Ergebnismenge iteriert werden. Der Zugriff auf die Spaltenwerte erfolgt direkt über die Membervariablen der Iterator-Deklaration:

```
while (iter.next()) {
  System.out.println(iter.tNr() + " " + iter.tBez());
}
iter.close();
```

6.3.2 Data Access Beans (DAB) - (1)

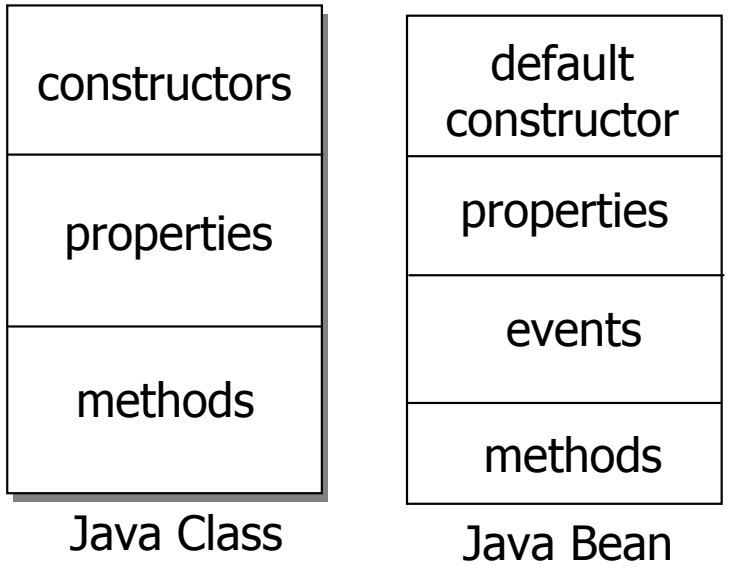
Das Konzept der **Data Access Beans (DAB)** erleichtert die Verwendung der JDBC-API und erlaubt eine objektorientierte Sicht und objektorientiertes Handling von relationlalen Daten.

Eigenschaften von Java Beans – im Allgemeinen
Eine Java Bean ist charakterisiert durch *Eigenschaften (properties)*, *Ereignisse (events)* und M*ethoden (methods),* die sie bereitstellt.

Eigenschaften beschreiben den internen Zustand einer Bean der normalerweise mit einem zugehörigen Methodenpaar von `get-` und `set-`Methoden abgefragt bzw. verändert werden kann.

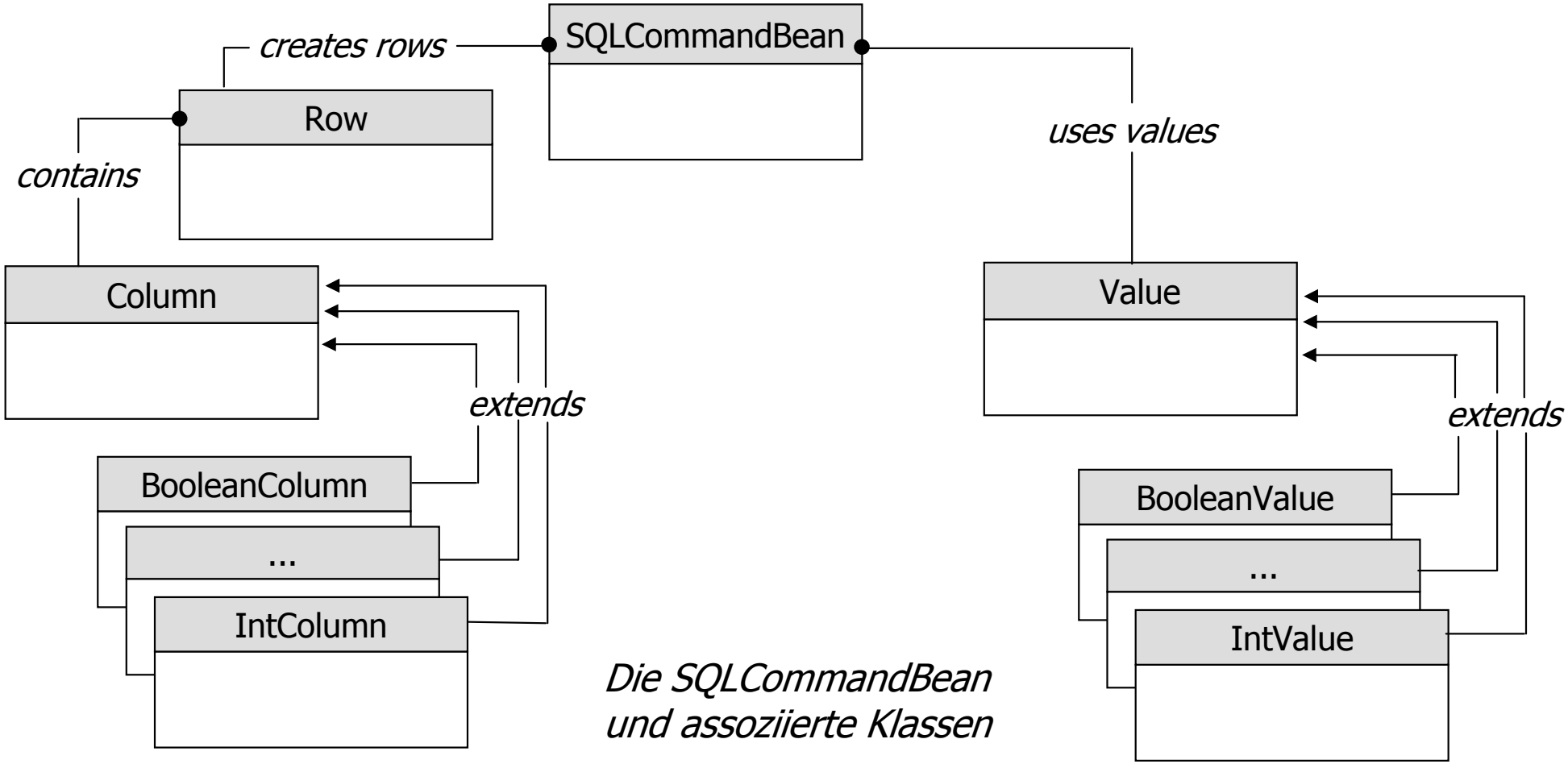
Die Kommunikation mit anderen Beans wird realisiert durch ***Ereignisse*** (entsprechend des gleichen Ereignismodells wie es auch AWT- und Swing-Komponenten benutzen).

Die ***Methoden***, die von einer Bean bereitgestellt werden, sind die öffentliche Methoden der Bean, ausgenommen die `get-` und `set-`Methoden und solche zum Registrieren und Entfernen von Event Listenern.
Jede Java Bean muss den **Default Constructor** (parameterlos) benutzen und sollte keine öffentlichen Instanzattribute haben.



6.3.2 Data Access Beans (DAB) - (2)

Verwendung einer generischen Database Bean – ein Beispiel*)



*) vgl.: Bergsten. *JavaServer Pages*. O'Reilly 2001 (Kapitel 17)

6.3.2 Data Access Beans (DAB) - (3)

Die Klasse *SQLCommandBean* und ihre Verwendung (Fortsetzung des Beispiels)

```
package com.ora.jsp.sql;

import java.util.*;
import java.sql.*;
import com.ora.jsp.sql.value.*;

public class SQLCommandBean {
   private Connection conn;
   private String sqlValue;
   private Vector values;
   private boolean isExceptionThrown = false;

public void setConnection(Connection con) {
   this.conn = conn;
}

public void setSqlValue(String sqlValue) {
   this.sqlValue = sqlValue;
}
...
```

diese property hält die aktuelle connection

query-String für ein beliebiges (Prepared) Statement object

6.3.2 Data Access Beans (DAB) - (4)

Verwendung der *S*QLCommandBean durch eine beliebige Applikation:
(Fortsetzung des Beispiels)

```
SQLCommandBean sqlBean = new SQLCommandBean();

sqlBean.setConnection(ds.getConnection());
String sqlValue = "SELECT *
                   FROM MyTable
                   WHERE IntCol = ? AND TextCol = ?";
sqlBean.setSqlValue(sqlValue);

Vector values = new Vector();
values.addElement(new IntValue(10));
values.addElement(new StringValue "Hello!"));
sqlBean.setValues(values);
```

```
public Vector executeQuery()
      throws SQLException,UnsupportedTypeException {
  Vector rows = null;
  ResultSet rs = null;
  PreparedStatement pstmt = null;
  Statemente stmt = null;
  try {
     if (values != null && values.size() > 0){
       // use a PreparedStatement and set all values
       pstmt = conn.prepareStatement(sqlValue);
       setValues(pstmt,values);
       rs = pstmt.executeQuery();
     }
     else {
       // use a Statement
       stmt = conn.createStatement();
       rs = stmt.executeQuery(sqlValue);
     }
     // store the result as vector of row-objects
     rows = toVector(rs);
  }
  finally { //... close all stream objects
  }
  return rows;
}
```

Die Methode *executeQuery()* der Klasse *SQLCommandBean*
(Fortsetzung des Beispiels)