6.3 Architekturkonzepte: objektorientierte Sprache / relationale Datenbank

- Für den Zugriff aus einer objektorientierten Programmiersprache auf eine relationale Datenbank ist eine DBMS-unabhängige Schnittstelle wünschenswert. Bei der Nutzung einer solchen Schnittstelle müssen die folgenden strukturellen Unterschiede berücksichtigt werden:

objektorientierte Konzepte	relationale Einschränkungen
• abstrakte und komplexe Datentypen: – Klassenstrukturen (Attribute und Methoden) → Kapselung – Collectiondatentypen → Implementierung mehrwertiger Referenzattribute ist möglich.	• nur elementare Datentypen (1 CNF) → keine Collectiondatentypen → keine Möglichkeit der Implementierung mehrwertiger Referenzattribute (= Foreign Keys)
• Vererbung – wird in der Implementierung unterstützt.	• Vererbung - kann in der Analyse modelliert werden, muss aber selbst implementiert werden.
• Objektidentitäten – können zur Referenzierung genutzt werden.	• Objektidentitäten - können nicht zur Referenzierung genutzt werden.

6.3 Architekturkonzepte: objektorientierte Sprache / relationale Datenbank

- *Microsoft* stellt seit 1992 unter Windows mit **ODBC (Open Database Connectivity)** eine Schnittstelle für relationale DBMS zur Verfügung.

- *Javasoft* entwickelte mit **JDBC** eine vergleichbare Schnittstelle für Java-Plattformen, die seit dem JDK 1.1 Bestandteil der Standard-API ist.

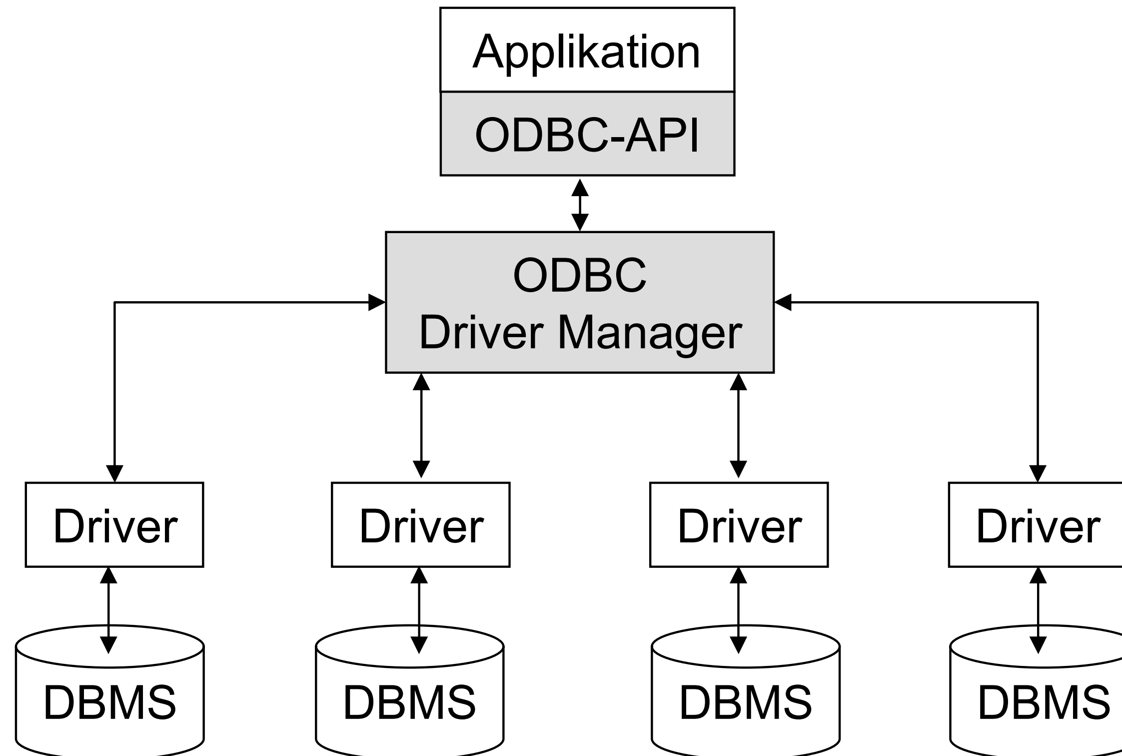
6.3.1 ODBC & Data Access Objects (DAO)

- ODBC ist eine Schnittstellendefinition, die es verschiedenen DBMS-Herstellern erlaubt, Treiber für das eigene DBMS in das standardisierte ODBC-API-Modell zu integrieren.

- Es ermöglicht einer Applikation den Zugriff mit der gleichen Syntax auf unterschiedliche relationale Datenquellen über die gleiche API.

6.3.1 ODBC

Eine Applikation richtet SQL-Anfragen über die **ODBC-API** an den **Driver Manager**, der die Anfragen entweder selbst verarbeitet oder sie an den zuständigen Treiber (Driver) weiterreicht.



- Der Treiber ist für die Kommunikation mit dem jeweiligen DBMS zuständig und muss vom Hersteller geliefert werden.

6.3.1 Joint Engine Technology (JET) und Data Access Objects (DAO) – (1)

- Eine funktionale Erweiterung des ODBC-API ist die

  Joint Engine Technology – JET.

  Hinter dieser Technologie verbirgt sich eine relationale Datenbankengine, die für alle Windows-Plattformen zur Verfügung steht.

- Eine solche Engine besteht aus einer Sammlung von DLLs, die den Zugriff auf verschiedene DBMS entweder direkt, über ISAM (= Indexed Sequentiell Access Method) oder ODBC unterstützen.

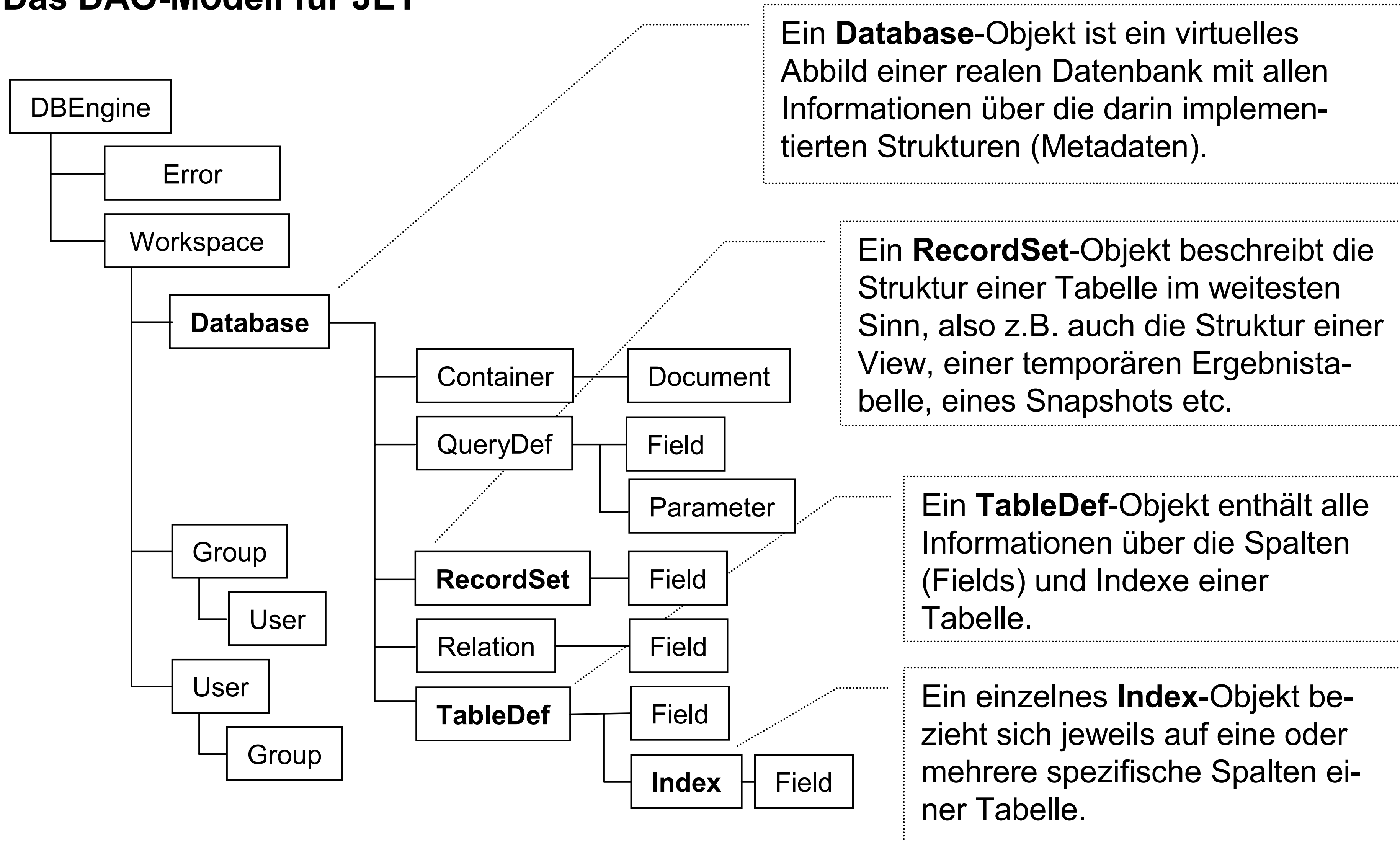
- JET kapselt die einzelnen DB-Schnittstellen in einem Objektmodell, der Zugriff auf die Engine erfolgt also über eine objektorientierte Schnittstelle.

  Die für den Zugriff erforderlichen Objekte nennt man

  Data Access Objects – DAO.

6.3.1 Joint Engine Technology (JET) und Data Access Objects (DAO) – (2)

Das DAO-Modell für JET



6.3.1 Joint Engine Technology (JET) und Data Access Objects (DAO) – (3)

Anmerkung zum DAO-Modell von Folie 6/51:

- Zu jeder Klasse, für die mehrere Objekte instanziiert werden können, also zu allen Klassen mit Ausnahme der Klasse *DBEngine*, gibt es entsprechende Collections, die den gleichen Bezeichner wie die Klasse im Plural haben.
  - *Beispiel:* Collection zur Klasse *Database*: *Databases* etc.

- Dies bedeutet im Beispiel auf der nächsten Folie, dass der Array-Bezeichner *Tabelle.Fields[...]* die Field-collection des Objektes *Tabelle* darstellt, also genau die Menge aller Spalten der Tabelle.

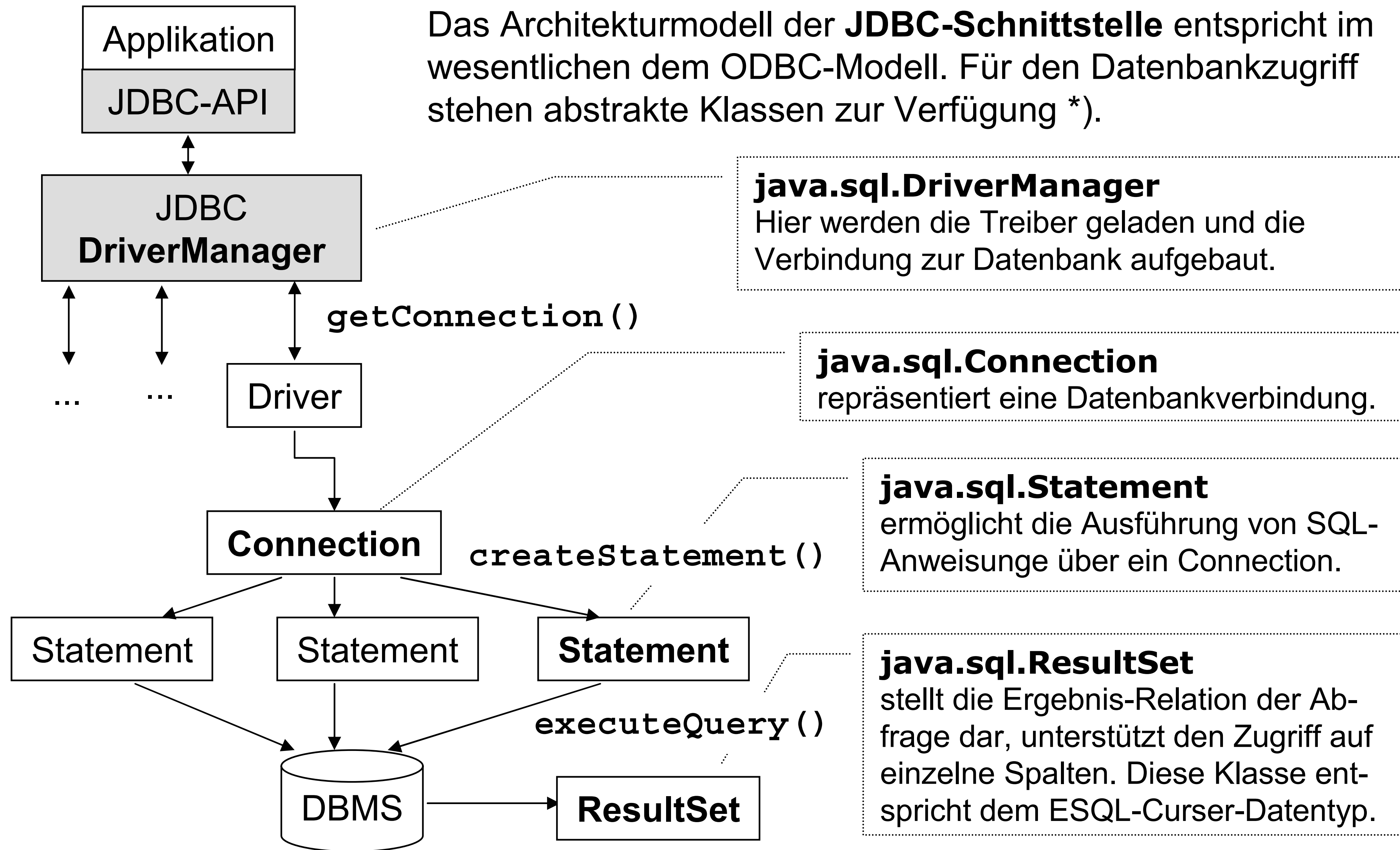
- In folgendem *Beispiel* wird ein neuer Datensatz in eine Tabelle eingefügt.

  Voraussetzung hierfür ist die Existenz eines Database-Objektes, das die aktuelle DB darstellt:

6.3.1 ODBC & DAO: ein Beispiel zur Anwendung der JET-DAOs

Vor dem Zugriff auf eine Tabelle « Name » muss die Tabelle geöffnet werden. *Methodenaufruf*: Database.OpenRecordSet(Name, ...)	Tabelle:=Database.**OpenRecordSet**(Name, dbOpenTable, dbConsistent)
Der Methodenaufruf Tabelle.AddNew instanziiert ein neues „Zeilenobjekt“ einer Tabelle.	Tabelle.**AddNew**
Die Spalten der neuen Zeile erhalten ihre Werte.	Tabelle.**Fields**['Field_1'].**Value** := Value_1 Tabelle.**Fields**['Field_2'].**Value** := Value_2 ...
Der Datensatz wird geschrieben.	Tabelle.**Update**

6.3.2 JDBC und Data Access Beans



Das Architekturmodell der **JDBC-Schnittstelle** entspricht im wesentlichen dem ODBC-Modell. Für den Datenbankzugriff stehen abstrakte Klassen zur Verfügung *).

java.sql.DriverManager
Hier werden die Treiber geladen und die Verbindung zur Datenbank aufgebaut.

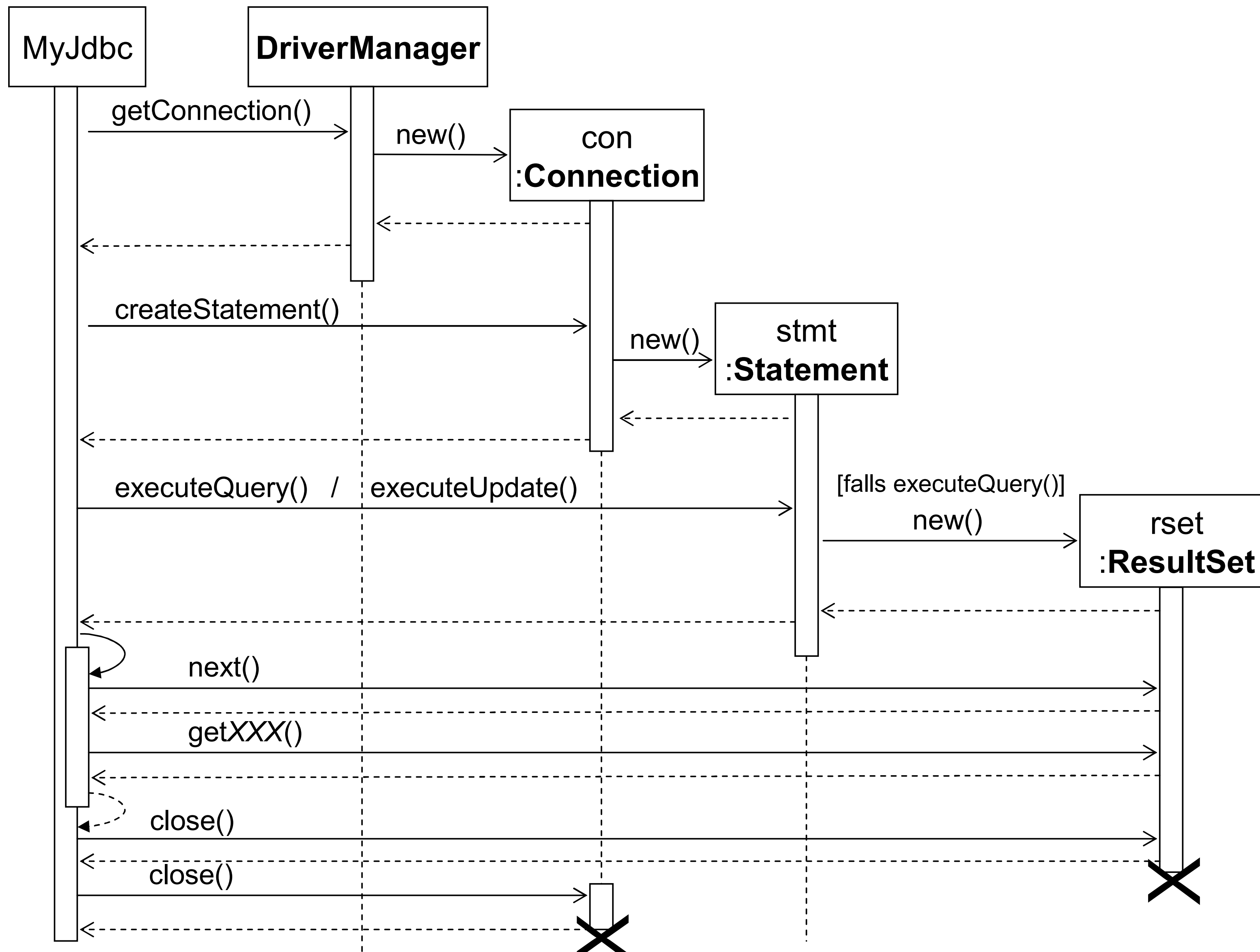
java.sql.Connection
repräsentiert eine Datenbankverbindung.

java.sql.Statement
ermöglicht die Ausführung von SQL-Anweisunge über ein Connection.

java.sql.ResultSet
stellt die Ergebnis-Relation der Abfrage dar, unterstützt den Zugriff auf einzelne Spalten. Diese Klasse entspricht dem ESQL-Curser-Datentyp.

*) Alle Klassen und Interfaces des JDBC sind im **Java-Paket java.sql** enthalten.

6.3.2 JDBC – das Zusammenspiel der Objekte im Verlauf einer Session



6.3.2 Datenbanktreiber und Datenbankquellen (1)

- Die Kernelemente von JDBC sind die Interfaces
  - **Driver** und
  - **DataSources**.

  Der Zugriff auf eine Datenquelle (Datasource) erfolgt über einen logischen Namen mit Hilfe des **Java Naming and Directory Interfaces (JNDI)**.

- Die Implementierung dieser Schnittstellen erfolgt durch den JDBC-Treiberhersteller.

- Es gibt unterschiedliche Architekturen für die Treiberimplementierung.

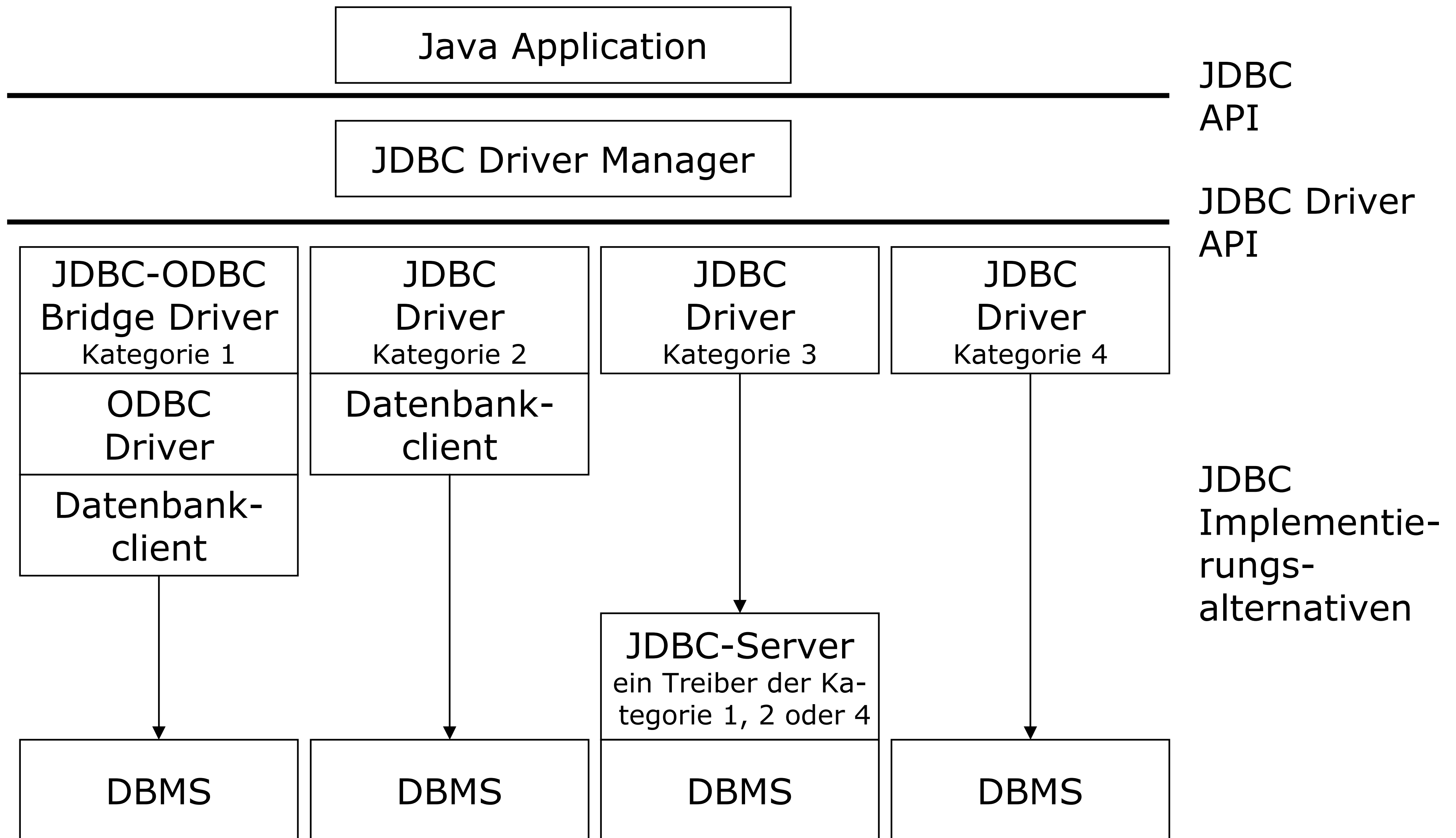
  Die JDBC-Spezifikation unterscheidet **vier Kategorien**, die auf der nächsten Folie vorgestellt werden.

6.3.2 Datenbanktreiber und Datenbankquellen (2)

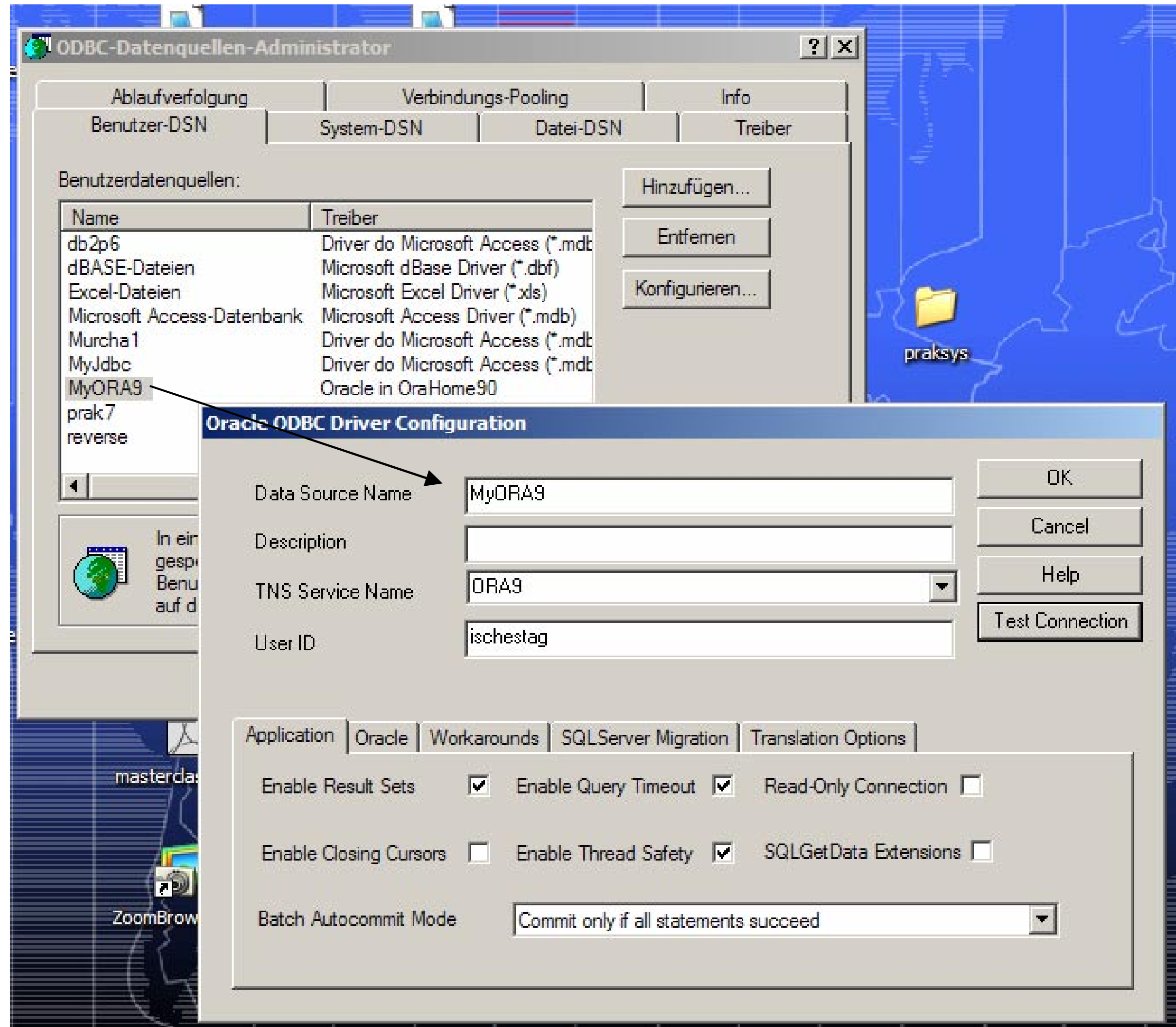
Kategorie	Beschreibung
1	Treiber dieser Kategorie bilden das JDBC API auf ein anderes Zugriffsprotokoll ab, z.B. ODBC (wie etwa die ODBC-Bridge von SUN `sun.jdbc.odbc.JdbcOdbcDriver`).
2	Treiber dieser Kategorie sind zum Teil in Java, zum Teil in anderen (nativen) Programmiersprachen implementiert und benötigen prinzipiell noch native Bibliotheken, z.B. der DB2-Treiber von IBM unter Windows.
3	Treiber der Kategorie 3 sind reine Java-Programme, die über ein datenbankunabhängiges Protokoll mit einem Middlewareserver kommunizieren.
4	Treiber der Kategorie 4 sind reine Java-Programme, die über ein datenbankspezifisches Protokoll direkt mit der Datenbank kommunizieren.

6.3.2 Datenbanktreiber und Datenbankquellen (3)

- Die vier Implementierungsmöglichkeiten für JDBC Treiber



Die Daten der **Leihbibliothek** aus dem Praktikum sind in auf dem Oracle-Server gespeichert.



Der Zugriff auf eine Datenbank im Microsoft-Umfeld erfolgt mittels einer **JDBC-ODBC-Bridge**.

Um mit Hilfe eines geeigneten Treibers auf die DB zugreifen zu können, muss im ODBC-Setup im Rahmen der „Systemsteuerung" dem Treiber die Datenquelle über einen logischen Namen bekannt gemacht werden:

Die angelegte *Oracle Datenbank* ist im System über den TNS Service Name „ORA9" bekannt und kann nun über den *Datenquellen-namen* „MyORA9" angesprochen werden.

6.3.2 JDBC - ein erstes Beispiel

```
try {
  // Verbindungsaufbau
  Connection con = DriverManager.getConnection(url, user, passwd);

  // Anfrageausführung
  Statement stmt = con.createStatement();
  ResultSet rs   = stmt.executeQuery("select leihnr, datum from ausleihe");

  while (rs.next()) {
    System.out.println(rs.getString(1) + ", " + rs.getDate(2) );
  rs.close();
  con.close();
  }
}

catch (SQLException exc){
   //Fehlerbehandlung
   System.out.println(exc);
}
```

- Zur Fehlerbehandlung stellt die Klasse **SQLException** für alle SQL- und DBMS-Fehler entsprechende Exceptions zur Verfügung.

6.3.2 JDBC - Anfrageausführung und Anfrageformen (1)

Zur **Ausführung einer Anfrage** sind drei Schritte notwendig:

1. Instanziierung eines **Statement**-Objektes zur aktuellen **Connection**:

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

2. Anweisung einer Stringvariablen zuweisen:

```
String query = " select leihnr, datum from ausleihe";
```

3. Anweisung ausführen und Ergebnis einem **ResultSet**-Objekt zuweisen:

```
ResultSet rset = stmt.executeQuery(query);
```

Im dritten Schritt wird jeweils unterschieden, ob eine retrieval- oder eine update-Operation ausgeführt wird:

- Ausführung von retrieval-Operationen (SELECT): **executeQuery(...)**
- Ausführung von update-Operationen (DELETE, UPDATE, INSERT): **executeUpdate(...)**

Diese Methode executeUpdate( ) gibt bei den genannten DML-Anweisungen als Integer die Anzahl der geschriebenen Datensätze zurück.

6.3.2 JDBC - Anfrageausführung und Anfrageformen (2)

Es sind drei Arten von Statements möglich :

1. **Statement** (Das Basis-Interface für alle Statements): Objekte dieses Typs erlauben die Ausführung einfacher Anfragen ohne Parameter

2. **PreparedStatement**: kapselt ein vorkompiliertes Statement, das parametrisierbar ist. Dieses Interface definiert ein spezielles Statement: der SQL-Abfragestring wird bereits zum DBMS geschickt, wenn das Objekt instanziiert wird, er wird vorkompiliert und ist bereit zur Ausführung. Für die endgültige Ausführung werden dann die aktuellen Parameter zum DBMS geschickt. Zuvor müssen diese aktuellen Parameter zugewiesen werden mit entsprechenden **setXXX( )**-Methoden, die die Position des "?" und den aktuellen Wert übergeben:

```
PreparedStatement pstmt =
con.prepareStatement("INSERT INTO ausleihe VALUES (?, ?, ?, ?, ?)");
```

```
pstmt.setString(1,"L029");
pstmt.setString(2,"E0111");
...
pstmt.setInteger(5,1);     ...
```

Die **executeXXX()**-Methode benötigt nun nicht mehr den Abfragestring:

```
pstmt.executeUpdate();
```

3. **CallableStatement**: StoredProcedure-Aufruf mit Parameterübergabe.

6.3.2 JDBC - Anfrageausführung und Anfrageformen (3)

Das **Navigieren in der Ergebnismenge (ResultSet)** einer Abfrage entspricht der Verwendung des **ESQL-Datentyps CURSOR**:

- Der **Zugriff auf die nächste verfügbare Zeile** wird unterstützt durch die Methode

  ```
  rset.next();
  ```

- Der **Zugriff auf die Spalte der aktuellen Zeile** eines ResultSet erfolgt durch Methoden des Typs

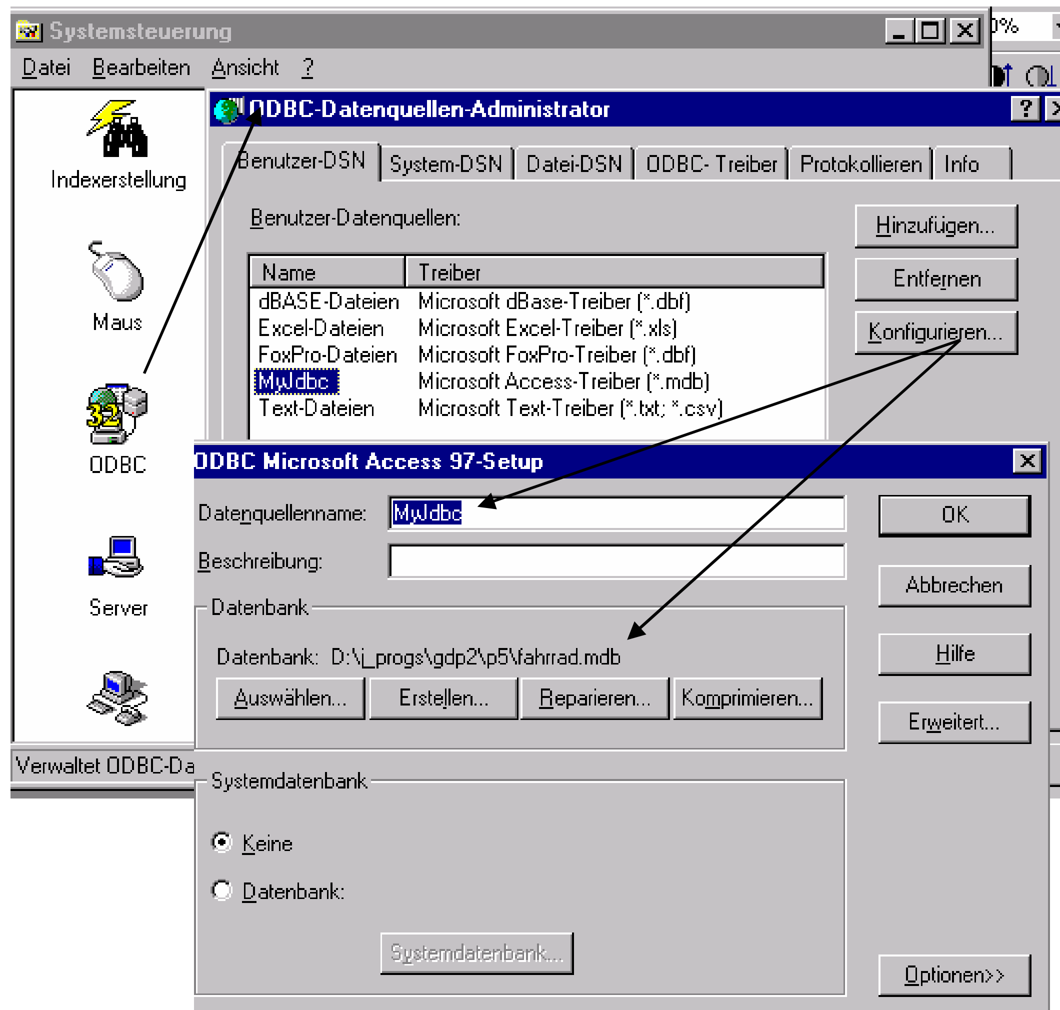
  rset.get*type*(*index*);

  die den Index der zugehörigen Ergebnisspalte (beginnend mit 1) als aktuellen Parameter erhalten. Die Methode rset.getDouble(1) gibt z.B. den Wert der ersten Spalte einer Ergebnisrelation zurück, die vom Datentyp double oder einem kompatiblen Datentyp sein muss.

Das **ESQL Cursor Konzept** ... und die entsprechende **JDBC Syntax**

ESQL	JDBC
declare **cursor** c is select ...; open cursor c;	**ResultSet** rset = stmt.executeQuery("select ... ");
fetch c into v1, v2, ...;	rset.next(); v1 = rset.get*type*(1); v2 = rset.get*type*(2); ...
close c;	rset.close();

Das folgende Beispiel erläutert die Berücksichtigung der strukturellen Unterschiede der objektorientierten und der relationalen Konzept bei der Nutzung einer ODBC- /JDBC-API (vgl. Folie 6 / 47) am Beispiel einer „Stückliste“:



Die Daten einer **Fahrrad-Stückliste** sind in zwei Tabellen einer relationalen Datenbank gespeichert:
TEIL und LISTE.

Als Datenbank-System wird **MS Access 2000** ausgewählt.

Die angelegte *MS Access-DB* heisst "fahrrad.mdb" und wird über den *Datenquellennamen* "MyJdbc" angesprochen.

1. *Schritt*: Laden der geeigneten Treiber-Klasse aus dem Paket **java.sql**:

```
import java.sql.*;

String driverClass =
      "sun.jdbc.odbc.JdbcOdbcDriver";
try {
    Class.forName(driverClass);
}
catch (ClassNotFoundException exc) {
...
}
```

2. *Schritt*: Generierung eines Connection-Objektes, das über den entsprechenden Treiber die Verbindung zur DB aufbaut:

```
try {
    Connection con =
    DriverManager.getConnection
    ("jdbc:odbc:MyJdbc");
...
```

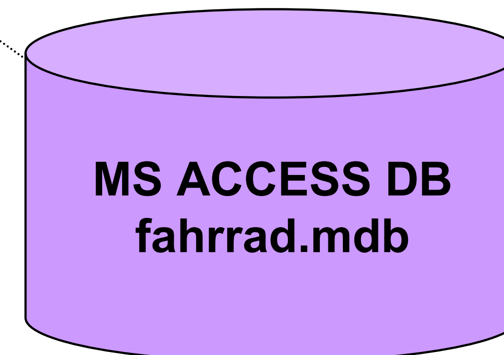
Datenquellen-Name **MyJdbc**

TEIL : Tabelle

TBEZ	TNR
Fahrrad	MB_538
Reifen	MB_539
Rahmen	MB_540
Sattel	MB_541
Antrieb	MB_542
Gabel	MB_543
Schlauch	MB_544
Felge	MB_545
Mantel	MB_546
Speiche	MB_547
Ventil	MB_548
Lager	MB_549
Polster	MB_550
Stange	MB_551
Kettenrad	MB_552
Kette	MB_553
Pedal	MB_554
Kurbel	MB_555

LISTE : Tabelle

CTNR	PTNR
MB_539	MB_538
MB_540	MB_538
MB_541	MB_538
MB_542	MB_538
MB_543	MB_538
MB_544	MB_539
MB_545	MB_539
MB_546	MB_539
MB_547	MB_539
MB_548	MB_539
MB_549	MB_539
MB_550	MB_541
MB_551	MB_541
MB_552	MB_542
MB_553	MB_542
MB_554	MB_542
MB_555	MB_542



6.3.2 Objektrelationales Mapping – Beispieldatenbank „Stückliste“ (3)

3. *Schritt*: Generierung eines **Statement-Objektes** zum Absetzen von DB-Queries über das Connection-Objekt:

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

4. *Schritt*: Generierung eines **ResultSet-Objektes**, das die Ergebnisse der Anfrage (Query) als Tabellenstruktur enthält:

```
ResultSet rs = stmt.executeQuery("Select * from TEIL");
...
```

Das ResultSet ist ein Iterator-Datentyp, der die „Recordstrukturen“ der Tabellenzeilen kennt und jede Zeile als Objekt mit den einzelnen Spaltenwerten verwaltet:

```
while (rs.next()) {
  System.out.println(rs.getString(2)+": "+rs.getString(1));
}
```

TEIL : Tabelle

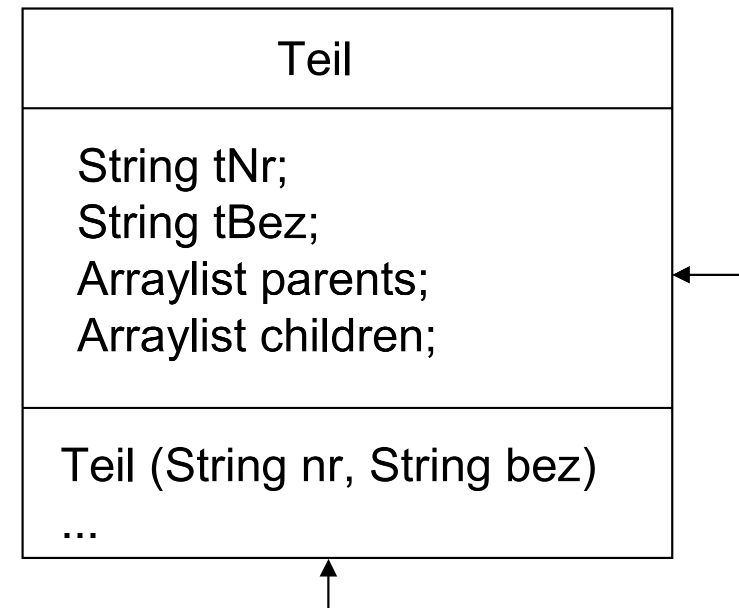
TBEZ	TNR
Fahrrad	MB_538
Reifen	MB_539
Rahmen	MB_540
Sattel	MB_541
Antrieb	MB_542
Gabel	MB_543
Schlauch	MB_544
Felge	MB_545
Mantel	MB_546
Speiche	MB_547
Ventil	MB_548
Lager	MB_549
Polster	MB_550
Stange	MB_551
Kettenrad	MB_552
Kette	MB_553
Pedal	MB_554
Kurbel	MB_555
*	

(1) **(2)**

6.3.2 Objektrelationales Mapping – Beispieldatenbank „Stückliste“ (4)

Um aus den beiden Tabellen einen Vector **sListe** mit den gewünschten Teil-Objekten zu generieren, lesen wir zunächst die Tabelle TEIL:

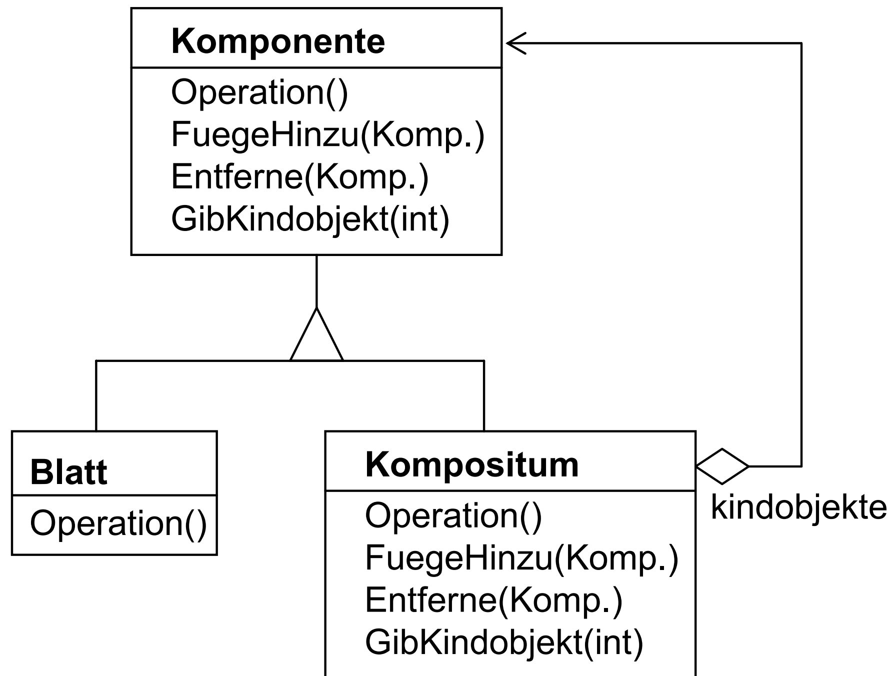
```
Vector sListe = new Vector();
...
rs = stmt.executeQuery("Select * from TEIL");
while (rs.next()) {
  s1=rs.getString(2);
  s2=rs.getString(1);
  sListe.addElement(new Teil(s1,s2));
}
rs.close();
```



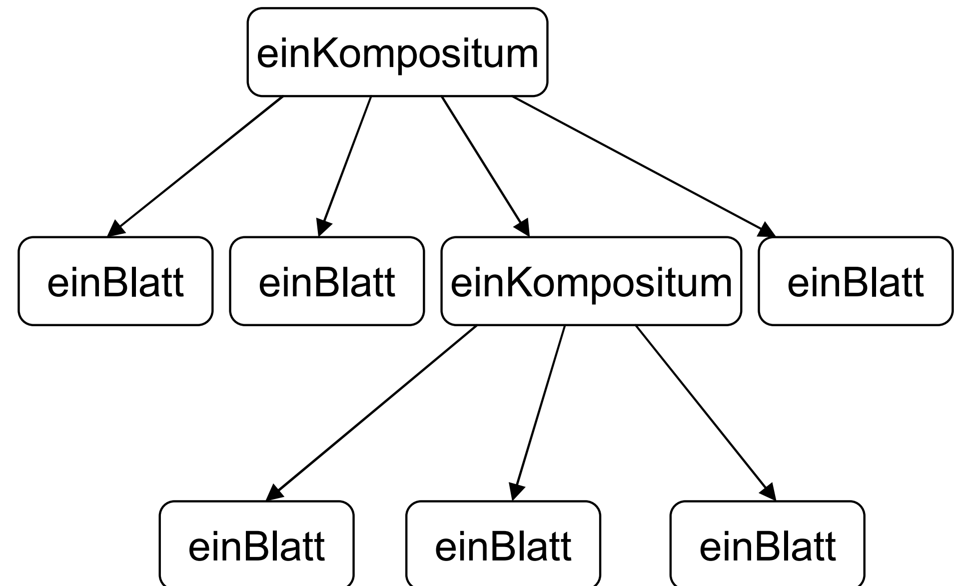
Dann wird die Tabelle LISTE gelesen, und jedem Child ein Parent-Objekt zugeordnet – und umgekehrt:

Welcher Collection-Datentyp ist geeignet,
die parent- und child-Zuordnung
zu den bestehenden Objekten zu unterstützen?

Wie müsste man die Teil-Klasse und das Einlesen der Daten erweitern, um das **Kompositium-Pattern** vollständig zu implementieren?



Klassendiagramm für ein Komponentensystem



rekursive Struktur aus Komponenten