1 Einleitung

1.1 Die Programmierschnittstelle JDBC

Java DataBase Connectivity bzw. *JDBC*™ bezeichnet eine *Programmierschnittstelle* (Application Programming Interface oder *API*), über die man in Java-Programmen Informationen aus Datenbanksystemen verarbeiten, d.h. suchen, anzeigen, erzeugen und verändern kann. Für die Formulierung entsprechender Anweisungen an das Datenbanksystem wird gewöhnlich die standardisierte Datenbanksprache SQL verwendet. JDBC setzt damit auf einer recht tiefen sprachlichen Ebene an und hat dementsprechend schlichte Fähigkeiten im Vergleich zu den meist sehr anspruchsvollen Datenbankentwicklungswerkzeugen. Deshalb wird JDBC auch als „low-level“ oder „call level“ SQL-Schnittstelle für die Java-Plattform bezeichnet.

JDBC besteht aus zwei Komponenten:

1. aus *Datenbanktreibern*, die den Anschluß von Java-Anwendungen an Datenbanksysteme wie DB/2, Oracle, MS ACCESS oder MiniSQL ermöglichen, und
2. aus dem Paket `java.sql`, bestehend aus einem *Manager* für diese Treiber, *Interfaces* als *Schnittstellen zu den Treibern* sowie Hilfsklassen für Datum, Uhrzeit und gültige JDBC-Typen.

Die *Treiber* sind datenbank- und herstellerabhängige Java-Klassen, die sich dem Anwender aber als Implementierungen von JDBC-Interfaces einheitlich präsentieren. Sie dienen dem Zweck, Java-Programmen herstellerneutrale, also von der speziellen Datenbank weitgehend unabhängige Programmierschnittstellen (APIs) anzubieten. Demgemäß sind alle JDBC-Treiber Implementierungen

der Interfaces des Pakets `java.sql` (`Connection`, `Statement`, `ResultSet` etc.). Auch die Treiber selbst sind zu Paketen geschnürt, beispielsweise der dem JDK 1.1 beiliegende ODBC-Treiber zum Paket (Package) `sun.jdbc.odbc`.

Neben den Schnittstellenfestlegungen verfügt `java.sql` noch über einen Treibermanager, mit dem Treiberobjekte verwaltet werden können. Objekte von JDBC-konformen Treiberklassen registrieren sich beim Treibermanager des Java-Programms stets selbst. Jede JDBC-Applikation hat genau einen solchen Treibermanager (auch dann, wenn der Manager umgangen und, was ohne weiteres möglich ist, direkt mit den Treibern gearbeitet wird). Der Treibermanager verwaltet lediglich die Treiberobjekte selbst, nicht deren Verbindungen zu Datenbanksystemen. Wird er verwendet, so werden über ihn zwar die Verbindungen zu den Datenbanken hergestellt, die darauf folgenden Datenbankanfragen werden aber *direkt* über die Treiber abgewickelt. (Dabei werden als Typen die JDBC-Interfaces und nicht die von Treiber zu Treiber variierenden Treiberklassen verwendet, also etwa `ResultSet` statt z.B. `OdbcJdbcResultSet` oder `MsqlResultSet`.)

Zusammenfassend ergibt sich daraus etwa eine Struktur wie in Abbildung 1-1 veranschaulicht:

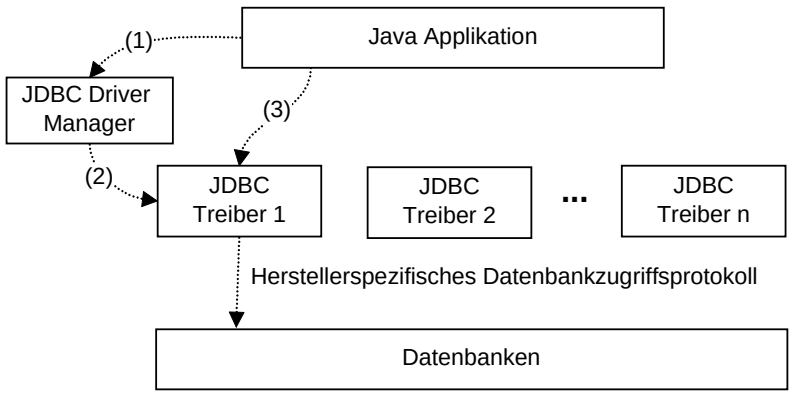

Abbildung 1-1: JDBC-Schichten

Gezeigt sind beispielhaft (1) die Registrierung eines Treiberobjektes, (2) die Verbindungsaufnahme mit einer Datenbank über den Treibermanager und (3) die Abwicklung der Datenbankmanipulationen über den Treiber direkt, d.h. über das Objekt vom Typ `Connection`, das bei der Verbindungsaufnahme erzeugt wurde.

1.2 JDBC-Konformität

Treiber können sich dann als JDBC COMPLIANT™, d.h. JDBC-konform bezeichnen, wenn sie SUNs Konformitätstest bestehen und damit u.a. *mindestens* dem sog. *ANSI/ISO*[*] *SQL/92 Entry Level* genügen. JDBC-Konformität garantiert also einen kleinsten gemeinsamen funktionellen Nenner. JDBC-konforme Treiber in einer JDBC-Anwendung sollten demnach ohne sonstige Code-Änderungen austauschbar sein, d.h. ein Wechsel von einem DBMS zu einem anderen sollte bei konformen Treibern allenfalls minimalen Aufwand verursachen.

Die Treiber selbst geben Auskunft darüber, ob sie JDBC-konform sind oder nicht. Wie diese Information in einem Java-Programm erfragt werden kann, wird im folgenden Beispiel für drei unterschiedliche Treiber gezeigt (zwei davon, der ODBC/Access- und der Oracle-Treiber, sind konform und antworten mit `true`, der dritte, für MiniSQL, ist es nicht und liefert entsprechend `false`).

Programm 1-1: JDBC-Konformität

```
// Programm 1-1:  ./Einfuehrung/JdbcKonform.java
public class JdbcKonform {
  public static void main(String[] args) throws Exception {
    System.out.println("ODBC    " + (new
        sun.jdbc.odbc.JdbcOdbcDriver()).jdbcCompliant());
    System.out.println("MiniSQL " + (new
        com.imaginary.sql.msql.MsqlDriver()).jdbcCompliant());
    System.out.println("Oracle8 " + (new
        oracle.jdbc.driver.OracleDriver()).jdbcCompliant());
  }
}                                    // Ende class JdbcKonform
```

Die Klasse `OdbcJdbcDriver` ist Bestandteil des JDK 1.1, `MsqlDriver` ist der Treiber für den SQL-Server MiniSQL der Firma Hughes Technologies, und der Treiber `OracleDriver` der Firma Oracle ist zuständig für Oracle-Datenbanken.

JDBC ist Grundlage und Rahmen sowohl für die Programmierung von Datenbankanwendungen als auch für die Entwicklung von Datenbanktreibern (zur Unterstützung von letzterem stellt SUN zusätzlich zu den APIs noch den bereits erwähnten Konformitätstest bereit). Das Schwergewicht liegt in diesem Buch auf ersterem, nämlich der Anwendungsprogrammierung mit JDBC.

---

[*] ANSI: American National Standards Institute
ISO: International Organization of Standardization

JDBC-Konformität garantiert, daß ein Treiber eine definierte Mindestleistung anbietet. Darüber hinaus kann der Treiber aber nach Belieben zusätzliche Eigenschaften haben, die der Anwender über Objekte des Typs `DatabaseMetaData` bei dem Treiber erfragen kann. So kann beipielsweise festgestellt werden, in welcher Stufung er ANSI/ISO SQL/92 unterstützt, was im folgenden Java-Programm gezeigt wird.

Als erstes wird die Klasse `sun.jdbc.odbc.JdbcOdbcDriver` geladen. Sie ist Bestandteil des JDK 1.1 und zusammen mit den Standard-Klassenbibliotheken in der Datei `classes.zip` enthalten (`rt.jar` bei Java 2). Beim Laden registriert sich der Treiber in einem Treibermanager, und mittels dieses Treibermanagers wird die Verbindung zur Datenbank `Kurse` hergestellt. Danach werden Metadaten der Datenbank erfragt.

Programm 1-2: ANSI-Stufe von MS Access

```
// Programm 1-2:  ./Einfuehrung/AnsiStufeAccess.java
import java.sql.*;
public class AnsiStufeAccess {
  public static void main(String[] args) throws Exception {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection c =
      DriverManager.getConnection("jdbc:odbc:Kurse");
    DatabaseMetaData meta = c.getMetaData();
    System.out.println(meta.getDatabaseProductName() + " "
                       + meta.getDatabaseProductVersion() );
    System.out.println("Entry Level  "
                       + meta.supportsANSI92EntryLevelSQL());
    System.out.println("Intermediate "
                       + meta.supportsANSI92IntermediateSQL());
    System.out.println("Full Level   "
                       + meta.supportsANSI92FullSQL());
  }
}                                  // Ende class AnsiStufeAccess
```

Die Antworten der dem Beispiel zugrundeliegenden MS Access-Datenbank sind

```
ACCESS 3.5 Jet
Entry Level  true
Intermediate false
Full Level   false
```

d.h. der ANSI SQL/92 Entry Level wird unterstützt, die höheren Stufen dagegen nicht.

Ein ähnliches Programm folgt für die Überprüfung des Oracle8-DBMS. An Stelle des ODBC-Treibers für die Access-Datenbank wird nun der passende

Treiber für die Oracle-Datenbank geladen (`oracle.jdbc.driver.OracleDriver`). Dazu muß der Klassenpfad `classpath` auf das Treiberpaket eingestellt sein (siehe auch Abschnitt 4.3.2). Mittels dieses Treibers wird über das Internet eine Verbindung zur Datenbank `orcl` auf dem Datenbankserver `localhost` hergestellt. `localhost` ist sozusagen das `this`-Objekt im Internet, d.h. der Server befindet sich auf dem gleichen Computer wie das Clientprogramm `AnsiStufeOracle`.

Programm 1-3: ANSI-Stufe von Oracle8 Personal Edition

```
// Programm 1-3:  ./Einfuehrung/AnsiStufeOracle.java
import java.sql.*;
public class AnsiStufeOracle {
  public static void main(String[] args) throws Exception {
    Class.forName("oracle.jdbc.driver.OracleDriver");
      Connection c = DriverManager.getConnection(
                    "jdbc:oracle:thin:@localhost:1521:orcl",
                    "Kurse","Oracle");
    DatabaseMetaData meta = c.getMetaData();
    System.out.println(meta.getDatabaseProductName() + " "
                       + meta.getDatabaseProductVersion() );
    System.out.println("Entry Level  "
                       + meta.supportsANSI92EntryLevelSQL());
  }
}                                  // Ende class AnsiStufeOracle
```

Die Anwort ist überraschenderweise, daß keine der ANSI-Stufen von SQL unterstützt wird, auch nicht die Eingangsstufe, obwohl Oracle im Programm 1-1 (`JdbcKonform`) JDBC-Konformität in Anspruch nimmt:

```
Oracle Oracle8 Personal Edition Release 8.0.4.0.0
PL/SQL Release 8.0.4.0.0
Entry Level  false
```

Anders als bei der Überprüfung der Konformität genügt es nicht, einen Treiber zu instanziieren und diesen direkt abzufragen. Denn hinter ODBC- bzw. JDBC-Treibern (vgl. Abschnitt 4.6) können sich die unterschiedlichsten Datenbanken verbergen. Konsequenterweise muß erst eine Verbindung zur konkreten Datenbank hergestellt sein, bevor dann über ein `DatabaseMetaData`-Objekt die sogenannten *Metadaten* des Datenbanksystems, so z.B. die ANSI-Stufen, festgestellt werden können.

1.3 Das JDBC-Umfeld

Das JDBC-Umfeld ist bestimmt durch *verteilte relationale* Datenbanksysteme, d.h. ein erfolgreicher Umgang mit JDBC erfordert die Kenntnis eines reichhaltigen Sortiments unterschiedlicher Methoden, die überwiegend gar nicht oder nur am Rande unmittelbar mit Java zusammenhängen. Denn JDBC baut auf relationalen Datenbanksystemen auf, spricht deren Sprache SQL (ohne sie selbst zu verstehen) und agiert in verteilten Umgebungen. Diese Situation ist im folgenden Bild symbolisiert.

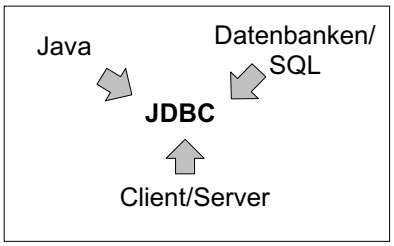

Abbildung 1-2: Das JDBC-Umfeld

Erfolgreiches Programmieren mit den JDBC-APIs setzt also Kenntnisse über ein recht breites Methodenspektum voraus:

- Relationale Datenbanken: Relationen/Tabellen, Operationen auf Tabellen und Datenmodellierung
- Recherchieren sowie Erzeugen, Löschen und Pflegen von Daten in relationalen Datenbanken: SQL
- Verteilte Anwendungen: Client/Server, Client/Server-Ebenen, Internet/ Intranet

Mittels Streifzügen durch all diese Themen wird das erforderliche Instrumentarium für erfolgreiches Programmieren mit JDBC erarbeitet. Dabei wird der Bezug zu Java und JDBC nie gänzlich außer Sichtweite geraten.

1.4 Grundstruktur von JDBC-Anwendungen

Um möglichst von Anfang an Sachverhalte mit Java/JDBC-Beispielprogrammen zu illustrieren, wird bereits hier eine knappe Einführung in JDBC gegeben. Das geschieht anhand des Programms `FuenfSchritte`, das zugleich auch Muster für die meisten Beispiele in den Folgekapiteln ist.

In jeder JDBC-Anwendung sind in der Regel die folgenden fünf Schritte erkennbar:

1. einen Treiber registrieren,
2. über den Treiber das Programm mit der Datenbank verbinden,
3. ein SQL-Anweisungsobjekt erzeugen,
4. eine Anweisung ausführen und
5. das Resultat der Anweisung verarbeiten, z.B. anzeigen.

Als weitere Schritte dürfen das Schließen offener Datenbankverbindungen und das Deregistrieren von Treibern nicht übersehen werden, auch wenn sie eine eher untergeordnete Rolle spielen.

Die Schritte 1 und 2 werden oft nur ein einziges Mal beim Programmstart ausgeführt, während die folgenden Schritte 3, 4 und 5 sich so oft wiederholen, wie es das Anwendungsprogramm erfordert. Dazu ein einfaches Beispiel (die Schritte sind durch die vorangestellten Nummern gekennzeichnet und entsprechen den aufgeführten Typen):

Programm 1-4: Phasen der JDBC-Programmierung

```
  // Programm 1-4:  ./Einfuehrung/FuenfSchritte.java
  import java.sql.*;
  public class FuenfSchritte {
    public static void main(String[] args) throws Exception {

1     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

2     Connection c =
            DriverManager.getConnection("jdbc:odbc:Kurse",
                                        "gast" ,"");

3     Statement s = c.createStatement();

4     ResultSet r = s.executeQuery(
          "SELECT * FROM Personen WHERE nachname LIKE 'K%'");

5    while(r.next()) {
       System.out.println(r.getString("vorname") + " " +
                          r.getString("nachname"));
      }

      c.close();
    }
  }                                  // Ende class FuenfSchritte
```

Die Programmschritte im einzelnen:

1. Mit `Class.forName()` wird die Klasse `JdbcOdbcDriver` aus dem Paket `sun.jdbc.odbc` *geladen, instanziiert* und die Instanz im Treibermanager *registriert*. Jedes JDBC-Programm hat genau ein statisches *Treiberregister* (vgl. Abschnitt 4.2.1), das mit dem Laden des Treibers angelegt wird.
2. Die *Verbindung* mit einer Datenbank wird mit `getConnection()` mittels des Treibermanagers über einen String hergestellt, der Ähnlichkeit mit der Syntax einer URL hat (`mailto:x@y.z`, `http://www.jugs.org` etc.) und deshalb auch JDBC-URL heißt: `jdbc:odbc:Kurse`. Die danach folgende Nutzerkennung `"gast"` und das Paßwort `""` (leerer String) dienen der Anmeldung bei dem DBMS, das die Tabellen der Datenbank `Kurse` zur Bearbeitung freigeben soll. Das Resultat ist ein Objekt vom Typ `Connection`. Danach spielt der Treibermanager keine Rolle mehr.

   Da die Verbindungsaufnahme zu einer Datenbank ein sehr zeitintensiver Vorgang ist, sollte eine Verbindung in der Regel so lange aufrechterhalten werden, bis sie definitiv nicht mehr benötigt wird. Die bestehenden Verbindungen muß der Anwender selbst unter Kontrolle halten.
3. Durch Aufruf der Methode `createStatement()` wird ein Objekt vom Typ `Statement` zur Anwendung an der Datenbank `Kurse` erzeugt.
4. Sodann wird mit Hilfe dieses `Statement`-Objektes eine `SELECT`-SQL-Anweisung auf die Datenbank angewendet. Die Ausführung der SQL-Anweisung erfolgt mittels der Methode `executeQuery()` im `Statement`-Objekt `s`. Dabei werden mit der SQL-Anweisung

   ```
   SELECT * FROM Personen WHERE nachname LIKE 'K%'
   ```

   in der Tabelle `Personen` alle Zeilen ausgewählt, in denen `nachname` mit einem `K` beginnt (`'K%'`, mit `%` als Jokerzeichen für den dem `K` evtl. noch folgenden Zeichenkettenrest). Die `executeQuery()`-Methode gibt eine entsprechende Tabelle als sog. `ResultSet` zurück. Sie hat die gleichen Spaltennamen wie die Tabelle `Personen`, in der Regel allerdings mit reduzierter Zeilenzahl. (Die Spalten*zahl* ließe sich dadurch reduzieren, daß an Stelle des Zeichens `*`, das stellvertretend für *alle* Spalten steht, z.B. die Spaltenliste `vorname`, `nachname` verwendet würde.)
5. In der `while`-Schleife werden mit der „Cursor“-Methode `next()` nacheinander die Tabellenzeilen der Ergebnistabelle (`ResultSet`) eingelesen und aus den Zeilen jeweils `vorname` und `nachname` extrahiert (`getString("`*`spaltenName`*`")`) und ausgedruckt.

   Mit dem `ResultSet`-Objekt (Ziffer 4) wird ein *Cursor* erzeugt, der nach dem ersten `next()`-Aufruf auf die erste Zeile der Ergebnistabelle weist,

sofern die Tabelle nicht leer ist. Der Cursor wird mit jedem Aufruf von `next()` um eine Zeile weiterbewegt, bis das Tabellenende erreicht ist. Es gibt in JDBC Version 1.2 keine Funktionen, um den Cursor eine Zeile zurück, drei Zeilen nach vorn oder ganz an den Anfang bzw. das Ende der Tabelle zu stellen (zu JDBC Version 2.0 siehe Abschnitt 7.1).

Vor Abschluß des Programms sollten noch die Datenbankressourcen freigegeben, d.h. die Verbindung geschlossen werden (`c.close()`). Dieser Schritt kann aber auch der virtuellen Maschine überlassen werden, die beim Beenden eines Programms automatisch die Ressourcenfreigabe bewirkt.

Zur Fehlervermeidung sollten drei Regeln frühzeitig verinnerlicht werden:

- Zu jedem `Statement`-Objekt gibt es *höchstens ein* gültiges `ResultSet`-Objekt.
- Auf ein Datenelement in einer Zeile eines `ResultSet` kann *höchstens einmal* zugegriffen werden (z.B. mit `getString("vorname")`). Außerdem ist es ratsam, eine `ResultSet`-Zeile grundsätzlich von *links nach rechts* einzulesen.
- In der Version 1.2 von JDBC sind die Zeilen einer Tabelle nur in *einer Richtung* durchschreitbar, und zwar vorwärts (`next()`).