Teil 6:

DB Programmierung: JDBC, Batches, Stored Procedures, Triggers

Ein RDBMS kann von anderen Programmen (z.B. Java) aus verwendet werden, um die Daten einer Anwendung in einer Datenbank zu verwalten. Das Anwendungsprogramm (Client) sendet SQL-Befehle zum RDBMS (Server) und erhält das Ergebnis zurück (Client-/Server-Programmiermodell). Die Programmierschnittstelle (API, application programming interface) für ein konkretes Beispiel, nämlich für Java, wird im Folgenden beschrieben. Ähnliche APIs existieren heute für praktisch alle gängigen Programmiersprachen. In der Microsoft-Welt steht zum Beispiel seit der Einführung von .NET eine Bibliothek namens ADO.NET (ActiveX Data Objects) zur Verfügung, die in Teilbereichen mit JDBC verwandt ist.

Wenn man von der Java-Welt ausgeht, handelt es sich um die Integration von Java mit Datenbanken, und man spricht deshalb von *JDBC* (Java Database Connectivity) in Anlehnung an einen älteren Standard in der Microsoft Welt, nämlich *ODBC* (Open Database Connectivity), das ein Datenbank-API für C-Programme darstellt.

JDBC stellt eine Basisbibliothek für DB-Operationen zur Verfügung. Es bietet keine Unterstützung für eine Abbildung von Tupel auf Klassen (Object/Relational Mapping). Für diese Art von Aufgaben existieren diverse Toolkits, zum Beispiel *Hibernate* (hibernate.bluemars.net) oder *JDO* (Java Data Objects, java.sun.com).

Neben der Programmierung in einer herkömmlichen Programmiersprache können die meisten RDBMS auch selbst Programme (Batch-Dateien, Stored Procedures, Triggers) verwalten, mit denen DB-Operationen programmgesteuert durchgeführt werden können (siehe weiter unten).

JDBC

Mit JDBC kann ein Java-Programm eine Verbindung zu einem RDBMS herstellen und dynamisch (also zur Laufzeit des Programmes) SQL-Befehle als Zeichenketten (String) erzeugen und abschicken. Man spricht daher auch von *dynamischem SQL*. Erst zur Laufzeit des Programms steht fest, welche SQL-Befehle ausgeführt werden und wie diese Befehle aussehen.

Eine Alternative zu dynamischem SQL ist *Embedded SQL*, bei dem SQL-Befehle direkt in den Java-Programmcode eingebettet werden. Das Programm wird mit einem speziellen Compiler übersetzt, der die embedded-SQL-Anweisungen sonderbehandelt. In diesem Fall stehen die SQL-Anweisungen schon vor der Programmausführung fest und man spricht daher auch von *statischem SQL*.

JDBC deckt nur dynamisches SQL ab, weil es flexibler ist. Allerdings hat man keine Möglichkeit, Fehler in SQL-Anweisungen schon durch den Java-Compiler zu finden. Alle SQL-bezogenen Fehler scheinen erst zur Ausführungszeit auf, was bei größeren Projekten ein beträchtliches Problem für die Qualität der Software darstellt.

Alle Klassen und Schnittstellen in Bezug auf JDBC sind im Paket *java.sql* enthalten.

Ein Online-Tutorial für JDBC befindet sich auf http://java.sun.com/docs/books/tutorial/jdbc/.

Treiber (Driver)

Damit ein Java-Programm auf ein bestimmtes RDBMS zugreifen kann, muss eine vom RDBMS-Hersteller bereitgestellte Software vorhanden sein, die den Zugriff (Netzwerkprotokoll, etc) implementiert. Die Implementierung ist inhärent RDBMS-abhängig, aber die Schnittstelle für ein benutzendes Java-Programm ist RDBMS-unabhängig. Man spricht von einem *JDBC-Treiber*. JDBC erfordert als ersten Schritt das Laden eines Treibers für ein bestimmtes RDBMS durch das Laden einer Klasse, die den Treiber implementiert. Mit folgender Anweisung wird ein Treiber geladen:

```
Class.forName("fully qualified name of driver class");
```

Man beachte, dass der Name der Treiberklasse ein String ist. Er ist nicht fest im Programm verankert und kann zum Beispiel durch einen Parameter, den man beim Programmstart angibt, festgelegt werden. Der konkrete Name der Treiberklasse für ein bestimmtes RDBMS ergibt sich aus dem Studium der Dokumentation des RDBMS oder des Treiberherstellers. Meist wird ein Archiv (.jar Datei) bereitgestellt, dass diese Klasse enthält. Dieses Archiv muss im Klassenpfad der Anwendung vorhanden sein, damit die Klasse vom Java Laufzeitsystem gefunden wird. Allenfalls muss der Klassenpfad (CLASSPATH) entsprechend erweitert werden.

Je nach Art der Implementierung eines JDBC-Treibers unterscheidet man:

1. **Typ 1**: *JDBC-ODBC-Bridge*. Der Treiber baut auf ODBC auf und erlaubt Zugriff zu allen RDBMS, für die ein ODBC-Treiber vorhanden ist. Diese Art von Treiber benötigt neben Java-Code noch native-Code zum Zugriff auf ODBC. Dadurch sind solche Treiber inhärent plattformspezifisch und nur einsetzbar, wenn ODBC vorhanden ist, also unter Microsoft Windows. Im Java SDK ist ein JdbcOdbc-Treiber enthalten, der aber für Produktionsprogramme nicht verwendet werden sollte. Er enthält diverse Einschränkungen und ist nicht garantiert stabil. Dieser Treiber kann mit `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")` geladen werden.
2. **Typ 2**: *Native Code Driver*. Der Treiber baut auf einer native-Code Bibiliothek für ein bestimmtes RDBMS auf und bietet eine JDBC-Schnittstelle an. Vom Prinzip her ähnlich wie Typ 1 aber ohne Bezug auf ODBC. Daher kann der Funktionsumfang und die Qualität besser sein, es liegt aber auch hier eine Plattformabhängigkeit vor, da nicht der gesamte Treiber in Java implementiert ist.
3. **Typ 3**: *Pure-Java Network Driver*. Hier liegt am Client eine rein Java-basierte Treiberimplementierung vor. Der Treiber kommuniziert aber nicht direkt mit dem RDBMS sondern mit einem Hilfsserver (Middleware), der seinerseits mit der RDBMS kommuniziert, wofür er einen beliebigen anderen Treiber verwenden kann. Durch die Indirektion ergibt sich hier eine geringere Effizienz. Der Vorteil ist, dass die Treiberimplementierung am Client sehr klein und plattformunabhängig ist. Das Programm wird dadurch insgesamt kleiner und kann zum Beispiel schneller über ein (langsames) Netzwerk geladen und auf jeder Java Plattform (JVM) ausgeführt werden.
4. **Typ 4**: *Pure-Java Native Protocol Treiber*. Hier liegt der Treiber vollständig in Java implementiert vor und kommuniziert direkt mit dem RDBMS über dessen eigenes Netzwerkprotokoll. Diese Treiberart ist für die meisten Anwendungen am besten geeignet. In der Zwischenzeit liegen Typ-4 Treiber für die meisten wichtigen RDBMS vor (DB2, MySql, MsSql, Oracle, Postgres, etc.).

Treibermanager

In einem Java-Programm können mehrere verschiedene RDBMS gleichzeitig verwendet werden. Daher kann es sein, dass nicht nur ein Treiber sondern mehrere geladen werden. Die Verwaltung der geladenen Treiber wird durch den Treibermanager (Klasse `DriverManager`) durchgeführt. Jeder geladene Treiber registriert sich beim Treibermanager unter Angabe eines Namens. Ein Typ 1-Treiber registriert sich zum Beispiel mit dem Namen 'odbc'. Dieser Name identifiziert den zugeordneten Treiber beim Aufbau einer Verbindung.

Verbindung (Connection)

Der Aufbau einer Verbindung zu einem bestimmten RDBMS erfolgt mittels

```
Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");
```

Der Parameter *url* bedeutet dabei eine JDBC-Pseudo-URL, also einen String, der immer mit dem Präfix 'jdbc:' (Protokoll) beginnt, gefolgt vom registrierten Treibernamen (Sub-Protokoll), gefolgt von ':' und einem treiberspezifischen Rest.

```
§ JdbcURL = "jdbc:" SubProtocol ":" Parameter.
```

Die Methode *getConnection()* ist überladen. Es existieren Varianten mit expliziter und impliziter Angabe von Benutzername und Passwort. Der treiberspezifische Rest des url-Parameters muss in der Dokumentation eines bestimmten Treibers nachgeschlagen werden.

Für den RDBMS-Zugriff mittels Typ 1-Treiber sieht der url-Paramter etwa so aus: 'jdbc:odbc:myDatabase'.

Die gesamte Befehlsfolge zum Aufbau einer Verbindung ergibt sich als:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection(
    "jdbc:odbc:myDatabase", "myLogin", "myPassword");
```

Das Ergebnis von `getConnection` ist ein Objekt vom Typ *Connection*. Dieses Objekt stellt den Ausgangspunkt für Datenbankoperationen für eine bestimmte Datenbank dar. Wird die Verbindung im Programm nicht mehr benötigt, sollte sie möglichst bald explizit geschlossen werden.

```
con.close();
```

Die Schnittstelle *Connection* stellt eine Reihe von Operationen zur Verfügung, von denen die wichtigsten im Folgenden erläutert sind. Viele der Methoden können eine SQL-bezogene Ausnahme werfen (*SqlException*).

DatabaseMetaData

Mit der Methode *getMetaData()* kann Information über den Treiber und das RDBMS abgefragt werden. Das Ergebnis ist ein Objekt vom Typ *DatabaseMetaData* und enthält Methoden zum Abfragen diverser Eigenschaften des verwendeten SQL-Dialektes, Treiber-Eigenschaften und Limitationen, Datenbank-Schemainformationen etc.

Statement

Mit der Methode *createStatement()* kann ein Objekt erzeugt werden, mit dessen Hilfe eine SQL-Anweisung an das RDBMS gesendet wird. Die SQL-Anweisungen werden unterschieden in zwei Gruppen, (1) Queries und (2) Updates. Queries führen eine SELECT-Anweisung durch und liefern das Ergebnis als *ResultSet* zurück. Updates führen alle anderen Operationen durch (CREATE, DROP, ALTER, INSERT, UPDATE, DELETE) und liefern einen Zähler zurück, der angibt, wie viele Tupel von der Operation betroffen waren.

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

ResultSet

Ein *ResultSet* repräsentiert das Ergebnis einer Abfrage, also eine Tupelmenge. Aus praktischen Gründen wird das Ergebnis aber nicht sofort in den Speicher geladen (es könnte ja sehr groß sein!), sondern es kann nur sequentiell gelesen werden. Wenn ein ResultSet nicht mehr benötigt wird, sollte es möglich bald geschlossen werden (am besten in finally-Klausel). Mit der Operation *next()* kommt man zum nächsten Tupel und kann feststellen ob man schon am Ende angelangt ist. Innerhalb eines Tupels können die Attributwerte mittels diverser *get*-Methoden unter Angabe eines Namens oder einer Nummer ausgelesen werden. Die typische Verwendung eines ResultSets ist im Folgenden gezeigt:

```
int lifnr = ...;
ResultSet rs = stmt.executeQuery(
               "SELECT * FROM Artikel WHERE lifnr = " + lifnr);
try {
  while (rs.next()) {
    int artnr = rs.getInt("artnr"); //NULL?
    String bezeichnung = rs.getString("bezeichnung");
    //allgemein: XXX x = rs.getXXX(col);
    ...
  }
}
```

```
finally {
  rs.close();
}
```

Zum Lesen der Attributwerte muss eine Abbildung der SQL-Datentypen auf Java-Datentypen erfolgen. Jdbc legt die Abbildung für alle häufig verwendeten SQL-Datentypen fest. Zum Beispiel kann der SQL-Typ *INTEGER* mittesl *getInt* auf den Java-Typ *int* abgebildet werden oder CHAR und VARCHAR mittels *getString* auf *String*. Bei Verwendung von *getInt* (und anderen numerischen Java-Basistypen) wird SQL-NULL auf den Wert 0 abgebildet da es keine Möglichkeit gibt, SQL-NULL mittels *int* auszudrücken.

Wird ein Attributwert mit der Methode *getObject()* gelesen, so erfolgt eine Umwandlung des SQL-Typs in eine Java-Klasse gemäß folgender Tabelle. SQL-NULL wird hier auf Java *null* abgebildet.

SQL Typ	*Java Klasse*
CHAR, VARCHAR, LONGVARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	Boolean
TINYINT, SMALLINT, INTEGER	Integer
BIGINT	Long
REAL	Float
FLOAT, DOUBLE	Double
DATETIME (TIMESTAMP)	java.sql.Timestamp
TIME	java.sql.Time
DATE	java.sql.Date
BINARY, VARBINARY, LONGVARBINARY	byte[]

ResultSetMetaData

Die Methode *getMetaData()* eines ResultSets liefert ein Objekt vom Typ *ResultSetMetaData*, das die Schema-Information des ResultSets enthält. Damit kann festgestellt werden, wie viele Attribute vorliegen und welchen Namen und welche SQL-Typen die Attribute besitzen. Mit diesem Wissen können Ergebnismengen beliebiger Queries verarbeitet werden und zum Beispiel in einer Tabelle angezeigt werden. Folgendes Beispiel listet alle Attribute eines ResultSets mit Namen und SQL-Typnummer auf.

```
ResultSetMetaData rsmd = rs.getMetaData();
int cnt = rsmd.getColumnCount();
for (int i = 1; i <= cnt; i++) {
  System.out.print ( "name = " + rsmd.getColumnName(i));
  System.out.println(", type = " + rsmd.getColumnType(i));
}
```

Transaktionen

Transaktionen werden bei JDBC nicht durch Ausführen der SQL-Operationen BEGIN TRANSACTION/COMMIT TRANSACTION gesteuert, sondern durch Umschalten des sogenannten Auto-Commit Modus. Eine Verbindung ist zu Beginn immer im Modus *Auto-Commit*. Das bedeutet, dass jede Operation eine eigene Transaktion darstellt, die sofort abgeschlossen wird. Möchte man mehrere Operationen zu einer Transaktion zusammenfassen, so muss der Modus auf *Manual-Commit* geändert werden. Anschließend kann eine beliebige Folge von Datenbankoperationen ausgeführt werden, die zu einer

Transaktion zusammengefasst wird. Die Transaktion wird mit *commit()* oder *rollback()* beendet. Ein Rücksetzen auf Modus *Auto-Commit* bedeutet ebenfalls ein Commit der aktuellen Transaktion.

```
con.setAutoCommit(false);
... any number of db operations
con.commit();
```

Hinweis: Eine Verbindung (Connection) kann nur eine Transaktion zu einem Zeitpunkt verwalten. Wenn in einem Java-Programm, dass zum Beispiel mehrere Threads enthält parallel mehrere Transaktionen benötigt werden, so müssen mehrere Verbindungen geöffnet werden.

Isolationsstufe

Eine Verbindung befindet sich immer in einer der 4 ANSI-Isolationsstufen (soferne das RDBMS Transaktionen überhaupt unterstützt). Mit einem Aufruf von *setTransactionIsolation()* kann zwischen Transaktionen (nicht innerhalb einer laufenden Transaktion) die Isolationsstufe auf einen der folgenden Werte gesetzt werden.

```
Connection.TRANSACTION_READ_UNCOMMITTED
Connection.TRANSACTION_READ_COMMITTED
Connection.TRANSACTION_REPEATABLE_READ
Connection.TRANSACTION_SERIALIZABLE.
(Connection.TRANSACTION_NONE)
```

Prepared Statements

Da die Übersetzung und Optimierung von SQL-Befehlen im Datenbankserver eine aufwändige Operation ist, kann man über JDBC auch mit sogenannten *Prepared Statements* arbeiten. Das sind Anweisungen, die nur einmal übersetzt werden und beliebig oft unter Angabe der benötigten Parameter ausgeführt werden können. Mit der Connection-Methode *prepareStatement()* kann ein Prepared Statement unter Angabe eines parametrisierten SQL-Befehls erzeugt werden. Die Parameter sind dabei durch '?' ausgedrückt. Folgendes Beispiel erzeugt ein PreparedStatement für die Suche nach Artikel eines bestimmten Lieferanten, dessen lifnr durch einen Parameter spezifiziert wird.

```
PreparedStatement pstmt = con.prepareStatement(
                          "SELECT * FROM Artikel WHERE lifnr = ?");
```

Die Parameterwerte müssen vor der Ausführung durch entsprechende setXXX-Operationen angegeben werden, zum Beispiel setzt folgende Anweisung den ersten Parameter als Integer-Wert.

```
pstmt.setInt(1, 77);
```

Hinweis: Die Parameternummer entspricht dem N-ten Fragezeichen in der SQL-Anweisung beginnend mit 1. Benannte Parameter sind (leider) nicht möglich.

Performance Hints

Bei manchen Treibern werden alle Tupel eines ResultSets auf einmal vom Server zum Client übertragen, was bei großen ResultSets zu OutOfMemoryError im Client führen kann. Bei anderen wiederum könnte es sein, dass alle Tupel einzeln übertragen werden, was eine beträchtliche Verlangsamung darstellt.

Die Statement-Methode *setFetchSize()* setzt für eine Anweisung die Anzahl der Zeilen, die vom Server zum Client auf einmal übertragen werden sollen. Das ist aber nur als Hinweis zu verstehen, nicht als unbedingtes Muss. In folgendem Beispiel wird empfohlen, jeweils 1000 Tupel auf einmal zu übertragen.

```
stmt.setFetchSize(1000);
```

JDBC Escapes

Um zu RDBMS-unabhängigen Programmen zu kommen, muss von den Unterschieden der existierenden SQL-Dialekte abstrahiert werden. Das ist nicht immer zu 100% möglich. In einigen Fällen ist es aber möglich, und zwar durch Verwendung der sogenannten JDBC-Escape Syntax. Statt SQL direkt zu verwenden, wird eine spezielle Notation für bestimmte SQL-Ausdrücke verwendet, die vom JDBC-Treiber in den jeweiligen SQL-Dialekt umgewandelt wird. Leider wird dieser Mechanismus nicht oder nicht vollständig von allen Treibern unterstützt. Ein Ausdruck wird dabei in geschwungene Klammern eingeschlossen und mit einem Operator versehen. Der Operator muss vom Treiber in native SQL umgewandelt werden. Nach dem Operator können noch zusätzliche operatorspezifische Parameter stehen.

§ JdbcEscapeExpr = "{" Operator {Params} "}".

Beispiel für die Definition von DATETIME-Werten unabhängig vom native-SQL.

```
{ts '2002-05-01 12:00'}
```

Die explizite Umwandlung einer Anweisung mit JDBC-Escapes in eine native SQL Anweisung erfolgt mit der Methode *nativeSQL()*.

```
String nativeSql = con.nativeSql("...{op ...} ...");
```

Cursors

JDBC unterstützt neben dem sequentiellen Auslesen eines ResultSets auch navigierbare ResultSets, sogenannte *Cursors*, sofern diese von einem RDBMS und dem JDBC-Treiber angeboten werden. In einem navigierbaren ResultSet kann die Leseposition durch entsprechende Methodenaufrufe geändert werden. Im allgemeinen Fall ist dazu aber das Anlegen einer Hilfstabelle durch den Server erforderlich, was bei großen ResultSets eine gewisse Verlangsamung der Ausführung bewirken kann. Es dauert meist länger bis man die erste Zeile des ResultSets lesen kann, weil vorher die ganze Ergebnistabelle erzeugt und gespeichert werden muss. Bei einer großen Zahl von Clients wird der Server zum Flaschenhals, weil er Hilfstabellen für alle Clients aufbauen muss und dazu natürlich entsprechend viel Speicher benötigt.

In manchen Fällen, zum Beispiel wenn nur eine Tabelle selektiert wird, kann seitens des RDBMS auf die Erzeugung einer Hilfstabelle verzichtet werden und die ursprüngliche Tabelle kann als Basis für den Cursor verwendet werden. Man spricht dann von einem *updatable ResultSet*. Die Schnittstelle *ResultSet* bietet für diesen Fall auch entsprechende *updateXXX*-Operationen.

Generell ist bei der Verwendung von Cursors Vorsicht geboten, weil sie nicht in jedem RDBMS verfügbar sind und in manchen Fällen nicht gut skalieren.

JDBC Batches

Ein JDBC *Batch* ist eine Folge von SQL-Anweisungen (ohne SELECT), die vom RDBMS gemeinsam übersetzt werden und einen gemeinsamen Ausführungsplan ergeben. Die meisten RDBMS unterstützen dieses Konzept und JDBC erlaubt die Definition und Ausführung von Batches, sofern sie vom Treiber unterstützt werden. Das Ergebnis ist in diesem Fall eine Folge von Update-Counts der ausgeführten SQL-Anweisungen. Der Vorteil von Batches besteht darin, dass weniger round-trips zwischen Client und Server erforderlich sind und die Ausführung dadurch beschleunigt wird.

Eine typische Anwendung von Batches ist das Laden einer Tabelle mit einer großen Zahl von Tupel. Anstelle für jedes Tupel ein eigenes INSERT-Statement aufzurufen, kann ein Batch mit jeweils zum Beispiel 100 INSERTs verwendet werden, was in vielen Situationen deutlich schneller ist.

Ein Batch-Statement wird durch eine Folge von *addBatch(cmd)*-Aufrufen zusammengesetzt, wobei jeweils ein SQL-Befehl als String angegeben wird.

```
stmt.addBatch("...");
stmt.addBatch("...");
int[] res = stmt.executeBatch();
```

Batch-Dateien

Die meisten RDBMS erlauben die Ausführung von Batches, also Folgen von SQL-Kommandos, die in Dateien gespeichert sind. Meist ist es möglich, nicht nur sequentielle Folgen von SQL-Anweisungen zu definieren, sondern auch Bedingungen und Schleifen auszudrücken. Viele RDBMS gehen soweit, eine vollständige Programmiersprache für Batches anzubieten, die allerdings nicht standardisiert ist. Die Bandbreite reicht von Pascal- über Basic-Derivate bis zu embedded Java.

Batch-Dateien können meist in eine Folge von Batches unterteilt sein, die durch ein bestimmtes Zeichen (zum Beispiel ';') oder ein bestimmtes Schlüsselwort (zum Beispiel "GO") voneinander getrennt sind. Jeder Batch stellt dabei eine eigene Übersetzungseinheit dar und die Batches werden in textueller Reihenfolge ausgeführt.

Die Art der Aktivierung einer Batch-Datei ist RDBMS-spezifisch. Meist gibt es die Möglichkeit der Ausführung über ein Befehlzeilenkommando einerseits und über interaktive Tools (in einem Fenster) andererseits. Die Batch-Dateien sind jedenfalls ausserhalb der Datenbank gespeichert.

Beispiel für eine Batch-Datei aus zwei Batches:

```
...statement 1 of batch 1
...statement 2 of batch 1
GO
...statement 1 of batch 2
...statement 2 of batch 2
...statement 3 of batch 2
GO
```

Stored Procedures

Stored Procedures sind Batches, die parametrisiert sein können und unter einem Namen aufgerufen werden. Eine Stored Procedure wird vom RDBMS als Ganzes kompiliert und optimiert und in einer übersetzten Form (als Execution Plan) innerhalb der Datenbank abgespeichert. Für die Ausführung müssen nur die Parameter bereitgestellt werden, eine Übersetzung findet nicht mehr statt. Stored Procedures sind daher eng verwandt mit Prepared Statements (Sonderfall mit nur einer Anweisung und ohne explizitem Namen) und viele RDBMS verwenden intern den gleichen Mechanismus für beide Konzepte.

Die Syntax für das Definieren von Stored Procedures ist RDBMS-spezifisch und es kann diverse Einschränkungen bezüglich der erlaubten SQL-Anweisungen und Parametertypen geben. Meist ist es möglich, nicht nur sequentielle Folgen von SQL-Anweisungen zu definieren, sondern auch Bedingungen und Schleifen auszudrücken.

Vereinfachte Syntax für MS Sql Server:

```
§ CreateProcStat = "CREATE" "PROCEDURE"
                   ProcName [ProcParam {"," ProcParam}]
                   "AS" {SqlStat}.

§ ProcParam = "@"ParamName Typ ["OUTPUT"].

§ ExecProcStat = "EXECUTE" ProcName Expr {"," Expr}.

§ DropProcStat = "DROP "PROCEDURE" ProcName.
```

Die Option OUTPUT bedeutet dabei einen Ausgabeparameter.

Beispiel:

```
CREATE PROCEDURE LifArtikel @lifnr INTEGER AS
  SELECT * FROM Artikel WHERE lifnr = @lifnr
```

```
EXECUTE LifArtikel 77

DROP PROCEDURE LifArtikel
```

Stored Procedures können auch via JDBC aufgerufen werden und zwar durch Erzeugung eines CallableStatament (das ist eine Unterklasse von *PreparedStatement*) durch *PrepareCall().* Ein *CallableStatement* erlaubt nicht nur das Setzen von Parametern durch die setXXX-Methoden sondern auch das Lesen von Ausgabeparametern durch entsprechende getXXX-Methoden. zusätzlich zu diesen Ausgabeparametern besteht das Ergebnis imallgemeinen Fall aus einer beliebigen Folge von Update-Counts und ResultSets.

```
CallableStatement cstmt = con.prepareCall();
boolean isResultSet = cstmt.execute("{call LifArtikel 77}");
if (isResultSet) {
      ResultSet rs = cstmt.getResultSet();
      ...
}
```

Trigger

Trigger sind *parameterlose Stored Procedures*, die bestimmten *Ereignissen* zugeordnet sind. Tritt ein solches Ereignis ein, so wird der zugeordnete Trigger automatisch vom RDBMS aufgerufen. Die Ereignisse sind das *Einfügen*, *Ändern* oder *Löschen* von Tupel einer bestimmten Tabelle. Ein typisches Anwendungsbeispiel eines Triggers ist das Protokollieren von Änderungsoperationen einer Tabelle in einer Hilfstabelle, damit später die Historie eines Datensatzes rekonstruiert werden kann.

Die Syntax für die Definition eines Triggers ist (leider) RDBMS-spezifisch. Daneben gilt es zu beachten, dass auch die Semantik des Trigger-Mechanismus variiert: Werden Trigger innerhalb der initiierenden Transaktion ausgeführt oder bilden sie eine eigene Transaktion? Können Triggeroperationen weitere Trigger auslösen? Sind rekursive Trigger-Aktivierungen erlaubt?

Durch die Verwendung von Triggers werden DB-Anwendungen inhärent RDBMS-spezifisch und erfordern zum Teil einen beträchtlichen Portierungsaufwand, wenn das RDBMS gewechselt wird. Der positive Aspekt ist, dass die Aktivierung der Triggeraktionen vom RDBMS verwaltet und damit garantiert wird und nicht von einem möglicherweise falschen oder unvollständigen Anwendungsprogramm. Die Anwendungsprogramme merken von den Triggern nichts, außer dass die Ausführungszeit entsprechend der durchgeführten Trigger-Aktivitäten erhöht wird.

Vereinfachte Syntax für MS SQL:

```
§ CreateTriggerStat = "CREATE" "TRIGGER" TriggerName "ON" TableName
                    ("FOR" | "INSTEAD" "OF")
                    ("INSERT" | "UPDATE" | "DELETE")
                    "AS" {SqlStat}.

§ DropTriggerStat = "DROP "TRIGGER" TriggerName.
```

Innerhalb der Trigger-Anweisungen stehen in MS SQL die geänderten Daten, die den Trigger ausgelöst haben, über spezielle Tabellen (*deleted*, *inserted*) zur Verfügung. Zum Beispiel kann in einem DELETE-Trigger auf die gelöschten Daten mittels 'SELECT * FROM deleted' zugegriffen werden. In Ms SQL gehören alle Trigger-Aktionen zur auslösenden Transaktion.

Beispiel:

```
CREATE TRIGGER OnArtikelDelete ON Artikel FOR DELETE AS
INSERT INTO AlteArtikel SELECT * FROM deleted
```