

SQL Services in Java

Unverhofft kommt oft. Das sollte einer Disziplin, die versucht, alles planerisch in den Griff zu bekommen, eigentlich nicht passieren, aber nicht selten erweist es sich, dass scheinbare Nebenthemen weit mehr Aufwand erfordern als vermutet. Das gilt auch für die SQL Services, die als Sammlung einiger weniger Methoden angefangen haben, mit denen die Handhabung der Java SQL-Klassen vereinfacht, sicherer und übersichtlicher gemacht werden sollte. Bei ersten Versuchen mit MySQL hatte sich nämlich rasch herausgestellt, dass die Anwendungsprogrammierung durch die unvermeidbaren Formalitäten im Umgang mit dieser SQL-Schnittstelle aufgebläht und speziell wegen der Exception-Behandlung sehr unübersichtlich wurde. Wenn der eigentlich wichtige Handlungsstrang von einer Menge zusätzlicher und zugleich unvermeidlicher Aktivitäten überlagert wird, die sich zudem in ähnlicher Form wiederholen, liegt deren Auslagerung nahe.

Einmal ins „Leben“ gerufen, begann ein rasantes Wachstum der SQL Services, das mit dem hier vorgelegten Stand keineswegs abgeschlossen ist. Das liegt daran, dass die Java SQL-Klassen für jeden Datentyp in den relationalen Tabellen je einen Placeholder Setter für die Fragezeichen-Positionen in den SQL Statements, einen Getter für das Lesen des jeweiligen Attributs und einen Updater für die Zuweisung von Attribut-Inhalten vorsehen. Die diesbezüglichen Methoden in `SQL_Util` dienen hauptsächlich als Wrapper für die Schnittstellen-Methoden in den Java SQL-Klassen. Derzeit sind sie für die Datentypen Date, Integer, String und Timestamp implementiert, was für sehr viele Zwecke völlig ausreicht. Wer möchte, kann jedoch anhand des jeweiligen Implementierungsschemas leicht weitere Methoden zu `SQL_Util` hinzufügen.

Die Klasse `SQL_Util` besteht keineswegs nur aus Verpackungen für Java SQL-Methodenaufrufe. Sie stützt sich vielmehr zusätzlich auf die Klasse `SQL_Register`, die dafür sorgt, dass vermeidbare Fehler deutlich unwahrscheinlicher werden. Das Konzept dahinter manifestiert sich im Teil B1 des Bands 3 unter der Bezeichnung „SQL Array“. Dieser Array wird von Methoden in `SQL_Register` angelegt, auf- und umgebaut. Seine Inhalte stammen einerseits aus dem jeweiligen Anwendungsprogramm, das die SQL Statements vorgibt, und andererseits aus den Rückmeldungen der Java SQL-Klassen, speziell von Handles für Prepared Statements und Result Sets sowie von Zählerwerten. Eine zentrale Rolle spielt hierbei die SQL ID jedes einzelnen Statements. Sie stammt ebenfalls aus dem jeweiligen Anwendungsprogramm und dient analog zu den File IDs der File Services zur numerischen Identifikation bei jedem Aufruf, der den SQL Array betrifft.

Jede Methode in `SQL_Util`, die sich auf den SQL Array bezieht, prüft die Gültigkeit der übergebenen SQL ID und verifiziert, ob ein benötigtes Handle vorhanden ist beziehungsweise ob der Platz für ein neues Handle noch frei ist. Das macht es deutlich unwahrscheinlicher, dass Fehler im Anwendungsprogramm sich auf die Java SQL-Schnittstelle auswirken und beispielsweise Exceptions auslösen, weil Handles undefiniert sind. Natürlich kann dieser Prüfaufwand nicht jeglichen Unsinn verhindern, aber doch die Anwendungsprogrammierung deutlich sicherer und auf jeden Fall wesentlich einfacher machen. Das zeigt die Betrachtung der diversen `SQL_Example`-Varianten in den folgenden Kapiteln auf den ersten Blick.

Last but not least bleibt hervorzuheben, dass die SQL Services keine Black Box sind, in der irgendwelche versteckten Aktionen ablaufen, von denen nur spärliche Informationen nach außen dringen. Ganz im Gegenteil: Auch in den Methoden der SQL Services sind Trace-Aufrufe implementiert, die bei Bedarf aktiviert werden, um das Verhalten jedes Aufrufs exakt nachvollziehen zu können. Somit ist auch die Hinzunahme weiterer Methoden problemlos möglich. Sie wird sich im Allgemeinen an den vorhandenen Methoden orientieren.

Die nun folgenden Kapitel beschreiben und dokumentieren die Komponenten der SQL Services. Da es sich dabei des Öfteren um relativ triviale Dinge handelt, beschränkt sich die Kommentierung auf Aspekte, die nicht auf der Hand liegen. JSP-Diagramme sind nicht erforderlich.

13.1 Hilfsklassen

13.1.1 Klasse SQL_Code

```
package sql_service;

/**
 * This enumeration defines the result codes of the calls to the SQL service
 * methods and associates them with the OK / error messages.
 */
public enum SQL_Code
{
    OK (0, "OK"),
    BEFORE_FIRST_FAILED (-200, "beforeFirst() failed"),
    BUILD_CONN_FAILED (-201, "buildConnection() failed"),
    COMMIT_FAILED (-300, "commit() failed"),
    CONN_CLOSE_FAILED (-301, "Close of connection failed"),
    CONNECTION_EXISTS (-302, "Connection exists already"),
    CONNECTION_INEXISTENT (-303, "Connection does not exist"),
    CURSOR_CLOSE_FAILED (-304, "Close of cursor failed"),
    DELETE_ROW_FAILED (-400, "deleteRow() failed"),
    EXEC_UPDATE_FAILED (-500, "Executing a prepared statement failed"),
    FETCH_CURSOR_FAILED (-600, "Fetch from cursor failed"),
    GET_DATE_FAILED (-700, "Getting a date column value failed"),
    GET_DECIMAL_FAILED (-701, "Getting a decimal column value failed"),
    GET_INT_FAILED (-702, "Getting an integer column value failed"),
    GET_STRING_FAILED (-703, "Getting a string column value failed"),
    GET_TS_FAILED (-704, "Getting a timestamp column value failed"),
    ILLEGAL_MAX_ENTRIES (-900, "Max. no. of SQL entries <= 0"),
    ILLEGAL_SQL_ID (-901, "SQL ID < 0 or > max. value"),
    INSERT_ROW_FAILED (-902, "insertRow() failed"),
    MAX_ENTRIES_DEFINED (-1300, "Max. no. of SQL entries already defined"),
    MISS_PREP_STMT_H_ENTRY (-1301, "Missing PreparedStatement handle entry"),
    MISS_PREP_STMT_H_PARM (-1302, "Missing PreparedStatement handle parameter"),
    MISS_RESULT_SET_H_ENTRY (-1303, "Missing ResultSet handle entry"),
    MISS_RESULT_SET_H_PARM (-1304, "Missing ResultSet handle parameter"),
    MISSING_SQL_STMT (-1305, "Missing SQL Statement"),
    MOVE_TO_INSERT_ROW_FAIL (-1306, "moveToInsertRow() failed"),
    NULL_CHECK_FAILED (-1400, "Checking for a NULL value failed"),
    OPEN_CURSOR_FAILED (-1500, "Open Cursor failed"),
    PREP_CLOSE_FAILED (-1600, "Close of prepared statement failed"),
    PREP_HANDLE_ALLOCATED (-1601, "Handle for PreparedStatement exists already"),
    PREP_STMT_H_MISMATCH (-1602, "Severe: Prepared Statement handle mismatch"),
    RESULT_SET_ALLOCATED (-1800, "Handle for ResultSet exists already"),
    RESULT_SET_H_MISMATCH (-1801, "Severe: ResultSet handle mismatch"),
    SET_AUTOCOMMIT_FAILED (-1900, "Setting the autocommit mode failed"),
    SET_DATE_P_H_FAILED (-1901, "Placeholder set (date) failed"),
```

```

SET_DECIMAL_P_H_FAILED ( -1902, "Placeholder set (decimal) failed"),
SET_INT_P_H_FAILED      ( -1903, "Placeholder set (int) failed"),
SET_STRING_P_H_FAILED   ( -1904, "Placeholder set (string) failed"),
SET_TS_P_H_FAILED       ( -1905, "Placeholder set (timestamp) failed"),
SQL_COMPILE_FAILED      ( -1906, "SQL compile failed"),
SQL_ENTRY_ALLOCATED     ( -1907, "SQL Entry already allocated"),
SQL_ENTRY_NOT_ALLOCATED ( -1908, "SQL Entry is not allocated"),
SQL_ID_NOT_IN_USE       ( -1909, "SQL ID not in use"),
SQL_STMT_MISSING       ( -1910, "SQL Statement missing"),
UPDATE_DATE_FAILED     ( -2100, "updateDateColumnValue() failed"),
UPDATE_DECIMAL_FAILED   ( -2101, "updateDecimalColumnValue() failed"),
UPDATE_INT_FAILED       ( -2102, "updateIntColumnValue() failed"),
UPDATE_ROW_FAILED      ( -2103, "updateRow() failed"),
UPDATE_STRING_FAILED    ( -2104, "updateStringColumnValue() failed"),
UPDATE_TO_NULL_FAILED   ( -2105, "setToNullValue() failed"),
UPDATE_TS_FAILED        ( -2106, "updateTimestampColumnValue() failed"),
INTERNAL_FATAL_ERROR    ( -9999, "Internal fatal error");

private final int    code;
private final String message;

SQL_Code(int code, String message)
{
    this.code    = code;
    this.message = message;
}

public int    getCode()    { return code; }

public String getMessage()
{
    if (this.code == OK.getCode())
        { return message; }
    else
        { return "+++ " + message; }
}
}

```

Anmerkungen

- Alle Fehlercodes sind negative Zahlen, die in Hunderterblöcke anhand der jeweiligen Anfangsbuchstaben gegliedert sind. Beispiele: B ist der zweite Buchstabe im Alphabet; folglich beginnen die B-Codes mit -200. Nach I mit -9nn folgt M mit -13nn, weil dazwischen J, K und L fehlen.
- Innerhalb eines Anfangsbuchstabens sind die verbalen Fehlercodes alphabetisch aufsteigend sortiert und die zugehörigen numerischen Codes diesen streng absteigend (also eindeutig) zugeordnet.
- Neue Fehlercodes sollten sich an diesem Schema orientieren. Wenn es zu einem Anfangsbuchstaben schon mindestens einen Fehlercode gibt, dann sollten neue Codes dort einfach angefügt werden, auch

wenn dadurch die alphabetische Sortierung nicht (mehr) gewährleistet ist. Dadurch behalten vorhandene Codes ihre numerischen Werte.

13.1.2 Klasse Trace

```
package sql_service;

/**
 * This enumeration defines the switches for the trace statements.
 */
public enum Trace
{
    DETAIL          ( false, "Display detail data"),
    METHOD           ( false, "Display method names"),
    SQL_AE_STATUS   ( false, "Display SQL array entry status"),
    SQL_AE_UPD      ( false, "Display SQL array entry update"),
    SQL_ARRAY       ( false, "Display SQL array contents");

    private final boolean traceFlag;
    private final String  purpose;

    Trace(boolean traceFlag, String purpose)
    {
        this.traceFlag = traceFlag;
        this.purpose     = purpose;
    }

    public boolean active()    { return traceFlag; }

    public String  getPurpose() { return purpose; }
}
```

13.2 Klasse SQL_Util

Die „Abbildung 13.1: Methoden in SQL_Util“ auf Seite 363 entstammt der „Abbildung 10.2: Klassendiagramm der SQL Services“. Sie enthält dieselben Bezeichner, ist aber in Gruppen von Methoden gegliedert, die jeweils einem bestimmten Zweck dienen beziehungsweise zur Menge der Mini-Methoden am Ende der Klasse gehören. Bei der Anordnung der Methoden wurde darauf geachtet, diese „von oben nach unten“ zu gliedern, also von der Connection-Ebene über die Statement-Ebene und die Cursor-Ebene schließlich zur Attribut-Ebene. Somit ist auch klar, wo weitere Methoden ihren Platz finden sollten. Um den ziemlich umfangreichen Code dieser Klasse für den Druck zu gliedern, werden die Bezeichnungen rechts von den geschweiften Klammern als Titel der Unterkapitel verwendet.

Diese statische Klasse ist nur eine Sammlung von Methoden. Sie besitzt keine Klassenvariablen. Das macht es sehr einfach, weitere Methoden hinzuzufügen, da es nur wenige interne Abhängigkeiten gibt, die allesamt demselben Schema folgen und nur der Fehlervermeidung beziehungsweise der Fehlerbehandlung dienen.

SQL_Util (static)	
- SQL_Util()	
+ buildConnection() + setAutoCommitMode() + commit() + closeConnection()	connection & SQL array creation methods
+ compileSQL_Statement()	SQL array entry creation method
+ setDatePlaceholder() + setDecimalPlaceholder() + setIntPlaceholder() + setStringPlaceholder() + setTimestampPlaceholder()	statement parameter methods
+ executeUpdate() + closePreparedStatement()	statement execution & termination methods
+ openCursor() + deleteRow() + fetchFromCursor() + insertRow() + moveToInsertRow() + updateRow() + closeCursor()	cursor based methods
+ checkForNullValue() + getDateColumnValue() + getDecimalColumnValue() + getIntColumnValue() + getStringColumnValue() + getTimestampColumnValue() + setToNullValue() + updateDateColumnValue() + updateDecimalColumnValue() + updateIntColumnValue() + updateStringColumnValue() + updateTimestampColumnValue()	attribute based methods
- checkSQL_ID() - handleStdSQL_Exception() - handleSQL_Exception()	SQL ID check & exception handling methods
+ internalError() + getConnectionHandle() + getErrorCode() + getErrorCount() + getErrorMessage() + getExecCalls() + getRowCount() + getMaxSQL_Entries() + getPrintFlag() + hasInputData() + hasReachedEOS() + setPrintFlag()	small miscellaneous methods

Abbildung 13.1: Methoden in SQL_Util

13.2.1 Einleitende Deklarationen

```
package sql_service;

import com.mysql.cj.jdbc.MysqlDataSource;
import java.math.BigDecimal;
import java.sql.Timestamp;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Date;
import static sql_service.SQL_Register.handleResults;

/**
 * The static class SQL_Util wraps access operations for MySQL.
 */

public final class SQL_Util
{
    public final static int END_REACHED = 1;
    public final static int ROW_FETCHED = 2;
    public final static int IS_NOT_NULL = 2;
    public final static int IS_NULL      = 1;

    // Constants from the class ResultSet
    public final static int CLOSE_CURSORS_AT_COMMIT
        = ResultSet.CLOSE_CURSORS_AT_COMMIT;
    public final static int CONCUR_READ_ONLY
        = ResultSet.CONCUR_READ_ONLY;
    public final static int CONCUR_UPDATABLE
        = ResultSet.CONCUR_UPDATABLE;
    public final static int FETCH_FORWARD
        = ResultSet.FETCH_FORWARD;
    public final static int FETCH_REVERSE
        = ResultSet.FETCH_REVERSE;
    public final static int FETCH_UNKNOWN
        = ResultSet.FETCH_UNKNOWN;
    public final static int HOLD_CURSORS_OVER_COMMIT
        = ResultSet.HOLD_CURSORS_OVER_COMMIT;
    public final static int TYPE_FORWARD_ONLY
        = ResultSet.TYPE_FORWARD_ONLY;
    public final static int TYPE_SCROLL_INSENSITIVE
        = ResultSet.TYPE_SCROLL_INSENSITIVE;
    public final static int TYPE_SCROLL_SENSITIVE
        = ResultSet.TYPE_SCROLL_SENSITIVE;
```

```
/**
 * dummy constructor for objects of class SQL_Util
 */
private SQL_Util() { }
```

Anmerkungen

- Die einleitende Liste von Import-Anweisungen muss gegebenenfalls bei der Hinzunahme weiterer SQL-Datentypen erweitert werden.
- Die vier Konstantendefinitionen am Anfang der Klasse verwenden willkürlich zugeordnete positive Zahlen, weil die negativen Zahlen für Fehlercodes reserviert sind.
- Die anschließenden Konstanten aus der Klasse `ResultSet` dienen dazu, diese Werte unverändert aus der Klasse `SQL_Util` an die Anwendungsebene zu exportieren. Somit müssen die aufrufenden Programme die Klasse `ResultSet` nicht selbst importieren, obwohl sie nur diese Konstanten benötigen.

13.2.2 Connection & SQL array creation methods

```
/**
 * buildConnection() builds a connection to the MySQL server, based on the
 * MysqlDataSource class. The connection handle must not exist in the register.
 * If the getConnection() request to MySQL fails, an error code is returned.
 * Otherwise the SQL array with the parameterized number of entries is
 * allocated by the register class.
 */
public static int buildConnection(String URL,
                                String user,
                                String password,
                                int SQL_Entries)
{
    Connection newConn;
    String methodName = "buildConnection()";

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_Entries = " + SQL_Entries); }

    MysqlDataSource datasource = new MysqlDataSource();
    datasource.setURL(URL);
    datasource.setUser(user);
    datasource.setPassword(password);

    if (SQL_Register.getConnectionHandle() != null)
    {
        return handleResults(SQL_Code.CONNECTION_EXISTS.getCode(),
                            SQL_Code.CONNECTION_EXISTS.getMessage());
    }
}
```

```

    }

    try
    {
        newConn = datasource.getConnection();

        SQL_Register.storeConnectionHandle(newConn);
    }
    catch (SQLException ex)
    {
        return handleSQL_Exception(-1, ex,
                                    SQL_Code.BUILD_CONN_FAILED.getCode(),
                                    SQL_Code.BUILD_CONN_FAILED.getMessage());
    }

    return SQL_Register.setupSQL_Array(SQL_Entries);
}

/**
 * setAutoCommitMode() sets the auto-commit mode.
 * * If the mode is "on", every manipulation (i.e. INSERT / UPDATE / DELETE)
 *   is made permanent in the affected table(s).
 * * If the mode is "off", the transaction mode is enabled. All manipulations
 *   are pending until they are confirmed by a COMMIT.
 */
public static int setAutoCommitMode(boolean mode)
{
    Connection conn;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: setAutocommitMode()"); }

    if (Trace.DETAIL.active())
        { System.out.println("\t mode = " + mode); }

    conn = SQL_Register.getConnectionHandle();

    if (conn == null)
    {
        return handleResults(SQL_Code.CONNECTION_INEXISTENT.getCode(),
                             SQL_Code.CONNECTION_INEXISTENT.getMessage());
    }

    try
    { conn.setAutoCommit(mode); }
    catch (SQLException ex)
    {

```

```

        return handleSQL_Exception(-1, ex,
                                   SQL_Code.SET_AUTOCOMMIT_FAILED.getCode(),
                                   SQL_Code.SET_AUTOCOMMIT_FAILED.getMessage());
    }

    return handleResults(SQL_Code.OK.getCode(),
                        SQL_Code.OK.getMessage());
}

/**
 * commit() terminates the current transaction and makes all changes permanent.
 */
public static int commit()
{
    Connection conn;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: commit()"); }

    conn = SQL_Register.getConnectionHandle();

    if (conn == null)
    {
        return handleResults(SQL_Code.CONNECTION_INEXISTENT.getCode(),
                            SQL_Code.CONNECTION_INEXISTENT.getMessage());
    }

    try
        { conn.commit(); }
    catch (SQLException ex)
    {
        return handleSQL_Exception(-1, ex,
                                   SQL_Code.COMMIT_FAILED.getCode(),
                                   SQL_Code.COMMIT_FAILED.getMessage());
    }

    return handleResults(SQL_Code.OK.getCode(),
                        SQL_Code.OK.getMessage());
}

/**
 * closeConnection() closes all open cursors and existing prepared statements
 * before it releases the connection to the MySQL server.
 * The connection handle must exist in the register.
 * If the "close" request to MySQL fails, an error code is returned.
 */
public static int closeConnection()

```

```

{
    int            SQL_Entries;
    String         methodName = "closeConnection()";
    Connection     conn;
    SQL_Entry      SQL_Entry;
    PreparedStatement prepStmt;
    ResultSet      resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    conn = SQL_Register.getConnectionHandle();

    if (conn == null)
    {
        return handleResults(SQL_Code.CONNECTION_INEXISTENT.getCode(),
                               SQL_Code.CONNECTION_INEXISTENT.getMessage());
    }

    if (Trace.SQL_ARRAY.active())
    {
        System.out.println("Display SQL array contents at " + methodName);

        SQL_Register.showSQL_Array();
    }

    // close all open cursors and existing prepared statements
    SQL_Entries = SQL_Register.getSQL_ArrayEntries();

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_Entries = " + SQL_Entries); }

    for (int i = 0; i < SQL_Entries; i++)
    {
        // garbage collection
        SQL_Entry = null;

        SQL_Entry = SQL_Register.getSQL_Entry(i);

        if (SQL_Entry != null)
        {
            if ((resultSet = SQL_Entry.getResultSet()) != null)
            {
                try
                { resultSet.close(); }
                catch (SQLException ex)
                {

```

```

        return handleStdSQL_Exception(SQL_Code.CURSOR_CLOSE_FAILED,
                                      i, methodName, ex);
    }
}

if ((prepStmt = SQL_Entry.getPreparedStmt()) != null)
{
    try
    {
        prepStmt.close();
    }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.PREP_CLOSE_FAILED,
                                      i, methodName, ex);
    }
}

// reset the SQL entry
SQL_Register.deregister(i);
}

// close the connection
try
{
    conn.close();

    SQL_Register.storeConnectionHandle(null);
}
catch (SQLException ex)
{
    return handleSQL_Exception(-1, ex,
                               SQL_Code.CONN_CLOSE_FAILED.getCode(),
                               SQL_Code.CONN_CLOSE_FAILED.getMessage());
}

SQL_Register.resetSQL_Array();

return handleResults(SQL_Code.OK.getCode(),
                    SQL_Code.OK.getMessage());
}

```

Anmerkungen

- `buildConnection()` hat sowohl die Aufgabe, die Verbindung zur MySQL Server-Instanz aufzubauen als auch den SQL Array in der parametrisierten Größe anzulegen. Erst wenn beides geklappt hat, kann das Anwendungsprogramm überhaupt mit den SQL Services arbeiten.

- Die Entscheidung für oder gegen den Transaktionsmodus ist nur bei Stapelverarbeitung möglich. `setAutoCommitMode()` ist in einer Online-Umgebung unzulässig, weil dort der Transaktionsmodus gilt.
- Dasselbe gilt für `commit()`. Ein Online-Programm darf keinen Commit absetzen, weil das eine Aufgabe des Transaktionsmanagers ist. Bei Stapelverarbeitungen entscheidet dagegen das Anwendungsprogramm, ob es überhaupt einen Commit vollzieht, und wenn ja, wie oft.

Dabei ist zu beachten, dass es bei zunehmender Menge unbestätigter Änderungen zu Lock-Eskalationen oder anderen systeminternen Behinderungen kommen kann, wodurch konkurrierende Prozesse heruntergebremsst oder sogar faktisch angehalten werden können. Gegenseitige Lock-Verklammerungen, Deadlock oder Deadly Embrace genannt, sind dagegen schon bei wenigen Zugriffen möglich, wenn konkurrierende Prozesse auf dieselben Ressourcen zugreifen und dort Sperren etabliert werden. Sie sind prinzipiell nicht ausschließbar, sondern können lediglich durch ein geeignetes Applikationsdesign entschärft werden.

Ein Commit betrifft sämtliche unbestätigten Änderungen an allen Tabellen, mit denen ein Prozess arbeitet. Daher gehört er zur Connection-Ebene. Wenn nach einem Commit ein Abbruch erfolgt, nimmt das RDBMS alle Änderungen zurück, die der betreffende Prozess seither vorgenommen hat. Stapelverarbeitungen im Transaktionsmodus, die keinen Commit vornehmen, verlieren dann folglich die gesamte bis dahin geleistete Arbeit.

- Die Methode `closeConnection()` kappt systematisch alle aktuell (noch) aktiven „Drähte“ zu MySQL, also die Verbindungen auf der Basis von Prepared Statements und Result Sets, schließt dann die Connection und „löscht“ den SQL Array.

13.2.3 SQL array entry creation method

```
/**
 * compileSQL_Statement() checks first whether the connection to MySQL exists.
 * If the connection exists, the method
 * a) checks the SQL_ID for being valid. Invalid SQL_IDs are rejected.
 * b) checks the SQL Statement parameter for a non-null String value.
 * c) checks the SQL array entry with this SQL_ID for being unused.
 * d) acquires the new entry in the SQL array for the SQL statement.
 * e) stores the SQL statement there.
 * f) sends the statement to MySQL for compiling it.
 * g) stores the returned PreparedStatement handle in the SQL entry.
 * The possible versions of the prepareStatement() invocation are covered
 * by setting the to-be-ignored parameters to -1 in the calls to this method.
 * If a parameter is wrong or another error occurs, an error code is returned.
 */
public static int compileSQL_Statement(int    SQL_ID,
                                       String SQL_Statement,
                                       int     resultSetType,
                                       int     resultSetConcurrency,
                                       int     resultSetHoldability)
{
    int         result;
    String      methodName = "compileSQL_Statement()";
    Connection  conn;
    PreparedStatement prepStmt;
```

```

if (Trace.METHOD.active())
    { System.out.println("\n SQL_Util: " + methodName); }

if (Trace.DETAIL.active())
    { System.out.println("\t SQL_ID = " + SQL_ID); }

conn = SQL_Register.getConnectionHandle();

if (conn == null)
{
    return handleResults(SQL_Code.CONNECTION_INEXISTENT.getCode(),
                        SQL_Code.CONNECTION_INEXISTENT.getMessage());
}

if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
    { return result; }

if (SQL_Statement == null)
    { return internalError(SQL_Code.MISSING_SQL_STMT,
                        SQL_ID, methodName); }

if (SQL_Register.getSQL_Statement(SQL_ID) != null)
    { return internalError(SQL_Code.SQL_ENTRY_ALLOCATED,
                        SQL_ID, methodName); }

// store the SQL statement in the array entry
result = SQL_Register.register(SQL_ID, SQL_Statement);

if (Trace.DETAIL.active())
    { System.out.println("\n result = " + result); }

if (result != SQL_Code.OK.getCode())
    { return result; }

// try to compile the statement
try
{
    if (resultSetType          >= 0 &&
        resultSetConcurrency >= 0 &&
        resultSetHoldability >= 0)
    {
        prepStmt = conn.prepareStatement(SQL_Statement,
                                         resultSetType,
                                         resultSetConcurrency,
                                         resultSetHoldability);
    }
}

```

```

        else
            if (resultSetType          >= 0 &&
                resultSetConcurrency >= 0)
            {
                prepStmt = conn.prepareStatement(SQL_Statement,
                                                  resultSetType,
                                                  resultSetConcurrency);
            }
            else
                { prepStmt = conn.prepareStatement(SQL_Statement); }
        }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.SQL_COMPILE_FAILED,
                                       SQL_ID, methodName, ex);
    }

    result = SQL_Register.register(SQL_ID, prepStmt);

    if (Trace.DETAIL.active())
        { System.out.println("\t result = " + result); }

    return result;
}

```

Anmerkungen

- `compileSQL_Statement()` prüft zunächst, ob die Connection zu MySQL geöffnet ist.
- Wenn ja, verifiziert es die übergebene SQL ID und das SQL Statement, das nicht null sein darf.
- Zudem stellt es mittels `SQL_Register` sicher, dass der zur SQL ID gehörende Array-Eintrag noch kein SQL Statement enthält.
- Ist auch das richtig, dann baut es per `register()`-Aufruf von `SQL_Register` einen Eintrag im SQL Array auf und speichert dort das SQL Statement. Dieses existiert ab da sowohl im Anwendungsprogramm als auch im SQL Array.
- Die Connection-Methode `prepareStatement()` kompiliert das SQL Statement und verleiht ihm zugleich die für die Konkurrenzverarbeitung erforderlichen Eigenschaften, falls nötig. Hierfür sind die drei letzten Parameter von `compileSQL_Statement()` da, die seitens des rufenden Programms mit denjenigen Konstanten bestückt werden können, welche am Anfang der Klasse `SQL_Util` aus `ResultSet` importiert werden. Sie betreffen folglich nur die cursor-basierte Verarbeitung von Ergebnismengen, die durch Select Statements angefordert werden.
- Für den Fall, dass das Programm einen Cursor nutzen möchte, der auf einer Basistabelle operiert, welche unter konkurrierenden Zugriffen steht, sind anwendungsseitig die beiden ersten Parameter, also `resultSetType` und `resultSetConcurrency`, mit passenden Konstantenwerten vorzugeben. Soll ein Cursor zusätzlich Commits überstehen, dann wird der Parameter `resultSetHoldability` benötigt. Die Konstanten aus `ResultSet` müssen hierfür je nach Anwendungsfall sorgfältig ausgewählt werden.
- Für andere Lesezugriffe müssen nur die beiden ersten Parameter sinnvoll bestückt werden.

- Manipulations-DML benötigt keinen der drei Parameter. Hierfür dient der letzte Aufruf von `prepareStatement()`. Dieser wird ausgewählt, wenn alle drei Parameter den Wert -1 haben.
- Im Erfolgsfall meldet Java SQL ein Prepared Statement Handle zurück. `compileSQL_Statement()` speichert es per `register()`-Aufruf von `SQL_Register` im neuen SQL Array Entry. Erst dann können Platzhalter-Variable mit den anschließend vorgestellten Methoden zugewiesen werden.
- Die Vorgaben für das Prepared Statement, die mit den drei letzten Parametern gemacht werden können, stellen hinsichtlich des Verhaltens des jeweiligen RDBMS im Konkurrenzbetrieb nur das absolute Minimum dar. Jedes RDBMS hat seine eigene Strategie, um die Datenkonsistenz zu wahren. Es ist daher unumgänglich, sich intensiv damit zu befassen, um die richtigen Entscheidungen bei der Auswahl des sogenannten Isolation Level und der lock-relevanten Klauseln für das Select Statement zu treffen. MySQL, zum Beispiel, beherrscht Lock-Methoden wie Gap Locks, die in anderen RDBMS nicht existieren.
- Die folgenden Kapitel zu den SQL_Example-Programmen gehen von Standardeinstellungen des RDBMS aus, auch dann, wenn Konkurrenzverarbeitung möglich ist. Bei MySQL gilt für InnoDB als Standard der Isolation Level `REPEATABLE READ`, der bereits erhebliche Sperrauswirkungen entfaltet. Nimmt man bei den cursor-basierten Zugriffen noch die Select-Klausel `FOR UPDATE` hinzu, dann werden alle Zeilen gesperrt, die das Programm gelesen hat (!), so wie es ein Update derselben Zeilenmenge auch tun würde. Es ist also ratsam, mit diesen Optionen sehr sorgfältig umzugehen, weil es sonst zu massiven Blockaden kommen kann.
- Tabellen, die mit einem Lock Statement gesperrt werden, benötigen keine parametrisierten Angaben zur Konkurrenzverarbeitung beim `prepareStatement()`-Aufruf.

13.2.4 Statement parameter methods

```
/**
 * setDatePlaceholder() sets a placeholder of a given SQL statement to a
 * date value. Note: A JDBC DATE is a Java Date.
 * If the SQL_ID is invalid / if the PreparedStatement handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int setDatePlaceholder(int SQL_ID,
                                     int position,
                                     Date date)
{
    int result;
    String errorMsg;
    String methodName = "setDatePlaceholder()";
    PreparedStatement prepStmt;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID    = " + SQL_ID);
        System.out.println("\t position = " + position);
        System.out.println("\t date      = " + date);
    }
}
```

```

        if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
            { return result; }

        if ((prepStmt = SQL_Register.getPreparedStatement(SQL_ID)) == null)
            { return internalError(SQL_Code.MISS_PREP_STMT_H_ENTRY,
                                   SQL_ID, methodName); }

        try
            { prepStmt.setDate(position, date); }
        catch (SQLException ex)
        {
            errorMsg = SQL_Code.SET_DATE_P_H_FAILED.getMessage() +
                " for SQL ID >" + SQL_ID + "< in " + methodName + "with" +
                " position = " + position + " & date = >" + date + "<";

            return handleSQL_Exception(SQL_ID, ex,
                                       SQL_Code.SET_DATE_P_H_FAILED.getCode(),
                                       errorMsg);
        }

        return handleResults(SQL_Code.OK.getCode(),
                              SQL_Code.OK.getMessage());
    }

    /**
     * setDecimalPlaceholder() sets a placeholder of a given SQL statement to a
     * decimal value. Note: A JDBC DECIMAL is a Java BigDecimal.
     * If the SQL_ID is invalid / if the PreparedStatement handle is missing /
     * if another error occurs, an error code is returned.
     */
    public static int setDecimalPlaceholder(int SQL_ID,
                                           int position,
                                           BigDecimal decimal)
    {
        int result;
        String errorMsg;
        String methodName = "setDecimalPlaceholder()";
        PreparedStatement prepStmt;

        if (Trace.METHOD.active())
            { System.out.println("\n SQL_Util: " + methodName); }

        if (Trace.DETAIL.active())
        {
            System.out.println("\t SQL_ID = " + SQL_ID);
            System.out.println("\t position = " + position);
        }
    }

```

```

        System.out.println("\t decimal  = " + decimal);
    }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((prepStmt = SQL_Register.getPreparedStatement(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_PREP_STMT_H_ENTRY,
                               SQL_ID, methodName); }

    try
        { prepStmt.setBigDecimal(position, decimal); }
    catch (SQLException ex)
    {
        errorMsg = SQL_Code.SET_DECIMAL_P_H_FAILED.getMessage() +
            " for SQL ID >" + SQL_ID + "< in " + methodName + "with" +
            " position = " + position + " & decimal = >" + decimal + "<";

        return handleSQL_Exception(SQL_ID, ex,
                                    SQL_Code.SET_DECIMAL_P_H_FAILED.getCode(),
                                    errorMsg);
    }

    return handleResults(SQL_Code.OK.getCode(),
                        SQL_Code.OK.getMessage());
}

/**
 * setIntPlaceholder() sets a placeholder of a given SQL statement to an
 * integer value.
 * If the SQL_ID is invalid, if the PreparedStatement handle is missing, or
 * if another error occurs, an error code is returned.
 */
public static int setIntPlaceholder(int SQL_ID,
                                     int position,
                                     int value)
{
    int          result;
    String       errorMsg;
    String       methodName = "setIntPlaceholder()";
    PreparedStatement prepStmt;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {

```

```

        System.out.println("\t SQL_ID    = " + SQL_ID);
        System.out.println("\t position = " + position);
        System.out.println("\t value    = " + value);
    }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
    { return result; }

    if ((prepStmt = SQL_Register.getPreparedStatement(SQL_ID)) == null)
    { return internalError(SQL_Code.MISS_PREP_STMT_H_ENTRY,
        SQL_ID, methodName); }

    try
    { prepStmt.setInt(position, value); }
    catch (SQLException ex)
    {
        errorMsg = SQL_Code.SET_INT_P_H_FAILED.getMessage() +
            " for SQL ID >" + SQL_ID + "< in " + methodName +
            " with position = " + position + " & value = " + value;

        return handleSQL_Exception(SQL_ID, ex,
            SQL_Code.SET_INT_P_H_FAILED.getCode(),
            errorMsg);
    }

    return handleResults(SQL_Code.OK.getCode(),
        SQL_Code.OK.getMessage());
}

/**
 * setStringPlaceholder() sets a placeholder of a given SQL statement to a
 * string value. Note: A JDBC CHAR is a Java String.
 * If the SQL_ID is invalid / if the PreparedStatement handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int setStringPlaceholder(int    SQL_ID,
                                       int    position,
                                       String string)
{
    int          result;
    String        errorMsg;
    String        methodName = "setStringPlaceholder()";
    PreparedStatement prepStmt;

    if (Trace.METHOD.active())
    { System.out.println("\n SQL_Util: " + methodName); }

```

```

if (Trace.DETAIL.active())
{
    System.out.println("\t SQL_ID    = " + SQL_ID);
    System.out.println("\t position = " + position);
    System.out.println("\t string   = " + string);
}

if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
{ return result; }

if ((prepStmt = SQL_Register.getPreparedStatement(SQL_ID)) == null)
{ return internalError(SQL_Code.MISS_PREP_STMT_H_ENTRY,
    SQL_ID, methodName); }

try
{ prepStmt.setString(position, string); }
catch (SQLException ex)
{
    errorMsg = SQL_Code.SET_STRING_P_H_FAILED.getMessage() +
        " for SQL ID >" + SQL_ID + "< in " + methodName + "with" +
        " position = " + position + " & string = >" + string + "<";

    return handleSQL_Exception(SQL_ID, ex,
        SQL_Code.SET_STRING_P_H_FAILED.getCode(),
        errorMsg);
}

return handleResults(SQL_Code.OK.getCode(),
    SQL_Code.OK.getMessage());
}

/**
 * setTimestampPlaceholder() sets a placeholder of a given SQL statement to a
 * timestamp value. Note: A JDBC DATETIME(6) is a Java Timestamp.
 * If the SQL_ID is invalid / if the PreparedStatement handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int setTimestampPlaceholder(int    SQL_ID,
                                         int      position,
                                         Timestamp timestamp)
{
    int      result;
    String    errorMsg;
    String    methodName = "setTimestampPlaceholder()";
    PreparedStatement prepStmt;

    if (Trace.METHOD.active())

```

```

        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID      = " + SQL_ID);
        System.out.println("\t position  = " + position);
        System.out.println("\t timestamp = " + timestamp);
    }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((prepStmt = SQL_Register.getPreparedStatement(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_PREP_STMT_H_ENTRY,
                               SQL_ID, methodName); }

    try
        { prepStmt.setTimestamp(position, timestamp); }
    catch (SQLException ex)
    {
        errorMsg = SQL_Code.SET_TS_P_H_FAILED.getMessage() +
            " for SQL ID >" + SQL_ID + "< in " + methodName + "with" +
            " position = " + position + " & timestamp = >" + timestamp +
"<";

        return handleSQL_Exception(SQL_ID, ex,
                                   SQL_Code.SET_TS_P_H_FAILED.getCode(),
                                   errorMsg);
    }

    return handleResults(SQL_Code.OK.getCode(),
                        SQL_Code.OK.getMessage());
}

```

Anmerkungen

- Die vier Placeholder Setter-Methoden sind identisch aufgebaut. Sie überprüfen zunächst, ob alle Voraussetzungen für den Aufruf der jeweiligen Java SQL Setter-Methode gegeben sind und nehmen diesen dann vor.
- Datentypverwechslungen und falsche Positionsnummern sind dabei die häufigsten Fehler. Daher ist es entscheidend wichtig, die Zuweisungen der Placeholder-Positionsnummern im Anwendungsprogramm extrem sorgfältig vorzunehmen. Die SQL_Example-Programme stützen sich dabei auf die Enumeration `ColumnID`. Dort werden auch die Spaltennummern definiert, die mit den Positionsnummern identisch sein können. Nur in diesen Fällen genügt dafür eine einzige Zuweisung.
- Die Existenz der Connection muss nicht erneut geprüft werden, weil dies schon in `compileSQL_Statement()` geschah.

13.2.5 Statement execution & termination methods

```

/**
 * executeUpdate() executes the prepared statement for a given SQL_ID.
 * If the SQL_ID is invalid / if the PreparedStatement handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int executeUpdate(int SQL_ID)
{
    int            result;
    int            rowCount;
    String          methodName = "executeUpdate()";
    PreparedStatement prepStmt;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID    = " + SQL_ID); }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((prepStmt = SQL_Register.getPreparedStatement(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_PREP_STMT_H_ENTRY,
                               SQL_ID, methodName); }

    // try to execute the statement
    try
    {
        rowCount = prepStmt.executeUpdate();

        SQL_Register.incExecCalls(SQL_ID);

        SQL_Register.updateRowCount(SQL_ID, rowCount);
    }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.EXEC_UPDATE_FAILED,
                                       SQL_ID, methodName, ex);
    }

    return handleResults(SQL_Code.OK.getCode(),
                        SQL_Code.OK.getMessage());
}

```

Anmerkungen

- Die Methode `executeUpdate()` wird für die Ausführung von Manipulations-DML benötigt, also meistens für Insert, Update und Delete. Sie verifiziert wie üblich die übergebene SQL ID und beschafft dann aus dem SQL Array per `getPreparedStatement()`-Aufruf von `SQL_Register` das jeweilige Prepared Statement Handle.
- Wenn anschließend die gleichnamige Methode `executeUpdate()` von Java SQL fehlerfrei ausgeführt wird, meldet sie die Anzahl der betroffenen Zeilen zurück. Andernfalls tritt eine Exception auf.
- Die Aufrufe von `incExecCalls()` und `updateRowCount()` von `SQL_Register` aktualisieren den zugehörigen Eintrag im SQL Array.

```

/**
 * closePreparedStatement() closes a prepared statement for a given SQL_ID.
 * If the affected entry is in use for a cursor based query, the result set
 * must be closed beforehand.
 * If the SQL_ID is invalid / if the PreparedStatement handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int closePreparedStatement(int SQL_ID)
{
    int          result;
    String        methodName = "closePreparedStatement()";
    PreparedStatement prepStmt;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID   = " + SQL_ID); }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((prepStmt = SQL_Register.getPreparedStatement(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_PREP_STMT_H_ENTRY,
                               SQL_ID, methodName); }

    try
        { prepStmt.close(); }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.PREP_CLOSE_FAILED,
                                       SQL_ID, methodName, ex);
    }

    result = SQL_Register.deregister(SQL_ID, prepStmt);

    return result;
}

```

Anmerkungen

- `closePreparedStatement()` deaktiviert einen Eintrag im SQL Array. Das erscheint als sinnvoll, wenn nach Abschluss einer Programmphase verhindert werden soll, dass andere Teile des Anwendungsprogramms das betroffene SQL Statement irrtümlich erneut ausführen lassen.
- Falls es sich um einen cursor-basierten Zugriff handelt, muss zuvor der betreffende Cursor explizit geschlossen werden. `closePreparedStatement()` prüft aber nicht, ob das gemacht wurde.
- Diese Methode verifiziert zunächst wie üblich ihre Situation und schließt dann das betroffene Prepared Statement. Die Methode `deregister()` von `SQL_Register` trägt das Handle aus. Das SQL Statement in diesem Eintrag bleibt erhalten. Dieser kann daher nicht mit `compileSQL_Statement()` überschrieben werden.

13.2.6 Cursor based methods

```
/**
 * openCursor() opens a cursor, based on the prepared statement of the query
 * in an SQL array entry. The handle of the result set is stored in the entry.
 * If the SQL_ID is invalid / if the PreparedStatement handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int openCursor(int SQL_ID)
{
    int                result;
    String             methodName = "openCursor()";
    PreparedStatement   prepStmt;
    ResultSet          resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID    = " + SQL_ID); }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((prepStmt = SQL_Register.getPreparedStatement(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_PREP_STMT_H_ENTRY,
                               SQL_ID, methodName); }

    // try to open a cursor for the query
    try
    {
        resultSet = prepStmt.executeQuery();

        SQL_Register.incExecCalls(SQL_ID);
    }
}
```

```

        catch (SQLException ex)
        {
            return handleStdSQL_Exception(SQL_Code.OPEN_CURSOR_FAILED,
                                           SQL_ID, methodName, ex);
        }

        result = SQL_Register.register(SQL_ID, resultSet);

        return result;
    }

```

Anmerkungen

- Das Öffnen eines Cursor ist sowohl ein formaler Akt, um ein Result Set Handle zu beschaffen als auch eine datenbanktechnische Operation, die mit einem mehr oder minder hohen Aufwand verbunden ist. MySQL baut nämlich beim Cursor Open die Ergebnistabelle ganz oder teilweise auf, wobei das Vorhandensein eines geeigneten Index, an welchem entlang die Zugriffe auf die Basisdaten erfolgen, von entscheidender Bedeutung für die Performance ist. In eher seltenen Fällen enthält schon ein Index sämtliche im Select spezifizierten Spaltenwerte. Ist beides nicht der Fall, steigen die Beschaffungsaufwände für das RDBMS erheblich. Dann müssen nämlich die angeforderten Zeilen erst anhand der Filterkriterien in der jeweiligen Where-Klausel sequentiell gelesen und die in diesen Zeilen enthaltenen Spaltenwerte gemäß der Vorgabe durch den Select extrahiert werden. Meist muss die vorläufige Ergebnismenge zudem sortiert werden, bevor dem Anwendungsprogramm die allererste Zeile angeliefert werden kann. Dieser Vorgang heißt Materialisierung. Er verursacht einen erheblichen Aufwand je nach Tabellengröße und Filterfaktoren.
- Auf einer materialisierten Ergebnismenge kann keine Konkurrenzverarbeitung stattfinden. Diese ist nur dann möglich, wenn die Zeilen unmittelbar aus der jeweiligen Basistabelle gelesen werden. Dabei errichtet das RDBMS Sperren, welche die Datenkonsistenz garantieren, solange der Lesezugriff auf eine bestimmte Zeile andauert oder diese Zeile geändert beziehungsweise gelöscht wird. Sobald ein Cursor, der für Konkurrenzverarbeitung präpariert wurde, zur nächsten Zeile weiterschaltet, wandert auch die Lesesperre dorthin. Update- und Delete-Sperren bleiben dagegen bis zum nächsten Commit erhalten. Konkurrierende Prozesse können auf gesperrte Zeilen entweder überhaupt nicht zugreifen oder sie müssen mit Phantom-Ergebnissen zurechtkommen, falls das jeweilige RDBMS Uncommitted Reads erlaubt.
- Da MySQL im Gegensatz zu DB2 keine Singleton Selects kennt, muss jeder Lesezugriff, auch der auf eine einzelne Zeile, per Cursor erfolgen. Weil in der Praxis immer wieder Fälle auftreten, bei denen ein angeblich eindeutiger Zugriff mehrere Zeilen adressiert, obliegt es der sorgfältigen Implementierung, solche Fehlerfälle auszuschließen. Wenn man sich nicht völlig sicher ist, empfiehlt sich ein weiterer Lesezugriff, der keinesfalls eine weitere Zeile bereitstellen darf. DB2 meldet bei einem Singleton Select, der mehr als eine Zeile abliefert, einen Fehlercode zurück. In beiden Fällen liegt aber keine Garantie vor, dass es in irgendeinem exotischen Fall doch noch zu mehreren Ergebniszeilen kommen könnte. Vielmehr kann das nur durch ein korrektes Datenbankdesign in Verbindung mit der Nutzung eindeutiger Indizes gesichert werden. Ein Programm, das in solchen Fällen die Existenz mehrerer Ergebniszeilen ignorierte, verursachte einen Schaden von mehreren hunderttausend Euro!
- `openCursor()` prüft eingangs die SQL ID und das Vorhandensein eines zugehörigen Prepared Statement, bevor versucht wird, den jeweiligen Cursor zu öffnen. Gelingt das, meldet `executeQuery()` ein Result Set Handle zurück. `incExecCalls()` in `SQL_Register` zählt den erfolgreichen Aufruf. Mit `register()` in `SQL_Register` speichert `SQL_Util` das Result Set Handle im SQL Array an der SQL ID-Position.

```

/**
 * deleteRow() deletes the row at the current cursor position.
 */
public static int deleteRow(int SQL_ID)
{
    int result;
    String methodName = "deleteRow()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID    = " + SQL_ID); }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                               SQL_ID, methodName); }

    // delete the row
    try
        { resultSet.deleteRow(); }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.DELETE_ROW_FAILED,
                                       SQL_ID, methodName, ex);
    }

    return handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());
}

```

Anmerkungen

- deleteRow() ist die erste Methode in SQL_Util, die auf einem Cursor basiert. Sie stellt ein Modell für die weiteren cursor-basierten Methoden dar, denn die darin eingangs codierten Prüfungen wiederholen sich immer wieder. Es wurde versucht, diesen repetitiven Code auszulagern, was aber nur teilweise gelang.
- deleteRow() setzt voraus, dass mit fetchFromCursor() eine Position in der Ergebnismenge etabliert wurde. Diejenige Zeile, auf welche der Cursor aktuell zeigt, wird gelöscht. Die eigentliche Löschung findet allerdings erst beim Commit statt, denn es ist ja möglich, dass dieser nicht erreicht wird und MySQL dann alle unbestätigten Manipulationen zurückrollt.
- Diese Methode offeriert eine Alternative dazu, eine Zeile per Delete Statement mit Vorgabe einer eindeutigen Zeilenidentifikation zu löschen. Ein solches Statement muss ja nicht nur kompiliert werden, sondern es erfordert bei seiner Ausführung, die betreffende Zeile exakt zu lokalisieren. Diese Aufwände entfallen komplett und es ist garantiert, dass wirklich nur eine Zeile gelöscht wird.

```

/**
 * fetchFromCursor() reads a result row or reaches the end of the result set
 * which is represented by the SQL array entry with the given SQL_ID.
 * In case the fetch operation encounters the end of the result set, the
 * signal END_REACHED is returned. Otherwise ROW_READ is signalled.
 * Note: Contrary to other RDBMS like DB2, the data in the row is NOT brought
 * into the method's local data space. Instead, the row's contents must be
 * read value by value with the column-oriented getter methods.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int fetchFromCursor(int SQL_ID)
{
    int      result;
    int      fetchResult;
    boolean   rowFetched;
    String    methodName = "fetchFromCursor()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID    = " + SQL_ID); }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                               SQL_ID, methodName); }

    // try to read the 1st or next row
    try
    {
        rowFetched = resultSet.next();

        if (rowFetched)
            { fetchResult = ROW_FETCHED; }
        else
            { fetchResult = END_REACHED; }

        if ((result = SQL_Register.storeFetchResults(SQL_ID, rowFetched)) !=
            SQL_Code.OK.getCode())
            { return result; }
    }
}

```

```

catch (SQLException ex)
{
    return handleStdSQL_Exception(SQL_Code.FETCH_CURSOR_FAILED,
                                  SQL_ID, methodName, ex);
}

handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());

return fetchResult;
}

```

Anmerkungen

- `fetchFromCursor()` liest eine Zeile der Ergebnismenge. Falls diese leer ist, oder der vorherige Fetch deren letzte Zeile in Lesefolge beschafft hat, meldet er `END_REACHED` zurück.
- Andernfalls beschafft der Fetch alle Spaltenwerte, die der ihm zugrundeliegende Select angefordert hat. Wenn der Cursor auf einer Basistabelle operiert, findet erst in diesem Moment die Projektion – also die Auswahl der Ergebnisspalten – statt.
- Die Methode `storeFetchResults()` in `SQL_Register` registriert, ob eine Zeile oder keine Zeile gelesen wurde, setzt daraufhin den Lesestatus im aktuellen Eintrag und erhöht im ersten Fall den Zeilenzähler.
- `fetchFromCursor()` überträgt keine Spalteninhalte in den Adressbereich des Anwendungsprogramms. Dieses muss die innerhalb des RDBMS bereitgestellten Daten einzeln mit den jeweils typkompatiblen `getXXXColumnValue()`-Methoden abrufen, wobei xxx den Datentyp repräsentiert.

```

/**
 * insertRow() inserts a row from the staging position.
 * Afterwards the cursor is placed before the first existing row in order
 * to avoid confusion.
 */
public static int insertRow(int SQL_ID)
{
    int      result;
    String    methodName = "insertRow()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID    = " + SQL_ID); }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                               SQL_ID, methodName); }
}

```

```

// insert the row
try
    { resultSet.insertRow(); }
catch (SQLException ex)
{
    return handleStdSQL_Exception(SQL_Code.INSERT_ROW_FAILED,
                                   SQL_ID, methodName, ex);
}

// position before the 1st existing row
try
    { resultSet.beforeFirst(); }
catch (SQLException ex)
{
    return handleStdSQL_Exception(SQL_Code.BEFORE_FIRST_FAILED,
                                   SQL_ID, methodName, ex);
}

return handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());
}

```

Anmerkungen

- `insertRow()` ist für Nutzer anderer RDBMS als MySQL, beispielsweise DB2, ein Novum. Dort erfolgt das Einfügen neuer Zeilen nämlich stets per Insert Statement. MySQL macht es möglich, neue Zeilen auch per Cursor einzufügen. Hierfür gibt es die Staging Area, die als Montageplattform dient.
- Nachdem die in der Staging Area vorbereitete Zeile eingefügt wurde, positioniert die Methode wieder auf den Anfang der Ergebnismenge des Cursor. Damit gibt sie die in der Staging Area gebundenen Ressourcen wieder frei. Wenn Zeile um Zeile eingefügt wird, wiederholt sich der ganze Vorgang jedesmal, beginnend mit `moveToInsertRow()` und endend mit `insertRow()`.
- Es findet keine Zeilenzählung für cursor-basierte Inserts in den SQL Services statt. Das muss das jeweilige Anwendungsprogramm selbst erledigen.

```

/**
 * moveToInsertRow() moves the cursor to the staging position.
 * The values for the new row must then be assigned using the
 * updateXXXColumnValue() methods, where XXX is the type of the column.
 * After assigning all non-NULL values, the method insertRow() can be called.
 */
public static int moveToInsertRow(int SQL_ID)
{
    int result;
    String methodName = "moveToInsertRow()";
    ResultSet resultSet;

    if (Trace.METHOD.active())

```

```

        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID    = " + SQL_ID); }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                               SQL_ID, methodName); }

    // move the cursor to the staging position
    try
        { resultSet.moveToInsertRow(); }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.MOVE_TO_INSERT_ROW_FAIL,
                                       SQL_ID, methodName, ex);
    }

    return handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());
}

```

Anmerkungen

- `moveToInsertRow()` nimmt die üblichen Prüfungen vor und positioniert dann auf die Staging Area.
- Anfangs ist dieser Montageort leer. Die neue Zeile muss Schritt für Schritt mit Attributwerten für all jene Spalten befüllt werden, die als `NOT NULL` deklariert sind, damit sie überhaupt anschließend eingefügt werden kann. Optionale Spalteninhalte, falls vorhanden, können nun ebenfalls vorgegeben werden. Diese Zuweisungen sind typabhängig, erfordern also die Nutzung der jeweils typkompatiblen `updateXXXColumnValue()`-Methode, wobei `xxx` den Datentyp repräsentiert.

```

/**
 * updateRow() updates the row at the current cursor position.
 */
public static int updateRow(int SQL_ID)
{
    int      result;
    String    methodName = "updateRow()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID    = " + SQL_ID); }

```

```

        if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
            { return result; }

        if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
            { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                                   SQL_ID, methodName); }

        // update the row
        try
            { resultSet.updateRow(); }
        catch (SQLException ex)
            {
                return handleStdSQL_Exception(SQL_Code.UPDATE_ROW_FAILED,
                                                SQL_ID, methodName, ex);
            }

        return handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());
    }

```

Anmerkung

updateRow() ändert alle Spalteninhalte, für die zuvor mittels updateXXXColumnName() neue Inhalte bereitgestellt wurden. Beim DB2 wird dagegen verlangt, die zur Änderung vorgesehenen Spalten im Select Statement mit FOR UPDATE OF einzeln zu benennen. Generell müssen die betroffenen Spalten in der Ergebnisliste aufgeführt sein.

```

/**
 * closeCursor() closes a cursor for a given SQL_ID.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int closeCursor(int SQL_ID)
{
    int      result;
    String    methodName = "closeCursor()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID    = " + SQL_ID); }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

```

```

if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
    { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                          SQL_ID, methodName); }

try
    { resultSet.close(); }
catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.CURSOR_CLOSE_FAILED,
                                       SQL_ID, methodName, ex);
    }

SQL_Register.resetRowCount(SQL_ID);

result = SQL_Register.deregister(SQL_ID, resultSet);

return result;
}

```

Anmerkung

`closeCursor()` beendet die Verarbeitung der mit `openCursor()` eröffneten Ergebnismenge. Änderungen, die im Transaktionsmodus vorgenommen und bis dahin nicht per Commit bestätigt wurden, gehen verloren. Daher sollte ein solcher Cursor erst nach Commit geschlossen werden.

13.2.7 Attribute based methods

```

/**
 * checkForNullValue() checks whether the last column value read is NULL for
 * a given SQL_ID whose row was read beforehand via fetchFromCursor().
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int checkForNullValue(int SQL_ID)
{
    int      result;
    boolean  nullIndicator;
    String   methodName = "checkForNullValue()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID   = " + SQL_ID); }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())

```

```

        { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                                SQL_ID, methodName); }

    // try to check for a NULL value
    try
    {
        nullIndicator = resultSet.isNull();

        if (nullIndicator)
            { result = IS_NULL; }
        else
            { result = IS_NOT_NULL; }
    }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.NULL_CHECK_FAILED,
                                        SQL_ID, methodName, ex);
    }

    return result;
}

```

Anmerkungen

- In den meisten Fällen meldet Java SQL aus einer `getXXXColumnValue()`-Methode den Wert `null` zurück, falls das betreffende Attribut sowohl `NULLABLE` als auch im jeweils vorliegenden Fall tatsächlich in der gelesenen Zeile undefiniert ist, also das `NULL`-Attribut gesetzt ist.
- Nur in einigen Fällen, beispielsweise bei den Zahlen außer `BigDecimal`, wird stattdessen ein Default-Wert zurückgemeldet, der wie ein realer Inhalt aussieht.
- Die typneutrale Methode `checkForNullValue()` wird also nur in diesen wenigen Fällen benötigt. Sie ist nach dem Lesen eines Ergebnis-Attributs aufzurufen, das als `NULLABLE` deklariert ist.
- Dies ist die erste Methode, die ein Attribut in der aktuellen Zeile verwendet, also derjenigen Zeile, auf die der Cursor verweist, dessen SQL ID als Parameter erwartet wird. Es gibt hier keine Spaltennummer!
- Anders als DB2 in Kombination mit COBOL oder PL/1 kennt Java SQL keine Null-Indikatoren, die jedem einzelnen Attribut starr zugeordnet sind, für die gesamte Liste der Select-Ergebnisspalten bereitgestellt werden und jederzeit nach einem Lesezugriff abgefragt werden können.
- Der Unterschied zwischen `NULL` mit der Bedeutung „Spalteninhalt existiert in dieser Zeile nicht“ und `null` als programminterne Repräsentation von „ist undefiniert“ wird anhand von `checkForNullValue()` deutlich hervorgehoben. `Null` und `null` sind nicht dasselbe, werden aber teilweise von Java SQL scheinbar synonym verwendet. Aufpassen!

```

/**
 * getDateColumnValue() obtains the value of an date column for a given
 * SQL_ID whose row was read beforehand via fetchFromCursor().

```

```

* The column is identified by its number.
* If the SQL_ID is invalid / if the ResultSet handle is missing /
* if another error occurs, an error code is returned.
*/
public static Date getDateColumnValue(int SQL_ID,
                                     int columnIndex)
{
    Date      columnValue = null;
    String    methodName = "getDateColumnValue()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID      = " + SQL_ID);
        System.out.println("\t columnIndex = " + columnIndex);
    }

    if ((checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return null; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
    {
        internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                     SQL_ID, methodName);
        return null;
    }

    // try to get the Date column value
    try
        { columnValue = resultSet.getDate(columnIndex); }
    catch (SQLException ex)
    {
        handleStdSQL_Exception(SQL_Code.GET_DATE_FAILED,
                              SQL_ID, methodName, ex);
        return null;
    }

    handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());

    if (Trace.DETAIL.active())
        { System.out.println("\t columnValue = " + columnValue); }

    return columnValue;
}

```

Anmerkung

`getDateColumnValue()` erfordert den `import` von `java.sql.Date`. (siehe Kapitel „13.2.1 Einleitende Deklarationen“ auf Seite 364 f). Nach den üblichen Prüfschritten liest die Methode in der aktuellen Zeile dasjenige Date-Attribut aus, das die in `columnIndex` vorgegebene Spaltennummer besitzt. Die `SQL_Example`-Programme stützen sich dabei auf die Enumeration `ColumnID`. Dort werden die Spaltennummern und eventuell die Positionsnummern definiert. Wenn der Spalteninhalt laut Tabellendefinition nullable und zugleich undefiniert ist, meldet Java SQL das Ergebnis `null` zurück.

```
/**
 * getDecimalColumnValue() obtains the value of an decimal column for a given
 * SQL_ID whose row was read beforehand via fetchFromCursor().
 * The column is identified by its number.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static BigDecimal getDecimalColumnValue(int SQL_ID,
                                              int columnIndex)
{
    BigDecimal columnValue = null;
    String      methodName = "getDecimalColumnValue()";
    ResultSet   resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID      = " + SQL_ID);
        System.out.println("\t columnIndex = " + columnIndex);
    }

    if ((checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return null; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
    {
        internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                     SQL_ID, methodName);
        return null;
    }

    // try to get the Date column value
    try
        { columnValue = resultSet.getBigDecimal(columnIndex); }
    catch (SQLException ex)
    {
```

```

        handleStdSQL_Exception(SQL_Code.GET_DECIMAL_FAILED,
                               SQL_ID, methodName, ex);
        return null;
    }

    handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());

    if (Trace.DETAIL.active())
        { System.out.println("\t columnValue = " + columnValue); }

    return columnValue;
}

```

Anmerkung

`getDecimalColumnValue()` erfordert den import von `java.math.BigDecimal`. (siehe Kapitel „13.2.1 Einleitende Deklarationen“ auf Seite 364 f). Nach den üblichen Prüfschritten liest die Methode in der aktuellen Zeile dasjenige `BigDecimal`-Attribut aus, das die in `columnIndex` vorgegebene Spaltennummer besitzt. Wenn der Spalteninhalt laut Tabellendefinition nullable und zugleich undefiniert ist, meldet Java SQL das Ergebnis `null` zurück.

```

/**
 * getIntColumnValue() obtains the value of an integer column for a given
 * SQL_ID whose row was read beforehand via fetchFromCursor().
 * The column is identified by its number.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static Integer getIntColumnValue(int SQL_ID,
                                         int columnIndex)
{
    Integer    columnValue = null;
    String     methodName = "getIntColumnValue()";
    ResultSet  resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID      = " + SQL_ID);
        System.out.println("\t columnIndex = " + columnIndex);
    }

    if ((checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return null; }
}

```

```

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
    { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                          SQL_ID, methodName); }

    // try to get the Integer column value
    try
    { columnValue = resultSet.getInt(columnIndex); }
    catch (SQLException ex)
    {
        handleStdSQL_Exception(SQL_Code.GET_INT_FAILED,
                               SQL_ID, methodName, ex);

        return null;
    }

    handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());

    if (Trace.DETAIL.active())
    { System.out.println("\t columnValue = " + columnValue); }

    return columnValue;
}

```

Anmerkung

Nach den üblichen Prüfschritten liest die Methode in der aktuellen Zeile dasjenige Integer-Attribut aus, das die in `columnIndex` vorgegebene Spaltennummer besitzt. Wenn der Spalteninhalt laut Tabellendefinition nullable und zugleich undefiniert ist, meldet Java SQL das Ergebnis `null` zurück.

```

/**
 * getStringColumnValue() obtains the value of a String = CHAR column for a
 * given SQL_ID whose row was read beforehand via fetchFromCursor().
 * The column is identified by its number.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static String getStringColumnValue(int SQL_ID,
                                           int columnIndex)
{
    String    columnValue = null;
    String    methodName = "getStringColumnValue()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
    { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {

```

```

        System.out.println("\t SQL_ID      = " + SQL_ID);
        System.out.println("\t columnIndex = " + columnIndex);
    }

    if ((checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
    { return null; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
    {
        internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                      SQL_ID, methodName);
        return null;
    }

    // try to get the String column value
    try
    { columnValue = resultSet.getString(columnIndex); }
    catch (SQLException ex)
    {
        handleStdSQL_Exception(SQL_Code.GET_STRING_FAILED,
                               SQL_ID, methodName, ex);
        return null;
    }

    handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());

    if (Trace.DETAIL.active())
    { System.out.println("\t columnValue = " + columnValue); }

    return columnValue;
}

```

Anmerkung

`getStringColumnValue()` liest nach den üblichen Prüfschritten in der aktuellen Zeile dasjenige String-Attribut aus, das die in `columnIndex` vorgegebene Spaltennummer besitzt. Wenn der Spalteninhalt nullable und zugleich undefiniert ist, meldet Java SQL das Ergebnis `null` zurück. Ein leerer String ist dagegen nicht `NULL`, sondern besitzt lediglich die Länge 0.

```

/**
 * getTimestampColumnValue() obtains the value of an date&time column for a
 * given SQL_ID whose row was read beforehand via fetchFromCursor().
 * The column is identified by its number.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static Timestamp getTimestampColumnValue(int SQL_ID,

```

```

                                int columnIndex)
{
    Timestamp columnValue = null;
    String    methodName = "getTimestampColumnValue()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID      = " + SQL_ID);
        System.out.println("\t columnIndex = " + columnIndex);
    }

    if ((checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return null; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
    {
        internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                      SQL_ID, methodName);
        return null;
    }

    // try to get the Timestamp column value
    try
        { columnValue = resultSet.getTimestamp(columnIndex); }
    catch (SQLException ex)
    {
        handleStdSQL_Exception(SQL_Code.GET_TS_FAILED,
                               SQL_ID, methodName, ex);
        return null;
    }

    handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());

    if (Trace.DETAIL.active())
        { System.out.println("\t columnValue = " + columnValue); }

    return columnValue;
}

```

Anmerkung

getTimestampColumnValue() erfordert den import von java.sql.Timestamp. (siehe Kapitel „13.2.1 Einleitende Deklarationen“ auf Seite 364 f). Nach den üblichen Prüfschritten liest die Methode in der aktu-

ellen Zeile dasjenige Timestamp-Attribut aus, das die in `columnIndex` vorgegebene Spaltennummer besitzt. Wenn der Spalteninhalt nullable und zugleich undefiniert ist, meldet Java SQL das Ergebnis `null` zurück.

```
/**
 * setToNullValue() sets a column value in the current row (pointed to by the
 * cursor) to NULL for a given SQL_ID.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int setToNullValue(int SQL_ID,
                                int columnIndex)
{
    int      result;
    String    methodName = "setToNullValue()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID    = " + SQL_ID); }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                               SQL_ID, methodName); }

    // try to set the NULL value
    try
    { resultSet.updateNull(columnIndex); }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.UPDATE_TO_NULL_FAILED,
                                       SQL_ID, methodName, ex);
    }

    return handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());
}
```

Anmerkung

`setToNullValue()` ist eine typneutrale Methode, um Attribute, die `NULLABLE` sind, auf `NULL` zu setzen. Sie nimmt eingangs die üblichen Prüfschritte vor und setzt dann das `NULL`-Attribut. Was auch immer sich in der aktuellen Zeile auf der Position dieses Attributs befinden mag, wird dadurch für ungültig erklärt.

```

/**
 * updateDateColumnValue() updates the value of a DATE column for
 * a given SQL_ID whose row was read beforehand via fetchFromCursor().
 * The column is identified by its number.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int updateDateColumnValue(int SQL_ID,
                                       int columnIndex,
                                       Date newDateValue)
{
    int result;
    String methodName = "updateDateColumnValue()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID      = " + SQL_ID);
        System.out.println("\t columnIndex = " + columnIndex);
        System.out.println("\t newDateValue = " + newDateValue);
    }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                               SQL_ID, methodName); }

    // try to update the Date column value
    try
        { resultSet.updateDate(columnIndex, newDateValue); }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.UPDATE_DATE_FAILED,
                                       SQL_ID, methodName, ex);
    }

    return handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());
}

```

Anmerkungen

- `updateDateColumnValue()` belegt eine DATE-Spalte in der aktuellen Zeile mit einem neuen Inhalt. Wenn diese Spalte nullable ist, wird zugleich das `NULL`-Attribut ausgeschaltet.
- Die aktuelle Zeile ist diejenige, auf welche der Cursor positioniert ist. Sie existiert entweder in der Tabelle, die dieser Cursor bearbeitet oder sie steht in der Staging Area.
- Die Methode löst im ersten Fall keinen Zugriff auf die aktuelle Zeile in der Tabelle aus, sondern greift auf einen internen Speicherbereich zu. Der Zugriff erfolgt später durch `updateRow()`.
- Im zweiten Fall dient der Aufruf dazu, eine noch unfertige Zeile weiter aufzubauen. Auch die Staging Area ist ein interner Speicherbereich. Der Zugriff erfolgt später durch `insertRow()`.

```
/**
 * updateDecimalColumnValue() updates the value of a DECIMAL column for
 * a given SQL_ID whose row was read beforehand via fetchFromCursor().
 * The column is identified by its number.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int updateDecimalColumnValue(int      SQL_ID,
                                           int      columnIndex,
                                           BigDecimal newDecimalValue)
{
    int      result;
    String    methodName = "updateDecimalColumnValue()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID          = " + SQL_ID);
        System.out.println("\t columnIndex    = " + columnIndex);
        System.out.println("\t newDecimalValue = " + newDecimalValue);
    }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                               SQL_ID, methodName); }

    // try to update the Date column value
    try
        { resultSet.updateBigDecimal(columnIndex, newDecimalValue); }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.UPDATE_DECIMAL_FAILED,
```

```

        SQL_ID, methodName, ex);
    }

    return handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());
}

```

Anmerkung

`updateDecimalColumnValue()` belegt eine DECIMAL-Spalte in der aktuellen Zeile mit einem neuen Inhalt. Wenn diese Spalte nullable ist, wird zugleich das NULL-Attribut ausgeschaltet. Ansonsten gelten die vorhergehenden Anmerkungen zu `updateDateColumnValue()`.

```

/**
 * updateIntColumnValue() updates the value of an integer column for a given
 * SQL_ID whose row was read beforehand via fetchFromCursor().
 * The column is identified by its number.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int updateIntColumnValue(int    SQL_ID,
                                       int    columnIndex,
                                       int    newIntValue)
{
    int    result;
    String  methodName = "updateIntColumnValue()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID      = " + SQL_ID);
        System.out.println("\t columnIndex = " + columnIndex);
        System.out.println("\t newIntValue = " + newIntValue);
    }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                               SQL_ID, methodName); }

    // try to update the integer column value
    try
        { resultSet.updateInt(columnIndex, newIntValue); }

```

```

        catch (SQLException ex)
        {
            return handleStdSQL_Exception(SQL_Code.UPDATE_INT_FAILED,
                                           SQL_ID, methodName, ex);
        }

        return handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());
    }

```

Anmerkung

`updateIntColumnValue()` belegt eine INTEGER-Spalte in der aktuellen Zeile mit einem neuen Inhalt. Wenn diese Spalte nullable ist, wird zugleich das NULL-Attribut ausgeschaltet. Ansonsten gelten die vorhergehenden Anmerkungen zu `updateDateColumnValue()`.

```

/**
 * updateStringColumnValue() updates the value of a String = CHAR column for
 * a given SQL_ID whose row was read beforehand via fetchFromCursor().
 * The column is identified by its number.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int updateStringColumnValue(int    SQL_ID,
                                           int    columnIndex,
                                           String newStringValue)

{
    int    result;
    String  methodName = "updateStringColumnValue()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: " + methodName); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID          = " + SQL_ID);
        System.out.println("\t columnIndex   = " + columnIndex);
        System.out.println("\t newStringValue = " + newStringValue);
    }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
        { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
        { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
                               SQL_ID, methodName); }

```

```

        // try to update the String column value
        try
        {
            resultSet.updateString(columnIndex, newStringValue);
        }
        catch (SQLException ex)
        {
            return handleStdSQL_Exception(SQL_Code.UPDATE_STRING_FAILED,
                                           SQL_ID, methodName, ex);
        }

        return handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());
    }

```

Anmerkungen

- `updateStringColumnValue()` belegt eine CHAR- oder VARCHAR-Spalte in der aktuellen Zeile mit einem neuen Inhalt. Wenn diese Spalte nullable ist, wird zugleich das NULL-Attribut ausgeschaltet.
- CHAR-Spalten haben eine feste Länge. In der Dokumentation der Klasse `ResultSet` wird nicht erklärt, was geschieht, wenn ein übergebener String kürzer oder länger ist. Es empfiehlt sich, keine Längenabweichung von der datenbankseitigen Vorgabe zuzulassen, um definierte Ergebnisse zu erzielen.
- VARCHAR-Spalten können Zeichenketten vom leeren String bis hin zur maximalen String-Länge enthalten, die beim Anlegen der Tabelle definiert wurde. Sie werden häufig dazu verwendet, variabel lange Texte abzuspeichern. Auch hier ist undefiniert, wie Java SQL sich verhält, wenn ein zu langer String angeliefert wird. Er wird spätestens von MySQL abgeschnitten, was nicht im Sinn der beabsichtigten Datenbanknutzung sein dürfte. Das Programm sollte daher auch hier die datenbankseitige Vorgabe respektieren.
- Ansonsten gelten die Anmerkungen zu `updateDateColumnValue()`.

```

/**
 * updateTimestampColumnValue() updates the value of a Timestamp column for
 * a given SQL_ID whose row was read beforehand via fetchFromCursor().
 * The column is identified by its number.
 * If the SQL_ID is invalid / if the ResultSet handle is missing /
 * if another error occurs, an error code is returned.
 */
public static int updateTimestampColumnValue(int SQL_ID,
                                             int columnIndex,
                                             Timestamp newTimestampValue)
{
    int result;
    String methodName = "updateTimestampColumnValue()";
    ResultSet resultSet;

    if (Trace.METHOD.active())
    {
        System.out.println("\n SQL_Util: " + methodName);
    }

    if (Trace.DETAIL.active())
    {
        System.out.println("\t SQL_ID          = " + SQL_ID);
    }

```

```

        System.out.println("\t columnIndex      = " + columnIndex);
        System.out.println("\t newTimestampValue = " + newTimestampValue);
    }

    if ((result = checkSQL_ID(SQL_ID, methodName)) != SQL_Code.OK.getCode())
    { return result; }

    if ((resultSet = SQL_Register.getResultSet(SQL_ID)) == null)
    { return internalError(SQL_Code.MISS_RESULT_SET_H_ENTRY,
        SQL_ID, methodName); }

    // try to update the Timestamp column value
    try
    { resultSet.updateTimestamp(columnIndex, newTimestampValue); }
    catch (SQLException ex)
    {
        return handleStdSQL_Exception(SQL_Code.UPDATE_TS_FAILED,
            SQL_ID, methodName, ex);
    }

    return handleResults(SQL_Code.OK.getCode(), SQL_Code.OK.getMessage());
}

```

Anmerkungen

- `updateTimestampColumnValue()` belegt eine `TIMESTAMP`- oder `DATETIME`-Spalte in der aktuellen Zeile mit einem neuen Inhalt. Wenn diese Spalte nullable ist, wird zugleich das `NULL`-Attribut ausgeschaltet.
- `TIMESTAMP`-Spalten können Werte von '1970-01-01 00:00:01' UTC bis '2038-01-19 03:14:07' UTC enthalten. Die Sekunden können bis zu 6 Nanosekundenstellen haben; die Anzahl der datenbankseitigen Nanosekundenstellen ist also beim Anlegen einer `TIMESTAMP`-Spalte von 0 bis 6 wählbar. Übergibt das Programm mehr Stellen, dann wird abgeschnitten oder aufgerundet, je nach gewählter SQL Mode Option `TIME_TRUNCATE_FRACTIONAL`.
- `DATETIME`-Spalten können Werte von '1000-01-01 00:00:00' bis '9999-12-31 23:59:59' enthalten. Es gibt hier keine Sekundenbruchteile.
- `updateTimestampColumnValue()` unterscheidet nicht zwischen den beiden Timestamp-Typen. Das Anwendungsprogramm ist allein dafür verantwortlich, ein korrektes Timestamp-Objekt als Parameter zu übergeben.
- Ansonsten gelten die Anmerkungen zu `updateDateColumnValue()`.

13.2.8 SQL ID check & exception handling methods

```

/**
 * checkSQL_ID() checks the SQL_ID parameter for correctness.
 */
private static int checkSQL_ID(int SQL_ID, String methodName)
{
    String errorMsg;

```

```

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: checkSQL_ID() for " + methodName); }

    if (Trace.DETAIL.active())
        { System.out.println("\t SQL_ID    = " + SQL_ID); }

    if (SQL_ID < 0 || SQL_ID >= SQL_Register.getSQL_ArrayEntries())
    {
        errorMsg = SQL_Code.ILLEGAL_SQL_ID.getMessage() +
            " (" + (SQL_Register.getSQL_ArrayEntries()-1) +
            "): >" + SQL_ID + "< in " + methodName;

        return handleResults(SQL_Code.ILLEGAL_SQL_ID.getCode(), errorMsg);
    }

    return SQL_Code.OK.getCode();
}

```

Anmerkung

checkSQL_ID() prüft die übergebene SQL ID auf Einhaltung des Bereichs von 0 bis zur Anzahl der SQL Array-Einträge minus 1. Die SQL Services funktionieren nur dann sicher, wenn jede Aktion unterhalb der Connection-Ebene mit der jeweils korrekten SQL ID angefordert wird. Deshalb verwenden die nachfolgenden SQL_Example-Programme die Enumeration SQL_ID, welche ab SQL_Example1 sowohl die SQL IDs als auch das jeweils zugehörige SQL Statement auflistet. SQL_Example0 enthält die SQL Statements dagegen noch als String-Konstante in der main().

```

/**
 * handleStdSQL_Exception() prepares the data for handleSQL_Exception()
 * in standard cases.
 */
private static int handleStdSQL_Exception(SQL_Code    codeInstance,
                                           int         SQL_ID,
                                           String      methodName,
                                           SQLException excp)
{
    String errorMsg = codeInstance.getMessage() +
        " for SQL ID >" + SQL_ID + "< in " + methodName;

    return handleSQL_Exception(SQL_ID, excp,
                               codeInstance.getCode(),
                               errorMsg);
}

```

Anmerkung

`handleStdSQL_Exception()` dient dazu, eine standardisierte Reaktion auf eine SQL Exception auszulösen. Ein typischer Aufruf sieht wie folgt aus:

```
return handleStdSQL_Exception(SQL_Code.OPEN_CURSOR_FAILED,
                             SQL_ID, methodName, ex);
```

Der erste Parameter ist die Instanz von `OPEN_CURSOR_FAILED` in der Klasse `SQL_Code` der SQL Services. Aus dieser Instanz entnimmt `handleStdSQL_Exception()` die beiden Attribute `code` und `message` per Getter-Methode. Die Fehlernachricht wird hier nur vorbereitet und dann von `handleSQL_Exception()` verwertet.

```
/**
 * handleSQL_Exception() prints the data of the exception.
 */
private static int handleSQL_Exception(int SQL_ID,
                                       SQLException excp,
                                       int exitCode,
                                       String errorMsg)
{
    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Util: handleSQL_Exception()"); }

    if (Trace.DETAIL.active())
    {
        System.out.println("\n exitCode = " + exitCode +
                           ", errorMsg = " + errorMsg);

        System.out.println("SQL_ID = " + SQL_ID);
    }

    // save the exitCode and the errorMsg in the SQL_Register
    SQL_Register.handleResults(exitCode, errorMsg);

    // pause the thread: let the print output complete
    try
        { Thread.sleep(1000); }
    catch (Exception ex) { }

    excp.printStackTrace();

    SQL_Register.showSQL_Array();

    return exitCode;
}
```

Anmerkungen

`handleSQL_Exception()`

- gibt die von `handleStdSQL_Exception()` erzeugte Fehlermeldung aus, falls gewünscht,
- speichert sie und den Fehlercode als Inhalte der Klassenvariablen `errorCode` und `errorMessage` in der Klasse `SQL_Register` per `handleResults()`.
- wartet etwa eine Sekunde wegen der Ausgabe von Exception-Meldungen seitens Java SQL durch einen anderen Thread (den Java automatisch erzeugt),
- druckt den Stack Trace, also die hierarchische Liste der Codepositionen bei der Exception, und
- listet alle Einträge im SQL Array mit den dortigen Detailinformationen auf.

Diese Methode kann nur in einer Offline-Umgebung sinnvoll eingesetzt werden. Online-Nutzer dürfen keine Exception-Meldungen erhalten.

13.2.9 Small miscellaneous methods

```
public static int internalError(SQL_Code codeInstance,
                                int      SQL_ID,
                                String   methodName)
{
    String errorMsg = codeInstance.getMessage() +
        " for SQL ID >" + SQL_ID + "< in " + methodName;

    return handleResults(codeInstance.getCode(),
                        errorMsg);
}
```

Anmerkung

`internalError()` arbeitet wie `handleStdSQL_Exception()`. Hier geht es aber um applikationsbezogene Fehler, beispielsweise das Fehlen von Handles wegen versäumter Compile- oder Open-Aufrufe. `handleResults()` ist eine Methode der Klasse `SQL_Register`, die nur dann einen Fehlertext ausgibt, wenn das Print-Flag eingeschaltet ist.

```
/**
 * The getXXX() / hasXXX() / setPrintFlag() methods encapsulate the
 * corresponding SQL_Register methods.
 */

public static Connection getConnectionHandle()
{ return SQL_Register.getConnectionHandle(); }

public static int getErrorCode()
{ return SQL_Register.getErrorCode(); }

public static int getErrorCount()
{ return SQL_Register.getErrorCount(); }

public static String getErrorMessage()
{ return SQL_Register.getErrorMessage(); }
```

```

public static int      getExecCalls(int SQL_ID)
{ return SQL_Register.getExecCalls(SQL_ID); }

public static int      getRowCount(int SQL_ID)
{ return SQL_Register.getRowCount(SQL_ID); }

public static int      getMaxSQL_Entries()
{ return SQL_Register.getSQL_ArrayEntries(); }

public static boolean  getPrintFlag()
{ return SQL_Register.getPrintFlag(); }

public static boolean  hasInputData(int SQL_ID)
{ return SQL_Register.hasInputData(SQL_ID); }

public static boolean  hasReachedEOS(int SQL_ID)
{ return SQL_Register.hasReachedEOS(SQL_ID); }

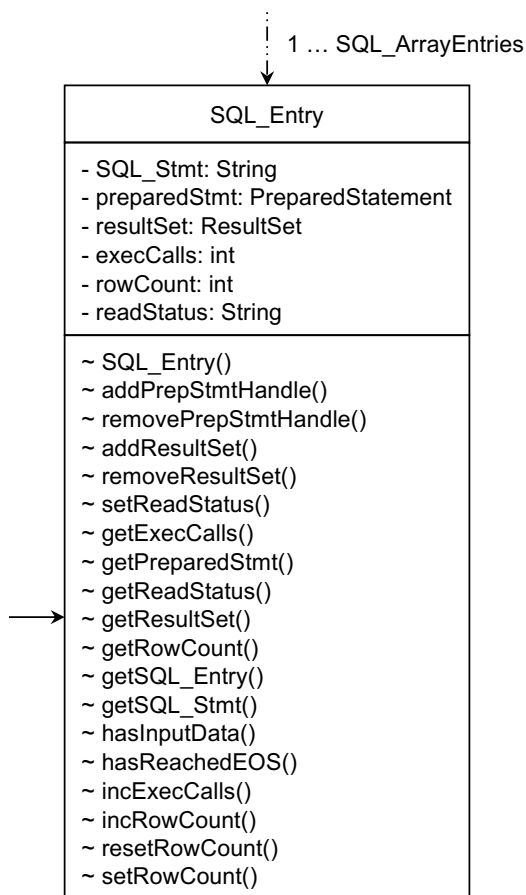
public static void      setPrintFlag(boolean errMsgPrint)
{ SQL_Register.setPrintFlag(errMsgPrint); }

```

Klassen SQL_Entry und SQL_Register

Die Klasse SQL_Register verwaltet den SQL Array, dessen Einträge durch die Klasse SQL_Entry definiert sind. Um die Register-Methoden zu verstehen, muss zuvor SQL_Entry gezeigt werden. Die Methoden dieser Klasse sind so trivial, dass sie im folgenden nicht kommentiert werden.

13.3 Klasse SQL_Entry



```
package sql_service;
```

```
import java.sql.PreparedStatement;
```

```
import java.sql.ResultSet;
```

```
/**
```

```
 * The class SQL_Entry describes the data and the behaviour of an SQL-related
 * array entry within the SQL_Register.
 */
```

```
*/
```

```
class SQL_Entry
```

```
{
```

```
    private final String          SQL_Stmt;
    private      PreparedStatement preparedStmt;
    private      ResultSet        resultSet;
    private      int              execCalls;
    private      int              rowCount;
    private      String           readStatus;
```

```
    /**
```

```

    * constructor for objects of class SQL_Entry
    */
SQL_Entry(String SQL_Stmt)
{
    this.SQL_Stmt      = SQL_Stmt;
    this.preparedStmt  = null;
    this.resultSet     = null;
    this.execCalls     = 0;
    this.rowCount      = 0;
    this.readStatus    = " ";
}

/**
 * addPrepStmtHandle() stores the handle of the PreparedStatement object.
 */
void addPrepStmtHandle(PreparedStatement preparedStmt)
{ this.preparedStmt = preparedStmt; }

/**
 * removePrepStmtHandle() removes the handle of the PreparedStatement object.
 */
void removePrepStmtHandle(PreparedStatement preparedStmt)
{ this.preparedStmt = null; }

/**
 * addResultSet() stores the handle of the result set for the given query.
 */
void addResultSet(ResultSet resultSet)
{ this.resultSet = resultSet; }

/**
 * removeResultSet() removes the handle of the result set for the given query.
 */
void removeResultSet(ResultSet resultSet)
{ this.resultSet = null; }

/**
 * setReadStatus() sets the readStatus to 'D' = Data or to 'E' = End,
 * depending on the result of the last FETCH statement.
 */
void setReadStatus(boolean rowFetched)
{
    if (rowFetched)
        { this.readStatus = "D"; }
    else
        { this.readStatus = "E"; }
}

```

```

int          getExecCalls()      { return this.execCalls; }

PreparedStatement getPreparedStmt() { return this.preparedStmt; }

String       getReadStatus()    { return this.readStatus; }

ResultSet    getResultSet()     { return this.resultSet; }

int          getRowCount()       { return this.rowCount; }

SQL_Entry    getSQL_Entry()      { return this; }

String       getSQL_Stmt()       { return this.SQL_Stmt; }

boolean      hasInputData()
{ return ("D".equals(this.readStatus)); }

boolean      hasReachedEOS()
{ return ("E".equals(this.readStatus)); }

void         incExecCalls()       { this.execCalls++; }

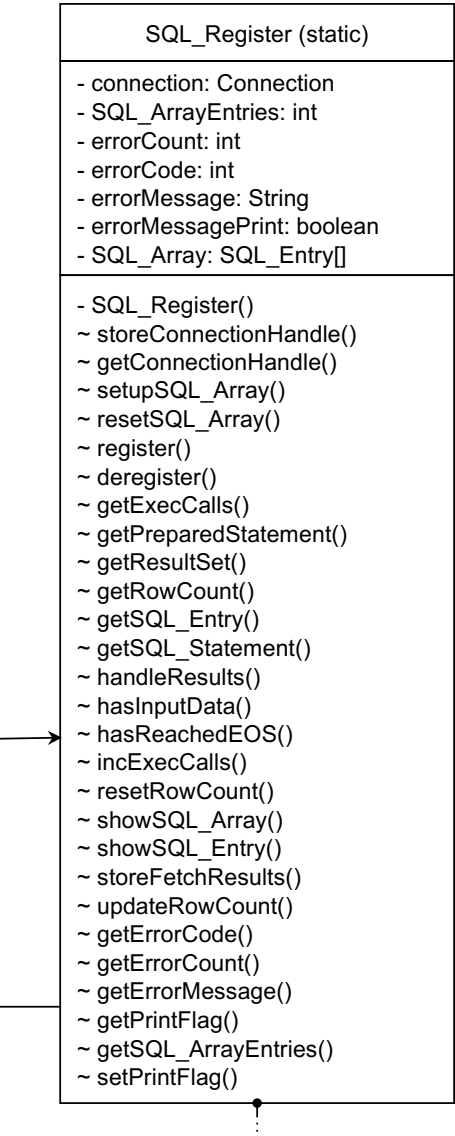
void         incRowCount()        { this.rowCount++; }

void         resetRowCount()      { this.rowCount = 0; }

void         setRowCount(int value)
{ this.rowCount = value; }
}

```

13.4 Klasse SQL_Register



Die Klasse `SQL_Register` enthält Klassenvariablen und führt Instanzvariable im SQL Array. Sie ist deutlich anders als `SQL_Util` aufgebaut. Die Gliederung von `SQL_Util` in Schichten von Methoden ist hier nicht so klar ausgeprägt, weil die Verwaltung und Verwendung des SQL Array im Vordergrund stehen. Viele Methoden sind so trivial, dass sie keine Kommentierung erfordern.

13.4.1 Einleitende Deklarationen

```
package sql_service;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.PreparedStatement;
```

```

/**
 * The class SQL_Register is the repository for the SQL related references and
 * for associated information.
 * It supports the class SQL_Util by
 * - allocating the SQL array, based on the "SQL_Entries" parameter,
 * - holding the connection handle,
 * - registering new statements plus the associated handles and de-registering
 *   them on demand,
 * - checking the requests against the repository data,
 * - returning the correct preparedStatement and resultSet references,
 * - counting the executions of SQL statements,
 * - storing the result row counts (by delegation),
 * - returning the SQL statement for a given SQL ID,
 * - outputting entry-related data on request (example: shutdown imminent), and
 * - returning error feedback values to the callers, if necessary.
 * The methods of SQL_Register are private (constructor only) or packet private.
 */
class SQL_Register
{
    private static Connection    connection;
    private static int          SQL_ArrayEntries;
    private static int          errorCount;
    private static int          errorCode;
    private static String        errorMessage;
    private static boolean       errorMessagePrint;
    private static SQL_Entry[]   SQL_Array;

    static
    {
        SQL_ArrayEntries = 0;
        errorCount       = 0;
        errorCode        = SQL_Code.OK.getCode();
        errorMessage     = SQL_Code.OK.getMessage();
        errorMessagePrint = false;
    }

    /**
     * dummy constructor for objects of class SQL_Register
     */
    private SQL_Register() { }

```

13.4.2 Connection-Methoden

```

/**
 * storeConnectionHandle() stores the static connection handle.
 */
static void storeConnectionHandle(Connection conn)

```

```

{
    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: storeConnectionHandle()"); }

    connection = conn;
}

/**
 * getConnectionHandle() returns the static connection handle.
 */
static Connection getConnectionHandle()
{
    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: getConnectionHandle()"); }

    return connection;
}

```

13.4.3 Methoden zum Allokieren und Deallokieren des SQL Array

```

/**
 * setupSQL_Array() allocates the SQL_Array. If SQL_ArrayEntries is already
 * positive, the array exists and the request is rejected.
 * The size parameter SQL_Entries must be positive.
 */
static int setupSQL_Array(int SQL_Entries)
{
    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: setupSQL_Array(" +
            SQL_Entries + ")"); }

    if (SQL_ArrayEntries > 0) // 2nd call?
    {
        return handleResults(SQL_Code.MAX_ENTRIES_DEFINED.getCode(),
            SQL_Code.MAX_ENTRIES_DEFINED.getMessage());
    }

    if (SQL_Entries <= 0)
    {
        return handleResults(SQL_Code.ILLEGAL_MAX_ENTRIES.getCode(),
            SQL_Code.ILLEGAL_MAX_ENTRIES.getMessage());
    }

    SQL_ArrayEntries = SQL_Entries;
    SQL_Array = new SQL_Entry[SQL_Entries];

    return handleResults(SQL_Code.OK.getCode(),

```

```

        SQL_Code.OK.getMessage());
    }

    /**
     * resetSQL_Array() sets SQL_ArrayEntries to zero and "deletes" the SQL_Array.
     */
    static void resetSQL_Array()
    {
        if (Trace.METHOD.active())
            { System.out.println("\n SQL_Register: resetSQL_Array()"); }

        SQL_ArrayEntries = 0;
        SQL_Array        = null;
    }

```

13.4.4 Methoden zum Verwalten der SQL Array-Einträge

```

    /**
     * This register() method assigns a new SQL array entry where it stores the
     * new SQL statement. The entry with the given - valid - SQL_ID must not be
     * in use and the SQL statement must exist.
     */
    static int register(int    SQL_ID,
                       String SQL_Stmt)
    {
        String errorMsg;

        if (Trace.METHOD.active())
            { System.out.println("\n SQL_Register: register(" +
                                SQL_ID + ", SQL_Stmt)"); }

        if (Trace.DETAILED.active())
            { System.out.println("\t SQL_Stmt = " + SQL_Stmt); }

        if (SQL_ID < 0 || SQL_ID >= SQL_ArrayEntries)
        {
            errorMsg = SQL_Code.ILLEGAL_SQL_ID.getMessage() +
                " (" + (SQL_ArrayEntries-1) +
                "): >" + SQL_ID + "<";

            return handleResults(SQL_Code.ILLEGAL_SQL_ID.getCode(),
                                errorMsg);
        }

        if (SQL_Stmt == null)
        {
            errorMsg = SQL_Code.SQL_STMT_MISSING.getMessage() +

```

```

        " for SQL ID >" + SQL_ID + "<";

        return handleResults(SQL_Code.SQL_STMT_MISSING.getCode(),
                               errorMsg);
    }

    if (SQL_Array[SQL_ID] != null)
    {
        errorMsg = SQL_Code.SQL_ENTRY_ALLOCATED.getMessage() +
            " for SQL ID >" + SQL_ID + "<";

        return handleResults(SQL_Code.SQL_ENTRY_ALLOCATED.getCode(),
                               errorMsg);
    }

    SQL_Array[SQL_ID] = new SQL_Entry(SQL_Stmt);

    if (Trace.SQL_AE_UPD.active())
        { showSQL_Entry(SQL_ID); }

    return handleResults(SQL_Code.OK.getCode(),
                          SQL_Code.OK.getMessage());
}

/**
 * This register() method adds the - existing - handle of the PreparedStatement
 * object to the SQL array entry with the given - valid - SQL_ID.
 * The entry must not already contain such a handle.
 */
static int register(int          SQL_ID,
                    PreparedStatement preparedStmt)
{
    String errorMsg;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: register(" +
            SQL_ID + ", preparedStmt)"); }

    if (SQL_ID < 0 || SQL_ID >= SQL_ArrayEntries)
    {
        errorMsg = SQL_Code.ILLEGAL_SQL_ID.getMessage() +
            " (" + (SQL_ArrayEntries-1) +
            "): >" + SQL_ID + "<";

        return handleResults(SQL_Code.ILLEGAL_SQL_ID.getCode(),
                               errorMsg);
    }
}

```

```

    if (SQL_Array[SQL_ID] == null)
    {
        errorMsg = SQL_Code.SQL_ENTRY_NOT_ALLOCATED.getMessage() +
            " for SQL ID >" + SQL_ID + "<";

        return handleResults(SQL_Code.SQL_ENTRY_NOT_ALLOCATED.getCode(),
            errorMsg);
    }

    if (SQL_Array[SQL_ID].getPreparedStmt() != null)
    {
        errorMsg = SQL_Code.PREP_HANDLE_ALLOCATED.getMessage() +
            " for SQL ID >" + SQL_ID + "<";

        return handleResults(SQL_Code.PREP_HANDLE_ALLOCATED.getCode(),
            errorMsg);
    }

    if (preparedStmt == null)
    {
        errorMsg = SQL_Code.MISS_PREP_STMT_H_PARM.getMessage() +
            " for SQL ID >" + SQL_ID + "<";

        return handleResults(SQL_Code.MISS_PREP_STMT_H_PARM.getCode(),
            errorMsg);
    }

    SQL_Array[SQL_ID].addPrepStmtHandle(preparedStmt);

    if (Trace.SQL_AE_UPD.active())
        { showSQL_Entry(SQL_ID); }

    return handleResults(SQL_Code.OK.getCode(),
        SQL_Code.OK.getMessage());
}

/**
 * This register() method adds the - existing - handle of the ResultSet
 * object to the SQL array entry with the given - valid - SQL_ID.
 * The entry must not already contain such a handle.
 */
static int register(int SQL_ID,
    ResultSet resultSet)
{
    String errorMsg;

```

```

if (Trace.METHOD.active())
    { System.out.println("\n SQL_Register: register(" +
        SQL_ID + ", resultSet)"); }

if (SQL_ID < 0 || SQL_ID >= SQL_ArrayEntries)
{
    errorMsg = SQL_Code.ILLEGAL_SQL_ID.getMessage() +
        " (" + (SQL_ArrayEntries-1) +
        "): >" + SQL_ID + "<";

    return handleResults(SQL_Code.ILLEGAL_SQL_ID.getCode(),
        errorMsg);
}

if (SQL_Array[SQL_ID] == null)
{
    errorMsg = SQL_Code.SQL_ENTRY_NOT_ALLOCATED.getMessage() +
        " for SQL ID >" + SQL_ID + "<";

    return handleResults(SQL_Code.SQL_ENTRY_NOT_ALLOCATED.getCode(),
        errorMsg);
}

if (SQL_Array[SQL_ID].getResultSet() != null)
{
    errorMsg = SQL_Code.RESULT_SET_ALLOCATED.getMessage() +
        " for SQL ID >" + SQL_ID + "<";

    return handleResults(SQL_Code.RESULT_SET_ALLOCATED.getCode(),
        errorMsg);
}

if (resultSet == null)
{
    errorMsg = SQL_Code.MISS_RESULT_SET_H_PARM.getMessage() +
        " for SQL ID >" + SQL_ID + "<";

    return handleResults(SQL_Code.MISS_RESULT_SET_H_PARM.getCode(),
        errorMsg);
}

SQL_Array[SQL_ID].addResultSet(resultSet);

if (Trace.SQL_AE_UPD.active())
    { showSQL_Entry(SQL_ID); }

return handleResults(SQL_Code.OK.getCode(),

```

```

        SQL_Code.OK.getMessage());
    }

```

Anmerkungen

- Die vorangehenden `register()`-Methoden dienen dazu, die Einträge im SQL Array anzulegen und mit den beiden Sorten von Handles zu füllen.
- Jede der drei Methoden verifiziert die Korrektheit der SQL ID. Beim ersten Aufruf darf der Eintrag im Array noch nicht existieren und bei den beiden anderen muss er existieren.
- Die für die jeweilige Aktion erforderliche inhaltliche Information, also das SQL Statement im ersten Fall und das von Java SQL gelieferte Handle in den beiden anderen Fällen muss vorhanden sein. Null-Werte werden nicht akzeptiert.

```

/**
 * This deregister() method removes the - existing - handle of the ResultSet
 * object from the SQL array entry with the given - valid - SQL_ID.
 * The entry must contain an identical handle.
 */
static int deregister(int      SQL_ID,
                      ResultSet resultSet)
{
    String errorMsg;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: deregister(" +
                             SQL_ID + ", resultSet)"); }

    if (SQL_ID < 0 || SQL_ID >= SQL_ArrayEntries)
    {
        errorMsg = SQL_Code.ILLEGAL_SQL_ID.getMessage() +
            " (" + (SQL_ArrayEntries-1) +
            "): >" + SQL_ID + "<";

        return handleResults(SQL_Code.ILLEGAL_SQL_ID.getCode(),
                             errorMsg);
    }

    if (SQL_Array[SQL_ID] == null)
    {
        errorMsg = SQL_Code.SQL_ENTRY_NOT_ALLOCATED.getMessage() +
            " for SQL ID >" + SQL_ID + "<";

        return handleResults(SQL_Code.SQL_ENTRY_NOT_ALLOCATED.getCode(),
                             errorMsg);
    }

    if (SQL_Array[SQL_ID].getResultSet() == null)

```

```

{
    errorMsg = SQL_Code.MISS_RESULT_SET_H_ENTRY.getMessage() +
        " for SQL ID >" + SQL_ID + "<";

    return handleResults(SQL_Code.MISS_RESULT_SET_H_ENTRY.getCode(),
        errorMsg);
}

if (resultSet == null)
{
    errorMsg = SQL_Code.MISS_RESULT_SET_H_PARM.getMessage() +
        " for SQL ID >" + SQL_ID + "<";

    return handleResults(SQL_Code.MISS_RESULT_SET_H_PARM.getCode(),
        errorMsg);
}

if (resultSet != SQL_Array[SQL_ID].getResultSet())
{
    errorMsg = SQL_Code.RESULT_SET_H_MISMATCH.getMessage() +
        " for SQL ID >" + SQL_ID + "<";

    return handleResults(SQL_Code.RESULT_SET_H_MISMATCH.getCode(),
        errorMsg);
}

SQL_Array[SQL_ID].removeResultSet(resultSet);

if (Trace.SQL_AE_UPD.active())
    { showSQL_Entry(SQL_ID); }

return handleResults(SQL_Code.OK.getCode(),
    SQL_Code.OK.getMessage());
}

/**
 * This deregister() method removes the - existing - handle of the
 * PreparedStatement object from the SQL array entry with the given - valid -
 * SQL_ID. The entry must contain an identical handle.
 */
static int deregister(int SQL_ID,
    PreparedStatement preparedStmt)
{
    String errorMsg;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: deregister(" +

```

```

        SQL_ID + ", preparedStmt)); }

if (SQL_ID < 0 || SQL_ID >= SQL_ArrayEntries)
{
    errorMsg = SQL_Code.ILLEGAL_SQL_ID.getMessage() +
        " (" + (SQL_ArrayEntries-1) +
        "): >" + SQL_ID + "<";

    return handleResults(SQL_Code.ILLEGAL_SQL_ID.getCode(),
        errorMsg);
}

if (SQL_Array[SQL_ID] == null)
{
    errorMsg = SQL_Code.SQL_ENTRY_NOT_ALLOCATED.getMessage() +
        " for SQL ID >" + SQL_ID + "<";

    return handleResults(SQL_Code.SQL_ENTRY_NOT_ALLOCATED.getCode(),
        errorMsg);
}

if (SQL_Array[SQL_ID].getPreparedStmt() == null)
{
    errorMsg = SQL_Code.MISS_PREP_STMT_H_ENTRY.getMessage() +
        " for SQL ID >" + SQL_ID + "<";

    return handleResults(SQL_Code.MISS_PREP_STMT_H_ENTRY.getCode(),
        errorMsg);
}

if (preparedStmt == null)
{
    errorMsg = SQL_Code.MISS_PREP_STMT_H_PARM.getMessage() +
        " for SQL ID >" + SQL_ID + "<";

    return handleResults(SQL_Code.MISS_PREP_STMT_H_PARM.getCode(),
        errorMsg);
}

if (preparedStmt != SQL_Array[SQL_ID].getPreparedStmt())
{
    errorMsg = SQL_Code.PREP_STMT_H_MISMATCH.getMessage() +
        " for SQL ID >" + SQL_ID + "<";

    return handleResults(SQL_Code.PREP_STMT_H_MISMATCH.getCode(),
        errorMsg);
}

```

```

SQL_Array[SQL_ID].removePrepStmtHandle(preparedStmt);

if (Trace.SQL_AE_UPD.active())
    { showSQL_Entry(SQL_ID); }

return handleResults(SQL_Code.OK.getCode(),
                    SQL_Code.OK.getMessage());
}

/**
 * This deregister() method removes the attributes of an SQL array entry.
 */
static int deregister(int SQL_ID)
{
    String errorMsg;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: deregister(" + SQL_ID + ")"); }

    if (SQL_ID < 0 || SQL_ID >= SQL_ArrayEntries)
    {
        errorMsg = SQL_Code.ILLEGAL_SQL_ID.getMessage() +
            " (" + (SQL_ArrayEntries-1) +
            "): >" + SQL_ID + "< (De-Register Input)";

        return handleResults(SQL_Code.ILLEGAL_SQL_ID.getCode(),
                            errorMsg);
    }

    SQL_Array[SQL_ID] = null;

    return handleResults(SQL_Code.OK.getCode(),
                        SQL_Code.OK.getMessage());
}

```

Anmerkungen

- Die vorangehenden `deregister()`-Methoden bauen spiegelbildlich zu den `register()`-Methoden die Handles ab und nullifizieren den jeweils genannten Wert beziehungsweise den gesamten Array-Eintrag.
- Jede der drei Methoden verifiziert die Korrektheit der SQL ID. Beim ersten und beim zweiten Aufruf muss der Eintrag im Array existieren. Beim dritten Aufruf, der den Eintrag „löscht“, wird nicht geprüft, ob dieser Eintrag überhaupt existiert.
- Soll ein Handle gelöscht werden, dann erwartet die betreffende `deregister()`-Methode ein identisches Handle als Parameter. Damit wird es unwahrscheinlicher, dass versehentlich ein noch aktives Handle gelöscht wird.

Die nun folgenden Methoden sind sämtlich trivial. Wenn sie eine SQL_ID als Parameter besitzen, dann wird diese geprüft, bevor die jeweilige Methode auf den SQL Array zugreift. Da die Getter-Methoden typgebundene Ergebnisse abliefern sollen, reagieren sie bei Aufruf mit fehlerhafter SQL_ID durch Rückgabe eines typgerechten, aber sinnlosen Werts. Das sollte eine Fehlerbehandlung auslösen. Nur im Fall der Methoden mit boolean-Ergebnis ist das nicht möglich. Dann könnte gegebenenfalls eine Exception stattfinden, was aber dem Geschick und Geschmack der Nutzer der SQL Services anheimgestellt wird.

```
/**
 * getExecCalls() obtains the exec call count for a given SQL ID.
 * If the SQL_ID is invalid or if the entry is empty, -1 is returned.
 */
static int getExecCalls(int SQL_ID)
{
    int execCalls = -1;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: getExecCalls(" + SQL_ID + ")"); }

    if (SQL_ID >= 0                &&
        SQL_ID <  SQL_ArrayEntries &&
        SQL_Array[SQL_ID] != null)
        { execCalls = SQL_Array[SQL_ID].getExecCalls(); }

    if (Trace.DETAIL.active())
        { System.out.println("\t execCalls = " + execCalls); }

    return execCalls;
}

/**
 * getPreparedStatement() obtains a PreparedStatement reference for a given
 * SQL_ID. If the SQL_ID is invalid or if the entry is empty, null is returned.
 * If the handle is nonexistent, getPreparedStatement() returns null.
 */
static PreparedStatement getPreparedStatement(int SQL_ID)
{
    PreparedStatement preparedStatement = null;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: getPreparedStatement(" +
                               SQL_ID + ")"); }

    if (SQL_ID >= 0                &&
        SQL_ID <  SQL_ArrayEntries &&
        SQL_Array[SQL_ID] != null)
        { preparedStatement = SQL_Array[SQL_ID].getPreparedStatement(); }
}
```

```

        if (Trace.SQL_AE_STATUS.active())
            { showSQL_Entry(SQL_ID); }

        return preparedStatement;
    }

    /**
     * getResultSet() obtains a ResultSet reference for a given SQL_ID.
     * If the SQL_ID is invalid or if the entry is empty, null is returned.
     * If the handle is nonexistent, getResultSet() returns null.
     */
    static ResultSet getResultSet(int SQL_ID)
    {
        ResultSet resultSet = null;

        if (Trace.METHOD.active())
            { System.out.println("\n SQL_Register: getResultSet(" + SQL_ID + ")"); }

        if (SQL_ID >= 0                &&
            SQL_ID <  SQL_ArrayEntries &&
            SQL_Array[SQL_ID] != null)
            { resultSet = SQL_Array[SQL_ID].getResultSet(); }

        if (Trace.SQL_AE_STATUS.active())
            { showSQL_Entry(SQL_ID); }

        return resultSet;
    }

    /**
     * getRowCount() obtains the row count for a given SQL ID.
     * If the SQL_ID is invalid or if the entry is empty, -1 is returned.
     */
    static int getRowCount(int SQL_ID)
    {
        int rowCount = -1;

        if (Trace.METHOD.active())
            { System.out.println("\n SQL_Register: getRowCount(" + SQL_ID + ")"); }

        if (SQL_ID >= 0                &&
            SQL_ID <  SQL_ArrayEntries &&
            SQL_Array[SQL_ID] != null)
            { rowCount = SQL_Array[SQL_ID].getRowCount(); }

        if (Trace.DETAIL.active())
            { System.out.println("\t rowCount = " + rowCount); }
    }

```

```

        return rowCount;
    }

    /**
     * getSQL_Entry() obtains a copy of the SQL array entry for a given SQL ID.
     * If the SQL_ID is invalid or if the entry is empty, null is returned.
     */
    static SQL_Entry getSQL_Entry(int SQL_ID)
    {
        SQL_Entry sqlEntry = null;

        if (Trace.METHOD.active())
            { System.out.println("\n SQL_Register: getSQL_Entry(" + SQL_ID + ")"); }

        if (SQL_ID >= 0                &&
            SQL_ID <  SQL_ArrayEntries &&
            SQL_Array[SQL_ID] != null)
            { sqlEntry = SQL_Array[SQL_ID].getSQL_Entry(); }

        if (Trace.SQL_AE_STATUS.active())
            { showSQL_Entry(SQL_ID); }

        return sqlEntry;
    }

    /**
     * getSQL_Statement() obtains the SQL statement for a given SQL ID.
     * If the SQL_ID is invalid or if the entry is empty, null is returned.
     */
    static String getSQL_Statement(int SQL_ID)
    {
        String SQL_Stmt = null;

        if (Trace.METHOD.active())
            { System.out.println("\n SQL_Register: getSQL_Statement(" +
                                SQL_ID + ")"); }

        if (SQL_ID >= 0                &&
            SQL_ID <  SQL_ArrayEntries &&
            SQL_Array[SQL_ID] != null)
            { SQL_Stmt = SQL_Array[SQL_ID].getSQL_Stmt(); }

        if (Trace.DETAIL.active())
            { System.out.println("\t SQL_Stmt = " + SQL_Stmt); }

        return SQL_Stmt;
    }

```

```

}

/**
 * handleResults() is called when the result of an action is OK or an error
 * occurred. It sets the static variables errorCode and errorMessage.
 * In case when a method is unable to return a result code (example: its
 * return value is a String), a dummy value can be returned by this method
 * after calling handleResults() with the action's real result code.
 * Then the caller of this method can use getErrorCode() for verifying the
 * real result. getErrorMessage() then returns the associated text.
 * Notes: a) handleResults() prints error messages only when the flag
 *         errorMessagePrint is true.
 *         b) handleResults() is also used by SQL_Util.
 */
static int handleResults(int code, String message)
{
    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: handleResults(" +
                           code + ", " + message + ")"); }

    errorCode    = code;
    errorMessage = message;

    if (code != SQL_Code.OK.getCode())
    {
        errorCount++;

        if (errorMessagePrint)
            { System.out.println(message); }
    }

    return code;
}

/**
 * hasInputData() calls the array entry object's method hasInputData()
 * for a given SQL ID.
 * If the SQL_ID is invalid or if the entry is empty, false is returned.
 */
static boolean hasInputData(int SQL_ID)
{
    boolean result = false;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: hasInputData(" + SQL_ID + ")"); }

    if (SQL_ID >= 0
        &&

```

```

        SQL_ID < SQL_ArrayEntries &&
        SQL_Array[SQL_ID] != null)
    { result = SQL_Array[SQL_ID].hasInputData(); }

    if (Trace.DETAIL.active())
        { System.out.println("\t result = " + result); }

    return result;
}

/**
 * hasReachedEOS() calls the array entry object's method hasReachedEOS()
 * for a given SQL ID. EOS = End Of Set = read status "E".
 * If the SQL_ID is invalid or if the entry is empty, true is returned.
 */
static boolean hasReachedEOS(int SQL_ID)
{
    boolean result = true;

    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: hasReachedEOS(" + SQL_ID + ")"); }

    if (SQL_ID >= 0 &&
        SQL_ID < SQL_ArrayEntries &&
        SQL_Array[SQL_ID] != null)
        { result = SQL_Array[SQL_ID].hasReachedEOS(); }

    if (Trace.DETAIL.active())
        { System.out.println("\t result = " + result); }

    return result;
}

/**
 * incExecCalls() increments the exec call count for a given SQL ID by 1.
 * If the SQL_ID is invalid or if the entry is empty, an error code is returned.
 */
static int incExecCalls(int SQL_ID)
{
    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: incExecCalls(" + SQL_ID + ")"); }

    if (SQL_ID < 0 || SQL_ID >= SQL_ArrayEntries)
        { return SQL_Code.ILLEGAL_SQL_ID.getCode(); }

    if (SQL_Array[SQL_ID] == null)
        { return SQL_Code.SQL_ID_NOT_IN_USE.getCode(); }
}

```

```

    SQL_Array[SQL_ID].incExecCalls();

    if (Trace.SQL_AE_UPD.active())
        { showSQL_Entry(SQL_ID); }

    return SQL_Code.OK.getCode();
}

/**
 * resetRowCount() resets the row count for a given SQL ID.
 * This call is used when a cursor is closed.
 * If the SQL_ID is invalid or if the entry is empty, an error code is returned.
 */
static int resetRowCount(int SQL_ID)
{
    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: resetRowCount(" + SQL_ID + ")"); }

    if (SQL_ID < 0 || SQL_ID >= SQL_ArrayEntries)
        { return SQL_Code.ILLEGAL_SQL_ID.getCode(); }

    if (SQL_Array[SQL_ID] == null)
        { return SQL_Code.SQL_ID_NOT_IN_USE.getCode(); }

    SQL_Array[SQL_ID].resetRowCount();

    if (Trace.SQL_AE_UPD.active())
        { showSQL_Entry(SQL_ID); }

    return SQL_Code.OK.getCode();
}

/**
 * showSQL_Array() shows the SQL statements etc in all active array entries.
 * It is used in a shutdown scenario.
 */
static void showSQL_Array()
{
    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: showSQL_Array()"); }

    if (SQL_Array != null)
    {
        for (int i = 0; i < SQL_ArrayEntries; i++)
            { showSQL_Entry(i); }
    }
}

```

```

    }

    /**
     * showSQL_Entry() shows the SQL statements etc in an active array entry.
     */
    static void showSQL_Entry(int i)
    {
        if (Trace.METHOD.active())
            { System.out.println("\n SQL_Register: showSQL_Entry(" + i + ")"); }

        if (SQL_Array[i] != null)
        {
            System.out.println(" SQL Array Entry " + i+ ":");
            System.out.println("   SQL Statement: " +
                               SQL_Array[i].getSQL_Stmt());

            if (SQL_Array[i].getPreparedStmt() != null)
                { System.out.println("   Prepared Statement handle exists"); }

            if (SQL_Array[i].getResultSet() != null)
                { System.out.println("   Result Set handle exists"); }

            System.out.println("   exec calls : " +
                               SQL_Array[i].getExecCalls());

            System.out.println("   row count : " +
                               SQL_Array[i].getRowCount());

            System.out.println("   read status: " +
                               SQL_Array[i].getReadStatus());
        }
    }

    /**
     * storeFetchResults() sets the read status after a FETCH for a given SQL ID
     * and it increments the row count by 1, if a row was read.
     * If the SQL_ID is invalid or if the entry is empty, an error code is returned.
     */
    static int storeFetchResults(int SQL_ID,
                                boolean rowFetched)
    {
        if (Trace.METHOD.active())
            { System.out.println("\n SQL_Register: storeFetchResults(" +
                                SQL_ID + ", " + rowFetched + ")"); }

        if (SQL_ID < 0 || SQL_ID >= SQL_ArrayEntries)
            { return SQL_Code.ILLEGAL_SQL_ID.getCode(); }
    }

```

```

    if (SQL_Array[SQL_ID] == null)
        { return SQL_Code.SQL_ID_NOT_IN_USE.getCode(); }

    SQL_Array[SQL_ID].setReadStatus(rowFetched);

    if (rowFetched)
        { SQL_Array[SQL_ID].incRowCount(); }

    if (Trace.SQL_AE_UPD.active())
        { showSQL_Entry(SQL_ID); }

    return SQL_Code.OK.getCode();
}

/**
 * updateRowCount() sets the row count for a given SQL ID.
 * This call is used when a non-cursor statement was executed.
 * If the SQL_ID is invalid or if the entry is empty, an error code is returned.
 */
static int updateRowCount(int SQL_ID,
                           int value)
{
    if (Trace.METHOD.active())
        { System.out.println("\n SQL_Register: updateRowCount(" +
                               SQL_ID + ", " + value + ")"); }

    if (SQL_ID < 0 || SQL_ID >= SQL_ArrayEntries)
        { return SQL_Code.ILLEGAL_SQL_ID.getCode(); }

    if (SQL_Array[SQL_ID] == null)
        { return SQL_Code.SQL_ID_NOT_IN_USE.getCode(); }

    SQL_Array[SQL_ID].setRowCount(value);

    if (Trace.SQL_AE_UPD.active())
        { showSQL_Entry(SQL_ID); }

    return SQL_Code.OK.getCode();
}

static int      getErrorCode()          { return errorCode; }

static int      getErrorCount()         { return errorCount; }

static String    getErrorMessage()     { return errorMessage; }

```

```
static boolean getPrintFlag()          { return errorMessagePrint; }

static int      getSQL_ArrayEntries() { return SQL_ArrayEntries; }

static void      setPrintFlag(boolean errMsgPrint)
{ errorMessagePrint = errMsgPrint; }
}
```