TECHNICAL REPORT

ISO/IEC TR 19075-3

First edition
2015-07-01

Information technology — Database languages — SQL Technical Reports —

Part 3: SQL Embedded in Programs using the Java™ programming language

Technologies de l'information — Langages de base de données — SQL rapports techniques —

Partie 3: SQL intégrées dans des programmes utilisant le langage de programmation de Java™

Reference number
ISO/IEC TR 19075-3:2015(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 19075-3 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

ISO/IEC TR 19075 consists of the following parts, under the general title *Information technology — Database languages — SQL Technical Reports*:

— Part 1: XQuery Regular Expression Support in SQL

— Part 2: SQL Support for Time-Related Information

— Part 3: SQL Embedded in Programs Using the Java™ Programming Language

— Part 4: SQL With Routines and Types Using the Java™ Programming Language

— Part 5: Row Pattern Recognition in SQL

NOTE 1 — The individual parts of multi-part technical reports are not necessarily published together. New editions of one or more parts may be published without publication of new editions of other parts.

Introduction

The organization of this part of ISO/IEC 19075 is as follows:

1) Clause 1, “Scope”, specifies the scope of this part of ISO/IEC 19075.
2) Clause 2, “Normative references”, identifies additional standards that, through reference in this part of ISO/IEC 19075, constitute provisions of this part of ISO/IEC 19075.
3) Clause 3, “Use of SQL in programs written in Java”, provides a tutorial on the embedding of SQL expressions and statements in programs written in the Java programming language.

Information technology — Database languages — SQL Technical Reports —

Part 3:
SQL Embedded in Programs Using the Java™ Programming Language

1 Scope

This Technical Report describes the support for the use of SQL within programs written in Java.

The Report discusses the following features of the SQL Language:

— The embedding of SQL expressions and statements in programs written in the Java programming language

(Blank page)

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 ISO and IEC standards

[ISO9075-1] ISO/IEC 9075-1:2011, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*.

[ISO9075-2] ISO/IEC 9075-2:2011, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*.

[ISO9075-10] ISO/IEC 9075-10:2008, *Information technology — Database languages — SQL — Part 10: Object Language Bindings (SQL/OLB)*.

2.2 Other international standards

[Unicode] The Unicode Consortium, *The Unicode Standard*. (Information about the latest version of the Unicode standard can be found by using the "Latest Unicode Version" link on the "Enumerated Versions of The Unicode Standard" page.)
`http://www.unicode.org/versions/enumeratedversions.html`

[Java] *The Java™ Language Specification, Third Edition*, James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, Prentice Hall, June 14, 2005, ISBN 0-321-24678-0.

[JDBC] *JDBC™ 4.0 Specification*, Final v1.0, Lance Andersen, Sun Microsystems, Inc., November 7, 2006.

[JNDI] *Java Naming and Directory Interface™*, Sun Microsystems, Inc. `http://java.sun.com/-j2se/1.5.0/docs/guide/jndi/index.html`.

[JavaBeans] *The JavaBeans™ 1.01 Specification*
`http://java.sun.com/products/javabeans/docs/spec.html`

(Blank page)

3 Use of SQL in programs written in Java

3.1 Design goals

The following items represent the major design features of [ISO9075-10].

— Provide a concise, legible mechanism for embedding SQL-statements in a program that otherwise conforms to [Java].

— Syntactic and semantic check of SQL-statements prior to program execution.

  SQL/OLB can use an implementation-defined mechanism at translate time to check embedded SQL-statements to make sure that they are syntactically and semantically correct.

— Allow the syntax and semantics of SQL-statements to be location-independent.

  The syntax and semantics of SQL-statements in an SQL/OLB program do not depend on the configuration under which SQL/OLB is running. This makes it possible to implement SQL/OLB programs that run on the client, in the SQL-server, or in a middle tier.

— Provide facilities that enable the programmer to move between the SQL/OLB and JDBC environments by sharing a single SQL-connection in both environments.

— Provide for binary portability of translated and compiled Java SQL-client applications such that they can be used transparently with multiple SQL-servers. In addition, binary portability profiles allow for customization and optimization of SQL-statements within an SQL/OLB application.

3.2 Advantages of SQL/OLB over JDBC

JDBC provides a complete, low-level SQL interface from Java to SQL-implementations. SQL/OLB is designed to fill a complementary role by providing a higher-level programming interface to SQL-implementations in such a manner as to free the programmer from the tedious and complex programming interfaces found in lower-level APIs.

The following are some major differences between the two:

— SQL/OLB source programs are smaller than equivalent JDBC programs since the translator can implicitly handle many of the tedious programming chores that dynamic interfaces require.

— SQL/OLB programs can type-check SQL code at translate time using an implementation-dependent mechanism. JDBC, being a completely dynamic API, can not.

— SQL/OLB programs allow direct embedding of Java host expressions within SQL-statements. JDBC requires a separate call statement for each bind variable and specifies the binding by position number.

— SQL/OLB enforces strong typing of query outputs and values returned and allows type checking on calls. JDBC passes values to and from SQL without compile time type checking.

— SQL/OLB provides simplified rules for invoking SQL-invoked routines. [JDBC] requires a generic call to an SQL-invoked routine, *fun*, to have the following syntax:

```
prepStmt.prepareCall("{call fun(...)}");       // For SQL-invoked procedures
prepStmt.prepareCall("{? = call fun(...)}");   // For SQL-invoked functions
```

SQL/OLB provides simplified notations:

```
#sql { CALL fun(...) };             // SQL-invoked procedure
// Declare x
...
#sql x = { VALUES(fun(...)) };      // SQL-invoked function
                                    // VALUES is an SQL construct
```

3.3 Consistency with existing embedded SQL languages

Programming languages containing embedded SQL are called *host languages*. Java differs from the traditional host languages (Ada, C, COBOL, Fortran, MUMPS (M), Pascal, PL/I) in ways that significantly affect its embedding of SQL.

— Java has automatic storage management (also known as “garbage collection”) that simplifies the management of storage for data retrieved from SQL-implementations.

— All Java types representing composite data, and data of varying sizes, have a distinguished value **`null`**, which can be used to represent the SQL **`NULL`** value. This gives Java programs an alternative to the indicator variables that are part of the interfaces to other host languages.

— Java is designed to support programs that are *automatically heterogeneously portable* (also called “super portable” or simply “downloadable”). That, along with Java's type system of classes and interfaces, enables *component software*. In particular, an SQL/OLB translator, written in Java, can call components that are specialized by SQL-implementations, in order to leverage the existing authorization, schema checking, type checking, transactional, and recovery capabilities that are traditional of SQL-implementations, and to generate code optimized for particular SQL-implementations.

— Java is designed for binary portability in heterogeneous networks, which promises to enable binary portability for applications that use SQL.

— SQL/OLB extends the traditional concept of embedded host variables by allowing generalized host expressions.

3.4 Profile customization overview

This Subclause describes how implementation-specific “customized” SQL execution control can be added to SQL/OLB applications. The SQL/OLB runtime framework uses the following interfaces:

— **SQLJ.runtime.profile.RTStatement** to execute SQL-statements.

— **SQLJ.runtime.profile.RTResultSet** to describe query results.

— **SQLJ.runtime.profile.ConnectedProfile** to create RTStatement objects corresponding to particular SQL-statements.

An implementation is able to control SQL execution by providing an implementation of the **RTStatement**, **RTResultSet**, and **ConnectedProfile** interfaces. An implementation is able to redirect control to their implementation by registering customization hooks with the application profiles.

For example, if the client connects to SQL-server **A**, then a customization that understands SQL-server **A**'s system will be used. If the client connects to SQL-server **B**, then SQL-server **B**'s customization will be used. In the absence of a connection specific customization, the default JDBC based customization will be used. Like the profile object, customization objects are serializable. This allows the customization state to be stored and restored with the profile. In this manner, an implementation-dependent deployment tool is able to load the profile, inspect and precompile the SQL-statements it contains, register an appropriate customization, and store the profile in persistent storage. Then at application runtime, the profile and the registered implementation-dependent customization will both be restored, and the customization will be used to execute the SQL-statements.

3.4.1 Profile customization process

The profile customization process is the act of registering profile customization objects with the profile(s) associated with an application. The profile customization process can be generalized to the following steps:

1) Discover the profile objects within a JAR file.

2) For each profile, deserialize the profile object from the appropriate JAR entry.

3) Create an SQL-connection with which the profile will be customized.

4) Create and register a profile customization with the profile.

5) Serialize the customized profile back to persistent storage.

6) Recreate the JAR contents using the customized serialized profiles.

Of the above steps, only step 4) is likely to change from implementation to implementation. While step 3) is implementation-dependent, it can be done using a parameterized tool and JDBC. The rest of the steps involve actions that can be performed by any generic utility without specific knowledge of the customization being performed.

The act of creating and registering a customization object with a profile (step 4 above) is abstractly defined by the Java interface **SQLJ.runtime.profile.util.ProfileCustomizer**. The intent of defining this interface is to allow SQL implementations to concentrate on writing profile customizers and customization objects (step 4 above), while tools and application implementations concentrate on writing generic tools that apply customizers to application profiles (steps 1 – 3 and 5 – 6 above).

The profile customizer interface is able to support most customization registration requirements. However, it is not required that all utilities that register customization objects with a profile implement this interface. SQL/OLB applications will be able to run and leverage all implementation-specific customization objects registered with a profile, regardless of whether or not they were registered by a profile customizer. The primary benefit of conforming to the profile customizer interface is to be able to take advantage of existing and future automated profile customization utilities that are able to load, call and manipulate profile customizers.

3.4.2 Profile customization utilities

Profile customizers can be instantiated and used by automated general-purpose profile customization utilities. An implementation might include a command-line based tool that serves as a customization utility prototype. In addition to a command line-based utility, other useful customization utilities might include:

— GUI-based IDEs used to drag-and-drop customizations into profiles.

— Tight integration of customization utilities with SQL-implementations to automatically customize the profiles loaded into the SQL-server.

— Background “SQL/OLB installer” process used as administrative tool to discover and customize SQL/OLB applications for available SQL-schemas.

NOTE 2 — Implementors are encouraged to implement utilities using these and other ideas. Making such tools publically available will greatly benefit and facilitate the SQL/OLB binary-portability effort.

3.5 Examples

3.5.1 Example of Profile generation and naming

Suppose we have the following file, **Bar.SQLJ**, which defines package **COM.foo**, and contains three <executable clause>s associated with two <connection context>s.

```
package COM.foo;
#sql context MyContext;
public class Bar
{
  public static void doSQL(MyContext ctx) throws SQLException
  {
     // 1: explicit context
     #sql [ctx] { UPDATE TAB1 SET COL1 = COL1 + 2 };
     // 2: implicit context
     #sql { INSERT INTO TAB2 VALUES(3, 'Hello there') };
     // 3: explicit context again
     #sql [ctx] { DELETE FROM TAB1 WHERE COL1 > 500 };
  }
}
```

Two profiles are created for this file; they are named **COM.foo.Bar_SJProfile0** and **COM.foo.Bar_SJProfile1**. **COM.foo.Bar_SJProfile0** contains information describing <executable clause>s 1 and 3, and is stored in a file called **Bar_SJProfile0.ser**. **Com.foo.Bar_SJProfile1** describes clause 2, and is stored in file **Bar_SJProfile1.ser**.

3.5.2 Example of a JAR manifest file

Working again with the file **Bar.SQLJ** from the last example, if the Bar application were packaged for deployment as a JAR file, the JAR's manifest can be used by SQL/OLB customization utilities to locate the application's profile files. To allow that use, the profile section of the manifest file would have the following entries:

— **Name: COM/foo/Bar_SJProfile0.ser SQLJProfile: TRUE**

— **Name: COM/foo/Bar_SJProfile1.ser SQLJProfile: TRUE**

3.5.3 Host variables

The following query contains host variable **:x** (which is the Java variable, Java field, or parameter **x** visible in the scope containing the query):

```
SELECT COL1, COL2 FROM TABLE1 WHERE :x > COL3
```

3.5.4 Host expressions

Host expressions are evaluated from left to right and can cause side effects. For example:

```
SELECT COL1, COL2 FROM TABLE1 WHERE :(x++) > COL3
```

Host expressions are always passed to and retrieved from the SQL-server using pure value semantics. For instance, in the above example, the value of **x++** is determined prior to statement execution and its determined value is the value that is passed to the SQL-server for statement execution.

```
SELECT COL1, COL2 FROM TABLE1 WHERE :(x[--i]) > COL3
```

In the above example, prior to statement execution, the value of **i** is decremented by 1 (one) and then the value of the **i**-th element of **x** is determined and passed to the SQL-server for statement execution.

Consider the following example of an SQL/PSM <assignment statement>:

```
SET :(z[i++]) = :(x[i++]) + :(y[i++])
```

Assume that **i** has an initial value of 1 (one). Host expressions are evaluated in lexical order.

Therefore, the array index used to determine the location in the array **z** is 1 (one), after which the value of **i** is incremented by 1 (one). Conseqently, the array index used to determine the location in the array **x** is 2, after which the value of **i** is incremented by 1 (one). As a result, the array index used to determine the location in the array **y** is 3, after which the value of **i** is incremented by 1 (one). The value of **i** in the Java space is now 4. The statement is then executed. After statement execution, the output value is assigned to **z[1]**.

Assignments to output host expressions are also performed in lexical order. For example, consider the following call to an SQL-invoked procedure **foo** that returns the values 2 and 3.

```
CALL foo( :OUT x, :OUT x )
```

After execution, **x** has the value 3.

3.5.5 SQL/OLB clauses

The following SQL/OLB clause is permitted to appear wherever a Java statement can legally appear and its purpose is to delete all of the rows in the table named **TAB**:

```
#sql { DELETE FROM TAB };
```

The following Java method, when invoked, inserts its arguments into an SQL table. The method body consists of an SQL/OLB executable clause containing the host expressions **x**, **y**, and **z**.

```
void m (int x, String y, float z) throws SQLException
{
   #sql { INSERT INTO TAB1 VALUES (:x, :y, :z ) };
}
```

The following method selects the address of the person whose name is specified by the input host expression **name** and then retrieves an associated address from the assumed table **PEOPLE**, with columns **NAME** and **ADDRESS**, into the output host expressions **addr**, where it is then permitted to be used, for example, in a call to `System.out.println`:

```
void print_address (String name) throws SQLException
{
   String addr;
   #sql { SELECT ADDRESS INTO :addr
          FROM PEOPLE
          WHERE :name = NAME };
}
```

3.5.6 Connection contexts

In the following SQL/OLB clause, the connection context is the value of the Java variable **myconn**.

```
#sql [myconn] { SELECT ADDRESS INTO :addr
                FROM PEOPLE
                WHERE :name = NAME } ;
```

The following illustrates an SQL/OLB connection clause that defines a connection context class named "**Inventory**":

```
#sql context Inventory;
```

3.5.7 Default connection context

If an invocation of an SQL/OLB translator indicates that the default connection context class is class **Green**, then all SQL/OLB clauses that use the default connection will be translated as if they used the explicit connection context object **Green.getDefaultContext( )**. For example, the following two SQL/OLB clauses are equivalent if the default connection context class is class **Green**:

```
#sql { UPDATE TAB SET COL = :x };
#sql [Green.getDefaultContext()] { UPDATE TAB SET COL = :x };
```

Programs are permitted to install a connection context object as the default connection by calling `setDefault-Context`. For example:

```
Green.setDefaultContext(new Green(argv[0], autoCommit));
```

`argv[0]` is assumed to contain a URL. **`autoCommit`** is a boolean flag that is **true** if auto commit mode should be on, and **false** otherwise.

3.5.8 Iterators

3.5.8.1 Positional bindings to columns

The following is an example of an iterator class declaration that binds by position. It declares an iterator class called **`ByPos`**, with two columns of types `String` and `int`.

```
#sql public iterator ByPos (String, int);
```

Assume a table PEOPLE with columns **`FULLNAME`** and **`BIRTHYEAR`**:

```
CREATE TABLE PEOPLE ( FULLNAME VARCHAR(50),
                      BIRTHYEAR NUMERIC(4,0) )
```

An iterator object of type **`ByPos`** is used in conjunction with a **`FETCH...INTO`** statement to retrieve data from table **`PEOPLE`**, as illustrated in the following example:

```
{
  ByPos    positer;              // declare iterator object
  String   name = null;
  int      year = 0;
  // populate it
  #sql positer = { SELECT FULLNAME, BIRTHYEAR
                   FROM PEOPLE };
  #sql { FETCH :positer INTO :name, :year };
  while ( !positer.endFetch() )
   {
     System.out.println(name + " was born in " + year);
     #sql { FETCH :positer INTO :name, :year };
   }
}
```

The predicate method `endFetch()` of the iterator object returns **true** if no more rows are available from the iterator (specifically, it becomes **true** following the first `FETCH` that returns no data).

The first SQL/OLB clause in the block above effectively executes its query and constructs an iterator object containing the result set returned by the query, and assigns it to variable **`positer`**. The type of the iterator object is derived from the assignment target, which is of type **`ByPos`**.

The second SQL/OLB clause in that block contains a **`FETCH...INTO`** statement. The SQL/OLB translator checks that the types of host variables in the **`INTO`** clause match the positionally corresponding types of the iterator columns. The types of the SQL columns in the query shall be convertible to the types of the positionally corresponding iterator columns, according to the SQL to Java type mapping of SQL/OLB. Those conversions are statically checked at SQL/OLB translation time if an SQL-connection to an exemplar schema is provided to the translator.

3.5.8.2 Named bindings to columns

The following is an example of an iterator class declaration that binds by name. It declares an iterator class called **`ByName`**, the named accessor methods **`fullNAME`** and **`birthYEAR`** of which correspond to the columns **`FULLNAME`** and **`BIRTHYEAR`**:

```
#sql public iterator ByName (String   fullNAME,
                             int      birthYEAR);
```

That iterator class can then be used as follows:

```
{
  ByName    namiter;          // define iterator object
  #sql namiter = { SELECT FULLNAME, BIRTHYEAR
                   FROM PEOPLE };
  String    s;
  int       i;
  // advances to next row
  while ( namiter.next() )
  {
    i = namiter.birthYEAR(); // returns column named BIRTHYEAR
    s = namiter.fullNAME();  // returns column named FULLNAME
    System.out.println(s + " was born in "+i);
  }
}
```

In this example, the first SQL/OLB clause constructs an iterator object of type **`ByName`**, as that is the type of the assignment target in that clause. That iterator has generated accessor methods `birthYEAR()` and `fullNAME()` that return the data from the result set columns with those names.

The names of the generated accessor methods are an *exact case-sensitive match* with their definitions on the iterator declaration clause. Matching a specific accessor method to a specific column name in the `SELECT` list expressions is performed using a case-insensitive match.

Two column names that differ only in the case of one or more characters shall use the SQL `AS` clause to avoid ambiguity, even if one or both of those column names are specified using delimited identifiers.

Method `next()` advances the iterator object to successive rows of the result set. It returns **true** if a next row is available and **false** if it fails to retrieve a next row because the iterator contains no more rows.

A Java compiler will detect type mismatch errors in the uses of named accessor methods. Additionally, if a connection to an exemplar schema is provided at translate time, then the SQL/OLB translator will statically check the validity of the types and names of the iterator columns against the SQL queries associated with it.

3.5.8.3 Providing names for columns of queries

If the expressions selected by a query are unnamed, or have SQL names that are not legal Java identifiers, then SQL column aliases can be used to name them. Consider a table named **`"Trouble!"`** with a column called **`"Not a legal Java identifier"`**:

```
CREATE TABLE "Trouble!" (
    "Not a legal Java identifier" VARCHAR(10),
    col2                          FLOAT )
```

The following line generates an iterator class called **`xY`**.

```
#sql iterator xY (String x, double Y);
```

The SQL/OLB clause in the following block uses column aliases to associate that column's name with an expression in the query:

```
{
  xY it;
  #sql it = { SELECT "Not a legal Java identifier" AS "x",
                     COL2 * COL2 AS Y
              FROM "Trouble!" };
  while (it.next()) { System.out.println(it.x() + it.Y());
  }
}
```

The first line declares a local variable of that iterator class.

The second line initializes that variable to contain a result set obtained from the specified query.

The **`while( )`** loop calls the named accessor methods of the iterator to obtain and print data from its rows.

3.5.9 Invoking SQL-invoked routines

An SQL/OLB executable clause, appearing as a Java statement, can call an SQL-invoked procedure by means of the SQL **`CALL`** statement. For example:

```
#sql { CALL SOME_PROC(:INOUT myarg) };
```

Support for invoking SQL-invoked routines is not required for conformance to Core SQL/OLB.

SQL-invoked procedures can have **`IN`**, **`OUT`**, or **`INOUT`** parameters. In the above case, the value of host variable **`myarg`** is changed by the execution of that clause.

An SQL/OLB executable clause can invoke an SQL-invoked function by means of the SQL **`VALUES`** construct. For example, assume an SQL-invoked function **`F`** that returns an integer. The following example illustrates an invocation of that function that then assigns its result to Java local variable **`x`**.

```
{
  int x;
  #sql x ={ VALUES ( F(34) ) };
}
```

3.5.10 Using multiple SQL/OLB contexts and connections

The following program demonstrates the use of multiple concurrent connections. It uses one user-defined context to access a table of employees through one connection and another user-defined context to access employee department information via a separate connection. By using distinct contexts, it is possible for the employee and department information to be stored on physically different SQL-servers.

```
// declare a new context class for obtaining departments
#sql context   DeptContext;
#sql context   EmpContext;
#sql iterator  Employees (String ename, int deptno);
class MultiSchema {
  void masterRoutine( String deptURL, String empURL )
       throws SQLException
  {
    // create a context for querying department info
    DeptContext deptCtx = new DeptContext( deptURL, true );
    // a second connection
    EmpContext empCtx = new EmpContext( empURL, true );
    printEmployees(deptCtx, empCtx);
    deptCtx.close();
    empCtx.close();
  }
// performs a join on deptno field of two tables
// accessed from different connections.
void printEmployees(DeptContext deptCtx, EmpContext empCtx)
     throws SQLException
  {
    // obtain the employees from the emp table connection context
    Employees emps;
    #sql [empCtx] emps = { SELECT ENAME, DEPTNO FROM EMP };
    // for each employee, obtain the department name
    // using the dept table connection context
    while (emps.next())
    {
      String dname;
      #sql [deptCtx]
      {
        SELECT DNAME INTO :dname
        FROM DEPT
        WHERE DEPTNO = :(emps.deptno())
      };
      System.out.println("employee: " + emps.ename() +
                         ", department: " + dname);
    }
    emps.close();
  }
}
```

For now, it is sufficient to note that `close()` executed against the connection contexts `DeptContext` and `EmpContext`, and against the iterator `emps`, frees the resources associated with the object against which it is invoked.

A programmer might wish to release the resources maintained by the connection context (*e.g.*, ConnectedProfile, and RTStatement objects) without actually closing the underlying SQL-connection. To this end, connection context classes also support a **close** method that takes a boolean argument indicating whether or not to close the underlying SQL-connection. Pass the constant **CLOSE_CONNECTION** if the SQL-connection should be closed, and **KEEP_CONNECTION** if it should be retained. The variant of **close** that takes no arguments is a shorthand for calling **close(CLOSE_CONNECTION)**.

As a final point, even if not using multiple SQL/OLB connection context objects, explicit manipulation of connection objects is recommended. This allows applications to avoid hidden global state (*e.g.*, Java "static variables") that would be necessarily used to implement the <SQL connection statement>. In particular, Java "applets" and other multi-threaded programs are usually coded to avoid contention of global state. Such programs should store connection objects in local variables and use them explicitly in SQL/OLB clauses.

3.5.11 SQL execution control and status

An execution context can be supplied explicitly as an argument to each SQL-statement.

```
ExecutionContext execCtx = new ExecutionContext();
   #sql [execCtx] { DELETE FROM EMP WHERE SAL > 10000 };
```

If explicit execution context objects are used, each SQL-statement can be executed using a different execution context object. If an explicit connection context object is also being used, both are available to be queried and modified during execution of the SQL-statement.

```
#sql [connCtx, execCtx] { DELETE FROM EMP
                          WHERE SAL > 10000 };
```

If an execution context object is not supplied explicitly as an argument to an SQL-statement, then a default execution context object is used implicitly. The default execution context object for a particular SQL-statement is obtained via the `getExecutionContext()` method of the connection context object used in the operation. For example:

```
#sql [connCtx] { DELETE FROM EMP WHERE SAL > 10000 };
```

The preceding example uses the execution context object associated with the connection context object given by `connCtx`. If neither a connection context object nor an execution context object is explicitly supplied, then the execution context object associated with the default connection context object is used.

The use of an explicit execution context object overrides the execution context boject associated with the connection context object, referenced explicitly or implicitly by an SQL clause.

The following code demonstrates the use of some `ExecutionContext` methods.

```
{
  ExecutionContext execCtx = new ExecutionContext();
  // Wait only 3 seconds for operations to complete
  execCtx.setQueryTimeout(3);
  try {
    // delete using explicit execution context
    // if operation takes longer than 3 seconds,
    // SQLException is thrown
    #sql [execCtx] { DELETE FROM EMP WHERE SAL > 10000 };
    System.out.println
      ("removed " + execCtx.getUpdateCount() + " employees");
  }
  catch(SQLException e) {
    // Assume a timeout occurred
    System.out.println("SQLException has occurred with" + " exception " + e );
  }
}
```

3.5.12 Multiple java.sql.ResultSet objects from SQL-invoked procedure calls

If execution of an SQL-statement produces multiple results, the resources are not released until all results have been processed using **getNextResultSet**. Accordingly, if an SQL-invoked procedure might return side-channel result sets, then the calling program should process all results using **getNextResultSet** until null is returned. Further, if one or more side-channel result sets have been left open, they should be explicitly closed, because their associated resources cannot be released until they are closed.

If the invocation of an SQL-invoked procedure does not produce side-channel result sets, then there is no need to call **getNextResultSet**. All resources are automatically reclaimed as soon as the CALL execution completes.

The following code snippet demonstrates how multiple results are processed. The example assumes that an SQL-invoked procedure named "**`multi_results`**" exists and produces one or more side-channel result sets when executed.

```
#sql [execCtx] { CALL MULTI_RESULTS() };
ResultSet rs;
while ((rs = execCtx.getNextResultSet()) != null)
  { // process result set
    ...
    rs.close();
  }
```

The following snippet demonstrates how multiple result sets can be processed simultaneously. The example assumes an SQL-invoked procedure named "**`multi_results`**" exists and produces between 2 and 10 side-channel result sets when executed.

```
#sql [execCtx] { CALL MULTI_RESULTS() };
ResultSet[] rsets = new ResultSet[10];
ResultSet rs;
int rsCounter = 0;
// access the ResultSets
while ((rs = execCtx.getNextResultSet(Statement.KEEP_CURRENT_RESULT)) != null)
  { rsets[rsCounter++] = rs;
  }
// process ...
// close
for (int ii=0; ii < rsCounter; ii++)
```

```
  { rsets[ii].close();
  }
```

3.5.13 Creating an SQL/OLB iterator object from a java.sql.ResultSet object

An SQL/OLB iterator object can be created from a `java.sql.ResultSet` object with the <iterator conversion clause>. Once an iterator object has been created this way, portable code should not issue any further calls to the `java.sql.ResultSet` object, because the result of doing so is implementation-defined.

As an example, assume we have the following iterator declaration:

```
#sql iterator Employees ( String ename, double sal ) ;
```

The following method uses JDBC to perform a dynamic query and uses an instance of the above iterator declaration to view the results. It illustrates the use of an iterator conversion statement.

```
public void listEarnings(Connection conn, String whereClause)
    throws SQLException
{
  // prepare a java.sql.Statement object to execute a dynamic query
  PreparedStatement stmt = conn.prepareStatement();
  String query = "SELECT ename, sal FROM emp WHERE ";
  query += whereClause;
  ResultSet rs = stmt.executeQuery(query);
  Employees emps;
  // Use the iterator conversion statement to create a
  // SQL/OLB iterator from a java.sql.ResultSet object
  #sql emps = { CAST :rs };
  while (emps.next()) {
       System.out.println(emps.ename() +
                        " earns " + emps.sal());
  }
  emps.close(); // closing emps also closes rs
  stmt.close();
}
```

3.5.14 Obtaining a java.sql.ResultSet object from an iterator object

Every SQL/OLB iterator object, whether typed or untyped, has a **getResultSet** method that returns a `java.sql.ResultSet` object representation of its data. For portable code, the `getResultSet()` method should be invoked before the first `next()` method invocation on the iterator object. And, once the `java.sql.ResultSet` object has been produced, all operations to fetch data, or update the ResultSet, should be through that `java.sql.ResultSet` object; doing so avoids potential problems due to the implementation-defined nature of the synchronization (if any) between the iterator object and its `java.sql.ResultSet` object.

As an example, the following method uses a weakly typed iterator to hold to results of an SQL/OLB query and then process them using a `java.sql.ResultSet` object:

```
public void showEmployeeNames() throws SQLException
{
   SQLJ.runtime.ResultSetIterator iter;
   #sql iter = { SELECT ename FROM emp };
   ResultSet rs = iter.getResultSet();
   while (rs.next()) {
     System.out.println("employee name: " + rs.getString(1));
   }
   iter.close(); // close the iterator, not the result set
}
```

3.5.15 Working with user-defined types

NOTE 3 — Readers of this Subclause should note that some of the examples herein depend on optional features of the SQL language and of the JDBC specification. As a consequence, not all examples are guaranteed to work on all SQL/OLB implementations. They are provided for educational purposes only.

Consider the following type mapping information to be specified in file `addrpckg/address-map.properties`:

```
# file: addressmap.properties
class.addrpckg.Address = STRUCT ADDRESS
class.addrpckg.BusinessAddress = STRUCT BUSINESS
class.addrpckg.HomeAddress = STRUCT HOME
class.addrpckg.ZipCode = DISTINCT ZIPCODE
```

The first entry defines that the Java class Address in package `addrpckg` corresponds to the SQL user-defined type ADDRESS. It further indicates that the SQL type is a structured type.

The type map specified in the above file can be attached to a connection context class as part of the connection context declaration in the following way:

```
#sql context Ctx with (typeMap = "addrpckg.addressmap")
```

The SQL/OLB translator and runtime will interpret the specified type map `"addrpckg.addressmap"` as a Java resource bundle family name, and look for an appropriate properties or class file using the Java class path. This means that the type map can easily be packaged with the rest of the SQL/OLB application or application module.

It is now possible to define host variables or iterators based on the Java types that participate in the type map:

```
#sql public iterator ByPos (String, int, addrpckg.Address);
```

Assume a table **PEOPLE** with columns **FULLNAME**, **BIRTHYEAR**, and **ADDRESS**:

```
CREATE TABLE PEOPLE (
    FULLNAME      CHARACTER VARYING(50),
    BIRTHYEAR     NUMERIC(4,0),
    ADDR          ADDRESS )
```

An iterator object of type **ByPos** is used in conjunction with a **FETCH...INTO** statement to retrieve data, including instances of the user-defined type **ADDRESS** from table **PEOPLE**, as illustrated in the following example:

```
{
  ByPos positer;        // declare iterator object
```