TECHNICAL REPORT

ISO/IEC TR 19075-4

First edition
2015-07-01

Information technology — Database languages — SQL Technical Reports —

Part 4: SQL with Routines and types using the Java™ programming language

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

Partie 4: SQL avec des Routines et Types Utilisant le Langage de Programmation de Java™

Reference number
ISO/IEC TR 19075-4:2015(E)

Contents

Page

Foreword . . . v
Introduction . . . vi
1 Scope . . . 1
2 Normative references . . . 3
2.1 ISO and IEC standards . . . 3
2.2 Other international standards . . . 3
3 Routines tutorial . . . 5
3.1 Technical components . . . 5
3.2 Overview . . . 5
3.3 Example Java methods: region and correctStates . . . 6
3.4 Installing region and correctStates in SQL . . . 7
3.5 Defining SQL names for region and correctStates . . . 8
3.6 A Java method with output parameters: bestTwoEmps . . . 9
3.7 A CREATE PROCEDURE best2 for bestTwoEmps . . . 11
3.8 Calling the best2 procedure . . . 12
3.9 A Java method returning a result set: orderedEmps . . . 12
3.10 A CREATE PROCEDURE rankedEmps for orderedEmps . . . 14
3.11 Calling the rankedEmps procedure . . . 15
3.12 Overloading Java method names and SQL names . . . 15
3.13 Java main methods . . . 17
3.14 Java method signatures in the CREATE statements . . . 18
3.15 Null argument values and the RETURNS NULL clause . . . 19
3.16 Static variables . . . 21
3.17 Dropping SQL names of Java methods . . . 22
3.18 Removing Java classes from SQL . . . 22
3.19 Replacing Java classes in SQL . . . 23
3.20 Visibility . . . 24
3.21 Exceptions . . . 24
3.22 Deployment descriptors . . . 25
3.23 Paths . . . 28
3.24 Privileges . . . 30
3.25 Information Schema . . . 30
4 Types tutorial . . . 33
4.1 Overview . . . 33
4.2 Example Java classes . . . 33

4.3 Installing Address and Address2Line in an SQL system. 35
4.4 CREATE TYPE for Address and Address2Line. 36
4.5 Multiple SQL types for a single Java class. 37
4.6 Collapsing subclasses. 38
4.7 GRANT and REVOKE statements for data types. 40
4.8 Deployment descriptors for classes. 40
4.9 Using Java classes as data types. 41
4.10 SELECT, INSERT, and UPDATE. 42
4.11 Referencing Java fields and methods in SQL. 43
4.12 Extended visibility rules. 43
4.13 Logical representation of Java instances in SQL. 44
4.14 Static methods. 45
4.15 Static fields. 46
4.16 Instance-update methods. 46
4.17 Subtypes in SQL/JRT data. 48
4.18 References to fields and methods of null instances. 49
4.19 Ordering of SQL/JRT data. 51
Index. 53

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 19075-4 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, “Routines tutorial”, provides a tutorial on the use of routines written in the Java programming language within SQL expressions and statements.
4) Clause 4, “Types tutorial”, provides a tutorial on the use of user-defined types written in the Java programming language within SQL expressions and statements.

Information technology — Database languages — SQL Technical Reports —

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

1 Scope

This Technical Report provides a tutorial of SQL Routines and Types Using the Java™ Programming Language.

The Report discusses the following features of the SQL Language:

— The use of routines written in the Java programming language within SQL expressions and statements.

— the use of user-defined types written in the Java programming language within SQL expressions and statements.

(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)*.

[ISO9075-11] ISO/IEC 9075-11:2011, *Information technology — Database languages — SQL — Part 11: Information and Definition Schemas (SQL/Schemata)*.

[ISO9075-13] ISO/IEC 9075-13:2008, *Information technology — Database languages — SQL — Part 13: SQL Routines and Types Using the Java™ Programming Language (SQL/JRT)*.

2.2 Other international standards

[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.

[JVM] *The Java™ Virtual Machine Specification, Second Edition*, Tim Lindholm and Frank Yellin, Addison-Wesley, 1999, ISBN 0-201-43294-3, as amended by *Clarifications and Amendments to the Java Virtual Machine Specification*, `http://java.sun.com/docs/books/jvms/second_edition/jvms-clarify.html`.

[J2SE] *Java™ Platform Standard Edition 6 API Specification*, `http://java.sun.com/javase/6/-docs/api/index.html`.

[Serialization] *Java™ Object Serialization Specification*, version 6.0 `http://java.sun.com/javase/-6/docs/platform/serialization/spec/serialTOC.html`.

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

(Blank page)

3 Routines tutorial

3.1 Technical components

Part 13 of ISO/IEC 9075 includes the following:

— New built-in procedures.

- `SQLJ.INSTALL_JAR` — to load a set of Java classes in an SQL system.
- `SQLJ.REPLACE_JAR` — to supersede a set of Java classes in an SQL system.
- `SQLJ.REMOVE_JAR` — to delete a previously installed set of Java classes.
- `SQLJ.ALTER_JAVA_PATH` — to specify a path for name resolution within Java classes.

— New built-in schema.

The built-in schema named `SQLJ` is assumed to be in all catalogs of an SQL system that implements the SQL/JRT facility, and to contain all of the built-in procedures of the SQL/JRT facility.

— Extensions of the following SQL statements:

- CREATE PROCEDURE/FUNCTION — to specify an SQL name for a Java method.
- DROP PROCEDURE/FUNCTION — to delete the SQL name of a Java method.
- CREATE TYPE — to specify an SQL name for a Java class.
- DROP TYPE — to delete the SQL name of a Java class.
- GRANT — to grant the USAGE privilege on Java JARs.
- REVOKE — to revoke the USAGE privilege on Java JARs.

— Conventions for returning values of OUT and INOUT parameters, and for returning SQL result sets.

— New forms of reference: Qualified references to the fields and methods of columns whose data types are defined on Java classes.

— Additional views and columns in the Information Schema.

3.2 Overview

This tutorial shows a series of example Java classes, indicates how they can be installed, and shows how their static, public methods can be referenced with SQL/JRT facilities in an SQL-environment.

The example Java methods assume an SQL table named EMPS, with the following columns:

— NAME — the employee's name.

— ID — the employee's identification.

— STATE — the state in which the employee is located.

— SALES — the amount of the employee's sales.

— JOBCODE — the job code of the employee.

The table definition is:

```
CREATE TABLE emps (
    name    VARCHAR(50),
    id      CHARACTER(5),
    state   CHARACTER(20),
    sales   DECIMAL (6,2),
    jobcode INTEGER );
```

The example classes and methods are:

— `Routines1.region` — A Java method that maps a US state code to a region number. This method doesn't use SQL internally.

— `Routines1.correctStates` — A Java method that performs an SQL UPDATE statement to correct the spelling of *state* codes. The old and new spellings are specified by input-mode parameters.

— `Routines2.bestTwoEmps` — A Java method that determines the top two employees by their sales, and returns the columns of those two employee rows as output-mode parameter values. This method creates an SQL result set and processes it internally.

— `Routines3.orderedEmps` — A Java method that creates an SQL result set consisting of selected employee rows ordered by the sales column, and returns that result set to the client.

— `Over1.isOdd` and `Over2.isOdd` — Contrived Java methods to illustrate overloading rules.

— `Routines4.job1` and `Routines5.job2` — Java methods that return a string value corresponding to an integer jobcode value. These methods illustrate the treatment of null arguments.

— `Routines6.job3` — Another Java method that returns a string value corresponding to an integer jobcode value. This method illustrates the behavior of static Java variables.

Unless otherwise noted, the methods that invoke SQL use JDBC. One of the methods is shown in both aversion using JDBC and a version using SQL/OLB. The others could also be coded with SQL/OLB.

It is assumed that the import statements `import java.sql.*;` and `java.math.*;` have been included in all classes.

3.3 Example Java methods: region and correctStates

This clause shows an example Java class, `Routines1`, with two simple methods.

— The `int`-valued static method `region` categorizes 9 states into 3 geographic regions, returning an integer indicating the region associated with a valid state or throwing an exception for invalid states. This method will be called as a function in SQL.

— The void method correctStates updates the EMPS table to correct spelling errors in the state column. This method will be called as a procedure in SQL.

```
public class Routines1 {
  //An int method that will be called as a function
  public static int region(String s) throws SQLException {
    if (s.equals("MN") || s.equals("VT") || s.equals("NH")) return 1;
      else if (s.equals("FL") || s.equals("GA") || s.equals("AL")) return 2;
      else if (s.equals("CA") || s.equals("AZ") || s.equals("NV")) return 3;
      else throw new SQLException("Invalid state code", "38001");
  }
  //A void method that will be called as a stored procedure
  public static void correctStates (String oldSpelling, String newSpelling)
      throws SQLException {
    Connection conn = DriverManager.getConnection ("jdbc:default:connection");
    PreparedStatement stmt = conn.prepareStatement
        ("UPDATE emps SET state = ? WHERE state = ?");
    stmt.setString(1, newSpelling);
    stmt.setString(2, oldSpelling);
    stmt.executeUpdate();
    stmt.close();
    conn.close();
    return;
  }
}
```

3.4 Installing region and correctStates in SQL

The source code for Java classes such as Routines1 will normally be in one or more Java files (*i.e.*, files with file type “java”). When you compile them (using the javac compile command), the resulting code will be in one or more class files (*i.e.*, files with file type “class”). You then typically collect a set of class files into a Java JAR, which is a ZIP-coded collection of files.

To use Java classes in SQL, you load a JAR containing them into the SQL system by calling the SQL SQLJ.INSTALL_JAR procedure. The SQLJ.INSTALL_JAR procedure is a new built-in SQL procedure that makes the collection of Java classes contained in a specified JAR available for use in the current SQL catalog. For example, assume that you have assembled the above Routines1 class into a JAR with local file name “~/classes/Routines1.jar”:

```
SQLJ.INSTALL_JAR('file:~/classes/Routines1.jar', 'routines1_jar', 0)
```

— The first parameter of the SQLJ.INSTALL_JAR procedure is a character string specifying the URL of the given JAR. This parameter is never folded to upper case.

— The second parameter of the SQLJ.INSTALL_JAR procedure is a character string that will be used as the name of the JAR in the SQL system. The JAR name is an SQL qualified name, and follows SQL conventions for qualified names.

The JAR name that you specify as the second parameter of the SQLJ.INSTALL_JAR procedure identifies the JAR within the SQL system. That is, the JAR name that you specify is used only in SQL, and has nothing to do with the contents of the JAR itself. The JAR name is used in the following contexts, which are described in later clauses:

- As a parameter of the `SQLJ.REMOVE_JAR` and `SQLJ.REPLACE_JAR` procedures.
- As a qualifier of Java class names in SQL CREATE PROCEDURE/FUNCTION statements.
- As an operand of the extended SQL GRANT and REVOKE statements.
- As a qualifier of Java class names in SQL CREATE TYPE statements.

The JAR name may also be used in follow-on facilities for downloading JARs from the SQL system.

— JARs can also contain *deployment descriptors*, which specify implicit actions to be taken by the `SQLJ.INSTALL_JAR` and `SQLJ.REMOVE_JAR` procedures. The third parameter of the `SQLJ.INSTALL_JAR` procedureis an integer that specifies whether you do or do not (indicated by non-zero or zero values, respectively) want the `SQLJ.INSTALL_JAR` procedure to execute the actions specified by a deployment descriptor in the JAR.

The name of the INSTALL_JAR procedure is qualified with the schema name SQLJ. All built-in procedures of the SQL/JRT facility are defined to be contained in that built-in schema. The SQLJ schema is assumed to be present in each catalog of an SQL system that implements the SQL/JRT facility.

The first two parameters of `SQLJ.INSTALL_JAR` are character strings, so if you specify them as literals, you will use single quotes, not the double quotes used for SQL delimited identifiers.

The actions of the `SQLJ.INSTALL_JAR` procedure are as follows:

— Obtain the JAR designated by the first parameter.

— Extract the class files that it contains and install them into the current SQL schema.

— Retain a copy of the JAR itself, and associate it with the value of the second parameter.

— If the third parameter is non-zero, then perform the actions specified by the deployment descriptor of the JAR.

After you install a JAR with the `SQLJ.INSTALL_JAR` procedure, you can reference the static methods of the classes contained in that JAR in the CREATE PROCEDURE/FUNCTION statement, as we will describe in the next Subclause.

3.5 Defining SQL names for region and correctStates

Before you can call a Java method in SQL, you shall define an SQL name for it. You do this with new options on the SQL CREATE PROCEDURE/FUNCTION statement. For example:

```
CREATE PROCEDURE correct_states(old CHARACTER(20), new CHARACTER(20))
  MODIFIES SQL DATA
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines1_jar:Routines1.correctStates';
CREATE FUNCTION region_of(state CHARACTER(20)) RETURNS INTEGER
  NO SQL
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines1_jar:Routines1.region';
```

The CREATE PROCEDURE and CREATE FUNCTION statements specify SQL names and Java method signatures for the Java methods specified in the EXTERNAL NAME clauses. The format of the method names in the external name clause consists of the JAR name that was specified in the `SQLJ.INSTALL_JAR` procedure followed by the Java method name, fully qualified with the package name(s) (if any) and class name.

The CREATE PROCEDURE for `correct_states` specifies the clause MODIFIES SQL DATA. This indicates that the specified Java method modifies (via INSERT, UPDATE, or DELETE) data in SQL tables. The CREATE FUNCTION for `region_of` specifies NO SQL. This indicates that the specified Java method performs no SQL operations.

Other clauses that you can specify are READS SQL DATA, which indicates that the specified Java method reads (through SELECT) data in SQL tables, but does not modify SQL data, and CONTAINS SQL, which indicates that the specified method invokes SQL operations, but neither reads nor modifies SQL data. The alternative CONTAINS SQL is the default.

You use the SQL procedure and function names that you define with such CREATE PROCEDURE/FUNCTION statements as normal SQL procedure and function names:

```
SELECT name, region_of(state) AS region
FROM emps
WHERE region_of(state) = 3;
CALL correct_states ('GEO', 'GA');
```

You can define multiple SQL names for the same Java method:

```
CREATE PROCEDURE state_correction(old CHARACTER(20), new CHARACTER(20))
  MODIFIES SQL DATA
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines1_jar:Routines1.correctStates';
CREATE FUNCTION state_region(state CHARACTER(20)) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines1_jar:Routines1.region';
```

The various SQL function and procedure names for a Java method can be used equivalently:

```
SELECT name, state_region(state) AS region
FROM emps
WHERE region_of(state) = 2;
CALL state_correction ('ORE', 'OR');
```

The SQL names are normal 3-part SQL names, and the first two parts of the 3-part names are defaulted as defined in SQL for CREATE PROCEDURE and CREATE FUNCTION statements.

There are other considerations for the CREATE PROCEDURE/FUNCTION statement, dealing with parameter data types, overloaded names, and privileges, which we will discuss in later Subclauses.

3.6 A Java method with output parameters: bestTwoEmps

The parameters of the `region` and `correctStates` methods are all input-only parameters. This is the normal Java parameter convention.

SQL procedures also support parameters with mode OUT and INOUT. The Java language does not directly have a notion of output parameters. SQL/JRT therefore uses arrays to return output values for parameters of Java methods. That is, if you want an `Integer` parameter to return a value to the caller, you specify the type of that parameter to be `Integer[ ]`, *i.e.* an array of `Integer`. Such an array will contain only one element:

the input value of the parameter is contained in that element when the method is called, and the method sets the value of that element to the desired output value.

As we will see in the following clauses, this use of arrays for output parameters in the Java methods is visible only to the Java method. When you call such a method as an SQL procedure, you supply normal scalar data items as parameters. The SQL system performs the mapping between those scalar data items and Java arrays implicitly.

The following Java method illustrates the way that you code output parameters in Java. This method, `bestTwoEmps`, returns the `name`, `id`, `region`, and `sales` of the two employees that have the highest `sales` in the regions with numbers higher than a parameter value. That is, each of the first 8 parameters is an OUT parameter, and is therefore declared to be an array of the given type.

The following version of the `bestTwoEmps` method uses SQL/OLB for statements that access SQL:

```
public class Routines2 {
  public static void bestTwoEmps (
      String[ ] n1, String[ ] id1, int[ ] r1, BigDecimal[ ] s1,
      String[ ] n2, String[ ] id2, int[ ] r2, BigDecimal[ ] s2,
      int regionParm) throws SQLException {
    #sql iterator ByNames (String name, String id, int region, BigDecimal sales);
    n1[0]= "****"; n2[0]= "****"; id1[0]= ""; id2[0]= "";
    r1[0]=0; r2[0]=0; s1[0]= new BigDecimal(0); s2[0]= new BigDecimal(0);
    ByNames r = null;
    try {
      #sql r = {SELECT name, id, region_of(state) AS region, sales
                FROM emp
                WHERE region_of(state) > :regionParm
                  AND sales IS NOT NULL
                ORDER BY sales DESC};
      if (r.next()) {
        n1[0] = r.name();
        id1[0] = r.id();
        r1[0] = r.region();
        s1[0] = r.sales();
      }
      else return;
      if (r.next()) {
        n2[0] = r.name();
        id2[0] = r.id();
        r2[0] = r.region();
        s2[0] = r.sales();
      }
      else return;
    } finally {
      if (r != null) r.close();
      }
  }
}
```

Note that since the above Java method uses SQL/OLB for SQL operations, it does not have to explicitly obtain a connection to the SQL system. By default, SQL/OLB executes any SQL contained in a routine in the context of the SQL statement invoking that routine.

For comparison, here's a version of the `bestTwoEmps` method using JDBC instead of SQL/OLB:

```
public class Routines2 {
  public static void bestTwoEmps (
      String[ ] n1, String[ ] id1, int[ ] r1, BigDecimal[ ] s1,
      String[ ] n2, String[ ] id2, int[ ] r2, BigDecimal[ ] s2,
      int regionParm) throws SQLException {
    n1[0]= "****"; n2[0]= "****"; id1[0]= ""; id2[0]= "";
    r1[0]=0; r2[0]=0; s1[0]= new BigDecimal(0); s2[0]= new BigDecimal(0);
    java.sql.PreparedStatement stmt = null;
    try {
      Connection conn = DriverManager.getConnection
          ("jdbc:default:connection");
      stmt.conn.prepareStatement
          ("SELECT name, id, region_of(state) AS region, sales
           FROM emp
           WHERE region_of(state) > ?
             AND sales IS NOT NULL
           ORDER BY sales DESC");
      stmt.setInt(1, regionParm)
      ResultSet r = stmt.executeQuery();
      if (r.next()) {
        n1[0] = r.getString("name");
        id1[0] = r.getString("id");
        r1[0] = r.getInt("region");
        s1[0] = r.getBigDecimal("sales");
      }
      else return;
      if (r.next()) {
        n2[0] = r.getString("name");
        id2[0] = r.getString("id");
        r2[0] = r.getInt("region");
        s2[0] = r.getBigDecimal("sales");
      }
      else return;
    } finally {
        if (stmt != null) stmt.close()
      };
  }
}
```

3.7 A CREATE PROCEDURE best2 for bestTwoEmps

Assume that you call the `SQLJ.INSTALL_JAR` procedure for a JAR containing the `Routines2` class with the `bestTwoEmps` method:

```
SQLJ.INSTALL_JAR ('file:~/classes/Routines2.jar', 'routines2_jar', 0)
```

As indicated previously, in order to call a method such as `bestTwoEmps` in SQL, you shall define an SQL name for it, using the CREATE PROCEDURE statement:

```
CREATE PROCEDURE best2 (
    OUT n1 CHARACTER VARYING(50), OUT id1 CHARACTER VARYING(5), OUT r1 INTEGER,
    OUT s1 DECIMAL(6,2),
    OUT n2 CHARACTER VARYING(50), OUT id2 CHARACTER VARYING(5), OUT r2 INTEGER,
    OUT s2 DECIMAL(6,2), region INTEGER)
  READS SQL DATA
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines2_jar:Routines2.bestTwoEmps';
```

For parameters that are specified to be OUT or INOUT, the corresponding Java parameter shall be an array of the corresponding data type.

3.8 Calling the best2 procedure

After you have installed the `Routines2` class in an SQL system and executed the CREATE PROCEDURE for `best2`, you can call the `bestTwoEmps` method as if it were an SQL stored procedure, with normal conventions for OUT parameters. Such a call could be written with embedded SQL, CLI, ODBC, or JDBC. The following is an example of such a call using JDBC:

```
java.sql.CallableStatement stmt = conn.prepareCall(
    "{call best2(?,?,?,?,?,?,?,?,?)}");
  stmt.registerOutParameter(1, java.sql.Types.STRING);
  stmt.registerOutParameter(2, java.sql.Types.STRING);
  stmt.registerOutParameter(3, java.sql.Types.INTEGER);
  stmt.registerOutParameter(4, java.sql.Types.DECIMAL);
  stmt.registerOutParameter(5, java.sql.Types.STRING);
  stmt.registerOutParameter(6, java.sql.Types.STRING);
  stmt.registerOutParameter(7, java.sql.Types.INTEGER);
  stmt.registerOutParameter(8, java.sql.Types.DECIMAL);
  stmt.setInt(9, 3);
  stmt.executeUpdate();
  String n1 = stmt.getString(1);
  String id1 = stmt.getString(2);
  int r1 = stmt.getInt(3);
  BigDecimal s1 = stmt.getBigDecimal(4);
  String n2 = stmt.getString(5);
  String id2 = stmt.getString(6);
  int r2 = stmt.getInt(7);
  BigDecimal s2 = stmt.getBigDecimal(8);
```

3.9 A Java method returning a result set: orderedEmps

SQL stored procedures can generate SQL result sets as their output. An SQL result set (as defined in JDBC and SQL) is an ordered sequence of SQL rows. SQL result sets aren't processed as normal function result values, but are instead bound to caller-specified iterators or cursors, which are subsequently used to process the rows of the result set.

The following Java method, `orderedEmps`, generates an SQL result setand then returns that result set to the client. Note that the `orderedEmps` method internally generates the result set in the same way as the `bestTwoEmps` method. However, the `bestTwoEmps` method processes the result set within the `bestTwoEmps` method itself, whereas this `orderedEmps` method returns the result set to the client as an SQL result set.

To write a Java method that returns a result set to the client, you specify the method to have an additional parameter that is a single-element array of either the Java `ResultSet` class or a class generated by an SQL/OLB iterator declaration (“`#sql iterator...`”).

The following version of the `orderedEmps` procedure uses SQL/OLB to access the SQL server, and returns the result set as an SQL/OLB iterator, `SalesReport`:

```
// #sql public iterator SalesReport (String name, int region, BigDecimal sales);
  public class Routines3 {
    public static void orderedEmps (int regionParm, SalesReport[ ] rs)
        throws SQLException {
      #sql rs[0] = { SELECT name, region_of(state) AS region, sales
                     FROM emp
                     WHERE region_of(state) > :regionParm
                       AND sales IS NOT NULL
                     ORDER BY sales DESC };
      return;
    }
  }
```

The `SalesReport` iterator class could be a public static inner class of `Routines3`. However, the above example presumes existence of an "`*.sqlj`" file, named `SalesReport.sqlj`, in the same package as `Routines3`, containing the public definition of the `SalesReport` iterator. That is, `SalesReport.sqlj` contains:

```
#sql public iterator SalesReport (String name, int region, BigDecimal sales);
```

Assume, for this example, that both class `Routines3` and the iterator `SalesReport` are defined in a package named `classes`.

For comparison, the following shows `orderedEmps` written using JDBC instead of SQL/OLB.

```
public class Routines3 {
  public static void orderedEmps(int regionParm, ResultSet[ ] rs)
      throws SQLException {
    Connection conn = DriverManager.getConnection ("jdbc:default:connection");
    java.sql.PreparedStatement stmt = conn.prepareStatement
        ("SELECT name, region_of(state) AS region, sales
          FROM emp WHERE region_of(state) > ?
           AND sales IS NOT NULL
          ORDER BY sales DESC");
    stmt.setInt (1, regionParm);
    rs[0] = stmt.executeQuery();
    return;
  }
}
```

The method sets the first element of the `ResultSet[]` parameter to reference the Java `ResultSet` containing the SQL result set to be returned. The method does *not* close either the returned `ResultSet` object *or* the Java statement object that generated the result set. The SQL system will implicitly close both of those objects.

You can call a method such as `orderedEmps` in Java in the normal manner, supplying explicit arguments for both parameters. You can also call it in SQL, as a stored procedure that generates a result set to be processed in the SQL manner. We illustrate how this is done in the following two clauses.

Each of the above `orderedEmps` examples has a single result set parameter, `rs`, in which you can only return a single result set. You can also specify multiple result set parameters.

Note that, in comparison to the prior examples of `bestTwoEmps`, there is no `try...finally` block to close the SQL/OLB iterator or `ResultSet`, `rs[0]`, or the JDBC PreparedStatement, `stmt`. For a result set to be returned from a stored procedure it shall not be explicitly closed, which means, in the case of JDBC, that the statement executed to generate the result set also shall not be explicitly closed.

3.10 A CREATE PROCEDURE rankedEmps for orderedEmps

Assume that you call the `SQLJ.INSTALL_JAR` procedure for a JAR containing the `Routines3` class with the `orderedEmps` method:

```
SQLJ.INSTALL_JAR( 'file:~/classes/Routines3.jar', 'routines3_jar', 0)
```

As with previous methods, you will now need to define an SQL name for the `orderedEmps` method before you can call it as an SQL procedure. As above, you will do this with a CREATE PROCEDURE statement that specifies an EXTERNAL...LANGUAGE JAVA clause to reference the `orderedEmps` method. The following is an example CREATE PROCEDURE...DYNAMIC RESULT SETS for the above `orderedEmps` method:

```
CREATE PROCEDURE rankedEmps (region INTEGER)
    READS SQL DATA
    DYNAMIC RESULT SETS 1
    LANGUAGE JAVA PARAMETER STYLE JAVA
    EXTERNAL NAME 'routines3_jar:classes.Routines3.orderedEmps';
```

A CREATE PROCEDURE statement for a Java method that generates SQL result sets has the following characteristics:

— The DYNAMIC RESULT SETS clause indicates that the procedure generates one or more result sets. The integer specified in the DYNAMIC RESULT SETS clause is the maximum number of result sets that the procedure will generate. If an execution generates more than this number of result sets, a warning will be issued, and only the specified number of result sets will be returned.

— The SQL signature specifies only the parameters that the caller explicitly supplies.

— The specified Java method actually has one or more additional, trailing parameters, whose data types shall be a Java array of either `java.sql.ResultSet` or an implementation of `sqlj.runtime.ResultSetIterator`.

The above CREATE PROCEDURE statement could be used to reference either an SQL/OLB-based or JDBC-based version of `Routines3.orderedEmps`. When it is necessary to choose a particular implementation, the Java method signature of the desired Java method shall be explicitly stated. For the SQL/OLB-based `orderedEmps`:

```
CREATE PROCEDURE rankedEmps (region INTEGER)
    READS SQL DATA
    DYNAMIC RESULT SETS 1
    LANGUAGE JAVA PARAMETER STYLE JAVA
    EXTERNAL NAME
        'routines3_jar:classes.Routines3.orderedEmps(int, classes.SalesReport[])';
```

And, for the JDBC-based orderedEmps:

```
CREATE PROCEDURE rankedEmps (region INTEGER)
    READS SQL DATA
    DYNAMIC RESULT SETS 1
    LANGUAGE JAVA PARAMETER STYLE JAVA
    EXTERNAL NAME
        'routines3_jar:classes.Routines3.orderedEmps(int, java.sql.ResultSet[])';
```

The only difference in the above CREATE PROCEDURE `rankedEmps` statements is in the Java method signature's description of the dynamic result set returned. In both cases, a fully qualified class name is provided for, respectively, the SQL/OLB iterator (remember that `SalesReport` is in the package named `classes`) and the JDBC result set.

It's worth observing that the two CREATE PROCEDURE `rankedEmps` statements above wouldn't be allowed by SQL, because the names and signatures are identical and SQL would not be able to determine which one to invoke when requested by an application. They could be permitted if they were created in different schemas, however.

The next clause will show an example invocation of this procedure.

3.11 Calling the rankedEmps procedure

After you have installed the `Routines3` class in an SQL system and executed the CREATE PROCEDURE for `rankedEmps`, you can call the `rankedEmps` procedure as if it were an SQL stored procedure. Such a call could be written with any facility that defines mechanisms for processing SQL result sets — that is, SQL/CLI, JDBC, and SQL/OLB. The following is an example of such a call using JDBC:

```
java.sql.CallableStatement stmt = conn.prepareCall( "{call rankedEmps(?)}");
  stmt.setInt(1, 3);
  ResultSet rs = stmt.executeQuery();
  while (rs.next()) {
    String name = rs.getString(1);
    int region = rs.getInt(2);
    BigDecimal sales = rs.getBigDecimal(3);
    System.out.print("Name = " + name);
    System.out.print("Region = " + region);
    System.out.print("Sales = " + sales);
    System.out.println();
}
```

Note that the call of the `rankedEmps` procedure supplies only the single parameter that was declared in the CREATE PROCEDURE statement. The SQL system then implicitly supplies, as applicable, a parameter that is an empty array of `ResultSet` or an empty array of `classes.SalesReport`, and calls the Java method. That Java method assigns the output result set or iterator to the array parameter. And, when the Java method completes, the SQL system returns the result set or iterator in that output array element as an SQL result set.

3.12 Overloading Java method names and SQL names

When you use CREATE PROCEDURE/FUNCTION statements to specify SQL names for Java methods, the SQL names can be overloaded. That is, you can specify the same SQL name in multiple CREATE PROCEDURE/FUNCTION statements. Note that support for such SQL overloading is an optional feature.

Consider the following Java classes and methods. These are contrived routines intended only to illustrate overloading, and we won't show the routine bodies.

```
public class Over1 {
  public static int isOdd (int i) {...};
  public static int isOdd (float f) {...};
  public static int testOdd (double d) {...};
}
public class Over2 {
  public static int isOdd (java.sql.Timestamp t) {...};
  public static int oddDateTime (java.sql.Date d) {...};
  public static int oddDateTime (java.sql.Time t) {...};
}
```

Note that the `isOdd` method name is overloaded in the `Over1` class, and the `oddDateTime` method name is overloaded in the `Over2` class.

Assume that the above classes are in a JAR `~/classes/Over.jar`, which you install:

```
SQLJ.INSTALL_JAR ('file:~/classes/Over.jar', 'over_jar', 0)
```

To reference these methods in SQL, you will of course need to specify SQL names for them with CREATE FUNCTION statements. These CREATE FUNCTION statements can specify SQL names that are overloaded. The overloading of the SQL names is completely separate from the overloading in the Java names. This is illustrated in the following.

Recall that you can specify the same Java method in multiple CREATE PROCEDURE/FUNCTION statements.

```
CREATE FUNCTION odd (INTEGER) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'over_jar:Over1.isOdd';
CREATE FUNCTION odd (REAL) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'over_jar:Over1.isOdd';
CREATE FUNCTION odd (DOUBLE PRECISION) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'over_jar:Over1.testOdd';
CREATE FUNCTION odd (TIMESTAMP) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'over_jar:Over2.isOdd';
CREATE FUNCTION odd (DATE) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'over_jar:Over2.oddDateTime';
CREATE FUNCTION odd (TIME) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'over_jar:Over2.oddDateTime';
CREATE FUNCTION is_odd (INTEGER) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'over_jar:Over1.isOdd';
CREATE FUNCTION test_odd (REAL) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'over_jar:Over1.isOdd';
```

Note the following characteristics of these CREATE FUNCTION statements:

— The SQL name `odd` is defined on the two `isOdd` methods and the `testOdd` method of `Over1`, and also the `isOdd` method and two `oddDateTime` methods of `Over2`. That is, the SQL name `odd` *spans* both overloaded and non-overloaded Java names.

— The SQL names `is_odd` and `test_odd` are defined on the two `isOdd` methods of `Over1`. That is, those two different SQL names are defined on the same Java name.

The rules governing overloading are those of the SQL language as defined in Subclause 11.60, "<SQL-invoked routine>", in [ISO9075-2], and in Subclause 10.4, "<routine invocation>", in [ISO9075-2]. This includes:

— Rules governing what parameter combinations can be overloaded. That is, the legality (or not) of the following CREATE statements is determined by SQL language rules:

```
CREATE FUNCTION is_odd (INTEGER) RETURNS INTEGER...
CREATE FUNCTION is_odd (SMALLINT) RETURNS INTEGER...
CREATE PROCEDURE is_odd (SMALLINT) ...
```

— Rules governing the resolution of calls using overloaded SQL names. That is, the determination of which Java method is called by "`odd(x)`" for some data item "`x`" is determined by SQL language rules.

The EXTERNAL NAME clauses of the above CREATE FUNCTION statements specify only the JAR name and method name of the Java method. For example:

```
CREATE FUNCTION odd (INTEGER) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'over_jar:Over1.isOdd';
```

You can also include the Java method signature (*i.e.*, a list of the parameter data types) of a method in the EXTERNAL NAME clause. For example:

```
CREATE FUNCTION odd (INTEGER) RETURNS INTEGER
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'over_jar:Over1.isOdd (int)';
```

The group of eight example CREATE FUNCTION statements, shown earlier in this clause, do not require Java method signatures, but you can include them for clarity. Subclause 3.14, "Java method signatures in the CREATE statements", describes cases where the Java method signature is required.

3.13 Java main methods

[Java] places special no requirements on any method named `main`. However, a JVM recognizes a method named `main`, with the following Java method signature, as the method to invoke when only a class name is provided:

```
public static void main (String[ ]);
```

If you specify a Java method named main in an SQL CREATE PROCEDURE...EXTERNAL statement, then that Java method shall have the above Java method signature. The signature of the SQL procedure can either be:

— A single parameter that is an array of CHARACTER or CHARACTER VARYING. That array is passed to the Java method as the String array parameter. Note: This SQL method signature can only be used in SQL systems that support array data types in SQL.

— Zero or more parameters, each of which is CHARACTER or CHARACTER VARYING. Those *N* parameters are passed to the Java method as a single *N* element array of String.

3.14 Java method signatures in the CREATE statements

Consider the following method, `job1`, which has an integer parameter and returns a `String` with the job corresponding with a jobcode value:

```
public class Routines4 {
  //A String method that will be called as a function
  public static String job1 (Integer jc) throws SQLException {
    if (jc == 1) return "Admin";
    else if (jc == 2) return "Sales";
    else if (jc == 3) return "Clerk";
else if (jc == null) return null;
    else return "unknown jobcode";
  }
}
```

Note that we suffix the method name with a “1” in anticipation of subsequent variants of the method.

Assume that you install this class in SQL:

```
SQLJ.INSTALL_JAR ('file:~/classes/Routines4.jar', 'routines4_jar', 0)
```

You might want to specify an SQL function `job_of1` defined on the `job1` method:

```
CREATE FUNCTION job_of1(jc INTEGER) RETURNS CHARACTER VARYING(20)
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines4_jar:Routines4.job1';
```

However, as written above, this CREATE statement is not valid. Note that the data type of the parameter of the Java method `job1` is Java `Integer` (which is short for `java.lang.Integer`), and we have specified the SQL data type INTEGER for the corresponding parameter of the SQL `job_of1` function. However, the detailed rules for the external Java form of the SQL CREATE PROCEDURE/FUNCTION statement specifies that the default Java parameter data type for an SQL INTEGER parameter is the Java `int` data type, not the Java `Integer` data type. Subclause 3.15, “Null argument values and the RETURNS NULL clause”, describes some reasons why you may want to specify Java `Integer` rather than Java `int`.

If you want to specify an SQL CREATE PROCEDURE/FUNCTION statement for a Java method whose parameter data types include Java types differing from their default Java types, then you specify those data types in a Java method signature in the CREATE statement. This Java method signature is written after the Java method name in the EXTERNAL NAME clause. For example, the above CREATE statement for the `job1` method would be written as:

```
CREATE FUNCTION job_of1(jc INTEGER) RETURNS CHARACTER VARYING(20)
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines4_jar:Routines4.job1(java.lang.Integer)';
```

If you specify data types in the Java method signature of a CREATE statement that specifies DYNAMIC RESULT SETS, then you shall include the implicit trailing result set or iterator parameters in that Java method signature. You do not, however, include those trailing parameters in the SQL signature. For example, you would write the CREATE of Subclause 3.10, “A CREATE PROCEDURE rankedEmps for orderedEmps”, as follows:

```
CREATE PROCEDURE rankedEmps (region INTEGER)
  READS SQL DATA
  DYNAMIC RESULT SETS 1
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines3_jar:Routines3.orderedEmps (int, java.sql.ResultSet[ ]);
```

3.15 Null argument values and the RETURNS NULL clause

Consider the Java method `job1` and the corresponding SQL function `job_of1`, which we defined in Subclause 3.14, “Java method signatures in the CREATE statements”.

You can call the SQL function `job_of1` in SQL statements such as the following:

```
SELECT name, job_of1(jobcode)
FROM emps
WHERE job_of1(jobcode) <> 'Admin';
```

Suppose that a row of the EMPS table has a null value in the JOBCODE column. Note that the Java data type of the parameter of the `job1` method is Java `Integer` (that is, `java.lang.Integer`). The Java `Integer` data type is a class, rather than a scalar data type, so its values include both numeric values, and also the null reference value. When an SQL null value is passed as an argument to a Java parameter whose data type is a Java class, the null SQL value is passed as a Java null reference. Such a null reference can be tested within the Java method, as shown in `Routines4.job1`.

Now consider the following similar method, which specifies its parameter data type to be the Java scalar data type `int`, rather than the Java class `Integer`.

```
public class Routines5 {
  //A String method that will be called as a function
  public static String job2 (int jc)
      throws SQLException {
    if (jc == 1) return "Admin";
    else if (jc == 2) return "Sales";
    else if (jc == 3) return "Clerk";
    else return "unknown jobcode";
  }
}
```

Assume that you install this class in SQL:

```
SQLJ.INSTALL_JAR( 'file:~/classes/Routines5.jar', 'routines5_jar', 0)
CREATE FUNCTION job_of2 (jc INTEGER) RETURNS CHARACTER VARYING(20)
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines5_jar:Routines5.job2';
```

You can then call the SQL function `job_of2` in SQL statements such as the following:

```
SELECT name, job_of2 (jobcode)
FROM emps
WHERE job_of2(jobcode) <> 'Admin';
```

When this SELECT statement encounters a row of the EMPS table in which the JOBCODE column is null, the effect of the null value on the call(s) of the `job_of2` function is different than for the previous `job_of` function. The `job_of2` function is defined on the method `Routines5.job2`, whose parameter has the scalar data type `int`, rather than the class data type `java.lang.Integer`. The Java `int` data type (and other Java scalar data types) has no null reference value, and no other representation of a null value. Therefore, if the `job2` method is invoked with a null SQL value, then an exception condition is raised.

To summarize:

— The following Java data types have null reference values, and can accommodate SQL arguments that are null:

  `java.lang.String`, `java.math.BigDecimal`, `byte[ ]`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`, `java.lang.Double`, `java.lang.Float`, `java.lang.Integer`, `java.lang.Short`, `java.lang.Long`, `java.lang.Boolean`

— The following Java data types are scalar data types that cannot accommodate nulls. An exception condition will be raised if an argument value passed as such a parameter data type is null:

  `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`

The exception condition that is raised when you attempt to pass a null argument to a Java parameter that is a non-nullable data type is analogous to the traditional SQL exception condition that is raised when you attempt to FETCH or SELECT a null column value into a host variable for which you did not specify a null indicator variable. In both cases, the “receiving” parameter or variable is unable to accommodate the actual run-time null value, so an exception condition is raised.

When you code Java methods specifically for use in SQL, you will probably tend to specify Java parameter data types that are the nullable Java data types. You may, however, also want to use Java methods in SQL that were not coded for use in SQL, and that are more likely to specify Java parameter data types that are the scalar (non-nullable) Java data types.

You can call such functions in contexts where null values will occur by invoking them conditionally, *e.g.*, in CASE expressions. For example:

```
SELECT name,
       CASE
         WHEN jobcode IS NOT NULL THEN job_of2 (jobcode)
         ELSE NULL
       END
FROM emps
WHERE CASE
        WHEN jobcode IS NOT NULL THEN job_of2 (jobcode)
        ELSE NULL
      END<> 'Administrator';
```

You can also make such CASE expressions implicit, by specifying the RETURNS NULL ON NULL INPUT option in the CREATE FUNCTION statement:

```
CREATE FUNCTION job_of22 (jc INTEGER) RETURNS CHARACTER VARYING(20)
  RETURNS NULL ON NULL INPUT
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines5_jar:Routines5.job2';
```

When an SQL function is called whose CREATE FUNCTION statement specifies RETURNS NULL ON NULL INPUT, then if the runtime value of any argument is null, the result of the function call is set to null, and the function itself is not invoked.

The following SELECT statement invokes the `job_of22` function.

```
SELECT name, job_of22(jobcode)
```

```
FROM emps
WHERE job_of22(jobcode) <> 'Administrator';
```

This SELECT is equivalent to the previous SELECT that invokes the `job_of2` function within CASE expressions. That is, the RETURNS NULL ON NULL INPUT clause in the CREATE FUNCTION statement for `job_of22` makes the null-testing CASE expressions implicit.

The RETURNS NULL ON NULL INPUT option applies to *all* of the parameters of the function, not just to the parameters whose Java data type is not nullable.

The convention that the RETURNS NULL ON NULL INPUT option defines for a function is the same convention that is followed for most built-in SQL functions and operators: if any operand is null, then the value of the operation is null.

The alternative to the RETURNS NULL ON NULL INPUT clause is CALLED ON NULL INPUT, which is the default.

You can specify the same Java method in multiple CREATE FUNCTION statements (*i.e.*, defining SQL synonyms), and those CREATE FUNCTION statements can either specify RETURNS NULL ON NULL INPUT or CALLED ON NULL INPUT, as illustrated by the above `job_of2` and `job_of22`.

If you create multiple SQL functions named `job_of22` (with different numbers and/or types of parameters), you can specify (or default to) CALLED ON NULL INPUT in some of the CREATE FUNCTION `job_of22` statements, and specify RETURNS NULL ON NULL INPUT in others. The actions of the RETURNS NULL ON NULL INPUT clause are taken after overloading resolution has been done and a particular CREATE FUNCTION statement has been identified.

The RETURNS NULL ON NULL INPUT and CALLED ON NULL INPUT clauses can only be specified in CREATE FUNCTION statements, that is, not in CREATE PROCEDURE statements. This is because there is no equivalent conditional treatment of procedure calls that would be as generally useful.

3.16 Static variables

Java static methods can be contained in Java classes that have static variables, and, in Java, static methods can both reference and set static variables. For example:

```
public class Routines6 {
  static String jobs;
  public static void setJobs (String js) throws SQLException {jobs=js;}
  public static String job3(int jc) throws SQLException {
    if (jc < 1 || jc * 5 > length(jobs)+1) return "Invalid jobcode";
    else return jobs.substring(5*(jc-1), 5*jc);
  }
}
```

Assume that you install this class in an SQL system:

```
SQLJ.INSTALL_JAR('file:~/classes/Routines6.jar', 'routines6_jar', 0);
```

The class `Routines6` has a static variable `jobs`, which is set by the static method `setJobs` and referenced by the static method `job3`. A class such as `Routines6` that dynamically modifies the values of static variables is well-defined in Java, and can be useful. However, when such a class is installed in an SQL system, and the methods `setJobs` and `job3` are defined as SQL procedures and functions (<SQL-invoked routine>), the scope of the assignments to the static variable `jobs` is implementation-dependent. That is, the scope of that variable is not specified, and is likely to differ across implementations (and possibly across the releases of a given implementation).

For example:

```
CREATE PROCEDURE set_jobs (js CHARACTER VARYING(100))
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines6_jar:Routines6.setJobs';
CREATE FUNCTION job_of3 (jc integer) RETURNS CHARACTER VARYING(20)
  RETURNS NULL ON NULL INPUT
  LANGUAGE JAVA PARAMETER STYLE JAVA
  EXTERNAL NAME 'routines6_jar:Routines6.job3';
CALL set_jobs ('AdminSalesClerk');
SELECT name, job_of3 (jobcode)
FROM emps
WHERE job_of3(jobcode) <> 'Admin';
```

This appears to be a straightforward use of the `Routines6` class in SQL. The call of `set_jobs` specifies a list of job code values, which a user might reasonably assume is “cached” by the SQL-environment and used in subsequent calls of `job_of3`. However, since the scope of the static variable `jobs` in the SQL-environment is implementation-dependent, the answers to the following questions regarding the values passed to the `set_jobs` procedure are likely to differ across implementations:

— Is the `set_jobs` value visible only to the current session? Or also to concurrent sessions and to later non-concurrent sessions?

— Does the `set_jobs` value persist across a COMMIT? Is it reset by a ROLLBACK?

The implication of this uncertainty is that you should not use classes that assign to static variables in SQL. Note, however, that such assignments will not (necessarily) be detected by the SQL implementation, either when you CREATE PROCEDURE/FUNCTION or when you call a routine.

You can prevent assignments to static variables in Java by declaring them with the `final` property.

3.17 Dropping SQL names of Java methods

After you have created SQL procedure or function names for Java methods, you can drop those SQL names with a normal SQL DROP statement:

```
DROP FUNCTION region RESTRICT;
```

A DROP statement has no effect on the Java method (or class) on which the SQL name was defined. Dropping an SQL procedure or function implicitly revokes any granted privileges for that routine.

3.18 Removing Java classes from SQL

You can completely uninstall a JAR with the `SQLJ.REMOVE_JAR` procedure. For example:

```
SQLJ.REMOVE_JAR ('routines_jar', 0);
```

As noted earlier, JARs can contain *deployment descriptors*, which specify implicit actions to be taken by the `SQLJ.INSTALL_JAR` and `SQLJ.REMOVE_JAR` procedures. The second parameter is an integer that specifies whether you do or do not (indicated by non-zero or zero values, respectively) want the `SQLJ.REMOVE_JAR` procedure to execute the actions specified by a deployment descriptor in the JAR.

After the `SQLJ.REMOVE_JAR` procedure performs any actions specified by the JAR's deployment descriptor file(s), there shall be noremaining SQL procedure or function whose external name references any method of any class in the specified JAR. Any such remaining SQL procedures or functions shall be explicitly dropped before the `SQLJ.REMOVE_JAR` procedure will be able to complete successfully.

3.19 Replacing Java classes in SQL

Assume that you have installed a Java JAR in SQL, and you want to replace some or all of the contained classes, *e.g.*, to correct or improve them. You can do this by using the `SQLJ.REMOVE_JAR` procedure to remove the current JAR, and then using the `SQLJ.INSTALL_JAR` procedure to install the new version. However, you will probably have executed one or more SQL DDL statements that depend on the methods of the classes that you want to replace. That is, you may have executed one or more of the following DDL operations:

— CREATE PROCEDURE/FUNCTION statements referencing the classes.

— GRANT statements referencing those SQL procedures and functions.

— CREATE PROCEDURE/FUNCTION statements for SQL procedures and functions that invoke those SQL procedures and functions.

— CREATE VIEW/TABLE statements for SQL views and tables that invoke those SQL procedures and functions.

The rules for the `SQLJ.REMOVE_JAR` procedure require that you drop all SQL procedure/functions that directly reference methods of a class before you can remove the JAR containing the class. And, SQL rules for RESTRICT, as specified in the SQL <drop routine statement>, require that you drop all SQL objects (tables, views, SQL-server modules, and routines whose bodies are written in SQL) that invoke a procedure/function before you drop the procedure/function.

Thus, if you use the `SQLJ.REMOVE_JAR` and `SQLJ.INSTALL_JAR` procedures to replace a JAR, you will have to drop the SQL objects that directly or indirectly depend on the methods of the classes in the JAR, and then re-create those items.

The `SQLJ.REPLACE_JAR` procedure avoids this requirement, by performing an instantaneous *remove* and *install*, with suitable validity checks. You can therefore call the `SQLJ.REPLACE_JAR` procedure without first dropping the dependent SQL objects.

For example, in Subclause 3.4, “Installing region and correctStates in SQL”, we installed the class of `Routines1` with the following statement:

```
SQLJ.INSTALL_JAR( 'file:~/classes/Routines1.jar', 'routines1_jar', 0)
```

You can replace that JAR with a statement such as:

```
SQLJ.REPLACE_JAR( 'file:~/revised_classes/Routines1.jar', 'routines1_jar')
```

Note that the JAR name shall be the same. It identifies the existing JAR, and will subsequently identify the replacement JAR. The URL of the replacement JAR can be the same as or different from the URL of the original JAR.

In the general case, there will be classes in the old JAR that are not in the new JAR, classes that are in both JARs, and classes that are in the new JAR and not in the old JAR. These are referred to respectively as unmatched old classes, matching old/new classes, and unmatched new classes.

The validity requirements on the replacement JAR are:

— There shall be no SQL procedure or function whose routine descriptor's <external routine name> specified an <external Java reference string> that references any method of any unmatched old class (since all unmatched old classes will be removed).

— Any CREATE PROCEDURE/FUNCTION statement that references a method of a matching class shall be a valid statement for the new class.

— There shall be no SQL user-defined type whose descriptor's <jar and class name> references any unmatched old class.

— Any CREATE TYPE statement that references a method of a matching class shall be a valid statement for the new class.

If these requirements are satisfied, the `SQLJ.REPLACE_JAR` procedure deletes the old classes (both unmatched and matching) and installs the new classes (both unmatched and matching).

3.20 Visibility

The `SQLJ.INSTALL_JAR` procedure will install any Java classes into the SQL system. However, not all methods of all classes can be referenced in SQL. Only *visible* methods of *visible* classes can be referenced in SQL. The notion of visible classes and methods is based on the concept of *mappable* data types. They may be summarized as follows:

— A Java data type is *mappable* to SQL (and vice versa) if and only if it has a corresponding SQL data type, or it is an array that is used for OUT parameters, or it is an array that is used for result sets.

— A Java method is *mappable* (to SQL) if and only if the data type of each parameter is mappable, and the result type is either a mappable data type or is `void`.

A Java method is *visible* in SQL if and only if it is `public`, `static`, and mappable.

Only the visible installed methods can be referenced in SQL. Other methods simply don't exist in SQL. Attempts to reference them will raise implementation-defined syntax errors such as *unknown name*.

Non-visible classes and methods can, however, be used by the visible methods.

3.21 Exceptions

SQL exception conditions are defined for the SQL/JRT procedures. For example, if the URL argument specified in calls to `SQLJ.INSTALL_JAR` or `SQLJ.REPLACE_JAR` (*etc.*) is invalid, an SQL exception condition (`java.sql.SQLException`) with a specified SQLSTATE will be raised. These exception conditions are specified in the definitions of the procedures. Java exceptions that are thrown during execution of a Java method in SQL can be caught within Java, and if this is done, then those exceptions do not affect SQL processing.

Any Java exceptions that are uncaught when a Java method called from SQL completes will be returned in SQL as SQL exception conditions.

For example, in Subclause 3.3, “Example Java methods: region and correctStates”, we defined the Java method `Routines1.region`. And, in Subclause 3.5, “Defining SQL names for region and correctStates”, we defined the SQL function name `region_of` for the Java method `Routines1.region`.

The Java method `Routines1.region` throws an exception if the argument value is not in a specified range of values:

```
public class routines1 {
  public static int region(String s) throws SQLException {
    if (s.equals ("MN") || s.equals ("VT") || s.equals ("NH")) return 1;
    else if (s.equals ("FL") || s.equals ("GA") || s.equals ("AL")) return 2;
    else if (s.equals ("CA") || s.equals ("AZ") || s.equals ("NV")) return 3;
    else throw new SQLException("Invalid state code", "38001");
  }
}
```

Assume that the EMPS table contains a row for which the value of the STATE column is 'TX'. The following SELECT will therefore raise an exception condition when it encounters that row of EMPS:

```
SELECT name, region_of(state)
FROM emps
WHERE region_of(state) = 1;
```

The call of the `region_of` function with an invalid parameter ('TX') will raise the SQL exception condition with the SQLSTATE of '38001'. The SQL message text associated with that exception will be the following string:

```
'Invalid state code'
```

The message text and SQLSTATE may be specified in the Java exception specified in the Java `throw` statement. If that exception does not specify an SQLSTATE, then the default SQL exception condition for an uncaught Java exception is raised.

When a Java method executes an SQL statement, any exception condition raised in the SQL statement will be raised in the Java method as a Java exception that is specifically the `SQLException` subclass of the Java `Exception` class. The effect of such an SQL exception condition on the outer SQL statement that called the Java method is implementation-defined. For portability, a Java method that is called from SQL, that itself executes an SQL statement, and that catches an `SQLException` from that inner SQL statement should re-throw that `SQLException`.

3.22 Deployment descriptors

When you install a JAR containing a set of Java classes into SQL, you shall execute one or more CREATE PROCEDURE/FUNCTION statements before you can call the static methods of those classes as SQL procedures and functions. And, you may also want to perform various GRANT statements for the SQL names created by those CREATE PROCEDURE FUNCTION statements. When you later remove a JAR, you will want to execute corresponding DROP PROCEDURE/FUNCTION statements and REVOKE statements.

If you plan to install a JAR in several SQL systems, the various CREATE, GRANT, DROP, and REVOKE statements will often be the same for each such SQL system. One way that you could simplify the install and remove actions would be as follows:

— Provide methods called “`afterInstall`” and “`beforeRemove`” to be executed as an “install script” and “remove script”, performing such actions as the following:

  - The `afterInstall` method: The CREATE and GRANT statements that you want to be performed when the JAR is installed.
  - The `beforeRemove` method: The DROP and REVOKE statements (the inverse of the actions of the `afterInstall` method) that you want to be performed when the JAR is removed.

  That is, the `afterInstall` and `beforeRemove` methods would use SQL/OLB or JDBC to invoke SQL for the desired CREATE, GRANT, DROP, and REVOKE statements.

— Include the `afterInstall` and `beforeRemove` methods in a class, which you might call the `deploy` class, and include that `deploy` class in the JAR that you plan to distribute.

— Instruct recipients of the JAR to do the following to install the JAR:

  - Call the `SQLJ.INSTALL_JAR` procedure for the JAR.
  - Execute a CREATE procedure statement for the `afterInstall` method, giving it an SQL name such as `after_install`. Note that this “bootstrap” action cannot be included in the `afterInstall` method itself.
  - Call the `after_install` procedure. Note: We can assume that the `after_install` procedure will include a CREATE PROCEDURE statement to give the `beforeRemove` method an SQL name such as `before_remove`.

— Instruct recipients of the JAR to proceed as follows to remove the JAR:

  - Call the `before_remove` procedure.
  - Drop the `after_install` and `before_remove` procedures. Note that this action cannot be included in the `beforeRemove` procedure itself.
  - Call the `SQLJ.REMOVE_JAR` procedure.

Note that this simplification of the install and remove actions still requires several manual steps. SQL/JRT therefore provides a mechanism, called *deployment descriptors*, with which you can specify the SQL statements that you want to be executed implicitly by the `SQLJ.INSTALL_JAR` and `SQLJ.REMOVE_JAR` procedures.

If you want the deployment descriptors in a JAR to be interpreted when you install and remove the JAR, then you specify a non-zero value for the `deploy` parameter of the `SQLJ.INSTALL_JAR` procedure and similarly for the `undeploy` parameter of the `SQLJ.REMOVE_JAR` procedure. If a JAR contains a deployment descriptor, then the SQLJ.INSTALL_JAR procedure will use that deployment descriptor to determine the CREATE and GRANT statements to execute after it has installed the classes of the JAR. The corresponding `SQLJ.REMOVE_JAR` procedure uses the deployment descriptor todetermine the DROP and REVOKE statements to execute before it removes the JAR and its classes.

A deployment descriptor is a text file containing a list of SQL CREATEand GRANT statements to be executed when the JAR is installed, and a list of SQL DROP and REVOKE statements to be executed when the JAR is removed.

For example, suppose that you have incorporated the above classes `Routines1`, `Routines2`, and `Routines3` into a single JAR. The following is a possible deployment descriptor that you might want to include in that JAR.

Notes:

— Within a deployment descriptor file, you use the JAR name“`thisjar`” as a placeholder JAR name in the EXTERNAL NAME clauses of CREATE statements.

— The various user names in this example are of course hypothetical.

```
SQLActions[ ] = {
  "BEGIN INSTALL
    CREATE PROCEDURE correct_states (old CHARACTER(20), new CHARACTER(20))
      MODIFIES SQL DATA
      LANGUAGE JAVA PARAMETER STYLE JAVA
      EXTERNAL NAME 'thisjar:Routines1.correctStates';
    GRANT EXECUTE ON correct_states TO Baker;
    CREATE FUNCTION region_of(state CHARACTER(20)) RETURNS INTEGER
      NO SQL
      LANGUAGE JAVA PARAMETER STYLE JAVA
      EXTERNAL NAME 'thisjar:Routines1.region';
    GRANT EXECUTE ON region_of TO PUBLIC;
    CREATE PROCEDURE best2 (OUT n1 CHARACTER VARYING(50), OUT id1 CHARACTER(5),
                            OUT region1 INTEGER, OUT s1 DECIMAL(6,2),
                            OUT n2 CHARACTER VARYING(50), OUT id2 CHARACTER(5),
                            OUT region2 INTEGER, OUT s2 DECIMAL(6,2),
                            region INTEGER)
      READS SQL DATA
      LANGUAGE JAVA PARAMETER STYLE JAVA
      EXTERNAL NAME 'thisjar:Routines2.bestTwoEmps';
    GRANT EXECUTE ON best2 TO Baker, Cook, Farmer;
    CREATE PROCEDURE ordered_emps (region INTEGER)
      READS SQL DATA
      DYNAMIC RESULT SETS 1
      LANGUAGE JAVA PARAMETER STYLE JAVA
      EXTERNAL NAME 'thisjar:Routines3.rankedEmps';
    GRANT EXECUTE ON ordered_emps TO PUBLIC;
  END INSTALL",
  "BEGIN REMOVE
    REVOKE EXECUTE ON correct_states FROM Baker RESTRICT;
    DROP PROCEDURE correct_states RESTRICT;
    REVOKE EXECUTE ON region_of FROM PUBLIC RESTRICT;
    DROP FUNCTION region_of RESTRICT;
    REVOKE EXECUTE ON best2 FROM Baker, Cook, Farmer RESTRICT;
    DROP PROCEDURE best2 RESTRICT;
    REVOKE EXECUTE ON ordered_emps FROM PUBLIC RESTRICT;
    DROP PROCEDURE ordered_emps RESTRICT;
  END REMOVE"
}
```

Assume that `deploy_routines.txt` is the name of a text file containing the above deployment descriptor. You would build a JARcontaining the following:

— The text file `deploy_routines.txt`.

— The class files for `Routines1`, `Routines2`, and `Routines3`.

— A manifest file with the following manifest entry:

```
Name: deploy_routines.txt
SQLJDeploymentDescriptor: TRUE
```

This manifest entry identifies the file `deploy_routines.txt` as a deployment descriptor in the JAR, for the`SQLJ.INSTALL_JAR` and `SQLJ.REMOVE_JAR` procedures to interpret.

Deployment descriptor files can contain syntax errors. In general, any error that can arise in a CREATE or GRANT statement can occur in a deployment descriptor file.

You may want to install a JAR that contains a deployment descriptor file withoutperforming the deployment actions. For example, those actions may contain syntax errors, or may simply be inappropriate for some SQL system. You can do this by specifying a zero value for the `deploy` parameter of the `SQLJ.INSTALL_JAR` procedure, and a zero value for the `undeploy` parameter of the `SQLJ.REMOVE_JAR` procedure.

3.23 Paths

In the preceding clauses, the example JARs and their Java classes referenced other Java classes in the packages `java.lang` and `java.sql`. The JARs and their Java classes that you install can also reference Java classes in other JARs that you have installed or will install. For example, suppose that you have three JARs, containing Java classes relating to administration, project management, and property management.

```
SQLJ.INSTALL_JAR ('file:~/classes/admin.jar', 'admin_jar', 0);
```

At this point, you can execute CREATE PROCEDURE/FUNCTION statements referencing the methods of classes in `admin_jar`. And, you can call those procedures and functions. If, at runtime, the Java methods of `admin_jar` reference system classes or other Java classes that are contained in `admin_jar`, then those references will be resolved implicitly. If the `admin_jar` methods reference Java classes that are contained in `property_jar` (which we will install below), then an exception condition will be raised for an unresolved class reference.

```
SQLJ.INSTALL_JAR ('file:~/classes/property.jar', 'property_jar', 0);
SQLJ.INSTALL_JAR ('file:~/classes/project.jar', 'project_jar', 0);
```

These calls of `SQLJ.INSTALL_JAR` install `property_jar` and `project_jar`. However, references to the `property_jar` classes by classes in `admin_jar` will still not be resolved. Similarly, references within `property_jar` to classes in `project_jar` will not be resolved, and vice versa.

To summarize:

— When you install a JAR, any references within the classes of that JAR to system classes, or to classes that are contained in the same JAR, will be implicitly resolved.

— References to any other classes, installed or not, are unresolved.

— You can install JARs that have unresolved class references, and you can use CREATE PROCEDURE/FUNCTION statements to define SQL routines on the methods of those classes.

— When you call SQL routines defined on Java methods, exceptions for unresolved class references may occur at any time allowed by [Java].

Invoking classes that contain unresolved references can be useful:

— To use or to test partially-written applications.

— To use classes that have some methods that are not appropriate for use in an SQL-environment. For example, a class that has display-oriented or interactive methods that are used in other Java-enabled environments, but not within an SQL system.

To resolve references to classes in other JARs, you use the `SQLJ.ALTER_JAVA_PATH` procedure.

```
SQLJ.ALTER_JAVA_PATH ('admin_jar', '(property.*,property_jar)
                                    (project.*, project_jar)');
SQLJ.ALTER_JAVA_PATH ('property_jar', '(project.*,project_jar)');
SQLJ.ALTER_JAVA_PATH ('project_jar', '(*, property_jar) (*, admin_jar)');
```

The `SQLJ.ALTER_JAVA_PATH` procedure has two arguments, both of which are character strings. In a call `SQLJ.ALTER_JAVA_PATH(JX, PX)`:

— JX is the name of the JAR for which you want to specify a path. This is the JAR name that you specified in the INSTALL_JAR procedure.

— PX is the path of JARs in which you want unresolved class names that are referenced by classes contained in JX to be resolved. The path argument is a character string containing a list of path elements (not comma-separated). Each path element is a parenthesized pair (comma-separated), in which the first item is a pattern, and the second item is a JAR name.

Suppose that at runtime, some method of a class C that is contained in JAR JX is being evaluated. And, suppose that within the execution of class C, a reference to some other class named XC is encountered, such that no class named XC is defined in JAR JX. The path PX specified for JAR JX in the `SQLJ.ALTER_JAVA_PATH` call determines the resolution, if any, of class name XC:

— Each path element '(PAT_i, J_i)' is examined.

— If PAT_i is a fully qualified class name that is equivalent to XC, then XC shall be defined in JAR J_i. If it is not, then the reference to XC is unresolved.

— If PAT_i is a package name followed by an '*', and XC is the name of a class in that package, then XC shall be defined in JAR J_i. If it is not, then the reference to XC is unresolved.

— If PAT_i is a single '*', then if XC is defined in JAR J_i, that resolution is used; otherwise, subsequent path elements are tested.

The paths that we specified above for `admin_jar`, `property_jar`, and `project_jar` therefore have the following effect:

— When executing within `admin_jar`, classes that are in the `property` or `project` packages will be resolved in `property_jar` and `project_jar`, respectively.

— When executing within `property_jar`, classes that are in the `project` package will be resolved in `project_jar`.

— When executing within `project_jar`, all classes will first be resolved in `property_jar`, and then in `admin_jar`.

Note that if a class C contained in `property_jar` directly contains a reference to a class AC contained in `admin_jar`, then that reference to AC will be unresolved, since `admin_jar` is not specified in the path for `property_jar`. But, if that class C invokes a method `project.C2.M` of a class contained in `project_jar`, and `project.C2.M` references class AC, then that reference to AC will be resolved in `admin_jar`, since `admin_jar` is specified in the path for `project_jar`. That is, while class C is being executed, the path specified for `property_jar` is used, and while class C2 is being executed, the path specified for `project_jar` is used. Thus, as execution transfers to classes contained in different JARs, the current path changes to the path specified for each such JAR. In other words, the path specified for a JAR J1 applies only to class references that occur directly within the classes of J1, not to class references that occur in some class contained in another JAR that is invoked from a class of J1.

The path that you specify in a call of the `SQLJ.ALTER_JAVA_PATH` procedure becomes a property of the specified JAR. A given JAR has at most one path. The path (if any) for a JAR applies to all users of the classes and methods in the JAR.

When you call the `SQLJ.ALTER_JAVA_PATH` procedure, the path you specify replaces the current path (if any) for the specified JAR. The effect of this replacement on currently running classes and methods is implementation-defined.

When you execute the `SQLJ.ALTER_JAVA_PATH` procedure, you shall be the owner of the JAR that you specify as the first argument, and you shall have the USAGE privilege on each JAR that you specify in the path argument.

The path facility is an optional feature.

3.24 Privileges

The SQL privilege system is extended for SQL/JRT.

First, the SQLJ build-in procedures are considered to be SQL-schema statements, and as such require implementation-defined privileges to be invoked.

Second, the USAGE privilege is defined for JARs. USAGE is needed on a JAR in order to:

— Reference it in a CREATE PROCEDURE/FUNCTION/TYPE statement.

— List it in an SQL-Java path in an `SQLJ.ALTER_JAVA_PATH` procedure call.

The user who installs a JAR is the owner of that JAR and implicitly has USAGE on the JAR, and can grant USAGE to other users and roles. Only the owner can replace, remove, or alter the JAR.

USAGE privileges on a JAR is an optional feature.

3.25 Information Schema

Additional views and columns are defined for the Information Schema to describe external Java routines and external Java types:

— JARS lists the JARs installed in a database.

— METHOD_SPECIFICATIONS is augmented to include information about static field methods.

— ROUTINES contains information about external Java routines.

— USAGE_PRIVILEGES contains information on USAGE privileges granted on JARs.

— USER_DEFINED_TYPES is augmented to include information about external Java types.

In addition, the usage of JARs by routines, types, and other JARs is shown in a collection of new usage views:

— JAR_JAR_USAGE lists the JARs used in the SQL-Java path of a given JAR.

— ROUTINE_JAR_USAGE names the JAR used in an external Java routine.

— TYPE_JAR_USAGE names the JAR used in an external Java type.

These Information Schema views are optional features.

(Blank page)

4 Types tutorial

4.1 Overview

This tutorial clause shows a series of example Java classes and their methods, and shows how they can be installed in an SQL system and used as data types in SQL.

4.2 Example Java classes

This Subclause shows example Java classes `Address` and `Address2Line`.

— The `Address` class represents street addresses in the USA, with a `street` field containing a street name and building number, and a `zip` field containing a postal code.

— The `Address2Line` class is a subclass of the `Address` class. It adds one additional field, named `line2`, which would contain data such as an apartment number.

— The `Address` and `Address2Line` classes both have the following methods:

- A default niladic constructor.
- A constructor with parameters.
- A `toString` method to return a string representation of an address.

— The `Address` and `Address2Line` classes are both specified to implement the Java interfaces `java.io.Serializable` and `java.sql.SQLData`.

A Java class that will be used as a data type in SQL shall implement either the Java interface `java.io.Serializable` or the Java interface `java.sql.SQLData` or both. This is required to transfer class instances between JVMs and between Java and SQL.

It is assumed that the import statements `import java.sql.*;` and `java.math.*;` have been included in all classes.

The following is the text of the `Address` class:

```
public class Address implements java.io.Serializable, java.sql.SQLData {
    public String street;
    public String zip;
    public static int recommendedWidth = 25;
    private String sql_type; // For the java.sql.SQLData interface
    // A default constructor
    public Address ( ) {
      street = "Unknown";
      zip = "None";
    }
    // A constructor with parameters
    public Address (String S, String Z) {
      street = S;
      zip = Z;
    }
    // A method to return a string representation of the full address
    public String toString( ) {
    return "Street=" + street + " ZIP=" + zip;
    }
    // A void method to remove leading blanks
    // This uses the static method Misc.stripLeadingBlanks.
    public void removeLeadingBlanks( ) {
      street = Misc.stripLeadingBlanks(street);
      zip = Misc.stripLeadingBlanks(zip);
    }
    // A static method to determine if two addresses
    // are in arithmetically contiguous zones.
    public static String contiguous(Address a1, Address a2) {
      if (Integer.parseInt(a1.zip) == Integer.parseInt(a2.zip)+1 ||
          Integer.parseInt(a1.zip) == Integer.parseInt(a2.zip)-1)
        return("yes");
      else
        return("no");
    }
    // java.sql.SQLData implementation:
    public void readSQL (SQLInput in, String type)
        throws SQLException {
      sql_type = type;
      street = in.readString();
      zip = in.readString();
    }
    public void writeSQL (SQLOutput out)
        throws SQLException {
      out.writeString(street);
      out.writeString(zip);
    }
    public String getSQLTypeName () {
      return sql_type;
    }
}
```

The following is the text of the `Address2Line` class, which is a subclass of the `Address` class:

```
public class Address2Line extends Address
        implements java.io.Serializable, java.sql.SQLData {
    public String line2;
    // A default constructor
    public Address2Line () {
      super() ;
      line2 = " ";
    }
    // A constructor with parameters
    public Address2Line (String S, String L2, String Z) {
      street = S;
      line2 = L2;
      zip = Z;
  }
```

```
    // A method to return a string representation of the full address
    public String toString() {
      return "Street=" + street +" Line2=" + line2 + " ZIP=" + zip;
    }
    // A void method to remove leading blanks.
    // Note that this is an imperative method that modifies the instance.
    // This uses the static method Misc.stripLeadingBlanks defined below.
    public void removeLeadingBlanks( ) {
      line2 = Misc.stripLeadingBlanks(line2);
      super.removeLeadingBlanks() ;
    }
    // java.sql.SQLData implementation:
    public void readSQL (SQLInput in, String type)
        throws SQLException {
      super.readSQL(in,type);
      line2 = in.readString();
    }
    public void writeSQL (SQLOutput out)
        throws SQLException {
      super.writeSQL(out);
      out.writeString(line2);
    }
}
    //The following class and method is used only internally in the above Java methods.
    //We won't define an SQL function for this method.
public class Misc {
    // remove leading blanks from a String
    public static String stripLeadingBlanks(String s) {
      int scan;
      for (scan=0; scan < s.length() ; scan++)
        if ( !java.lang.Character.isSpace(s.charAt(scan)) )
          break;
      if (scan == s.length() ) return"";
      else return s.substring(scan);
    }
}
```

4.3 Installing Address and Address2Line in an SQL system

To install classes such as `Address` and `Address2Line` in an SQL system, you proceed as in Clause 3, “Routines tutorial”. The source code for the classes will be in files with filetype `java`, which you compile using the `javac` command to produce object code files with filetype `class`. You then assemble those `class` files into a Java JAR with filetype `jar`, and you place that JAR in a directory for which you can specify a URL. Assume that `file:~/classes/AddrJar.jar` is such a URL. Now, you can install the classes into an SQL system by calling the `SQLJ.INSTALL_JAR` procedure that was described in Clause 3, “Routines tutorial”:

```
SQLJ.INSTALL_JAR ('file:~/classes/AddrJar.jar', 'address_classes_jar', 0);
```

4.4 CREATE TYPE for Address and Address2Line

Before you can use a Java class as an SQL data type, you shall define SQL names for the SQL data type and its fields and methods. You do this with extended forms of the SQL CREATE TYPE statement.

An implementation of SQL/JRT may support these extended forms of the CREATE TYPE statement explicitly as standalone SQL statements, or in deployment descriptor files, or may support animplementation-defined mechanism that achieves the same effect as the CREATE TYPE statement. Deployment descriptor files are included in JARs, and executed implicitly during calls of the built-in SQL/JRT procedure `SQLJ.INSTALL_JAR` that specify a deploy action (third parameter non-zero). This is described in Subclause 3.22, “Deployment descriptors”. In this Annex, we will show the CREATE TYPE statements as standalone SQL statements.

The following SQL CREATE TYPE statements reference the above Java `Address` and `Address2Line` classes:

```
CREATE TYPE addr EXTERNAL NAME 'address_classes_jar:Address'
     LANGUAGE JAVA
      AS (
    street_attr       CHARACTER VARYING(50) EXTERNAL NAME 'street',
    zip_attr          CHARACTER(10) EXTERNAL NAME 'zip' )
    STATIC METHOD rec_width ()
      RETURNS INTEGER
      EXTERNAL VARIABLE NAME 'recommendedWidth',
    CONSTRUCTOR METHOD addr ()
      RETURNS addr SELF AS RESULT
      EXTERNAL NAME 'Address',
    CONSTRUCTOR METHOD addr (s_parm CHARACTER VARYING(50),
                             z_parm CHARACTER(10))
      RETURNS addr SELF AS RESULT
      EXTERNAL NAME 'Address',
    METHOD to_string ()
      RETURNS CHARACTER VARYING(255)
      EXTERNAL NAME 'toString',
    METHOD remove_leading_blanks ()
      RETURNS addr SELF AS RESULT
      EXTERNAL NAME 'removeLeadingBlanks',
    STATIC METHOD contiguous (A1 addr, A2 addr)
      RETURNS CHARACTER(3)
      EXTERNAL NAME 'contiguous';
CREATE TYPE addr_2_line
      UNDER addr
      EXTERNAL NAME 'address_classes_jar:Address2Line'
      LANGUAGE JAVA
      AS (
    line2_attr        CHARACTER VARYING (100) EXTERNAL NAME 'line2' )
    CONSTRUCTOR METHOD addr_2_line ()
      RETURNS addr_2_line SELF AS RESULT
      EXTERNAL NAME 'Address2Line',
    CONSTRUCTOR METHOD addr_2_line (s_parm  CHARACTER VARYING(50),
                                    s2_parm CHARACTER(100),
                                    z_parm  CHARACTER(10))
      RETURNS addr_2_line SELF AS RESULT
      EXTERNAL NAME 'Address2Line',
    METHOD strip ()
```

```
      RETURNS addr_2_line SELF AS RESULT
      EXTERNAL NAME 'removeLeadingBlanks';
```

These CREATE TYPE statements are an extension of the SQL CREATE TYPE statement as defined in [ISO9075-2], Subclause 11.51, "<user-defined type definition>". The above extensions add the EXTERNAL clauses, which are patterned after the EXTERNAL clause of the SQL CREATE PROCEDURE/FUNCTION statement, and the METHOD clauses, which are patterned after SQL CREATE PROCEDURE/FUNCTION statements.

In this Technical Report, we'll describe the basic elements of these CREATE TYPE statements, and defer to later sections discussions of the following less intuitive clauses:

— The Java static field `recommendedWidth` of the `Address` class is represented in the SQL CREATE TYPE by a static method with no arguments, named `rec_width`. This is described in Subclause 4.15, "Static fields".

— The Java `void` method `removeLeadingBlanks` of the `Address` class is represented in the SQL CREATE TYPE for the `addr` type by a method, `remove_leading_blanks` that specifies RETURNS SELF AS RESULT. The `removeLeadingBlanks` and `strip` methods of the `Address2Line` class are treated similarly. This is described in Subclause 4.16, "Instance-update methods". The `strip` method is included to illustrate that multiple SQL methods can reference a single Java method.

— The other clauses of the CREATE TYPE statements are straightforward transliterations of the signatures of the Java classes.

The EXTERNAL clause following the CREATE TYPE clause shall reference a Java class that is in its identified installed JAR. This is referred to as the *subject Java class*, and the SQL data type is the *subject SQL data type*.

If the EXTERNAL clause of a METHOD clause references a Java constructor method (*i.e.*, a method with no explicitly specified return type whose name is the same as the class name), then the SQL method name shall be the same as the SQL data type name. That is, the same conventions for constructor function calls will be used in SQL as in Java.

SQL data types such as `addr` and `addr_2_line` that are defined on Java classes are referred to as *external Java data types*.

4.5 Multiple SQL types for a single Java class

You can define more than one SQL data type on a given Java class. For example:

```
CREATE TYPE another_addr
      EXTERNAL NAME 'address_classes_jar:Address'
      LANGUAGE JAVA
      AS (
    zip_part    CHARACTER(10) EXTERNAL NAME 'zip',
    street_part CHARACTER VARYING(50) EXTERNAL NAME 'street')
    STATIC METHOD rec_width_part () RETURNS INTEGER
      EXTERNAL VARIABLE NAME 'recommendedWidth',
    CONSTRUCTOR METHOD another_addr ()
      RETURNS another_addr SELF AS RESULT
      EXTERNAL NAME 'Address',
    CONSTRUCTOR METHOD another_addr (s_parm CHARACTER VARYING(50),
                                     z_parm CHARACTER(10))
      RETURNS another_addr SELF AS RESULT
      EXTERNAL NAME 'Address',
    METHOD string_rep ()
      RETURNS CHARACTER VARYING(255)
      EXTERNAL NAME 'toString',
    STATIC METHOD contig (A1 another_addr,
                          A2 another_addr)
      RETURNS CHARACTER(3)
      EXTERNAL NAME 'contiguous';
```

The SQL data type `another_addr` is a different data type than the `addr` data type. The two data types aren't comparable, assignable, or union compatible. You can include or omit an SQL data type that is a subtype of the `another_addr` type for “2 line” data. If you define such a subtype, with a name such as `another_2_line`, then instances of `another_2_line` are specializations of `another_addr`, and not of `addr`.

4.6 Collapsing subclasses

Given Java classes and subclasses such as `Address` and `Address2Line`, you can either define SQL data types for each such class, or for a subset of those classes.

Assume that in SQL you only want to use the Java class `Address2Line`. You can define an SQL data type for that class without a corresponding SQL data type for the `Address` class. For example:

```
CREATE TYPE complete_addr
      EXTERNAL NAME 'address_classes_jar:Address2Line'
      LANGUAGE JAVA
      AS (
    zip_attr    CHARACTER(10) EXTERNAL NAME 'zip',
    street_attr CHARACTER VARYING(50) EXTERNAL NAME 'street',
    line2_attr  CHARACTER VARYING(100) EXTERNAL NAME 'line2' )
    STATIC METHOD rec_width ()
      RETURNS INTEGER
      EXTERNAL VARIABLE NAME 'recommendedWidth',
    CONSTRUCTOR METHOD complete_addr ()
      RETURNS complete_addr SELF AS RESULT
      EXTERNAL NAME 'Address2Line',
    CONSTRUCTOR METHOD complete_addr (s_parm  CHARACTER VARYING(50),
                                      s2_parm CHARACTER(100),
                                      z_parm  CHARACTER(10))
      RETURNS complete_addr SELF AS RESULT
      EXTERNAL NAME 'Address2Line',
    STATIC METHOD contiguous (A1 complete_addr,
                              A2 complete_addr)
      RETURNS CHARACTER(3)
      EXTERNAL NAME 'contiguous',
    METHOD to_string ()
      RETURNS CHARACTER VARYING(255)
      EXTERNAL NAME 'toString',
    METHOD strip ()
      RETURNS complete_addr SELF AS RESULT
      EXTERNAL NAME 'removeLeadingBlanks';
```

Note that this CREATE TYPE includes attribute and method definitions for attributes and methods of the superclass, `Addr`. You can include such superclass attributes and methods in a CREATE TYPE only if the CREATE TYPE does not specify UNDER. That is, if a CREATE TYPE specifies a supertype with an UNDER clause, then the CREATE TYPE can only include attributes and methods of its immediate subject Java class.

The subsets of the classes that you can specify in CREATE TYPE statements are restricted. For example, assume that you install a hierarchy of classes `Person`, `Employee`, `Manager`, and `Director`, where each is a subclass of the preceding. You can then define SQL data types for the following subsets of the classes:

— `Person`, `Employee`, `Manager`, and `Director`: This is the full subset. Each SQL data type can include only members of its subject Java class.

— Any one of `Person`, `Employee`, `Manager`, or `Director`. That type can include members from any of its superclasses.

— `Manager` and `Director`: The SQL data type for `Manager` can include members from `Person` and `Employee`. The SQL data type for `Director` can include only members of `Director`.

— `Employee`, `Manager`, and `Director`: The SQL data type for `Employee` can include members from `Person`. The SQL data types for `Manager` and `Director` can include only members of those classes.

— `Employee` and `Manager`. The SQL data type for `Employee` can include members from `Person`. The SQL data types for `Manager` can include only members of that class.

— `Person`, `Employee`, and `Manager`, or `Person` and `Employee`. Each class can include only members of its subject Java class.

The subsets that are not allowed are those that omit an intermediate level of subclass. That is, you cannot define SQL data types for (only) the following subsets of the classes:

— `Person` and `Manager`, or `Person`, `Manager`, and `Director`.

— `Person` and `Director`.

— `Person`, `Employee`, and `Director`, or `Employee` and `Director`.

The rule is simpler than the explanation:

> If a CREATE TYPE statement for SQL type S2 specifies “UNDER S1”, then the subject Java class of S1 shall be the direct superclass of the subject Java class of S2.

Subclause 4.5, “Multiple SQL types for a single Java class”, describes how you can define multiple SQL data types on a single Java class. This also can be done for subtype hierarchies. For example, let P_i, E_i, M_i, and D_i be SQL data types defined on `Person`, `Employee`, `Manager`, and `Director`. For a given number *i*, each type is defined to be a subtype of the preceding *i* type. You can define SQL data types such as:

— E1 and M1, and P2 and E2. That is, M1 is defined to be a subtype of E1, and E2 is defined to be a subtype of P2. In this case, E1 and E2 are different types. Instances of E1 are not specializations of P2.

— P1, E1, and M1, and M2 and D2. That is, E1 is defined to be a subtype of P1, M1 is defined to be a subtype of E1, and D2 is defined to be a subtype of M2. In this case, M1 and M2 are different types. Instances of M2 are not specializations of either P1 or E1, and instances of D2 are not specializations of either P1, E1, or M1.

4.7 GRANT and REVOKE statements for data types

After you have performed the CREATE TYPE statements shown in the preceding clause, you can perform normal SQL GRANT statements to grant the SQL USAGE privilege on the new data type:

```
GRANT USAGE ON TYPE addr TO PUBLIC;

GRANT USAGE ON TYPE addr_2_line TO admin;
```

The syntax and semantics for GRANT and REVOKE of the USAGE privilege for user-defined types are as specified in [ISO9075-2], and are not further described by SQL/JRT.

4.8 Deployment descriptors for classes

You may want to perform the same set of SQL CREATE and GRANT statements in any SQL system in which you install a given JAR of Java classes, together with the corresponding SQL DROP and REVOKE statements when you remove that JAR. You can automate this process by specifying those SQL statements in a *deployment descriptor* file in the JAR. A deployment descriptor file contains a list of CREATE and GRANTstatements to be executed when the JAR is installed, and a list of REVOKE and DROP statements to be executed when the JAR is removed.

The following is an example deployment descriptor file for the above Javaclasses and SQL CREATE and GRANT statements.

```
SQLActions[ ] = {
  "BEGIN INSTALL
     CREATE TYPE addr
         EXTERNAL NAME 'thisJar:Address'
         LANGUAGE JAVA
         AS (
       zip_attr         CHARACTER(10) EXTERNAL NAME 'zip',
       street_attr      CHARACTER VARYING(50) EXTERNAL NAME 'street')
       STATIC METHOD rec_width()
         RETURNS INTEGER
         EXTERNAL VARIABLE NAME 'recommendedWidth',
       CONSTRUCTOR METHOD addr ()
         RETURNS addr SELF AS RESULT
         EXTERNAL NAME 'Address',
       CONSTRUCTOR METHOD addr (s_parm CHARACTER VARYING(50),
                              z_parm CHARACTER(10))
         RETURNS addr SELF AS RESULT
         EXTERNAL NAME 'Address',
       METHOD to_string ()
         RETURNS CHARACTER VARYING(255)
         EXTERNAL NAME 'toString',
       METHOD remove_leading_blanks ()
         RETURNS addr SELF AS RESULT
         EXTERNAL NAME 'removeLeadingBlanks',
       METHOD strip ()
         RETURNS addr SELF AS RESULT
         EXTERNAL NAME 'removeLeadingBlanks',
       STATIC METHOD contiguous (a1 addr, a2 addr)
         RETURNS CHARACTER(3)
```

```
        EXTERNAL NAME 'contiguous';
    GRANT USAGE ON TYPE addr TO PUBLIC;
    CREATE TYPE addr_2_line UNDER addr
        EXTERNAL NAME 'thisJar:Address2Line'
        LANGUAGE JAVA
        AS (
      line2_attr       CHARACTER VARYING(100) EXTERNAL NAME 'line2' )
      CONSTRUCTOR METHOD addr_2_line ()
        RETURNS addr_2_line SELF AS RESULT
       EXTERNAL NAME 'Address2Line',
      CONSTRUCTOR METHOD addr_2_line (s_parm  CHARACTER VARYING(50),
                                      s2_parm CHARACTER(100),
                                      z_parm CHARACTER(10) )
        RETURNS addr_2_line SELF AS RESULT
        EXTERNAL NAME 'Address2Line',
      METHOD strip ()
        RETURNS addr_2_line SELF AS RESULT
        EXTERNAL NAME 'removeLeadingBlanks';
    GRANT USAGE ON TYPE addr_2_line TO admin;
  END INSTALL",
 "BEGIN REMOVE
    REVOKE USAGE ON TYPE addr_2_line FROM admin RESTRICT;
    DROP TYPE addr_2_line RESTRICT;
    REVOKE USAGE ON TYPE addr FROM PUBLIC RESTRICT;
    DROP TYPE addr RESTRICT;
 END REMOVE"
}
```

4.9 Using Java classes as data types

After you have installed a set of Java classes with the `SQLJ.INSTALL_JAR` procedure, and executed the appropriate SQL CREATE statements to specify SQL types defined on the Java classes, you can specify those external Java data types as the data types of SQLcolumns. For example:

```
CREATE TABLE emps (
    name         CHARACTER VARYING(30),
    home_addr    addr,
    mailing_addr addr_2_line
)
```

In this table, the `name` column is an ordinary SQL character string, and the `home_addr` and `mailing_addr` columns are instances of the external Java data types.

SQL columns whose data types are external Java data types are referred toas *SQL/JRT columns*.

Alternatively, if the implementation of SQL/JRT supports typed tables as specified in [ISO9075-2], you can use the SQL type to create a typed table. Other tables can then reference the objects in the typed table. This representation allows the objects in the typed table to be shared (*i.e.*, referenced from multiple objects).

For example, you could store objects of type `addr` in a typed table `addresses` and reference them from one or more other tables:

```
CREATE TABLE addresses OF addr (
    REF IS id SYSTEM GENERATED ) ;
```

```
CREATE TABLE companies (
    name CHARACTER VARYING(100),
    address REF(addr) SCOPE addresses
) ;
CREATE TABLE emps2 (
    name         CHARACTER VARYING(30),
    home_addr    REF(addr) SCOPE addresses,
    mailing_addr addr_2_line
) ;
```

In a typed table such as `addresses`, each attribute of the type becomes a separate column of the same name in the typed table. In addition, the typed table has an implicit identifier column, which identifies a row (*i.e.*, an object) in the table. In the example above, the name of this column is `id` and the values for the column are automatically generated by the database system. [ISO9075-2] supports additional generation mechanisms for object identifiers, which can be defined through extended syntax in the CREATE TYPE statement.

You can store references to the objects of the `addresses` table in columns of type `REF(addr)`. The definition for these columns also identifies the `addresses` table as the scope of the reference column.

4.10 SELECT, INSERT, and UPDATE

After you have specified SQL/JRT columns such as `emps.home_addr`and `emps.mailing_addr`, the values that you assign to those columns shall be Java instances. Such instances are initially generated by calls to constructor methods, using the `NEW` operator as in Java. For example:

```
INSERT INTO emps VALUES ( 'John Doe', NEW addr(), NEW addr_2_line() )
INSERT INTO emps VALUES ( 'Bob Smith', NEW addr('432 Elm Street', '95123'),
                         NEW addr_2_line('PO Box 99', 'attn: Bob Smith', '99678') )
```

The initial values specified for the SQL/JRT columns are the results of constructor method invocations. Note the use of the `NEW` keyword, whose role is the same in the facilities of SQL/JRT as in Java.

Values of such columns can also be copied from one table to another. For example, assume the following additional table:

```
CREATE TABLE trainees (
    name         CHARACTER(30),
    home_addr    addr,
    mailing_addr addr_2_line
);
INSERT INTO emps
      ( SELECT * FROM trainees
        WHERE name IN ('Bill Baker', 'Chuck Morgan', 'Frank Jones') ) ;
```

Inserting objects into typed tables uses the same syntax as for regular base tables. For example:

```
INSERT INTO addresses
    VALUES ('1357 Ocean Blvd.', '99111')
```

Reference values can be obtained either directly from the referenced table (using the identifier column), or from other reference columns. For example, the following statement obtains a reference value stored in the companies table and inserts it into the `emps2` table. This results in a situations where the `addr` object is “shared” by multiple referencing parties, thereby avoiding multiple redundant copies of the same `addr` object.

```
INSERT INTO emps2
    VALUES ( 'Rob White , NEW addr( '165 Oak Street', '95234' ),
             ( SELECT address FROM companies
               WHERE name = 'eBiz Unlimited' ) )
```

4.11 Referencing Java fields and methods in SQL

You can invoke the methods andreference and update the fields of SQL/JRT columns such as `emps.home_addr` and `emps.mailing_addr` using SQL field qualification.

```
SELECT home_addr.to_string() , mailing_addr.to_string()
FROM emps
WHERE name = 'Bob Smith';
SELECT name, home_addr.zip_attr
FROM emps
WHERE home_addr.street_attr= '456 Shoreline Drive';
UPDATE emps
    SET home_addr.street_attr = '457 Shoreline Drive',
        home_addr.zip_attr = '99323'
WHERE home_addr.to_string() LIKE '%456%Shore%';
```

You can also access columns of objects in typed tables and invoke methods on objects in typed tables through references by using the dereference operator (“`->`”).

```
SELECT name, mailing_addr->to_string()
FROM emps2
WHERE name = 'Bob Smith';
SELECT name, mailing_addr->street_attr
FROM emps2
WHERE mailing_addr->zip_attr = '99111';
```

4.12 Extended visibility rules

We have now defined SQL data types on the Java classes `Address` and `Address2Line`, and shown how you can use those classes as the data types of SQL columns.

Defining those SQL data types on the Java classes has one additional effect. Those SQL data types and the Java classes that they are defined upon are now added to the list of corresponding Java and SQL data types, so that we can now use Java methods whose data types are those Java classes. For example:

```
public class Utility {
    // A function version of the removeLeadingBlanks method of Address.
    public static Address stripLeadingBlanks(Address a) {
      return a.removeLeadingBlanks() ;
    }
    // A function version of the removeLeadingBlanks method of Addr2Line.
    public static Addr2Line stripLeadingBlanks(Addr2Line a) {
      return a.removeLeadingBlanks() ;
    }
}
```