H2-2000-344

American National Standard for Information Technology ANSI NCITS 331.2-2000

SQLJ Part 2: SQL Types using the Java™ Programming Language

June 30, 2000

THIS COVER PAGE WILL BE REPLACED BEFORE PUBLICATION

TABLE OF CONTENTS

Foreword

(This foreword is not a part of American National Standard NCITS 331.2-2000.)

This Standard (American National Standard NCITS 331.2-2000, *Information Systems — SQLJ — Part 2: SQLJ SQL Types using the Java™ Programming Language*) is a new standard.

ANSI (the American National Standards Institute) is the United States national standards body charged with development of American National Standards.

This Standard was approved as an American National Standard by the American National Standards Institute on **XXX xx, 200x**.

Requests for interpretation, suggestions for improvement or addenda, or defect reports are welcome.

They should be sent to the NCITS Secretariat, Information Technology Industry Council (ITIC), 1250 Eye Street, NW, Suite 200, Washington, DC 20005.

This Standard was processed and approved for submittal to ANSI by the Accredited Standards Committee NCITS (National Committee for Information Technology Standards). Committee approval of this Standard does not necessarily imply that all committee members voted for approval.

NCITS Membership at the time BSR NCITS 331.2-2000 was approved by NCITS to be forwarded for final approval by BSR:

NCITS Chairman	**NCITS Vice Chair**	**NCITS Secretary**
Ms. Karen Higgenbottom	Mr. Dave Michael	Ms. Monica Vago

*Non-Response **Abstain

PRODUCERS=nn

To Be Supplied

CONSUMERS=nn

To Be Supplied

GENERAL INTEREST=nn

To Be Supplied

Introduction

The organization of this Part of this American National Standard is as follows:

1) Clause 1, “Scope”, specifies the scope of this part of ANSI NCITS 331.
2) Clause 2, “Normative references”, identifies additional standards and publicly-available specifications that, through reference in this part of ANSI NCITS 331, constitute provisions of this part of ANSI NCITS 331.
3) Clause 3, "Introduction", introduces SQLJ Part 2 and its concepts.
4) Clause 4, "Tutorial", describes SQLJ Part 2 features.
5) Clause 5, "SQL elements", defines new SQL statement extensions.
6) Clause 6, "Java topics", defines the conventions used in deployment descriptor files.
7) Clause 7, "Status codes", defines new SQLSTATE class and subclass codes for conditions raised by SQLJ Part 2 implementations.

In the text of this part of ANSI NCITS 331, Clauses begin a new odd-numbered page. Any resulting blank space is not significant.

All Clauses of this part of ANSI NCITS 331 are normative, including Clause 4, ‘‘Tutorial’’.

Introduction

1. SCOPE

This part of ANSI NCITS 331 specifies the manner in which SQL datatypes may be created using the Java™ programming language. (Java is a registered trademark of Sun Microsystems, Inc.)

2. NORMATIVE REFERENCES

1) ANSI/ISO/IEC 9075-2:1999, *Database Language SQL — Foundation*.
2) ANSI/ISO/IEC 9075-4:1999, *Information technology — Database languages — SQL — Part 4: Persistent Stored Modules (SQL/PSM).*
3) ANSI/ISO/IEC 9075-10:2000, *Information Systems — Database Languages — SQL — Part 10: Object Language Bindings (SQL/OLB).*
4) *The Java Language Specification*, James Gosling, Bill Joy, and Guy Steele, Addison-Wesley, 1996.

3. INTRODUCTION

3.1 SQLJ

The term "SQLJ" refers to a series of specifications for ways to use the Java™ programming language with SQL. The specifications are in several parts:

- ANSI/ISO/IEC 9075-10:2000, *Database Language SQL — Part 10: SQL/OLB*

  Specifications for embedding SQL statements in Java methods.

- NCITS 331.1, *SQLJ — Part 1: SQL Routines using the Java™ Programming Language*

  Specifications for calling Java static methods as SQL stored procedures and user-defined functions.

- NCITS 331.2, *SQLJ — Part 2: SQL Types using the Java™ Programming Language*

  Specifications for using Java classes as SQL user-defined datatypes.

This document is the *SQLJ: SQL Types using the Java ™ Programming Language* specification. It defines SQL extensions for using Java classes as datatypes in SQL.

Database Language SQL — Part 10: SQL/OLB specifies facilities for a way to invoking SQL facilities from a Java environment.

SQLJ: SQL Routines using the Java ™ Programming Language specifies SQL extensions for installing Java classes in an SQL system, for invoking static methods of Java classes in SQL as SQL functions and stored procedures, for obtaining specified output values of parameters, and for returning SQL result sets.

SQLJ Part 2 — SQL Types using the Java ™ Programming Language

SQLJ: SQL Types using the Java ™ Programming Language assumes the facilities of *SQLJ: SQL Routines*. The *SQLJ: SQL Types* specification does not repeat or subsume the specifications of *SQLJ: SQL Routines*.

SQLJ: SQL Types does not require or depend on *SQL/OLB.*

Taken together the collection of SQLJ facilities provide a way to write Java classes whose methods invoke SQL, and to use those classes and their methods in an SQL system. This provides a way to write SQL functions, procedures, and datatypes using the Java language.

3.2 Technical components

3.2.1 SQLJ: SQL Routines using the Java™ Programming Language

SQLJ: SQL Routines using the Java ™ Programming Language includes the following:

- New built-in procedures:
  - **sqlj.install_jar** — to load a Java jar file containing a set of Java classes in an SQL system.
  - **sqlj.remove_jar** — to delete a previously installed Java jar file and its Java classes.
- A type of file, called a *deployment descriptor file*, which is a "script" of SQL **create**, **grant**, **drop**, and **revoke** statements. If a jar file that is referenced in a call of the **sqlj.install_jar** procedure contains a deployment descriptor file, then the SQL statements in that deployment descriptor file may be executed during the install or remove actions.
- Extended 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.
- Conventions for returning values of **out** and **inout** parameters, and for returning SQL result sets.

3.2.2 SQLJ: SQL Types using the Java™ Programming Language

SQLJ: SQL Types using the Java™ Programming Language includes the following:

- Extensions of the following SQL statements:
  - **create type** — to specify an SQL name for a Java class. Similar to the **create procedure/function** of *SQLJ: SQL Routines* and the **create type** of SQL/Foundation.
  - **drop type** — to delete the SQL name of a Java class. Similar to the **drop procedure/function** of *SQLJ: SQL Routines* and the **drop type** of SQL/Foundation.
- New forms of reference: Qualified references to the fields and methods of columns whose datatypes are defined on Java classes.

3.3 Conformance

An implementation of NCITS xxx is conformant if it implements all capabilities specified in this standard that are not specified as optional, and if it identifies which of those optional capabilities it implements.

3.4 Organization of the document

The remaining sections of the document are organized as follows:

- The *Tutorial* section describes the features of *SQLJ: SQL Types*.
- The *SQL elements* section defines SQL statement extensions, and new rules for SQL element references and method calls.
- The *Java topics* section defines the deployment descriptors for Java classes used as SQL datatypes.
- The *Status codes* section defines new SQLSTATE exception codes for SQLJ exceptions.

4. TUTORIAL

4.1 Overview

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

4.2 Example Java classes

This section 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 datatype in SQL must 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 Java environments and between Java and SQL.

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 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");
}

// 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( );
        }
        // 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 Using Java classes in SQL: introduction

4.3.1 Installing Address and Address2Line in an SQL system

To install classes such as *Address* and *Address2Line* in an SQL system, you proceed as in *SQLJ: SQL Routines*. 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" file with filetype *jar*, and you place that jar file 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 *SQLJ: SQL Routines*:

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

4.3.2 CREATE TYPE for Address and Address2Line

Before you can use a Java class as an SQL datatype, you must define SQL names for the SQL datatype and its fields and methods. You do this with extended forms of the SQL **create type** statement.

An implementation of *SQLJ: SQL Types* may support these extended forms of the **create type** statement explicitly as standalone SQL statements, or in deployment descriptor files, or may support an implementation-defined mechanism that achieves the same effect as the **create type** statement. Deployment descriptor files are included in jar files, and executed implicitly during calls of the built-in SQLJ procedure **sqlj.install_jar** that specify a deploy action (third parameter non-zero). This is described in section 6.1, “Deployment descriptor files”. In this Tutorial section, 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 varchar(50) external name ' street',
   zip_attr char(10) external name 'zip'
   )
   static method rec_width ( ) returns integer
         external variable name 'recommendedWidth',
   method addr ( ) returns addr external name 'Address',
   method addr (s_parm varchar(50), z_parm char(10)) returns addr
         external name 'Address',
   method to_string ( ) returns varchar(255) external name ‘toString’,
   method remove_leading_blanks ( ) returns addr self as result
         external name ‘removeLeadingBlanks’,
   static method contiguous (A1 addr, A2 addr) returns char(3)
```

```
            external name 'contiguous';
    create type addr_2_line
        under addr
        external name 'address_classes_jar:Address2Line'
        language java
        as(line2_attr varchar(100) external name 'line2')
        method addr_2_line ( ) returns addr_2_line external name 'Address2Line',
        method addr_2_line
            (s_parm varchar(50), s2_parm char(100), z_parm char(10))
            returns addr_2_line
            external name 'Address2Line',
        method to_string ( ) returns varchar(255) external name 'toString',
        method remove_leading_blanks ( ) returns addr_2_line self as result
            external name ‘removeLeadingBlanks’,
        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. 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 section 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 *recommended*W*idth* 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 section 4.3.17, “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 section 4.3.18, “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 must reference a Java class that is in a jar installed in the current catalog. This is referred to as the *subject Java class*, and the SQL datatype is the *subject SQL datatype*.

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 must be the same as the SQL datatype name. I.e. the same conventions for constructor function calls will be used in SQL as in Java.

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

4.3.3 Multiple SQL types for a single Java class

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

```
create type another_addr
        external name 'address_classes_jar:Address'
        language java
        as( zip_part char(10) external name 'zip',
        street_part varchar(50) external name 'street' )
        static method rec_width_part ( ) returns integer
                external variable name 'recommendedWidth',
        method another_addr ( ) returns another_addr external name 'Address',
        method another_addr (s_parm varchar(50), z_parm char(10))
                returns another_addr external name 'Address',
        method string_rep ( ) returns varchar(255) external name ‘toString’,
        static method contig (A1 another_addr, A2 another_addr) returns char(3)
                external name 'contiguous'
```

The SQL datatype *another_addr* is a different datatype than the *addr* datatype. The two datatypes aren’t comparable, assignable, or union compatible. You can include or omit an SQL datatype 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*.

4.3.4 “Collapsing” subclasses

Given Java classes and subclasses such as *Address* and *Address2Line*, you can either define SQL datatypes 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 datatype for that class, without a corresponding SQLdatatype for the *Address* class. For example:

```
create type complete_addr
        external name 'address_classes_jar:Address2Line'
        language java
        as( zip_attr char(10) external name 'zip',
        street_attr varchar(50) external name 'street',
        line2_attr varchar(100) external name 'line2' )
        static method rec_width ( ) returns integer
                external variable name 'recommendedWidth',
        method complete_addr ( ) returns complete_addr
                external name 'Address2Line',
```

```
method complete_addr
        (s_parm varchar(50), s2_parm char(100), z_parm char(10))
         returns complete_addr
        external name 'Address2Line',
static method contiguous (A1 complete_addr, A2 complete_addr)
        returns char(3) external name 'contiguous'

method to_string ( ) returns varchar(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**. I.e. 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 datatypes for the following subsets of the classes:

- *Person*, *Employee*, *Manager*, and *Director*: This is the full subset. Each SQL datatype 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, whether immediate or indirect.
- *Manager* and *Director*: The SQL datatype for Manager can include members from Person and Employee. The SQL datatype for Director can include only members of Director.
- *Employee*, *Manager*, and *Director*: The SQL datatype for Employee can include members from Person. The SQL datatypes for Manager and Director can include only members of those classes.
- *Employee* and *Manager*. The SQL datatype for Employee can include members from Person. The SQL datatypes 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. I.e. you cannot define SQL datatypes 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 must be the immediate superclass of the subject Java class of S2.

Section 4.3.3, "Multiple SQL types for a single Java class" described how you can define multiple SQL datatypes 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 datatypes 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 datatypes such as:

- E1 and M1, and P2 and E2. I.e. 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. I.e. 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.

4.3.5 GRANT and REVOKE statements for datatypes

After you have performed the **create type** statements shown in the preceding section, you can perform normal SQL **grant** statements to grant the SQL **usage** privilege on the new datatype:

```
grant usage on type addr to public;
grant usage on type addr2line to admin;
```

The syntax and semantics for **grant** and **revoke** of the **usage** privilege for user-defined types are as specified in SQL99, and are not further described by this standard.

4.3.6 Deployment descriptors for classes

As described in *SQLJ: SQL Routines*, 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 file of Java classes, together with the corresponding SQL **drop** and **revoke** statements when you remove that jar file. You can automate this process by specifying those SQL statements in a *deployment descriptor* file in the jar file. A deployment descriptor file contains a list of **create** and **grant** statements to be executed when the jar file is installed, and a list of **revoke** and **drop** statements to be executed when the jar file is removed.

The following is an example deployment descriptor file for the above Java classes and SQL **create** and **grant** statements.

```
SQLActions[ ] = {
    "BEGIN INSTALL
        create type addr
            external name 'address_classes_jar:Address'
            language java
            as( zip_attr char(10) external name 'zip',
                street_attr varchar(50) external name 'street' )
```

```
                    static method rec_width( ) returns integer
                        external variable name 'recommendedWidth',
                    method addr ( ) returns addr external name 'Address',
                    method addr (s_parm varchar(50), z_parm char(10)) returns addr
                            external name 'Address',
                    method to_string ( ) returns varchar(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 char(3)
                        external name 'contiguous' ;
            grant usage on type addr to public;
            create type addr_2_line under addr
                    external name 'address_classes_jar:Address2Line'
                    language java
                    as(line2_attr varchar(100) external name 'line2' )
                    method addr_2_line ( ) returns addr_2_line
                        external name 'Address2Line',
                    method addr_2_line
                        (s_parm varchar(50), s2_parm char(100), z_parm char(10))
                            returns addr_2_line
                            external name 'Address2Line',
                    method to_string ( ) returns varchar(255) external name 'toString',
                    method remove_leading_blanks ( )
                        returns addr_2_line self as result
                        external name 'removeLeadingBlanks',
                    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 from public restrict;
            drop type addr restrict;
            revoke usage on type addr_2_line from admin restrict;
            drop type addr_2_line restrict;
        END REMOVE"
}
```

4.3.7 Using Java classes as datatypes

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 datatypes as the datatypes of SQL columns. For example:

```
create table emps (
        name varchar(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 datatypes..

SQL columns whose datatypes are external Java datatypes are referred to as *SQLJ columns*.

Alternatively, if the SQLJ implementation supports typed tables as specified in SQL99, 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 varchar(100),
        address ref(addr) scope addresses
    );


create table emps2 (
        name varchar(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. SQL99 supports additional generation mechanisms for object identifiers, which can be defined through extended syntax in the **create type** statement (see section 5.1, "*CREATE TYPE statement*" and the SQL99 specification for more details).

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.

4.3.8 SELECT, INSERT, and UPDATE

After you have specified SQLJ columns such as *emps.home_addr* and *emps.mailing_addr*, the values that you assign to those columns must 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 SQLJ columns are the results of constructor function calls. Note the use of the **new** keyword, whose role is the same in the *SQLJ: SQL Types* facilities as in Java.

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

```
create table trainees (
        name char(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’) )
```

4.3.9 Referencing Java fields and methods in SQL

You can invoke the methods and reference and update the fields of SQLJ 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’;
```

4.3.10 Extended visibility rules

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

Defining those SQL datatypes on the Java classes has one additional effect. Those SQL datatypes and the Java classes that they are defined upon are now added to the list of corresponding Java and SQL datatypes, so that we can now use Java methods whose datatypes 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( );
        }
}

create function strip(a addr) returns addr
external name 'address_classes_jar:Utility.stripLeadingBlanks'
language java parameter style java;

create function strip(a addr_2_line) returns addr_2_line
external name 'address_classes_jar:Utility.stripLeadingBlanks'
language java parameter style java;
```

Note that the **create function** statement has no syntax to indicate that the referenced method specifies **self as result**. Because the referenced methods have that specification, the two *strip* functions both return copies of their input parameters.

4.3.11 Logical representation of Java instances in SQL

We saw in section 4.3.8, “SELECT, INSERT, **and** UPDATE” that the values assigned to such SQLJ columns are assigned from other SQLJ columns or from the results of calling Java constructors or other methods. Hence, the values assigned to SQLJ columns are ultimately derived from values constructed by Java methods in the Java VM. Such values are represented in SQLJ columns by a value that is obtained from either the Java interface *java.io.Serializable* or the Java interface *java.sql.SQLData*. One or both of those interfaces must be implemented by a Java class that is used as a datatype in SQL. The value obtained from that interface is effectively a copy of the Java instance.

For example:

```
insert into emps
values ('Don Green', new addr('234 Stone Road', '99777'),
                     new addr_2_line( ) )
```

The *addr* constructor method with the **new** operator constructs an *addr* instance and returns a reference to it. However, since the target is an SQLJ column, the SQL system uses the interface *java.io.Serializable* or *java.sql.SQLData* to obtain data that is effectively a copy of the new Java value, and copies that value into the new row of the *emps* table.

The *addr_2_line* constructor method operates the same way as the *addr* method, except that it returns a default instance rather than an instance with specified parameter values. The action taken is, however, the same as for the *addr* instance.

Note that the values stored into SQLJ columns are copies of Java instances, not references. For example:

```
insert into emps (name, home_addr)
```

```
    values ('Sally Green',
        (select home_addr from emps e2 where e2.name='Don Green')
    )
```

This **insert** statement copies the *home_addr* column from the 'Don Green' row to the new 'Sally Green' row. Note that the column value, which contains a copy of the Java instance, is itself copied. Thus, the *home_addr* columns of the 'Sally Green' row and the 'Don Green' row are independent copies, not references to a shared copy. In particular, the following statement has no effect on the 'Sally Green' *home_addr*:

```
update emps
set home_addr.zip_attr = '94608'
where name = 'Don Green';
```

The values stored in SQLJ columns are "reassembled" when a column is passed as a parameter to a function that is defined on a Java method. For example:

```
update emps
set home_addr = strip(home_addr)
where substring(home_addr.street_attr, 1, 1) = ' '
```

The *strip* function is an SQL function defined on the Java static method *Utility.stripLeadingBlanks*. The parameter datatype of the function is the *addr* datatype. When we pass the *home_addr* column as an argument, the value in the current row is reassembled into the Java VM, and a reference to the reassembled value is passed to the method *Utility.stripLeadingBlanks*. The result of that function is of datatype *Address*, which corresponds with the SQL datatype *addr*. The Java interface *java.io.Serializable* or *java.sql.SQLData* is applied to this returned value, and the result is copied back into the column.

Finally, consider the role of SQL nulls. For example:

```
insert into emps (name)
values ('Mike Green');
```

The **insert** statement specifies no values for the *home_addr* or *mailing_addr* columns, so those columns will be set to **null**, in the same manner as any other SQL column whose value is not specified in an **insert**. This null value is generated entirely in SQL, and initialization of the *mailing_addr* column does not involve the Java VM at all.

4.3.12 Converting objects between SQL and Java

While application programmers or end users manipulating Java objects in the database through SQL statements need not be aware of the specific mechanism used to achieve that conversion, the developer of the Java class itself needs to prepare for it in the form of implementing special Java interfaces (i.e., *Serializable* or *SQLData*). The **create type** statement introduces a clause for specifying the mechanism or interface for converting or communicating object state information between the SQL database and Java in the scope of SQL statements. As mentioned above, a conversion from SQL to Java can potentially take place when an object that has been persistently stored in the SQL database is accessed from inside an SQL statement to retrieve attribute (or field) values, or to invoke a method on the object, or when the object is used as an input argument in the invocation of a method. A conversion in the opposite direction, from Java to SQL, may be required

when a newly created or modified object, or an object that is the return value of a method invocation needs to be persistently stored in the database.

SQLJ supports two different options to specify object state conversion, which appear immediately after the "**language java**" clause.

- If the **create type** statement specifies **using serializable**, then the Java interface *java.io.Serializable* is used for object state conversion.
- If the **create type** specifies **using sqldata**, then the Java interface *java.sql.SQLData* defined in JDBC 2.0 is used for object state conversion.
- If the **create type** does not specify a **using** clause, then it is implementation-defined whether the Java interface *java.io.Serializable* or the Java interface *java.sql.SQLData* will be used for object state conversion.

4.3.13 USING SERIALIZABLE

If the **using** clause of a **create type** statement specifies **serializable,** then object state communication is based on the Java interface *java.io.Serializable.* The Java class referenced in the **external name** clause of the **create type** statement must specify "*implements Serializable*" and must provide a niladic constructor.

In this case, the SQL object state that is stored persistently and made available to methods of the SQL type is defined entirely by the Java serialized object state. The attributes defined for the SQL type must correspond to public fields of the corresponding Java class, which must be listed in the attribute external name clauses. Consequently, the SQL attributes define access to those portions of the object state that are intended to become visble inside SQL statements, but might not comprise the complete state of the object (which may include additional private or public fields in the Java class).

4.3.14 USING SQLDATA

If the **using** clause of a **create type** statement specifies **sqldata**, then object state communication is based on the Java interface *java.sql.SQLData* defined in JDBC 2.0. The Java class referenced in the **external name** clause of the **create type** statement must specify '*implements java.sql.SQLData*' and must provide a niladic constructor.

In this case, the attributes defined in the statement comprise the complete state of the SQL object type. Additional public or private attributes defined in the Java class do not become part of the SQLJ object state. The Java object representation may be entirely different from the SQL object attributes, if desired. For example, an SQL Point type may define a geometric point in terms of cartesian coordinates, while the corresponding Java class defines it using polar coordinates. The only requirement to be met by the implementor of the Java class is that the implementation of the SQLData read/writeSQL methods reads/ writes the attributes in the same order in which they are defined in the **create type** statement.

To improve portability, is is possible to also specify external names for SQL attributes, even if **using sqldata** is specified. However, the external name clauses are ignored in this case, because they are not needed for implementing attribute access in SQL or for converting objects between SQL and Java.

4.3.15 Developing for Portability

The following guidelines provide maximum portability of Java classes across different SQLJ implementations that may not support both the **serializable** and the **sqldata** options:

- The Java class used for implementing the SQL type should implement both java.io.Serializable and java.sql.SQLData.
- The Java class should define the complete object state that needs to become persistent or has to be preserved across invocations as public Java fields.
- The external names of the SQL attributes should be specified.
- The **using** clause should be omitted in the **create type** statement, so that an implementation that does not support both interfaces can default to the interface that it supports.

4.3.16 Static methods

The methods of a Java class can be specified as either **static** or non-**static**. For example, in the *Address* class, the *toString* method is non-**static** and the *contiguous* method is **static**.

The **method** clauses of SQL **create type** statements can also specify that a method is **static** or non-**static**. For example, the **create type** for the *addr* SQL type specifies that *to_string* is a non-**static** method and *contiguous* is a **static** method.

In Java and SQL, a non-**static** method is referenced by qualification on an instance of the class/type. For example, assume that JAI and SAI are respectively Java and SQL variables of type/class *Address* or *addr*. You would reference the *toString* or *to_string* methods of those instances by the expressions *JAI.toString( )* or *SAI.to_string( )*.

In Java, a **static** method can be referenced by qualification on *either* the class or on an instance of the class. For example, you can reference the *contiguous* method as either *Address.contiguous*(…) or as *JAI.contiguous*(…)

In SQL, a **static** method is referenced by qualification on the type, not on an instance. For example, you reference the *contiguous* method as *addr::contiguous(…)*. You cannot reference the SQL *contiguous* method as e.g. *SAI.contiguous(..).* Note that in SQL, static method qualification on the type name specifies a double-colon as the qualification punctuation, rather than a single dot. This avoids ambiguities with other SQL constructs.

Note: In addition to referencing static methods by such field qualification, you can also reference static methods by specifying standalone procedures or functions, using the facilities of *SQLJ: SQL Routines*. For example:

```
create function contig_function (A1 addr, A2 addr) returns char(3)
    external name 'address_classes_jar:Address.contiguous'
    language java parameter style java;
```

4.3.17 Static fields

The fields of a Java class can be specified as either **static** or non-**static**. In the example *Address* class, the *street* and *zip* fields are non-**static** and the *recommendedWidth* field is **static.**

The static fields of a java class can be specified as **final**, which makes them read-only. Non-**final** fields can be updated. Users do not always specify the **final** clause for read-only static fields.

The SQL **create type** does not include a facility for specifying atributes to be **static**. This is because of the difficulty in specifying what the scope, persistence, and transactional properties of static fields would be in a database environment.

The SQL **create type** does, however, provide a shorthand mechanism for read-only access to the values of Java static fields. This is illustrated in the create type for *addr*, which specifies a **static method** clause for the *recommendedWidth* field:

```
create type addr external name 'address_classes_jar:Address'
        language java
        using serializable
        as(zip_attr char(10) external name 'zip',
                street_attr varchar(50) external name 'street')
        static method rec_width ( ) returns integer
                external variable name 'recommendedWidth',
        ...
```

The **static method** clause for *rec_width* specifies that it is an integer-valued method with no parameters. The **external** clause for a static method would normally specify the name of a static method of the Java class. In this case, however, the **external** clause specifies the keyword **variable**, and gives the name of a static field of the Java class. When a **static method** clause specifies **external variable**, the method must have no parameters, and the specified Java name must be that of a static field. Such a static method is invoked in the normal manner, and returns the value of the specified Java static field.

Given such a declaration, you can reference the *rec_width* method in the same manner as other static methods, and access the *recommendedWidth* field:

```
        select * from emps
        where length(home_addr.street_attr) > addr::rec_width( )
```

SQL provides no way to update the values of Java static fields.

4.3.18 Instance-update methods

A non-static Java class method is invoked by qualification on an instance of the class. For example, assuming that JAI is an instance of the Java *Address* class, you would reference the *toString* or *removeLeadingBlanks* methods as *JAI.toString( )* or *JAI.removeLeadingBlanks( )*.

Such non-static methods generally reference the fields of the instance that qualifies the method reference, e.g. the instance JAI. The *toString* method references the instance JAI in a read-only manner, returning a string representation of that instance. The *removeLeadingBlanks* method, however, references the qualifying instance in a manner that updates the value of the instance. That update is intended to be a side-effect of the method invocation.

Read-only methods such as *toString* fit naturally into SQL. For example, given the above *emps* table:

```
select name, home_addr.to_string( )
from emps
where home_addr.to_string( ) <> x;
```

1) As described in section 4.3.11, “Logical representation of Java instances in SQL”, Java instances stored in SQL columns and variables are copies of the Java values, not references to such values. Therefore, methods such as *removeLeadingBlanks* that have side-effects on the qualifying instances do not fit naturally into the SQL framework. For this reason, the SQL **create type** for a Java class provides a special mechanism for referencing Java methods that have side effects. This is illustrated by the **method** clause for *remove_leading_blanks*:

```
create type addr external name 'address_classes_jar:Address'
    language java
    using serializable
    as (...)
    method remove_leading_blanks ( ) returns addr self as result
        external name 'removeLeadingBlanks';
```

Recall that the *removeLeadingBlanks* method of the Java Address class is a **void** method. You might therefore expect to specify the SQL *remove_leading_blanks* as a **void** method, i.e. a “procedure method”. However, the SQL **create type** does not provide a way to specify **void** methods or “procedure methods”. This is because such methods would almost always perform side effects on the qualifying instance, and would therefore not be suitable for a value-oriented SQL context.

The SQL *remove_leading_blanks* method specifies the clause **returns self as result**. This clause has the following significance:

- The returns type of the method is defined to be the containing SQL datatype. I.e. the SQL *remove_leading_blanks* method is an *addr*-valued method. This is the case irrespective of the returns type of the underlying Java method. In the typical case, the underlying Java method will be a **void** method, but as we will discuss below, this is not required.
- At runtime, the specified java method is invoked in the normal manner, and updates the fields of a copy of the qualifying instance. When the invocation is complete, the SQL system then makes a copy of the updated value of the qualifying instance, and returns that copy as the result of the method.

As example invocation of *remove_leading_blanks* is as follows:

```
update emps
set home_addr = home_addr.remove_leading_blanks( )
```

```
where …
```

Such an **update** statement proceeds in the normal manner to process each row of the *emps* table, and to perform the **set** actions in each row for which the **where** clause is true. For such a row, the value of the *home_addr* column is passed to the Java virtual machine, which evaluates the *removeLeadingBlanks* method for that instance of the *Address* class. That method performs side effects on the fields of that copy of the current *home_addr* column, and returns. The SQL system then makes a copy of that updated value of the *Address* instance, and returns that copy as the result of the call to *remove_leading_blanks*. That copy is then assigned back to the *home_addr* column of the current row.

Consider a somewhat different invocation of *remove_leading_blanks*:

```
select name, home_addr.remove_leading_blanks( ).street_attr
from emps
where …
```

This **select** statement processes the *emps* rows, and evaluates the select-list for selected rows. The second element of that select-list invokes the *remove_leading_blanks* method of the *home_addr* column. As above, this invocation passes a copy of the *home_addr* value to the Java VM, where the *removeLeadingBlanks* method updates the copy. The SQL system then returns a copy of that updated copy, and extracts the *street_attr* attribute. That *street_attr* attribute will reflect the removal of leading blanks that has been done. However, these actions do not affect the value of the *home_addr* column in the *emps* table.

This **self as result** mechanism provides a general way for SQL to apply the side-effects of arbitrary Java methods.

Java methods that update the qualifying instance will commonly be written as void methods. In some cases, however, such methods are written to return e.g. integer values that provide some sort of status feedback, e.g. an “OK” indication. For this reason, you can specify the **returns self as result** clause for arbitrary Java methods, irrespective of the returns type of the method. Note, however, that this return value that the method invocation explicitly provides is simply discarded by the SQL system, which replaces that explicit returned value with the implicit copy of the qualifying instance.

4.3.19 Subtypes in SQLJ data

Recall the example Java classes *Address* and *Address2Line*., and the corresponding SQL datatypes *addr* and *addr_2_line*. The *Address2Line* class is a subclass of the *Address* class, so you can make use of the substitutability and method overloading characteristics of Java.

For example, you can assign *addr_2_line* values to *addr* columns. We can illustrate this with the *emps* table, in which the *home_addr* column is an *addr* and the *mailing_addr* column is an *addr_2_line*:

```
update emps
set home_addr = mailing_addr
where home_addr is null
```

For the rows in which we perform the above **set** clause, the *home_addr* column will contain an *addr_2_line*, even though the declared type of *home_addr* is *addr*.

Such an assignment implicitly converts an instance of a class to an instance of a superclass of that class.

A conversion from a class to one of its superclasses is called a *widening conversion*, and a conversion from a class to one of its subclasses is called a *narrowing conversion*.

Widening conversions do not have to be specified explicitly. They can be done implicitly with normal assignments. Narrowing conversions must be performed by calling Java methods that perform the narrowing internally and return the narrowed result.

> **Note:** It would be possible to extend the SQL **cast** function to support narrowing conversions.

Neither widening conversions nor narrowing conversions modify the actual instance value or its runtime datatype. Widening and narrowing conversions (assuming no exceptions) simply specify the class to be used for the compile-time type. Thus, when you store *addr_2_line* values from the *mailing_addr* column into the *home_address* column, those values still have the run-time type of *addr_2_line*. The effect of this can be seen in the following example.

Recall that that the *addr* type and the *addr_2_line* subtype both have a method named *toString*, which returns a *String* form of the complete address data.

Consider the following call of the *to_string* method:

```
select name, home_addr.to_string( ) from emps
where home_addr.to_string( ) not like '%Line2=%'
```

For each row of *emps*, the declared type of the *home_addr* column is *addr*, but the runtime typc of the *home_addr* value will be either *addr* or *addr_2_line*, depending on the effect of the previous **update** statement. For rows in which the runtime value of the *home_addr* column is an *addr*, the *to_string* method of the *addr* class will be invoked, and for rows in which the runtime value of the *home_addr* column is an *addr_2_line*, the *to_string* method of the *addr_2_line* subclass will be invoked.

The way that this runtime selection of the *to_string* method is performed is as follows:

- At compile time, the SQL system determines that the calls of *home_addr.to_string( )* are syntactically correct, and that the result type is suitable (e.g. for the **like** predicate).
- At runtime, the SQL system will process the calls of *home_addr.to_string( )* for each row of *emps* in the following steps:
  - The value of the *home_addr* column for the row is reassembled into the Java VM, and a reference R for that reassembled value is obtained.
  - The invocation R.*toString( )* is passed to the Java VM for evaluation. The Java VM performs the run time selection of the appropriate *toString* method, and returns the result.

4.3.20 References to fields and methods of null instances

Assume that you insert the following row into the *emps* table:

```
insert into emps (name) values ('Charles Green')
```

Note that the *home_address* and *mailing_address* columns are both null, since no values were specified for them.

Consider the following **select** statement:

```
select name, home_addr.zip_attr from emps
where home_addr.zip_attr in ('95123', '95125', '95128')
```

The intention of this select is to retrieve the given values of those *emps* rows for which the *zip* field of *home_addr* as one of the specified values. This would not include the rows of *emps* for which *home_addr* is null.

When we execute this **select** statement, the **where** clause will be evaluated for each row of *emps*, including the rows in which the *home_addr* column is null. In Java, and other programming languages, if you attempt to reference a field of a null instance, an exception is raised. If we use that rule in SQL, then the above **select** would raise an exception if the *home_addr* column if any row of *emps* were null. Note that this is an exception for the entire **select** statement, not for particular rows. To get the desired effect, we would have to write the **select** as follows:

```
select name, home_addr.zip_attr from emps
where case when home_addr is not null then home_addr.zip_attr else null end
        in ('95123', '95125', '95128')
```

In fact, if we specify that field references to null instances raise an exception, then virtually all **where** clause references to fields would have to be written with such a **case** expression. This would be exceedingly tedious, so the SQLJ rule for field references to null instances is different from Java:

> *If the value of the instance specified in a field reference is null, then the field reference is null.*

This rule is equivalent to specifying that the above **case** expression is implicit.

This rule therefore allows you to write the **select** in the original form. For rows whose *home_addr* column is null, the field reference *home_addr.zip_attr* will be null.

This rule for field references with null instances only applies to field references in ‘value’, or ‘right-hand-side’ contexts, not to field references that are targets of assignments or **set** clauses.

For example:

```
update emps
        set home_addr.zip_attr = '99123'
        where name = 'Charles Green'
```

This **where** clause will obviously be true for the 'Charles Green' row, so the **update** statement will try to perform the given **set** clause. This will raise an exception, since you cannot assign a value to a field of a null instance. This is because the null instance has no field to which a value can be assigned.

In other words, field references to fields of null instances are valid and return the null value in right-hand-side contexts, and cause exceptions in left-hand-side contexts.

Exactly the same considerations apply to invocations of methods of null instances, and the same rule is applied.

For example, suppose that we modify the previous example and invoke the *to_string* method of the *home_addr* column:

```
select name, home_addr.to_string( ) from emps
where home_addr.to_string( ) = 'Street=234 Stone Road ZIP=99777'
```

If we apply the strict Java rule, then invocations of the *to_string* method for rows in which the *home_addr* column is null will raise an exception. We would therefore, as above, need to code the **select** as follows:

```
select name, home_addr.to_string( ) from emps
where case when home_addr is not null
        then home_addr.to_string( ) else null end
            = 'Street=234 Stone Road ZIP=99777'
```

We therefore extend the Java rule for method invocation in the same manner that we extended the Java rule for field references:

> *If the value of the instance specified in an instance method invocation is null, then the result of the invocation is null.*

4.3.21 Ordering of SQLJ data

In an earlier section we created the *emps* table, with columns *home_addr* and *mailing_addr* whose datatypes are declared to be the Java classes, respectively, *Address* and *Address2Line*. Now suppose that you reference those columns in statements such as the following:

```
select distinct * from emps E1, emps E2
            where E1.home_addr = E2.home_addr
            and E1.mailing_addr > E2.mailing_addr
union
select distinct * from emps E1, emp E2
            where E1.mailing_addr = E2.mailing_addr
            and E1.home_addr > E2.home_addr
            group by home_addr
            order by home_addr, mailing_addr
```

This statement involves numerous references to *home_addr* and *mailing_addr* that imply ordering relationships:

1) The **distinct** keyword is defined in terms of equality of rows, which is specified as a pairwise comparison of corresponding columns. I.e. to determine if two rows of emps are **distinct**, you have to compare their respective home_addr and mailing_addr columns.
2) The direct comparisons using “=” and “>” etc all require ordering properties.
3) The **union** operator doesn’t specify **union all**, so it will eliminate duplicates. This will require the same kind of comparisons as the **distinct** clause.
4) The **group by** requires partitioning the rows into sets with equal values of the grouping column.
5) The **order by** requires determination of the ordering properties of the order columns.

When you create an external Java datatype with a **create type**…**external language java statement**, the new external Java datatype has no ordering capability. I.e. its "*ordering form*" is "*none*". Instances of an external Java datatype whose ordering form is none cannot be used in any of the above ordering relationships.

To define ordering for an external Java datatype, you use the **create ordering** statement:

```
create_ordering_statement ::=
        create ordering for sql_datatype_name ordering_form
ordering_form ::=
        equals only by ordering_category
        | order full by ordering_category
ordering_category ::=
        map with ordering_routine
        | relative with ordering_routine
        | relative with comparable interface
        | state
```

The significance of the **equals only** and **full** alternatives is as follows:

- **equals only** specifies that instances of the associated class can be referenced in equals (=) and not equals (<>) operations, **select distinct**, **union** with duplicate elimination, and **group by**, but not in other ordering contexts.
- **full** specifies that instances of the associated class can be referneced in any ordering context.

The **state** clause specifies that instances will be ordered on the values of the attributes of the type.

The **map** clause specifies the name of a method or function that will map instances of the associated class to values of some built-in SQL datatype, whose ordering defines the ordering of the associated class. The map routine needn't define a 1-1 into correspondence. It can map distinct instance values to the same result. This would be done in order to equate 6/8 and ¾ for a class that implements rational numbers. It can also be done for folded comparisons, and other cases where it is desirable to equate distinct instances.

The **relative with *ordering_routine*** clause specifies the name of a method or function that compares instances of the associated class and returns an integer result. The runtime result value for two instances X and Y is —1, 0, or +1 to indicate respectively that X is *less than*, *equal to*, or *greater than* Y.

The **relative with comparable interface** clause may be used only in orderings for SQL datatypes whose subject Java class implements *java.lang.Comparable.* The *int compareTo* method of the subject Java class determines the relative ordering for two instances X and Y, returning —1, 0, or +1 to indicate respectively that X is *less than*, *equal to*, or *greater than* Y.

5. SQL ELEMENTS

5.1 CREATE TYPE statement

Function

Specify an SQL name for a Java class.

Syntax

create_type_statement ::=
    create type sql_datatype_name
        [**under** sql_datatype_name]
        external name 'class_name**'**
        language java
        [**using** interface_spec]
        as sql_representation
        [**instantiable** | **not instantiable**]
        final | **not final**
        [reference_type_specification]
        [method_spec_list]
sql_datatype_ name ::= [[identifier1.]identifier2.]identifier3
sql_representation ::= **(**attribute_spec [{ **,** attribute_spec}…] **)**
interface_spec ::= **sqldata** | **serializable**
reference_type_specification ::=
    ref is system generated
    | **ref from** (sql_attribute_name [{, sql_attribute_name}…])
    | **ref using** sql_predefined_type [ref_cast]
ref_cast ::=
    [**cast** (**source as ref**) **with** identifier]
    [**cast** (**ref as source**) **with** identifier]
attribute_spec ::=
    sql_attribute_name sql_datatype [**external name '**java_field_name**'**]
method_spec_list ::= method_spec [{ **,** method_spec}…]
method_spec ::= function_method_spec | static_field_method_spec
function_method_spec ::=
    [**static**] **method** sql_method_name sql_function_signature
    method_spec_characteristic **…**

method_spec_characteristic ::=
    data_access_indication
    | { **deterministic** | **not deterministic** }
    | { **returns null on null input** | **called on null input** }
    | external name 'java_method_name [java_signature]**'**
data_access_indication ::=
    no sql
    | **contains sql**
    | **reads sql data**
    | **modifies sql data**
static_field_method_spec ::=
     static method sql_method_name ( ) **returns** sql_datatype
    external variable name 'java_field_name**'**
sql_function_signature ::= **(** [sql_parameters]**) returns** sql_datatype [**self as result**]
sql_parameters ::= **(** sql_parameter [{, sql_parameter}…] **)**
sql_parameter ::= [**in**] [sql_identifier] sql_datatype
class_name ::= jar_id**:**java_class_name
java_class_name ::= [packages.]class_identifier
jar_id ::= sql_identifier
packages ::= package_identifier[.package_identifier]…
package_identifier ::= java_identifier
class_identifier ::= java_identifier
java_field_name ::= java_identifier
java_method_name ::= java_identifier
java_signature ::= *--Defined in SQLJ: SQL Routines*

Definitions and Rules

create type — A form of the **create** statement that specifies an SQL type name for a Java class.

*sql_datatype_nam*e — The qualified SQL name of the SQL type.

The sql_datatype_name is referred to as the *subject SQL datatype name*, and the SQL datatype that it defines is referred to as the *subject SQL datatyp*e.

The identifiers identifier1, identifier2, and identifier3 are the three elements of an SQL 3-part name. The defaults for identifier1 and identifier2 are determined by normal SQL rules.

sql_datatype — An SQL data type.

under sql_datatype_name — Specifies that the subject sql_datatype_name is a subtype of the SQL datatype identified by the sql_datatype_name.

The create_type_statement for the sql_datatype_name specified for **under** must not specify **final**

The sql_datatype_name specified after the **under** keyword is referred to as the immediate SQL supertype name, and the SQL datatype that it identifies is referred to as the immediate SQL supertype.

Rules for the immediate SQL supertype are specified in the *Description* below.

external — Specifies that the **create** statement defines an SQL name for a datatype defined in a programming language other than SQL.

name - Specifies the name of a Java class in a jar installed in the current catalog. A reference to the SQL datatype name is effectively a synonym for the specified Java class.

jar_id — The name of a jar in the current catalog and schema.

java_class_name — The fully-qualified name of a Java class in the specified jar.

The java_class_name is referred to as the *subject Java class name*, and the Java class that it identifies is referred to as the *subject Java class*.

Rules for the subject Java class are specified in the *Description* below.

language java — Specifies that the external datatype is written in Java.

An SQL datatype that is defined with a **create type** that specifies **external language java** is referred to as an *external Java datatype*.

All methods defined for an external Java datatype are implicitly **parameter style java**. Note that **parameter style java** cannot be explicitly specified in the method_spec_characteristic.

using — Specifies the interface and mechanism used when converting between an instance of the subject SQL type and a Java object. Such conversions are performed when an SQLJ column is specified as a (subject) parameter in a method or function invocation, or when a Java object returned from a method or function invocation is stored in an SQLJ column.

If a **using** clause is not specified, then the default interface_spec is implementation-defined.

serializable — Specifies that conversions between Java objects and SQL representations is performed as specified by the Java interface *java.io.Serializable*. The method *java.io.Serializable.writeObject()* is effectively used to convert a Java object to an SQL representation, and the method *java.io.Serializable.readObject()* is effectively used to convert an SQL representation to a Java object.

If **serializable** is specified, then the subject Java class must implement the Java interface *java.io.Serializable.*

sqldata — Specifies that conversions between Java objects and SQL representation is performed as specified by the Java interface *java.sql.SQLData*, as defined in JDBC 2.0. The method *java.sql.SQLData.writeSQL()* is effectively used to convert a Java object to an SQL representation, and the method *java.sql.SQLData.readSQL()* is effectively used to convert an SQL representation to a Java object..

If **sqldata** is specified, then the subject Java class must implement the Java interface *java.sql.SQLData.*

as sql_representation — Specifies the SQL attributes of the subject SQL datatype.

instantiable — Specifies that instances of the subject SQL datatype may be constructed. See section 5.5, "SQLJ method call".

not instantiable — Specifies that instances of the subject SQL datatype may not be constructed. See section 5.5, "SQLJ method call".

final — Specifies that no create_type_statement may specify the sql_datatype_name of the subject SQL datatype in the **under** clause.

not final — Specifies that create_type_statements may specify the sql_datatype_name of the subject SQL datatype in the **under** clause.

sql_predefined_type **—** An SQL predefined type.

reference_type_specification — Specifies how references (or object identifiers) are generated when the subject SQL datatype is the target type of a reference. References can be system-generated, derived from one or more attributes of the subject SQL datatype, or user-generated. For user-generated references, a predefined type is specified as the representation type of the reference, and **cast** functions can be specified that cast between the reference type and its representation type.

attribute_spec — Specifies an attribute of the subject SQL datatype.

The *sql_attribute_name* is the SQL name of the attribute.

The *sql_datatype* is the datatype of the attribute.

If the *interface_spec* is explicitly or implicitly **serializable**, then each *attribute_spec* must specify the *java_field_name*. The java_field_name is referred to as the *corresponding Java field name* of the sql_attribute_name. Rules for the corresponding Java field name of an sql_attribute_name are specified in the *Description* below.

method_spec_list — Specifies the methods of the subject SQL datatype.

method_spec — Specifies a method of the subject SQL datatype.

The *sql_method_name* is the SQL name of the method.

function_method_spec — A method of the SQL type that corresponds to a method of the java class. I.e. a method that is not a *static_field_method_spec*.

self as result — Specifies that the SQL method has a result type that is the subject SQL datatype, and that the result of a call of the method will be a copy of the state of the instance after completion of the call. See the *Description* below and section 4.3.18, “Instance-update methods”.

method_spec_characteristic — Specifies properties of the SQL method.

data_access_indication — Specifies the SQL facilities that the Java method is allowed to perform. The restrictions apply directly to the specified method itself and to any methods that it invokes, directly or indirectly.

If you don't specify a data_access_indication, then **contains sql** is the default.

no sql — The method cannot invoke SQL operations.

contains sql — The method can invoke SQL operations, but cannot read or modify SQL data. I.e. the method cannot perform SQL **open**, **close**, **fetch**, **select**, **insert**, **update**, or **delete** operations. The **contains sql** option is the default *data_access_indication*.

reads sql data — The method can invoke SQL operations, and can read SQL data, but cannot modify SQL data. I.e. the method cannot perform SQL **insert**, **update**, or **delete** operations.

modifies sql data — The method is allowed to invoke SQL operations and to read and modify SQL data.

deterministic — Specifies that for a given set of argument values, the method always returns the same result. The implementation is therefore permitted to retain lists of argument and result values from an invocation of the method, and to return those result values for subsequent invocations that specify the same argument values without executing the method on those subsequent invocations.

not deterministic — Specifies that the method does not have the **deterministic** property. This is the default, if neither **deterministic** nor **not deterministic** is specified.

returns null on null input — Specifies the action to be taken for an invocation of the method that specifies a null member_reference or any null **in** or **inout** parameter. See section 5.5, "SQLJ method call".

called on null input — Specifies the action to be taken for an invocation of the method that specifies any null **in** or **inout** parameter. See section 5.5, "SQLJ method call".

java_method_name — Specifies the method of the subject Java class that the *sql_method_name* references. The *java_method_name* is referred to as the corresponding Java method name of the *sql_method_name*.

Rules for the corresponding Java method name of an *sql_method_name* are specified in the *Description* below.

static — Specifies that the method is *static*.

If static is specified, then the method is called a *type method*. If static is not specified, then the method is called an *instance method*.

Type methods are referenced by qualification on the type name. Instance methods are referenced by qualification on instance expressions.

static_field_method_spec — A static method of the SQL type that returns the value of the Java static field specified in the **external variable name** clause. This is a shorthand that provides read-only SQL access to static fields of the Java class.

sql_parameter — Specifies a parameter of the routine.

Description

1) The rules for the sql_datatype_name, sql_attribute_names, and sql_method_names are as specified in SQL.
2) The character set supported, and the maximum length of the class_name, the java_field_name, and java_method_name, are implementation-defined.
3) Let SDT be an external Java datatype, and JC be the subject Java class of SDT.
4) The Java class JC can be the subject Java class of other external Java datatypes. Each such external Java datatype is a distinct datatype.
5) The Java class JC must be a *public* class.
6) The Java class JC must implement the Java interface *java.io.Serializable* or the Java interface *java.sql.SQLData* or both.
7) If SDT specifies an SQL supertype SSDT, then:
   a) The SQL datatype SSDT must be an external Java datatype.
   b) The Java class JC must be an immediate subclass of the subject Java class of SSDT.
8) If the *interface_spec* is explicitly or implicitly **serializable**, then for each *attribute_spec*, AS, of SDT:
   a) Let JFN be the java_field_name of AS.
   b) JFN must be the name of a field of JC or a superclass of JC. Let JF be that field of JC or a superclass of JC that would be referenced by Java name resolution for dynamic fields.
   c) JF must not be the subject field of any other attribute_spec of SDT, and if **under** is specified, then JF must not be the subject field of any supertype of SDT.
   d) JF must be a *public* field.
   e) Let SAT be the sql_datatype of AS, and JFT be the Java datatype of JF.
   f) SAT and JFT must be *simply mappable* or *object mappable*, as defined in the section "*CREATE PROCEDURE/FUNCTION Statement*" of "*SQLJ: SQL Routines*".
   g) JF is the *subject field* of attribute AS.
9) For each *function_method_spec*, FMS, of SDT:
   a) Let JMN be the java_method_name of FMS.

b) If FMS specifies **self as result**, then

   i) FMS must not specify **static**.

   ii) The returns datatype of FMS must be the subject SQL datatype.

c) If JMN is the same as JC, then the sql_method_name must be the subject SQL datatype name.

   Note: This restriction retains the characteristic that constructor methods have the same name as the type.

d) Let SS be the sql_signature of FMS.

e) If FMS specifies a Java signature, then let JS be that Java signature. Otherwise, a Java signature, JS, is determined from SS as specified in the *Description* section of the **create procedure/function** statement of *SQLJ: SQL Routines.*

f) The method name JMN and Java signature JS must identify exactly one Java method in class JC or the supertypes of class JC, using Java overloading resolution. Let JM be that Java method. JM must be visible.

g) If FMS specifies **static** then JM must be static. If FMS does not specify **static**, then JM must not be static.

h) JM is the *subject Java method* of FMS.

i) If FMS does not specify **self as result,** then:

- Let SFR be the **returns** sql_datatype of FMS. Let JFR be the Java return datatype of JM.
- SFR and JFR must be *simply mappabble* or *object mappable,* as defined in the section "*CREATE PROCEDURE/FUNCTION statement*" of "*SQLJ: Routines*".

10) For each *static_field_method_spec*, SFMS, of SDT:

a) Let JFN be the java_field_name of SFMS. Let FI be the identifier specified in JFN. If JFN specifies a java_class_name, then let SFC be that class name; otherwise, let SFC be JC.

b) FI must be the name of a field of SFC. Let JSF be that field.

c) JSF must be a *public static* field.

d) Let SRT be the sql_datatype specified in the **returns** clause of SFMS. Let JFT be the Java datatype of JSF.

e) SRT and JFT must be *simply mappable* or *object mappable*, as defined in the section "*CREATE PROCEDURE/FUNCTION Statement*" of "*SQLJ: SQL Routines*".

f) JSF is the *subject static field* of SFMS.

11) JC may contain fields and methods (public and private) for which no corresponding attribute or method is specified in SDT.

12) The subject SQL datatype initially has an ordering form of *none*.

Note: The *ordering form* of a datatype indicates what comparisons and ordering operations are allowed for instances of the datatype. The initial default ordering form is *none*. Other ordering forms are specified with the **create ordering** statement. See section 4.3.21, "Ordering of SQLJ data".

13) A **create type** statement specifying a subject SQL datatype ST and a subject Java class JC implicitly extends the datatype mappings defined in the JDBC mapping tables. A row *(ST, JC)* is added to the table "*JDBC Types Mapped to Java Object Types*", and a row *(JC, ST)* is added to the table "*Java Object Types Mapped to JDBC Types*".

Privileges

The privilege rules are those of the SQL **create type** statement.

Optional features

1) The **create type** statement can be specified either within a deployment descriptor file, as an SQL DDL statement, or by an implementation-defined mechanism that achieves the same effect as the **create type** statement. An implementation must support one or more of these techniques. It is implementation-defined which of these techniques an implementation supports.
2) If the implementation does not support overloading, then the sql_method_name of a method_spec must not be the same as the sql_method_name of any other method_spec in the same **create type** statement.
3) Support of the static_field_method_spec is optional.
4) Support of the reference_type_specification is optional.
5) An implementation must support an interface_spec of **serializable** or **SQLData** or both.

5.2 CREATE ORDERING statement

Function

Specify an ordering for a Java-SQL datatype.

Syntax

create_ordering_statement ::=
    create ordering for sql_datatype_name ordering_form
ordering_form ::=
    equals only by ordering_category
    | **order full by** ordering_category
ordering_category ::=
    map with ordering_routine
    | relative with ordering_routine
    | relative with comparable interface
    | **state**
ordering_routine ::=
    function sql_function_name
    | **method** sql_method_name
sql_datatype_ name ::= [[identifier1.]identifier2.]identifier3
sql_function_ name ::= [[identifier1.]identifier2.]identifier3
sql_method_name ::= See section 5.1, *"CREATE TYPE statement*"

Definitions and Rules

*sql_datatype_nam*e — The qualified SQL name of an SQL datatype. That SQL datatype is referred to as the *subject SQL datatype*.

ordering_form — Specifies the ordering properties of the subject SQL datatype.

Rules for the ordering_form are specified in the *Description* below.

ordering_category — Specifies the function that performs the **map** transformation or the **relative** comparison for the subject SQL datatype.

Rules for the ordering_category are specified in the *Description* below.

Description

1) The rules for the sql_datatype_name and sql_function_names are as specified in SQL.

2) The subject SQL datatype must be an external Java datatype. Let JC be the subject Java class of that external Java datatype.
3) The current *ordering form* of the subject SQL datatype must be *none*.
4) If **map with function** is specified, then:
   a) Let MF be the sql_function_name specified in the **map** clause.
   b) There must be exactly one function MF that has exactly one parameter, and whose parameter datatype is the subject SQL datatype.
   c) The result datatype of MF must be a predefined SQL datatype.
   d) For any given instances X and Y of the subject SQL datatype, and any ordering relation R:

      X R Y **if and only if** MF(X) R MF(Y)
5) If **map with method** is specified, then:
   a) Let MM be the sql_method_name specified in the **map** clause.
   b) The subject datatype must have exactly one method M that has no parameters.
   c) The result datatype of MM must be a predefined SQL datatype.
   d) For any given instances X and Y of the subject SQL datatype, and any ordering relation R:

      X R Y **if and only if** X.MM( ) R Y.MM( )
6) If **relative with function** is specified, then:
   a) Let RF be the sql_function_name specified in the **map** clause.
   b) There must be exactly one function RF that has exactly two parameters, and whose parameter datatypes are both the subject SQL datatype.
   c) The result datatype of RF must be SQL INTEGER.
   d) For any given instances X and Y of JC:

      X < Y **if and only if** RF(X, Y) = -1

      X = Y **if and only if** RF(X, Y) = 0

      X > Y **if and only if** RF(X, Y) = 1
7) If **relative with method** is specified, then:
   a) Let RM be the sql_method_name specified in the **map** clause.
   b) The subject SQL datatype must have exactly one method RM that has exactly one parameter, whose parameter datatype is the subject SQL datatype.
   c) The result datatype of RM must be SQL INTEGER.
   d) For any given instances X and Y of JC:

      X < Y **if and only if** X.RM(Y) = -1

      X = Y **if and only if** X.RM(Y) = 0

      X > Y **if and only if** X.RM(Y) = 1
8) If **relative with comparable** interface is specified, then:

a) JC must implement the Java interface *java.lang.Comparable*. That Java interface requires an implementing Java class to have a method named *compareTo*, whose result datatype is Java *int*.

b) For any given instances X and Y of JC:

X < Y **if and only if** X.compareTo(Y) = -1

X = Y **if and only if** X.compareTo(Y) = 0

X > Y **if and only if** X.compareTo(Y) = 1

9) If **state** is specified, then

a) Let A1, A2,…, An be the sql_attribute_names specified in the attribute_specs, in order.

b) Each Ai must be the name of an attribute of the subject SQL datatype.

c) For each attribute Ai, let Di be the datatype of Ai.

i) If the create_ordering_statement specifies **order equals only**, then no Di that is an external Java datatype may have an ordering form of *none*.

ii) If the create_ordering_statement specifies **order full**, then each Di that is an external Java datatype must have an ordering form of *order full*.

d) For any given instances X and Y of JC, and any ordering relation R:

X R Y **if and only if** (X.A1, X.A2,…, X.An) R (Y.A1, Y.A2, …, Y.An)

Optional features

1) The **create ordering** statement is optional.

5.3 DROP TYPE statement

Function

Drop an external-Java datatype.

Syntax

drop_datatype_statement ::=
    drop type sql_datatype_name **restrict**

Definitions and Rules

sql_datatype_name — The SQL name that is the name of an SQL datatype. That SQL datatype is referred to as the *subject SQL datatype*.

Description

1) The SQL rules for **restrict** are enforced.
2) The definition of the subject SQL datatype is deleted from the SQL system catalogs.

Privileges

1) The current user must be the owner of the subject SQL datatype.

Optional Features

1) The **drop type** statement can be specified either within a deployment descriptor file, as an SQL DDL statement, or by an implementation-defined mechanism that achieves the same effect as the **drop type** statement. An implementation must support one or more of these techniques. It is implementation-defined which of these techniques an implementation supports.

5.4 SQLJ member references

Function

Reference a field or method of a class instance or a method of a class.

Syntax

member_reference ::=
    instance_expression**.**member_name
    | sql_datatype_name**::**sql_method_name
    | reference_expression**->**member_name
instance_expression ::=
    sql_expression
    | member_reference
reference_expression ::=
    sql_expression
member_name ::= sql_attribute_name | sql_method_name

Definitions and Rules

member_reference — An expression that denotes a field or method of a class instance or a method of a class.

instance_expression — An expression whose datatype is an instance of an external Java datatype. The member_reference is a reference to a method or field of the given instance.

reference_expression — An expression whose datatype is an SQL reference type.

sql_expression — An SQL expression whose datatype is an external Java datatype.

sql_datatype_name — An SQL datatype name. This must be an SQL datatype that is an external Java datatype.

sql_method_name — The name of a static method of the external Java datatype denoted by the sql_datatype_name.

member_name — The name of an attribute or method of the class instance denoted by the instance_expression.

Description

1) Case:

a) If a member_reference immediately contains an instance_expression, then the datatype or signature of the member_reference is the datatype or signature of the attribute or method of the instance denoted by the instance_expression whose name is the member_name.

b) If a member_reference immediately contains an sql_datatype_name, then the signature of the member_reference is the signature of the method of the datatype denoted by the sql_datatype_name whose name is the sql_method_name .

c) If a member_reference immediately contains a reference_expression, then the datatype or signature of the member_reference is the datatype or signature of the attribute or method of the instance denoted by the reference_expression whose name is the member_name.

2) If the instance_expression or reference expression of a member_reference whose *member_name* is a *field_name* is a null instance value, then:

   a) If the member_reference is the target of a data transfer in a **fetch**, **select** or **update** command, or as the argument of an **output** parameter in a procedure call, then an exception is raised.

   b) Otherwise, the member_reference has the null value.

3) If a member_reference specifies an instance_expression, then:

   a) If the **create type** statement that defined the SQL type of the instance_expression implicitly or explicitly specified **serializable**, then Java serialization is effectively used to obtain a Java object from the value of the instance_expression, and the Java field that corresponds to the attribute specified in the member_name is accessed.

   b) If the **create type** statement that defined the SQL type of the instance_expression implicitly or explicitly specified **SQLData,** then the member of the instance expression is directly accessible by the SQLJ implementation and its value is returned as the value of that member_reference.,

4) The "**.**" qualification takes precedence over any operator, such as "+", "=", etc. For example, an expression such as

```
X.A1.B1 + X.A1.B2
```

   In such an expression, the plus operation is performed after the members have been referenced.

Optional features

1) Support for reference_expression and the "->" operator is optional..

5.5 SQLJ method call

Function

Invoke a method of an instance of an external Java type. A method call can be used wherever an SQLJ function call can be used.

Syntax

```
method_call ::=
        member_reference ([parameters])
      | new sql_datatype_name ([parameters])
parameters ::=  parameter [{, parameter}…]
parameter ::=  expression
```

Definitions and Rules

method_call — Invocation of a static or dynamic method or a datatype constructor.

member_reference — A member reference that denotes a method.

parameters — The list of parameters to be passed to the method. If there are no parameters, then the empty parentheses must be included.

Actions

1) If the member_reference immediately contains an sql_datatype_name, then let T be that datatype. Otherwise, let T be the datatype of the instance_expression immediately contained in the member_reference.
2) If **new** is not specified, then:
   a) SQL overloading rules are applied to the non-constructor methods of datatype T, the member_name MN, and the number and datatypes of the arguments to identify a particular method_spec MS.
   b) If MS is a static_field_method_spec, then:
      (1) Let SSF be the *subject static field* of MS.
      (2) Return the value of SSF as the result of the SQLJ method call.
      (3) Do not perform the remaining actions of this section.
   c) Let JM be the subject Java method of MS.
3) If **new** is specified, then:
   a) The create_type_statement for T must not specify **not instantiable**.

b) SQL overloading rules are applied to the constructor methods of datatype T, the sql_datatype_name, and the number and datatypes of the arguments to identify a particular method_spec MS.

c) Let JM be the subject Java method of MS.

d) JM must be a constructor method. That method constructs a new instance of the specified SQLJ class in the Java VM and returns a reference JR to that new instance.

e) The Java object referenced by JR is effectively converted to an SQL representation using the interface specified in the explicit or implicit **using** clause of the **create type** statement for T.

4) If the value of the member_reference is null (i.e. a reference to a member of a null instance), then the result of the invocation is null.

5) If MS specifies **returns null on null input**, then if the runtime value of any **in** or **inout** argument is null, return a null value as the result of the function. In this case the following steps of this *Description* do not apply to this call of the function.

6) If MS specifies **called on null input**, or specifies neither **returns null on null input** nor **called on null input**, then for each parameter of JF whose Java datatype is **boolean**, **byte**, **short**, **int**, **long**, **float**, or **double**, if the runtime value of the corresponding argument is an SQL null, then an exception is raised: *Java execution—invalid null*.

7) The SQL object identified by the instance_expression of the member_reference and all of the parameters of the member_reference are effectively converted to a Java representation. For the instance_expression and all parameters whose types are SQLJ types, the conversion is performed as specified by the Java interface specified in the **create type** statement that the defined the SQLJ type.

8) Execute the Java method JM.

a) Whether this execution is performed with the user-name of the user who created the **create function** statement CF, or with the user-name of the current user is implementation defined.

b) The scope and persistence of any modifications of static variables that are made during the execution is implementation-dependent.

c) If an SQL exception is raised during this execution, then the effect on the outermost containing SQL statement execution is implementation-defined.

Note: For portability, a java method executed in an SQL system should re-throw any SQL exception that it catches.

9) If the method execution completes with an uncaught Java exception, E, then:

a) An SQL exception is raised with the SQLSTATE value specified in section *7.2, "SQLSTATE"*.

b) Perform no further actions for the function call.

10) Case:

a) If MS does not specify **self as result**, then return the value of the method execution as the value of the method_call.

b) If MS specifies **self as result**, then let SI be the value of the instance_expression of the member_reference of the method_call. Return the state of SI after the method execution as the value of the method_call.

11) If the method_call resulted in a Java object that corresponds to an SQLJ type, then the resulting Java object is effectively converted to an SQL representation as specified by the Java interface specified by the explicit or implicit **using** clause of the **create type** for T.

Optional Features

1) Support of the syntax "**new** sql_datatype_name([parameters])" is optional. If it is not supported, then the mechanism used to invoke a constructor is implementation-defined.

6. JAVA TOPICS

6.1 Deployment descriptor files

Function

Supply information for actions to be taken by the **sqlj.install_jar** and **sqlj.remove_jar** procedures.

Model

As specified in *SQLJ: SQL Routines*, a deployment descriptor file is a text file contained in a jar file, which is specified with the following property in the manifest for the jar file:

Name: file_name
SQLJDeploymentDescriptor: TRUE

Properties

1) As specified in *SQLJ: SQL Routines*, the text contained in a deployment descriptor file must have the following form:

descriptor_file ::=
SQLActions [ ] = { [**"**action_group**"** [**,** **"**action_group**"**]] **}**
action_group ::= install_actions | remove_actions
install_actions ::=
BEGIN INSTALL [command **;**]…END INSTALL
remove_actions ::=
BEGIN REMOVE [command **;**]…END REMOVE
command ::= sql_statement | implementor_block
sql_statement ::= --*See below*
implementor_block ::=
BEGIN implementor_name sql_token… END implementor_name
implementor_name ::= sql_identifier
sql_token ::= --*See below*

2) In addition to the sql_statements specified in *SQLJ: SQL Routines*, an sql_statement specified in an install_actions must be either:

   a) A **create type** statement that specifies **external**…**language java**. The types created by those statements are called the *deployed types* of the deployment descriptor file.

   b) A **grant** statement that specifies the usage privilege for a deployed type.

   c) A **create ordering** statement that specifies ordering for a deployed type.

3) In addition to the sql_statements specified in *SQLJ: SQL Routines*, an sql_statement specified in a remove_actions must be either:

   a) A **drop type** statement for a deployed type.

   b) A **revoke** statement for the **usage** privilege on a deployed type.

7. STATUS CODES

7.1 Class and subclass values for uncaught Java exceptions

When the execution of a Java method completes with an uncaught Java exception, E, then:

1) Let EM be the result of the Java method call "E.getMessage( )"
2) EM is the message test associated with the SQL exception.
3) Case:
   a) If the class of E is *java.sql.SQLException*, then let SS be the result of the Java method call "E.getSQLState()":
      i) If the length of SS is 5 or more, and the first two characters of SS are "38", and the third, fourth, and fifth characters are not "000", then let C be "38" and let SC be the third, fourth, and fifth characters of SS.
      ii) Otherwise, let C be "39" and SC be "001".
   b) If the class of E is not *java.sql.SQLException*, then let C be "38" and SC be "000".
4) C and SC are the class and subclass of the SQLSTATE for the SQL exception.

7.2 SQLSTATE

The SQLSTATE class and subclass values for *SQLJ: SQL Routines* and *SQLJ: SQL Types* facilities are as follows:

Condition	Class	Subcondition	Subclass
Java DDL	46	***Invalid URL***	001
Java DDL	46	***Invalid jar name***	002
Java DDL	46	***Invalid class deletion***	003
Java DDL	46	***Invalid jar name***	004
Java DDL	46	***Invalid replacement***	005
Java DDL	46	***Invalid grantee***	006
Java DDL	46	***Invalid signature***	007
Java DDL	46	***Invalid method specification***	008
Java DDL	46	***Invalid REVOKE***	009
Java execution	46	***Invalid null value***	101
Java execution	46	***Invalid jar name in path***	102
Java execution	46	***Unresolved class name***	103
Java execution	46	***Too many result sets***	104
Uncaught Java exception	38	(no subclass)	000
User-defined (see above)	38	User-defined (see above)	mmm

Table 1: *SQLSTATE class and subclass values*