ISO/IEC JTC 1/SC 32 N 0630

Date: 2001-03-19

REPLACES: --

ISO/IEC JTC 1/SC 32

Data Management and Interchange

Secretariat: United States of America (ANSI)
Administered by Pacific Northwest National Laboratory on behalf of ANSI

DOCUMENT TYPE	Working Draft Text
TITLE	ISO/IEC WD 9075-13 Database Language SQL --- Part 13: Routines and Types using the Java™ Programming Language (SQL/JRT)
SOURCE	Jim Melton (Editor)
PROJECT NUMBER	1.32.03.05.13.00
STATUS	This will be discussed in the WG 3 meetings.
REFERENCES	
ACTION ID.	FYI
REQUESTED ACTION	Comments for an editing meeting are requested
DUE DATE	
Number of Pages	194
LANGUAGE USED	English
DISTRIBUTION	P & L Members SC Chair WG Conveners and Secretaries

Douglas Mann, Secretariat, ISO/IEC JTC 1/SC 32

Pacific Northwest National Laboratory *, 901 D Street, SW., Suite 900, Washington, DC, 20024-2115, United States of America

Telephone: +1 703 575 2114; Facsimile; +1 703 671 9180; E-mail: MannD@battelle.org

available from the JTC 1/SC 32 WebSite http://www.jtc1sc32.org/

*Pacific Northwest National Laboratory (PNL) administers the ISO/IEC JTC 1/SC 32 Secretariat on behalf of ANSI

H2-2001-023R1
WG3:E3A-002R1
SQL/JRT 2001-01-30

(Working Draft)

Database Language SQL --- Part 13:

Routines and Types

using the Java™ Programming Language (SQL/JRT)

January 30, 2001

This document is a preliminary draft base document for the project proposed in H2-2000-330, "Project split proposal for SQLJ Parts 1 and 2 (SQL/JRT)". The document is a merge of the following:

- H2-2000-266r1, "*SQLJ—Part 1: SQL Routines using the Java™ Programming Language"*
- H2-2000-344, *"SQLJ—Part 2: SQL Types using the Java™ Programming Language"***.**

TABLE OF CONTENTS

Foreword

(This foreword is not a part of American National Standard ANSI/ISO/IEC 9075-J:200J.)

This Standard (American National Standard ANSI/ISO/IEC 9075-J: 200J, *Information Systems —Routines and Types using the Java™ Programming Language(SQL/JRT)*), replaces NCITS 331.1, *SQLJ – Part 1: SQL Routines using the Java™ Programming Language* and NCITS 331.2, *SQLJ – Part 2: SQL Types using the Java™ Programming Language.*

This Americal National Standard specifies the ability to call Java static methods as SQL stored procedures and user-defined functions, and to use Java classes as SQL user-defined data types.

ANSI/ISO/IEC 9075 consists of the following parts, under the general title *Information Systems – Database Language ---SQL:*

- Part 1: Framework (SQL/Framework)
- Part 2: Foundation (SQL/Foundation)
- Part 3: Call-Level Interface (SQL/CLI)
- Part 4: Persistent Stored Modules (SQL/PSM)
- Part 5: Host Language indings (SQL/Bindings)
- Part 10: Object Level Bindings (SQL/OLB)
- Part J: Routines and Types using the Java™ Programming Language (SQL/JRT)

This American National Standard contains the following Informative Annexes that are not considered part of the Standard:

- Annex (informative): Routines Tutorial
- Annex (informative): Types Tutorial
- Annex (informative): SQL conformance summary
- Annex (informative): Implementation-defined elements
- Annex (informative): Implementation-dependent elements
- Annex (informative): Incompatibilities with NCITS 331.1 and 331.2

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, 200J.

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 311.1-1999 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) The clause "*Scope*" specifies the scope of this part of ISO/IEC 9075.

2) The clause "*Normative references*" identifies additional standards and publicly-available specifications that, through reference in this part of ISO/IEC 9075, constitute provisions of this part of ISO/IEC 9075.

3) The clause "*Definitions, notations, and conventions*" defines the notations and conventions used in this part of ISO/IEC 9075.

4) The clause "*Concepts*"presents concepts used in the definition of SQL/JRT.

5) The clause "*Lexical elements*"defines the lexical elements of SQL/JRT.

6) The clause "*Predicates*"specifies predicate extensions for SQL/JRT.

7) The clause "*Additional Common elements*"describes a number of elements used in the definition of SQL/JRT.

8) The clause "*Schema definition and manipulation*" defines the schema definition and manipulation statements associated with the definition of SQL/JRT.

9) The clause "*Access control*" defines new SQL statement extensions, and new rules for SQL function and procedure calls.

10) The clause "*Built-in procedures*"defines new built-in procedures for SQL/JRT.

11) The clause "*Java topics*" defines the Java facilities supported by implementations of SQL/JRT and the conventions used in deployment descriptor files.

12) The clause "*Definition schema*" defines extensions to the definition schema for SQL/JRT.

13) The clause "*Status codes*" defines SQLSTATE values related to SQL/JRT.

14) The clause "*Conformance*" defines the criteria for conformance to this part of ISO/IEC 9075.

15) The clause "*Annex(Informative) – Routines tutorial*" is an Informative Annex. It describes SQL/JRT features for defining and using SQL stored procedures based on Java static methods.

16) The clause "*Annex(Informative) – Types tutorial*" is an Informative Annex. It describes SQL/JRT features for defining and using SQL data types based on Java classes.

17) The clause "*Annex(Informative) – SQL/JRT feature taxonomy*", is an Informative Annex. It lists those features that are not required in conforming implementations of this part of ISO/IEC 9075.

18) The clause "*Annex(Informative) – Implementation-defined elements*" is an Informative Annex. It lists those features for which the body of this part of ISO/IEC 9075 states that the syntax, the meaning, the returned results, the effect on SQL-data and/or schemas, or any other behavior is partly or wholly implementation-defined.

19) The clause "*Annex(Informative) – Implementation-dependent elements*", is an Informative Annex. It lists those features for which the body of this part of ISO/IEC 9075 states that the syntax, the meaning, the returned results, the effect on SQL-data and/or schemas, or any other behavior is partly or wholly implementation-dependent.

20) The clause "*Annex(Informative) – Incompatibilities with NCITS 331.1 and 331.2*", is an Informative Annex. It lists the incompatibilities between this part of ISO/IEC 9075 and NCITS 331.1 and 331.2.

1. SCOPE

This standard specifies the manner in which SQL routines and types 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.

5) *JDBC™ Database Access with Java™: A tutorial and annotated reference,* Graham, Hamilton, Cattell, and Fisher, Addison-Wesley, 1997.

6) *W3C Architecture domain: Naming and Addressing (URLs*), `http://www.w3.org/Addressing/Activity.html`

7) NCITS 331.1, *SQLJ – Part 1: SQL Routines using the Java™ Programming Language*

8) NCITS 331.2, *SQLJ – Part 2: SQL Types using the Java™ Programming Language*

3. DEFINITIONS, NOTATIONS, AND CONVENTIONS

3.1 Definitions

3.1.1 Definitions provided in Part 13

<Insert this paragraph>For the purposes of this part of ISO/IEC 9075, the definitions given in ISO/IEC 9075-1, ISO/IEC 9075-2, ISO/IEC 9075-5, and ISO/IEC 9075-10 and the following definitions apply.

a) **default connection:** An SQL-connection to the current SQL-implementation, SQL-session, and SQL- transaction established with the data source URL 'jdbc:default:connection'.

b) **deployment descriptor file:** a text file, contained in a JAR, that contains *deployment descriptor*s, which specify actions to be taken by the SQLJ.INSTALL_JAR and SQLJ.REMOVE_JAR procedures. E.g., when a JAR is installed one or more CREATE PROCEDURE/FUNCTION statements and associated GRANT statements can be specified in the deployment descriptors and executed as part of the install process.

c) external Java data type: An SQL User-defined type defined with a CREATE TYPE that specifies EXTERNAL LANGUAGE JAVA.

d) **external Java routine:** An external routine defined with a CREATE PROCEDURE/FUNCTION that specifies EXTERNAL LANGUAGE JAVA.

e) **installed JAR:** A JAR whose existence has been registered with the SQL-environment and whose contents have been copied into that SQL-environment due to execution of either the procedure SQLJ.INSTALL_JAR or SQLJ.REPLACE_JAR

f) **Java Archive (JAR):** A zip formatted file containing a set of Java class and ser files, and optionally a deployment descriptor file. JARs are a normal vehicle for distributing Java programs and the mechanism chosen to provide the implementation of external Java routines and external Java data types to an SQL-environment.

g) **subject Java class:** the Java class identified by a *subject Java class name*, <Java class name>, in the JAR identified by the <jar id> in the immediately containing <jar and class name>.

h) **subject Java class name:** the fully-qualified, package and class name of a Java class, specified by <Java class name> immediately contained in <jar and class name>.

3.2 Notations

<Insert this paragraph> The syntax notation used in this part of ISO/IEC 9075 is an extended version of BNF ("Backus normal Form" or "Backus Naur Form"). This version of BNF is fully described in Subclause 6.1, "notation", of ISO/IEC 9075-1.

3.3 Conventions

<Insert this paragraph> Except as otherwise specified in this part of ISO/IEC 9075, the conventions used in this part of ISO/IEC 9075 are identical to those described in ISO/IEC 9075-1 and ISO/IEC 9075-2.

3.3.1 Use of terms

3.3.1.1 Exceptions

<Modified paragraph> The phrase "an exception condition is raised:", followed by the name of a condition, is used in General Rules and elsewhere to indicate that:

- The execution of a statement is unsuccessful.
- the application of General Rules may be terminated.
- Diagnostic information is made available.
- Execution of the statement has no effect on SQL-data or schemas.

3.3.1.2 Other terms

To be added.

3.3.2 Relationships to other parts of ISO/IEC 9075

3.3.2.1 Clause, subclause, and table relationships

To be added.

Table 3-1 Clause, Subclause, and table relationships

Clause, Subclause, or	Corresponding Clause,	Part containing

Definitions, notations, and conventions

Table in this part of ISO/IEC 9075	Subclause, or Table from another part	correspondence
Clause 1, "Scope"	Clause 1, "Scope"	ISO/IEC 9075-2
Clause 2, "Normative references"	Clause 2, " Normative references "	ISO/IEC 9075-2
Clause 3, "Definitions, notations, and conventions"	Clause 3, "Definitions, notations, and conventions"	ISO/IEC 9075-2
Subclause 3.1, "Definitions"	Subclause 3.1, " Definitions"	ISO/IEC 9075-2
Subclause 3.1.1, "Definitions provided in Part J"	Subclause 3.1, " Definitions provided in Part J"	(none)
Subclause 3.3, "Conventions"	Subclause 3.3, " Conventions"	ISO/IEC 9075-2
Subclause 3.3.1, "Use of terms"	Subclause 3.3.1, " Use of terms"	ISO/IEC 9075-2
Subclause 3.3.1.1,"Exceptions"	Subclause 6.2.3.1, "Exceptions"	ISO/IEC 9075-1
Subclause 3.3.1.2,"Other terms"	Subclause 6.2.3.7, " Other terms"	ISO/IEC 9075-1
Subclause 3.3.2, "Relationship to other parts of ISO/IEC 9075"	(none)	(none)
Subclause 3.3.2.1, "Clause, Subclause, and Table relationships"	(none)	(none)
Subclause 3.4, "Object identifier for Database Language SQL"	Subclause 6.3, "Object identifier for Database Language SQL"	ISO/IEC 9075-1
Clause 4, "Concepts"	Clause 4, "Concepts"	ISO/IEC 9075-2
Clause 4.1 The Java™ programming language		

Clause 4.2 SQL-invoked routines Clause 4.3, "Java class name resolution" Clause 4.4 SQL result sets Clause 4.5 Parameter mapping Clause 4.6 Unhandled Java exceptions	Clause 4.23 SQL-invoked routines	ISO/IEC 9075-2
Clause 4.7 Data types	Clause 4.1 Data types	ISO/IEC 9075-2
Clause 4.8 User-defined types	Clause 4.8 User-defined types	ISO/IEC 9075-2
Clause 4.8.1 Observers and mutators	Clause 4.8.1 Observers and mutators	ISO/IEC 9075-2
Clause 4.8.2 Constructors	Clause 4.8.2 Constructors	ISO/IEC 9075-2
Clause 4.8.3 Subtypes and supertypes	Clause 4.8.3 Subtypes and supertypes	ISO/IEC 9075-2
Clause 4.8.4 User-defined type comparison and assignment	Clause 4.8.4 User-defined type comparison and assignment	ISO/IEC 9075-2
Clause 4.8.5 Transforms for user-defined types	Clause 4.8.5 Transforms for user-defined types	ISO/IEC 9075-2

3.4 Object identifier for Database Language SQL

(To be added.)

4. CONCEPTS

4.1 The Java™ programming language

The Java™ programming language is a class-based, object-oriented language. The SQL/JRT standard uses the following Java concepts and terminology

A *class* is the basic element of Java programs. A class consists of a set of variable declarations and methods.

A variable is local to a class, to instances of the class, or to a method. A variable that is declared *static* is local to the class. Other variables declared in the class are local to instances of the class. Those variables are called fields of the class. A variable declared in a method is local to the method.

A *class instance* consists of an instance of each of the fields of the class. Class instances are strongly typed by the class name.

A *subclass* is a class that is declared to *extend* (at most) one other class. That other class is called the *direct superclass* of the subclass. A subclass has all of the variables and methods of its direct and indirect superclasses, and may be used interchangeably with them.

A *method* is an executable routine. A method can be declared *static*, in which case it is called a *class method*. Otherwise, it is called an *instance method*. Class methods can be referenced by qualifying the method name with either the class name or the name of an instance of the class. Instance methods are referenced by qualifying the method name with the name of an instance of the class. The method body of an instance method can reference the variables local to that instance.

The *signature* of a method consists of the method name, the result data type, and the number of parameters and their data types.

A *package* is a set of related classes. A class either specifies a package, or is part of an anonymous default package. A class can use *import* statements to specify other packages whose classes can be referenced.

Classes, fields, and methods can be declared as *public, private, protected, or friendly*. A *public* variable or method can be accessed by any method. A *private* variable or method can only be referenced by methods in the same class. A *protected* variable or method can only be referenced by methods of the same class or subclasses thereof. A *friendly* method is a method that is not declared as public, private or protected. A *friendly* method can only be called by methods of other classes in the same package.

An *interface* is a Java construct consisting of a set of method signatures having no method bodies. A class can be declared to implement an interface, in which case the class is required to have methods with the signatures specified in the interface.

The Java *Serializable* interface defines a transformation between a Java instance and a byte stream containing sufficient information to identify the class of the instance and to reconstruct the instance.

The Java *SQLData* interface defines a transformation between a Java instance and an SQL user-defined data type.

The source for a Java class is normally stored in a *java file* with the file-type "java", e.g. *myclass.java*. Java is normally compiled to a byte coded instruction set that is portable to any platform supporting Java. A file containing such byte code is normally stored in a c*lass file* with the file-type "class", e.g. *myclass.class*.

A set of class files can be assembled into a *Java archive* file, or *jar* file. A jar file is a zip formatted file containing a set of Java class files. Jar files are the normal vehicle for distributing Java programs.

4.2 SQL-invoked routines

An *SQL-invoked routine* is an SQL-invoked procedure or an SQL-invoked function. An SQL-invoked routine comprises at least a <schema qualified routine name>, a sequence of <SQL parameter declaration>s, and a <routine body>.

The following 2 paragraphs are moved to immediately after the 1'st paragraph from later in this subclause.

An SQL-invoked routine can be an SQL routine or an external routine. An SQL routine is an SQL-invoked routine whose <language clause> specifies SQL. The <routine body> of an SQL routine is an <SQL procedure statement>; the <SQL procedure statement> forming the <routine body> can be any SQL-statement, including an <SQL control statement>, but excluding an <SQL schema statement>, <SQL connection statement>, or <SQL transaction statement>.

An external routine is one whose <language clause> does not specify SQL. The <routine body> of an external routine is an <external body reference> whose <external routine name> identifies a program written in some standard programming language other than SQL.

insert the following paragraphs after the above two relocated paragraphs:

External routines appear in two similar but fundamentally differing forms, where the key differentiator is whether or not the <language clause> for the external routine specifies Java. When external language Java is specified, the external routine is an *external Java routine*; some differences from other external routines include:

- For any other external routine the implementation of any method that isn't an *SQL routine* exists externally to the *SQL-environment*; for external Java routines the implementation of the methods is provided by a specified subject Java routine that exists in the *SQL-environment* in an *installed JAR.*

- Because a *subject Java routine* is not required to be completely self-contained (i.e., to have no references to classes outside of itself), a mechanism is provided to allow a *subject Java routine* to reference classes contained in other installed JARs. See the clause "SQL-Java paths."

Note xx: Once an external Java routine has been created, its use in SQL statements executed by the containing SQL-implementation is similar to that of other external routines.

Note to the Editor: end of text to be inserted.

An SQL-invoked routine is an element of an SQL-schema and is called a *schema-level routin*e.

An SQL-invoked routine *SR* is said to be *dependent* on a user-defined type *UDT* if and only if *SR* is created during the execution of the <user-defined type definition> that created *UDT*. An SQL-invoked routine that is dependent on a user-defined type may not be destroyed by a <drop routine statement>. It is destroyed implicitly by a <drop data type statement>.

A <predicate> *P* is said to be dependent on an SQL-invoked routine *SR* if and only if *SR* is the ordering function included in the user-defined descriptor of a user-defined type *UDT* and one of the following conditions is true:

— *P* is a <comparison predicate> that immediately contains a <row value expression> whose declared type is some user-defined type *T1* whose comparison type is *UD*T.

— *P* is a <quantified comparison predicate> that immediately contains a <row value expression> that has some field whose declared type is some user-defined type *T1* whose comparison type is *UD*T.

— *P* is a <unique predicate> that immediately contains a <table subquery> that has a column whose declared type is some user-defined type *T1* whose comparison type is *UD*T.

— *P* is a <match predicate> that immediately contains a <row value expression> that has some field whose declared type is some user-defined type *T1* whose comparison type is *UD*T.

— *P* is a <comparison predicate> with some corresponding value whose declared type is some array type whose element type is a user-defined type *T1* whose comparison type is *UD*T.

— *P* is a <quantified comparison predicate> that immediately contains a <row value expression> that has some field whose declared type is some array type whose element type is a user-defined type *T1* whose comparison type is *UD*T.

— *P* is a <unique predicate> that immediately contains a <table subquery> that has a column whose declared type is some array type whose element type is a user-defined type *T1* whose comparison type is *UD*T.

— *P* is a <match predicate> that immediately contains a <row value expression> that has some field whose declared type is some array type whose element type is a user-defined type *T1* whose comparison type is *UD*T.

NOTE 30 – ‘‘Comparison type’’ is defined in the clause "*User-defined type comparison and assignment*".

A <set function specification> *SFS* is said to be *dependent* on an SQL-invoked routine *SR* if and only if all the following are true:

— *SR* is the ordering function included in the user-defined descriptor of a user-defined type *UD*T.

— *SFS* is a <general set function> whose <set function type> *SFS* is MAX or MIN or *SFS* is a <general set function> whose <set qualifier> is DISTINCT.

— The declared type of the <value expression> of *SFS* is *UD*T.

A <group by clause> *GBC* is said to be *dependent* on an SQL-invoked routine *SR* if and only if all the following are true:

— *SR* is the ordering function included in the user-defined descriptor of a user-defined type *UD*T.

— The declared type of a grouping column of *GBC* is *UD*T.

An *SQL-invoked procedure* is an SQL-invoked routine that is invoked from an SQL <call statement>. An SQL-invoked procedure may have input SQL parameters, output SQL parameters, and SQL parameters that are both input SQL parameters and output SQL parameters. The format of an SQL-invoked procedure is specified by <SQL-invoked procedure> (see Subclause 11.49, ‘‘<SQL-invoked routine>’’).

An *SQL-invoked function* is an SQL-invoked routine whose invocation returns a value. Every parameter of an SQL-invoked function is an input parameter, one of which may be designated as the result SQL parameter. The format of an SQL-invoked function is specified by <SQL-invoked function> (see Subclause 11.49, ‘‘<SQL-invoked routine>’’). An SQL-invoked function can be a *type-preserving functio*n; a type-preserving function is an SQL-invoked function that has a result SQL parameter. The result data type of a type-preserving function is some subtype of the data type of its result SQL parameter.

update the following paragraph as indicated

An *SQL-invoked method* is an SQL-invoked function that is specified by <method specification designator> (see Subclause 11.49, ‘‘<SQL-invoked routine>”). There are two kinds of SQL-invoked methods: *instance SQL-invoked methods* and *static SQL-invoked method*s **further characterized by whether or not the SQL-invoked method is an external Java routine**. All SQL-invoked methods are associated with a structured type, also known as the *type of the metho*d. **If the SQL-invoked method is not an external Java routine, the ~~The~~** <routine characteristics> of **~~an~~ that** SQL-invoked method are specified by a <method specification> contained in the <user-defined type definition> of the type of the method. An instance SQL-invoked method **that is not an external Java routine** satisfies the following conditions:

— Its first parameter, called the *subject parameter*, has a declared type that is a user-defined type. The type of the subject parameter is the type of the method. A parameter other than the subject parameter is called an *additional parameter*.

— Its descriptor is in the same schema as the descriptor of the data type of its subject parameter.

insert the following paragraph

If the SQL-invoked method is an external Java routine the <routine characteristics> of that SQL-invoked method are specified by <method characteristic>s contained in the <user-defined type definition> of the type of the method. An instance SQL-invoked method that is an external Java routine satisfies the following conditions:

— It has no subject parameter. Its first parameter, if any, is treated no differently than any other parameter.

— Its descriptor is in the same schema as the descriptor of the structured type of the method.

Note to the Editor: end of text to be inserted.

update the following paragraph as indicated

A static SQL-invoked method**, whether or not it is an external Java routine,** satisfies the following conditions:

— It has no subject parameter. Its first parameter, if any, is treated no differently than any other parameter.

— Its descriptor is in the same schema as the descriptor of the structured type of the method. The name of this type (or of some subtype of it) is always specified together with the name of the method when the method is to be invoked.

An SQL-invoked function that is not an SQL-invoked method is an *SQL-invoked regular functio*n. An SQL-invoked regular function is specified by <function specification> (see Subclause 11.49, ‘‘<SQL-invoked routine>”).

A *null-call function* is an SQL-invoked function that is defined to return the null value if any of its input arguments is the null value. A null-call function is an SQL-invoked function whose <null-call clause> specifies ‘‘RETURNS NULL ON NULL INPUT’’.

The following 2 paragraphs are moved to the beginning of the subclause.
~~An SQL-invoked routine can be an *SQL routine* or an *external routine*. An SQL routine is an SQL-invoked routine whose <language clause> specifies SQL. The <routine body> of an SQL routine is an <SQL procedure statement>; the <SQL procedure statement> forming the <routine body> can be any SQL-statement, including an <SQL control statement>, but excluding an <SQL schema statement>, <SQL connection statement>, or <SQL transaction statement>.~~

~~An external routine is one whose <language clause> does not specify SQL. The <routine body> of an external routine is an <external body reference> whose <external routine name> identifies a program written in some standard programming language other than SQL.~~

An SQL-invoked routine is uniquely identified by a <specific name>, called the *specific name* of the SQL-invoked routine.

Note to Reviewers: Again, Much can be simplified as SQL/JRT and SQL/Foundation are brought closer into alignment. The following is a specific example where a lot of simplification will occur. Only the first four sentences are changed to hint at the effect of not making a more common syntax. I.e., additional changes are needed if the syntax is not better aligned. After the first 4 sentences, changes shown are needed even with better syntax alignment.

update the following paragraph as indicated
SQL-invoked routines are invoked differently depending on their form. SQL-invoked procedures are invoked by <call statement>s **of ISO/IEC 9075-2 or, when the SQL-invoked routine is an external Java routine <SQL procedure call> (see the clause "SQL/JRT procedure call”).** SQL-invoked regular functions are invoked by <routine invocation>s **of ISO/IEC 9075-2 or, when the SQL-invoked regular function is an external Java routine <SQL/JRT function call> (see the clause "SQL/JRT function call”).** Instance SQL-invoked methods are invoked by <method invocation>s **of ISO/IEC 9075-2 or, when the SQL-invoked regular method is an external Java routine <method call> (see the clause "SQL/JRT method call”),** while static SQL-invoked methods are invoked by <static method invocation>s **of ISO/IEC 9075-2 or, when the static SQL-invoked method is an external Java routine by <member reference> (see the clause "SQL/JRT member references”) and <method call> (see the clause "SQL/JRT method call”).** An invocation of an SQL-invoked routine specifies the <routine name> of the SQL-invoked routine and supplies a sequence of argument values corresponding to the <SQL parameter declaration>s of the SQL-invoked routine. A *subject routine* of an invocation is an SQL-invoked routine that may be invoked by a <routine invocation>. After the selection of the subject routine of a <routine invocation>, the SQL arguments are evaluated and the SQL-invoked routine that will be executed is selected. If the subject routine is an instance SQL-invoked method **that is not**

an external Java routine, then the SQL-invoked routine that is executed is selected from the set of overriding methods of the subject routine. (The term ‘‘set of overriding methods’’ is defined in the General Rules of Subclause 10.4, ‘‘<routine invocation>’’.) The overriding method that is selected is the overriding method with a subject parameter the type designator of whose declared type precedes that of the declared type of the subject parameter of every other overriding method in the type precedence list of the most specific type of the value of the SQL argument that corresponds to the subject parameter. See the General Rules of Subclause 10.4, ‘‘<routine invocation>’’. **If the instance SQL-invoked method is an external Java routine the term ”set of overriding methods” is not applicable; for such methods the capabilities provided by overriding methods duplicate Java’s own mechanisms and the subject routine executed is the one that would be invoked when no overriding methods are specified**. If the subject routine is not an SQL-invoked method, then the SQL-invoked routine executed is that subject routine. After the selection of the SQL-invoked routine for execution, the values of the SQL arguments are assigned to the corresponding SQL parameters of the SQL-invoked routine and its <routine body> is executed. If the SQL-invoked routine is an SQL routine, then the <routine body> is an <SQL procedure statement> that is executed according to the General Rules of <SQL procedure statement>. If the SQL-invoked routine is an external routine, then the <routine body> identifies a program written in **Java, or** some standard programming language other than SQL that is executed according to the rules of that ~~**standard**~~ programming language.

The <routine body> of an SQL-invoked routine is always executed under the same SQL-session from which the SQL-invoked routine was invoked. Before the execution of the <routine body>, a new context for the current SQL-session is created and the values of the current context preserved. When the execution of the <routine body> completes the original context of the current SQL-session is restored.

update the following paragraph as indicated

If the SQL-invoked routine is an external routine, then an effective SQL parameter list is constructed before the execution of the <routine body>. The effective SQL parameter list has different entries depending on the parameter passing style of the SQL-invoked routine. **When the SQL-invoked routine is not an external Java routine, the** ~~**The**~~ value of each entry in the effective SQL parameter list is set according to the General Rules of Subclause 10.4, ‘‘<routine invocation>’’ **of ISO/IEC 9075-**2, and passed to the program identified by the <routine body> according to the rules of Subclause 13.6, ‘‘Data type correspondences’’ **of ISO/IEC 9075-**2. **When the SQL-invoked routine is an external Java routine, the value of each entry in the effective SQL parameter list is set and passed to the program identified by <routine body> according to the General Rules of Subclause <?> "SQL/JRT procedure call’’**. After the execution of that program, if the parameter passing style of the SQL-invoked routine is SQL, then the SQL-implementation obtains the values for output parameters (if any), the value (if any) returned from the program, the value of the SQLSTATE, and the value of the message text (if any) from the values assigned by the program to the effective SQL parameter list. **If the parameter passing style of the SQL-invoked routine is JAVA, then such values are obtained according to the General Rules of Subclause <?> "SQL/JRT**

procedure call''. If the parameter passing style of the SQL-invoked routine is GENERAL, then such values are obtained in an implementation-defined manner.

Different SQL-invoked routines can have equivalent <routine name>s. No two SQL-invoked functions in the same schema are allowed to have the same signature. No two SQL-invoked procedures in the same schema are allowed to have the same name and the same number of parameters. Subject routine determination is the process for choosing the subject routine for a given <routine invocation> given a <routine name> and an <SQL argument list>. Subject routine determination for SQL-invoked functions considers the most specific types of all of the arguments to the invocation of the SQL-invoked function in order from left to right. Where there is not an exact match between the most specific types of the arguments and the declared types of the parameters, type precedence lists are used to determine the closest match. See Subclause 9.4, ''Subject routine determination''.

update the following paragraph as indicated

If a <routine invocation> is contained in a <query expression> of a view, a check constraint, or an assertion, the <trigger action> of a trigger, or in an <SQL-invoked routine>, then the subject routine for that invocation is determined at the time the view is created, the check constraint is defined, the assertion is created, the trigger is created, or the SQL-invoked routine is created. If the subject routine is an SQL-invoked procedure, an SQL-invoked regular function, or a static SQL-invoked method, then the same SQL-invoked routine is executed whenever the view is used, the check constraint or assertion is evaluated, the trigger is executed, or the SQL-invoked routine is invoked. If the subject routine is an instance SQL-invoked method, then the SQL-invoked routine that is executed is determined whenever the view is used, the check constraint or assertion is evaluated, the trigger is executed, or the SQL-invoked routine is invoked, based on the most specific type of the value resulting from the evaluation of the SQL argument that correspond to the subject parameter. See the General Rules of Subclause 10.4, ''<routine invocation>'' **of ISO/IEC 9075-2 or of Subclause <?> "SQL/JRT method call"**.

All <identifier chain>s in the <routine body> of an SQL routine are resolved to identify the basis and basis referent at the time that the SQL routine is created. Thus, the same columns and SQL parameters are referenced whenever the SQL routine is invoked.

An SQL-invoked routine is either *deterministic* or *possibly non-deterministic*. An SQL-invoked function that is deterministic always returns the same return value for a given list of SQL argument values. An SQL-invoked procedure that is deterministic always returns the same values in its output and inout SQL parameters for a given list of SQL argument values. An SQL-invoked routine is possibly non-deterministic if, during invocation of that SQL-invoked routine, an SQL-implementation might, at two different times when the state of the SQL-data is the same, produce unequal results due to General Rules that specify implementation-dependent behavior.

An external routine either *does not possibly contain SQL* or *possibly contains SQL*. Only an external routine that possibly contains SQL may execute SQL-statements during its invocation.

Concepts

An SQL-invoked routine may or may not *possibly read SQL-dat*a. Only an SQL-invoked routine that possibly reads SQL-data may read SQL-data during its invocation.

An SQL-invoked routine may or may not *possibly modify SQL-dat*a. Only an SQL-invoked routine that possibly modifies SQL-data may modify SQL-data during its invocation.

An SQL-invoked routine has a *routine authorization identifie*r, which is (directly or indirectly) the authorization identifier of the owner of the schema that contains the SQL-invoked routine at the time that the SQL-invoked routine is created.

When the <routine body> of an SQL-invoked routine is executed and the new SQL-session context for the SQL-session is created, the SQL-session user identifier in the new SQL-session context is set to the current user identifier in the SQL-session context that was active when the SQL-session caused the execution of the <routine body>. The authorization stack of this new SQL-session context is initially set to empty and a new pair of identifiers is immediately appended to the authorization stack such that:

— The user identifier is the newly initialized SQL-session user identifier.

— The role name is the current role name of the SQL-session context that was active when the SQL-session caused the execution of the <routine body>.

The identifiers in this new entry of the authorization stack are then modified depending on whether the SQL-invoked routine is an SQL routine or an external routine. If the SQL-invoked routine is an SQL routine, then, if the routine authorization identifier is a user identifier, the user identifier is set to the routine authorization identifier and the role name is set to null; otherwise, the role name is set to the routine authorization and the user identifier is set to null.

If the SQL-invoked routine is an external routine, then the identifiers are determined according to the external security characteristic of the SQL-invoked routine:

— If the external security characteristic is DEFINER, then:

- If the routine authorization identifier is a user identifier, then the user identifier is set to the routine authorization identifier and the role name is set to the null value.
- Otherwise, the role name is set to the routine authorization identifier and the user identifier is set to the null value.

— If the external security characteristic is INVOKER, then the identifiers remain unchanged.

— If the external security characteristic is IMPLEMENTATION DEFINED, then the identifiers are set to implementation-defined values.

An SQL-invoked routine that is an external routine also has an *external routine authorization identifie*r, which is the <module authorization identifier>, if any, of the <SQL-client module definition> contained in the external program identified by the <routine body> of the external routine. If that <SQL-client module definition> does not specify a <module authorization identifier>, then the external routine authorization identifier is an implementation-defined authorization identifier.

The final value of the user identifier and role name in the authorization stack are used for privilege determination for access to the SQL objects, if any, referenced in the <SQL procedure statement>s that are executed during the execution of the <routine body>.

An SQL-invoked routine has a *routine SQL-pat*h, which is inherited from its containing SQL-schema, the current SQL-session, or the containing SQL-client module.

An SQL-invoked routine that is an external routine also has an *external routine SQL-pat*h, which is derived from the <module path specification>, if any, of the <SQL-client module definition> contained in the external program identified by the routine body of the external routine. If that <SQL-client module definition> does not specify a <module path specification>, then the external routine SQL-path is an implementation-defined SQL-path. For both SQL and external routines, the SQL-path of the current SQL-session is used to determine the search order for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is contained in a <preparable statement> or in a <direct SQL statement>. SQL routines use the routine SQL-path to determine the search order for the subject routines of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is not contained in a <preparable statement> or in a <direct SQL statement>. External routines use the external routine SQL-path to determine the search order for the subject routine of a <routine invocation> whose <routine name> does not contain a <schema name> if the <routine invocation> is not contained in a <preparable statement> or in a <direct SQL statement>.

update the following paragraph as indicated

An SQL-invoked routine is described by a *routine descripto*r. A routine descriptor contains:

— The routine name of the SQL-invoked routine.

— The <specific name> of the SQL-invoked routine.

— The routine authorization identifier of the SQL-invoked routine.

— The routine SQL-path of the SQL-invoked routine.

— The name of the language in which the body of the SQL-invoked routine is written.

— For each of the SQL-invoked routine's SQL parameters, the <SQL parameter name>, if it is specified, the <data type>, the ordinal position, and an indication of

whether the SQL parameter is an input SQL parameter, an output SQL parameter, or both an input SQL parameter and an output SQL parameter.

— An indication of whether the SQL-invoked routine is an SQL-invoked function or an SQL-invoked procedure.

— If the SQL-invoked routine is an SQL-invoked procedure, then the maximum number of dynamic result sets.

— An indication of whether the SQL-invoked routine is deterministic or possibly non-deterministic.

— Indications of whether the SQL-invoked routine possibly modifies SQL-data, possibly reads SQL-data, possibly contains SQL, or does not possibly contain SQL.

— If the SQL-invoked routine is an SQL-invoked function, then:

- The <returns data type> of the SQL-invoked function.
- If the <returns data type> simply contains <locator indication>, then an indication that the return value is a locator.
- An indication of whether the SQL-invoked function is a type-preserving function or not.
- An indication of whether the SQL-invoked function is a mutator function or not.
- If the SQL-invoked function is a type-preserving function, then an indication of which parameter is the result parameter.
- An indication of whether the SQL-invoked function is a null-call function.

— The creation timestamp.

— The last-altered timestamp.

— If the SQL-invoked routine is an SQL routine, then the SQL routine body of the SQL-invoked routine.

— If the SQL-invoked routine is an external routine, then:

- The <external routine name> of the external routine.
- The <parameter style> of the external routine.

- If the external routine specifies a <result cast>, then an indication that it specifies a <result cast> and the <data type> specified in the <result cast>. If <result cast> contains <locator indication>, then an indication that the <data type> specified in the <result cast> has a locator indication.
- The external security characteristic of the external routine.
- The external routine authorization identifier of the external routine.
- The external routine SQL-path of the external routine.
- The effective SQL parameter list of the external routine.
- For every SQL parameter that has an associated from-sql function *FS*F, the specific name of *FS*F.
- For every SQL parameter that has an associated to-sql function *TS*F, the specific name of *TS*F.
- If the SQL-invoked routine is an external function and if it has a to-sql function *TRF* associated with the result, then the specific name of *TR*F.
- For every SQL parameter whose <SQL parameter declaration> contains <locator indication>, an indication that the SQL parameter is a locator parameter.

— The <schema name> of the schema that includes the SQL-invoked routine.

— If the SQL-invoked routine is an SQL-invoked function, then an indication of whether the SQL-invoked function is an SQL-invoked method.

— If the SQL-invoked routine is an SQL-invoked method, then an indication of the user-defined type whose descriptor contains the corresponding method specification descriptor.

— If the SQL-invoked routine is an SQL-invoked method, then an indication of whether STATIC was specified.

— An indication of whether the SQL-invoked routine is dependent on a user-defined type.

** end of copied text **

4.3 Java class name resolution

Typical Java environments provides a class name resolution, or search path, mechanism based on an environmental variable called CLASSPATH. When a JVM encounters a

previously unseen reference to a class, the members of the list of directories and JAR files appearing in the classpath are examined in order until either the class is found or the end of the list is reached. Failure to locate a referenced class is a runtime error which will, often, cause the application that experiences it to terminate.

When a Java environment is transitioned to being, logically, inside an SQL environment the problem of managing the JVM's class name resolution continues to exist, but with a change in emphasis. To allow the creators of Java applications a greater degree of control over class name resolution, and the added security associated with that control, a classpath-like mechanism is defined to be a property of the JARs containing the Java applications, rather than as an environmental variable of the current session e.g., CURRENT_PATH for dynamic statements. I.e., if while executing, an external Java routine encounters a previously unseen class reference, it first searches for that class in the JAR containing the definition of the currently executing class, and if it is not found the class will be searched for in the manner specified by the SQL-Java path associated with that JAR (if any). See subclause <?> "SQL-Java paths".

4.4 SQL result sets

To be added.

4.5 Parameter mapping

1) An SQL data type ST and a Java data type JT are *simply mappable* if and only if ST and JT are specified respectively in column 1 and column 2 of a row of the JDBC data mapping table "*JDBC Types mapped to Java Types*" (reference [5], table 21.1). The Java data type JT is the *corresponding Java data type* of ST.

2) An SQL data type ST and a Java data type JT are *object mappable* if and only if ST and JT are specified respectively in column 1 and column 2 of a row of the JDBC data mapping table "*JDBC Types mapped to Java Object Types*" (reference [5], table 21.3).

3) An SQL data type ST and a Java data type JT are *output mappable* if and only if JT is a one dimensional array of a data type JT2 and ST is either simply mappable or object mappable to JT2. I.e., JT is "JT2[ ]".

4) An SQL array of the SQL data type ST and a Java data type JT are *array mappable* if and only if JT is a one dimensional array of a data type JT2 and ST is either simply mappable or object mappable to JT2.

5) An SQL data type ST and a Java data type JT are *mappable* if and only if ST and JT are simply mappable, object mappable, output mappable, or array mappable.

6) A Java data type is *mappable* if and only if it is mappable to some SQL data type.

7) A Java class is *result set oriented* if and only if it is either:

   a) A class that implements the Java interface *java.sql.ResultSet*.

   b) A public class that implements the Java interface *sqlj.runtime.ResultSetIterator*. **Note:** these classes are generated by *SQL/OLB* iterator declarations ("*#sql iterator*").

8) A Java data type is *result set mappable* if and only if it is an array of a result set oriented class.

9) A Java method with M parameters is *mappable* (to SQL) if and only if for some N in [0, M], the data types of the first N parameters are mappable, the last M-N parameters are result set mappable, and the result type is either simply mappable, object mappable, or is **void**.

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

4.6 Unhandled Java exceptions

Java exceptions that are thrown during execution of a Java method in SQL can be caught, or handled, within Java and if this is done, then those exceptions do not affect SQL processing. Any Java exceptions that are uncaught when a Java method called from SQL completes will appear in the SQL-environment as SQL exception conditions.

The message text and SQLSTATE may be specified in the Java exception specified in the Java **throw** statement. If that exception specifies an SQLSTATE, the first two characters of that SQLSTATE must be "38". (If this requirement is violated, then the effects are implementation-defined.) If that exception does not specify an SQLSTATE, then the default SQLSTATE for an uncaught Java exception is raised. See subclause 9.1, "*Class and subclass values for uncaught Java exception*s" of ISO/IEC 9075-10.

When a Java method executes an SQL statement, any exception condition raised in the SQL statement will be raised in the Java method as a Java exception that is specifically the *SQLException* subclass of the Java *Exception* class. As stated in ISO/IEC 9075-10, the effect of such an SQL exception condition on the outer SQL statement that called the Java method is implementation-defined. For portability, a Java method called from SQL, that itself executes an SQL statement and that catches an *SQLException* from that inner SQL statement should re-throw that *SQLException.*

4.7 Data types

update the last paragraph as indicated

Each host language has its own data types, which are separate and distinct from SQL data types, even though similar names may be used to describe the data types. Mappings of

SQL data types to data types in host languages are described in Subclause 11.49, ‘‘<SQL-invoked routine>’’ **in ISO/IEC 9075-2**, ~~**and**~~ Subclause 16.1, "<embedded SQL host program>" in ISO/IEC 9075-5, **and Subclause 8.1, "<embedded SQL host program>" in ISO/IEC 9075-10**. Not every SQL data type has a corresponding data type in every host language.

4.8 User-defined types

A user-defined type is a schema object, identified by a <user-defined type name>. The definition of a user-defined type specifies a number of components, including in particular a list of attribute definitions. Although the attribute definitions are said to define the representation of the user-defined type, in fact they implicitly define certain functions (observers and mutators) that are part of the interface of the user-defined type; physical representations of user-defined type values are implementation-dependent.

insert the following paragraphs after the 1’st paragraph:

User-defined types appear in two similar but fundamentally differing forms, where the key differentiator is whether or not the create type statement for the user-defined type specifies an external language of Java. When external language Java is specified, the user-defined type is an *external Java data type* and the create type statement defines a mapping of the user-defined type's attributes and methods directly to the public attributes and methods of a *subject Java class*. This is different from user-defined types that are not external Java data types; some differences include:

- for any other user-defined type, there is no requirement for an association with an underlying class; each method of a user-defined type that is not an external Java data type can be written in a different language (including SQL and FORTRAN, but not including Java). Whereas for an external Java data type all methods must be in Java and (implicitly) have a parameter style of Java.

- for any other user-defined type there is no explicit association between a user-defined type's attributes and any external representation of their content. Further the mapping between a user-defined type's methods and external methods is made over time by subsequent CREATE METHOD statements; for external Java data types the association between the user-defined type's attributes and methods and the public attributes and methods of a *subject Java class* is specified by the create type statement, though it may later be altered by an alter type statement. Further, for external Java data types the mechanism used to transform the SQL-environment's representation of an instance of a user-defined type into an instance of a Java class can be via either the Java interface java.io.serializable (not to be confused with the isolation level of serializable) or the JDBC-defined interface SQLData. See the clause “<user-defined type definition>”.

- for any other user-defined type there is no explicit support of static attributes; for external Java data types the create type statement is allowed to include <static field

method spec>s that define observer methods against specified static attributes of the subject Java class.

The scope and persistence of any modifications to static attributes made during the execution of a Java method is implementation-dependent.

- for any other user-defined type the implementation of any method that isn't an *SQL routine* exists externally to the *SQL-environment*; for external Java data types the implementation of the methods is provided by a specified subject Java class that exists in the *SQL-environment* in an *installed JAR*.

- external Java dataypes may only be *structured type*s, not *distinct types*

- support for the specification of overriding methods is not provided for methods that are external Java routines.

Note xx: Once an external Java data type has been created, its use in SQL statements executed by the containing SQL-implementation is similar to that of other user-defined types.

Note to the Editor: end of text to be inserted after the 1'st paragraph.

The representation of a user-defined type is expressed either as a single data type (some predefined data type, called the *source type*), in which case the user-defined type is said to be a *distinct type*, or as a list of attribute definitions, in which case it is said to be a *structured type*.

The definition of a user-defined type may include a <method specification list> consisting of one or more <method specification>s. A <method specification> is either an <original method specification> or an <overriding method specification>. Each <original method specification> specifies the <method name>, the <specific name>, the <SQL parameter declaration list>, the <returns data type>, the <result cast from type> (if any), the <transform group specification> (if any), whether the method is type-preserving, the <language clause>, the <parameter style> if the language is not SQL, whether STATIC is specified, whether the method is deterministic, to what extent the method accesses SQL (possibly writes SQL data, possibly reads SQL data, possibly contains SQL, or does not possibly contain SQL), and whether the method should be evaluated as NULL whenever any argument is NULL, without actually invoking the method.

Each <overriding method specification> specifies the <method name>, the <specific name>, the <SQL parameter declaration list> and the <returns data type>. For each <overriding method specification>, there must be an <original method specification> with the same <method name> and <SQL parameter declaration list> in some proper supertype of the user-defined type. Every SQL-invoked method in a schema must correspond to exactly one <original method specification> or <overriding method specification> associated with some user-defined type existing in that schema.

A method *M* that corresponds to an <original method specification> in the definition of a structured type *T1* is an *original method* of *T*1. A method *M* that corresponds to an <overriding method specification> in the definition of *T1* is an *overriding method* of *T*1.

A method *M* is a *method of type T1* if one of the following holds:

— *M* is an original method of *T*1.

— *M* is an overriding method of *T*1.

— There is a proper supertype *T2* of *T1* such that *M* is an original or overriding method of *T2* and such that there is no method *M3* such that *M3* has the same <method name> and <SQL parameter declaration list> as *M* and *M3* is an original method or overriding method of a type *T3* such that *T2* is a proper supertype of *T3* and *T3* is a supertype of *T*1.

If *T1* is a subtype of *T2* and *M1* is a method of *T1* such that there exists a method *M2* of *T2* such that *M1* and *M2* have the same <method name> and the same unaugmented <SQL parameter declaration list>, then *M1* is an *inherited method* of *T1* from *T*2

Replace the description of the user-defined type descriptor with:

A user-defined type is described by a user-defined type descriptor. A user-defined type descriptor contains:

— An indication of whether the user-defined type is an external Java data type.

— The name of the user-defined type. This is the type designator of that type, used in type precedence lists (see Subclause 9.5, ‘‘Type precedence list determination’’).

— An indication of whether the user-defined type is ordered.

— The ordering form for the user-defined type (EQUALS, FULL, or NONE).

— The ordering category for the user-defined type (RELATIVE, **COMPARABLE**, MAP, or STATE).

— A <specific routine designator> identifying the ordering function, depending on the ordering category.

— If the user-defined type is a direct subtype of another user-defined type, then the name of that user-defined type.

— An indication of whether the user-defined type is instantiable or not instantiable.

— An indication of whether the user-defined type is final or not final.

— The transform descriptor of the user-defined type.

— If the user-defined type is a structured type, then whether the reference type for which the structured type is the referenced type has a user-defined representation, a derived representation, or a system-defined representation, and the list of attributes of the derived representation.

NOTE 5 – ‘‘user-defined representation’’, ‘‘derived representation’’, and ‘‘system-defined representation’’ of a reference type are defined in Subclause 4.10, ‘‘Reference types’’.

In addition, if the user defined type is an external Java data type

— The <jar and class name> of the user-defined type.

— The <interface spec> of serializable or sqldata.

— The attribute descriptor of every originally-defined attribute and every inherited attribute of the user-defined type.

— If <method specification list> is specified, then for each <method specification> contained in <method specification list>, a *method spec descriptor* that includes:

- The <method name>.
- The <SQL parameter declaration list>.
- The <returns data type>, and indication of SELF AS RESULT.
- The <result cast from type>, if any.
- The package, class, and name of the Java routine corresponding to this method and, if specified, its signature.
- An indication of whether STATIC is specified.
- If STATIC is specified an indication whether this is a static field method.
- An indication whether the method is deterministic.
- An indication whether the method possibly writes SQL data, possibly reads SQL data, possibly contains SQL, or does not possibly contain SQL.
- An indication whether the method should not be invoked if any argument is the null value, in which case the value of the method is the null value.

And, in addition, if the user defined type is not an external Java data type

— An indication of whether the user-defined type is a structured type or a distinct type.

— If the representation is a predefined data type, then the descriptor of that type; otherwise the attribute descriptor of every originally-defined attribute and every inherited attribute of the user-defined type.

— If the <method specification list> is specified, then for each <method specification> contained in <method specification list>, a *method specification descriptor* that includes:

- The <method name>.
- The <SQL parameter declaration list> augmented to include the implicit first parameter with parameter name SELF.
- The <language name>.
- If the <language name> is not SQL, then the <parameter style>.
- The <returns data type>.
- The <result cast from type>, if any.
- An indication as to whether the <method specification> is an <original method specification> or an <overriding method specification>.
- If the <method specification> is an <original method specification>, then an indication of whether STATIC is specified.
- An indication whether the method is deterministic.
- An indication whether the method possibly writes SQL data, possibly reads SQL data, possibly contains SQL, or does not possibly contain SQL.
- An indication whether the method should not be invoked if any argument is the null value, in which case the value of the method is the null value.

NOTE 6 – The characteristics of an <overriding method specification> other than the <method name>, <SQL parameter declaration list>, and <returns data type> are the same as the characteristics for the corresponding <original method specification>.

Note to the Editor: end of text to replace the existing descriptor text

4.8.1 Observers and mutators

Corresponding to every attribute of every structured type is exactly one implicitly-defined observer function and exactly one implicitly-defined mutator function. These are both SQL-invoked functions. Further, the mutator function is a type-preserving function.

Let *A* be the name of an attribute of structured type *T* and let *AT* be the data type of A. The signature of the observer function for this attribute is FUNCTION A(T) and its result data type is *A*T. The signature of the mutator function for this attribute is FUNCTION A(*T* RESULT, *A*T) and its result data type is T.

Let *V* be a value in data type *T* and let *AV* be a value in data type *A*T. The invocation A(V,*A*V) returns *MV* such that A(*M*V)=*AV* and for every attribute *A' (A'* 6 = A) o*f* T, *A'*(*M*V)=*A'*(V). The most specific type of *MV* is the most specific type of V.

4.8.2 Constructors

Associated with every structured type *ST* is at least one *constructor functio*n, implicitly defined when *ST* is defined. The constructor function is defined if and only if *ST* is instantiable.

The signature of the constructor function for structured type *T* is T( ) and its result data type is T. The invocation T( ) returns a value *V* such that *V* is not null and, for every attribute *A* of T, A(V) returns the default value of A. The most specific type of *V* is T.

4.8.3 Subtypes and supertypes

As a consequence of the <subtype clause> of <user-defined type definition>, two structured types T_a and T_b that are not compatible can be such that T_a is a subtype of T_b. See Subclause 11.40, ''<user-defined type definition>''.

A type T_a is a *direct subtype* of a type T_b if T_a is a proper subtype of T_b and there does not exist a type T_c such that T_c is a proper subtype of T_b and a proper supertype of T_a.

A type T_a is a subtype of type T_b if one of the following pertains:

— T_a and T_b are compatible;

— T_a is a direct subtype of T_b;or

— T_a is a subtype of some type T_c and T_c is a direct subtype of T_b.

By the same token, T_b is a supertype of T_a and is a direct supertype of T_a in the particular case where T_a is a direct subtype of T_b.

If T_a is a subtype of T_b and T_a and T_b are not compatible, then T_a is proper subtype of T_b and T_b is a proper supertype of T_a. A type cannot be a proper supertype of itself.

A type with no proper supertypes is a maximal supertype. A type with no proper subtypes is a leaf type.

Let T_a be a maximal supertype and let *T* be a subtype of T_a. The set of all subtypes of T_a (which includes T_a itself) is called a *subtype family* of *T* or (equivalently) of T_a. A subtype family is not permitted to have more than one maximal supertype.

Every value in a type *T* is a value in every supertype of T. A value *V* in type *T* has exactly one most specific type *MST* such that *MST* is a subtype of *T* and *V* is not a value in any proper subtype of *MS*T. The most specific type of value need not be a leaf type. For example, a type structure might consist of a type PERSON that has STUDENT and EMPLOYEE as its two subtypes, while STUDENT has two direct subtypes UG_STUDENT and PG_STUDENT. The invocation STUDENT() of the constructor function for STUDENT returns a value whose most specific type is STUDENT, which is not a leaf type.

If T_a is a subtype of T_b, then a value in T_a can be used wherever a value in T_b is expected. In particular, a value in T_a can be stored in a column of type T_b, can be substituted as an argument for an input SQL parameter of data type T_b, and can be the value of an invocation of an SQL-invoked function whose result data type is T_b.

A type *T* is said to be the *minimal common supertype* of a set of types *S* if *T* is a supertype of every type in *S* and a subtype of every type that is a supertype of every type in S.

NOTE 7 – Because a subtype family has exactly one maximal supertype, if two types have a common subtype, they must also have a minimal common supertype. Thus, for every set of types drawn from the same subtype family, there is some member of that family that is the minimal common supertype of all of the types in that set.

If a structured type *ST* is defined to be not instantiable, then the most specific type of every value in *ST* is necessarily of some proper subtype of *S*T.

If a user-defined type *UDT* is defined to be final, then *UDT* has no proper subtypes. As a consequence, the most specific type of every value in *UDT* is necessarily *UD*T.

Users must have the UNDER privilege on a type before they can define any direct subtypes of it. A type can have more than one direct subtype. However, a type can have at most one direct supertype.

A <user-defined type definition> for type *T* can include references to components of every direct supertype of T. Effectively, components of all direct supertype representations are copied to the subtype's representation.

4.8.4 User-defined type comparison and assignment

update the following paragraph as indicated

Let *comparison type* of a user-defined type T_a be the user-defined type T_b that satisfies all the following conditions:

a) The type designator of T_b is in the type precedence list of T_a.

b) The user-defined type descriptor of T_b includes an ordering form that is EQUALS or FULL.

c) The descriptor of no type T_C whose type designator precedes that of T_b in the type precedence list of T_a includes an ordering form that includes EQUALS or FULL.

If there is no such type T_b, then T_a has no comparison type.

Let *comparison form* of a user-defined type T_a be the ordering form included in the user-defined type descriptor of the comparison type of T_a.

Let *comparison category* of a user-defined type T_a be the ordering category included in the user-defined type descriptor of the comparison type of T_a.

Let *comparison function* of a user-defined type T_a be the ordering function included in the user-defined type descriptor of the comparison type of T_a **, if any**.

Two values *V1* and *V2* of whose declared types are user-defined types *T1* and *T2* are comparable if and only if *T1* and *T2* are in the same subtype family and each have some comparison type *CT1* and *CT2*, respectively. *CT1* and *CT2* constrain the comparison forms and comparison categories of *T1* and *T2* to be the same — they must be the same throughout a type family. If the comparison category is **COMPARABLE, then no comparison functions shall be specified for *T1* and *T2*; if the comparison category is** either STATE or RELATIVE, then the comparison functions of *T1* and *T2* are constrained to be equivalent; if the comparison category is MAP, they are not constrained to be equivalent.

NOTE 8 – Explicit CAST functions or attribute comparisons can be used to make both values of the same subtype family or to perform the comparisons on attributes of the user-defined types.

NOTE 9 – ‘‘Subtype’’ and ‘‘subtype family’’ are defined in Subclause 4.8.3, ‘‘Subtypes and supertypes’’.

An expression *E* whose declared type is some user-defined type *UDT1* is assignable to a site *S* whose declared type is some user-defined type *UDT2* if and only if *UDT1* is a subtype of *UDT*2. The effect of the assignment of *E* to *S* is that the value of *S* is V, obtained by the evaluation of E. The most specific type of *V* is some subtype of *UDT*1, possibly *UDT1* itself, while the declared type of *S* remains *UDT2*.

4.8.5 Transforms for user-defined types

Transforms are SQL-invoked functions that are automatically invoked when values of user-defined types are transferred from SQL-environment to host languages or vice-versa. A transform is associated with a user-defined type.

A transform identifies a list of *transform groups* of up to two SQL-invoked functions, called the *transform function*s, each identified by a group name. The group name of a transform group is an <identifier> such that no two transform groups for a transform have the same group name. The two transform functions are:

— **from-sql function** — This SQL-invoked function maps the user-defined type value into a value of an SQL predefined type, and gets invoked whenever a user-defined type value is passed to a host language program or an external routine.

— **to-sql function** — This SQL-invoked function maps a value of an SQL predefined type to a value of a user-defined type and gets invoked whenever a user-defined type value is supplied by a host language program or an external routine.

A transform is defined by a <transform definition>. A transform is described by a *transform descriptor*. A transform descriptor includes a possibly empty list of *transform group descriptor*s, where each transform group descriptor includes:

— The group name of the transform group.

— The specific name of the from-sql function, if any, associated with the transform group.

— The specific name of the to-sql function, if any, associated with the transform group.

**** end of copied text ****

4.9 Built-in procedures

SQL/JRT differs slightly from other parts of ISO/IEC 9075 in its treatment of the schema object introduced to install the external Java routines and external Java data types in an SQL-environment, i.e., in its treatment of JARs. Rather than define new SQL-schema statements for, e.g., adding or dropping a JAR with optional clauses to cause execution of its contained deployment descriptor, SQL/JRT introduces a set of four built-in procedures and a new schema in which those procedures are defined.

The new schema named *sqlj* is, like information_schema, defined to exist in all catalogs of an SQL system that implement SQL/JRT, and to contain all of the built-in procedures of SQL/JRT.

SQL/JRT's New built-in procedures are:

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

4.10 Privileges

update the following paragraph as indicated

A privilege authorizes a given category of <action> to be performed on a specified base table, view, column, domain, character set, collation, translation, user-defined type, trigger, ~~**or**~~ SQL-invoked routine **, or JAR** by a specified <authorization identifier>.

update the following paragraph as indicated

Each privilege is represented by a *privilege descriptor*. A privilege descriptor contains:

— The identification of the base table, view, column, domain, character set, collation, translation, user-defined type, table/method pair, trigger, ~~**or**~~ SQL-invoked routine module**, or JAR** that the descriptor describes.

— The <authorization identifier> of the grantor of the privilege.

— The <authorization identifier> of the grantee of the privilege.

— Identification of the <action> that the privilege allows.

— An indication of whether or not the privilege is grantable.

— An indication of whether or not the privilege has the WITH HIERARCHY OPTION specified.

The <action>s that can be specified are:

— INSERT

— INSERT (<column name list>)

— UPDATE

— UPDATE (<column name list>)

— DELETE

— SELECT

— SELECT (<column name list>)

— SELECT (<privilege method list>)

— REFERENCES

— REFERENCES (<column name list>)

— USAGE

— UNDER

— TRIGGER

— EXECUTE

A privilege descriptor with an <action> of INSERT, UPDATE, DELETE, SELECT, TRIGGER, or REFERENCES is called a *table privilege descriptor* and identifies the existence of a privilege on the table identified by the privilege descriptor.

A privilege descriptor with an <action> of SELECT (<column name list>), INSERT (<column name list>), UPDATE (<column name list>), or REFERENCES (<column name list>) is called a *column privilege descriptor* and identifies the existence of a privilege on the columns in the table identified by the privilege descriptor.

A privilege descriptor with an <action> of SELECT (<privilege method list>) is called a *table/method privilege descriptor* and identifies the existence of a privilege on the methods of the structured type of the table identified by the privilege descriptor.

A table privilege descriptor specifies that the privilege identified by the <action> (unless the <action> is DELETE) is to be automatically granted by the grantor to the grantee on all columns subsequently added to the table.

Update the following paragraph as indicated

A privilege descriptor with an <action> of USAGE is called a *usage privilege descriptor* and identifies the existence of a privilege on the domain, user-defined type, character set, collation, ~~**or**~~ translation **, or JAR** identified by the privilege descriptor.

A privilege descriptor with an <action> of UNDER is called an *under privilege descriptor* and identifies the existence of the privilege on the structured type identified by the privilege descriptor.

A privilege descriptor with an <action> of EXECUTE is called an *execute privilege descriptor* and identifies the existence of a privilege on the SQL-invoked routine identified by the privilege descriptor.

A grantable privilege is a privilege associated with a schema that may be granted by a <grant statement>. The WITH GRANT OPTION clause of a <grant statement> specifies whether the <authorization identifier> recipient of a privilege (acting as a grantor) may grant it to others.

Privilege descriptors that represent privileges for the owner of an object have a special grantor value, ‘‘_SYSTEM’’. This value is reflected in the Information Schema for all privileges that apply to the owner of the object.

A schema that is owned by a given schema <user identifier> or schema <role name> may contain privilege descriptors that describe privileges granted to other <authorization

identifier>s (grantees). The granted privileges apply to objects defined in the current schema.

**** end of copied text ****

insert the following paragraph
The SQL privileges for SQL/JRT facilities are as follows:

- SQLJ.INSTALL_JAR, SQLJ.REPLACE_JAR, and SQLJ.REMOVE_JAR procedures:

  The privileges required to invoke these procedures is implementation-defined. This is similar to the implementation-defined privileges required for e.g. creating a schema.

- SQLJ.ALTER_JAVA_PATH procedure:

  You must be the owner of the specified jar, and also have the USAGE privilege on each jar in the path argument.

- CREATE PROCEDURE/FUNCTION and DROP PROCEDURE/FUNCTION:

  These are governed by the normal SQL privileges for CREATE and DROP.

- EXECUTE privilege on Java methods referenced by SQL names:

  This is governed by the normal SQL EXECUTE privilege on SQL routine names.

  It is implementation-defined whether a Java method called by an SQL name executes with "definer's rights" or “invoker’s rights”. I.e. whether it executes with the user-name of the user who performed the CREATE PROCEDURE/FUNCTION or the user-name of the current user.

**** end of inserted text ****

4.11 JARs

A JAR is a zip formatted file containing a set of Java class and ser files, and optionally a deployment descriptor file. Installed JARs provide the implementation of external Java routines and external Java data types to an SQL-environment.

JAR files are created outside the SQL-environment, and copied into the SQL-environment by the SQLJ.INSTALL_JAR procedure. No subsequent SQL statement or procedure modifies an installed JAR in any way, other than to remove it from the SQL-environment, to replace it in its entirety, or to alter its SQL-Java path. In particular, no SQL operation adds classes to a JAR, removes classes from a JAR, or replaces classes in a JAR. The reason for this “no modification” principle for installed JAR is that JARs are

often signed, and often contain “manifest” data that might be invalidated by modification of JARs by the SQL-environment.

Each installed JAR is represented by a *JAR descriptor*. A JAR descriptor contains:

- The catalog name, schema name, and JAR identifier of the JAR.
- The SQL-Java path of the JAR.

4.11.1 Deployment descriptor files

When a JAR file is installed one or more CREATE PROCEDURE/FUNCTION statements must be executed before the static methods of its contained Java classes can be used as SQL-invoked routines and one or more CREATE TYPE statements are required before its contained classes can be used as user-defined types. Further, GRANT statements may need to be performed against newly created SQL-invoked routines and user-defined types. Later, when a JAR file is removed corresponding DROP PROCEDURE/FUNCTION STATEMENTS, DROP TYPE, and REVOKE statements need to be executed.

If a JAR is to be installed in several SQL implementations, the CREATE, GRANT, DROP, and REVOKE statements will often be the same for each implementation. To assist the automation of repeated installations, deployment descriptor files contain SQL/JRT's variants of SQL-schema statements. These statements are grouped into multi-statement install and remove action groups respectively executed by SQLJ.INSTALL_JAR and SQLJ.REMOVE_JAR procedures when deployment is requested. In addition, an implementation-defined implementor block is provided to allow specification of custom install and remove actions. Since the SQL-schema statements refer to their containing JAR in the EXTERNAL NAME clauses of CREATE statements, within a deployment descriptor file, the JAR name *“thisjar”* is used as a place holder JAR name for the containing JAR.

SQL/JRT provides a new mechanism to execute its variants of SQL-schema statements, namely by requesting deployment during invocation of SQLJ.INSTALL_JAR and SQLJ.REMOVE_JAR procedures. A conforming SQL-implementation is required to support either deployment descriptor based execution of its SQL-schema statements (Feature J531, "Deployment") or another standard statement execution mechanism such as direct invocation or embedded SQL (Feature J511, "Commands"); a conforming SQL-implementation is not required to support both features.

5. LEXICAL ELEMENTS

5.1 <token> and <separator>

Function

Specify lexical units (tokens and separators) that participate in SQL language.

Format

<non-reserved word> ::=

!! All alternatives from ISO/IEC 9075-2

| COMPARABLE | INTERFACE | JAVA | SERIALIZABLE | SQLDATA

<reserved word> ::=

!! All alternatives from ISO/IEC 9075-2

| JAR

Syntax Rules

No additional Syntax Rules.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Conformance Rules

No additional Conformance Rules.

5.2 Names and identifiers

Function

Specify names.

Format

<jar name> ::= [<schema name> <period>]<jar id>

<jar id> ::= <identifier>

<Java class name> ::= [<packages> <period>]<class identifier>

<packages> ::= <package identifier>[<period><package identifier>]…

<package identifier> ::= <Java identifier>

<class identifier> ::= <Java identifier>

<Java field name> ::= <Java identifier>

<Java method name> ::= <Java identifier>

<Java identifier> ::= !! See the Syntax Rules

Syntax Rules

1) [Insert this SR] <Java identifier> shall be a valid identifier according to the rules of Java parsing and lexical analysis.

2) [Insert this SR] The character set supported, and the maximum length of the <package identifier>, <class identifier>, <Java field name>, and <Java method name> are implementation-defined.

3) 14.x) [Insert this SR after SR 14] Two <jar name>s are equivalent if and only if they have the same <jar id> and the same <schema name>, regardless of whether the <schema name>s are implicit or explicit.

Access Rules

None.

General Rules

1) [Insert this GR] A <jar name> identifies a JAR.

2) [Insert this GR] A <jar id> represents an unqualified JAR name.

3) [Insert this GR] A <Java class name> identifies a fully qualified Java class.

4) [Insert this GR] A <packages> identifies a fully qualified Java package.

5) [Insert this GR] A <package identifier> represents an unqualified Java package name.

6) [Insert this GR] A <class identifier> represents an unqualified Java class name.

7) [Insert this GR] A <Java field name> represents the name of a field within a Java class.

8) [Insert this GR] A <Java method name> represents the name of a method within a Java class.

6. PREDICATES

6.1 <comparison predicate>

Function

Specify a comparison of two row values.

Format

No additional Format items.

Syntax Rules

Replace Note 126:

NOTE 126 – The comparison form and comparison categories included in the user-defined type descriptors of both *UDT1* and *UDT2* are constrained to be the same — they must be the same throughout a type family. If the comparison category is **COMPARABLE, then no comparison functions shall be specified for *T1* and *T2*; if the comparison category is** either STATE or RELATIVE, then the comparison functions of *UDT1* and *UDT2* are constrained to be equivalent; if the comparison category is MAP, they are not constrained to be equivalent.

Replace Note 127:

NOTE 127 – If the comparison form is FULL, then the comparison category is constrained to be **COMPARABLE,** RELATIVE or MAP; if the comparison form is EQUALS, then the comparison category is also permitted to be STATE.

Access Rules

No additional Access Rules.

General Rules

1) [insert before NOTE 128 GR 1)b)iii)4)] If the comparison category of *UDTx* is COMPARABLE, then:

A) The subject SQL data type must be an external Java data type. Let JC be the subject Java class of that external Java data type.

NOTE xxx - Syntax Rules in <user-defined ordering definition> require that JC implement the Java interface *java.lang.Comparable*. The interface *java.lang.Comparable* requires an implementing Java class to have a method named *compareTo*, whose result data type is Java *int*.

B) Let *XJV* be the value of X in the associated Java environment (JVM). Let *YJV* be the value of Y in that associated Java environment.

C) X = Y

has the same result as if the JVM executed the boolean expression

```
XJV.compareTo( YJV )==0
```

D) X < Y

has the same result as if the JVM executed the boolean expression

```
XJV.compareTo( YJV )==-1
```

E) X <> Y

has the same result as if the JVM executed the boolean expression

```
XJV.compareTo( YJV )!=0
```

F) X > Y

has the same result as if the JVM executed the boolean expression

```
XJV.compareTo( YJV )==1
```

G)X <= Y

has the same result as if the JVM executed the boolean expression

```
XJV.compareTo( YJV )==-1 || XJV.compareTo( YJV )==0
```

H) X >= Y

has the same result as if the JVM executed the boolean expression

```
XJV.compareTo( YJV )==1 || XJV.compareTo( YJV )==0
```

Conformance Rules

No additional Conformance Rules.

7. ADDITIONAL COMMON ELEMENTS

7.1 <Java parameter declaration list>

Function

Specify the syntax and rules for <Java parameter declaration list>.

Syntax Rules

<Java parameter declaration list> ::=

<left paren> [<Java parameters>] <right paren>

<Java parameters> ::= <Java data type> [{<comma> <Java data type>}**...**]

<Java data type> ::= --*See below*.

Syntax Rules

1) A <*Java data type*> is a Java data type that is *mappable* or *result set mappable*, as specified in clause 4.5, "Parameter mapping". The <Java data type> names are case sensitive, and must be fully qualified with their package names, if any.

Access Rules

None.

General Rules

None.

7.2 <SQL Java path>

Function

Specify the syntax and rules for SQL-Java paths

Syntax

<SQL Java path> ::= [<path element>...]

<path element> ::=

<left paren> <referenced class> <comma> <resolution jar> <right paren>

<referenced class> ::=

[<packages><period>]<asterisk>

| [<packages><period>]<class identifier>

<resolution jar>::= <jar name>

Syntax Rules

None.

Access Rules

None.

General Rules

1) When a Java class CJ in a jar J is being executed in an SQL system, let P be the path (if any) associated with jar J by a call of the SQLJ.ALTER_JAVA_PATH procedure.

2) Any static or dynamic reference in CJ to a class name CN that is not a supported system class and is not contained in jar J is resolved as follows:

   For each path element PE (if any) in P:
   a) Let RC and RJ be the <referenced class> and <resolution jar> of PE.
   b) If RJ is not the name of an installed jar, then an exception condition is raised: *Java execution—invalid jar name in path.*
   c) If RC does not terminate with an "*", and RC is equal to CN, then:
      i) If CN is the name of a class C in the jar RJ, then CN resolves to class C.

      ii) If CN is not the name of a class in the jar RJ, then an exception condition is raised: *Java execution—unresolved class name.*

d) If RC terminates with an "*" and includes one or more package names P1, P2,...,PK, then:
   i) If the name CN begins with the K package names P1, P2,...,PK, and CN is the name of a class C in the jar RJ, then CN resolves to class C.

   ii) If the name CN begins with the K package names P1, P2,...,PK, and CN is not the name of a class in the jar RJ, then an exception condition is raised: *Java execution—unresolved class name*.

e) If RC is an "*" with no package names, then:
   i) If the name CN is the name of a class C in the jar RJ, then CN resolves to class C.

   ii) If CN is not the name of a class in the jar RJ, then continue the above "For each path element" loop.

3) If the above “For each path element” loop terminates without having resolved CN, then an exception condition is raised: *Java execution—unresolved class name*.

Conformance Rules

1) Without feature J601, "SQL/Java paths", conforming SQL/JRT shall not contain an <SQL Java path>.

7.3 SQL/JRT function call

Function

Call an SQL/JRT method as an SQL function.

Syntax

<SQL/JRT function call> ::= <SQL routine name> ([<arguments>])

<arguments> ::= <expression> [{,<expression>}…]

<SQL routine name> ::= --See below.

Syntax Rules

1) The <SQL routine name> is a 3-part SQL name, with normal rules for defaulting the first two parts.

2) The <SQL routine name> must be an <SQL routine name> specified in one or more **create function** statements.

Access Rules

None.

General Rules

1) SQL overloading rules as defined in [2], Subclause 9.1, "<routine invocation>", are applied to the <SQL routine name> and the data types of the <arguments> to identify a particular SQL function created by an SQL CREATE FUNCTION statement, CF. Let JF be the Java method identified by CF.

2) If CF specifies RETURNS NULL ON NULL INPUT, then if the runtime value of any argument is null, return a null value as the result of the function. In this case the following General Rules do not apply to this call of the function.

3) If CF 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 data type is *boolean*, *byte*, *short*, *int*, *long*, *float*, or *double*, if the runtime value of the corresponding argument is the SQL null value, then an exception condition is raised: *Java execution—invalid null value*.

4) Execute the Java method JF.

a) If the CREATE FUNCTION statement for the SQL-routine_name specified DETERMINISTIC, then the implementation is permitted to retain lists of argument and result values from invocations of the function, and to return those result values for subsequent invocations that specify the same argument values, without executing the function on those subsequent invocations.

b) Whether an 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.

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

d) If an SQL exception condition is raised during the execution of JF, 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.

5) Case:

a) If the execution of JF completes with an uncaught Java exception, E, then:

i) An SQL exception condition is raised with the SQLSTATE value specified in the clause "*Status codes*".

ii) Perform no further actions for the function call.

b) Otherwise, return the value of the method execution as the value of the <SQL/JRT function call>.

7.4 SQL/JRT procedure call

Function

Call an SQL/JRT method as an SQL stored procedure.

Syntax

The syntax of SQL procedure calls are specified in SQL/PSM, in proprietary SQL procedural languages, and in remote procedure call facilities of CLI, ODBC, and JDBC.

Syntax Rules

1) An <SQL procedure call> is syntax of an SQL procedure call. For example, the syntax defined in SQL-PSM96 [2] is “CALL <SQL routine name> <SQL argument list>”.

2) The <SQL routine name> of an <SQL procedure call> is an SQL <qualified name> appearing in the <SQL procedure call>, with normal rules for defaulting the catalog and schema identifier parts.
3) The <SQL argument list> of an <SQL procedure call> is a possibly empty list of <arguments> of the <SQL procedure call>.

Access Rules

None.

General rules

1) The <SQL routine name> must be an <SQL routine name> specified in one or more CREATE PROCEDURE statements.

2) SQL overloading rules, as defined in SQL [2], Subclause 9.1, “<routine invocation>”, are applied to the <SQL routine name> and the number of <arguments> to identify a particular stored procedure created by a CREATE PROCEDURE statement, SP. Let JSP be the Java method identified by SP.

3) For each parameter of JSP whose Java data type is *boolean, byte, short, int, long, float,* or *double,* if the parameter mode of the corresponding parameter of SP is **in** and the runtime value of the corresponding argument is the SQL null value, then an exception condition is raised: *Java execution—invalid null value.*

4) For each parameter of JSP whose Java data type is an array of *byte*, if the parameter mode of the corresponding parameter of SP is INOUT and the runtime value of the corresponding argument is the SQL null value, then an exception condition is raised: *Java execution—invalid null value.*

5) For each parameter P of JSP whose parameter mode is OUT or INOUT:

   a) Let TP be the SQL parameter_data type of P, and let JTP be the Java data type of the corresponding parameter of JSP.

   b) Let A be the value of the argument in the <SQL argument list> of the procedure call that corresponds with P.

   c) Let PA be a Java array of length 1 and data type JTP, initialized as specified by Java. I.e. a Java object ***new*** *JTP[1]*.

   d) If the parameter mode is INOUT, then set PA[0] to A.

   e) Replace A with PA in the <arguments> passed to JSP.

6) For each parameter P of JSP whose Java data type is result set mappable, generate a one-element array of the specified type containing a single NULL element, and supply that array as an implicit argument.

7) Execute JSP.

   a) If the CREATE PROCEDURE statement for the SQL-routine_name specified DETERMINISTIC, then the implementation is permitted to retain lists of argument values and OUT and INOUT parameter results from invocations of the procedure, and to return those OUT and INOUT parameter results for subsequent invocations that specify the same argument values, without executing the procedure on those subsequent invocations.

   b) Whether an execution is performed with the user-name of the user who created the CREATE PROCEDURE statement CF, or with the user-name of the current user is implementation-defined.

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

   d) If an SQL exception condition 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.

8) Case:

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

i) An SQL exception condition is raised with the SQLSTATE value specified in Table 13-1: ***SQLSTATE class and subclass values*** according to the rules specified in clause 13.1, "Class and subclass values for uncaught Java exceptions".

ii) Perform no further actions for the procedure call.

b) Otherwise,

i) For each parameter P of SP whose parameter mode is OUT or INOUT:

(1) Let A and PA be as defined above.

(2) Set the value of A to the contents of PA[0].

ii) If SP specifies DYNAMIC RESULT SETS**,** then:

(1) Let RSN be a set containing the first element of each of the arrays generated above for the result set mappable parameters. Let RS be the non-null elements of RSN.

(2) Let MAX be the integer specified in the DYNAMIC RESULT SETS clause of P. Let OPN be the number of elements in RS. If OPN is not greater than MAX, then let RET be OPN. If OPN is greater than MAX, then let RET be MAX, and raise an SQL warning: *Java execution—attempt to return too many result sets.*

(3) If the JDBC connection object that created any element of RS is closed, then the effect is implementation-defined.

(4) If any element of RS is not an object returned by a connection to the current SQL system and SQL session, then the effect is implementation-defined.

(5) Let ORS be the elements of RS in the order that they were opened in SQL. Let SRS be an ordered set of SQL result sets copied from the first RET elements of ORS.

(6) For each result set RSi in RS, close RSi and close the statement object that created RSi.

(7) Return the elements of SRS to the calling routine.

9) Whether the call of JSP returns update counts as defined in JDBC is implementation-defined.

Conformance Rules

None.

7.5 <member reference>

Function

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

Syntax

<member reference> ::=

<instance expression>**.**<member name>

| <user-defined type 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>

Syntax Rules

1) A <member reference> is an expression that denotes a field or method of a class instance or a method of a class.

2) An <instance expression> is an expression whose data type is an instance of an external Java data type. The <member reference> is a reference to a method or field of the given instance.

3) A <reference expression> is an expression whose data type is an SQL reference type.

4) An <SQL expression> is an SQL expression whose data type is an external Java data type.

5) A <user-defined type name> is an SQL data type name. This must be an SQL data type that is an external Java data type.

6) An <SQL method name> is the name of a static method of the external Java data type denoted by the <user-defined type name>.

7) A <member name> is the name of an attribute or method of the class instance denoted by the <instance expression>.

8) Case:

   a) If a <member reference> immediately contains an <instance expression>, then the data type or signature of the <member reference> is the data type 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 a <user-defined type name>, then the signature of the <member reference> is the signature of the method of the data type denoted by the <user-defined type name> whose name is the <SQL method name> .

   c) If a <member reference> immediately contains a <reference expression>, then the data type or signature of the <member reference> is the data type or signature of the attribute or method of the instance denoted by the <reference expression> whose name is the <member name>.

9) 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.

Access Rules

None.

General Rules

1) 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.

2) 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 SQL/JRT implementation and its value is returned as the value of that <member reference>.,

Conformance Rules

1) Without feature J611, "References", conforming SQL/JRT shall not contain a <reference expression> or an "->" operator.

7.6 <method call>

Function

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

Syntax

<method call> ::=

<member reference> ([<parameters>])

| **new** <user-defined type name> ([<parameters>])

<parameters> ::= <parameter> [{, <parameter>}…]

<parameter> ::= <expression>

Syntax Rules

1) A <method call> is an invocation of a static or dynamic method or a data type constructor.

2) A <member reference> is a member reference that denotes a method.

3) The <parameters> is a list of <parameter>s to be passed to the method. If there are no <parameter>s, then the empty parentheses must be included.

Access Rules

None.

General Rules

1) If the <member reference> immediately contains a <user-defined type name>, then let T be that data type. Otherwise, let T be the data type 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 data type T, the <member name> MN, and the number and data types of the <arguments> to identify a particular <method specification> MS.

   b) If MS is a <static field method spec>, then:

i) Let SSF be the *subject static field* of MS.

ii) Return the value of SSF as the result of the SQL/JRT method call.

iii) Do not perform the remaining actions of this clause.

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

3) If NEW is specified, then:

a) The <user-defined type definition> for T must not specify NOT INSTANTIABLE.

b) SQL overloading rules are applied to the constructor methods of data type T, the <user-defined type name>, and the number and data types of the <arguments> to identify a particular <method specification> 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 SQL/JRT 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 General Rules 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 data type 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 SQL/JRT types, the conversion is performed as specified by the Java interface specified in the CREATE TYPE statement that the defined the SQL/JRT 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 clause *13.2, "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 SQL/JRT 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.

Conformance Rules

1) Without Feature J571, "NEW operator", conforming SQL/JRT shall not contain the syntax "NEW <user-defined type name>([<parameters>])". If that feature is not supported, then the mechanism used to invoke a constructor is implementation-defined.

7.7 <privileges>

Function

Specify privileges.

Format

<object name> ::=

!! All alternatives from ISO/IEC 9075-2
| JAR <jar name>

Syntax Rules

1) <replace SR 3)> If <object name> specifies a <domain name>, <collation name>, <character set name>, <translation name>, <user-defined type name>, or <jar name> then <privileges> shall specify USAGE. Otherwise, USAGE shall not be specified.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Conformance Rules

1) Without Feature J561, "Jar privileges", an <action> shall not specify USAGE on an <object name> that immediately contains a <jar name>.

7.8 <language clause>

Function

Specify a standard programming language.

Format

<language name> ::=

!! All alternatives from ISO/IEC 9075-2

| JAVA

Syntax Rules

No additional Syntax Rules

Access Rules

No additional Access Rules

General Rules

1) <update GR 1)> With the exception of the language JAVA, whose standard is given in the normative reference *"The Java Language Specification"*, the standard programming language specified by the clause is defined in the International Standard identified by the <language name> keyword. Table 17, ‘‘Standard programming languages’’, specifies the relationship.

Conformance Rules

No additional Conformance Rules.

8. SCHEMA DEFINITION AND MANIPULATION

8.1 <SQL-invoked routine>

Function

Define an SQL-invoked routine.

Format

<parameter style> ::=

!! All alternatives from ISO/IEC 9075-2

| JAVA

<external Java reference string> ::=

<jar name> <colon> <Java class name> <period> <Java method name>

[<Java parameter declaration list>]

Syntax Rules

1) <insert SR 3.1)> If <SQL-invoked routine> specifies LANGUAGE JAVA then an <SQL parameter declaration> specified in <SQL-invoked function> shall not contain RESULT.

2) <insert SR 3.2)> If <SQL-invoked routine> specifies LANGUAGE JAVA then neither the <returns clause> contained in <SQL-invoked function> nor an <SQL parameter declaration> contained in a <SQL-invoked function> or <SQL-invoked procedure> shall contain <locator indication>.

3) <insert SR 3.3)> The maximum value of <maximum dynamic result sets> is implementation-defined.

4) <insert SR 3.4)> The character set supported for <external Java reference string> is implementation-defined.

5) <update SR 4) a)> Let *UDTN* be the <user-defined type name> immediately contained in <method specification designator>. Let *UDT* be the user-defined type identified by *UDTN*. *UDT* shall not be an *external Java type*.

6) <update SR 5) a)> <routine characteristics> shall contain at most one <language clause>, at most one <parameter style clause>, at most one <specific name>, at most one <deterministic characteristic>, at most one <SQL-data access indication>, at most one <null-call clause>, at most one <transform group specification>, and at most one

<dynamic result sets characteristic>. If LANGUAGE JAVA is specified then <parameter style clause> shall specify a <parameter style> of JAVA.

7) <update SR 5) i)> An <SQL-invoked routine> that specifies or implies LANGUAGE SQL is called an *SQL routine*; an <SQL-invoked routine> that does not specify LANGUAGE SQL is called an *external routine*. An external routine that specifies LANGUAGE JAVA is called an *external Java routine*.

8) <insert SR) 5) i.1)> If *R* is an external Java routine, the <external routine name> immediately contained in <external body reference> shall specify a <character string literal>; let V be the value of that <character string literal>. V shall conform to the Format and Syntax Rules of an <external Java reference string>.

   NOTE xx: *R* is defined by ISO/IEC 9075-2 to be the SQL-invoked routine specified by <SQL-invoked routine>.

9) <insert SR) 5) i.2)> If *R* is an external Java routine, then the <Java method name> is the name of one or more Java methods in the class specified by <Java class name> in the jar specified by <jar name>. The combination of <Java class name> and <Java method name> represent a fully qualified Java class name and method name. The method name can reference a method of the class, or a method of a superclass of the class.

10) <update the first paragraph of SR 19) e)> If PARAMETER STYLE GENERAL or PARAMETER STYLE JAVA is specified, then let the *effective SQL parameter list* be a list of *PN* parameters such that, for *i* ranging from 1 (one) to *P*N, the i-th effective SQL parameter list entry is defined as follows.

11) <update SR 19) g)> If <language clause> does not specify JAVA then any <data type> in an effective SQL parameter list entry shall specify a data type listed in the SQL data type column for which the corresponding row in the host data type column is not ‘‘none’’.

<insert a new NOTE after SR 19)> NOTE xx: The rules for parameter type correspondence when LANGUAGE JAVA is specified are given in Subclause <?> 'Parameter Mapping'.

Access rules

1) <insert AR 1.1)> If *R* is an external Java routine, then the applicable privileges of *A* shall include USAGE privilege on the jar referenced in the <external Java reference string>.

NOTE xx: the above references to *R* and *A* are defined in the Syntax Rules of ISO/IEC 9075-2's subclause 11.49, '<SQL-invoked routine>.'

General Rules

1) <insert before GR 1)> 0.1) Validation of the explicit or implicit <Java parameter declaration list> may be performed when <SQL-invoked routine> is executed, or when its created procedure or function is invoked. It is implementation-defined at which of these two times the validations are performed. Validation involves the following:

   a) Let EJR be the <external Java reference string>, and let *JN*, *JCLSN*, and *JMN*, respectively be the <jar name>, <Java class name>, and <Java method name> specified in EJR.

   b) Let SPDL be the effective SQL parameter list.

   c) Case:

      i) If the method identified by JMN is "main", then:

         (1) The statement must be a <schema procedure>.

         (2) The statement must not specify <dynamic result set characteristic>.

         (3) If EJR specifies a <Java parameter declaration list> JPDL, then it must be the following:

             (String[ ])

         (4) If EJR does not specify a <Java parameter declaration list>, then let JPDL be the following signature:

             (String[ ])

         (5) SPDL must be either:

            (a) A single parameter, which is an SQL ARRAY of CHAR or VARCHAR. At runtime, this parameter is passed as a Java array of java/lang/String.

            Note xx: This signature can only be specified if the SQL system supports Feature S201, ''SQL routines on arrays''.

            (b) Zero or more parameters, each of which is CHAR or VARCHAR. At runtime, these parameters are passed as a Java array of java/lang/String (with possibly zero elements).

      ii) If EJR does not specify a <Java parameter declaration list>, then determine a <Java parameter declaration list>, *JPDL*, from SPDL as follows:

         (1) For each parameter *P* of *SPDL* whose <parameter mode> is **IN**, or that does not specify an explicit <parameter mode>, if P is not an SQL array let the *corresponding Java parameter data type* of *P* be the corresponding

Java data type of the <parameter ype> of P; if P is an SQL array let *JT* be the corresponding Java data type of the <parameter ype> of P, and let the *corresponding Java parameter data type* of *P* be an array of *JT*, i.e. be *JT[]*.

Note yy: The *corresponding Java parameter data type* of *P* is defined in subclause <?>, 'Parameter mapping'.

(2) For each parameter *P* of *SPDL* whose parameter mode is **INOUT** or **OUT,** let *JT* be the corresponding Java data type of the <parameter ype> of P, and let the *corresponding Java parameter data type* of *P* be an array of *JT*, i.e. be *JT[]*.

(3) The <Java parameters> of *JPDL* is a list of the corresponding Java parameter data types of *SPDL*. Note that JPDL does not specify parameter names. I.e. the parameter names of the Java method do not have to match the SQL parameter names.

(4) Case

(a) If the CREATE statement does not specify DYNAMIC RESULT SETS, or if it specifies DYNAMIC RESULT SETS 0, then let JCS be the set of visible Java methods of class *JCLSN*, in *JN* whose method name is *JMN* and whose signature is *JPDL*.

(b) If the CREATE statement specifies DYNAMIC RESULT SETS N, for some positive N, then let SPN be the number of <SQL parameter declaration>s in SPDL. Let JCS be the set of visible Java methods of class *JCLSN*, in *JN* whose method name is *JMN*, whose first SPN parameter data types are those of JPDL, and whose last K parameter data types, for some positive K, are result set mappable.

iii) If EJR specifies a <Java parameter declaration list>, JPDL, then:

(1) Let JPN be the number of <Java data type>s in JPDL, and SPN be the number of <SQL parameter declaration>s in SPDL.

(2) If the CREATE statement is a <schema procedure> that does not specify DYNAMIC RESULT SETS or that specifies DYNAMIC RESULT SETS 0, then JPN must be equal to SPN. If the statement is a <schema procedure> that specifies DYNAMIC RESULT SETS N, for some positive N, then JPN must be greater than SPN, and each <Java data type> in JPDL whose ordinal position is greater than SPN must be result set mappable.

(3) For each <SQL parameter declaration> SP in SPDL, let ST be the <SQL data type> of SP and let JT be the corresponding <Java data type> in JPDL:

(a) If SP specifies I**N,** or does not specify a <parameter mode>, then JT and ST must be mappable.

(b) If SP specifies OUT or INOUT, then JT and ST must be output mappable.

d) Let JCS be the set of visible Java methods of class *JCLSN*, in *JN* whose method name is *JMN*, and whose signature is JPDL.

e) There must be exactly one Java method JM in JCS. The <SQL-invoked routine> is *associated* with Java method JM.

2) <insert before GR 1)> 0.2) Let JRT be the returns data type of JM:

a) If the CREATE statement is a <schema procedure>, then JRT must be **void**.

b) If the CREATE statement is a <schema function>, then JRT and SRT must be simply mappable, or object mappable.

3) <update GR 3)m)> If the SQL-invoked routine is an external routine, then the routine descriptor includes an indication of whether the *parameter passing style* is PARAMETER STYLE JAVA, PARAMETER STYLE SQL or PARAMETER STYLE GENERAL.

4) <update GR 6)a) i)>If *R* is not an external Java routine and the <SQL data access indication> in the descriptor of *R* is MODIFIES SQL DATA, READS SQL DATA, or CONTAINS SQL, then:

Conformance Rules

1) Without Feature J531, "Deployment", conforming SQL/JRT shall not specify an <SQL-invoked routine> in a <descriptor file>.

2) Without Feature J511, "Commands", conforming SQL/JRT shall not specify a <user –defined type statement> that specifies LANGUAGE JAVA outside of a <descriptor file>.

3) Without feature J581, "Output parameters", a conforming SQL/JRT <SQL-invoked routine> shall not specify <parameter mode> OUT or INOUT.

4) Without feature J631, "Java signatures", a <Java parameter declaration list> shall be equivalent to the default signature as determined in General Rule 2(b).

5) The SQL data types recognized by JDBC are a superset of those defined by ISO/IEC 9075-2. Without feature J521, “JDBC data types”, a <Java data type> shall have a corresponding ANSI SQL data type.

8.2 <drop routine statement>

Function

Destroy an SQL-invoked routine.

Format

No additional Format Items.

Syntax Rules

No additional Syntax Rules.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Conformance Rules

1) Without Feature J531, "Deployment", conforming SQL/JRT shall not specify a <drop routine statement> in a <descriptor file>.

2) Without Feature J511, "Commands", conforming SQL/JRT shall not specify a <drop routine statement> that drops an external Java routine outside of a <descriptor file>.

8.3 <user-defined type definition>

Function

Define a user-defined type.

Format

<user-defined type body> ::=
    <user-defined type name>
    [<subtype clause>]
    [<external Java type clause>]
     [AS <representation>]
    [<instantiable clause>]
    <finality>
    [<reference type specification>]
    [<cast option>]
    [<method specification list>]
<external Java type clause ::=
    <external Java class clause>
    LANGUAGE JAVA
    [USING <interface specification>]
<interface specification> ::= SQLDATA | SERIALIZABLE
<method specification> ::=
    !! All alternatives from ISO/IEC 9075-2
    <static field method spec>
<method characteristic> ::=
    !! All alternatives from ISO/IEC 9075-2
    | <external Java method clause>
<static field method spec> ::=
     STATIC METHOD <SQL method name> ( ) RETURNS <SQL data type>
    EXTERNAL VARIABLE NAME '<Java field name>'
<external Java class clause> ::= EXTERNAL NAME '<jar and class name>'

<external Java method clause> ::=

EXTERNAL NAME **'**<Java method and parameter declarations>**'**

<Java method and parameter declarations> ::=

<Java method name**>** [<Java parameter declaration list>]

<jar and class name> ::= <jar id>**:**<Java class name>

Syntax Rules

1) <insert SR3.1> If <external Java type clause> is specified, then:

   a) <external Java class clause> specifies that the CREATE statement defines an SQL name for a data type defined in a programming language other than SQL.

   b) <jar and class name> specifies the name of a Java class in an installed jar. A reference to the SQL data type name is effectively a synonym for the specified Java class.

   c) A <jar id> is the name of a jar.

   d) A <Java class name> is the fully-qualified name of a Java class in the specified jar.

   e) LANGUAGE JAVA specifies that the external data type is written in Java.

      i) All methods defined for an external Java data type are implicitly PARAMETER STYLE JAVA. Note that PARAMETER STYLE JAVA cannot be explicitly specified in the <method characteristic>.

   f) 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 SQL/JRT 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 SQL/JRT column.

      i) If a USING clause is not specified, then the default <interface spec> is implementation-defined.

   g) 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.

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

h) 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..

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

2) <insert SR 3.2> <external Java method clause> shall be specified only if <external Java type clause> is specified. <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>.
3) <SR 3.3> <static field method spec> shall be specified only if <external Java type clause> is specified. <static field method spec> is 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.

4) <SR 3.4> If <external Java type clause> is specified, then the none of the following clauses shall be specified:

a) <overriding method specification>

b) A <representation> that is a <predefined type>.

c) SELF AS LOCATOR.

d) INSTANCE METHOD.

Access Rules

No additional Access Rules.

General Rules

1) <insert GR 0.1> If EXTERNAL LANGUAGE JAVA is specified, then:

a) The character set supported, and the maximum length of the <jar and class name>, the <Java field name>, and <Java method name>, are implementation-defined.

b) Let SDT be an external Java data type, and JC be the subject Java class of SDT.

c) The Java class JC can be the subject Java class of other external Java data types. Each such external Java data type is a distinct data type.

d) The Java class JC must be a *public* class.

e) The Java class JC must implement the Java interface *java.io.Serializable* or the Java interface *java.sql.SQLData* or both.

f) If SDT specifies an SQL supertype SSDT, then:

   i) The SQL data type SSDT must be an external Java data type.

   ii) The Java class JC must be an immediate subclass of the subject Java class of SSDT.

g) For each <original method specification>, FMS, of SDT:

   i) Let JMN be the <Java method name> of FMS.
   ii) If FMS specifies SELF AS RESULT, then

      (1) FMS must not specify STATIC.

      (2) The returns data type of FMS must be the subject SQL data type.

   iii) If JMN is the same as JC, then the <SQL method name> must be the subject SQL data type name.
      (a) **Note:** This restriction retains the characteristic that constructor methods have the same name as the type.

   iv) Let SPDL be the <<SQL parameter declaration list> of FMS. Let JMN be the <Java method name> specified in the EXTERNAL NAME clause of FMS.

   v) If the EXTERNAL NAME clause of FML does not specify a <Java parameter declaration list>, then determine a <Java parameter declaration list>, *JPDL*, from SPDL as follows:

      (1) For each parameter *P* of *SPDL* whose <parameter mode>, let the *corresponding Java parameter data type* of *P* be the corresponding Java data type of the parameter_data type of *P*.

      (2) The <Java parameters> of *JPDL* is a list of the corresponding Java parameter data types of *SPDL*. Note that JPDL does not specify parameter names. I.e. the parameter names of the Java method do not have to match the SQL parameter names.

   vi) If the EXTERNAL NAME clause of FML specifies a <Java parameter declaration list> JDPL, then:

(1) Let JPN be the number of <Java data type>s in JPDL, and SPN be the number of <SQL parameter declarations> in SPDL.

(2) JPN must be equal to SPN.

(3) For each <SQL parameter declaration> SP in SPDL, let ST be the <SQL data type> of SP and let JT be the corresponding <Java data type> in JPDL. JT and ST must be mappable

vii) The method name JMN and <Java parameter declaration list> JPDL 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.

viii) If FMS specifies STATIC then JM must be static. If FMS does not specify STATIC, then JM must not be static.

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

x) If FMS does not specify SELF AS RESULT**,** then:

h) Let SFR be the RETURNS <SQL data type> of FMS. Let JFR be the Java return data type of JM.

i) SFR and JFR must be *simply mappabble* or *object mappable,* as defined in the clause "Parameter mapping".

j) For each <static field method spec>, SFMS, of SDT:

i) 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.

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

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

iv) Let SRT be the <SQL data type> specified in the RETURNS clause of SFMS. Let JFT be the Java data type of JSF.

v) SRT and JFT must be *simply mappable* or *object mappable,* as defined in clause 4.5. "Parameter mapping".

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

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

l) The subject SQL data type initially has an ordering form of *none*.

(1) **Note**: The *ordering form* of a data type indicates what comparisons and ordering operations are allowed for instances of the data type. The initial default ordering form is *none*. Other ordering forms are specified with the CREATE ORDERING statement. See the clause "Ordering of SQL/JRT data".

m) A CREATE TYPE statement specifying a subject SQL data type ST and a subject Java class JC implicitly extends the data type 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*".

Conformance Rules

1) Without Feature J531, "Deployment", conforming SQL/JRT shall not specify a <user-defined type definition> in a <descriptor file>.

2) Without Feature J511, "Commands", conforming SQL/JRT shall not specify a <user-defined type definition> that specifies LANGUAGE JAVA outside of a <descriptor file>.

3) Without Feature J591, "Overloading", the <SQL method name> of a <method specification> must not be the same as the <SQL method name> of any other <method specification> in the same CREATE TYPE statement.

4) Without Feature J641, "Static fields", conforming SQL/JRT must not specify a <static field method spec>.

5) Without Feature J541, "Serializable", a conforming SQL/JRT <user-defined type definition> shall not specify SERIALIZABLE.

6) Without Feature J551, "SQLDATA", a conforming SQL/JRT <user-defined type definition> shall not specify SQLDATA.

7) A conforming SQL/JRT implementation shall support at least one of Feature J541, "Serializable" and Feature J551, "SQLDATA".

8.4 <attribute definition>

Function

Define an attribute of a structured type.

Format

<attribute definition> ::=

<attribute name>

<data type>

[<reference scope check>]

[<attribute default>]

[<collate clause>]

[EXTERNAL NAME '<Java field name>'] !!New

Syntax Rules

1) If the <attribute definition> is contained in a <user-defined type definition> whose <interface specicfication> is explicitly or implicitly SERIALIZABLE, then the <attribute definition> must specify the <Java field name>. The <Java field name> is referred to as the *corresponding Java field name* of the <SQL attribute name>..

2) An <attribute definition> that specifies EXTERNAL shall not specify a <reference scope check>, <attribute default>, or <collate clause>.

Access Rules

No additional Access Rules.

General Rules

1) If the <attribute definition> is contained in a <user-defined type definition> whose <interface specicfication> is explicitly or implicitly SERIALIZABLE, then for each <attribute definition>, 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 definition> 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 data type> of AS, and JFT be the Java data type of JF.

f) SAT and JFT must be *simply mappable* or *object mappable*, as defined in clause 4.5, "Parameter mapping".

g) JF is the *subject field* of attribute AS.

Conformance Rules

No additional Conformance Rules.

8.5 <user-defined ordering definition>

Function

Define a user-defined ordering for a user-defined type.

Format

<ordering category> ::=

!! All alternatives from ISO/IEC 9075-2
| <comparable category>

<comparable category> ::=

RELATIVE WITH COMPARABLE INTERFACE

Syntax Rules

1) [Replace SR 4) with] If <comparable category>, <relative category> or <state category> is specified, then *UDT* shall be a maximal supertype.

2) [Insert before SR 6)] If <comparable category> is specified, then *UDT* shall be an external Java data type. Let JC be the subject Java class of that external Java data type. JC shall implement the Java interface *java.lang.Comparable*.

3) [Replace SR 6)b) with] If <comparable category> is not specified, then

Access Rules

No additional Access Rules.

General Rules

1) [Replace GR 3) with] Case:

   a) If <relative category> is specified, then the ordering category in the user-defined type descriptor of *UDT* is set to RELATIVE.

   b) if <map category> is specified, then the ordering category in the user-defined type descriptor of *UDT* is set to MAP.

   c) If <comparable category> is specified, then the ordering category in the user-defined type descriptor of *UDT* is set to COMPARABLE.

d) Otherwise, the ordering category in the user-defined type descriptor of *UDT* is set to STATE.

Conformance Rules

1) Without Feature J531, "Deployment", conforming SQL/JRT shall not specify a <user-defined ordering definition> in a <descriptor file>.

8.6 <drop user-defined ordering statement>

Function

Destroy a user-defined ordering method.

Format

No additional Format Items.

Syntax Rules

No additional Syntax Rules.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Conformance Rules

1) Without Feature J531, "Deployment", conforming SQL/JRT shall not specify a <drop user-defined ordering statement> in a <descriptor file>.

8.7 <drop data type statement>

Function

Destroy a user-defined type.

Format

No additional Format items.

Syntax Rules

No additional Syntax Rules.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Conformance Rules

1) Without Feature J531, "Deployment", conforming SQL/JRT shall not specify a <drop type statement> in a <descriptor file>.

2) Without Feature J511, "Commands", conforming SQL/JRT shall not specify a <drop type statement> that drops an external Java type outside of a <descriptor file>.

9. ACCESS CONTROL

9.1 <grant privilege statement>

Function

Define privileges.

Format

No additional Format Items.

Syntax Rules

No additional Syntax Rules.

Access Rules

No additional Access Rules.

General Rules

No additional General Rules.

Conformance Rules

1) Without Feature J531, "Deployment", conforming SQL/JRT shall not specify a <grant privilege statement> in a <descriptor file>.

2) Without Feature J511, "Commands", conforming SQL/JRT shall not specify a <grant privilege statement> that grants USAGE privilege on a JAR outside of a <descriptor file>.

9.2 <revoke statement>

Function

Destroy privileges and role authorizations.

Format

No additional Format items.

Syntax Rules

1) <replace SR 10)a)iii)4)> *P* and *D* are both usage privilege descriptors. The action and the identified domain, character set, collation, translation, user-defined type, or jar of *P* are the same as the action and the identified domain, character set, collation, translation, user-defined type, or jar of D, respectively.

2) <insert SR 29)c)> *DT* is an external Java data type and the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE on the jar whose <jar name> is contained in the <jar and class name> of the descriptor of *DT*.

3) <insert between SR 34) and SR 35)> Let *JR* be any JAR descriptor included in S1. *JR* is said to be *impacted* if the revoke destruction action would result in *A1* no longer having in its applicable privileges USAGE privilege on a JAR whose name is contained in a <resolution jar> contained in the SQL-Java path of *JR*.

4) <insert SR 35)p)> if *RD* is an external Java routine, USAGE on the jar whose <jar name> is contained in <external Java reference string> contained in the <external routine name> of the descriptor of *RD*.

5) <update SR 37)> If RESTRICT is specified, then there shall be no abandoned privilege descriptors, abandoned views, abandoned table constraints, abandoned assertions, abandoned domain constraints, lost domains, lost columns, lost schemas, impacted domains, impacted columns, impacted collations, impacted character sets, impacted JARs, abandoned user-defined types, forsaken column descriptors, forsaken domain descriptors, or abandoned routine descriptors.

Access Rules

1) No additional Access Rules.

General rules

1) <insert between GR 17) and SR 18)> If the object identified by <object name> of the <revoke statement> specifies <jar name>, let *J* be the jar identified by that <jar name>. For every impacted JAR descriptor, *JR*, and for each <path element>, *PE*, contained in the SQL-Java path of *JR* whose immediately contained <resolution jar> is *J*, the SQL-Java path of the JAR descriptor *JR* is modified such that it does not contain *PE*.

Conformance Rules

1) Without Feature J531, "Deployment", conforming SQL/JRT shall not specify a <revoke statement> in a <descriptor file>.

2) Without Feature J511, "Commands", conforming SQL/JRT shall not specify a <revoke statement> that revokes USAGE privilege on a JAR outside of a <descriptor file>.

10. BUILT-IN PROCEDURES

This clause defines the SQL/JRT built-in procedures.

10.1 SQLJ.INSTALL_JAR procedure

Function

Install a set of Java classes into the current SQL catalog and schema.

Signature

```
SQLJ.INSTALL_JAR (
        <url> IN VARCHAR(*),
        <jar> IN VARCHAR(*)
        <deploy> IN INTEGER)
```

Access Rules

1) The privileges required to invoke the INSTALL_JAR procedure are implementation-defined.

General Rules

1) The SQLJ.INSTALL_JAR procedure is a DDL operation, and subject to the implementation's rules for performing DDL operations within transactions. If an invocation of SQLJ.INSTALL_JAR raises an exception condition, then the effect on the install actions is implementation-defined.

2) The maximum lengths of the VARCHAR parameters are implementation-defined.

3) If the value of the <url> parameter does not identify a valid jar file, then an exception condition is raised: *Java DDL—invalid URL*.

4) If the value of the <jar> parameter after removal of leading and trailing space characters, does not have the format defined by the <jar name> BNF, then an exception condition is raised: *Java DDL—invalid jar name*.

   Let JN be the explicitly or implicitly qualified name *<catalog name>.<schema name>.<jar id>*.

5) If there is an installed jar whose name is JN, then an exception condition is raised: *Java DDL—invalid jar name*.

6) The jar is installed with the name *JN*. All contents of the jar are installed, including both visible and non-visible Java classes, and other items contained in the jar. The non-visible Java classes and other items can be referenced by other Java methods within the jar.

7) A privilege descriptor is created that defines the privilege USAGE on the JAR identified by <jar> to the <authorization identifier> that owns the schema identified by the implicit or explicit <schema name> of the <jar>. The grantor for the privilege descriptor is set to the special grantor value ''_SYSTEM''. The privilege is grantable.

8) If the value of the *deploy* parameter is not zero, and if the jar file contains one or more deployment descriptor files, then the *install actions* implied by those instances are performed in the order in which the deployment descriptor files appear in the manifest. **Note:** deployment descriptor files and their install actions are specified in clause 11.2, "*Deployment descriptor files*".

Conformance Rules

1) Without Feature J531, "Deployment", conforming SQL/JRT shall not specify non-zero values of the <deploy> parameter.

10.2 SQLJ.REPLACE_JAR procedure

Function

Replace a previously installed jar.

Signature

SQLJ.REPLACE_JAR (

<url> IN VARCHAR(*)

<jar> IN VARCHAR(*),

)

Access Rules

1) The privileges required to invoke the SQLJ.REPLACE_JAR procedure are implementation-defined.

2) The current user must be the owner of the specified jar.

General Rules

1) The SQLJ.REPLACE_JAR procedure is a DDL operation, and subject to the implementation's rules for performing DDL operations within transactions. If an invocation of SQLJ.REPLACE_JAR raises an exception condition, then the effect on the install actions is implementation-defined.

2) The maximum lengths of the VARCHAR parameters are implementation-defined.

3) If the value of the <url> parameter identifies a valid jar file, then refer to the classes in that new jar file as the *new classes*. If the value of the <*url*> parameter does not identify a valid jar file, then an exception condition is raised: *Java DDL—invalid URL*.

4) If the value of the <jar> parameter after removal of leading and trailing space characters, does not have the format defined by the <jar name> BNF, then an exception condition is raised: *Java DDL—invalid jar name*.

   Let JN be the explicitly or implicitly qualified name <*catalog name*>.<*schema name*>.<*jar id*>.

5) If there is an installed jar with jar name JN, then refer to that jar as the *old jar file*. Refer to the classes in the old jar file as the *old classes*. If there is not an installed jar

with <jar name> JN, then an exception condition is raised: *Java DDL—invalid jar name*.

6) Let the *matching old (new) classes* be the old (new) classes whose fully qualified class names are the names of new (old) classes. Let the *unmatched old (new) classes* be the old (new) classes that are not matching old (new) classes.

7) Let the *dependent SQL names* of a jar file be the <schema qualified routine name>s whose CREATE PROCEDURE/FUNCTION statements specify an <external Java reference> clause that references any method in any class contained in that jar file.

8) If any dependent SQL names of the old jar file reference a method in an unmatched old class, then an exception condition is raised: *Java DDL—invalid class deletion*.

   Note: This rule prohibits deleting classes that are referenced by an SQL CREATE…EXTERNAL statement. This prohibition does not, however, prevent deletion of classes that are referenced only indirectly by other Java classes.

9) For each dependent SQL name of the old jar file that references a method in a matching old class, let CS be the SQL CREATE statement that created the SQL name. If CS is not a valid CREATE statement for the corresponding new class, then an exception condition is raised: *Java DDL—invalid replacement*.

10) The old jar file and all visible and non-visible old classes are deleted.

11) The new jar file and all visible and non-visible new classes are installed and associated with the specified <jar name>. That jar becomes the *associated jar* of each new class. All contents of the new jar are installed, including both visible and non-visible Java classes, and other items contained in the jar. The non-visible Java classes and other items can be referenced by other Java methods within the jar.

12) The effect of SQLJ.REPLACE_JAR on currently executing SQL statements that use SQL routines whose implementation has been replaced is implementation-dependent.

10.3 SQLJ.REMOVE_JAR procedure

Function

Remove a Java jar file and its classes from a specified catalog.

Signature

SQLJ.REMOVE_JAR(<jar> IN VARCHAR(*), <undeploy> IN INTEGER)

Access Rules

1) The privileges required to invoke the SQLJ.REMOVE_JAR procedure are implementation-defined.

2) The current user must be the owner of the specified jar.

General rules

1) The SQLJ.REMOVE_JAR procedure is a DDL operation, and subject to the implementation's rules for performing DDL operations within transactions. If an invocation of SQLJ.REMOVE_JAR raises an exception condition, then the effect on the remove actions is implementation-defined.

2) The maximum length of the VARCHAR parameter is implementation-defined.

3) If the value of the <jar> parameter after removal of leading and trailing space characters, does not have the format defined by the <jar name> BNF, then an exception condition is raised: *Java DDL—invalid jar name*.

   Let JN be the explicitly or implicitly qualified name <*catalog name*>.<*schema name*>.<*jar id*>.

4) If there is an installed jar with jar name JN, then refer to that jar as the *old jar file*. Refer to the classes in the old jar file as the *old classes*. If there is not an installed jar with <jar name> JN, then an exception condition is raised: *Java DDL—invalid jar name*. Equality of jar names is determined by the SQL rules for equivalence of identifiers as specified in [1], Subclause 5.2, "<token> and <separator>".

5) If the value of the <undeploy> parameter is non-zero, then if the jar file contains one or more instances of the *DeploymentDescriptor* class, then the <remove action>s implied by those instances are performed in the reverse of the order in which the deployment descriptor files appear in the manifest. Those actions are specified in clause 11.2, "*Deployment descriptor files*". Note that these actions are performed prior to examining the condition specified in the following step.

6) Let the *dependent SQL names* of a jar file be as defined in the SQLJ.REPLACE_JAR procedure. I.e. the <schema qualified routine name>s whose CREATE PROCEDURE/FUNCTION statements specify an <external Java reference> clause that references any method in any class contained in that jar file.

7) If there are any dependent SQL names of the specified jar file, then an exception condition is raised: *Java DDL—invalid class deletion*.

   Note: This rule prohibits deleting classes that are referenced by an SQL CREATE…EXTERNAL statement. This prohibition does not, however, prevent deletion of classes that are referenced only indirectly by other Java classes.

8) The specified jar file and all visible and non-visible classes contained in it are deleted.

9) The USAGE privilege on the specified jar file is revoked from all users that have it.

10) The effect of SQLJ.REMOVE_JAR on currently executing SQL statements that use SQL routines whose implementation has been replaced is implementation-dependent.

Conformance Rules

1) Without Feature J531, "Deployment", conforming SQL/JRT shall not specify non-zero values of the <undeploy> parameter.

10.4 SQLJ.ALTER_JAVA_PATH procedure

Signature

SQLJ.ALTER_JAVA_PATH(<jar> IN VARCHAR(*), <path> IN VARCHAR(*))

Access Rules

1) The privileges required to invoke the SQLJ.ALTER_JAVA_PATH procedure are implementation-defined.

2) The current user must be the owner of the jar JN.

3) The current user must have the USAGE privilege on each jar referenced in the <path> parameter.

General Rules

1) The SQLJ.ALTER_JAVA_PATH procedure is a DDL operation, and is subject to the implementation-defined rules for performing DDL operations within transactions.

2) If the value of the <jar> parameter after removal of leading and trailing space characters, does not have the format defined by the <jar name> BNF, then an exception condition is raised: *Java DDL—invalid jar name*.

   Let JN be the explicitly or implicitly qualified name *<catalog name>.<schema name>.<jar id>*.

3) The maximum lengths of the VARCHAR parameters are implementation-defined.

4) When the SQLJ.ALTER_JAVA_PATH procedure is called, the current catalog and schema at the time of the call are the default for each omitted <catalog name> and <schema name> in the <resolution jar>s of the *<path>* parameter. Those defaults apply to any subsequent use of the <path> as specified below.

5) If the value of the <path> parameter does not have the format for <SQL Java path> defined by the clause "<SQL Java path>", then an exception condition is raised: *Java DDL—invalid path name*.

   Note that the <path> parameter can be an empty or all-blank string.

6) The value of the <path> parameter becomes the path associated with the jar denoted by JN, replacing the current path (if any) associated with that jar.

7) If an invocation of the SQLJ.ALTER_JAVA_PATH procedure raises an exception condition , then effect on the path associated with the jar is implementation-defined.

8) The effect of SQLJ.ALTER_JAVA_PATH on SQL statements that have already been prepared or are currently executing is implementation-dependent.

Conformance Rules

1) Without feature J601, "SQL-Java paths", conforming SQL/JRT shall not contain invocations of the SQLJ.ALTER_JAVA_PATH procedure.

11. JAVA TOPICS

11.1 Java facilities supported by SQL/JRT

11.1.1 Package *java.sql*

SQL systems that implement *SQL/JRT* will support the package *java.sql*, which is the JDBC driver. The other Java packages supplied by SQL systems that implement SQL/JRT are implementation-defined.

In an SQL system that implements *SQL/JRT*, the package *java.sql* supports the *default connection*. The default connection for a Java method invoked as an SQL routine has the following characteristics:

— The default connection is pre-allocated to provide efficient access to the database.

— The default connection is included in the current session and transaction.

— The authorization ID of the default connection is the current authorization ID.

— The JDBC AUTOCOMMIT setting of the default connection is *false*.

Other data source URLs that are supported by java.sql are implementation-defined.

11.1.2 System properties

SQL systems that implement *SQL/JRT: SQL Routines* will support the following system properties for use by the *getProperty* method of *java.lang.System*:

Key	***Description of associated value***
sqlj.defaultconnection	See note 1
sqlj.runtime	See note 2

Table 11-1: System properties

Notes:

1) If a Java method is executing in an SQL system, then the value of *sqlj.defaultconnection* will be the String “jdbc:default:connection”. Otherwise, it will be null.

2) The value of sqlj.runtime is the class name of an SQL/JRT runtime context class. This class is a subclass of the class *sqlj.runtime.RuntimeContext*. The *getDefaultContext* method of the class whose name is returned returns the default connection described in clause 11.1.1, "*Package java.sql*".

11.2 Deployment descriptor files

Function

Supply information for actions to be taken by the SQLJ.INSTALL_JAR and SQLJ.REMOVE_JAR procedures.

Model

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

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*

Description

1) The SQLActions must contain at most one <install actions> and at most one <remove actions>.

2) The <command>s specified in the <install actions> (if any) and <remove actions> (if any) specify the *install actions* and *remove actions* of the deployment descriptor file.

3) An <SQL statement> specified in an <install actions> must be either:

   a) A CREATE PROCEDURE or CREATE FUNCTION statement that specifies EXTERNAL…LANGUAGE JAVA. The procedures and functions created by those statements are called the *deployed routines* of the deployment descriptor file.

   b) A GRANT statement that specifies the EXECUTE privilege for a deployed routine.

   c) 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.

   d) A GRANT statement that specifies the usage privilege for a deployed type.

   e) A CREATE ORDERING statement that specifies ordering for a deployed type.

4) When a deployment descriptor file is executed by a call of the SQLJ.INSTALL_JAR procedure, if the <jar name> of any EXTERNAL NAME clause of a CREATE PROCEDURE/FUNCTION statement in an <install actions> is the <jar name> "*thisjar*", then "*thisjar*" is replaced with the <jar name> parameter of the SQLJ.INSTALL_JAR procedure.

5) An <SQL statement> specified in a <remove actions> must be either:

   a) A DROP PROCEDURE or DROP FUNCTION statement for a deployed routine.
   b) A REVOKE statement for the EXECUTE privilege on a deployed routine.
   c) A DROP TYPE statement for a deployed type.
   d) A REVOKE statement for the USAGE privilege on a deployed type.
   e) A DROP ORDERING statement that specifies ordering for a deployed type.

6) An <implementor block> specifies implementation-specific install actions and remove actions.
   a) An <SQL token> is an SQL lexical unit specified by the term "<token>" in the SQL standard [1] in Subclause 5.2, "<token> and <separator>". I.e. the comments, quotes, and double-quotes in an <implementor block> must follow SQL token conventions.
   b) An <implementor name> is an implementation-defined SQL identifier. The <implementor name>s specified following the BEGIN and END keywords must be the same.
   c) Whether an <implementor block> with a given <implementor name> contained in an <install actions> (<remove actions>) is interpreted as an install action (remove action) is implementation-defined. I.e. an implementation may or may not perform install or remove actions specified by some other implementation.

7) The deployment descriptor file format corresponds to the more general notion of a properties file supporting indexed properties. Therefore the deployment descriptor file can be used by the SQL system to instantiate a Java Bean having an indexed property, **SQLRoutines**. You can then customize the resulting Java Bean instance with ordinary Java Bean tools. For example, you can change the SQL procedures or permissions by changing the routine descriptors stored in the **SQLRoutines** property. The SQL system can then use the customized Java Bean instance to generate a modified version of the deployment descriptor file to use in a revised version of the jar file.

12. DEFINITION SCHEMA

12.1 USER_DEFINED_TYPES base table

Function

The USER_DEFINED_TYPES table has one row for each user-defined type.

Definition

```
CREATE TABLE USER_DEFINED_TYPES (
            :
            :
      ORDERING_CATEGORY  INFORMATION_SCHEMA.CHARACTER_DATA
            CONSTRAINT
            USER_DEFINED_TYPES_ORDERING_CATEGORY_CHECK
            CHECK ( ORDERING_CATEGORY
                  IN ( 'RELATIVE', 'MAP', 'STATE', 'COMPARABLE' ) ),
            :
            :
      )
```

Description

1) [Replace Description 7) with] The values of ORDERING_CATEGORY have the following meanings:

RELATIVE	Two values of this type can be compared with a relative routine.
MAP	Two values of this type may be compared with a map routine.
STATE	Two values of this type may be compared with a state routine.
COMPARABLE	Two values of this type may be compared with java.lang.Comparable's compareTo method.

13. STATUS CODES

13.1 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.

13.2 SQLSTATE

The SQLSTATE class and subclass values for SQL/JRT are as follows:

Condition	Class	Subcondition	Subclass
warning	01	attempt to return too many result sets	00E
Uncaught Java exception	38	(no subclass)	000
User-defined (see above)	38	User-defined (see above)	mmm
external routine invocation exception	39	invalid SQLSTATE returned	001
external routine invocation exception	39	null value not allowed	004
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 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

Table 13-1: *SQLSTATE class and subclass values*

14. CONFORMANCE

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

15. ANNEX (INFORMATIVE) --- ROUTINES TUTORIAL

15.1 Technical components

SQL/JRT includes the following:

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

  The built-in schema named *sqlj* is assumed to be in all catalogs of an SQL system that implements the SQL/JRT facility, and to contain all of the built-in procedures of the SQL/JRT facility.
- Extensions of the following SQL statements:
  - CREATE PROCEDURE/FUNCTION—to specify an SQL name for a Java method.
  - DROP PROCEDURE/FUNCTION—to delete the SQL name of a Java method.
  - CREATE TYPE – to specify an SQL name for a Java class.
  - DROP TYPE – to delete the SQL name of a Java class.
  - GRANT—to grant the USAGE privilege on Java jars.
  - REVOKE—to revoke the USAGE privilege on Java jars.
- Conventions for returning values of OUT and INOUT parameters, and for returning SQL result sets.
- New forms of reference: Qualified references to the fields and methods of columns whose data types are defined on Java classes.

15.2 Overview

This tutorial clause shows a series of example Java classes and their methods, and indicates how they can be installed and used in an SQL system with the SQL/JRT facilities.

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

- *name* –the employee's name
- *id* –the employee's identification
- *state* –the state in which the employee is located
- *sales* –the amount of the employee's sales
- *jobcode*—the job code of the employee.

The table definition is:

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

The example classes and methods are:

- *Routines1.region* – A Java method that maps a US state code to a region number. This method doesn't use SQL internally.
- *Routines1.correctStates* –A Java method that performs an SQL UPDATE statement to correct the spelling of *state* codes. The old and new spellings are specified by input-mode parameters.
- *Routines2.bestTwoEmps*—A Java method that determines the top two employees by their sales, and returns the columns of those two employee rows as output-mode parameter values. This method creates an SQL result set and processes it internally.
- *Routines3.orderedEmp*s—A Java method that creates an SQL result set consisting of selected employee rows ordered by the sales column, and returns that result set to the client.
- *Over1.isOdd* and *Over2.isOdd*—Contrived Java methods to illustrate overloading rules.
- *Routines4.job1 and Routines5*.job2—Java methods that return a string value corresponding to an integer jobcode value. These methods illustrate the treatment of null arguments.
- *Routines6.job3*—Another Java method that returns a string value corresponding to an integer jobcode value. This method illustrates the behavior of static Java variables.

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

15.3 Example Java methods: *region* and *correctStates*

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

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

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

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

```
            return;
        }
    }
```

15.4 Installing *region* and *correctStates* in SQL

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

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

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

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

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

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

  As a parameter of the SQLJ.REMOVE_JAR and SQLJ.REPLACE_JAR procedures.

  As a qualifier of Java class names in SQL CREATE PROCEDURE/FUNCTION statements.

  As an operand of the extended SQL GRANT and REVOKE statements.

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

- Jar files can also contain *deployment descriptors*, which specify implicit actions to be taken by the SQLJ.INSTALL_JAR and SQLJ.REMOVE_JAR procedures. The third parameter of the SQLJ.INSTALL_JAR procedure is an integer that specifies whether you do or do not (indicted by non-zero or zero values) want the INSTALL_JAR procedure to execute the actions specified by a *deployment descriptor* in the jar file. Deployment descriptors are further described in clause 15.22, "*Deployment descriptors*".

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

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

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

- Obtain the jar file designated by the first parameter.
- Extract the class files that it contains and install them into the current SQL schema.
- Retain a copy of the jar file itself, and associate it with the value of the second parameter.
- If the third parameter is non-zero, then perform the actions specified by the deployment descriptor of the jar file.

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

15.5 Defining SQL names for *region* and *correctStates*

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

```
CREATE PROCEDURE correct_states(old char(20), new char(20))
MODIFIES SQL DATA
EXTERNAL NAME 'routines1_jar:Routines1.correctStates'
LANGUAGE JAVA PARAMETER STYLE JAVA;

CREATE FUNCTION region_of(state char(20)) RETURNS integer
NO SQL
EXTERNAL NAME 'routines1_jar:Routines1.region'
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

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

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

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

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

```
SELECT name, region_of(state) AS region
FROM emps
WHERE region_of(state) = 3;


CALL correct_states ('GEO', 'GA');
```

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

```
CREATE PROCEDURE state_correction(old char(20), new char(20))
MODIFIES SQL DATA
EXTERNAL NAME 'routines1_jar:Routines1.correctStates'
LANGUAGE JAVA PARAMETER STYLE JAVA;


CREATE FUNCTION state_region(state char(20)) RETURNS integer
EXTERNAL NAME 'routines1_jar:Routines1.region'
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

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

```
SELECT name, state_region(state) AS region
FROM emps
WHERE region_of(state) = 2;


CALL state_correction ('ORE', 'OR');
```

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

There are other considerations for the **create procedure/function** statement, dealing with parameter data types, overloaded names, and privileges, which we will discuss in later clauses.

15.6 A Java method with output parameters: *bestTwoEmps*

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

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

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

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

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

```
public class Routines2  {
        public static void bestTwoEmps
                        (String[ ] n1, String[ ] id1, int[ ] r1, BigDecimal[ ] s1,
                        String[ ] n2, String[ ] id2, int[ ] r2, BigDecimal[ ] s2,
                        int regionParm) throws SQLException  {
                #sql iterator ByNames (String name, String id, int region, BigDecimal sales);
                n1[0]= "****";  n2[0]= "****";  id1[0]= ""; id2[0]= "";
                r1[0]=0; r2[0]=0; s1[0]= new BigDecimal(0); s2[0]= new BigDecimal(0);
                ByNames r;
                try {
                        #sql r = {SELECT name, id, region_of(state) as region, sales
                                FROM emp
```

```
                    WHERE region_of(state) > :regionParm
                    AND sales IS NOT NULL
                    ORDER BY sales DESC};
            if (r.next()) {
                    n1[0] = r.name();
                    id1[0] = r.id();
                    r1[0] = r.region();
                    s1[0] = r.sales();
                                } else  return;
            if (r.next()) {
                    n2[0] =  r.name();
                    id2[0] =  r.id();
                    r2[0] =  r.region();
                    s2[0] = r.sales();
                                } else  return;
        } finally r.close();
    }

}
```

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

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

```
public class Routines2  {
    public static void bestTwoEmps
            (String[ ] n1, String[ ] id1, int[ ] r1, BigDecimal[ ] s1,
            String[ ] n2, String[ ] id2, int[ ] r2, BigDecimal[ ] s2,
            int regionParm) throws SQLException  {
        n1[0]= "****";  n2[0]= "****";  id1[0]= ""; id2[0]= "";
        r1[0]=0; r2[0]=0; s1[0]= new BigDecimal(0); s2[0]= new BigDecimal(0);
        try {
            Connection conn = DriverManager.getConnection
                ("jdbc:default:connection");
```

```
                    java.sql.PreparedStatement stmt = conn.prepareStatement
                            ("SELECT name, id, region_of(state) as region, sales
                            FROM emp
                            WHERE region_of(state) > ?
                            AND sales IS NOT NULL
                            ORDER BY sales DESC");
                    stmt.setInt(1, regionParm)
                    ResultSet r = stmt.executeQuery();
                    if (r.next()) {
                            n1[0] = r.getString("name");
                            id1[0] = r.getString("id");
                            r1[0] = r.getInt("region");
                            s1[0] = r.getBigDecimal("sales");
                                    } else  return;
                    if (r.next()) {
                            n2[0] =  r.getString("name");
                            id2[0] =  r.getString("id");
                            r2[0] =  r.getInt("region");
                            s2[0] = r.getBigDecimal("sales");
                                    } else return;
            } finally stmt.close();
        }
}
```

15.7 A CREATE PROCEDURE *best2* for *bestTwoEmps*

Assume that you call the SQLJ.INSTALL_JAR procedure for a jar file containing the *Routines2* class with the *bestTwoEmps* method:

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

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

```
CREATE PROCEDURE best2
(OUT n1 varchar(50), OUT id1 varchar(5), OUT r1 integer, OUT s1 decimal(6,2),
```

```
    OUT n2 varchar(50), OUT id2 varchar(5), OUT r2 integer,
    OUT s2 decimal(6,2), region integer)
    READS SQL DATA
    EXTERNAL NAME 'routines2_jar:Routines2.bestTwoEmps'
    LANGUAGE JAVA PARAMETER STYLE JAVA;
```

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

15.8 Calling the *best2* procedure

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

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

15.9 A Java method returning a result set: *orderedEmps*

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

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

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

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

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

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

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

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

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

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

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

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

Each of the above *orderedEmps* examples has a single result set parameter, *rs*, in which you can only return a single result set. You can also specify multiple result set parameters. See the clauses "<SQL-invoked routine>" and "SQL/JRT procedure call" .

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

15.10 A CREATE PROCEDURE *rankedEmps* for *orderedEmps*

Assume that you call the SQLJ.INSTALL_JAR procedure for a jar file containing the *Routines3* class with the *orderedEmps* method:

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

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

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

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

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

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

- The specified Java method actually has one or more additional, trailing parameters, whose data types must be a Java array of either *ResultSet* or *ResultSetIterator*.

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

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

And, for the JDBC-based orderedEmps:

```
CREATE PROCEDURE ranked_emps (region integer)
```

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

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

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

15.11 Calling the *rankedEmps* procedure

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

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

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

15.12 Overloading Java method names and SQL names

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

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

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

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

Assume that the above classes are in a jar file "~/classes/Over.jar', which you install:

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

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

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

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

```
CREATE FUNCTION odd ( double precision ) RETURNS integer
    EXTERNAL NAME 'over_jar:Over1.testOdd'
    LANGUAGE JAVA PARAMETER STYLE JAVA;
CREATE FUNCTION odd ( timestamp ) RETURNS integer
    EXTERNAL NAME 'over_jar:Over2.isOdd'
    LANGUAGE JAVA PARAMETER STYLE JAVA;
CREATE FUNCTION odd ( date ) RETURNS integer
    EXTERNAL NAME 'over_jar:Over2.oddDateTime'
    LANGUAGE JAVA PARAMETER STYLE JAVA;
CREATE FUNCTION odd ( time ) RETURNS integer
    EXTERNAL NAME 'over_jar:Over2.oddDateTime'
    LANGUAGE JAVA PARAMETER STYLE JAVA;
CREATE FUNCTION is_odd ( integer ) RETURNS integer
    EXTERNAL NAME 'over_jar:Over1.isOdd'
    LANGUAGE JAVA PARAMETER STYLE JAVA;
CREATE FUNCTION test_odd ( real ) RETURNS integer
    EXTERNAL NAME 'over_jar:Over1.isOdd'
    LANGUAGE JAVA PARAMETER STYLE JAVA;
```

Note the following characteristics of these CREATE FUNCTION statements,

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

- The SQL names *is_odd* and *test_odd* are defined on the two *isOdd* methods of *Over1*. I.e. those two different SQL names are defined on the same Java name.

The rules governing overloading are those of the SQL language as defined in SQL/PSM, Subclause 10.18, "<SQL-invoked routine>", and Subclause 9.1, "<routine invocation>". This includes:

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

```
CREATE FUNCTION is_odd ( integer ) RETURNS int ...
CREATE FUNCTION is_odd( smallint ) RETURNS int ...
CREATE PROCEDURE is_odd (smallint) ...
```

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

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

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

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

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

The group of eight example CREATE FUNCTION statements, shown earlier in this clause, do not require Java method signatures, but you can include them for clarity. Clause 15.14, *"Java method signatures in the CREATE* statements*"* describes cases where the Java method signature is required.

15.13 Java *main* methods

The Java Language Specification places special requirements on any method named *main*. A method named main is required to have the following Java method signature:

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

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

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

- Zero or more parameters, each of which is CHAR or VARCHAR. Those N parameters are passed to the Java method as a single N element array of String.

15.14 Java method signatures in the CREATE statements

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

```
public class Routines4  {
    //A String method that will be called as a function
```

```
        public static String job1(Integer jc) throws SQLException {
            if (jc ==  1) return "Admin";
            else if (jc == 2) return "Sales";
            else if (jc == 3) return "Clerk";
            else if (jc == null) return null;
            else return "unknown jobcode";
        }
    }
```

Note: we suffix the method name with a "1" in anticipation of subsequent variants of the method.

Assume that you install this class in SQL:

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

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

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

However, as written above, this CREATE statement is not valid. Note that the data type of the parameter of the Java method job1 is Java *Integer (*which is short for *java.lang.Integer*) , and we have specified the SQL data type *integer* for the corresponding parameter of the SQL *job_of1* function. However, the detailed rules (clause "<SQL-invoked routinhe>*")* for the external Java form of the SQL CREATE PROCEDURE/FUNCTION statement specifies that the default Java parameter data type for an SQL *integer* parameter is the Java *int* data type, not the Java *Integer* data type. Clause 15.15, "*Null argument values and the RETURNS NULL clause*" describes some reasons why you may want to specify Java *Integer* rather than Java *int*.

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

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

If you specify data types in the Java method signature of a CREATE statement that specifies DYNAMIC RESULT SETS, then you must include the implicit trailing result set or iterator parameters in that Java method signature. You do not, however, include those trailing parameters in the SQL signature. For example, you would write the CREATE of clause 15.10, "*A CREATE PROCEDURE rankedEmps for orderedEmps*", as follows:

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

See the clause "<SQL-invoked routine>".

15.15 Null argument values and the RETURNS NULL clause

Consider the Java method, *job1,* and the corresponding SQL function *job_of1*, which we defined in Clause *15.14*, "Java method signatures in the CREATE statements".

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

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

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

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

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

Assume that you install this class in SQL:

```
SQLJ.INSTALL_JAR ('file:~/classes/Routines5.jar', 'routines5_jar', 0)

CREATE FUNCTION job_of2(jc integer) RETURNS varchar(20)
EXTERNAL NAME 'routines5_jar:Routines5.job2'
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

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

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

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

To summarize:

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

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

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

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

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

When you code Java methods specifically for use in SQL, you will probably tend to specify Java parameter data types that are the nullable Java data types. You may, however, also want to use

Java methods in SQL that were not coded for use in SQL, and that are more likely to specify Java parameter data types that are the scalar (non-nullable) Java data types.

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

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

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

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

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

The following SELECT statement invokes the *job_of22* function.

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

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

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

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

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

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

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

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

15.16 Static variables

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

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

Assume that you install this class in an SQL system:

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

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

For example:

```
CREATE PROCEDURE set_jobs(js varchar(100))
EXTERNAL NAME 'routines6_jar:Routines6.setJobs'
LANGUAGE JAVA PARAMETER STYLE JAVA;

CREATE FUNCTION job_of3(jc integer) RETURNS varchar(20)
RETURNS NULL ON NULL INPUT
EXTERNAL NAME 'routines6_jar:Routines6.job3'
LANGUAGE JAVA PARAMETER STYLE JAVA;

CALL set_jobs('AdminSalesClerk');

SELECT name, job_of3(jobcode)
FROM emps
WHERE job_of3(jobcode) <> 'Admin';
```

This appears to be a straightforward use of the *Routines6* class in SQL. The call of *set_jobs* specifies a list of job code values, which the instance "caches" and uses in subsequent calls of *job_of3*. However, since the scope of the static variable *jobs* in the SQL environment is implementation-dependent, the following questions regarding the values passed to the *set_jobs* procedure are likely to differ across implementation:

- Is the *set_jobs* value visible only to the current session? Or also to concurrent sessions and to later non-concurrent sessions?
- Does the *set_jobs* value persist across a COMMIT? Is it reset by a ROLLBACK?

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

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

15.17 Dropping SQL names of Java methods

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

```
DROP function region RESTRICT;
```

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

15.18 Removing Java classes from SQL

You can completely uninstall a jar file with the SQLJ.REMOVE_JAR procedure. For example:

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

As noted earlier, jar files can contain *deployment descriptors*, which specify implicit actions to be taken by the SQLJ.INSTALL_JAR and SQLJ.REMOVE_JAR procedures. The second parameter is an integer that specifies whether you do or do not (indicted by non-zero or zero values) want the REMOVE_JAR procedure to execute the actions specified by a *deployment descriptor* in the jar file. Deployment descriptors are further described in clause 15.22, "*Deployment descriptors*".

After the SQLJ.REMOVE_JAR procedure performs any actions specified by the jar's deployment descriptor file(s), there must be no remaining SQL procedure or function whose external name references any method of any class in the specified jar file. Any such remaining SQL procedures or functions must be explicitly dropped before the SQLJ.REMOVE_JAR procedure will be able to complete successfully.

15.19 Replacing Java classes in SQL

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

- CREATE PROCEDURE/FUNCTION statements referencing the classes.
- GRANT statements referencing those SQL procedures and functions.
- CREATE PROCEDURE/FUNCTION statements for SQL procedures and functions that invoke those SQL procedures and functions.
- CREATE VIEW/TABLE statements for SQL views and tables that invoke those SQL procedures and functions.

The rules for the SQLJ.REMOVE_JAR procedure require that you drop all SQL procedure/functions that directly reference methods of a class before you can remove the jar file containing the class. And, SQL rules for RESTRICT**,** as specified in the SQL <drop routine statement>, require that you drop all SQL objects (tables, views, SQL-server modules, and routines

whose bodies are written in SQL) that invoke a procedure/function before you drop the procedure/function.

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

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

For example, in clause 15.4, "*Installing region and correctStates in SQL*" we installed the class of "Routines1" with the following statement:

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

You can replace that jar file with a statement such as:

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

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

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

The validity requirements on the replacement jar file are:

- There must be no SQL procedure or function name that references any method of any unmatched old class (since all unmatched old classes will be removed).
- Any CREATE PROCEDURE/FUNCTION statement that references a method of a matching class must be a valid statement for the new class.

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

15.20 Visibility

The SQLJ.INSTALL_JAR procedure will install any Java classes into the SQL system. However, not all methods of all classes can be referenced in SQL. Only *visible* methods of *visible* classes can be referenced in SQL. The notion of visible classes and methods is based on the concept of *mappable* data types. The detailed definitions of *mappable* and *visible* are specified in clause 4.5, "Parameter mapping". They may be summarized as follows:

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

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

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

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

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

15.21 Exceptions

SQL exception conditions are defined for the SQL/JRT procedures. For example, if the URL argument specified in calls to SQLJ.INSTALL_JAR or SQLJ.REPLACE_JAR (etc) is invalid, an SQL exception condition (*java.sql.SQLException*) with a specified SQLSTATE will be raised. These exception conditions are specified in the definitions of the procedures, and are listed the clause 12, "Definition schema". Java exceptions that are thrown during execution of a Java method in SQL can be caught within Java, and if this is done, then those exceptions do not affect SQL processing.

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

For example, in the clause "*Example Java methods: region and correctStates"*, we defined the Java method *Routines1.region*. And, in the clause "*Defining SQL names for region and correctStates*" we defined the SQL function name *region_of* for the Java method *Routines1.region*.

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

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

Assume that the *emps* table contains a row for which the value of the *state* column is “TX”. The following SELECT will therefore raise an exception condition when it encounters that row of *emps*:

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

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

“Invalid state code”

The message text and SQLSTATE may be specified in the Java exception specified in the Java **throw** statement. If that exception specifies an SQLSTATE, the first two characters of that SQLSTATE must be “38”. (If this requirement is violated, then the effects are implementation-defined.) If that exception does not specify an SQLSTATE, then the default SQLSTATE for an uncaught Java exception is raised. See the clause "Class and subclass values for SQL/JRT exceptions".

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

15.22 Deployment descriptors

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

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

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

   a) The *afterInstall* method: The CREATE and GRANT statements that you want to be performed when the jar file is installed.

   b) The beforeRemove method: The DROP and REVOKE statements (the inverse of the actions of the *afterInstall* method) that you want to be performed when the jar file is removed.

I.e. the *afterInstall* and *beforeRemove* methods would use *SQL/OLB* or JDBC to invoke SQL for the desired CREATE, GRANT, DROP, and REVOKE statements.

2) Include the *afterInstall* and *beforeRemove* methods in a class, which you might call the *deploy* class, and include that *deploy* class in the jar file that you plan to distribute.

3) Instruct recipients of the jar file to do the following to install the jar file:

   a) Call the SQLJ.INSTALL_JAR procedure for the jar file.

   b) Execute a CREATE procedure statement for the *afterInstall* method, giving it an SQL name such as *after_install*. Note that this "bootstrap" action cannot be included in the *afterInstall* method itself.

   c) Call the *after_install* procedure. Note: We can assume that the *after_install* procedure will include a CREATE PROCEDURE statement to give the *beforeRemove* method an SQL name such as *before_remove*.

4) Instruct recipients of the jar file to proceed as follows to remove the jar file:

   a) Call the *before_remove* procedure.

   b) Drop the *after_install* and *before_remove* procedures. Note that this action cannot be included in the *beforeRemove* procedure itself.

   c) Call the SQLJ.REMOVE_JAR procedure.

Note that this simplification of the install and remove actions still requires several manual steps. SQL/JRT therefore provides a mechanism, called *deployment descriptors*, with which you can specify the SQL statements that you want to be executed implicitly by the SQLJ.INSTALL_JAR and SQLJ.REMOVE_JAR procedures.
If you want the deployment descriptors in a jar file to be interpreted when you install and remove the jar file, then you specify a non-zero value for the *deploy* parameter of the SQLJ.INSTALL_JAR procedure and similarly for the *undeploy* parameter of the SQLJ.REMOVE_JAR procedure. If a jar file contains a deployment descriptor, then the SQLJ.INSTALL_JAR procedure will use that deployment descriptor to determine the CREATE and GRANT statements to execute after it has installed the classes of the jar file. The corresponding SQLJ.REMOVE_JAR procedure uses the deployment descriptor to determine the DROP and REVOKE statements to execute before it removes the jar file and its classes.
A deployment descriptor is a text file containing a list of SQL CREATE and GRANT statements to be executed when the jar file is installed, and a list of SQL DROP and REVOKE statements to be executed when the jar file is removed.
For example, suppose that you have incorporated the above classes *Routines1*, *Routines2*, and *Routines3* into a single jar file. The following is a possible deployment descriptor that you might want to include in that jar file.

Notes:

i) Within a deployment descriptor file, you use the jar name "*thisjar*" as a placeholder jar name in the EXTERNAL NAME clauses of CREATE statements.

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

```
SQLActions[ ] = {
        "BEGIN INSTALL
                CREATE PROCEDURE correct_states
                        (old CHAR(20), new CHAR(20))
                        MODIFIES SQL DATA
                        EXTERNAL NAME 'thisjar:Routines1.correctStates'
                        LANGUAGE JAVA PARAMETER STYLE JAVA;
                GRANT EXECUTE ON correct_states TO Baker;

                CREATE FUNCTION region_of(state CHAR(20)) RETURNS integer
                        NO SQL
                        EXTERNAL NAME 'thisjar:Routines1.region'
                        LANGUAGE JAVA PARAMETER STYLE JAVA;
                GRANT EXECUTE ON region_of TO PUBLIC;

                CREATE PROCEDURE best2
                        (OUT n1 VARCHAR(50), OUT id1 CHAR(5),
                        OUT region1 INTEGER, OUT s1 DEC(6,2),
                        OUT n2 VARCHAR(50), OUT id2 CHAR(5),
                        OUT region2 INTEGER, OUT s2 DEC(6,2),
                        region INTEGER)
                        READS SQL DATA
                        EXTERNAL NAME 'thisjar:Routines2.bestTwoEmps'
                        LANGUAGE JAVA PARAMETER STYLE JAVA;
                GRANT EXECUTE ON best2 TO Baker, Cook, Farmer;

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

```
            GRANT EXECUTE ON ordered_emps TO PUBLIC;
        END INSTALL ",
        "BEGIN REMOVE
            REVOKE EXECUTE ON correct_states FROM Baker RESTRICT;
            DROP PROCEDURE correct_states RESTRICT;

            REVOKE EXECUTE ON region_of FROM PUBLIC RESTRICT;
            DROP FUNCTION region_of RESTRICT;

            REVOKE EXECUTE ON best2
                FROM Baker, Cook, Farmer RESTRICT;
            DROP PROCEDURE best2 RESTRICT;

            REVOKE EXECUTE ON ordered_emps FROM PUBLIC RESTRICT;
            DROP PROCEDURE ordered_emps RESTRICT;
        END REMOVE "
    }
```

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

1) The text file *deploy_routines.txt*.
2) The class files for *Routines1*, *Routines2*, and *Routines3*.
3) A manifest file with the following manifest entry:

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

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

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

You may want to install a jar file that contains a deployment file without performing the deployment actions. For example, those actions may contain syntax errors, or may simply be inappropriate for some SQL system. You can do this by specifying a zero value for the *deploy* parameter of the SQLJ.INSTALL_JAR procedure, and a zero value for the *undeploy* parameter of the SQLJ.REMOVE_JAR procedure.

15.23 Paths

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

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

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

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

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

To summarize:

- When you install a jar file, any references within the classes of that jar file to system classes, or to classes that are contained in the same jar file, will be implicitly resolved.
- References to any other classes, installed or not, are unresolved.
- You can install jar files that have unresolved class references, and you can use CREATE PROCEDURE/FUNCTION statements to define SQL routines on the methods of those classes.
- When you call SQL routines defined on Java methods, exceptions for unresolved class references may occur at any time allowed by the Java language specification.

Invoking classes that contain unresolved references can be useful:

- To use or to test partially-written applications.
- To use classes that have some methods that are not appropriate for use in an SQL environment. E.g. a class that has display-oriented or interactive methods that are used in other Java-enabled environments, but not within an SQL system.

To resolve references to classes in other jar files, you use the SQLJ.ALTER_JAVA_PATH procedure.

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

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

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

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

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

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

- Each path element "(PATi, Ji)" is examined.

- If PATi is a fully qualified class name that is equal to XC, then XC must be defined in jar Ji. If it is not, then the reference to XC is unresolved.

- If PATi is a sequence of package name qualifiers followed by an "*", and XC begins with the same package name qualifiers, then XC must be defined in jar Ji. If it is not, then the reference to is unresolved.

- If PATi is a single "*", then if XC is defined in jar Ji, that resolution is used; otherwise, subsequent path elements are tested.

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

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

- When executing within the *property_jar*, classes that are in the "project" package will be resolved in the *project_jar*.

- When executing within the *project_jar*, all classes will first be resolved in the *property_jar*, and then in the *admin_jar*.

Note that if a class C contained in the *property_jar* directly contains a reference to a class AC contained in the *admin_jar*, then that reference to AC will be unresolved, since the *admin_jar* is not specified in the path for the *property_jar*. But, if that class C invokes a method *project.C2.M* of a class contained in the *project_jar*, and *project.C2.M* references class AC, then that reference to AC will be resolved in the *admin_jar*, since the *admin_jar* is specified in the path for the *project_jar*.

I.e. while class C is being executed, the path specified for the *property_jar* is used, and while class C2 is being executed, the path specified for the *project_jar* is used. Thus, as execution transfers to classes contained in different jar files, the current path changes to the path specified for each such jar file. In other words, the path specified for a jar file J1 applies only to class references that occur directly within the classes of J1, not to class references that occur in some class contained in another jar file that is invoked from a class of J1.

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

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

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

The path facility is an optional feature.

16. ANNEX (INFORMATIVE) --- TYPES TUTORIAL

16.1 Overview

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

16.2 Example Java classes

This clause shows example Java classes *Address* and *Address2Line*.

- The *Address* class represents street addresses in the USA, with a *street* field containing a street name and building number, and a *zip* field containing a postal code.
- The *Address2Line* class is a subclass of the Address class. It adds one additional field, named line2, which would contain data such as an apartment number.
- The *Address* and *Address2Line* classes both have the following methods:
  - A default niladic constructor.
  - A constructor with parameters.
  - A *toString* method to return a string representation of an address.
- The *Address* and *Address2Line* classes are both specified to implement the Java interfaces *java.io.Serializable* and *java.sql.SQLData*.

> A Java class that will be used as a data type in SQL 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);
```

```
        }
}
```

16.3 Installing Address and Address2Line in an SQL system

To install classes such as *Address* and *Address2Line* in an SQL system, you proceed as in *SQL/JRT: 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 *SQL/JRT: SQL Routines*:

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

16.4 CREATE TYPE for Address and Address2Line

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

An implementation of *SQL/JRT: 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 SQL/JRT procedure SQLJ.INSTALL_JAR that specify a deploy action (third parameter non-zero). This is described in clause 15.22, "Deployment descriptors". In this Tutorial clause, 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 clause we'll describe the basic elements of these CREATE TYPE statements, and defer to later clauses 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 clause 16.19, "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 is treated similarly. This is described in clause 16.20, "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 its identified installed jar. This is referred to as the *subject Java class,* and the SQL data type is the *subject SQL data type*.

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

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

16.5 Multiple SQL types for a single Java class

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

```
CREATE TYPE another_addr
        EXTERNAL NAME 'address_classes_jar:Address'
        LANGUAGE JAVA
        AS( zip_part 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 data type *another_addr* is a different data type than the *addr* data type. The two data types aren't comparable, assignable, or union compatible. You can include or omit an SQL data type that is a subtype of the *another_addr* type for "2 line" data. If you define such a subtype, with a name such as *another_2_line*, then instances of *another_2_line* are specializations of *another_addr*, and not of *addr*.

16.6 "Collapsing" subclasses

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

Assume that in SQL you only want to use the Java class *Address2Line*. You can define an SQL data type for that class, without a corresponding SQLdata type 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 data types for the following subsets of the classes:

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

- Any one of *Person*, *Employee*, *Manager*, or *Director*. That type can include members from any of its superclasses, whether immediate or indirect.

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

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

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

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

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

- *Person* and *Manager*, or *Person*, *Manager*, and *Director*.

- *Person* and *Director*.

- *Person*, *Employee*, and *Director*, or *Employee and Director*.

The rule is simpler than the explanation:

> If a CREATE TYPE statement for SQL type S2 specifies "UNDER S1", then the subject Java class of S1 must be the immediate superclass of the subject Java class of S2.

Clause 16.5, "Multiple SQL types for a single Java class" described how you can define multiple SQL data types on a single Java class. This also can be done for subtype hierarchies. For example, let P*i*, E*i*, M*i*, and D*i* be SQL data types defined on Person, Employee, Manager, and Director. For a given number "*i*" each type is defined to be a subtype of the preceding "*i*" type. You can define SQL data types such as:

- E1 and M1, and P2 and E2. 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.

16.7 GRANT and REVOKE statements for data types

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

```
GRANT USAGE ON TYPE addr TO PUBLIC;
GRANT USAGE ON TYPE addr2line TO admin;
```

The syntax and semantics for GRANT and REVOKE of the USAGE privilege for user-defined types are as specified in ISO/IEC 9075, and are not further described by this standard.

16.8 Deployment descriptors for classes

You may want to perform the same set of SQL CREATE and GRANT statements in any SQL system in which you install a given jar 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"
}
```

16.9 Using Java classes as data types

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

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

Alternatively, if the SQL/JRT implementation supports typed tables as specified in ISO/IEC 9075, 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. ISO/IEC 9075 supports additional generation mechanisms for object identifiers, which can be defined through extended syntax in the CREATE TYPE statement (see The subclause "<user-defined type definition>" and the ISO/IEC 9075 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.

16.10 SELECT, INSERT, and UPDATE

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

Values of SQL/JRT 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') )
```

16.11 Referencing Java fields and methods in SQL

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

```
SELECT  home_addr.to_string( ), mailing_addr.to_string( )
FROM emps
WHERE name = 'Bob Smith';


SELECT name, home_addr.zip_attr
FROM emps
WHERE  home_addr.street_attr= '456 Shoreline Drive';


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

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

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


SELECT name, mailing_addr->street_attr
        FROM emps2
        WHERE mailing_addr->zip_attr = '99111';
```

16.12 Extended visibility rules

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

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

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


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.

16.13 Logical representation of Java instances in SQL

We saw in clause 16.10, "SELECT, INSERT, **and** UPDATE" that the values assigned to such SQL/JRT columns are assigned from other SQL/JRT columns or from the results of calling Java constructors or other methods. Hence, the values assigned to SQL/JRT columns are ultimately derived from values constructed by Java methods in the Java VM. Such values are represented in SQL/JRT 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 data type 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 SQL/JRT 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 SQL/JRT 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 SQL/JRT 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 data type of the function is the *addr* data type. 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 data type *Address*, which corresponds with the SQL data type *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.

16.14 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.

SQL/JRT 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.

16.15 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).

16.16 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 SQL/JRT 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.

16.17 Developing for Portability

The following guidelines provide maximum portability of Java classes across different SQL/JRT 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.

16.18 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 *SQL/JRT: 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;
```

16.19 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.

16.20 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;
```

3) As described in clause 16.13, “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 data type. 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.

16.21 Subtypes in SQL/JRT data

Recall the example Java classes *Address* and *Address2Line*., and the corresponding SQL data types *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 data type. 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.

16.22 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 SQL/JRT 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.*

16.23 Ordering of SQL/JRT data

In an earlier clause we created the *emps* table, with columns *home_addr* and *mailing_addr* whose data types 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 data type with a CREATE TYPE…EXTERNAL LANGUAGE JAVA statement, the new external Java data type has no ordering capability. I.e. its "*ordering form*" is "*none*". Instances of an external Java data type whose ordering form is none cannot be used in any of the above ordering relationships.

To define ordering for an external Java data type, you use the CREATE ORDERING statement:

```
<create ordering statement> ::=
        CREATE ORDERING FOR <user-defined type 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 data type, 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.

Annex: Types tutorial

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 data types 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.

17. ANNEX (INFORMATIVE) --- SQL/JRT FEATURE TAXONOMY

This Annex describes a taxonomy of features defined in this part of ISO/IEC 9075.

Table 17-1 SQL/JRT feature taxonomy contains a taxonomy of the features of SQL/JRT. In that table, the first column contains a counter that may be used to quickly locate rows of the table; these values otherwise have no use and are not stable - that is, they are subject to change in future editions of or even Technical Corrigenda without notice.

The "Feature ID" column of *Table 17-1 SQL/JRT feature taxonomy* specifies the formal identification of each feature and each subfeature contained in the table. The Feature ID is stable and can be depended on to remain constant. A Feature ID value comprises either a letter and three digits or a letter, three digits, a hyphen, and one or two additional digits. Feature ID values containing a hyphen and additional digits indicate "subfeatures" that help to define complete features, which are in turn indicated by Feature ID values without a hyphen. Only entire features are used to specify the contents of Core SQL and various packages.

The "Feature Description" column of *Table 17-1 SQL/JRT feature taxonomy* contains a brief description of the feature or subfeature associated with the Feature ID value.

Index	Feature ID	Feature Name	Feature Description
	J511	Commands	The following subclauses specify SQL statements: The subclause “<SQL-invoked routine>”. The subclause “<user-defined type definition>“.
	J521	JDBC data types	The subclause “<SQL-invoked routine>”. specifies JDBC data type clauses for CREATE PROCEDURE.
	J531	Deployment	The following subclauses specify SQL statements that can be specified in deployment files: The subclause “<SQL-invoked routine>”. The clause "*DROP PROCEDURE/FUNCTION statement*". The subclause “<user-defined type

			definition>“. The clause "<*user-defined ordering definition*>". The clause *"<drop user-defined ordering statement>".*
	J541	Serializable	The subclause “<user-defined type definition>“ *specifies the SERIALIZABLE clause of CREATE TYPE*.
	J551	SQLDATA	The subclause “<user-defined type definition>“ specifies the SQLDATA clause of CREATE TYPE.
	J561	Jar privileges	The clause "*GRANT statement*" specifies GRANT for jar files. The clause "*REVOKE statement*" specifies REVOKE for jar files. The clause “<privileges>” defines a JAR form of <object name> for granting or revoking USAGE on installed JARs.
	J571	New	The clause "*SQL/JRT method call*" specifies the NEW operator.
	J581	Output parameters	The subclause “<SQL-invoked routine>” specifies output parameter clauses for CREATE PROCEDURE.
	J591	Overloading	The subclause “<user-defined type definition>“ specifies CREATE TYPE statements for overloaded methods.
	J601	SQL-Java paths	The clause "*SQL/JAVA paths*" specifies Java paths.

			The clause "*SQLJ.ALTER_JAVA_PATH procedure*" specifies alter Java path.
	J611	References	The clause "*SQL/JRT member references*" specifies references. The subclause “<user-defined type definition>“ specifies CREATE TYPE with reference parameters.
	J631	Java signatures	The subclause “<SQL-invoked routine>” specifies CREATE PROCEDURE statements with result sets.
	J641	Static fields	The clause "*CREATE TYPE statement* " specifies CREATE TYPE statements with static field methods.

Table 17-1 SQL/JRT feature taxonomy

18. ANNEX (INFORMATIVE) --- IMPLEMENTATION-DEFINED ELEMENTS

To be added.

19. ANNEX (INFORMATIVE) --- IMPLEMENTATION-DEPENDENT ELEMENTS

To be added.

20. ANNEX (INFORMATIVE) --- INCOMPATIBILITIES WITH NCITS 331.1 AND 331.2

1) <jar name> is defined with the SQL '99 bnf non-terminal <schema name> which contains the optional <catalog name>. Rules for determining the value of unspecified <catalog name> and <schema name> (which 331.1 and 331.2 refer to as <catalog id> and <schema id>) now match those of SQL '99. Currently, 331.1 and 331.2 <catalog id> and <schema id> default, respectively to the *current catalog* and *current schema* (however an implementation determines them). This change makes rules applicable to, e.g., <table name>'s default catalog and schema determination apply to <jar name>s. See SQL/Foundation subclause 5.4 'Names and identifiers' for details.

2) The rules for JARs, that their catalog and schema must, whether explicitly of implicitly specified, match the current catalog and schema are removed.

3) [Found] places restrictions on ORDERINGs that are not specified by SQLJ. E.g., an ORDERING cannot be created for a type that already has one. Or, e.g., if Manager is a child of Employee and if Employee only supports EQUALS (equality/inequality) comparison then Manager is restricted to only supporting equality/inequality comparison. Likewise if Employee supports FULL comparisons, any ORDERING for Manager must also support FULL comparison.
(For more info on the restrictions, which are of the same 'flavor' as the above, see [Found].)

4) [Found] supports a DROP ORDERING statement not described by SQLJ Part 2. Given that CREATE ORDERING is allowed for sqlj.install_jar, it makes sense that DROP ORDERING be allowed in Deployment Descriptors for sqlj.remove_jar. (Assuming, of course, that the SQL environment claims to support feature J531. "Deployment".

5) Note that SQLJ Part 2's <create ordering statement> doesn't specify a Conformance Rule regarding using <create ordering statement>s in Descriptor Files. This paper corrects that oversight.

6) In aligning with the CREATE ORDERING statement, [Found]'s <user-defined ordering definition> makes use of <specific routine designator> which is more complex than what SQLJ Part 2 allows for in referencing existing functions.

7) GRANT and REVOKE of USAGE on JARs will support SQL:1999 WITH GRANT and GRANTED BY options, not allowed for in NCITS 331.1 and 331.2. GRANT and REVOKE are no longer limited to only being executed by the owner of the specified JAR and the USAGE privilege on a JAR can also be granted to roles.

8) The semantics of REVOKE JAR USAGE…CASCADE is defined. Note: CASCADE only has to be supported if a vendor claims to support Feature F034, "Extended REVOKE statement". Without that feature, a <drop behavior> of CASCADE shall not be specified in <revoke statement>.

9) In CREATE PROCEDURE/FUNCTION, add SQL/Foundation's support of <routine characteristic>s 'SPECIFIC <specific name>', and '<transform group specification>' to the bnf nonterminal <routine characteristic>.
   The <transform group specification>'s determination of to-sql and from-sql functions is based on that specified for PARAMETER STYLE GENERAL.
   Note: while <specific name> is required in Core SQL/99, only vendors supporting Feature S241, ''Transform functions'' have to support <transform group specification>.

10) In CREATE PROCEDURE/FUNCTION, add SQL/Foundation's <external security clause>.
    Note: only vendors supporting Feature T323, ''Explicit security for external routines'', have to support <external security clause>.

11) In CREATE PROCEDURE/FUNCTION, clarify SQL/Foundation's support of parameters that are SQL arrays.
    SQL/Foundation includes at least two Array-related Features: Feature T571, ''Array- returning external SQL-invoked functions'', and S201, ''SQL routines on arrays''. Feature T571 isn't applicable as 11.49, '<SQL-invoked routine>' SR 5)t)ii) says "If *R* is an array- returning external function, then PARAMETER STYLE SQL shall be either specified or implied"; that leaves S201 which says "Without Feature S201, ''SQL routines on arrays'', a <parameter type> shall not be an array type."
    Note: there are already hints that SQLJ procedures support SQL arrays, e.g., Feature J501, "SQL arrays", and discussion of supporting the Java procedure 'main', however rules for mapping to/from Java and SQL collection types are not provided. This paper attempts to define their semantics.

12) In CREATE PROCEDURE/FUNCTION, add SQL/Foundation's support of <result cast>.

13) In CREATE PROCEDURE/FUNCTION, add SQL/Foundation's <dispatch clause>.
    Note: <dispatch clause> is (as far as I can tell) a sometimes-required noise clause applicable to SQL-invoked functions whose parameter list includes a user-defined type or a ref type.

14) SQL/Foundation provides SQL-path, routine SQL-path, and external routine SQL-path, not explicitly discussed by SQL/JRT. It is unclear to what extent, if any, they induce any incompatibilities.

15) In DROP PROCEDURE/FUNCTION, add SQL/Foundation's support of <drop behavior>.
    Note that without Feature F032, ''CASCADE drop behavior'', a <drop behavior> of CASCADE shall not be specified in <drop routine statement>.

16) Unless explicit additions are made, by default SQL/JRT will support SQL/Foundation's <alter routine statement>. So far, those additions have not been made; though perhaps they should be as <alter routine statement> is nowhere discussed in SQLJ Parts 1 and 2.

17) Technical Corrigenda against SQL/Foundation added the ability for a CREATE TYPE statement to explicitly state that a method is a CONSTRUCTOR method. I.e., a method may now be an INSTANCE method, a STATIC method, or a CONSTRUCTOR method. Unless explicit additions are made, by default SQL/JRT will need to support SQL/Foundation's CONSTRUCTOR methods. Note, at the current moment this capability is being allowed

(pending objections, and pending additional research with regards to the extent to which constructor methods 'fall out' for free for SQL/JRT).

18) SQL/JRT adds support of a <drop data type statement>s' specification of the <drop behavior> CASCADE. Note that without Feature F032, ''CASCADE drop behavior'', a <drop behavior> of CASCADE shall not be specified in <drop data type statement>.

One may wonder how the choice of which features of SQL/Foundation should be supported and which ones shouldn't was made. It isn't completely arbitrary! While not claiming perfect consistency, one consideration was whether or not there is a SQL/Foundation 'Feature' associated with the specific capability; if yes it is more likely that the capability be allowed in SQL/JRT on the same basis. Another consideration was whether or not the capability impacts the underlying Java to SQL interface rather than being equally applicable to all external routines. If the Java to SQL interface is impacted then it is more likely that the feature Not be allowed in SQL/JRT. Impacting that interface is beyond the scope of this 'rewrite.' E.g., support of locator parameters is an example of a capability that seems to impact the Java to SQL interface, whereas the ability to specify a specific-name for an external routine is external language independent.

Index

To be added.