INTERNATIONAL STANDARD

**ISO/IEC 9075-10**

Fourth edition
2016-12-15

# Information technology — Database languages — SQL —

## Part 10: Object language bindings (SQL/OLB)

*Technologies de l'information — Langages de base de données — SQL —*

*Partie 10: Liaisons de langage objet (SQL/OLB)*



Reference number
ISO/IEC 9075-10:2016(E)



ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

# Contents

 

 

 

 

 

 

 

# Tables

## Table

**Page**

*(Blank page)*

# Foreword

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

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

The committee responsible for this document is ISO/IEC JTC 1, *Information technology*, SC 32, *Data management and interchange*.

This fourth edition of ISO/IEC 9075-10 cancels and replaces the third edition (ISO/IEC 9075-10:2008), which has been technically revised. It also incorporates Technical Corrigendum ISO/IEC 9075-10:2008/Cor.1:2010.

A list of all parts in the ISO/IEC 9075 series, published under the general title *Information technology — Database languages — SQL*, can be found on the ISO website.

NOTE The individual parts of multi-part standards are not necessarily published together. New editions of one or more parts can be published without publication of new editions of other parts.

# Introduction

The organization of this Part of this International Standard is as follows:

1) Clause 1, “Scope”, specifies the scope of this part of ISO/IEC 9075.

2) Clause 2, “Normative references”, identifies additional standards and publically-available specifications that, through reference in this part of ISO/IEC 9075, constitute provisions of this part of ISO/IEC 9075.

3) Clause 3, “Definitions, notations, and conventions”, defines the notations and conventions used in this part of ISO/IEC 9075.

4) Clause 4, “Concepts”, presents concepts used in the definition of the Object Language Bindings.

5) Clause 5, “Lexical elements”, defines the lexical elements of the language.

6) Clause 6, “Scalar expressions”, defines the elements of the language that produce scalar values.

7) Clause 7, “Additional common elements”, defines additional language elements that are used in various parts of the language.

8) Clause 8, “Embedded SQL”, defines the host language embeddings.

9) Clause 9, “SQLJ reserved names”, defines the reserved names for SQLJ.

10) Clause 10, “Common subelements”, defines the commonly used subelements for SQLJ.

11) Clause 11, “<SQLJ specific clause> and contents”, defines the syntax and rules for SQLJ constructs.

12) Clause 12, “Package sqlj.runtime”, specifies the SQLJ runtime package.

13) Clause 13, “Package sqlj.runtime.profile”, specifies the SQLJ runtime profile package.

14) Clause 14, “sqlj.runtime.profile.util.ProfileCustomizer”, specifies the SQLJ profile customizer class.

15) Clause 16, “Status codes”, defines SQLSTATE values related to Object Language Bindings.

16) Clause 17, “Conformance”, defines the criteria for conformance to this part of ISO/IEC 9075.

17) Annex A, “SQL Conformance Summary”, is an informative Annex. It summarizes the conformance requirements of the SQL language.

18) Annex B, “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) Annex C, “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) Annex D, “Deprecated features”, is an informative Annex. It lists features that the responsible Technical Committee intend will not appear in a future revised version of this part of ISO/IEC 9075.

21) Annex E, “Incompatibilities with ISO/IEC 9075:2008”, is an informative Annex. It lists incompatibilities with the previous version of this part of ISO/IEC 9075.



 

22) Annex F, “SQL feature taxonomy”, is an informative Annex. It identifies features of the SQL language specified in this part of ISO/IEC 9075 by an identifier and a short descriptive name. This taxonomy is used to specify conformance.

23) Annex G, “Defect reports not addressed in this edition of this part of ISO/IEC 9075”, is an informative Annex. It describes the Defect Reports that were known at the time of publication of this part of this International Standard. Each of these problems is a problem carried forward from the previous edition of ISO/IEC 9075. No new problems have been created in the drafting of this edition of this International Standard.

In the text of this part of ISO/IEC 9075, Clauses and Annexes begin new odd-numbered pages. Any resulting blank space is not significant.

All Clauses of this part of ISO/IEC 9075 are normative.

*(Blank page)*



 

# Information technology — Database languages — SQL —

Part 10:
**Object Language Bindings (SQL/OLB)**

## 1 Scope

ISO/IEC 9075-2 specifies embedded SQL for the programming languages: Ada, C, COBOL, Fortran, MUMPS, Pascal, and PL/I. This part of ISO/IEC 9075 defines similar features of Database language SQL that support embedding of SQL-statements into programs written in the Java™ programming language (Java is a registered trademark of Sun Microsystems, Inc.). The embedding of SQL into Java is commonly known as "SQLJ". This part of ISO/IEC 9075 specifies the syntax and semantics of SQLJ, as well as mechanisms to ensure binary portability of resulting SQLJ applications. In addition, it specifies a number of Java packages and their contained classes (including methods).

Throughout this part of ISO/IEC 9075, the terms "SQLJ" and "SQL/OLB" are used synonymously.

NOTE Additional explanatory material (non-normative) about certain facilities defined in ISO/IEC 9075-2 can be found in ISO/IEC TR 19075-3.

*(Blank page)*

# 2 Normative references

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

## 2.1 ISO and IEC standards

[ISO9075-1] ISO/IEC 9075-1:2016, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*.

[ISO9075-2] ISO/IEC 9075-2:2016, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*.

## 2.2 Other international standards

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

[JLS] *The Java™ Language Specification, Java SE 7 Edition*.
http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf.

[JDBC] *JDBC™ 4.1 Specification*
http://download.oracle.com/otn-pub/jcp/jdbc-4_1-mrel-spec/-jdbc4.1-fr-spec.pdf.

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

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

*(Blank page)*

# 3 Definitions, notations, and conventions

*This Clause modifies Clause 3, "Definitions, notations, and conventions", in ISO/IEC 9075-2.*

## 3.1 Definitions

*This Subclause modifies Subclause 3.1, "Definitions", in ISO/IEC 9075-2.*

### 3.1.1 Definitions provided in Part 10

For the purposes of this document, the following definitions apply:

**3.1.1.1 accessor method**
method that, when invoked, accesses column data returned by a result set iterator object

NOTE 2 — An *accessor method* is either a *named accessor method* or a *positioned accessor method*. A named accessor method is declared as the result of an <iterator declaration clause> containing an <iterator spec declaration> of <named iterator>. A named accessor method derives both its name and its result datatype from its defining <named iterator> clause. A positioned accessor method is declared as the result of an <iterator declaration clause> containing an <iterator spec declaration> of <positioned iterator>. A positioned accessor method derives its result datatype from its defining <positioned iterator> clause.

**3.1.1.2 customization**
implementation-specific process of tailoring an SQLJ application's embedded SQL to run against a target SQL-implementation

NOTE 3 — This frequently involves creating new versions of some of the object instances stored in the SQLJ translation-generated profile, to create customized profile object instances.

**3.1.1.3 generated connection class**
class whose methods, when invoked, maintain a named SQL-connection

NOTE 4 — The signature of this class is produced as a side effect of the direct inclusion of a <connection declaration clause> in a program written in the Java programming language.

**3.1.1.4 generated iterator class**
class whose methods, when invoked, provide access to the rows and columns of SQL queries associated with result set iterators

NOTE 5 — A *generated iterator class* is either a *generated named iterator class* or a *generated positioned iterator class*. The signature of a generated named iterator class is produced as a side effect of the inclusion of a <iterator declaration clause> that contains an <iterator spec declaration> of <named iterator>, and it specifies named accessor methods. The signature of a generated positioned iterator class is produced as a side effect of the inclusion of a <iterator declaration clause> that contains an <iterator spec declaration> of <positioned iterator>, and it specifies positioned accessor methods.

**3.1.1.5 getter method**

method defined on objects of either the RTStatement or RTResultSet class or a subclass of such a class, and that when invoked populates host variables of a given datatype when those host variables appear as bind variables

**3.1.1.6 implementation-specific**
possibly differing between SQL-implementations, but provided as part of each particular SQL-implementation

**3.1.1.7 installation (of an SQLJ application)**
implementation-defined, and possibly empty, phase that includes anything other than *SQLJ translation* and *customization* needed prior to the SQLJ application being able to execute against its target SQL-implementation

**3.1.1.8 Java primitive datatype**
one of the following Java types: boolean, byte, short, int, long, float, or double

NOTE 6 — For interoperability with JDBC, the Java primitive datatype char is intentionally omitted from this list.

**3.1.1.9 l-valued expression**
Java expression that is allowed to appear as the *LeftHandSide* of a Java *assignment*, as defined in [JLS]

NOTE 7 — For example, an l-valued expression may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access or an array access.

**3.1.1.10 profile**
Java serialized object produced by an SQLJ translator, containing information regarding the input required and output generated by individual SQL-statements, as well as the text of those statements

NOTE 8 — The serialized objects can then be accessed for additional processing by a customizer or by the SQLJ runtime system.

**3.1.1.11 profile file**
file containing one or more profiles generated as a result of an SQLJ translation

**3.1.1.12 runtime library component**
implementation of one or more of the classes, interfaces, and abstract classes defined in this document for use during execution of the SQLJ application

NOTE 9 — This could be a default, JDBC-based, implementation (e.g., an implementation of RTStatement using the `java.sql.Statement` interface), or an implementation-specific version used only after customization.

**3.1.1.13 setter method**
method defined on objects of the RTStatement class or a subclass of RTStatement, that when invoked passes bind variables of the given datatype as input parameters to the SQL-implementation

**3.1.1.14 SQLJ file**
file with the suffix '.sqlj' (or file type 'sqlj') containing Java source language or SQLJ constructs, that is input to SQLJ translation

**3.1.1.15 SQLJ translation**
process of transforming a Java application program containing embedded SQL into two or more different files, one identical to the original Java application except that use of embedded SQL is replaced with Java code invoking SQLJ's runtime API, and the others being profile files

**3.1.1.16 SQLJ translation unit**



 

collection of user-defined classes and property files defined or referenced by a given SQLJ file, and the translator-generated classes and profile files that result after that SQLJ file is processed by an SQLJ translator, and its resulting Java source file is compiled

## 3.2 Conventions

*This Subclause modifies Subclause 3.3, "Conventions", in ISO/IEC 9075-2.*

Insert this paragraph In addition, bolding is used when a term is first introduced in this part of ISO/IEC 9075.

### 3.2.1 Use of terms

*This Subclause modifies Subclause 3.3.1, "Use of terms", in ISO/IEC 9075-2.*

#### 3.2.1.1 Other terms

*This Subclause modifies Subclause 3.3.1.1, "Other terms", in ISO/IEC 9075-2.*

Insert this paragraph In this document, the word "object" is used in phrases of the form "a *java type* object", where *java type* is the name of a Java class or interface. If *java type* is the name of a Java class, then this phrase is meant to denote a Java object that is either an instance of the class *java type*, meaning that it has been created by invocation of a constructor of the class *java type*, or an instance of one of the subclasses of class *java type*.

Insert this paragraph If *java type* is the name of a Java interface, then this phrase denotes an object that is one of the following:

— An instance of a class that implements the interface *java type*.

— An instance of a class that implements an interface that extends the interface *java type*.

— An instance of a subclass of such class.

Insert this paragraph The following denotations also hold throughout this International Standard:

— A *connection context object* is an instance of a class that is generated by an SQLJ translator as the result of processing a <connection declaration clause>. The generated class implements the interface `sqlj.runtime.ConnectionContext`.

— An *execution context object* is an instance of the class `sqlj.runtime.ExecutionContext`.

— A *named iterator* is an instance of a class that is generated by an SQLJ translator as the result of processing an <iterator declaration clause> that contains a <named iterator>. The generated class implements the interface `sqlj.runtime.NamedIterator`.

— A *positioned iterator* is an instance of a class that is generated by an SQLJ translator as the result of processing a <iterator declaration clause> that contains a <positioned iterator>. The generated class implements the interface `sqlj.runtime.PositionedIterator`.

— An *iterator* is either a named iterator or a positioned iterator.

### 3.2.2 Specification of translator-generated classes

The conventions used in this document are defined in ISO/IEC 9075-1, with the following additions. Descriptions of translator-generated classes and their relationships to syntax elements contained in <executable spec clause> specified in the Subclauses of Clause 10, "Common subelements", and Clause 11, "<SQLJ specific clause> and contents", are specified in terms of:

— **Function**

Describes the purpose of the syntax element or translator-generated class.

— **Signature**

Defines the client-visible signature of the translator-generated class.

— **Definitions and Rules**

Defines the semantic rules for the syntax element or translator-generated class.

— **Profile EntryInfo Properties**

Defines the properties of the profile EntryInfo object created for this syntax element, if any. If an EntryInfo Java field is not listed explicitly in this heading, it defaults to the value described in Subclause 4.3.5.1, "EntryInfo overview".

— **Binary Composition**

Defines additional methods and/or calls to include for binary composition in the code generated for this clause, if any.

The property of binary composition states that each of the elements defined by <SQLJ specific clause>s can interoperate with elements defined in other <SQLJ specific clause>s, even if the <SQLJ specific clause>s are translated with different SQLJ translators. The **Binary Composition** headings define the minimal set of expected behavior for each <SQLJ specific clause> to achieve this interoperability.

The requirements defined in these headings augment the requirements defined in the **Signature** headings.

— **Code Generation**

Defines the runtime calls made to the ConnectedProfile, RTStatement, and RTResultSet interfaces of the **sqlj.runtime.profile** package by the various <SQLJ specific clause>s. These interfaces are implemented by implementation-specific runtime packages, and thus the calls made to them shall be uniformly specified.

These headings are distinguished from the **Binary Composition** headings in that they specify the internal implementation of the <SQLJ specific clause>s, whereas the **Binary Composition** headings define additions to the client visible signature of the <SQLJ specific clause>s.

The **Code Generation** headings only specify the calls that shall eventually be made to the aforementioned interfaces for each <SQLJ specific clause>. Unless otherwise stated, these headings do not specify the exact SQLJ translation of each <SQLJ specific clause>. An SQLJ translator is free to translate each <SQLJ specific clause> using any number of intermediate calls or helper classes, so long as the methods are eventually called as specified in these headings.

Not all term headings are used in all Subclauses containing descriptions of translator-generated classes or of syntax elements contained by <executable spec clause>.



 

# 4 Concepts

*This Clause modifies Clause 4, "Concepts", in ISO/IEC 9075-2.*

## 4.1 Embedded syntax

*This Subclause modifies Subclause 4.29, "Embedded syntax", in ISO/IEC 9075-2.*

Replace the last sentence of the lead text of the 1st paragraph For all <embedded SQL host program>s other than <embedded SQL Java program>s, such a hybrid compilation unit is defined to be equivalent to:

Insert after the 1st paragraph When the <embedded SQL host program> immediately contains an <embedded SQL Java program>, such a hybrid compilation unit is defined to be equivalent to a compilation unit in which the SQL-statements have been replaced by use of Java classes whose methods, when invoked, make use of JDBC.

## 4.2 Character strings

*This Subclause modifies Subclause 4.2, "Character strings", in ISO/IEC 9075-2.*

### 4.2.1 Unicode support

Java relies on the Unicode character set [Unicode] (also known as ISO/IEC 10646, *Universal Multi-Octet Coded Character Set (UCS)*; see [UCS]) for String data and for identifiers. That allows Java to represent most character data in a uniform way. [ISO9075-2] defines support for Unicode through its UTF8, UTF16, and UTF32 character sets, which represent different encodings for Unicode character data. When character data is moved between an SQL-server and an SQL/OLB host program, an SQL/OLB implementation that provides SQL character set support for UTF8, UTF16, and/or UTF32 is required to support implicit conversion between Java string data and the supported Unicode encodings. Any support for implicit conversions to and from character sets other than Unicode is implementation-defined. Because of Java's reliance on Unicode as an internal representation for character data, the SQL/OLB specification does not define support for host variables that hold character data based on character sets other than Unicode.

The rules in [ISO9075-2] for appearance of characters in SQL <token>s of SQL-statements also govern the appearance of characters in SQLJ clauses, with the exception of characters appearing in Java identifiers and Java host expressions. All characters appearing in an SQLJ clause shall be defined in the Unicode character set.

### 4.2.2 Character sets

*This Subclause modifies Subclause 4.2.7, "Character sets", in ISO/IEC 9075-2.*

— In 3rd paragraph augment 1st list item If <embedded SQL host program> immediately contains an <embedded SQL Java program>, then <SQL special character> shall include the <number sign> (#) in addition to the characters that it otherwise required to contain.

## 4.3 Introduction to SQLJ

### 4.3.1 Overview

[ISO9075-2] specifies *embedded SQL*, a method of embedding SQL into a host programming language. It defines details of such an embedding for the programming languages Ada, C, COBOL, Fortran, Java, M, Pascal, and PL/I. This Part of this International Standard specifies a different approach for the embedding of SQL into the Java programming language. The embedding of SQL into Java is commonly known as "SQLJ".

SQLJ provides a facility to embed certain SQL language constructs in Java source language. An SQLJ *translator* is a utility that transforms Java source langauge that contains SQLJ extensions into Java source language that accesses an SQL-implementation through a call interface. SQLJ also provides a facility to tailor the SQLJ extensions to run against a target SQL-implementation, overriding the default, JDBC-based, runtime. An SQLJ *customizer* is a utility that extends a SQLJ translation-generated profile file such that during runtime the SQL-statements described by the profile are executed in an implementation-specific manner.

The standards for most programming languages, including those for which embedded SQL is supported (*e.g.*, C, COBOL, and Fortran), have had as their goal the portability of the source program. The specification for the Java programming language has an associated specification for a Java Virtual Machine. Together, they have as their goal the portability of both the source program and the binary produced by compiling the Java source. SQLJ takes the goal of portability of the source file containing the embedded SQL much further. The goal for SQLJ is that the source file containing the embedded SQL, the Java source file produced by an SQLJ translator, and the binary produced by compiling the Java source, should all be portable. Furthermore, that portability should be not only amongst different Java Virtual Machines, but also amongst different SQL implementations. This Subclause elaborates upon how that degree of portability can be achieved.

### 4.3.2 SQL constructs

The following kinds of SQL constructs are permitted to appear in SQLJ programs:

— Queries: `SELECT` statements and expressions.

— SQL-data change statements (DML): `INSERT`, `UPDATE`, `DELETE`

— Data Statements: `FETCH`, `SELECT...INTO`

— Transaction control: `COMMIT`, `ROLLBACK`, *etc*.

— Data Definition Language (DDL; called "Schema Manipulation Language" in SQL): `CREATE`, `DROP`, *etc*.



 

— Calls to SQL-invoked procedures: *e.g.*, `CALL MYPROC(:x, :y, :z)`

— Invocations of SQL-invoked functions: *e.g.*, `VALUES( MYFUN(:x) )`

— Assignment statement: `SET`

### 4.3.3 SQLJ clauses

SQL-statements in SQLJ appear in *SQLJ clauses*. SQLJ clauses represent the mechanism by which SQL-statements in Java programs are communicated to the SQL-implementation.

Each SQLJ clause begins with the token **`#sql`**, which is not a legal Java identifier, and is terminated by a semicolon, and as such makes the clause and its SQL contents recognizable to an SQLJ translator.

The simplest SQLJ clauses are *executable clauses* and consist of the token **`#sql`** followed by an SQL-statement enclosed in braces ({ and }).

In an SQLJ executable clause, the tokens that appear inside of the braces are SQL <token>s and <separator>s, except for the tokens of the Java programming language appearing as host variables and parenthesized host expressions. All host variables and parenthesized host expressions shall be distinguished by the colon character in order for the translator to be able to identify them. SQL <token>s and <separator>s never occur outside of the single pair of braces of an SQLJ executable clause.

In general, SQL <token>s are case-insensitive (except for identifiers delimited by double quotes), and can be written in upper, lower, or mixed case. Java tokens, however, are case-sensitive. For clarity in examples, we write case-insensitive SQL <token>s in uppercase, and write Java tokens in lowercase or mixed case. Throughout this document, we use the lowercase **`null`** to represent the Java “null” value, and the uppercase **`NULL`** to represent the SQL null value.

Host expressions are also permitted to be used as assignment targets if the host expression evaluates to a Java l-valued expression.

### 4.3.4 Binary portability

#### 4.3.4.1 Binary portability requirements

This Subclause specifies the requirements for binary portable SQLJ applications, and overviews how binary portability is achieved. A binary portable SQLJ application has the following properties:

— Component-level interoperability

  Connection context objects defined by one application can be used in SQLJ executable clauses of another application. Similarly, result set iterator objects defined by one application can be instantiated and populated by SQLJ executable clauses of another application.

— Packaged description of SQL-statements

  Every SQLJ application shall include a set of well defined resource files that can be read to determine information about SQL-statements performed by the application.

— Configurable SQL execution control

The runtime execution of SQL-statements is controlled by implementation-specific components. Components can be added and removed at any time from a binary SQLJ application, as appropriate for the eventual runtime environment.

Binary portability is achieved though a framework that is based upon the following four aspects:

— Runtime library

This document specifies a runtime library composed of classes common to all applications produced by a particular SQLJ translator. This includes the classes in the standard SQLJ runtime packages:

- package **`sqlj.runtime`**

  Standard classes that a client will use directly in SQLJ source code.

- package **`sqlj.runtime.profile`**

  Standard classes used by the binary portability framework, but not used directly by the client.

The runtime library also includes concrete implementations of the interfaces and abstract classes in the above packages. These classes might vary depending on the SQLJ translator. The standard SQLJ runtime packages are described in Clause 12, "Package sqlj.runtime", Clause 13, "Package sqlj.runtime.profile", and Clause 14, "sqlj.runtime.profile.util.ProfileCustomizer", of this document.

— Profile files

Every SQLJ application will include a set of SQLJ translation-generated resources called *profiles* that describe the SQL-statements appearing in the application; a profile file contains profiles for a single SQLJ application. This document defines the requirements for the name, number and contents of profiles created for each SQLJ application.

— Generated class signature

The SQLJ-translation of declarative <SQLJ specific clause>s shall produce a uniform class signature such that not only clients, but also other applications are able to use these classes interchangeably. This document defines requirements for the signatures of classes created for SQLJ applications.

— Calls to runtime and profile

The SQLJ-translation of executable <SQLJ specific clause>s shall result in calls to routines in the SQLJ runtime libraries and profiles. Implementation-dependent SQL execution is accomplished by implementing runtime library components. The calls made to these components by each <SQLJ specific clause> are defined in order to allow a standardized implementation.

Given the portability framework, implementations are able to install custom SQL execution components into an SQLJ application. Note that implementations of the above list is not required to achieve custom SQL execution. Rather, the above list specifies the preconditions that shall be true to enable reliable addition of custom SQL execution components to an existing SQLJ application.



 

#### 4.3.4.2 Components of binary portable applications

An SQLJ application might consist of a number of different SQLJ translation units, translated using one or more SQLJ translators, and customized using one or more customizers. Each individual SQLJ translation unit is binary portable with every other SQLJ translation unit within the application and the application as a whole is binary portable with other applications so long as the required components of each SQLJ translation unit are available at runtime, and each component conforms to the requirements specified in this document.

In addition to the classes and profiles generated by a translator, there might also be a translator-specific runtime library component required to run the generated classes. In many cases, a translator will provide runtime library components that implement the SQLJ standard classes and interfaces defined in this document. Similarly, a customizer might require an implementation-specific runtime library component to run a customized version of the application. If more than one translator or customizer is used to build the application, then there might be a different runtime library component for each translator and customizer used.

In summary, a binary portable application consists of the following components:

1) SQLJ standard runtime classes.
2) For each translator **T** used to translate units in the application:
   a) Runtime classes used by translator-generated code (if any).
   b) For each unit translated by **T**:
      i) User-defined classes (which might contain executable SQLJ clauses).
      ii) User-defined property files containing maps for user-defined data types (if any).
      iii) Classes generated by **T** as a result of declarative SQLJ clauses.
      iv) Profiles generated by **T**.
      v) Auxiliary helper classes generated by the **T** (if any).
3) For each customizer used to customize profiles in the application, runtime classes used by the installed customization (if any).

The runtime library components of a particular translator or customizer need not be packaged with the application if the component already exists in the runtime environment. For example, an SQL-implementation might be configured to support the runtime classes of a particular translator and customization, or a set of runtime classes might be packaged as a downloadable plug-in for use in browsers.

### 4.3.5 Profile overview

SQLJ applications are *binary portable*, meaning that the same binary application can be run against any SQL-server implementation without modification of the original source code and without retranslation of the original source code using an implementation-specific SQLJ translator. Binary portability is achieved in part by implementing the default SQLJ runtime libraries on top of JDBC. Any implementation that supports an infrastructure that emulates the JDBC call interfaces and semantics will automatically be able to support SQLJ.

In addition, many implementations have systems that allow SQL-statements to be optimized and executed with greater performance than equivalent JDBC dynamic SQL. For these systems, it is desirable to be able to implement alternative SQLJ runtime interfaces instead of the runtime JDBC interfaces.

To accommodate this need, two requirements shall be satisfied. First, the SQLJ application shall include a complete, accessible description of the SQL-statements that it will perform. These descriptions are then used at application deployment time by implementation-specific tools that precompile and install the SQL-statements as appropriate for a particular DBMS. Second, the implementation shall be able to install a hook into an existing SQLJ application such that the SQL-statements are executed using an implementation-specific runtime rather than the default JDBC-based implementation. Each SQLJ application includes a set of SQLJ profiles that satisfy these two requirements.

An SQLJ profile is an instance of the Java class **sqlj.runtime.profile.Profile**. It describes every SQL-statement appearing within the original SQLJ source file. For each SQL-statement, the profile contains an entry that describes among other things; the operation text, the number, type and parameter mode of each parameter, and a description of the columns that are expected to be produced by the operation, if any. SQLJ profiles are packaged with the application either as serialized objects or as distinct subclasses of **sqlj.runtime.profile.Profile**. Thus, every SQLJ application includes a set of profiles that can be loaded at any time, and used to programmatically inspect the SQL-statements it might perform.

SQLJ profiles are *serializable*, meaning that their state might be stored to a file (or table column) and then restored at a later time. In addition to describing SQL-statements in the source file, profiles also contain a set of implementation-dependent *customization objects*. A customization object is an implementation-dependent object that is used by the SQLJ runtime to execute an SQL-statement described in the profile (the term “customization” refers to the fact that they are used to achieve implementation-dependent “customized” behavior in the program). Customizations can be registered and deregistered with a profile. At runtime, a profile selects the appropriate customization to use according to the SQL-connection established.



#### 4.3.5.1 EntryInfo overview

For each <executable clause> (except those describing a <fetch statement>), an EntryInfo object is created and stored in an SQLJ profile. An EntryInfo object contains a collection of Java fields that describe an <executable clause> as defined below:

— **SQL String**

A Java String containing the portion of the <executable clause> appearing between braces. Unless otherwise noted, the string contains the exact text of the original program source, including line breaks, other white space, and comments. Case is preserved. Any <embedded Java expression> appearing in the original <executable clause> is replaced with <dynamic parameter specification>.

— **Role**

Categorizes the <executable clause>. Unless otherwise stated, the role is STATEMENT. The role is used to distinguish operations that are likely to be handled in a special way by the runtime implementation, and are not meant to be an exhaustive list of all possible types of SQL-statements.

For example, the role SINGLE_ROW_QUERY indicates that the operation is a query that is expected to return only a single row. It is distinguished from the more general QUERY role since a runtime might be able to optimize queries that return only a single row. UPDATE, INSERT, and DELETE operations all fall into the general role of STATEMENT, since they are likely to be handled the same way by the underlying engine.

 

Table 1, “Association of roles with SQLJ <executable clause>s”, associates roles with corresponding SQLJ clauses.

**Table 1 — Association of roles with SQLJ <executable clause>s**

| Role | <executable clause> |
|---|---|
| QUERY | <query clause> |
| CALL | <call statement> |
| VALUES | <function clause> |
| POSITIONED | <delete statement: positioned> |
| POSITIONED | <update statement: positioned> |
| QUERY_FOR_UPDATE | <query clause> populating ForUpdate iterator |
| SINGLE_ROW_QUERY | <select statement: single row> |
| UNTYPED_SELECT | <query clause> populating sqlj.runtime.ResultSetIterator |
| COMMIT | <commit statement> |
| SAVEPOINT | <savepoint statement> |
| RELEASE_SAVEPOINT | <release savepoint statement> |
| ROLLBACK | <rollback statement> |
| SET_TRANSACTION | <set transaction statement> |
| ITERATOR_CONVERSION | <iterator conversion clause> |
| BLOCK | <compound statement> |
| STATEMENT | *all other operations* |
| OTHER | reserved for implementation-defined extensions |



— **Statement Type**

Statement type is CALLABLE_STATEMENT if the <executable clause> contains at least one <embedded Java expression> the <parameter mode> of which is OUT or INOUT. Otherwise, statement type is PREPARED_STATEMENT if the <executable clause> contains no <embedded Java expression> or all <embedded Java expression>s have <parameter mode> IN.

NOTE 10 — An entry with a role of CALL is permitted to have statement type PREPARED_STATEMENT if no <embedded Java expression> exists with <parameter mode> OUT.

— **Execute Type**

Describes the RTStatement execute method that is used at runtime to perform this operation, one of EXECUTE_QUERY, EXECUTE_UPDATE, or EXECUTE. The execute type is EXECUTE_UPDATE unless otherwise noted.

— **Parameter Attributes**

Describes the <embedded Java expression>s appearing in the <executable clause>. Parameter information is composed of a number of Java fields.

- **Param Count**

  Gives the number of <embedded Java expression>s appearing in the <executable clause>, **k**.

- **Param Info**

  A collection of TypeInfo objects (defined below) that describe the set of <embedded Java expression>s that appear in an <executable clause>. For each <embedded Java expression>, $\mathbf{HE_i}$, the *i*-th TypeInfo object, describes $\mathbf{HE_i}$ of the original <executable clause> (or, equivalently, the *i*-th <dynamic parameter specification> in the SQL String Java field), where *i* is in the range 1 (one) $\mathbf{\leq i \leq k}$. The TypeInfo object returned has the mode of the <parameter mode> of $\mathbf{HE_i}$. It has Java type name corresponding to the name of the type of $\mathbf{HE_i}$. If $\mathbf{HE_i}$ is a <simple variable>, then the TypeInfo object returned has the same name as that of the <simple variable>. Otherwise, if $\mathbf{HE_i}$ is a <complex expression>, then the TypeInfo object returned has name = null.

— **Result Set Column Java fields**

Describes the columns expected to be produced by the <executable clause>. Result set column information is composed of a number of Java fields.

- **Result Set Type**

  Describes the way in which the result set columns are expected to be bound. One of NO_RESULT, NAMED_RESULT, POSITIONED_RESULT. Result set type is NO_RESULT if the <executable clause> is an operation that does not produce a result set.

- **Result Set Count**

  The number of result set columns that the <executable clause> is expected to produce as indicated by the cardinality of the <iterator spec declaration>.

  If the result set type is NO_RESULT, then result set count is 0 (zero).

- **Result Set Info**

  A collection of TypeInfo objects that describe the <iterator spec declaration>. For result set usage of a TypeInfo, mode and dynamic parameter marker index have no meaning and are defaulted to return OUT and –1.

  This is empty unless otherwise stated.

- **Result Set Name**

  The class name of the iterator object populated by this <executable clause>. If the result set type is NO_RESULT, the result set name is null.

— **Descriptor**



 

Contains any extra information not otherwise provided by the other Java fields. The descriptor is null unless otherwise stated.

#### 4.3.5.2 TypeInfo overview

Each <embedded Java expression> and expected result set column of an <executable clause> is described by a TypeInfo object. For user-defined data types, the content of a TypeInfo object is in part determined using the associated connection context type map. This is the type map specified in the <connection declaration clause> of the connection context object with which the <executable clause> is associated. A TypeInfo object contains a collection of Java fields describing the <embedded Java expression> or result set column, as described below.

— **Java Type Name**

The name of the Java Class or primitive type that is the type of the <embedded Java expression> or result set column.

In most cases, the name returned is the same as the internal name of the type, as defined by [JLS]; primitive types have their simple names (*e.g.*, int), classes are fully qualified (*e.g.*, **java.sql.Date**), and nested classes are delimited with "$" (*e.g.*, **x.y.OuterClass$InnerClass**). Array naming uses a more readable convention than the internal type name: if the name returned represents an array, the string "[" is prepended to the full name of the component type. For example, an array of array of String has the name "**[[java.lang.String**".

— **SQL Type**

A **java.sql.Types**-defined constant that corresponds for predefined data types to the default mapping of the Java type of the <embedded Java expression> or result set column into an SQL type, as defined by [JDBC]. For user-defined data types that are covered by a property definition in the associated connection context type map, this field contains the SQL type (*i.e.*, **STRUCT**, **DISTINCT**, or **JAVA_OBJECT**) corresponding to the Java type name as defined in that property definition. If the property definition for the Java type does not specify an SQL type, then the following default mechanism is used for determining the SQL type: If the Java type of the <embedded Java expression> or result set column implements the interface **java.sql.SQLData**, then the SQL Type field is set to **STRUCT**; otherwise, it is set to **JAVA_OBJECT**. If no property entry is found in the connection type map for the given Java type name, or no type map has been associated with the connection context class, then the SQL Type is **OTHER**. This mapping is also given in columns one and two of Table 2, "SQLJ type properties".

NOTE 11 — SQL Type is not a representation of the SQL types. Instead, it exists as an established default mapping between Java types and [JDBC]-defined SQL type constants. It might be disregarded or remapped as appropriate by implementation-defined profile customizations.

— **SQL Type Name**

If the SQL Type field of the TypeInfo object is either **STRUCT**, **DISTINCT**, or **JAVA_OBJECT**, then this field contains a String giving the user-defined name of the SQL type corresponding to the Java type of the <embedded Java expression> or result set column, as defined by the associated connection context type map.

— **Mode**

One of IN, INOUT, or OUT. Gives the <parameter mode> of a host expression. For result set columns, the mode is always OUT.

— **Name**

Gives the name of the <embedded Java expression> or result set column, if available. If the TypeInfo object represents a <embedded Java expression> the rules for determining the name are specified in EntryInfo-Param Info. If the TypeInfo object represents a column of <named iterator> then name is determined by its <iterator spec declaration>. If the TypeInfo object represents a column of a <positioned iterator>, then name=null.

— **Dynamic Parameter Marker Index**

Gives the zero-based index of the <dynamic parameter specification> appearing in the SQL String that corresponds to the <embedded Java expression> represented by this TypeInfo object. The dynamic parameter marker index is –1 if this TypeInfo object describes a result set column.

### 4.3.5.3 SQLJ datatype properties

Every bind variable, return result, and column type is described in an SQLJ profile by means of a TypeInfo object.

Table 2, "SQLJ type properties", describes the datatypes supported by SQLJ.

— The first column lists the names of the Java datatypes that are supported by SQLJ.

— The second column lists the java.sql.Types constant value of the given datatype.

— The third column lists the "getter method" used to fetch bind variables of the given datatype either as an out-parameter (RTStatement) or column type (RTResultSet).

— The fourth column lists the "setter method" used to pass bind variables of the given datatype as input parameter (RTStatement) to the SQL-implementation.

**Table 2 — SQLJ type properties**

| Java type name | java.sql.Types value | getter method | setter method |
|---|---|---|---|
| boolean | BOOLEAN | getBooleanNoNull | setBoolean |
| byte | TINYINT | getByteNoNull | setByte |
| short | SMALLINT | getShortNoNull | setShort |
| int | INTEGER | getIntNoNull | setInt |
| long | BIGINT | getLongNoNull | setLong |
| float | REAL | getFloatNoNull | serFloat |
| double | DOUBLE | getDoubleNoNull | setDouble |
| java.lang.Boolean | BOOLEAN | getBooleanWrapper | setBooleanWrapper |
| java.lang.Byte | TINYINT | getByteWrapper | setByteWrapper |

| Java type name | java.sql.Types value | getter method | setter method |
|---|---|---|---|
| java.lang.Short | SMALLINT | getShortWrapper | setShortWrapper |
| java.lang.Integer | INTEGER | getIntWrapper | setIntWrapper |
| java.lang.Long | BIGINT | getLongWrapper | setLongWrapper |
| java.lang.Float | REAL | getFloatWrapper | setFloatWrapper |
| java.lang.Double | DOUBLE | getDoubleWrapper | setDoubleWrapper |
| java.lang.String | VARCHAR | getString | setString |
| java.math.BigDecimal | NUMERIC | getBigDecimal | setBigDecimal |
| byte[][1] | VARBINARY | getBytes | setBytes |
| java.sql.Array[6] | ARRAY | getArray | setArray |
| java.sql.Blob[4] | BLOB | getBlob | setBlob |
| java.sql.Clob[4] | CLOB | getClob | setClob |
| java.sql.Date | DATE | getDate | setDate |
| java.sql.Time | TIME | getTime | setTime |
| java.sql.Timestamp | TIMESTAMP | getTimestamp | setTimestamp |
| java.net.URL | DATALINK | getURL | setURL |
| sqlj.runtime.AsciiStream[2] | OTHER | getAsciiStream[3] | setAsciiStream |
| sqlj.runtime.BinaryStream[2] | OTHER | getBinaryStream[3] | setBinaryStream |
| sqlj.runtime.Character-Stream[2] | OTHER | getCharacterStream-Wrapper | setCharacterStreamWrap-per |
| sqlj.runtime.UnicodeStream[2] | OTHER | getUnicodeStream[3] | setUnicodeStream |
| java.sql.Ref | REF | getRef | setRef |
| java.sql.SQLXML | SQLXML | getSQLXML | setSQLXML |
| *any other class*[5] | STRUCT, DIS-TINCT, JAVA_OBJECT, OTHER | getObject | setObject |

| Java type name | java.sql.Types value | getter method | setter method |
|---|---|---|---|

[1] The Java type name Java field stored in a TypeInfo object for a byte array is named "[byte", not "byte[]".

[2] The **sqlj.runtime.XXXStream** classes are subclasses of **java.io.InputStream**. Explicit type names allow the type of the stream data to be encoded statically in the type name, and therefore to be determined at translate time rather than runtime. They also add a "length" Java field that is needed when streams are passed to an SQL-server.

[3] Getter methods for stream types are only available on the **RTResultSet** interface, but not the **RTStatement** interface. This is for symmetry with JDBC.

[4] The Blob and Clob data type implementations are recommended to be based on large object locators. In order to retrieve or provide the actual Lob values instead of location, one can use host variables of type BinaryStream, AsciiStream, or UnicodeStream instead of Blob and Clob.

[5] There is no single default java.sql.Types value for the Java type name in this case. The "java.sql.Types value" column lists the alternative values permitted in this case. The actual value stored in a TypeInfo object that is part of the profile entry is determined by the SQLJ translator according to the Rules given in Subclause 4.3.5.2, "TypeInfo overview".

[6] The Array data type implementation is recommended to be based on array locators.

### 4.3.6 Host variables

Arguments to embedded SQL-statements are passed through *host variables*, which are variables of the host language that appear in the SQL-statement. Host variable names are prefixed by a colon (`:`). A host variable contains an optional parameter mode identifier ( IN,OUT, INOUT ) followed by a Java host variable that is a Java identifier, naming a parameter, variable, or Java field. The evaluation of a Java identifier does not have side effects in a Java program, so it is permitted to appear multiple times in the Java code generated to replace an SQLJ clause.

### 4.3.7 Host expressions

SQLJ extends the traditional embedded support by allowing Java host expressions to appear directly in SQL-statements. Host expressions are prefixed by a colon (`:`) followed by an optional parameter mode identifier (IN, OUT, INOUT) followed by a parenthesized expression clause. An expression clause contains a Java expression that either evaluates to a single value (in the case of IN mode) or is a Java l-valued expression (in the case of an OUT or INOUT mode).

The evaluation of host expressions ***does have side effects in a Java program*** as they are evaluated by Java rather than the SQL-server. Host expressions are evaluated left to right within the SQL-statement prior to submission to the SQL-server.

Host expressions are always passed to and retrieved from the SQL-server using pure value semantics.

Assignments to output host expressions are also performed in lexical order.



 

### 4.3.8 Connection contexts

Each SQLJ executable clause requires, either explicitly or implicitly, a *connection context object* that designates the SQL-connection with which the SQL-statement specified in that clause will be executed.

The connection context object designates an SQL-implementation at which the SQL-statements will be executed, and the session and transaction in which they are executed. A connection context is an object of a *connection context class*, which is defined by means of an SQLJ *connection clause*. Given a `java.sql.Connection` object, a URL (see [RFC2368], and [RFC3986] for more details about URLs), or a URL and information such as a user name and password (it is also possible to include user name and password in a URL), a connection context class has methods for identifying and, if necessary, opening an SQL-connection. At run time, an SQLJ program uses one of those methods to establish an SQL-connection before any SQLJ clauses are executed.

NOTE 12 — The connection context object implicitly specifies an SQL-implementation, and the default catalog and schema, as per a `java.sql.Connection` object.

### 4.3.9 Default connection context

If an SQLJ clause contains an expression designating the connection context object on which it will be executed, then that clause is said to use an *explicit connection*. If the connection context object is omitted from a clause, then that clause is said to use the *default connection*.

The specification of the default connection context is implementation-defined. Portable applications should always use explicit connection contexts.

If an invocation of an SQLJ translator indicates that the default connection context class is some class **connctx**, then all SQLJ clauses that use the default connection context are translated as if they had explicitly used the connection context object **`connctx`**`.getDefaultContext()`.

Programs are permitted to install a connection context object as the default connection by calling `setDefaultContext`.

The default connection context object for a program is stored in a static variable of the default connection context class. Some SQLJ programs will wish to avoid using static variables. For example, Applets, reentrant libraries, and some multithreaded programs will avoid static variables. Those programs will wish to use SQLJ clauses with explicit connection contexts objects.

If an SQLJ program is executing as an external routine (or is otherwise executing in an environment that automatically provides a connection context), calls to method `ConnectionContext.getDefaultContext` always return an object representing the connection in which the program is executing. An SQLJ program can detect whether it is executing in an environment that implicitly supplies a connection context by calling `ConnectionContext.getDefaultContext` before it calls `ConnectionContext.setDefaultContext` to install a connection context object. An execution environment that automatically supplies a connection context will return a non null connection context object.

The SQL-connection used by the default connection context is defined by the data source bound to the name *`jdbc/defaultDataSource`* using JNDI. If this name is not defined, the SQL-connection used is implementation-defined.

NOTE 13 — "JNDI" is the Java Naming Directory Interface, defined in [JNDI].

### 4.3.10 Schema checking using exemplar schemas

At SQLJ translation time, a connection context class plays a different role. It symbolizes the "type" of SQL-schema to which the SQLJ program will connect at run time. The notion of the "type of an SQL-schema" is informal. It includes the names and privileges associated with tables and views, the "shapes" of their rows, SQL-invoked routines, and so forth. The type of a schema is symbolized by an *exemplar schema*, which is simply a schema that contains the tables, views, SQL-invoked routines, and privileges that would be required in order for the SQL-statements in SQLJ clauses to execute successfully. An exemplar schema might be the actual runtime schema, or it might be another schema that is a "typical" schema, in ways relevant to the SQLJ program being translated.

If an exemplar schema is being used, then the invoker of an SQLJ translator provides a mapping of connection context classes to exemplar schemas. An SQLJ translator connects to the exemplar schema in order to provide syntax checking, type checking and schema checking for all SQLJ clauses that will be executed in the connection context of the class "exemplified" by that schema. In that way, the exemplar schema represents the schema to which the application will connect at runtime. It is the responsibility of the application developer to pick an exemplar schema that represents the run time schemas in relevant ways, *e.g.* having tables, views, and SQL-invoked routines with the same names and types, and having privileges set appropriately.

If no connection to an appropriate exemplar schema or SQL-implementation for a connection type is established during SQLJ translation, then SQLJ clauses to be executed on connections of that type will not be schema checked at SQLJ translation time, and will instead be checked later at installation or customization time.

The mapping of connection context classes to exemplar schemas is provided to an SQLJ translator in an implementation-defined way, typically by pairing connection context class names with connect strings and passwords. For example, a client side SQLJ translator is permitted to require that mapping on the command line in an invocation of the translator. Those connect strings and passwords are then used as arguments to invocations of ConnectionContext class constructors that establish an SQL-connection to the exemplar schema.

Since the connection context is optional in an SQLJ clause, if the connection context is absent from an SQLJ clause, there shall be a default connection context class specified. The clause is then checked against the exemplar schema corresponding to the class of the default connection context object for the program.

### 4.3.11 Using multiple SQLJ contexts and connections

SQLJ supports concurrent connections to multiple SQL-servers. SQLJ models ach SQL-server that is connected at runtime as a distinct connection context class. Multiple schemas co-located within a single SQL-server are all accessible by a single connection context. Schemas located on different SQL-servers require separate connection contexts, one per SQL-connection. The specification of the appropriate connection context associated with an #sql statement allows type checking across multiple SQL-servers at translate time.

### 4.3.12 Dynamic SQL and JDBC/SQLJ Connection interoperability

The SQLJ language provides direct support for SQL-statements that are known at the time the program is written. If some or all of a particular SQL-statement cannot be determined until runtime, it is a dynamic operation. To perform dynamic SQL from an SQLJ program, the application may use JDBC. A `ConnectionContext` object contains a `java.sql.Connection` object that can be used to create `java.sql.Statement` objects needed for dynamic SQL.



 

#### 4.3.12.1 Creating an SQLJ ConnectionContext from a java.sql.Connection object

Every SQLJ ConnectionContext class includes a constructor that takes as an argument a `java.sql.Connection` object. This constructor is used to create an SQLJ connection context object that shares its underlying SQL-connection with that of the `java.sql.Connection` object.

#### 4.3.12.2 Obtaining a java.sql.Connection object from an SQLJ ConnectionContext

Every SQLJ ConnectionContext object has a **getConnection** method that returns a `java.sql.Connection` object. The `java.sql.Connection` object returned shares the underlying SQL-connection with the SQLJ connection context. It can be used to perform dynamic SQL as described in [JDBC].

#### 4.3.12.3 Connection sharing

An SQLJ ConnectionContext always contains a `java.sql.Connection` object and relies upon it to provide communication with the underlying SQL-connection. Accordingly, calls to methods that affect connection state on one object will also be reflected in the other object, as it is actually the underlying shared SQL-connection that is being affected.

JDBC defines the default values for session state of newly created connections. In most cases, SQLJ adopts these default values. However, whereas a newly created `java.sql.Connection` object has auto commit mode on by default, an SQLJ connection context requires the auto commit mode to be specified explicitly upon construction.

#### 4.3.12.4 Connection resource management

The **close** method of a connection context object causes the associated `java.sql.Connection` object and the underlying SQL-connection to be closed. However, because connection contexts are permitted to share the underlying SQL-connection with other connection contexts and/or `java.sql.Connection` objects, an option is available to release resources maintained by the connection context object, but not close the associated `java.sql.Connection` object and the underlying SQL-connection.

If a connection context object is not explicitly closed before it is garbage-collected, then **close(KEEP_CONNECTION)** is called by the finalize method of the connection context. This allows connection related resources to be reclaimed by the normal garbage collection process while maintaining the underlying SQL-connection for other JDBC and SQLJ objects that might be using it. Note that if no other JDBC or SQLJ objects are using the SQL-connection, then the SQL-connection will also be closed and reclaimed by the garbage collection process.

Both SQLJ connection context objects and `java.sql.Connection` objects respond to the **close** method. When writing an SQLJ program, it is sufficient to call the **close** method on only the connection context object. This is because closing the connection context will also close the `java.sql.Connection` object associated with it. However, it is not sufficient to close only the `java.sql.Connection` object returned by the **getConnection** method of a connection context. This is because the **close** method of a `java.sql.Connection` object will not cause the containing connection context to be closed, and therefore, resources maintained by the connection context will not be released until it is garbage-collected.

The **isClosed** method of a connection context returns **true** if any variant of the **close** method has been called on the connection context object. If **isClosed** is **true**, then calling **close** is a no-op, and the effect of calling any other method is implementation-dependent.

### 4.3.13 SQL execution control and status

The execution semantics of SQL-statements can be queried and modified via the execution context associated with the operation. An execution context exists as an instance of class `sqlj.runtime.ExecutionContext`.

The following `ExecutionContext` Java fields control the execution environment of SQL-statements. The **`getXXX`** and **`setXXX`** methods read and change the `XXX` value. Once set, they affect all SQL-statements subsequently executed on that execution context.

— **`MaxRows`** specifies the maximum number of rows to be returned by any query.

— **`MaxFieldSize`** specifies the maximum number of bytes to be returned as data for any column or output variable.

— **`QueryTimeout`** specifies the number of seconds to wait for an SQL-statement to complete.

NOTE 14 — Runtime support of the above ExecutionContext Java fields, if set to anything other than their respective default values, is not part of Core SQLJ. See Subclause 11.9, "<executable clause>".

The following `ExecutionContext` Java fields describe the results of the last SQL-statement executed.

— **`UpdateCount`** specifies the number of rows updated, inserted, or deleted during the last operation.

— **`SQLWarnings`** describes any warnings that occurred during the last operation.

An execution context is associated either explicitly or implicitly with each SQLJ executable clause appearing in an SQLJ program.

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

If an execution context is not supplied explicitly, a default execution context is used implicitly. The default execution context for a particular SQLJ executable clause is obtained via the `getExecutionContext()` method of the connection context used in the SQLJ executable clause.

If neither a connection context nor an execution context is explicitly supplied, the execution context associated with the default connection context is used.

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

### 4.3.14 Iterators

A capability central to SQL is the ability to execute queries that retrieve a "result set" of rows from an SQL-implementation. An SQLJ clause might evaluate a query and return a *result set iterator* object containing the result set selected by that query. Depending on the type of the iterator object, it might be used with the **`FETCH...INTO`** idiom of SQL to extract data into host variables, or it might return column data through named



 

accessor methods having the names and types of columns returned by the query. The iterator declaration clause is permitted to appear wherever a Java class definition may appear.

An SQLJ *iterator* is a Java object that implements the interface **sqlj.runtime.ResultSetIterator** and from which the data returned by an SQL query can be retrieved. In that role, it corresponds to the *cursor* of SQL, from which data are fetched. Unlike the cursor, however, an iterator object is a first class object. An iterator object can be passed as a parameter to a method, and can be used outside the SQLJ translation unit that creates it, without losing its static type for the purposes of type checking of component interfaces.

An iterator object has one or more columns with associated Java types. Names that are Java identifiers can optionally be provided for the iterator object columns. If the expressions selected by a query are unnamed, or have SQL names that are not valid Java identifiers, then SQL column aliases can be used to name them. The columns of an iterator object (which have Java types) are conceptually distinct from the columns of a query (which have SQL types), and therefore, a means of matching one to the other shall be chosen. SQLJ supports two mechanisms for matching iterator object columns to query columns. They are *bind by position* and *bind by name*.

*Bind by position* means that the left to right order of declaration of the iterator object columns places them in correspondence with the expressions selected in an SQL query. Traditional **`FETCH...INTO`** syntax is used to retrieve data from the iterator object into Java variables. An iterator class that binds by position is declared by providing a parenthesized, comma-separated list of data types, one per column of the rows returned by the iterator object. The list specifies only the data types of the columns and does not specify a name for the columns. The data types in the list shall appear in exactly the same sequence as the data types of the columns of the rows returned by the iterator object. The types of the SQL columns in the query shall be convertible to the types of the positionally corresponding iterator object columns, according to the SQL to Java type mappings of SQLJ. Those conversions are statically checked at SQLJ translation time if an SQL-connection to an exemplar schema is provided to the translator.

*Bind by name* means that the name of each iterator object column is matched to the name of a column returned by the SQL query, independent of the order in which that column appeared in the query. Named accessor methods are generated by the SQLJ translator for each column of the iterator object. The name of a named accessor method matches the name of a column returned by a query and its return type is the Java type of the iterator object column. The **`FETCH...INTO`** syntax is not permitted to be used with an iterator object of this type, as the named accessor methods provide the mechanism for transferring the data. An iterator class that binds by name is declared by providing a parenthesized, comma-separated list of data types and identifiers, one per column of the rows returned by the iterator object. The list specifies the data types and the name of each column of the rows returned by the iterator object. The sequence of data types and identifiers in the list need not be the same sequence as the columns of the rows returned by the iterator object. A Java compiler will detect type mismatch errors in the uses of named accessor methods. Additionally, if a connection to an exemplar schema is provided at translate time, then the SQLJ translator will statically check the validity of the types and names of the iterator object columns against the SQL queries associated with it.

An iterator declaration clause designates whether objects of that iterator type use bind by position or bind by name. The two styles of access to result set data are mutually exclusive; an iterator class supports either bind by position or bind by name, but not both. Program development tools might prefer to generate SQLJ programs using bind by position, since these tools can generate SQLJ code that is “correct by construction”. People writing SQLJ programs “by hand” might prefer to use bind by name, to make their applications resilient against changes to the program or SQL-schema.

### 4.3.15 Input and output assignability

An SQL type *ST* is *SQLJ output assignable* to a Java class or primitive type *JT* if Table 3, "SQLJ output assignability (part 1)", Table 4, "SQLJ output assignability (part 2)", Table 5, "SQLJ output assignability (part 3)", or Table 6, "SQLJ output assignability (part 4)", contains an 'x' in the cell identified by the column for the `java.sql.Types` value of *ST* and the row in which *JT* is specified in the first column. In addition, the following conditions shall hold for structured and distinct types (*i.e.*, `java.sql.Types` values STRUCT and DISTINCT).

— If the `java.sql.Types` value of *ST* is either DISTINCT or STRUCT, and *JT* is not one of the Java classes or primitive types identified in the first column of Table 5, "SQLJ output assignability (part 3)", (*i.e.*, "any other class/interface" applies), then the user-defined type map that is associated with the connection context class of the SQLJ clause for which output assignability is checked shall specify a Java class or primitive type *JT* that corresponds to *ST*.

— If the `java.sql.Types` value of *ST* is DISTINCT, and *JT* is one of the Java classes or primitive types identified in the first column of Table 5, "SQLJ output assignability (part 3)", then there exists an SQL type *ST1*, where *ST1* is either the representation type of *ST*, or a transform group has been specified for *ST* in the connection context class of the SQLJ clause for which output assignability is checked, and *ST1* is the result type of the from-sql transform function or method of that transform group. *ST1* shall be SQLJ output assignable to *JT*.

— If the `java.sql.Types` value of *ST* is STRUCT, and *JT* is one of the Java classes or primitive types identified in the first column of Table 5, "SQLJ output assignability (part 3)", then a transform group has been specified for *ST* in the connection context class of the SQLJ clause for which output assignability is checked, and the result type of the from-sql transform function or method of that transform group is SQLJ output assignable to *JT*.

**Table 3 — SQLJ output assignability (part 1)**

| | **java.sql.Types constants**[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **TI** | **SI** | **IN** | **BI** | **RL** | **FL** | **DB** | **DC** |
| boolean | x | x | x | x | x | x | x | x |
| byte | x | x | x | x | x | x | x | x |
| short | x | x | x | x | x | x | x | x |
| int | x | x | x | x | x | x | x | x |
| long | x | x | x | x | x | x | x | x |
| float | x | x | x | x | x | x | x | x |
| double | x | x | x | x | x | x | x | x |
| java.lang.Boolean | x | x | x | x | x | x | x | x |
| java.lang.Byte | x | x | x | x | x | x | x | x |
| java.lang.Short | x | x | x | x | x | x | x | x |

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **TI** | **SI** | **IN** | **BI** | **RL** | **FL** | **DB** | **DC** |
| java.lang.Integer | x | x | x | x | x | x | x | x |
| java.lang.Long | x | x | x | x | x | x | x | x |
| java.lang.Float | x | x | x | x | x | x | x | x |
| java.lang.Double | x | x | x | x | x | x | x | x |
| java.lang.String | x | x | x | x | x | x | x | x |
| java.math.BigDecimal | x | x | x | x | x | x | x | x |
| byte[] | | | | | | | | |
| java.sql.Array | | | | | | | | |
| java.sql.Blob | | | | | | | | |
| java.sql.Clob | | | | | | | | |
| java.sql.Date | | | | | | | | |
| java.sql.Ref | | | | | | | | |
| java.sql.Time | | | | | | | | |
| java.sql.Timestamp | | | | | | | | |
| sqlj.runtime.AsciiStream | | | | | | | | |
| sqlj.runtime.BinaryStream | | | | | | | | |
| sqlj.runtime.CharacterStream | | | | | | | | |
| sqlj.runtime.UnicodeStream | | | | | | | | |
| java.net.URL | | | | | | | | |
| java.sql.SQLXML | | | | | | | | |
| *any other class/interface* | | | | | | | | |

[1] where: TI corresponds to TINYINT, SI to SMALLINT, IN to INTEGER, BI to BIGINT, RL to REAL, FL to FLOAT, DB to DOUBLE, and DC to DECIMAL

**Table 4 — SQLJ output assignability (part 2)**

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **NU** | **BO** | **CH** | **VC** | **LC** | **CL** | **BI** | **VB** |
| boolean | x | x | x | x | x | | | |
| byte | x | x | x | x | x | | | |
| short | x | x | x | x | x | | | |
| int | x | x | x | x | x | | | |
| long | x | x | x | x | x | | | |
| float | x | x | x | x | x | | | |
| double | x | x | x | x | x | | | |
| java.lang.Boolean | x | x | x | x | x | | | |
| java.lang.Byte | x | x | x | x | x | | | |
| java.lang.Short | x | x | x | x | x | | | |
| java.lang.Integer | x | x | x | x | x | | | |
| java.lang.Long | x | x | x | x | x | | | |
| java.lang.Float | x | x | x | x | x | | | |
| java.lang.Double | x | x | x | x | x | | | |
| java.lang.String | x | x | x | x | x | | x | x |
| java.math.BigDecimal | x | x | x | x | x | | | |
| byte[] | | | | | | | x | x |
| java.sql.Array | | | | | | | | |
| java.sql.Blob | | | | | | | | |
| java.sql.Clob | | | | | | x | | |
| java.sql.Date | | | x | x | x | | | |
| java.sql.Ref | | | | | | | | |
| java.sql.Time | | | x | x | x | | | |

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **NU** | **BO** | **CH** | **VC** | **LC** | **CL** | **BI** | **VB** |
| java.sql.Timestamp | | | x | x | x | | | |
| sqlj.runtime.AsciiStream | | | x | x | x | | x | x |
| sqlj.runtime.BinaryStream | | | | | | | x | x |
| sqlj.runtime.CharacterStream | | | x | x | x | | x | x |
| sqlj.runtime.UnicodeStream | | | x | x | x | | x | x |
| java.net.URL | | | x | x | x | | | |
| java.sql.SQLXML | | | | | | | | |
| *any other class/interface* | | | | | | | | |

[1] where: NU correspondds to NUMERIC, BO to BOOLEAN, CH to CHAR, VC to VARCHAR, LC to LONGVARCHAR, CL to CLOB, BI to BINARY, and VB to VARBINARY

**Table 5 — SQLJ output assignability (part 3)**

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **LB** | **BL** | **DT** | **TM** | **TS** | **RF** | **DS** | **ST** |
| boolean | | | | | | | x | x |
| byte | | | | | | | x | x |
| short | | | | | | | x | x |
| int | | | | | | | x | x |
| long | | | | | | | x | x |
| float | | | | | | | x | x |
| double | | | | | | | x | x |
| java.lang.Boolean | | | | | | | x | x |
| java.lang.Byte | | | | | | | x | x |
| java.lang.Short | | | | | | | x | x |
| java.lang.Integer | | | | | | | x | x |

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **LB** | **BL** | **DT** | **TM** | **TS** | **RF** | **DS** | **ST** |
| java.lang.Long | | | | | | | x | x |
| java.lang.Float | | | | | | | x | x |
| java.lang.Double | | | | | | | x | x |
| java.lang.String | x | | x | x | x | | x | x |
| java.math.BigDecimal | | | | | | | x | x |
| byte[] | x | | | | | | x | x |
| java.sql.Array | | | | | | | | |
| java.sql.Blob | | x | | | | | x | x |
| java.sql.Clob | | | | | | | x | x |
| java.sql.Date | | | x | | x | | x | x |
| java.sql.Ref | | | | | | x | | |
| java.sql.Time | | | | x | x | | x | x |
| java.sql.Timestamp | | | x | | x | | x | x |
| sqlj.runtime.AsciiStream | x | | | | | | x | x |
| sqlj.runtime.BinaryStream | x | | | | | | x | x |
| sqlj.runtime.CharacterStream | x | | | | | | x | x |
| sqlj.runtime.UnicodeStream | x | | | | | | x | x |
| java.net.URL | | | | | | | x | x |
| java.sql.SQLXML | | | | | | | | |
| *any other class/interface* | | | | | | | x | x |

[1] where: LB corresponds to LONGVARBINARY, BL to BLOB, DT to DATE, TM to TIME, TS to TIMESTAMP, RF to REF, DS to DISTINCT, and ST to STRUCT

**Table 6 — SQLJ output assignability (part 4)**

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **JO** | **OT** | **DL** | **AR** | **XL** | | | |
| boolean | | | | | | | | |
| byte | | | | | | | | |
| short | | | | | | | | |
| int | | | | | | | | |
| long | | | | | | | | |
| float | | | | | | | | |
| double | | | | | | | | |
| java.lang.Boolean | | | | | | | | |
| java.lang.Byte | | | | | | | | |
| java.lang.Short | | | | | | | | |
| java.lang.Integer | | | | | | | | |
| java.lang.Long | | | | | | | | |
| java.lang.Float | | | | | | | | |
| java.lang.Double | | | | | | | | |
| java.lang.String | | | x | | | | | |
| java.math.BigDecimal | | | | | | | | |
| byte[] | | | | | | | | |
| java.sql.Array | | | | x | | | | |
| java.sql.Blob | | | | | | | | |
| java.sql.Clob | | | | | | | | |
| java.sql.Date | | | | | | | | |
| java.sql.Ref | | | | | | | | |
| java.sql.Time | | | | | | | | |

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **JO** | **OT** | **DL** | **AR** | **XL** | | | |
| java.sql.Timestamp | | | | | | | | |
| sqlj.runtime.AsciiStream | | | | | | | | |
| sqlj.runtime.BinaryStream | | | | | x | | | |
| sqlj.runtime.CharacterStream | | | | | x | | | |
| sqlj.runtime.UnicodeStream | | | | | x | | | |
| java.net.URL | | | x | | | | | |
| java.sql.SQLXML | | | | | x | | | |
| *any other class/interface* | x | x | | | | | | |
| [1] where: JO corresponds to JAVA_OBJECT, OT to OTHER, DL to DATALINK, AR to ARRAY, and XL to SQLXML | | | | | | | | |

A Java class or primitive type *JT* is *SQLJ input assignable* to an SQL type *ST* if Table 7, "SQLJ input assignability (part 1)", Table 8, "SQLJ input assignability (part 2)", Table 9, "SQLJ input assignability (part 3)", or Table 10, "SQLJ input assignability (part 4)", contains an 'x' for the cell identified by the column for the `java.sql.Types` value of *ST* and the row in which *JT* is specified in the first column. In addition, the following condition shall hold for structured and distinct types (*i.e.*, `java.sql.Types` values STRUCT and DISTINCT).

— If the `java.sql.Types` value of *ST* is either DISTINCT or STRUCT, and *JT* is not one of the Java classes or primitive types identified in the first column of Table 9, "SQLJ input assignability (part 3)", (*i.e.*, "any other class/interface" applies), then the user-defined type map that is associated with the connection context class of the SQLJ clause for which input assignability is checked shall specify a Java class or primitive type *JT* that corresponds to *ST*.

— If the `java.sql.Types` value of *ST* is DISTINCT, and *JT* is one of the Java classes or primitive types identified in the first column of Table 9, "SQLJ input assignability (part 3)", then there exists an SQL type *ST1*, where *ST1* is either the representation type of *ST*, or a transform group has been specified for *ST* in the connection context class of the SQLJ clause for which input assignability is checked, and *ST1* is the input parameter type of the to-sql transform function or method of that transform group. *JT* shall be SQLJ input assignable to *ST1*.

— If the `java.sql.Types` value of *ST* is STRUCT, and *JT* is one of the Java classes or primitive types identified in the first column of Table 9, "SQLJ input assignability (part 3)", then a transform group has been specified for *ST* in the connection context class of the SQLJ clause for which input assignability is checked, and *JT* is SQLJ input assignable to the input parameter type of the to-sql transform function or method of that transform group.



 

**Table 7 — SQLJ input assignability (part 1)**

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **TI** | **SI** | **IN** | **BI** | **RL** | **FL** | **DB** | **DC** |
| boolean | x | x | x | x | x | x | x | x |
| byte | x | x | x | x | x | x | x | x |
| short | x | x | x | x | x | x | x | x |
| int | x | x | x | x | x | x | x | x |
| long | x | x | x | x | x | x | x | x |
| float | x | x | x | x | x | x | x | x |
| double | x | x | x | x | x | x | x | x |
| java.lang.Boolean | x | x | x | x | x | x | x | x |
| java.lang.Byte | x | x | x | x | x | x | x | x |
| java.lang.Short | x | x | x | x | x | x | x | x |
| java.lang.Integer | x | x | x | x | x | x | x | x |
| java.lang.Long | x | x | x | x | x | x | x | x |
| java.lang.Float | x | x | x | x | x | x | x | x |
| java.lang.Double | x | x | x | x | x | x | x | x |
| java.lang.String | x | x | x | x | x | x | x | x |
| java.math.BigDecimal | x | x | x | x | x | x | x | x |
| byte[] | | | | | | | | |
| java.sql.Array | | | | | | | | |
| java.sql.Blob | | | | | | | | |
| java.sql.Clob | | | | | | | | |
| java.sql.Date | | | | | | | | |
| java.sql.Ref | | | | | | | | |
| java.sql.Time | | | | | | | | |

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **TI** | **SI** | **IN** | **BI** | **RL** | **FL** | **DB** | **DC** |
| java.sql.Timestamp | | | | | | | | |
| sqlj.runtime.AsciiStream | | | | | | | | |
| sqlj.runtime.BinaryStream | | | | | | | | |
| sqlj.runtime.CharacterStream | | | | | | | | |
| sqlj.runtime.UnicodeStream | | | | | | | | |
| java.net.URL | | | | | | | | |
| java.sql.SQLXML | | | | | | | | |
| *any other class/interface* | | | | | | | | |

[1] where: TI corresponds to TINYINT, SI to SMALLINT, IN to INTEGER, BI to BIGINT, RL to REAL, FL to FLOAT, DB to DOUBLE, and DC to DECIMAL

**Table 8 — SQLJ input assignability (part 2)**

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **NU** | **BO** | **CH** | **VC** | **LC** | **CL** | **BI** | **VB** |
| boolean | x | x | x | x | x | | | |
| byte | x | x | x | x | x | | | |
| short | x | x | x | x | x | | | |
| int | x | x | x | x | x | | | |
| long | x | x | x | x | x | | | |
| float | x | x | x | x | x | | | |
| double | x | x | x | x | x | | | |
| java.lang.Boolean | x | x | x | x | x | | | |
| java.lang.Byte | x | x | x | x | x | | | |
| java.lang.Short | x | x | x | x | x | | | |
| java.lang.Integer | x | x | x | x | x | | | |

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **NU** | **BO** | **CH** | **VC** | **LC** | **CL** | **BI** | **VB** |
| java.lang.Long | x | x | x | x | x | | | |
| java.lang.Float | x | x | x | x | x | | | |
| java.lang.Double | x | x | x | x | x | | | |
| java.lang.String | x | x | x | x | x | | x | x |
| java.math.BigDecimal | x | x | x | x | x | | | |
| byte[] | | | | | | | x | x |
| java.sql.Array | | | | | | | | |
| java.sql.Blob | | | | | | | | |
| java.sql.Clob | | | | | | x | | |
| java.sql.Date | | | x | x | x | | | |
| java.sql.Ref | | | | | | | | |
| java.sql.Time | | | x | x | x | | | |
| java.sql.Timestamp | | | x | x | x | | | |
| sqlj.runtime.AsciiStream | | | x | x | x | x | x | x |
| sqlj.runtime.BinaryStream | | | | | | | x | x |
| sqlj.runtime.CharacterStream | | | x | x | x | x | x | x |
| sqlj.runtime.UnicodeStream | | | x | x | x | x | x | x |
| java.net.URL | | | x | x | x | | | |
| java.sql.SQLXML | | | | | | | | |
| *any other class/interface* | | | | | | | | |

[1] where: NU corresponds to NUMERIC, BO to BOOLEAN, CH to CHAR, VC to VARCHAR, LC to LONGVARCHAR, CL to CLOB, BI to BINARY, and VB to VARBINARY

**Table 9 — SQLJ input assignability (part 3)**

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **LB** | **BL** | **DT** | **TM** | **TS** | **RF** | **DS** | **ST** |
| boolean | | | | | | | x | x |
| byte | | | | | | | x | x |
| short | | | | | | | x | x |
| int | | | | | | | x | x |
| long | | | | | | | x | x |
| float | | | | | | | x | x |
| double | | | | | | | x | x |
| java.lang.Boolean | | | | | | | x | x |
| java.lang.Byte | | | | | | | x | x |
| java.lang.Short | | | | | | | x | x |
| java.lang.Integer | | | | | | | x | x |
| java.lang.Long | | | | | | | x | x |
| java.lang.Float | | | | | | | x | x |
| java.lang.Double | | | | | | | x | x |
| java.lang.String | x | | x | x | x | | x | x |
| java.math.BigDecimal | | | | | | | x | x |
| byte[] | x | | | | | | x | x |
| java.sql.Array | | | | | | | | |
| java.sql.Blob | | x | | | | | x | x |
| java.sql.Clob | | | | | | | x | x |
| java.sql.Date | | | x | | x | | x | x |
| java.sql.Ref | | | | | | x | | |
| java.sql.Time | | | | x | x | | x | x |

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **LB** | **BL** | **DT** | **TM** | **TS** | **RF** | **DS** | **ST** |
| java.sql.Timestamp | | | x | | x | | x | x |
| sqlj.runtime.AsciiStream | x | | | | | | x | x |
| sqlj.runtime.BinaryStream | x | x | | | | | x | x |
| sqlj.runtime.CharacterStream | x | | | | | | x | x |
| sqlj.runtime.UnicodeStream | x | | | | | | x | x |
| java.net.URL | | | | | | | x | x |
| java.sql.SQLXML | | | | | | | | |
| *any other class/interface* | | | | | | | x | x |
| [1] where: LB corresponds to LONGVARBINARY, BL to BLOB, DT to DATE, TM to TIME, TS to TIMESTAMP, RF to REF, DS to DISTINCT, and ST to STRUCT | | | | | | | | |

**Table 10 — SQLJ input assignability (part 4)**

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **JO** | **OT** | **DL** | **AR** | **XL** | | | |
| boolean | | | | | | | | |
| byte | | | | | | | | |
| short | | | | | | | | |
| int | | | | | | | | |
| long | | | | | | | | |
| float | | | | | | | | |
| double | | | | | | | | |
| java.lang.Boolean | | | | | | | | |
| java.lang.Byte | | | | | | | | |
| java.lang.Short | | | | | | | | |
| java.lang.Integer | | | | | | | | |

| | java.sql.Types constants[1] | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Java Data Types and Classes** | **JO** | **OT** | **DL** | **AR** | **XL** | | | |
| java.lang.Long | | | | | | | | |
| java.lang.Float | | | | | | | | |
| java.lang.Double | | | | | | | | |
| java.lang.String | | | x | | | | | |
| java.math.BigDecimal | | | | | | | | |
| byte[] | | | | | | | | |
| java.sql.Array | | | | x | | | | |
| java.sql.Blob | | | | | | | | |
| java.sql.Clob | | | | | | | | |
| java.sql.Date | | | | | | | | |
| java.sql.Ref | | | | | | | | |
| java.sql.Time | | | | | | | | |
| java.sql.Timestamp | | | | | | | | |
| sqlj.runtime.AsciiStream | | | | | | | | |
| sqlj.runtime.BinaryStream | | | | | | | | |
| sqlj.runtime.CharacterStream | | | | | | | | |
| sqlj.runtime.UnicodeStream | | | | | | | | |
| java.net.URL | | | x | | | | | |
| java.sql.SQLXML | | | | | x | | | |
| *any other class/interface* | x | x | | | | | | |

[1] where: JO corresponds to JAVA_OBJECT, OT to OTHER, DL to DATALINK, AR to ARRAY, and XL to SQLXML

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

Under some situations, a single SQL CALL statement might return multiple `java.sql.ResultSet` objects. Because SQL has no mechanism to define `java.sql.ResultSet` objects as formal OUT or INOUT parameters, such `java.sql.ResultSet` objects are referred to as *side-channel* result sets. The ExecutionContext method "**getNextResultSet**" allows navigation through these results.

After implicit or explicit use of some ExecutionContext **execCon** in association with an SQL CALL statement, the first call to **execCon.getNextResultSet** returns the first side-channel result set produced by that CALL statement. Subsequent calls to **getNextResultSet** optionally close the current `java.sql.ResultSet` object, and advance to and return the next. **getNextResultSet** returns null if there are no further side-channel result sets.

#### 4.3.16.1 Resource management with multiple results

Under normal circumstances, the resources associated with the execution of an SQL-statement are released as soon as the execution completes. However, if there are multiple results, the resources are not released until all results have been processed using **getNextResultSet**. If an execution context with pending results is used to execute another SQL-statement, then the pending results are discarded.

If the invocation of an SQL-invoked procedure does not produce side-channel result sets, then all resources are automatically reclaimed as soon as the CALL execution completes.

### 4.3.17 JDBC/SQLJ ResultSet interoperability

To facilitate the interaction between dynamic SQL and SQLJ's strongly-typed iterators, SQLJ provides a way to obtain a `java.sql.ResultSet` object from an SQLJ iterator object and to create an SQLJ iterator object from a `java.sql.ResultSet` object.

#### 4.3.17.1 Creating an SQLJ iterator from a java.sql.ResultSet object

The SQLJ iterator conversion statement allows a `java.sql.ResultSet` object to be manipulated as an SQLJ strongly-typed iterator object. Given a `java.sql.ResultSet` object **rs** and a strongly typed SQLJ iterator object **iter**, the iterator conversion statement can be used to assign a new iterator object to **iter** based on the contents of **rs**.

```
#sql iter = {CAST :rs };
```

NOTE 15 — Closing an iterator object created by an iterator conversion statement will also close the associated `java.sql.ResultSet` object.

The iterator conversion statement can be used to instantiate an SQLJ strongly-typed iterator object from a `java.sql.ResultSet` object provided the type, name and number of columns in the `java.sql.ResultSet` object are compatible with those of the declared iterator object. See the <iterator conversion clause> for further details.

Once an iterator object has been created by an <iterator conversion clause>, the result of calling methods on the original `java.sql.ResultSet` object is implementation-defined.

### 4.3.17.2 Obtaining a java.sql.ResultSet object from an SQLJ iterator object

Every SQLJ iterator object has a **getResultSet** method that returns a `java.sql.ResultSet` object representation of its data. The **getResultSet** method is part of the `sqlj.runtime.ResultSetIterator` interface, which is implemented by SQLJ strongly typed iterator classes (both named and positioned). It allows query results to be processed using a `java.sql.ResultSet` object rather than an SQLJ iterator object.

NOTE 16 — Support for the **getResultSet** method is runtime implementation-defined, and is not part of Core SQLJ. See Subclause 12.2.5.3.4, "getResultSet ()".

### 4.3.17.3 Obtaining a java.sql.ResultSet object from an untyped iterator object

SQLJ does not support the direct creation of a `java.sql.ResultSet` object as the result of an SQLJ query. To obtain a `java.sql.ResultSet` object associated with an SQLJ query, an SQLJ iterator object is populated as the result of the query, and the **getResultSet** method of the iterator object is called to return a `java.sql.ResultSet` object, as described in the previous Subclause. In cases where the client needs only a `java.sql.ResultSet` object and does not wish to process results with a strongly-typed iterator object, a client is permitted to use an untyped ResultSetIterator object instead. An untyped ResultSetIterator object is declared as an instance of interface `sqlj.runtime.ResultSetIterator`.

The ResultSetIterator interface is the root interface of all SQLJ iterators and supports the **getResultSet** and **close** methods, among others. As such, it can be used to obtain the results of an SQLJ query and later return them to the client as a `java.sql.ResultSet` object. Further, it is used to release SQLJ related resources once the results have been processed.

An untyped ResultSetIterator object provides a convenient way to obtain the results of an SQLJ query and later access them using a `java.sql.ResultSet` object. Unlike its strongly-typed counterpart, the untyped ResultSetIterator object does not require an additional class declaration. If using an untyped ResultSetIterator object in an SQLJ query, translate-time type checking of the select list items is not performed.

### 4.3.17.4 Iterator and java.sql.ResultSet object resource management

Calling the **close** method of an SQLJ iterator object causes the associated `java.sql.ResultSet` object (if any) to be closed. If an iterator object is not explicitly closed before it is garbage collected, then the finalize method of the iterator object implicitly calls **close**. Iterator objects consume resources in the Java Virtual Machine and, typically, in the SQL-environment for as long as they remain open. So, it is important to explicitly close Iterator objects when the application is done with them rather than waiting for garbage collection.

Both SQLJ iterator objects and `java.sql.ResultSet` objects respond to the `close()` method. When an iterator object produces a `java.sql.ResultSet` object via the `getResultSet()` method, it is sufficient to close only the iterator object, as this will also close the associated `java.sql.ResultSet` object. However, it is not sufficient to close only the associated `java.sql.ResultSet` object, as this does not cause the containing iterator object to be closed, and therefore, resources maintained by the iterator object will not be released



 

until it is garbage-collected. These restrictions are true of untyped ResultSetIterator objects as well as named and positioned iterator objects.

The **isClosed** method of an iterator object returns **true** if the **close** method has been called on the iterator object. If **isClosed** is **true**, invocation of close has no effect, and the effect of invoking any other method is implementation-defined. The semantics of calling **close** on a `java.sql.ResultSet` object that has already been closed is implementation-defined.

### 4.3.18 Multi-threading considerations

SQLJ can be used to write multithreaded applications. The SQLJ runtime supports multiple threads sharing the same connection context. However, SQLJ programs are subject to synchronization limitations imposed by the underlying DBMS implementation. If a DBMS implementation mandates explicit synchronization of statements executed in a specific connection, then an SQLJ program using that implementation would require a similar synchronization of SQL-statements.

Whereas connection contexts can be safely shared between threads, execution contexts should only be shared if their use is properly synchronized. If an execution context is shared, the results of an SQL-statement performed by one thread will be visible in the other thread. If both threads are executing SQL-statements, a race condition can occur in which the results of an execution in one thread are overwritten by the results of an execution in the next thread before the first thread has processed the original results. Furthermore, if a thread attempts to execute an SQL-statement using an execution context that is currently being used to execute an operation in another thread, a runtime exception is thrown. To avoid such problems, each thread should use a distinct execution context whenever an SQL-statement is executed on a shared connection context.

### 4.3.19 User-defined data types

SQLJ supports the manipulation of instances of user-defined data types, such as structured types and distinct types. Instances of such data types can be retrieved into or created from host variables of an appropriate Java type, based on type mapping information specified for a specific connection context class. Java resource bundles are used as the mechanism for specifying type mapping information.

Type mapping is specified in one or more entries contained in a *properties file*. Each property entry in the file defines a correspondence between a Java class and an SQL user-defined type. The entry may indicate that the SQL type is a structured type or a distinct type with the keyword `STRUCT` or `DISTINCT` preceding the type name, respectively; such an indication is optional, and is only needed to resolve ambiguities in cases where the SQL type is required for registering **`OUT`** parameters.

Java classes used in the definition of a type mapping for structured and distinct types have to fulfill the requirements specified in the chapter “Customized Type Mapping” of [JDBC]. In other words, they have to implement the interface **java.sql.SQLData**, which is used by the SQLJ runtime implementation to supply the newly created instance of the Java class with data from the instance of the respective SQL type.

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

```
#sql context Ctx with (typeMap = "packagename.filename");
```

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

SQLJ applications can then define host variables or iterator objects based on the Java types that participate in the type map.

A positioned iterator object is used in conjunction with a FETCH...INTO statement to retrieve data, including instances of user-defined types mapped in the properties file.

In the same way, a Java variable whose type corresponds to a user-defined type can be used for the definition of named iterator objects, as host variables, and in host expressions.

The SQLJ translator also checks for type correctness for user-defined types.

This mechanism also handles SQL type hierarchies and, correspondingly, Java class hierarchies.

### 4.3.20 Batch updates

Batch updates allow statements to be grouped together and then sent as a batch to the SQL-implementation for execution using a single round trip. This feature is typically used for a series of UPDATE, INSERT, or DELETE statements within a loop. This subclause outlines how SQLJ supports batch updates.

#### 4.3.20.1 Batchable statements and batch compatibility

A batchable statement is a statement that is able to be grouped with one or more other statements for execution as a batch at runtime. Such a group of batchable statements is called a statement batch, or simply batch. As with JDBC, batching in SQLJ is an optional capability. Accordingly, whether a particular statement is batchable or not depends on the connection and customization used to execute the statement at runtime. In general, DML, DDL and SQL-invoked procedure calls with no **OUT** parameters are considered batchable.

The following types of statements are never batchable:

— Queries (single and multi row)

— Transaction control (COMMIT, ROLLBACK, SET TRANSACTION, SAVEPOINT, RELEASE SAVEPOINT)

— Statements with **OUT** parameters (SQL-invoked functions, PSM assignment, SQL-invoked procedures with outs or side-channel results, blocks with outs)

A statement is *batch compatible* with a particular statement batch if the statement is both batchable and compatible with (can be added to) the batch. Whether a particular statement is batch compatible with a particular statement batch depends on the connection and customization used to execute the statement at runtime. For an implementation based on [JDBC], a batchable statement with one or **IN** parameters is only batchable with other instances of the same statement. A batchable statement with no **IN** parameters is only batchable with other statements with no **IN** parameters. However, runtime implementations that do not rely solely on [JDBC] may



 

additionally allow unrelated statements with **IN** parameters to be batched together. Any batchable statement may potentially be batched with any other batchable statement that is created using the same connection.

Two batchable statements that are executed through different SQL-implementations (using different SQL-connections) are never batch compatible.

### 4.3.20.2 Statement batching API

In SQLJ, batch update capability is enabled using the `setBatching` method of the `ExecutionContext` class. When batching is enabled on a particular execution context object (via `setBatching`), then any batchable statement encountered is deferred for batched execution. In such cases, the execution context object is said to contain a *pending statement batch*. Subsequent re-execution of the statement causes the statement to be added again to the statement batch (with possibly different host expression values). A pending statement batch can be explicitly executed at any time using `ExecutionContext.executeBatch()`.

In terms of the earlier definitions, if a pending statement batch exists on a particular execution context object and a batch compatible statement is encountered, then it is added to the batch for deferred execution. A statement is batch compatible if it is the same as all others in the statement batch. A statement batch that contains only instances of the same statement (possibly differing only in host expression bind values) is called a *homogeneous batch*. It is also possible to have a *heterogeneous batch* in which one or more statements differ from others in the batch. Typically, a heterogeneous batch consists of statements that do not contain any bind expressions.

The fact that a particular statement is batchable does not mean that it is compatible with every statement batch. The runtime connection and customization ultimately determine batch compatibility. For implementations of [JDBC], statements with bind expressions can only be added to homogeneous batches, and heterogeneous batches can only contain statements without bind expressions. When a batchable statement is encountered that is not compatible with the current statement batch, then the statement batch is executed implicitly and the statement is added to a new statement batch. Similarly, when a statement that is not batchable (such as SELECT or COMMIT) is encountered, the statement batch is implicitly executed prior to the execution of the statement. Implicit batch execution allows programs to use batch updates without explicitly calling `executeBatch()`.

Note that the update counts resulting from the last implicitly executed batch can be obtained using the method `ExecutionContext.getBatchUpdateCounts()`.

A given execution context object can only manage one statement batch at a time. A client who wants to batch two statements that are not batch compatible with one another shall use two distinct execution context objects.

It should be noted that explicit specification of an execution context object is not required for batch updates. As an alternative, batching can be enabled on the execution context object contained within a particular connection context object.

### 4.3.20.3 Execution status and update counts

When a statement is batched instead of executed, calling `ExecutionContext.getUpdateCount()` returns the constant `ExecutionContext.NEW_BATCH_COUNT` if a new statement batch was created, or `ExecutionContext.ADD_BATCH_COUNT` if the statement was added to the pending statement batch. Checking for this constant is a reliable way to determine whether the last statement was batched, and if so, whether it was added to the pending batch or started a new batch.

When a pending statement batch exists, calling `ExecutionContext.executeBatch()` executes the batch. The update count is set to `ExecutionContext.EXEC_BATCH_COUNT`. An array of `int` is returned reflecting the individual update counts of each statement in the batch. The array is ordered according to the order in which statements were added to the batch. The update count array for the last batch executed can also be obtained using the method `ExecutionContext.getBatchUpdateCounts()`. This is particularly useful when the batch was updated implicitly rather than explicitly. The array returned by `getBatchUpdateCounts()` reflects the result of the last successful implicit or explicit call to `executeBatch()`. It is ordered according to the order in which commands were inserted into the batch, and each element either contains an update count, or the value -2 as a generic success indicator, or the value -3 as a generic failure indicator. If a failure occurs during batch execution that prevents the remainder of the batch to be executed, then the array returned may also be shorter than the original batch and—in this case—each element shall contain either a non-negative update value or the value -2 as a generic success indicator. The array is not updated when the call to `executeBatch()` results in an exception. The array is null if no batch has yet been completed.

### 4.3.20.4 Program semantics and exceptions

When a statement batch is executed using `executeBatch()`, the statements contained in the batch are executed in order. If execution of one of the statements results in an exception, the remaining statements are not executed and the exception is thrown by `executeBatch()`. Note, however that the exception does not rollback the statements that were executed earlier in the batch. When appropriate, the exception is an instance of `java.sql.BatchUpdateException`, which is a class that extends `java.sql.SQLException` and adds information about the statements in the batch that completed successfully. If a statement batch is implicitly executed as a result of executing another statement, and the execution of the batch results in an exception, the statement that triggered the batch execution is not executed.

Because exceptions can happen in the middle of a batch, it is generally recommended that autocommit is turned off when using batch updates. Disabling autocommit allows the application to decide whether or not to commit the transaction in the event that an error occurs and some of the commands in a batch fail to execute.

As implied by the above rules, the execution semantics of programs that use batch updates are somewhat different than programs that do not. These differences are summarized in the following list.

— A single exception is thrown for the batch of statements, not each individual statement.

— Once an exception occurs, the rest of the pending statements in the statement batch are not executed. There is no convenient way to handle the exception and continue execution of the rest of the statements.

— Statement execution is deferred until the batch is executed rather than when the statement is first encountered. When a batch is implicitly executed during the execution of another statement, an exception resulting from the batch execution may be appear to be thrown as the result of the current statement's execution.

When a batch is implicitly executed by another statement, the batch is executed before the statement is executed, but after **`IN`** parameters have been evaluated and passed to the statement. Deferring batch execution until after **`IN`** parameters have been bound allows the runtime engine to collect as much information as possible before determining whether a statement is compatible with a particular batch. This allows, for example, positioned updates using WHERE CURRENT OF to be batched if the input iterator object is the same iterator object in each case.



 

### 4.3.20.5 Batch cancellation and disabling

A pending statement batch can be canceled before execution using the method `ExecutionContext.cancel()`. Once `cancel()` has been called, the pending batch is cleared and can no longer be executed. The next batchable statement encountered will be added to a new statement batch.

It is the responsibility of the client to execute or cancel a pending statement batch before discarding the execution context object that contains the batch. The execution context object's finalizer will not implicitly execute or cancel a pending statement batch.

Batching can be disabled using the method `ExecutionContext.setBatching(false)`. Disabling batching in this way means that further statements will not be added to the pending batch. However, the pending batch, if any, is not affected. It will be executed by the next implicit or explicit call to `executeBatch()`, or canceled with a call to `cancel()`, as usual. A client can use the method `ExecutionContext.isBatching()` to determine whether or not batching is currently enabled on a particular execution context object. Note that this method is used only to determine whether batching is currently enabled, but not whether a pending batch exists.

### 4.3.20.6 Specification of a batching limit

The method `ExecutionContext.setBatchLimit()` permits users to specify that calls to `executeBatch()` should be performed implicitly. The batch limit may be given in the following ways:

— As a positive integer *n* — in this case, the `executeBatch()` method will be executed whenever the current batch size reaches *n*.

— As a constant `ExecutionContext.UNLIMITED_BATCH` — in this case, no implicit call to `executeBatch` will be performed, unless one of the conditions for implicit batch execution discussed earlier is met.

— As a constant `ExecutionContext.AUTO_BATCH` — in this case, the `executeBatch()` method will be executed at a point that is chosen by the SQLJ runtime implementation. The point when the current batch is executed implicitly should be chosen so that out-of-memory conditions due to batching are reasonably avoided.

By default, `ExecutionContext` objects are initialized to a batch limit of `UNLIMITED_BATCH`. By permitting users to specify a batch limit, an SQLJ program can very easily be changed to a batching SQLJ program.

Consider the scenario when one Java SQL-invoked procedure uses SQLJ to call another Java SQL-invoked procedure, which in turn uses SQLJ to execute another SQL-statement, and both execute in the same Java Virtual Machine with the same execution context object (often associated with the default connection context object). The behavior for batching in this situation conforms to the existing behavior for similar operations.

The batching attribute of the execution context object (set via `ExecutionContext.setBatching()`) behaves like the other execution control attributes (max fields size, max rows, query timeout). Once set, it affects the next SQLJ executable clause to start executing regardless of whether the next SQLJ executable clause is made at the same call level, in a recursive call level, or in an outer call level.

A pending batch is treated in the same way that pending side-channel results are. Just as pending side-channel results are implicitly cleaned up and closed when another SQL-statement is encountered or an outer call completes execution, pending batches are implicitly executed when another SQL-statement is encountered or an outer call completes execution. As an example, suppose batching is enabled and we execute a non-batchable SQL-statement that results in a call to a Java SQL-invoked procedure, which in turn performs some SQL-statements that are added to a new batch. If the called Java SQL-invoked procedure returns without executing the batch,

then the originating SQL-statement will implicitly execute the batch when execution control is returned. Implicit batch execution also happens when control returns from any batch execution (via `ExecutionContext.executeBatch()`) since statements contained in a batch could themselves add statements to a new batch when executed.

### 4.3.21 SQLJ language elements

Elements of the SQL language are treated in various ways by SQLJ.

— *Executable SQL statements*: This part of ISO/IEC 9075 directly adopts the executable SQL-statements (most of the <SQL schema statement>s, <SQL data statement>s, and <SQL transaction statement>s) that manipulate SQL data, definitions, and transactions, substantially as they are specified in embedded SQL and in SQL's module language.

— *Dynamic SQL*: SQLJ does not directly support dynamic SQL, which is handled separately by JDBC.

— *Declarations*: This part of ISO/IEC 9075 replaces <declare cursor> and <host variable definition> by declarations of Java types for declaring iterator classes and other data items that have SQL attributes.

— *Program control*: The <embedded exception declaration>, <SQL session statement>s, <SQL connection statement>s, and <SQL diagnostics statement>s that serve to knit together SQL and host language environments by managing exceptions, SQL-connections, and diagnostics are omitted in SQLJ. Java directly expresses the types of exceptions, SQL-connections, and diagnostics, and can manipulate those objects using standard programming techniques.

#### 4.3.21.1 <cursor name>

In SQL language, <cursor name> is a simple identifier. The equivalent SQLJ construct is <iterator host expression>. <iterator host expression> is a Java expression, the result type of which shall be an instance of a generated iterator class (that is, a generated named iterator class or a generated positioned iterator class), or a subclass of such a class.

#### 4.3.21.2 SQL-schema, SQL-data, and SQL-transaction statements

The SQL-schema, SQL-data, and SQL-transaction statements are treated as SQLJ clauses in this part of ISO/IEC 9075 and are consequentially treated as ordinary embedded SQL-statements.

#### 4.3.21.3 <SQL dynamic statement>

The categories of <SQL dynamic statement> and <dynamic declare cursor> are omitted from this part of ISO/IEC 9075. In addition, the dynamic statements PREPARE, DESCRIBE, EXECUTE, DEALLOCATE, GET DESCRIPTOR, and SET DESCRIPTOR are omitted from this part of ISO/IEC 9075 since, in Java application programs, dynamic operations are subsumed by [JDBC].



 

### 4.3.21.4 <SQL connection statement>

The <SQL connection statement> is replaced in this part of ISO/IEC 9075 by direct Java construction and manipulation of connection objects (defined by the interface `sqlj.runtime.ConnectionContext`). That enables the capability for SQLJ programs to open multiple SQL-connections simultaneously to the same or different SQL-servers by explicit use of connection objects in SQLJ clauses.

### 4.3.21.5 <host variable definition>

Embedded SQL specifies that <host variable definition>s are contained in special program sections bound by EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION. This part of ISO/IEC 9075 does not define a <host variable definition> section. Any Java variable, parameter, or Java field (of an object) is permitted to be used as a host variable.

This part of ISO/IEC 9075 extends traditional embedded SQL support by allowing Java host expressions to appear directly in embedded SQL-statements. Host expressions are prefixed by a colon (:) followed by an optional parameter mode identifier (IN, OUT, INOUT) followed by a parenthesized expression clause. An expression clause contains a Java expression that either evaluates to a single value (in the case of IN mode) or is a Java l-valued expression (in the case of an OUT or INOUT mode).

The evaluation of host expressions does have side effects in a Java program as they are evaluated by the Java Virtual Machine rather than the SQL-server. Host expressions are evaluated left to right within the SQL-statement prior to submission to the SQL-server.

### 4.3.21.6 <embedded exception declaration>

This part of ISO/IEC 9075 does not define an <embedded exception declaration>. In [ISO9075-2], the <embedded exception declaration> has these forms:

```
EXEC SQL WHENEVER exception_condition GOTO program_label;
EXEC SQL WHENEVER exception_condition CONTINUE;
```

The Java language does not support any form of “go to” statement; therefore the direct transliteration of the <embedded exception declaration> into Java is not possible. Instead, Java provides a **try...catch** statement that associates a handler for certain exceptions in the Java block in which those exceptions might be raised.

> NOTE 17 — In addition, Java has well-developed rules for declaring and handling exceptions; thus, the <embedded exception declaration> does not add value. Other object oriented languages have facilities for declaring and handling exceptions, similar to those in Java.

JDBC defines an exception, globally named `java.sql.SQLException`, as the superclass of exceptions that are returned from SQL. This part of ISO/IEC 9075 follows that precedent, to facilitate interoperability between this part of ISO/IEC 9075 and [JDBC].

### 4.3.21.7 <SQL diagnostics statement>

This part of ISO/IEC 9075 follows the [JDBC] methodology for handling return information traditionally found in the diagnostics areas of SQL. Abnormal termination and certain runtime errors (*e.g.*, NULL retrieval to non-nullable datatypes) are processed using exception handling. Other status information (*e.g.*, update count) are processed by using methods on the connection context and execution context objects.

### 4.3.21.8 Cursor declaration

The <declare cursor> statement of SQL language declares a single name for both a query and its associated result set in the host program. This part of ISO/IEC 9075, by contrast, distinguishes between a query and the result set that it returns. If an SQLJ clause containing a query is evaluated, it returns an iterator object containing the result set of rows selected by that query. The type of an iterator object is a Java class that encodes the number and types (and names) of columns in the result set, allowing type checking of operations on an iterator object.

Beneath the layer of abstraction provided by SQLJ iterators, an SQL-server creates and manipulates cursors. The *implicit cursor* of an iterator **UC** is the cursor manipulated by an SQL-server when methods are invoked against the corresponding instance of an object, the type of which implements `sqlj.runtime.ResultSetIterator`. The *implicit <declare cursor>* of an iterator **UC** is the <declare cursor> effectively performed by an SQL-server as a result of the execution of an <assignment clause> whose <assignment spec clause> immediately contains a <query clause>.

When <assignment spec clause> immediately contains <query clause>, the <query clause> provides the implicit <declare cursor>'s <query expression>. An implicit <declare cursor>'s <cursor returnability> is always WITH RETURN. The <Lval expression> **LV** immediately contained in <assignment clause> either refers to an object of a generated iterator class or to an object the type of which implements `sqlj.runtime.ResultSetIterator`. When **LV** refers to an object of a generated iterator class, the associated <iterator declaration clause>'s <declaration with list> specification of the iterator's sensitivity, holdability, and updateColumns respectively provide the implicit <declare cursor>'s <cursor sensitivity>, <cursor holdability>, and update <column name list> specifications. In addition, <cursor scrollability> is implicitly SCROLL if the associated <iterator declaration clause>'s <interface list> contains the <predefined interface class> `sqlj.runtime.Scrollable`; otherwise, it is implicitly NO SCROLL.

### 4.3.21.9 Input parameters to SQL-statements

This part of ISO/IEC 9075 extends the approach of SQL language for input parameters to SQL-statements by allowing generalized host expressions to appear wherever host variables are allowed to appear.

### 4.3.21.10 Extracting column values from SQLJ iterators

SQLJ supports two approaches to accessing column values from iterator objects: by position and by name. The <fetch statement> of SQL accesses columns only by position. In the following example, the first column in the row is assigned to `var1`, the second to `var2`, and the third to `var3`:



 

```
EXEC SQL FETCH cursor1 INTO :var1,:var2,:var3;
```

SQLJ supports a modified version of the FETCH statement. It also supports access to columns by name, through generated methods with the names and types of the columns.

#### 4.3.21.11 <open statement> and cursors

SQL has an <open statement> to open and re-open its named cursors that represent both a query and its set of result rows, (that is, its *result set*):

```
EXEC SQL OPEN cursor1;
```

This part of ISO/IEC 9075 does not provide an OPEN operation to open or re-open iterator objects.

This part of ISO/IEC 9075 does not name a static query nor treat it as data. Instead, a query returns an iterator object that is manipulated as data. An application can, in effect, name a query by writing it in an SQLJ clause in the body of a method. Methods are invoked by their names, and can return iterator objects as their values.

### 4.3.22 SQLJ, JDBC, and SQLExceptions and SQLWarnings

In the text of other parts of ISO/IEC 9075, the normal or abnormal termination of statement execution and expression evaluation is indicated by stating that a "condition" is "raised", followed by a statement of what specific condition is raised (*e.g.*, "a completion condition is raised: *no data*" or "an exception condition is raised: *data exception — division by zero*.").

This part of ISO/IEC 9075, because of its close relationship to the Java programming language and to [JDBC], uses different terminology. In this part of ISO/IEC 9075, many abnormal terminations are indicated by stating that an "SQLException condition" is "thrown", followed by a statement of the specific condition class and subclass (*e.g.*, "an SQLException condition is thrown: *OLB-specific error — unsupported feature*").

In other situations, the statement is made that "an exception is thrown" or "throws an exception", with no specification of what particular exception is thrown. The absence of that specification is caused by SQLJ's dependence on [JDBC]. When that sort of statement is made, the exception that is thrown is determined by the JDBC driver vendor, and not by ISO/IEC 9075. In practice, JDBC drivers specify the SQLSTATE, and other pertinent details, in the SQLWarning or SQLException objects resulting from a completion or exception condition originating in an SQL-implementation. However, when the completion or exception condition originates within the JDBC driver itself, details of the resulting SQLWarning or SQLException objects are implementation-defined.



### 4.3.23 Profile generation and naming

An SQLJ profile represents the SQL-statements performed on a particular connection context class within a particular source file. Every <executable clause> in an SQLJ program is associated with exactly one <connection context> (which might be implicit or explicit). <executable clause>s are grouped within a program according to the class of the associated <connection context> and this grouping is reflected in the profile. If the number

of <connection context> classes associated with <executable clause>s in an SQLJ program is greater than one, then a distinct profile is created for each <connection context> class.

The generated name of a profile is composed, textually, of parts, with no additional separators occurring between those parts. Its parts include:

— An optional representation of its associated package that will, if specified, be followed by a period.

— The original source file name without its filename extension.

— The predefined text string '_SJProfile'.

— A profile identification number that is unique among profile identification numbers in the source profile.

Informally presented in BNF form, this might appear as follows:

```
[ packagePart<period> ]filenamePart_SJProfileidentificationNumber
```

where:

— *packagePart* is the package name defined by the package declaration in the original source file. If there is no package declaration in the original source file, then this component of the name is omitted.

— filenamePart is the name of the original source file, without a filename extension. If the original source is not associated with a file that has a logical name, then filenamePart is the name of the first public class appearing in the source, or, in the absence of any public classes, the first class appearing in the source.

— identificationNumber is a non-negative integer used to uniquely identify the profile. A single source file can produce more than one profile. In such cases, the profiles produced are numbered consecutively, starting with 0 (zero). If a source file produces only one profile, *identificationNumber* is 0 (zero).

This naming convention enables easy recognition of profile files and determination of the source file with which they are associated.

### 4.3.24 SQLJ application packaging

After development of an SQLJ application has been completed, the application might be packaged for deployment as a JAR file. JAR (Java Archive) is a platform-independent file format specified by Java that aggregates many files into one. SQLJ applications are packaged as JAR files in order that they can be inspected and modified as a unit by profile customization utilities.

Every JAR file includes a manifest file that describes the contents of the JAR. For each SQLJ profile in the application, a section is created in the manifest file contained in the JAR file. The manifest file is used by the SQLJ customization utilities to locate and load the appropriate application profiles. The SQLJ profile section of the manifest file has entries that specify the name of the profile file. The name of a profile is composed, textually, of parts with no additional separators between those parts. Its parts include:

— A specification of the profile's package, given in path format.

— A directory separator '/' (<solidus>).

— The profile name without its filename extension.

— A period.



 

— The `class` or `ser` filename extension.

Informally presented in BNF form, this might appear as follows:

```
profileName ::=
    pathPart<solidus>profileFilePart<period>{ class | ser }
```

And, when used in the JAR manifest:

— **Name:** *profileName* **SQLJProfile: TRUE**

where:

— pathPart is the package name of the profile in path format, as specified by the manifest file format for "Name" headers.

— profileFilePart is the name of the profile.

— If the profile exists in class file format, then the name has the extension **.class**. Otherwise, if the profile exists as a serialized object, then the name has extension **.ser**. Only two file formats (.class and .ser) are currently supported. Other file formats might be added in the future. Note that the customization process will modify the contents of an existing profile such that any customized profile will exist in serialized format only.

### 4.3.25 Profile customizer interface

A *profile customizer* is a JavaBean component, as defined by [JavaBeans], that customizes a profile to allow implementation-defined features, extensions and/or behavior. A class is a profile customizer if it implements the **sqlj.runtime.profile.util.ProfileCustomizer** interface, provides an accessible parameterless constructor, and conforms to the JavaBeans API to expose its properties.

A profile customizer implements the following methods:

— **acceptsConnection**

```
public boolean acceptsConnection(java.sql.Connection conn)
```

Returns **true** if this customizer is able to customize profiles using the passed `java.sql.Connection` object, and returns **false** otherwise. A null connection indicates that customization will be performed "offline" (without a connection).

— **customize**

```
public boolean customize(sqlj.runtime.profile.Profile profile,
                         java.sql.Connection conn,
                         sqlj.framework.error.ErrorLog log)
```

Customizes the passed profile. If the profile was modified in the process of customization, then **true** is returned. Otherwise, **false** is returned.

See Clause 14, "sqlj.runtime.profile.util.ProfileCustomizer", for further details on these methods and an overview of the class usage.

### 4.3.26 Customization interface

Each profile object contains a number of Customization objects. Each Customization is an implementation-specific object implementing the **sqlj.runtime.profile.Customization** interface that is able to create an **sqlj.runtime.profile.ConnectedProfile** object. Customization objects implement two methods:

1) **acceptsConnection**

```
public boolean acceptsConnection ( java.sql.Connection conn )
```

Returns **true** if this Customization can create a connected profile object for the given `java.sql.Connection` object, and returns **false** otherwise.

2) **sqlj.runtime.profile.getProfile**

```
public sqlj.runtime.profile.ConnectedProfile getProfile (
            java.sql.Connection conn,
            sqlj.runtime.profile.Profile baseProfile )
       throws SQLException
```

Returns a connected profile for the baseProfile on the given `java.sql.Connection` object.

Documentation for this interface is specified in **sqlj.runtime.profile.Customization**.

#### 4.3.26.1 Customization usage

The **getConnectedProfile** method of a profile object is called by the code generated for an <executable clause>. For each <executable clause>, except those containing a <fetch statement>, **getConnectedProfile** is used to obtain a connected profile object. The connected profile object creates a RTStatement object that is used to execute the <executable clause>'s SQL-statement.

The **getConnectedProfile** method is implemented using the customization objects that are currently registered with the profile, as follows.

1) Let **this** represent the profile object on which **getConnectedProfile** is invoked.
2) Let **C** represent the `java.sql.Connection` object passed to the **getConnectedProfile** method.
3) Let **k** represent the number of customization objects currently registered with the profile.
4) Let **i** represent a number ranging from 1 (one) to **k**.
5) For each registered customization object $\mathbf{RC_i}$:
   a) Define that $\mathbf{RC_i}$ accepts **C** if the result of invoking the **acceptsConnection** method on $\mathbf{RC_i}$ passing **C** as an argument returns **true**.
   b)
```
// If RCi accepts C, then
//    return the connected profile for RCi
if RCi.acceptsConnection(C) then
   return RCi.getProfile(C, this);
```



 

6) If no registered Customization object accepts the `java.sql.Connection` object, then return the default ConnectedProfile object.

The default connected profile is implemented using calls to the JDBC API. This means that, by default, SQLJ applications will work with any compliant JDBC driver and therefore do not require a custom runtime implementation on the part of a particular implementation if a JDBC driver exists.

### 4.3.26.2 Customization registration

Customization objects can be registered, deregistered, and enumerated with a profile. The class **sqlj.runtime.profile.Profile** supports the following Customization object-related methods:

— **registerCustomization**

```
public void registerCustomization(Customization customization)
```

Registers a Customization object for this profile object. The Customization object is added after all currently registered Customization objects.

— **registerCustomization**

```
public void registerCustomization
              (Customization newCustomization,
               Customization nextCustomization)
```

Registers a Customization object for this profile object. The new Customization object is added to the list just prior to the next Customization object argument.

— **replaceCustomization**

```
public void replaceCustomization
              (Customization newCustomization,
               Customization oldCustomization)
```

Replaces a Customization object registration for this profile object. The new Customization object is added to the list in place of the old Customization object argument. The new Customization object retains the position of the old Customization object.

— **deregisterCustomization**

```
public void deregisterCustomization(Customization customization)
```

Drop a Customization object from the profile's list.

— **Enumeration**

```
public Enumeration getCustomizations()
```

Returns an enumeration of all Customization objects currently registered with the profile object.

See Clause 13, “Package sqlj.runtime.profile”, for further details on these methods.

Customization objects are serializable. This means that, once registered with a profile object, they are stored and restored with the profile object. Serialization allows the profile objects associated with an SQLJ application to be loaded at any time. Once loaded, any number of customization objects can be registered with the profile

object. The profile object and its registered customization objects can then be reserialized to persistent storage. When the SQLJ application is actually run, all the customization objects that were previously registered with the profile object are also loaded and used to determine what connected profile object should be used to execute the SQL-statements.



 

# 5 Lexical elements

*This Clause modifies Clause 5, "Lexical elements", in ISO/IEC 9075-2.*

## 5.1 <SQL terminal character>

*This Subclause modifies Subclause 5.1, "<SQL terminal character>", in ISO/IEC 9075-2.*

### Function

Define the terminal symbols of the SQL language and the elements of strings.

### Format

```
<SQL special character> ::=
    !! All alternatives from ISO/IEC 9075-2
  | <number sign>

<number sign> ::=
  #
```

### Syntax Rules

1) Insert this SR If <SQL special character> is not contained in an <embedded SQL Java program>, then <SQL special character> shall not immediately contain <number sign>.

2) Insert this SR If the character set SQL_TEXT does not include <number sign>, then <number sign> shall be immediately contained in an <SQL prefix> that is contained in an <embedded SQL Java program>.

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Conformance Rules

*No additional Conformance Rules.*

## 5.2 <token> and <separator>

*This Subclause modifies Subclause 5.2, "<token> and <separator>", in ISO/IEC 9075-2.*

### Function

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

### Format

```
<comment> ::=
    !! All alternatives from ISO/IEC 9075-2
  | <Java comment>

<Java comment> ::=
  <Java comment introducer> [ <comment character>... ] <newline>

<Java comment introducer> ::=
  <solidus> <solidus>
```

### Syntax Rules

1) Insert this SR There shall be no <separator> separating the first <solidus> and second <solidus> of a <Java comment introducer>.

2) Insert this SR If a <comment> is contained in an <embedded SQL Java program>, then

   Case:

   a) If the <comment> is contained in a <statement spec clause> or an <assignment spec clause> immediately contained in an <SQLJ specific clause> and is not contained in an <embedded Java expression>, then it shall be a <simple comment> or a <bracketed comment>.

   b) Otherwise, the <comment> shall be either a <bracketed comment> or a <Java comment>.

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Conformance Rules

*No additional Conformance Rules.*



 

# 6 Scalar expressions

*This Clause modifies Clause 6, "Scalar expressions", in ISO/IEC 9075-2.*

## 6.1 <value specification> and <target specification>

*This Subclause modifies Subclause 6.4, "<value specification> and <target specification>", in ISO/IEC 9075-2.*

### Function

Specify one or more values, host parameters, SQL parameters, dynamic parameters, or host variables.

### Format

```
No additional Format items.
```

### Syntax Rules

1) Insert this SR If <embedded variable specification> is contained in an <embedded SQL Java program>, then <embedded variable specification> shall not immediately contain <indicator variable>.

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Conformance Rules

*No additional Conformance Rules.*

*(Blank page)*

# 7 Additional common elements

*This Clause modifies Clause 10, "Additional common elements", in ISO/IEC 9075-2.*

## 7.1 <routine invocation>

*This Subclause modifies Subclause 10.4, "<routine invocation>", in ISO/IEC 9075-2.*

### Function

Invoke an SQL-invoked routine.

### Format

```
No additional Format items.
```

### Syntax Rules

1) Replace SR 9)h)iii)4) Case:

   a) If $XA_i$ is an <embedded variable name> contained in an <embedded SQL Java program>, then $P_i$ shall be SQLJ output assignable to $XA_i$.

   b) Otherwise, if $XA_i$ is an <embedded variable specification> or a <host parameter specification>, then $P_i$ shall be assignable to $XA_i$, according to the Syntax Rules of Subclause 9.1, "Retrieval assignment", with $XA_i$ as *TARGET* and $P_i$ as *VALUE*.

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Conformance Rules

*No additional Conformance Rules.*

*(Blank page)*

# 8 Embedded SQL

*This Clause modifies Clause 21, "Embedded SQL", in ISO/IEC 9075-2.*

## 8.1 <embedded SQL host program>

*This Subclause modifies Subclause 21.1, "<embedded SQL host program>", in ISO/IEC 9075-2.*

### Function

Specify an <embedded SQL host program>.

### Format

```
<embedded SQL host program> ::=
    !! All alternatives from ISO/IEC 9075-2
  | <embedded SQL Java program>

<statement or declaration> ::=
    !! All alternatives from ISO/IEC 9075-2
  | <SQLJ specific clause>

<SQL prefix> ::=
    !! All alternatives from ISO/IEC 9075-2
  | <number sign>sql !! 'sql' shall be lowercase

<embedded variable name> ::=
    !! All alternatives from ISO/IEC 9075-2
  | <embedded Java expression>
```

NOTE 18 — The <SQL prefix> for Java was chosen to be "#sql" since it is not a valid Java identifier, and as such cannot conflict with other Java syntax.

### Syntax Rules

1) Replace SR 2) Case:

   a) An <embedded SQL statement> or <embedded SQL MUMPS declare> that is contained in an <embedded SQL MUMPS program> shall contain an <SQL prefix> that is "<ampersand>SQL<left paren>". There shall be no <separator> between the <ampersand> and "SQL" nor between "SQL" and the <left paren>.

   b) An <embedded SQL statement> that is contained in an <embedded SQL Java program> shall contain an <SQL prefix> that is "<number sign>sql". There shall be no <separator> between the <number sign> and "sql" and "sql" shall be specified using lowercase letters.

c) An <embedded SQL statement>, <embedded SQL begin declare>, or <embedded SQL end declare> that is not contained in an <embedded SQL MUMPS program> or an <embedded SQL Java program> shall contain an <SQL prefix> that is "EXEC SQL".

2) Insert after SR 3)c) An <embedded SQL statement> contained in an <embedded SQL Java program>, shall contain an <SQL terminator> that is a <semicolon>.

NOTE 19 — Replace Note 748 With the exception of <embedded SQL Java program>, which does not support <embedded SQL declare section>s, there is no restriction on the number of <embedded SQL declare section>s that may be contained in an <embedded SQL host program>.

3) Insert this SR Case:

a) If <statement or declaration> is contained in an <embedded SQL Java program>, then <statement or declaration> shall immediately contain an <SQLJ specific clause>.

b) Otherwise, <SQLJ specific clause> shall not be specified.

4) Insert this SR Case:

a) If <embedded variable name> is contained in an <embedded SQL Java program>, then <embedded variable name> shall immediately contain an <embedded Java expression>.

b) Otherwise, <embedded Java expression> shall not be specified.

5) Replace SR 16) Case:

a) If <embedded Java expression> is contained in an <embedded SQL Java program>, then <expression>s immediately contained in <embedded Java expression> shall conform to scoping rules specified by [JLS].

b) Otherwise, any <host identifier> that is contained in an <embedded SQL statement> in an <embedded SQL host program> shall be defined in exactly one <host variable definition> contained in that <embedded SQL host program>. In programming languages that support <host variable definition>s in subprograms, two <host variable definition>s with different, non-overlapping scope in the host language are to be regarded as defining different host variables, even if they specify the same variable name. That <host variable definition> shall appear in the text of the <embedded SQL host program> prior to any <embedded SQL statement> that references the <host identifier>. The <host variable definition> shall be such that a host language reference to the <host identifier> is valid at every <embedded SQL statement> that contains the <host identifier>.

6) Replace SR 18) Case:

a) Each <expression> immediately contained in an <embedded Java expression> that is contained in an <embedded SQL Java program> has an accessible host language data type provided by the Java language environment. For predefined host language data types, an equivalent SQL <data type> can be found by first looking up the host language data type in Table 2, "SQLJ type properties"; the corresponding `java.sql.Types`-defined constant can then be used with the default mapping `java.sql.Types` to SQL <data type>s, as defined by [JDBC].

b) Otherwise, a <host variable definition> defines the host language data type of the <host identifier> and the equivalent SQL <host parameter data type>, as specified in the operative embedded language Subclause.

7) Replace SR 19) If <embedded SQL host program> does not contain an <embedded SQL Java program>, then <embedded SQL host program> shall contain a <host variable definition> that specifies SQLSTATE as the <host identifier>.



 

8) Replace the first paragraph of SR 21) Given an <embedded SQL host program> *H* that does not contain an <embedded SQL Java program>, there is an implied standard-conforming SQL-client module *M* and an implied host language program *P* derived from *H*. The derivation of the implied program *P* and the implied <SQL-client module definition> *M* of an <embedded SQL host program> *H* effectively precedes the processing of any host language program text manipulation commands such as inclusion or copying of text.

9) Replace the lead text of SR 22) For the <embedded SQL host program> *H* that does not contain an <embedded SQL Java program>, *M* is derived from *H* as follows:

10) Replace the lead text of SR 23) For the <embedded SQL host program> *H* that does not contain an <embedded SQL Java program>, *P* is derived from *H* as follows:

## Access Rules

*No additional Access Rules.*

## General Rules

*No additional General Rules.*

## Conformance Rules

*No additional Conformance Rules.*

## 8.2 <embedded SQL Java program>

### Function

Specify an <embedded SQL Java program>.

### Format

```
<embedded SQL Java program> ::=
  !! See the Syntax Rules.
```

### Syntax Rules

1) An <embedded SQL Java program> is a compilation unit that consists of Java text and SQL text. The Java text shall conform to [JLS]. The SQL text shall consist of one or more <embedded SQL statement>s.

### Access Rules

*None.*

### General Rules

*None.*

### Conformance Rules

1) Without Feature J001, "Embedded Java", conforming SQL language shall not specify an <embedded SQL Java program>.



 

# 9 SQLJ reserved names

This Subclause describes the names reserved for the SQLJ standard runtime and reference implementation components, and the names reserved by the SQLJ translator for naming temporary variables, generated classes, and resource files.

## 9.1 Naming runtime library components

The '`sqlj`' package name and any subpackages thereof (for example, '`sqlj.runtime`') are reserved for the use of the SQLJ standard runtime and reference implementation classes. Runtime library components associated with implementation-specific translator and customizer implementations shall use the Java-specified package naming conventions to avoid conflict with the libraries of the SQLJ reference implementation and other implementations.

The effect of violating SQLJ's reserved package name space is implementation-dependent.

## 9.2 Temporary variable names

An SQLJ translator replaces each occurrence of an <executable clause> with a generated Java statement block, which may contain a number of temporary variable declarations. The name of any such temporary declaration will include the prefix **__sJT_**. The following declarations are examples of those that might occur in an SQLJ-generated statement block.

```
int __sJT_index;
Object __sJT_key;
sqlj.runtime.profile.RTStatement __sJT_stmt;
```

The string "**__sJT_**" is a reserved prefix for SQLJ-generated variable names. The effect of using this string as a prefix for any the following is implementation-dependent:

1) host variable names.
2) Names of variables declared in blocks that include executable SQL-statements.
3) Names of parameters to methods that contain executable SQL-statements.
4) Names of Java fields in classes that contain executable SQL-statements, or the subclasses or enclosed classes of which contain executable SQL-statements.

## 9.3 Class and resource file names

### 9.3.1 Introduction

For each file translated by SQLJ, a number of internal classes and resource files might be generated as part of the SQLJ translation. The name of every such class and resource file has a prefix composed of the name of the original input file followed by the string "**_SJ**".

### 9.3.2 Generated classes

SQLJ internal classes are classes created during SQLJ translation for internal use by generated code. The input to the translation process should not contain references to SQLJ internal classes. SQLJ declared classes are classes created during SQLJ translation that are explicitly named and declared by the SQLJ class declaration constructs <connection declaration clause> and <iterator declaration clause>. The input to the translation process is allowed to contain references to SQLJ declared classes.

All generated classes appear in the same package as is declared in the original input file. Note that declared classes might themselves declare internal classes.

SQLJ internal classes might appear at the end of the translated input file, or might appear in a new Java file created during SQLJ translation. In the case of newly created Java files, the filename is the same as the short name of the generated internal class, and has the **.java** extension.

The effect of declaring a top-level class with a name of the form ***a***_SJ***b*** where ***a*** is the name of an existing class included in the SQLJ application and ***b*** has the form of a Java identifier is implementation-dependent. If the name of a file included with the application has the same format as names of files that might be generated by SQLJ, the effect is implementation-dependent.

### 9.3.3 Resource files and profiles

An SQLJ translator may generate a number of resource files to store information used by SQLJ generated code that is not conveniently represented as a Java class.

Resource files are named using the same rules as defined above for files containing generated internal classes; every resource filename starts with the name of the original input file name followed by the string "**_SJ**". See Subclause 4.3.23, "Profile generation and naming", for further details on names used for SQLJ profiles.

If the name of a file included with the application has the same format as the names that might be generated by the SQLJ implementation as the names of resource files, the effect is implementation-dependent.



 

# 10 Common subelements

## 10.1 <modifiers>

### Function

Represents valid Java class modifiers composed of Java class modifier keywords (*e.g.*, **static**, **public**, **private**, **protected**, *etc*.), as defined by [JLS]. <modifiers> represent one or more Java class modifier keywords (*e.g.*, **static public**).

### Format

```
<modifiers> ::=
  !! See the Syntax Rules
```

### Syntax Rules

1) <modifiers> specifies one or more Java class modifier keywords as defined by [JLS].

### Access Rules

*None.*

### General Rules

*None.*

### Conformance Rules

*None.*

## 10.2 <java class name>

### Function

Identify a valid Java class name as defined by [JLS].

### Format

```
<java class name> ::=
  !! See the Syntax Rules
```

### Syntax Rules

1) <java class name> specifies a valid Java class name, as defined by [JLS].

### Access Rules

*None.*

### General Rules

*None.*

### Conformance Rules

*None.*



 

## 10.3 <java id>

### Function

Identify a valid Java variable as defined by [JLS].

### Format

```
<java id> ::=
  !! See the Syntax Rules
```

### Syntax Rules

1) <java id> specifies a valid Java variable, as defined by [JLS].

### Access Rules

*None.*

### General Rules

*None.*

### Conformance Rules

*None.*

## 10.4 <java datatype>

### Function

Identify a valid Java data type as defined by [JLS].

### Format

```
<java datatype> ::=
  !! See the Syntax Rules
```

### Syntax Rules

1) <java datatype> specifies a valid Java data type, as defined by [JLS].

### Access Rules

*None.*

### General Rules

*None.*

### Conformance Rules

1) Without Feature J008, "Datalinks via SQL language", conforming SQL language shall not contain a <java datatype> that specifies `java.net.URL`.

2) Without Feature J010, "XML via SQL language", conforming SQL language shall not contain a <java datatype> that specifies `java.sql.SQLXML`.



 

## 10.5 <java constant expression>

### Function

Identify a valid Java constant expression as defined by [JLS].

### Format

```
<java constant expression> ::=
  !! See the Syntax Rules
```

### Syntax Rules

1) <java constant expression> specifies a valid Java constant expression, as defined by [JLS].

### Access Rules

*None.*

### General Rules

*None.*

### Conformance Rules

*None.*

## 10.6 <embedded Java expression>

### Function

Identifies a Java variable or a Java expression that resolves to a Java variable.

### Format

```
<embedded Java expression> ::=
  <colon> [ <parameter mode> ] <expression>

<expression> ::=
    <simple variable>
  | <left paren> <complex expression> <right paren>

<simple variable> ::=
  !! See the Syntax Rules

<complex expression> ::=
    <Rval expression>
  | <Lval expression>

<Rval expression> ::=
  !! See the Syntax Rules

<Lval expression> ::=
  !! See the Syntax Rules
```

### Syntax Rules

1) <simple variable> shall conform to the Java rules for simple name specified by [JLS] in section 6.2.

2) <Rval expression> shall conform to the Java rules for AssignmentExpression specified by [JLS] in section 15.26.

3) <Lval expression> shall conform to the Java rules for LeftHandSide specified by [JLS] in section 15.26.

4) Case:

   a) If <embedded Java expression> is contained in an argument for a parameter of the subject routine of a <call statement> whose <parameter mode> is IN, then

      Case:

      i) If <parameter mode> is specified, then <parameter mode> shall be IN.

      ii) Otherwise, a <parameter mode> of IN is implicit.

   b) If <embedded Java expression> is contained in an argument for a parameter of the subject routine of a <call statement> whose <parameter mode> is OUT, then a <parameter mode> of OUT shall be specified.

   c) If <embedded Java expression> is contained in an argument for a parameter of the subject routine of a <call statement> whose <parameter mode> is INOUT, then a <parameter mode> of INOUT shall be specified.



 

d) If <embedded Java expression> is contained in a <value specification> and is not contained in an argument for a parameter of the subject routine of a <call statement>, then

Case:

i) If <parameter mode> is specified, then <parameter mode> shall be IN.

ii) Otherwise, a <parameter mode> of IN is implicit.

e) If <embedded Java expression> is contained in an <assignment target> and is not contained in an argument for a parameter of the subject routine of a <call statement>, then

Case:

i) If <parameter mode> is specified, then <parameter mode> shall be OUT.

ii) Otherwise, a <parameter mode> of OUT is implicit.

## Access Rules

*None.*

## General Rules

*None.*

## Definitions and Rules

1) A <complex expression> is a proper superset of <simple variable>.

2) An <Rval expression> is a proper superset of <Lval expression>.

3) During execution of an SQLJ program, an <Rval expression> is evaluated to determine its value, according to the rules of Java expression evaluation. The determination of an <Rval expression>'s value results in all side effects of the <Rval expression> evaluation becoming visible.

4) During execution of an SQLJ program, an <Lval expression> is evaluated to determine a pair comprising both the value and the location of the <Lval expression>. The determination of both of these properties results in all side effects of the <Lval expression> evaluation becoming visible.

a) During execution of an SQLJ program, the value of <Lval expression> is determined according to the rules of Java expression evaluation.

b) At the same time, during execution of an SQLJ program, the location of <Lval expression> is determined as follows:

i) If <Lval expression> is a simple Java identifier, denoting a Java variable **X**, then the location of <Lval expression> is the location of **X**.

ii) If <Lval expression> references a Java field called **F** of a Java <Rval expression> denoting a Java object **O**, then the location of <Lval expression> is the location of:

**O**.**F**

iii) If <Lval expression> references an element of a Java <Rval expression> array **A** with index <Rval expression> **I**, then the location of <Lval expression> is the location of the array element:

**A**[**I**]

## Conformance Rules

1) Without Feature J008, "Datalinks via SQL language", conforming SQL language shall not contain an <embedded Java expression> whose Java type is `java.net.URL`.

2) Without Feature J010, "XML via SQL language", conforming SQL language shall not contain a <embedded Java expression> whose Java type is `java.sql.SQLXML`.



 

## 10.7 <implements clause>

### Function

Specifies a set of one or more interface classes to a generated class declaration.

### Format

```
<implements clause> ::=
  implements [ <interface list> ]

<interface list> ::=
  <interface element> [ { <comma> <interface element> }... ]

<interface element> ::=
    <predefined interface class>
  | <user defined interface class>

<user defined interface class> ::=
  <java class name>

<predefined interface class> ::=
    sqlj.runtime.ForUpdate
  | sqlj.runtime.Scrollable
```

### Syntax Rules

1) <user defined interface class> shall specify a user-defined interface class.

### Access Rules

*None.*

### General Rules

*None.*

### Definitions and Rules

1) The <interface list> is appended to the generated class definition by text substitution.

### Conformance Rules

*None.*

## 10.8 <declaration with clause>

### Function

Specifies a set of one or more initialized variables to a generated class declaration.

### Format

```
<declaration with clause> ::=
  with <left paren> <declaration with list> <right paren>

<declaration with list> ::=
  <with element> [ { <comma> <with element> }... ]

<with element> ::=
  <with keyword> <equals operator> <with value>

<with keyword> ::=
    <predefined iterator with keyword> <predefined connection with keyword>
  | <user defined with keyword>

<predefined iterator with keyword> ::=
    sensitivity
  | holdability
  | updateColumns

<predefined connection with keyword> ::=
    dataSource
  | typeMap
  | path
  | transformGroup

<user defined with keyword> ::=
  <java id>

<with value> ::=
  <java constant expression>
```

### Syntax Rules

1) No <with keyword> shall be specified more than once.

### Access Rules

*None.*

### General Rules

*None.*



 

## Definitions and Rules

1) Each <with element> is added as a public static final variable the type of which is derived from the type of its associated <java constant expression> to the generated class declaration.

2) Support for each <predefined iterator with keyword> is implementation-defined.

3) If the <predefined iterator with keyword> is **sensitivity**, then the <with value> shall be one of the **sqlj.runtime.ResultSetIterator** defined int constants `SENSITIVE`, `INSENSITIVE`, or `ASENSITIVE`.

   a) The keyword **sensitivity** specifies the semantics defined for <cursor sensitivity> in Subclause 14.1, "<declare cursor>", .

   b) The *effective sensitivity* of an iterator refers to the runtime value that would be returned by an invocation of `getSensitivity()` on that iterator. JDBC mandates that an implementation that cannot support the requested sensitivity enforce the closest matching sensitivity supported to that requested.

   c) The *default sensitivity* of an iterator refers to the compile time sensitivity value when the keyword **sensitivity** is not specified. The default sensitivity shall be **ASENSITIVE**, as specified in Subclause 14.1, "<declare cursor>", in .

4) If the <predefined iterator with keyword> is **holdability**, then the <with value> shall be the boolean value *true* or *false*.

   a) The keyword **holdability** specifies the semantics defined for <cursor holdability> in Subclause 14.1, "<declare cursor>", .

   b) The *default holdability* of an iterator when the keyword **holdability** is not specified shall be **WITHOUT HOLD**, as specified in Subclause 14.1, "<declare cursor>", in .

5) If the <predefined iterator with keyword> is **updateColumns**, then the <with value> shall be a String literal containing a comma-separated list of column names.

6) If an <iterator declaration clause> contains a <declaration with clause> that contains a <predefined iterator with keyword> of **updateColumns**, then the <iterator declaration clause> shall also contain an <implements clause> specifying a <predefined interface class> that contains **sqlj.runtime.ForUpdate**.

7) If the <predefined connection with keyword> is **dataSource**, then the <with value> shall be a String literal naming a JNDI resource of type *`javax.sql.DataSource`*, as specified by [JDBC].

8) If the <predefined connection with keyword> is **typeMap**, then the <with value> shall adhere to the following rules:

   a) The <with value> shall be a String literal containing one name or a comma-separated list of multiple names of Java resource bundle(s). The name of a resource bundle shall adhere to the required syntax for resource bundle family names as specified in [JLS], and shall refer to a Java properties class or file that contains type mapping information.

   b) A property definition contained in the class or file that is recognized by the SQLJ translator as defining a type mapping shall be specified in the following way:

      i) The name of the property has the following syntax:

```
<type map property name> ::=
  class. <java class name>
```

where <java class name> is a full class name that includes a package name.

ii) The value of the property has the following syntax:

```
<type map property name> ::=
  class. <java class name>

<type map property value> ::=
  [ <SQL type> ] <user-defined type name> [ TRANSFORM <group name> ]

<SQL type> ::=
    DISTINCT
  | STRUCT
  | JAVA_OBJECT
```

c) Each <type map property name> shall be unique across all type maps specified in the <with value> of a single <connection declaration clause>.

d) A <user-defined type name> that is contained in a <type map property value> shall not be contained in any other <type map property value> of the same type map or any other type map specified in the <with value> of the same <connection declaration clause>.

e) The class <java class name> has to specify that it implements either java.sql.SQLData or java.io.Serializable.

f) If <type map property value> *TMPV* contains a transform <group name>, then let *TG* be that <group name> and let *UDT* be the <user-defined type name> contained in *TMPV*. The <group specification> "*TG* FOR TYPE *UDT*" is called a **property group specification** of the type map properties file.

9) If the <predefined connection with keyword> is **path**, then the <with value> shall be a <schema name list>, and an <embedded path specification> of the form "`PATH <with value>`" is implicitly specified and precedes any <SQLJ specific clause> executed in the scope of the connection context class.

10) If the <predefined connection with keyword> is **transformGroup**, then the <with value> shall be of the form "{ <single group specification> | <multiple group specification> }". If <single group specification> is specified, then no property class or file contained in the <with value> of a **typeMap** shall contain a property group specification. An <embedded transform group specification> of the form "`TRANSFORM GROUP <with value>`" is implicitly specified and precedes any <SQLJ specific clause> executed in the scope of the connection context class.

11) If <multiple group specification> is specified, then no <user-defined type name> contained in the <multiple group specification> shall also be part of a property group specification that is contained in a property class or file specified in the <with value> of a **typeMap**.

12) If no <single group specification> is specified, then let *MGU* be the comma-separated list of all <group specification>s contained in a <multiple group specification> and all the property group specifications contained in the property classes or files specified as part of the <declaration with clause> of the <connection declaration clause>. If *MGU* is not empty, then an <embedded transform group specification> of the form "`TRANSFORM GROUP` *MGU*" is implicitly specified and precedes any <SQLJ specific clause> executed in the scope of the connection context class.

## Conformance Rules

1) Without Feature S071, "SQL paths in function and type name resolution", conforming SQL language shall not contain a <predefined connection with keyword> that simply contains **path**.



 

2) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a <predefined connection with keyword> that is **transformGroup**.

3) Without Feature S241, “Transform functions”, conforming SQL language shall not contain a user-defined type map specified using a <predefined connection with keyword> that simply contains **typeMap** and that contains a property group specification.

*(Blank page)*

# 11 <SQLJ specific clause> and contents

## 11.1 <SQLJ specific clause>

### Function

Specify an embedded SQLJ clause inside a Java application.

### Format

```
<SQLJ specific clause> ::=
    <connection declaration clause>
  | <iterator declaration clause>
  | <executable clause>
```

### Syntax Rules

*None.*

### Access Rules

*None.*

### General Rules

*None.*

### Conformance Rules

*None.*

## 11.2 <connection declaration clause>

### Function

Specify a named connection context declaration inside a Java application.

### Format

```
<connection declaration clause> ::=
  [ <modifiers> ]context <java class name>
      [ <implements clause> ] [ <declaration with clause> ]
```

### Syntax Rules

1) An <implements clause> shall not specify a <predefined interface class>.

2) A <declaration with clause> shall not specify a <predefined iterator with keyword>.

### Access Rules

*None.*

### General Rules

*None.*

### Definitions and Rules

1) SQLJ connection contexts are objects of classes that are defined by means of the <connection declaration clause> and result in the generation of a generated connection class declaration.

2) A <connection declaration clause> is permitted to appear anywhere a Java class definition may appear.

### Conformance Rules

*None.*



 

## 11.3 Generated connection class

### Function

To define the signature (*i.e.*, associated methods) of a generated connection class.

### Signature

In the following signature, let *withType* represent the <java datatype> of its associated <with keyword>.

```
<modifiers> class <java class name>
                implements sqlj.runtime.ConnectionContext
                [ , <interface list> ]      // Optional; not literal []
    {
      // Optional and repeatable; not literal [] or {} or ...
      [ { static public final withtype
              <with keyword> = <with value> ; }... ]
      <create connection constructors>

      public <java class name>
                ( ConnectionContext other )
                throws SQLException ;
      public <java class name>
                ( java.sql.Connection conn )
                throws SQLException ;
      static public <java class name>
                getDefaultContext ( ) ;
      static public void setDefaultContext
                ( <java class name> ctx ) ;
      public java.util.Map
                getTypeMap ( ) ;
    }
```

A generated connection class implements interface **sqlj.runtime.ConnectionContext**.

```
<create connection constructors> ::=
    <data source constructors>
  | <url constructors>

<data source constructors> ::=
    public <java class name> ()
        throws SQLException ;
  | public <java class name> ( String user, String password )
      throws SQLException ;

<url constructors> ::=
    public <java class name> ( String url, Properties info, boolean autoCommit )
        throws SQLException ;
  | public <java class name> ( String url, boolean autoCommit )
      throws SQLException ;
  | public <java class name> ( String url, String user, String password, boolean autoCommit
  )
      throws SQLException ;
```

## Definitions and Rules

1) A generated connection class is generated using the specified <modifiers> and <java class name> as a side effect of the direct inclusion of a <connection declaration clause>.

2) If the <connection declaration clause> contains a <declaration with clause> that specifies the <predefined connection with keyword> **dataSource**, then the generated connection class signature uses <data source constructors>; otherwise, the generated connection class signature uses <url constructors>.

3) If the <connection declaration clause> contains a <declaration with clause> that specifies the <predefined connection with keyword> **typeMap**, then let **TM** be the corresponding <with value>. The invocation of the method **getTypeMap ( )** of the generated connection class returns an instance of a class that implements **java.util.Map** that contains the user-defined type mapping information provided by the properties files listed in **TM** in the form specified in [JDBC]. This method is invoked by code generated by the SQLJ translator for <executable clause>s and <iterator declaration clause>s, but it can also be invoked to create a `java.util.Map` object that may be passed to methods of a `java.sql.Statement` object. The implementation of this method attempts to load the properties files based on the Java class path. If the <connection declaration clause> does not contain a <declaration with clause> that specifies the <predefined connection with keyword> **typeMap**, then this method returns Java null.

4) At runtime, a connection context object and its underlying `java.sql.Connection` object have an associated connection context user identifier, which is by default used as the current user identifier for all SQL-statements executed in the scope of the connection context object, and is defined as follows.

   Case:

   a) If the connection context object is created using <data source constructors> or <url constructors> that have a `user` parameter, or if a user name is provided as part of the `info` parameter, then the connection context user identifier is the user name provided.

   b) If the connection context object is created using the constructor that takes an existing connection context object, then the connection context user identifier is the user identifier of the existing connection context object.

   c) If the connection context object is created using the constructor that takes an existing `java.sql.Connection` object, then the connection context user identifier is the user name that was provided during creation of the `java.sql.Connection` object.

   d) Otherwise, the connection context user identifier is implementation-defined.

5) The semantics of constructors defined by <data source constructors> are as described for overloadings of method **getConnection** of class *`javax.sql.DataSource`* in [JDBC]. If one of these constructors is called, JNDI is used to obtain the data source object named by the <with value> of the **dataSource** <with keyword>. The data source object is used to create the connection. The auto commit mode is set as specified by the given data source.

6) The semantics of constructors defined by <url constructors> are as described for overloadings of method **`getConnection`** of class **`java.sql.DriverManager`** in [JDBC]. The connection is created with an auto commit mode set as specified by the value of the “autoCommit” argument.

7) The constructor that takes an existing connection context object as its argument causes the object on which the method is invoked to share its SQL-session, *i.e.*, its underlying SQL-connection. The auto commit mode is that of the passed connection context object.



 

8) The constructor that takes an existing `java.sql.Connection` object as its argument causes the object on which the method was invoked to share its SQL-session, *i.e.*, its underlying SQL-connection. The auto commit mode is that of the passed connection object.

9) Method **setDefaultContext** sets the default connection context for this class.

10) Case:

    a) If **setDefaultContext** has been called, then **getDefaultContext** returns the default connection context object for this class.

    b) If a data source is defined in JNDI for the name **`jdbc/defaultDataSource`**, then **getDefaultContext** returns a connection context object that uses the connection created by this data source.

    c) If a default SQL-connection exists in the runtime environment, then **getDefaultContext** returns a connection context object that shares the underlying default SQL-connection.

    d) Otherwise, **getDefaultContext** returns null.

11) All other methods are defined in **sqlj.runtime.ConnectionContext**.

## Binary Composition

The following rules are defined for binary composition in every generated connection class.

1) The generated class includes a **static public** method named **getProfileKey**:

```
public static Object getProfileKey
                      (sqlj.runtime.profile.Loader loader,
                       String profileName) throws SQLException;
```

`getProfileKey()` returns a key associated with the profile having the given profileName. If the key for a profile with this name already exists, then it is returned; otherwise, a new profile is instantiated with the given name and Loader and a new key for this profile is returned. An exception is thrown if a profile cannot be loaded with the given name and Loader.

The object returned is an opaque, implementation-defined key for use in a subsequent call to `getProfile()` or `getConnectedProfile()`. This method is used by translator-generated code that replaces <executable clause>s to obtain a key with which a particular profile can be identified within a connection context.

— **Parameters**:

- **loader** — The profile loader with which the profile should be loaded if it doesn't already exist.
- **profileName** — The fully qualified name of the profile.

— **Returns**:

- A key for the profile with the given name in this context.

— **Throws**:

- SQLException — If a profile with this name cannot be loaded.

2) The generated class includes a **static public** method named **getProfile**:

```
public static sqlj.runtime.profile.Profile getProfile (Object profileKey);
```

**getProfile** returns a top-level profile associated with profile key returned by an earlier call to **getProfileKey** in this context class. Each connection context class maintains a static set of profiles that collectively define all possible SQL-statements that are permitted to be performed on this context.

— **Parameters**:

- **profileKey** — the key associated with the desired profile.

— **Throws**:

- IllegalArgumentException — If the profileKey is null or invalid.

3) The generated class includes a **public** method named **getConnectedProfile**:

```
public sqlj.runtime.profile.ConnectedProfile
       getConnectedProfile(Object profileKey)
       throws SQLException;
```

**getConnectedProfile** returns the connected profile associated with a profileKey for this connection context object. Each connection context object maintains a set of connected profiles on which SQL-statements are prepared. Collectively, the set of connected profiles contained in a connection context represent the set of all possible SQL-statements that are permitted to be performed between the time the connection context object is created and the time it is destroyed.

The **profileKey** object shall be an object that was returned via a prior call to `getProfileKey()`. An exception is thrown if a connected profile object could not be created for this connection context.

For each <executable clause>, except those containing a <fetch statement>, **getConnectedProfile** is used by translator-generated code that replaces <executable clause>s to obtain a connected profile. The connected profile in turn creates a RTStatement object that is used to execute the <executable clause>'s SQL-statement.

— **Parameters**:

- **profileKey** — The key associated with the desired profile.

— **Throws**:

- IllegalArgumentException — If the profileKey is null or invalid.
- SQLException — If the connected profile object could not be created.

## Code Generation

In addition to managing the SQL-connection, the connection context class implementation is responsible for instantiating profile and connected profile objects at runtime, as follows:

1) Let **PL** represent a profile loader.

2) Let **PN** represent a profile name.

3) If **getProfileKey** is called with values **PL** and **PN**, and a key for **PN** does not exist, then:



 

a) A new profile is instantiated using the **instantiate** method:

```
sqlj.runtime.profile.Profile p =
    sqlj.runtime.profile.Profile.instantiate ( PL, PN ) ;
```

b) An implementation-dependent key is created and returned that is associated with **PN**, and that if referenced by members of this generated connection class uniquely identifies the newly-instantiated profile.

4) Let **PK** represent a profile key returned by a call to **getProfileKey**.

5) If **getProfile** is called with argument **PK**, then return the profile instantiated during the call to **getProfileKey** associated with **PK**.

6) If **getConnectedProfile** is called with argument **PK**, then:

a) Let **C** represent the underlying `java.sql.Connection` object associated with the current connection context object.

b) Let **P** represent the Profile associated with **PK**. This can be found using the static **getProfile** method.

c) Let **CP** represent the ConnectedProfile associated with **PK** in the current connection context object, or null if none exists.

d) If **CP** is Java null, then a new connected profile is created using the **getConnectedProfile** method of **P**:

```
CP = P.getConnectedProfile ( C ) ;
```

e) Return **CP**

# 11.4 <iterator declaration clause>

## Function

Specify either a positioned iterator class declaration or a named iterator class declaration inside a Java application. An *iterator* is an object that contains the result of the evaluation of a query. Iterators are objects that implement the interface **sqlj.runtime.ResultSetIterator**, and are declared by an SQLJ translator in response to an <iterator declaration clause>.

The SQLJ clause for declaring an iterator class has two forms, distinguishing a <named iterator> from a <positioned iterator>.

## Format

```
<iterator declaration clause> ::=
  [ <modifiers> ]iterator <java class name>
      [ <implements clause> ] [ <declaration with clause> ]
      <left paren> <iterator spec declaration> <right paren>

<iterator spec declaration> ::=
    <positioned iterator>
  | <named iterator>
```

## Syntax Rules

1) A <declaration with clause> shall not specify a <predefined iterator with keyword>.

## Access Rules

*None.*

## General Rules

*None.*

## Definitions and Rules

1) The two kinds of iterators, positional and named, are distinct and incompatible Java types implemented with different interfaces.

2) The two kinds of iterators, positional and named, cannot be used interchangeably. Separate class (interface) hierarchies for named and positional iterators enforce this restriction.

3) A <iterator declaration clause> is permitted to appear anywhere a Java class definition is permitted to appear.

4) Without Feature J002, “ResultSetIterator access to JDBC ResultSet”, if an implementation of either the `sqlj.runtime.ResultSetIterator` interface's public method `getResultSet()` or the `sqlj.run-`



 

`time.profile.RTResultSet` interface's public method `getJDBCResultSet()` is invoked, then an SQLexception condition shall be thrown: *OLB-specific error — unsupported feature*.

## Signature

From an <iterator declaration clause>, an SQLJ translator generates an iterator class. All iterator classes implement interface **`ResultSetIterator`**. The **`ResultSetIterator`** interface includes the public method `getResultSet()`, which, using the public method `getJDBCResultSet()` of `sqlj.runtime.profile.RTResultSet`, returns the `java.sql.ResultSet` object associated with this iterator.

If the <iterator declaration clause> contains an <implements clause> with the **Scrollable** interface, the iterator class implements the interface **Scrollable**.

## Conformance Rules

1) Without Feature J002, "ResultSetIterator access to JDBC ResultSet", conforming SQL language shall not contain an invocation of the `sqlj.runtime.ResultSetIterator` interface's public method `getResultSet()` or the `sqlj.runtime.profile.RTResultSet` interface's public method `getJDBCResultSet()`.

## 11.5 <positioned iterator>

### Function

Specify a *positioned iterator* inside a Java application.

### Format

```
<positioned iterator> ::=
  <java type list>

<java type list> ::=
  <java datatype> [ { <comma> <java datatype> }... ]
```

### Syntax Rules

*None.*

### Access Rules

*None.*

### General Rules

*None.*

### Conformance Rules

*None.*



 

## 11.6 Generated positioned iterator class

### Function

To define the signature (*i.e.*, associated methods) of a generated positioned iterator class.

### Signature

In a <positioned iterator>, no names are provided for the columns in the iterator class declaration, and an SQLJ translator will generate code for positional access to the columns of SQL queries associated with iterators of type <java class name>.

In the following signature, let *withType* represent the <java datatype> of its associated <with keyword>.

```
<modifiers> class <java class name>
            implements sqlj.runtime.PositionedIterator
                       // Optional; not literal []
                       [ , <interface list> ]
    {
        // Optional and repeatable; not literal [], {}, or ...
        [ { static public final withtype
                       <with keyword> =
                         <with value> ; }... ]
        // Methods are defined in sqlj.runtime.PositionedIterator
    }
```

For a <positioned iterator>, an SQLJ translator will generate an iterator class implementing interface **sqlj.runtime.PositionedIterator**.

### Definitions and Rules

1) A generated positioned iterator class is generated using the specified <modifiers>, <java class name>, <interface list>, and <declaration with clause> of its containing <iterator declaration clause>.

### Binary Composition

The following rules are defined for binary composition in every generated positioned iterator class.

1) The generated class includes a **public** constructor that has an RTResultSet parameter, and might throw an SQLException:

   ```
   public <java class name>
              ( sqlj.runtime.profile.RTResultSet rs )
                throws java.sql.SQLException ;
   ```

   This constructor is used by translator-generated code replacing any <assignment clause> that populates a <positioned iterator> result.

2) The generated class includes a **public** method named `next()` the semantics of which are the same as those defined for the `next()` method of the <named iterator> class:

```
public next ( )
    throws SQLException ;
```

This method is used by translator-generated code that replaces the <fetch statement> to advance the iterator to the next row.

3) Let **k** represent the cardinality of the <java type list>.

4) Let **i** represent a variable ranging from 1 (one) to **k**.

5) For each <java datatype> **JT** in <java type list>, the generated class includes a **public** method named **getCol** the return type of which is $\mathbf{JT_i}$:

```
public JTi getColi ( )
    throws SQLException ;
```

This method is used by translator-generated code that replaces the <fetch statement> to fetch the data corresponding the **i**-th column of the current iterator row.

## Code Generation

As described in the binary composition section, a <positioned iterator> object is constructed using an instance of class **sqlj.runtime.profile.RTResultSet**. The iterator class implementation shall use the passed RTResultSet to fetch data from the implicit cursor, as follows:

1) Let **RT** represent the **sqlj.runtime.profile.RTResultSet** object passed during construction of this iterator.

2) Let **k** be the cardinality of the <positioned iterator>.

3) Let **m** represent the number of columns in **RT**.

4) If **m ≠ k**, then an SQLException is thrown by the iterator constructor.

5) Let **i** represent a variable ranging from 1 (one) to **k**.

6) For each <java datatype> **JT** in <java type list>:

   a) Let **GM** represent the getter method corresponding to $\mathbf{JT_i}$, as given in Table 2, "SQLJ type properties".

   b) If **GM** is getObject, then the implementation of *getColi* returns the result of calling the getObject method on **RT** using the compile-time class of $\mathbf{JT_i}$:

```
JTi getColi ( ) throws SQLException
    {
        return RT.getObject ( i, JTi.class ) ;
    }
```

   c) If **GM** is not getObject, then the result of *getColi* returns the result of calling the **GM** method on **RT**:

```
JTi getColi ( ) throws SQLException
    {
```



 

```
        return RT.GM ( i ) ;
    }
```

NOTE 20 — The above requirements define that a method call to the underlying RTResultSet is made each time **getCol*i*** is called. By uniformly defining when the call is made, the underlying RTResultSet implementations are able to reliably implement optimizations such as preparing all results during the `next()` call or caching column results.

## 11.7 <named iterator>

### Function

Specify a *named iterator* inside a Java application.

### Format

```
<named iterator> ::=
  <java pair list>

<java pair list> ::=
  <java pair> [ { <comma> <java pair> }... ]

<java pair> ::=
  <java datatype> <java id>
```

### Syntax Rules

1) No <java id> contained in a <java pair list> shall be equivalent to any other <java id> in that <java pair list> (using a case-sensitive comparison).

### Access Rules

*None.*

### General Rules

*None.*

### Definitions and Rules

1) The <fetch statement> shall not be used in conjunction with <named iterator>.
2) An accessor method will be generated for each <java id>, with the following specifications:
   a) One accessor method will be generated for each specified <java id>.
   b) Each accessor method will have as its name the corresponding <java id> and be of the exact case as that specified by <java id>.
   c) Each accessor method will be of the form <java id>(), returning the corresponding Java type <java datatype> and throwing type SQLException.
   d) If<java datatype> is a Java primitive datatype and the column value is an SQL null value, then the accessor method will raise an exception of type sqlj.runtime.SQLNullException.



 

## Conformance Rules

*None.*

## 11.8 Generated named iterator class

### Function

To define the signature (*i.e.*, associated methods) of a generated named iterator class.

### Signature

For a <named iterator>, the SQLJ translator will generate accessors for each column in the <java pair list> in order to provide named access to the columns of SQL queries associated with an iterator of type <java class name>. In the following signature, let *withType* represent the <java datatype> of its associated <with keyword>.

```
<modifiers> class <java class name>
      implements sqlj.runtime.NamedIterator
                 [, <interface list> ]  // Optional, not literal []
  {
    // Optional and repeatable; not literal [], {}, or ...
    [ { static public final withtype
                <with keyword> = <with value> ; }... ]
    // Repeatable; not literal {} or ...
    { public <java datatype> <java id> ()
                              throws SQLException ; }...
    // All other Methods are defined in sqlj.runtime.NamedIterator
  }
```

For a <named iterator>, an SQLJ translator will generate a class implementing interface `NamedIterator`.

### Definitions and Rules

1) A generated named iterator class is generated using the specified <modifiers>, <java class name>, <interface list>, and <declaration with clause> of its containing <iterator declaration clause>.

### Binary Composition

The following rules are defined for binary composition in every generated named iterator class.

1) The generated class includes a **public** constructor that has an RTResultSet parameter, and might throw an SQLException.

   This constructor is used by translator-generated code replacing any <assignment clause> that populates a <named iterator> result.

### Code Generation

As described in the binary composition section, a <named iterator> object is constructed using an instance of class **sqlj.runtime.profile.RTResultSet**. The iterator class implementation shall use the passed RTResultSet to fetch data from the implicit cursor, as follows.

1) Let **RT** represent the **sqlj.runtime.profile.RTResultSet** object passed during construction of this iterator.



 

2) Let **k** be the cardinality of the <named iterator>.

3) Let **m** represent the number of columns in **RT**:

```
m = RT.getColumnCount ( ) ;
```

4) If **m** < **k**, then an SQLException is thrown by the iterator constructor.

5) Let **i** represent a variable ranging from 1 (one) to **k**.

6) For each <java pair> **JP** in <java pair list>:

   a) Let **JT** represent the <java datatype> of $\mathbf{JP_i}$.

   b) Let **JI** represent the <java id> of $\mathbf{JP_i}$.

   c) Let **n** represent the index of **JI** in **RT**, as defined by **findColumn**:

```
n = RT.findColumn ( JI ) ;
```

      NOTE 21 — Because `findColumn()`, due to its basis in JDBC, uses case-insensitive comparison, column names in the <query clause>'s select list that differ only in the case of one or more characters should use the SQL AS clause to avoid ambiguity, even if one or both of those column names are specified using <delimited identifier>s.

   d) Let **GM** represent the getter method corresponding to **JT**, as given in Table 2, "SQLJ type properties".

   e) If **GM** is getObject, then the implementation of the accessor method for $\mathbf{JP_i}$ returns the result of calling getObject on **RT**, using the compile-time class of **RT**:

```
JT JI ( ) throws SQLException
    {
        return RT.getObject ( n, JT.class ) ;
    }
```

   f) If **GM** is not getObject, then the implementation of the accessor method for $\mathbf{JP_i}$ returns the result of calling the **GM** method on **RT**:

```
JT JI ( ) throws SQLException
    {
        return RT.GM ( n ) ;
    }
```

NOTE 22 — The above requirements define that a method call to the underlying RTResultSet is made each time an accessor method is called. By uniformly defining when the call is made, the underlying RTResultSet implementations are able to reliably implement optimizations such as preparing all results during the `next()` call or caching column results.

NOTE 23 — It is not required that the **findColumn** method is called each time an accessor method is called since the result of **findColumn** is invariant for a particular column name in a particular RTResultSet object. Accordingly, **findColumn** need only be called once for each column of each iterator constructed.

## 11.9 <executable clause>

### Function

Specify the execution of an SQL-statement.

### Format

```
<executable clause> ::=
  [ <context clause> ] <executable spec clause>

<executable spec clause> ::=
    <statement clause>
  | <assignment clause>
```

### Syntax Rules

*None.*

### Access Rules

*None.*

### General Rules

*None.*

### Definitions and Rules

1) An <executable clause> is permitted to appear anywhere a Java statement is permitted to appear.

2) An <executable clause> might throw exception **`java.sql.SQLException`**.

3) All runtime exceptions raised during the execution of an <executable spec clause> will be caught as an SQLException as defined by [JDBC].

4) If a runtime exception is raised during the execution of an <executable clause>, then the values of any OUT or INOUT <embedded Java expression> is implementation-dependent.

5) Without Feature J003, "Execution control", if an implementation of the `sqlj.runtime.ExecutionContext` class's public methods **setMaxFieldSize(int)**, **setMaxRows(int)**, or **setQueryTimeout(int)** is invoked to set the corresponding `ExecutionContext` Java field to anything other than its default value, and an attempt is made to register a statement with such an `ExecutionContext` (which, as specified under Code Generation in this Subclause, invokes the `sqlj.runtime.profile.RTStatement` interface's methods of the same name), then an SQLException condition is thrown: *OLB-specific error — unsupported feature*.

6) Without Feature J004, "Batch update", if an implementation of the `sqlj.runtime.ExecutionContext` class's public methods `executeBatch()`, `getBatchLimit()`, `getBatchUpdateCounts()`,



 

`isBatching()`, `setBatching(boolean)`, or `setBatchLimit(int)` is invoked, then the result is implementation-defined.

7) Without Feature J009, "Multiple Open ResultSets", if an implementation of the `sqlj.runtime.ExecutionContext` class's public method **getNextResultSet(int)** is invoked with any value other than `java.sql.Statement.CLOSE_CURRENT_RESULT`, then an SQLException condition is thrown: *OLB-specific error — unsupported feature*.

8) Let **n** represent the number of <embedded Java expression>s appearing in the <executable clause>.

9) Let $\mathbf{BP_i}$, 1 (one) $\leq \mathbf{i} \leq \mathbf{n}$, represent the bind parameters in the SQL-statement corresponding to <executable clause>.

   a) Every bind parameter $\mathbf{BP_i}$ can either set an input value, or return an output value, or both.

   b) If <executable clause> is an <assignment clause>, then let $\mathbf{BP_0}$ represent the bind parameter than can return the output value of the SQL-statement.

10) The semantics of executing <executable spec clause> with <embedded Java expression>s $\mathbf{HE_i}$, with parameter mode $\mathbf{P_i}$, 1 (one) $\leq \mathbf{i} \leq \mathbf{n}$, are as follows.

   a) If <executable clause> contains an implicit or explicit <context clause>, then:

      i) Let **DC** represent the implementation-defined class name of the default connection context.

      ii) If <context clause> contains a <connection context> then set **CCtx** to the value specified by <connection context>; otherwise, set **CCtx** to the value of **DC**`.getDefaultContext()`.

      iii) If <context clause> contains an <execution context> then set **ECtx** to the value specified by <execution context>; otherwise, set **ECtx** to the value of **CCtx**`.getExecutionContext()`.

   b) If <executable clause> is an <assignment clause> (*i.e.*, it contains <Lval expression>), then set $\mathbf{L_{lhs}}$ to the location of <Lval expression>.

   c) For all **i**, 1 (one) $\leq \mathbf{i} \leq \mathbf{n}$, do:

      i) If $\mathbf{P_i}$ = IN, then $\mathbf{HE_i}$ shall be SQLJ input assignable to $\mathbf{V_i}$. The Syntax Rules of Subclause 9.2, "Store assignment", in are not applied when setting $\mathbf{V_i}$ to the value of $\mathbf{HE_i}$. $\mathbf{BP_i}$ can set an input value.

      ii) If $\mathbf{P_i}$ = INOUT, then $\mathbf{HE_i}$ shall be SQLJ input assignable to $\mathbf{V_i}$. The Syntax Rules of Subclause 9.2, "Store assignment", in are not applied when simultaneously setting $\mathbf{V_i}$ to the value of $\mathbf{HE_i}$ and setting $\mathbf{L_i}$ to the location of $\mathbf{HE_i}$. $\mathbf{BP_i}$ can set an input value and return an output value.

      iii) If $\mathbf{P_i}$ = OUT, then set $\mathbf{L_i}$ to the location of $\mathbf{HE_i}$. $\mathbf{BP_i}$ can return an output value.

   d) For every $\mathbf{BP_i}$, 1 (one) $\leq \mathbf{i} \leq \mathbf{n}$, that can set an input value, set $\mathbf{BP_i}$ to $\mathbf{V_i}$.

   e) Execute the SQL-statement, using connection context **CCtx** and execution context **ECtx**. Execution results in the following values becoming available:

i) Values $\mathbf{O_i}$ for every $\mathbf{BP_i}$, 1 (one) $\leq \mathbf{i} \leq \mathbf{n}$, that can return an output value shall be SQLJ output assignable to $\mathbf{HE_i}$. The Syntax Rules of Subclause 9.1, "Retrieval assignment", in are not applied.

ii) If <executable clause> is an <assignment clause>, then the value $\mathbf{O_0}$ for $\mathbf{BP_0}$ shall be SQLJ output assignable to <Lval expression>. The Syntax Rules of Subclause 9.1, "Retrieval assignment", in are not applied.

f) For all **i** , 1 (one) $\leq \mathbf{i} \leq \mathbf{n}$, if $\mathbf{P_i}$ = OUT, or INOUT, then set the value at Java location $\mathbf{L_i}$ to $\mathbf{O_i}$.

g) If <executable clause> is an <assignment clause>, then set the value at Java location $\mathbf{L_{lhs}}$ to $\mathbf{O_0}$.

## Binary Composition

Unless explicitly specified, this Subclause defines the semantics of Binary Composition for all <executable clause>s.

The binary portability rules state that an <executable clause> should be able to:

— Use a passed <connection context> the type of which is a context class generated by any SQLJ-conformant translator (not necessarily the current translator).

— Instantiate and populate an iterator-valued <Lval expression> the type of which is an iterator class generated by any SQLJ-conformant translator (not necessarily the current translator).

The Binary Composition sections of the generated iterator and connection context classes define methods to support the above requirements. In particular:

— A connection context class defines **getProfileKey** and **getConnectedProfile** methods from which a connected profile object can be created. The connection context returned by these methods shall be used to obtain the **sqlj.runtime.profile.RTStatement** object that executes the SQL-statement.

— Iterator classes can be constructed using an instance of an **sqlj.runtime.profile.RTResultSet**. Any <executable clause> returning an iterator type shall construct the iterator using an RTResultSet object returned from the RTStatement object of the previous step.

## Code Generation

With the exception of <fetch statement>, all <executable clause>s share a common mechanism for obtaining an executable statement from a connection context, as described below.

1) Let **LC** be the name of the loading context class. The loading context class is any class that appears in the current SQLJ translation unit. It might be a class that is generated as a side effect of the code generation for the <executable clause>.

   The purpose of the loading context class is to be able to associate a profile stored as a serialized resource file with the appropriate class(es) at runtime. A profile shall be uniquely identified given the name of the profile and its associated loading context class. The loading context class shall be able to be used to uniquely identify the profile associated with a particular <executable clause>. By default, the loading context class will have a non-null class loader that can be used to load a resource by name as a Java stream. This would allow the profile resource to be read from the same JAR file that contained the loading context class, for



 

example. In other cases, the class might contain other identifying information such as a schema lookup path that would allow an associated resource file to be found in an appropriate schema.

A system class is not permitted to be used as a loading context class.

2) Let **PN** be the name of the profile associated with the current executable clause, as defined in Subclause 4.3.23, "Profile generation and naming".

3) Let **CT** be the name of the <connection context> class, which is the type of **CCtx**.

4) Let **i** be the number of <executable clause>s appearing in the current SQLJ translation unit prior to the current <executable clause> the <connection context> class of which is the same as **CT**. If this is the first such <executable clause>, then **i** is 0 (zero).

5) A profile loader for the <executable clause> is obtained using **LC**.

```
sqlj.runtime.profile.Loader loader =
    sqlj.runtime.RuntimeContext.getRuntime ( ).getLoader ( LC.class ) ;
```

For a particular Java Virtual Machine invocation, the value of **loader** will not change. The **loader** variable does not need to be re-evaluated each time <executable clause> is executed. Accordingly, it is safe to store **loader** as a static variable.

6) A profile key is obtained from **CT** using the loader and **PN**:

```
Object profileKey = CT.getProfileKey ( loader, PN ) ;
```

The getProfileKey method is a static method invoked on **CT**.

For a particular Java Virtual Machine invocation, the value of **profileKey** will not change. The **profileKey** variable does not need to be re-evaluated each time <executable clause> is executed. Accordingly, it is safe to store **profileKey** as a static variable.

7) A connected profile is obtained from **CCtx** using **profileKey**:

```
sqlj.runtime.profile.ConnectedProfile connProfile =
    CCtx.getConnectedProfile ( profileKey ) ;
```

Since the value of **CCtx** is only known at runtime, **connProfile** shall be re-evaluated each time <executable clause> is encountered.

8) Let **ECtx** be the current execution context object.

9) If **ECtx** has batching enabled and a pending statement batch context object, then let **BC** be the pending statement batch context object. Otherwise, let **BC** be null.

10) If **ECtx** does not have batching enabled, then the statement object to be executed is obtained from **connProfile** using **i** and the user-defined type map associated with **CT**.

```
sqlj.runtime.profile.RTStatement stmt =
    connProfile.getStatement ( i, CT.getTypeMap() ) ;
```

Otherwise, the statement object to be executed is obtained from `connProfile` using **i**, **BC**, and the user-defined type map associated with **CT**.

```
sqlj.runtime.profile.RTStatement stmt =
    connProfile.getStatement ( i, BC, CT.getTypeMap() ) ;
```

The statement **stmt** is used to execute the SQL-statement described by the **i**-th entry of profile **PN**, which can be obtained as follows:

```
sqlj.runtime.profile.EntryInfo entry =
    CT.getProfile ( profileKey ).getEntryInfo ( i ) ;
```

Once the statement **stmt** has been obtained, but before the statement is executed, execution control methods are permitted to be called as needed given the current <execution context>, **ECtx**. Execution control methods are defined by the following RTStatement interface methods:

— getMaxFieldSize

— setMaxFieldSize

— getMaxRows

— setMaxRows

— getQueryTimeout

— setQueryTimeout

Once the statement **stmt** has been executed but before the **executionComplete** method is called, execution status methods can be called. Execution status methods are defined by the following RTStatement interface methods:

— getWarnings

— clearWarnings

Other method calls made to the statement **stmt** vary according to the type of <executable clause>. The following rules represent the default calls made to bind inputs, execute, fetch outputs, and release the statement.

Unless explicitly specified, the following rules define the statement specific Code Generation calls for all <executable clause>s.

1) Let **k** represent the number of <embedded Java expression>s appearing in the <executable clause>.

2) Let **i** represent a variable ranging from 1 (one) to **k**.

3) For each <embedded Java expression> **HE** appearing in the <executable clause>, if the <parameter mode> of $\mathbf{HE_i}$ is IN or INOUT, then:

   a) Let **JT** represent the <java datatype> of $\mathbf{HE_i}$.

   b) Let **SM** represent the setter method corresponding to **JT**, as given in Table 2, “SQLJ type properties”.

   c) $\mathbf{HE_i}$ is bound to the statement using **SM**:

   ```
   stmt.SM ( i, HE_i ) ;
   ```

4) If **ECtx** has a batch context object with a pending statement batch and one or more of the following conditions are false:



 

a) **ECtx** has batching enabled, as defined by the `isBatching()` method.

b) The statement is batchable, as defined by the `isBatchable()` method.

c) The statement is batch compatible, as defined by the `isBatchCompatible()` method.

then the pending statement batch is executed on the batch context object using the method `executeBatch()`.

```
BC.executeBatch();
```

5) If batching is enabled on **ECtx**, as defined by `isBatching()`, and the statement is batchable, as defined by `isBatchable()`, then the statement is placed into a batch context object which becomes the current batch context object.

```
BC = stmt.getBatchContext();
```

6) Otherwise, if batching is not enabled on **ECtx** or the statement is not batchable, then the statement is executed using `executeUpdate`:

```
stmt.executeUpdate ( ) ;
```

7) For each <embedded Java expression> **HE** appearing in the <executable clause>, if the <parameter mode> of $\mathbf{HE_i}$ is OUT or INOUT, then

a) Let **JT** represent the <java datatype> of $\mathbf{HE_i}$.

b) Let **GM** represent the getter method corresponding to **JT**, as given in Table 2, "SQLJ type properties".

c) If **GM** is getObject, then $\mathbf{HE_i}$ is fetched from the statement using getObject and the compile-time class of **JT**:

```
HEi = stmt.getObject ( i, JT.class ) ;
```

d) If **GM** is not getObject, then $\mathbf{HE_i}$ is fetched from the statement using **GM**:

```
HEi = stmt.GM ( i ) ;
```

8) A call to executeComplete defines the end of the statement method invocations. It is called even if an exception occurs in an earlier step.

```
stmt.executeComplete ( ) ;
```

## Conformance Rules

1) Without Feature J003, "Execution control", conforming SQL language shall not contain an invocation of the `sqlj.runtime.ExecutionContext` class's public methods **setMaxFieldSize(int)**, **setMaxRows(int)**, or **setQueryTimeout(int)** that sets the corresponding `ExecutionContext` Java field to anything other than its default value, and shall not contain an attempt is made to register a statement with such an `ExecutionContext` (which, as specified under Code Generation in this Subclause, invokes the `sqlj.runtime.profile.RTStatement` interface's methods of the same name).

2) Without Feature J004, “Batch update”, conforming SQL langauge shall not contain an invocation of an implementation of the `sqlj.runtime.ExecutionContext` class's public methods `executeBatch()`, `getBatchLimit()`, `getBatchUpdateCounts()`, `isBatching()`, `setBatching(boolean)`, or `setBatchLimit(int)`.

3) Without Feature J009, “Multiple Open ResultSets”, conforming SQL language shall not contain an invocation of an implementation of the `sqlj.runtime.ExecutionContext` class's public method **getNextResultSet(int)** with any value other than `java.sql.Statement.CLOSE_CURRENT_RESULT`.



 

## 11.10 <context clause>

### Function

Specify an execution context or statement context.

### Format

```
<context clause> ::=
  <left bracket> <context spec clause> <right bracket>

<context spec clause> ::=
    <connection context>
  | <execution context>
  | <connection context> <comma> <execution context>

<connection context> ::=
  <java id>

<execution context> ::=
  <java id>
```

### Syntax Rules

*None.*

### Access Rules

*None.*

### General Rules

*None.*

### Definitions and Rules

1) A <connection context> shall reference a Java expression the type of which is generated connection class, or a subclass of such a class.

2) If <connection context> is not explicitly stated, then the default connection is used for the executable statement.

3) An <execution context> shall reference a Java expression the type of which is **sqlj.runtime.Execution-Context**, or a subclass of such a class.

4) If <execution context> is not explicitly stated, then the <execution context> is taken from the statement's connection context.

## Conformance Rules

*None.*



 

## 11.11 <statement clause>

### Function

Specify the execution of a subset of SQL-statements.

### Format

```
<statement clause> ::=
  <left brace> <statement spec clause> <right brace>

<statement spec clause> ::=
    <SQL procedure statement>
  | <compound statement>
```

### Syntax Rules

1) An <SQL procedure statement> contained in a <statement spec clause> that immediately contains an <SQL executable statement> that immediately contains an <SQL control statement> shall immediately contain a <call statement>.

2) An <SQL procedure statement> contained in a <statement spec clause> shall not immediately contain an <SQL executable statement> that immediately contains an <SQL data statement> that immediately contains <open statement>, <close statement>, <free locator statement>, or <hold locator statement>.

3) An <SQL procedure statement> contained in a <statement spec clause> shall not immediately contain an <SQL executable statement> that immediately contains an <SQL transaction statement> that immediately contains <start transaction statement>.

4) An <SQL procedure statement> contained in a <statement spec clause> shall not immediately contain an <SQL executable statement> that immediately contains an <SQL connection statement>, <SQL session statement>, <SQL diagnostics statement>, or <SQL dynamic statement>.

### Access Rules

*None.*

### General Rules

*None.*

### Profile EntryInfo Properties

— **SQL String** — Default as described in Subclause 4.3.5.1, “EntryInfo overview”.

— **Role** — STATEMENT

## Conformance Rules

1) Without Feature J005, “Call statement”, conforming SQL language shall not contain a <statement spec clause> that contains a <call statement>.



 

## 11.12 <delete statement: positioned>

*This Subclause modifies Subclause 14.8, "<delete statement: positioned>", in ISO/IEC 9075-2.*

### Function

Delete a row of a table.

### Format

```
<delete statement: positioned> ::=
    !! All alternatives from ISO/IEC 9075-2
  | DELETE FROM <target table> WHERE CURRENT OF <iterator host expression>

<iterator host expression> ::=
  <embedded Java expression>
```

### Syntax Rules

1) Insert this SR Case:

   a) If <delete statement: positioned> is contained in an <embedded SQL Java program>, then <iterator host expression> shall be specified.

   b) Otherwise, <iterator host expression> shall not be specified.

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Definitions and Rules

1) <iterator host expression> shall result in an instance of either a generated positioned iterator class or a generated named iterator class that implements the interface **sqlj.runtime.ForUpdate**.

2) The DELETE privilege for the execution of <delete statement: positioned> is based upon the authorization identifier that was used to execute the <query clause> associated with <iterator host expression>.

### Profile EntryInfo Properties

— **SQL String** — Default as described in Subclause 4.3.5.1, "EntryInfo overview", with the <iterator host expression> replaced by a <dynamic parameter specification>.

— **Role** — POSITIONED

— **Parameter Java fields** — Describes the <embedded Java expression>s of the <delete statement: positioned>, including the <iterator host expression>.

— **Descriptor** — A 1-based java.lang.Integer value denoting the index of the <iterator host expression> within the Parameter Java fields. This is used to be able to conveniently determine which parameter corresponds to the positioned iterator.

## Conformance Rules

*No additional Conformance Rules.*



 

## 11.13 <update statement: positioned>

*This Subclause modifies Subclause 14.13, "<update statement: positioned>", in ISO/IEC 9075-2.*

### Function

Update a row of a table.

### Format

```
<update statement: positioned> ::=
    !! All alternatives from ISO/IEC 9075-2
  | UPDATE <target table> SET <set clause list> WHERE CURRENT OF <iterator host expression>
```

### Syntax Rules

1) Insert this SR Case:

   a) If <update statement: positioned> is contained in an <embedded SQL Java program>, then <iterator host expression> shall be specified.

   b) Otherwise, <iterator host expression> shall not be specified.

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Definitions and Rules

1) <iterator host expression> shall result in an instance of either a generated positioned iterator class or a generated named iterator class that implements the interface **sqlj.runtime.ForUpdate**.

2) The UPDATE (column-specific) privilege for the execution of <update statement: positioned> is based upon the authorization identifier that was used to execute the <query clause> associated with <iterator host expression>.

### Profile EntryInfo Properties

— **SQL String** — Default as described in Subclause 4.3.5.1, "EntryInfo overview", with the <iterator host expression> replaced by a <dynamic parameter specification>.

— **Role** — POSITIONED

— **Parameter Java fields** — Describes the <embedded Java expression>s of the <update statement: positioned>, including the <iterator host expression>.

— **Descriptor** — A 1-based java.lang.Integer value denoting the index of the <iterator host expression> within the Parameter Java fields. This is used to be able to conveniently determine which parameter corresponds to the positioned iterator.

## Conformance Rules

*No additional Conformance Rules.*



 

## 11.14 <select statement: single row>

*This Subclause modifies Subclause 14.7, "<select statement: single row>", in ISO/IEC 9075-2.*

### Function

Retrieve values from a specified row of a table.

### Format

```
No additional Format items.
```

### Syntax Rules

1) Replace SR 3)b)iv) For each <target specification> *TS* that is an <embedded variable specification>,

   Case:

   a) If <select statement: single row> is contained in an <embedded SQL Java program>, then the value of the corresponding element of the <select list> shall be SQLJ output assignable to *T*.

   b) Otherwise, the Syntax Rules of Subclause 9.1, "Retrieval assignment", shall apply to *TS* as *TARGET* and the corresponding element of the <select list> as *VALUE*.

### Access Rules

*No additional Access Rules.*

### General Rules

1) Replace GR 4)b)iv) For each <target specification> *TS* that is an <embedded variable specification>,

   Case:

   a) If the <select statement: single row> is contained in an <embedded SQL Java program>, then the corresponding value in the row of *Q* is assigned to *TS* as described in the following Code Generation.

   b) Otherwise, the General Rules of Subclause 9.1, "Retrieval assignment", are applied with the corresponding value in the row of *Q* as *VALUE* and *TS* as *TARGET*. The assignment of values to targets in the <select target list> is in an implementation-dependent order.

### Definitions and Rules

1) If the <select statement: single row> results in the generation of no rows, then an SQLException condition is thrown: *no data*.

2) If the <select statement: single row> results in the generation of more than one row, then an SQLException condition is thrown: *cardinality violation*.

3) Let **k** represent the cardinality of the <select target list>.

4) Let **j** represent the cardinality of the <select list>.

5) Let **SLE** denote an expression of <select list>.

6) If **k ≠ j**, then an SQLException is thrown: *OLB-specific error — invalid number of columns*.

7) Let $\mathbf{HE_i}$, 1 (one) **≤ i ≤ k**, represent an <embedded Java expression> in <select target list>.

   a) $\mathbf{HE_i}$ <expression> shall be either a <simple variable> or an <Lval expression>.

   b) If $\mathbf{HE_i}$ explicitly states a <parameter mode>, then the specified mode shall be **OUT**.

   c) Let **JT** represent the corresponding <java datatype> of $\mathbf{HE_i}$.

   d) If **JT** is a Java primitive datatype, and the value of the corresponding argument is an SQL null value, then raise an exception of type sqlj.runtime.SQLNullException.

## Profile EntryInfo Properties

— **SQL String** — The text of the <select statement: single row> with INTO <select target list> removed.

— **Role** — SINGLE_ROW_QUERY

— **Execute Type** — EXECUTE_QUERY

— **Parameter Java fields** — Describe all <embedded Java expression>s appearing in the <select statement: single row>, except those appearing in the <select target list>.

— **Result Set Column Java fields** — Describes the <embedded Java expression>s appearing in the <select target list> of the <select statement: single row>.

- **Result Set Type** — POSITIONED_RESULT
- **Result Set Count** — The cardinality of the <select target list>, **k**.
- **Result Set Info** — Returns a TypeInfo object describing a particular **HE** of the **<select target list>**. The *i*-th TypeInfo object describes $\mathbf{HE_i}$. The TypeInfo object returned has mode OUT, dynamic parameter marker index –1, and Java type name corresponding to the name of the type of $\mathbf{HE_i}$. If $\mathbf{HE_i}$ is a <simple variable>, then the TypeInfo object returned has the same name as that of the <simple variable>. Otherwise, if $\mathbf{HE_i}$ is a <complex expression>, then the TypeInfo object returned has a null name.
- **Result Set Name** — null

## Code Generation

1) Let **k** represent the number of <embedded Java expression>s appearing in the <select statement: single row>, not including those in the <select target list>.

2) Let **i** represent a variable ranging from 1 (one) to **k**.



 

3) For each <embedded Java expression> **HE$_i$** appearing in the <select statement: single row> that does not appear in the <select target list>:

   a) Let **JT** represent the <java datatype> of **HE$_i$**.

   b) Let **SM** represent the setter method corresponding to **JT**, as given in Table 2, "SQLJ type properties".

   c) **HE$_i$** is bound to the statement using **SM**.

```
stmt.SM ( i, HEi ) ;
```

4) If **ECtx** has a batch context object **BC** with a pending statement batch and one or more of the following conditions are false:

   a) **ECtx** has batching enabled, as defined by the `isBatching()` method.

   b) The statement is batchable, as defined by the `isBatchable()` method.

   c) The statement is batch compatible, as defined by the `isBatchCompatible()` method.

   then the pending statement batch is executed on the batch context object using the method `executeBatch()`.

```
BC.executeBatch();
```

5) An RTResultSet is created using executeQuery:

```
rs = stmt.executeQuery() ;
```

6) A call to executeComplete defines the end of the statement method invocations. It is called even if an exception occurs in an earlier step:

```
stmt.executeComplete ( ) ;
```

7) Let **n** represent the number of <embedded Java expression>s appearing in the <select target list>.

8) Let **m** represent the number of columns in the returned RTResultSet object:

```
m = rs.getColumnCount ( ) ;
```

9) If **m ≠ n**, then an SQLException is thrown by the generated code.

10) The RTResultSet object is advanced to its first and only row using `next()`:

```
rs.next ( ) ;
```

11) If the invocation of `next()` returns **false**, indicating that there were no rows in the RTResultSet object, then an SQLException is thrown by the generated code.

12) Let **j** represent a variable ranging from 1 (one) to **n**.

13) For each <embedded Java expression> **HE$_j$** in <select target list>:

   a) Let **JT** represent the <java datatype> of **HE$_j$**.

   b) Let **GM** represent the getter method corresponding to **JT**, as given in Table 2, "SQLJ type properties".

c) If **GM** is getObject, then **HE**$_{\mathbf{j}}$ is fetched from the RTResultSet object using getObject and the compile-time class of **JT**.

```
HEj = rs.getObject ( j, JT.class ) ;
```

d) If **GM** is not getObject, then **HE**$_{\mathbf{j}}$ is fetched from the RTResultSet object using **GM**.

```
HEj = rs.GM ( j ) ;
```

e) A subsequent call to `next()` is made on the RTResultSet object to verify that there are no further rows:

```
rs.next ( ) ;
```

f) If the subsequent call to `next()` returns **true**, indicating that there were additional rows, then an SQLException is thrown by the generated code.

g) The RTResultSet object is closed, even if an exception occurs in an earlier step:

```
rs.close ( ) ;
```

NOTE 24 — If an implementation is able to detect that more than one row is returned, then an exception condition may be raised prior to the second invocation of `rs.next()`. Applications should not rely upon the <select target list> containing the first row's values if there is more than one result row.

## Conformance Rules

*No additional Conformance Rules.*



 

## 11.15 <fetch statement>

*This Subclause modifies Subclause 14.5, "<fetch statement>", in ISO/IEC 9075-2.*

### Function

Position a cursor on a specified row of a table and retrieve values from that row.

### Format

```
<fetch statement> ::=
    !! All alternatives from ISO/IEC 9075-2
  | FETCH [ [ <fetch orientation> ] FROM ]
      <iterator host expression> INTO <fetch target list>
```

### Syntax Rules

1) Insert this SR Case:

   a) If <fetch statement> is contained in an <embedded SQL Java program>, then <iterator host expression> shall be specified.

   b) Otherwise, <iterator host expression> shall not be specified.

2) Replace SR 3) Case:

   a) If <fetch statement> is contained in an <embedded SQL Java program>, then let *DC* be the implicit <declare cursor> of <iterator host expression>, let *CR* be the implicit cursor of <iterator host expression>, and let *T* be the table defined by the <cursor specification> of *DC*.

   b) Otherwise, let *CN* be the <cursor name> in the <fetch statement>. *CN* shall be contained within the scope of one or more <cursor name>s that are equivalent to *CN*. If there is more than one such <cursor name>, then the one with the innermost scope is specified. Let *CR* be the cursor specified by *CN*. Let *T* be the table defined by the <cursor specification> of *CR*. Let *DC* be the <declare cursor> denoted by *CN*.

3) Insert after SR 9)b)iv) For each <target specification> $TS3_i$, 1 (one) **≤** *i* **≤** *NTS*, that is an <embedded variable name>,

   Case:

   a) If <fetch statement> is contained in an <embedded SQL Java program>, then the value of the corresponding column of table *T* shall be SQLJ output assignable to $TS3_i$.

   b) Otherwise, the Syntax Rules of Subclause 9.2, "Store assignment", in ISO/IEC 9075-2, are applied with $TS3_i$ as *TARGET* and $CS_i$ as *VALUE*.

### Access Rules

*No additional Access Rules.*

## General Rules

1) Replace GR 5)b) Otherwise, if the <fetch target list> contains more than one <target specification>, then values from the current row are assigned to their corresponding targets identified by the <fetch target list>. If <fetch statement> is not contained in an <embedded SQL Java program>, then the assignments are made in an implementation-dependent order. Let *TV* be a target and let *SV* denote its corresponding value in the current row of *CR*.

   Case:

   a) If *TV* is the <SQL parameter name> of an SQL parameter of an SQL-invoked routine, then the General Rules of Subclause 9.2, "Store assignment" are applied with *TS* as *TARGET* and *SV* as *VALUE*.

   b) Otherwise,

      Case:

      i) If <fetch statement> is contained in an <embedded SQL Java program>, then *SV* is assigned to *TV* as described in the following Code Generation.

      ii) Otherwise, the General Rules of Subclause 9.1, "Retrieval assignment", in , are applied to *TV* as *TARGET* and *SV* as *VALUE*.

## Definitions and Rules

1) If the execution of a <fetch statement> results in a row not found, then the values of the <embedded Java expression>s contained in the <fetch target list> are implementation-dependent.

2) If the execution of a <fetch statement> results in a row not found, then `endFetch()` becomes **true**.

   NOTE 25 — No SQLException is thrown for this condition.

3) <iterator host expression> shall result in an instance of a generated positioned iterator class or a subclass of such a class.

4) The SELECT privilege for the execution of <fetch statement> is based upon the authorization identifier that was used to execute the <query clause> associated with <iterator host expression>.

5) Let **k** represent the cardinality of the <fetch target list>.

6) Let **j** represent the cardinality of the associated iterator's <java type list>.

7) If **k ≠ j**, then an SQLException condition is thrown: *OLB-specific error — invalid number of columns*.

8) Let **SLE** denote a SELECT list expression of the associated iterator.

9) Let **IT** denote a <java datatype> in the associated iterator's <java type list>.

10) Let $\mathbf{HE_i}$, 1 (one) **≤ i ≤ k**, represent an <embedded Java expression> in <fetch target list>:

    a) $\mathbf{HE_i}$ <expression> shall be either a <simple variable> or an <Lval expression>.

    b) If $\mathbf{HE_i}$ explicitly states a <parameter mode>, then the specified mode shall be **OUT**.

    c) Let **JT** represent the corresponding <java datatype> of $\mathbf{HE_i}$.



 

d) **IT$_i$** shall be the same as **JT**.

e) If **JT** is a Java primitive datatype, and the value of the corresponding argument is an SQL null value, then raise an exception of type sqlj.runtime.SQLNullException.

## Profile EntryInfo Properties

The <fetch statement> is implemented as a client-side translation that populates the <embedded Java expression>s of the <fetch target list> using the contents of the current row of the iterator. Since the API defined by [JDBC] uses method execution instead of supporting <fetch statement>, <fetch statement> does not appear in the profile.

## Code Generation

The <fetch statement> represents a client-side translation that does not appear in the profile. Accordingly, it does not access a connection context or its contained connected profile.

1) Let **IE** represent the <iterator host expression>. **IE** shall be an instance of a class or subclass of *generated iterator class*.

2) Case:

   a) If <fetch orientation> specifies NEXT, then let **IEM** be `next()`.

   b) If <fetch orientation> specifies PRIOR, then let **IEM** be `previous()`.

   c) If <fetch orientation> specifies FIRST, then let **IEM** be `first()`.

   d) If <fetch orientation> specifies LAST, then let **IEM** be `last()`.

   e) If <fetch orientation> specifies ABSOLUTE, then let **IHE** be the value of the <simple value specification> and let **IEM** be `absolute(IHE)`.

   f) If <fetch orientation> specifies RELATIVE, then let **IHE** be the value of the <simple value specification> and let **IEM** be `relative(IHE)`.

3) The iterator is positioned on a row using **IEM**:

```
IE.IEM;
```

4) If the invocation of **IEM** returns **true**, then:

   a) Let **n** represent the number of <embedded Java expression>s appearing in the <fetch target list>.

   b) Let **j** represent a variable ranging from 1 (one) to **n**.

   c) For each <embedded Java expression> **HE$_j$** in <fetch target list>, **HE$_j$** is fetched from the iterator using getCol:

```
HEj = IE.getColj ( ) ;
```

## Conformance Rules

*No additional Conformance Rules.*

## 11.16 <assignment statement>

*This Subclause modifies Subclause 15.5, "<assignment statement>", in ISO/IEC 9075-4.*

### Function

Assign a value to an SQL variable, SQL parameter, host parameter, or host variable.

### Format

```
No additional Format items.
```

### Syntax Rules

1) Replace SR 9) Case:

   a) If <assignment statement> is contained in an <embedded SQL Java program> and the <assignment target> simply contains an <embedded variable name>, then let *AT* represent the <assignment target>, let *JT* represent the <java datatype> of *AT*, and let *ST* represent the SQL type of <assignment source>. *ST* shall be output assignable to *JT*.

   b) Otherwise, if the <assignment target> simply contains an <embedded variable name> or a <host parameter specification> and the <assignment source> is a <value expression>, then the Syntax Rules of Subclause 9.1, "Retrieval assignment", in [ISO9075-2], are applied to <assignment target> as *TARGET* and <assignment source> as *VALUE*.

### Access Rules

*No additional Access Rules.*

### General Rules

1) Replace GR 2) If <assignment target> is a <target specification> that is the <embedded variable name> of a host variable or embedded Java expression *T* or the <host parameter specification> of a host parameter *T*, then

   Case:

   a) If <assignment statement> is not contained in an <embedded SQL Java program>, then the value of <assignment source> is assigned to *T* according to the General Rules of Subclause 9.1, "Retrieval assignment", in [ISO9075-2], with the value of <assignment source> as *VALUE* and *T* as *TARGET*.

   b) Otherwise, the value of <assignment source> is assigned to *T* as specified in Subclause 11.9, "<executable clause>".

### Definitions and Rules

1) Let **AT** represent the <assignment target>.



 

2) Let **TT** represent the <java datatype> of **AT**.

3) Let **AS** represent the SQL type of <assignment source>.

4) **AT** shall be either a <simple variable> or <Lval expression>.

5) If **AT** explicitly states a <parameter mode>, then <parameter mode> shall specify **OUT**.

6) If **TT** is a Java primitive datatype, and the runtime value of **AS** is an SQL null value, then raise an exception of type sqlj.runtime.SQLNullException.

## Profile EntryInfo Properties

— **SQL String** — Uses the default as specified in Subclause 4.3.5.1, "EntryInfo overview".

— **Role** — STATEMENT

## Conformance Rules

*No additional Conformance Rules.*

## 11.17 <savepoint statement>

*This Subclause modifies Subclause 17.5, "<savepoint statement>", in ISO/IEC 9075-2.*

### Function

Establish a savepoint.

### Format

```
No additional Format items.
```

### Syntax Rules

*No additional Syntax Rules.*

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Profile EntryInfo Properties

— **SQL String** — Exact text of the matching production.

— **Role** — SAVEPOINT

— **Parameter Java fields** — No parameters allowed.

### Conformance Rules

*No additional Conformance Rules.*



 

## 11.18 <release savepoint statement>

*This Subclause modifies Subclause 17.6, "<release savepoint statement>", in ISO/IEC 9075-2.*

### Function

Destroy a savepoint.

### Format

```
No additional Format items.
```

### Syntax Rules

*No additional Syntax Rules.*

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Profile EntryInfo Properties

— **SQL String** — Exact text of the matching production.

— **Role** — RELEASE_SAVEPOINT

— **Parameter Java fields** — No parameters allowed.

### Conformance Rules

*No additional Conformance Rules.*

## 11.19 <commit statement>

*This Subclause modifies Subclause 17.7, "<commit statement>", in ISO/IEC 9075-2.*

### Function

Terminate the current SQL-transaction with commit.

### Format

```
No additional Format items.
```

### Syntax Rules

1) Insert this SR Neither AND CHAIN nor AND NO CHAIN shall be specified in a <commit statement> contained in a <statement clause>.

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Definitions and Rules

*None.*

NOTE 26 — Conformance to SQL/OLB requires support only of the COMMIT and optional WORK keywords.

### Profile EntryInfo Properties

— **SQL String** — Exact text of the matching production.

— **Role** — COMMIT

— **Parameter Java fields** — No parameters allowed.

### Conformance Rules

*No additional Conformance Rules.*



 

## 11.20 <rollback statement>

*This Subclause modifies Subclause 17.8, "<rollback statement>", in ISO/IEC 9075-2.*

### Function

Terminate the current SQL-transaction with rollback, or rollback all actions affecting SQL-data and/or schemas since the establishment of a savepoint.

### Format

```
No additional Format items.
```

### Syntax Rules

*No additional Syntax Rules.*

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Definitions and Rules

*None.*

NOTE 27 — Conformance to SQL/OLB requires support only of the ROLLBACK and optional WORK keywords. If support of Feature T271, "Savepoints", is claimed, then the <savepoint clause> is also supported.

### Profile EntryInfo Properties

— **SQL String** — Exact text of the matching production.

— **Role** — ROLLBACK

— **Parameter Java fields** — No parameters allowed.

### Conformance Rules

*No additional Conformance Rules.*

## 11.21 <set transaction statement>

*This Subclause modifies Subclause 17.2, "<set transaction statement>", in ISO/IEC 9075-2.*

### Function

Set the characteristics of the next SQL-transaction for the SQL-agent.

### Format

```
No additional Format items.
```

### Syntax Rules

1) Insert this SR If <set transaction statement> is contained in an <embedded SQL Java program>, then LOCAL shall not be specified.

2) Insert this SR If <set transaction statement> is contained in an <embedded SQL Java program>, then <set transaction statement> shall not contain a <transaction mode> that immediately contains <diagnostics size>.

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Profile EntryInfo Properties

— **SQL String** — Exact text of matching production

— **Role** — SET_TRANSACTION

— **Parameter Java fields** — No parameters allowed.

— **Descriptor** — An instance of class **sqlj.runtime.profile.SetTransactionDescriptor** that describes the <transaction access mode> and <isolation level>.

### Conformance Rules

*No additional Conformance Rules.*



 

## 11.22 <call statement>

*This Subclause modifies Subclause 16.1, "<call statement>", in ISO/IEC 9075-2.*

### Function

Invoke 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.*

### Definitions and Rules

1) If an <embedded Java expression> contained in an <SQL argument> does not explicitly state a <parameter mode>, then its <parameter mode> is implicitly IN.

### Profile EntryInfo Properties

— **SQL String** — Rewritten in JDBC specified procedure call syntax:

```
{ <call statement> }
```

— **Role** — CALL

— **Execute Type** — EXECUTE

### Code Generation

1) All <embedded Java expression>s with <parameter mode> of IN or INOUT are bound as specified in the default rules for <executable clause>.

2) If **ECtx** has a batch context object **BC** with a pending statement batch and one or more of the following conditions are false:

a) **ECtx** has batching enabled, as defined by the `isBatching()` method.

b) The statement is batchable, as defined by the `isBatchable()` method.

c) The statement is batch compatible, as defined by the `isBatchCompatible()` method.

then the pending statement batch is executed on the batch context object using the method `executeBatch()`.

```
BC.executeBatch();
```

3) If batching is enabled on **ECtx**, as defined by `isBatching()`, and the statement is batchable, as defined by `isBatchable()`, then the statement is placed into a batch context object which becomes the current batch context object.

```
BC = stmt.getBatchContext();
```

4) Otherwise, if batching is not enabled on **ECtx** or the statement is not batchable, then the statement is executed using execute:

```
stmt.execute ( ) ;
```

5) All <embedded Java expression>s with <parameter mode> of OUT or INOUT are assigned as specified in the default rules for <executable clause>.

6) A call to executeComplete defines the end of the statement method invocations. It is called only after all side-channel results have been visited using the associated execution context's getNextResultSet method. If no side-channel results are produced, executeComplete is called immediately. It is called even if an exception occurs in an earlier step:

```
stmt.executeComplete ( ) ;
```

## Conformance Rules

*No additional Conformance Rules.*



 

## 11.23 <assignment clause>

### Function

Assigns a value to a Java variable, Java field, or parameter.

### Format

```
<assignment clause> ::=
  <Lval expression> <equals operator>
      <left brace> <assignment spec clause> <right brace>

<assignment spec clause> ::=
    <query clause>
  | <function clause>
  | <iterator conversion clause>
```

### Syntax Rules

*None.*

### Access Rules

*None.*

### General Rules

*None.*

### Definitions and Rules

1) An <assignment clause> shall not appear in the control list of a **FOR** loop.

2) An <assignment clause> is not permitted to appear where a Java assignment expression, but not an assignment statement, is permitted to appear.

3) An <assignment clause> has the effect of evaluating <assignment spec clause> and assigning its value to `<Lval expression>`.

4) If the <assignment spec clause> is a <query clause>, then:

   a) Let **JT** represent the <java datatype> of `<Lval expression>`.

   b) **JT** shall refer to an object the type of which is generated iterator class or interface sqlj.runtime.ResultSetIterator.

   c) If **JT** is a generated positioned iterator class, then the number and types of columns of the query shall match those of the iterator class declaration.

5) If the <assignment spec clause> is a <function clause>, then:

a) Let **JT** represent the <java datatype> of `<Lval expression>`.

b) Let **FT** represent the SQL datatype that is returned by the invocation of the function.

c) If **JT** is a Java primitive datatype and the runtime value of the **FT** is an SQL null value, then raise an exception of type sqlj.runtime.SQLNullException.

d) **FT** shall be SQLJ output assignable to **JT**.

6) If the <assignment spec clause> is an <iterator conversion clause>, then:

a) Let **JT** be the <java datatype> of <Lval expression>.

b) **JT** shall refer to an object whose type is generated iterator class.

## Conformance Rules

*None.*



 

## 11.24 <query clause>

### Function

Specify a statement to retrieve multiple rows from a specified table.

### Format

```
<query clause> ::=
  <query expression>
```

### Syntax Rules

1) Let *QC* be the <query clause> and let *QE* be the <query expression> immediately contained in *QC*.

2) Let *AC* be the <assignment clause> whose <assignment spec clause> immediately contains *QC* and let *ACI* be the iterator object referenced by the <Lval expression> immediately contained in *AC*.

3) Let *IDC* be the implicit <declare cursor> of *ACI* that is the <declare cursor> effectively performed by an SQL-server as a result of the execution of *AC* and let *CS* be the cursor specified by *IDC*.

4) *QE* is the *simply underlying table specification* of *CS*.

### Access Rules

*None.*

### General Rules

1) Let *T* be the result of the <query expression>.

2) If *T* is empty, then a completion condition is raised: *no data*.

3) The result of <query clause> is *T*.

NOTE 28 — The result of <query clause> is effectively the sort table of the <query expression> with all extended sort key columns (if any) removed. “sort table” and “extended sort key column” are defined in Subclause 7.17, “<query expression>”, in [ISO9075-2].

### Definitions and Rules

1) Let *Q* be the table specified by the <query expression>.

2) Let **j** represent the degree of *Q*.

3) Let **SLE** denote a column of *Q*.

4) Let **IT** represent the iterator type of the associated iterator object.

5) Case:

a) If **IT** is interface sqlj.runtime.ResultSetIterator, then no further type checking is performed on **SLE**.

b) If **IT** is of type <positioned iterator>, then:

i) Let **k** represent the cardinality of the <java type list> of the associated iterator object.

ii) If **k ≠ j**, then an SQLException is thrown: *OLB-specific error — invalid number of columns*.

iii) Let **i** represent a variable ranging from 1 (one) to **k**.

iv) For each <java datatype> **JD** in <java type list>, let **i** be its associated range variable.

1) Let **ST** represent the SQL datatype of $\mathbf{SLE_i}$.

2) Let **JT** represent the Java datatype of $\mathbf{JD_i}$.

c) If **IT** is of type <named iterator>, then:

i) If **SLE** is not named in the query by a legal Java identifier, then **SLE** shall be given a column alias that is a legal Java identifier by means of the SQL phrase **AS *identifier***.

ii) Let **k** represent the cardinality of the <java pair list>.

iii) If **k** is greater than **j**, then an SQLException condition is thrown: *OLB-specific error — invalid number of columns*.

iv) Let **SLN** represent the SQL name or alias of an associated **SLE**.

v) Let **i** represent a variable ranging from 1 (one) to **k**.

vi) Let **n** represent a variable ranging from 1 (one) to **j**.

vii) For each <java pair> **JP** in <java pair list> let **i** be its associated range variable.

1) If there exists an **SLE**, the **SLN** of which is a case-insensitive match of the <java id> associated with $\mathbf{JP_i}$, then let **n** be the index of the first such **SLN**; otherwise, let **n** be 0 (zero).

2) If **n** is 0 (zero), then an SQLException condition is thrown: *OLB-specific error — invalid number of columns*.

3) Let **JT** represent the Java datatype of $\mathbf{JP_i}$.

4) Let **ST** represent the SQL datatype of $\mathbf{SLE_n}$.

d) **ST** shall be SQLJ output assignable to **JT**.

6) The constructor method of the corresponding <iterator declaration clause> shall be in scope.

## Profile EntryInfo Properties

— **SQL String** — Default as described in Subclause 4.3.5.1, “EntryInfo overview”.

— **Role** — If **IT** implements **interface sqlj.runtime.ForUpdate**, then role is QUERY_FOR_UPDATE. If **IT** is (not merely implements) interface sqlj.runtime.ResultSetIterator, then role is UNTYPED. Otherwise, role is QUERY



 

— **Execute Type** — If **IT** is interface sqlj.runtime.ResultSetIterator, then EXECUTE_UPDATE; otherwise, EXECUTE_QUERY

— **Parameter Java fields** — Describes the <embedded Java expression>s appearing in the <executable clause>, as described in Subclause 4.3.5.1, "EntryInfo overview". Additionally, if **IT** is interface sqlj.runtime.ResultSetIterator, then the <Lval expression> of the return result is described as an additional parameter.

  • Param Count — Gives the number of <embedded Java expression>s appearing in the <executable clause>, as described in Subclause 4.3.5.1, "EntryInfo overview". Additionally, if **IT** is interface sqlj.runtime.ResultSetIterator, then the count is incremented by one to reflect the parameter describing the <Lval expression> of the return result.

  • Param Info — Returns a TypeInfo object describing a particular <embedded Java expression>, as described in Subclause 4.3.5.1, "EntryInfo overview". If **IT** is interface sqlj.runtime.ResultSetIterator, then a TypeInfo object describing the <Lval expression> of the return result is returned as an additional parameter, appearing after all other parameters.

— **Result Set Column Java fields** — Describes the result set columns, as expected by **IT**.

  • **Result Set Name** — If **IT** is interface sqlj.runtime.ResultSetIterator, then null; otherwise, the class name of **IT**.

  • **Result Set Type** — If **IT** is of type <positioned iterator>, then `POSITIONED_RESULT`. If **IT** is of type <named iterator>, then `NAMED_RESULT`. If **IT** is interface sqlj.runtime.ResultSetIterator, then `NO_RESULT`.

  • **Result Set Count** — If **IT** is of type <positioned iterator>, then the cardinality of the <java type list> of the associated iterator, **k**. If *IT* is of type <named iterator>, then the cardinality of the <java pair list> of the associated iterator, **k**. If **IT** is interface sqlj.runtime.ResultSetIterator, then 0 (zero).

  • **Result Set Info** — If **IT** is of type <positioned iterator>, then returns a TypeInfo object describing a **JD** in <java type list>. The *i*-th TypeInfo object describes $\mathbf{JD_i}$. The TypeInfo object returned has name = null, mode = OUT, dynamic parameter marker index = –1, and Java type name corresponding to the name of the type of $\mathbf{JD_i}$. If *IT* is of type <named iterator>, then returns a TypeInfo object describing a **JP** in <java type list>. Since a named iterator is used, the order of the TypeInfo objects returned is implementation-dependent. For each **JP**, there exists exactly one TypeInfo object describing **JP**, which has name=<java id> of **JP**, mode = OUT, dynamic parameter marker index = –1, and Java type name corresponding to the name of the type of **JP**. If **IT** is interface sqlj.runtime.ResultSetIterator, then there are no TypeInfo objects returned.

## Code Generation

1) Let **k** represent the number of <embedded Java expression>s appearing in the <query clause>.

2) Let **i** represent a variable ranging from 1 (one) to **k**.

3) For each <embedded Java expression> $\mathbf{HE_i}$ appearing in the <query clause>:

   a) Let **JT** represent the <java datatype> of $\mathbf{HE_i}$.

   b) Let **SM** represent the setter method corresponding to **JT**, as given in Table 2, "SQLJ type properties".

   c) $\mathbf{HE_i}$ is bound to the statement using **SM**:

```
stmt.SM ( i, HEi ) ;
```

4) Let **IE** represent the <Lval expression> on the left hand side of the <assignment clause>.

5) Let **IT** represent the <java datatype> of **IE**.

6) If **ECtx** has a batch context object **BC** with a pending statement batch and one or more of the following conditions are false:

   a) **ECtx** has batching enabled, as defined by the `isBatching()` method.

   b) The statement is batchable, as defined by the `isBatchable()` method.

   c) The statement is batch compatible, as defined by the `isBatchCompatible()` method.

   then the pending statement batch is executed on the batch context object using the method `executeBatch()`.

```
BC.executeBatch();
```

7) Case:

   a) If **IT** is interface sqlj.runtime.ResultSetIterator, then:

      i) If batching is enabled on **ECtx**, as defined by `isBatching()`, and the statement is batchable, as defined by `isBatchable()`, then the statement is placed into a batch context object which becomes the current batch context object.

```
BC = stmt.getBatchContext();
```

      ii) The statement is executed using **executeUpdate**.

      iii) **IE** is fetched from the statement using **getObject**, the compile-time class of **IT**, and an index one greater than the number of <embedded Java expression>s in the <query clause>:

```
IE = stmt.getObject(k + 1, IT.class);
```

   b) Otherwise:

      i) An RTResultSet is created using **executeQuery**.

      ii) **IE** is assigned to the result of creating a new iterator object.

8) A call to **executeComplete** defines the end of the statement method invocations. It is called even if an exception occurs in an earlier step:

```
stmt.executeComplete ( ) ;
```

## Conformance Rules

*None.*



 

# 11.25 <function clause>

## Function

Invoke an SQL-invoked function.

## Format

```
<function clause> ::=
  VALUES <left paren> <routine invocation> <right paren>
```

## Syntax Rules

1) Let *RI* be the <routine invocation> immediately contained in the <function clause>.

2) The Syntax Rules of Subclause 7.1, "<routine invocation>" are applied with *RI* as *ROUTINE INVOCATION*, the SQL-path (if any) as *SQLPATH*, and the user-defined type of the static SQL-invoked method (if any) as *UDT*, yielding subject routine *SR* and static SQL argument list *SAL*.

3) *SR* shall be an SQL-invoked function.

## Access Rules

*No additional Access Rules.*

## General Rules

1) The General Rules of Subclause 7.1, "<routine invocation>", are applied with *SR* as *SUBJECT ROUTINE* and *SAL* as *STATIC SQL ARG LIST*, yielding value *V* that is the result of the <routine invocation>.

2) The value of <function clause> is *V*.

## Profile EntryInfo Properties

— **SQL String** — Rewritten in JDBC-specified function call syntax with all <embedded Java expression>s replaced by <dynamic parameter specification>.

```
{ ? = CALL <routine invocation> }
```

— **Role** — VALUES

— **Parameter Java fields** — Describes both the <Lval expression> of the return result and all <embedded Java expression>s appearing in the <function clause>.

- **Param Count** — Gives the number of <embedded Java expression>s appearing in the <function clause>, plus one for the return result.
- **Param Info** — Returns a TypeInfo object describing a particular <embedded Java expression>. The <Lval expression> of the containing <assignment clause> is the first TypeInfo object returned, at

index 1 (one). The *i*-th TypeInfo object describes the *i*-th <embedded Java expression> appearing in the original <assignment clause> (or, equivalently, the *i*-th <dynamic parameter specification> in the SQL String Java field), where *i* is a one-based index.

## Code Generation

1) All <embedded Java expression>s are bound according to the rules specified for <executable clause> with the exception that the parameter index is increased by 1 (one).

2) If **ECtx** has a batch context object **BC** with a pending statement batch and one or more of the following conditions are false:

   a) **ECtx** has batching enabled, as defined by the `isBatching()` method.

   b) The statement is batchable, as defined by the `isBatchable()` method.

   c) The statement is batch compatible, as defined by the `isBatchCompatible()` method.

   then the pending statement batch is executed on the batch context object using the method `executeBatch()`.

```
BC.executeBatch();
```

3) If batching is enabled on **ECtx**, as defined by `isBatching()`, and the statement is batchable, as defined by `isBatchable()`, then the statement is placed into a batch context object which becomes the current batch context object.

```
BC = stmt.getBatchContext();
```

4) The statement is executed using executeUpdate:

```
stmt.executeUpdate ( ) ;
```

5) Let **RE** represent the <Lval expression> on the left hand side of the <assignment clause>.

6) Let **RT** represent the <java datatype> of **RE**.

7) Let **GM** represent the getter method corresponding to **JT**, as given in Table 2, "SQLJ type properties".

8) Case:

   a) If **GM** is getObject, then **RE** is fetched from the statement using getObject and the compile-time class of **RT**:

```
RE = stmt.getObject ( 1, RT.class ) ;
```

   b) Otherwise, **RE** is fetched from the statement using **GM**:

```
RE = stmt.GM ( 1 ) ;
```

9) A call to executeComplete defines the end of the statement method invocations. It is called even if an exception occurs in an earlier step:

```
stmt.executeComplete ( ) ;
```



 

## Conformance Rules

1) Without Feature J006, “Assignment Function statement”, conforming SQL language shall not contain a <function clause>.

# 11.26 <iterator conversion clause>

## Function

Specify the conversion of a `java.sql.ResultSet` object into a strongly-typed iterator object.

## Format

```
<iterator conversion clause> ::=
  CAST <result set expression>

<result set expression> ::=
  <embedded Java expression>
```

## Syntax Rules

*None.*

## Access Rules

*None.*

## General Rules

*None.*

## Definitions and Rules

1) The <java datatype> of <result set expression> shall implement the interface java.sql.ResultSet.
2) After an iterator conversion, the result of further calls to the ResultSet object given by the <result set expression> are implementation-defined.
3) Closing the iterator assigned by the <Lval expression> will also close the ResultSet of the <result set expression>.
4) Let **RE** represent the runtime value of <result set expression>.
5) Let **j** represent the number of columns contained in **RE**.
6) Let **IT** represent the iterator type of the <Lval expression>.
7) Case:
   a) If **IT** is of type <positioned iterator>, then:
      i) Let **k** represent the cardinality of the <java type list> of the associated iterator.
      ii) If **k ≠ j**, then an SQLException condition is thrown: *OLB-specific error — invalid number of columns*.



 

iii) Let **i** represent a variable ranging from 1 (one) to **k**.

iv) For each <java datatype> **JD** in <java type list>, let **i** be its associated range variable.

1) Let **ST** represent the SQL datatype of the *i*-th column of **RE**.

2) Let **JT** represent the Java datatype of $\mathbf{JD_i}$.

b) If **IT** is of type <named iterator>, then:

i) Let **k** represent the cardinality of the <java pair list>.

ii) If **k** is greater than **j**, then an SQLException condition is thrown: *OLB-specific error — invalid number of columns*.

iii) Let **SLN** represent the name of an associated column of **RE**.

iv) Let **i** represent a variable ranging from 1 (one) to **k**.

v) Let **n** represent a variable ranging from 1 (one) to **j**.

vi) For each <java pair> **JP** in <java pair list>, let **i** be its associated range variable.

1) Let **n** be the index of the first column whose **SLN** is a case-insensitive match of the <java id> associated with $\mathbf{JP_i}$, or 0 (zero) if no such **SLN** exists.

2) If **n** is 0 (zero), then an SQLException condition is thrown: *OLB-specific error — invalid number of columns*.

3) Let **JT** represent the Java datatype of $\mathbf{JP_i}$.

4) Let **ST** represent the SQL datatype of the *n*-th column of **RE**.

c) **ST** shall be SQLJ output assignable to **JT**.

8) The constructor method of the corresponding <iterator declaration clause> shall be in scope.

## Profile EntryInfo Properties

— **SQL String** — Default as described in Subclause 4.3.5.1, “EntryInfo overview”

— **Role** — ITERATOR_CONVERSION

— Execute Type — EXECUTE_UPDATE

— Statement Type — CALLABLE_STATEMENT

— **Parameter Java fields** — Describes both the <Lval expression> of the return result and the <result set expression> appearing in the <iterator conversion clause>

- **Param Count** — 2
- **Param Info** — Returns a TypeInfo object describing a particular <embedded Java expression>. The <Lval expression> is the first TypeInfo object returned, at index 1 (one). The <result set expression> is the second TypeInfo object returned, at index 2.

## Code Generation

1) Let **RE** represent the <Lval expression> on the left hand side of the <assignment clause>.

2) Let **RT** represent the <java datatype> of **RE**.

3) Let **HE** represent the <result set expression> of the <iterator conversion clause>.

4) **HE** is bound to the statement using setObject:

```
stmt.setObject(2, HE);
```

5) If **ECtx** has a batch context object **BC** with a pending statement batch and one or more of the following conditions are false:

   a) **ECtx** has batching enabled, as defined by the `isBatching()` method.

   b) The statement is batchable, as defined by the `isBatchable()` method.

   c) The statement is batch compatible, as defined by the `isBatchCompatible()` method.

   then the pending statement batch is executed on the batch context object using the method `executeBatch()`.

```
BC.executeBatch();
```

6) If batching is enabled on **ECtx**, as defined by `isBatching()`, and the statement is batchable, as defined by `isBatchable()`, then the statement is placed into a batch context object which becomes the current batch context object.

```
BC = stmt.getBatchContext();
```

7) The statement is executed using executeUpdate:

```
stmt.executeUpdate();
```

8) **RE** is fetched from the statement using getObject and the compile-time class of **RT**:

```
RE = stmt.getObject(1, RT.class);
```

9) A call to executeComplete defines the end of the statement method invocations. It is called even if an exception occurs in an earlier step.

```
stmt.executeComplete();
```

## Conformance Rules

*None.*



 

## 11.27 <compound statement>

*This Subclause modifies Subclause 15.1, "<compound statement>", in ISO/IEC 9075-4.*

### Function

Specify a statement that groups other statements together.

### Format

```
<compound statement> ::=
  BEGIN { <SQL procedure statement> <semicolon> }... END
```

### Syntax Rules

*No additional Syntax Rules.*

### Access Rules

*No additional Access Rules.*

### General Rules

*No additional General Rules.*

### Definitions and Rules

1) <compound statement> is permitted to appear in a <statement clause>. It consists of one or more <SQL procedure statement>s (*i.e.* any of the SQL Constructs that are permitted to appear in a <statement clause>, except for a <compound statement>), terminated by semicolons, sandwiched between BEGIN and END:

```
#sql { BEGIN
         INSERT INTO RAIN (MONTH, RAINFALL) VALUES (:x, :y);
         SELECT MAX(RAINFALL) INTO :z FROM RAIN WHERE MONTH - :x;
       END };
```

2) SQLJ follows the SQL/PSM rules for the semantics of blocks in which a contained statement raises an exception and in which a host variable is referenced in multiple statements.

3) If an <embedded Java expression> containing an <Lval expression> has either an implicit or explicit <parameter mode> of OUT or INOUT in a given <SQL procedure statement> then let *LV* denote the location of the <Lval expression>. If another <embedded Java expression> containing an <Lval expression> has either an implicit or explicit <parameter mode> of IN or INOUT in a subsequent <SQL procedure statement> and the location of the <Lval expression> is *LV*, then the value of the <Lval expression> is implementation-defined.

## Profile EntryInfo Properties

— **SQL String** — Default as defined in Subclause 4.3.5.1, “EntryInfo overview”.

— **Role** — BLOCK

## Conformance Rules

1) Without Feature J007, “Compound statement”, conforming SQL language shall not contain a <compound statement>.



 

# 12 Package sqlj.runtime

## 12.1 Overview

The **sqlj.runtime** package defines the runtime classes and interfaces that are available to an SQLJ program. It includes utility classes such as AsciiStream that are used directly, and interfaces such as ResultSetIterator that appear as part of a generated class declaration.

## 12.2 SQLJ runtime interfaces

### 12.2.1 sqlj.runtime.ConnectionContext

#### 12.2.1.1 Interface Overview

```
public interface ConnectionContext
```

The ConnectionContext interface provides a set of methods that manage a set of SQL-statements performed during an SQL-session. A connection context object maintains a `java.sql.Connection` object on which dynamic SQL-statements are permitted to be performed. It also contains a default ExecutionContext object by which SQL-statement execution semantics are permitted to be queried and modified.

In addition to those methods defined by this interface, each concrete implementation *UserCtx* of a connection context object shall provide the following methods:

— Returns a profile key for a particular profile.Loader object and profile name:

```
public static Object getProfileKey
    ( sqlj.runtime.profile.Loader l,
      String profileName ) throws SQLException ;
```

— Returns a top level profile object for a particular profile key:

```
public static sqlj.runtime.profile.Profile getProfile ( Object key ) ;
```

— Returns the default connection context object for the UserCtx class:

```
public static UserCtx getDefaultContext ( ) ;
```

— Sets the default connection context object for the UserCtx class:

```
public static void setDefaultContext ( UserCtx dflt ) ;
```

— Constructs a connection context object:

```
public UserCtx ( ConnectionContext other )
       throws SQLException;
public UserCtx ( java.sql.Connection conn)
       throws SQLException;
```

If the connection context is specified to support <url constructors>, then:

```
public UserCtx ( String url, String user, String pwd, boolean autoCommit )
       throws SQLException ;
public UserCtx ( String url, Properties info, boolean autoCommit )
       throws SQLException ;
public UserCtx ( String url, boolean autoCommit )
       throws SQLException ;
```

If the connection context is specified to support <data source constructors>, then:

```
public UserCtx ( )
       throws SQLException ;
public UserCtx ( String user, String password )
       throws SQLException ;
```

Note that an invocation of UserCtx causes either a new `java.sql.Connection` object to be created or an existing Connection (or ConnectionContext) object to be reused. If the invocation causes an exception to be thrown, then the Connection object is closed only if the invocation caused it to be created.

Note that, for any UserCtx constructor that creates a `java.sql.Connection` object during construction, that Connection object will be automatically closed if the constructor call throws an exception. For any UserCtx constructor that uses an already opened `java.sql.Connection` object (or connection context object) passed from the client, that Connection object (or connection context object) will remain open even if the constructor call throws an exception.

#### 12.2.1.2 Variables

##### 12.2.1.2.1 CLOSE_CONNECTION

```
public static final boolean CLOSE_CONNECTION = true;
```

The underlying `java.sql.Connection` object should be closed.

## See Also

— Subclause 12.2.1.3.1, “close ()”

— Subclause 12.2.1.3.2, “close (boolean)”



 

#### 12.2.1.2.2 KEEP_CONNECTION

```
public static final boolean KEEP_CONNECTION = false;
```

The underlying `java.sql.Connection` object should not be closed.

## See Also

— Subclause 12.2.1.3.1, "close ()"

— Subclause 12.2.1.3.2, "close (boolean)"

### 12.2.1.3 Methods

#### 12.2.1.3.1 close ()

```
public abstract void close ( ) throws SQLException
```

Releases all resources used in maintaining SQL-session state on this connection context object, closes any open ConnectedProfile objects, and closes the underlying `java.sql.Connection` object. This method is equivalent to `close(CLOSE_CONNECTION)`.

## Throws

— SQLException — if unable to close the connection context object

## See Also

— Subclause 12.2.1.3.2, "close (boolean)"

#### 12.2.1.3.2 close (boolean)

```
public abstract void close ( boolean closeConnection )
    throws SQLException
```

Releases all resources used in maintaining SQL-session state on this connection context object and closes any open ConnectedProfile objects managed by this connection context object. Since the underlying `java.sql.Connection` object managed by this connection context object is permitted to be shared between multiple connection context objects, it is not always desirable to close the underlying `java.sql.Connection` object when `close()` is called. If the constant KEEP_CONNECTION is passed, the underlying `java.sql.Connection` object is not closed. Otherwise, if the constant CLOSE_CONNECTION is passed, the underlying Connection object is closed.

NOTE 29 — A connection context object is automatically closed at the time it is garbage-collected. A connection context object closed in such a way does not close the underlying `java.sql.Connection` object since it will also be automatically closed at the time it is garbage-collected.

## Parameters

— closeConnection — is CLOSE_CONNECTION if the underlying Connection object should also be closed

## Throws

— SQLException — if unable to close the connection context object

## See Also

— Subclause 12.2.1.2.1, "CLOSE_CONNECTION"

— Subclause 12.2.1.2.2, "KEEP_CONNECTION"

### 12.2.1.3.3 getConnectedProfile (Object)

```
public abstract ConnectedProfile getConnectedProfile ( Object profileKey )
       throws SQLException
```

Each connection context object maintains a set of ConnectedProfile objects on which SQL-statements are prepared. Collectively, the set of ConnectedProfile objects contained in a connection context object represent the set of all possible SQL-statements that are permitted to be performed between the time that this connection context object is created and the time that it is destroyed.

The profileKey object shall be an object that was returned via a prior call to `getProfileKey()`. An exception is thrown if a ConnectedProfile object could not be created for this connection context object.

## Parameters

— profileKey — the key associated with the desired profile object

## Returns

— The ConnectedProfile object associated with a profileKey for this connection context object.

## Throws

— SQLException — if the ConnectedProfile object could not be created

— IllegalArgumentException — if the profileKey is null or invalid



 

#### 12.2.1.3.4 getConnection ()

```
public abstract Connection getConnection ( )
```

Note that, depending on construction, the returned Connection object might be shared between many connection context objects.

## Returns

— The underlying `java.sql.Connection` object associated with this connection context object.

#### 12.2.1.3.5 getExecutionContext ()

```
public abstract ExecutionContext getExecutionContect ( )
```

The default execution context object is the execution context object used if no explicit context object is supplied during the execution of a particular SQL-statement.

The returned default ExecutionContext object refers to the default ExecutionContext object in this connection context object and, as such, any changes made to the returned object are visible in the connection context object.

## Returns

— The default execution context object used by this connection context object.

#### 12.2.1.3.6 getTypeMap ()

```
public abstract Map getTypeMap ( )
```

If the <connection declaration clause> contains a <declaration with clause> that specifies the <predefined connection with keyword> **typeMap**, then let **TM** be the corresponding <with value>. The invocation of the method `getTypeMap( )` returns an instance of a class that implements `java.util.Map` that contains the user-defined type mapping information provided by the properties files listed in **TM** in the form specified in [JDBC]. If the <connection declaration clause> does not contain a <declaration with clause> that specifies **typeMAP**, then this method returns Java null.

## Returns

— The user-defined tyep map associated with the ConnectionContext in the format specified in [JDBC], or Java null if there is no associated type map.

#### 12.2.1.3.7 isClosed ()

```
public abstract boolean isClosed ( )
```

Returns **true** if this execution context object has been closed; otherwise, returns **false**.

## Returns

— If this execution context object has been closed, then **true**; otherwise, **false**.

### 12.2.2 sqlj.runtime.ForUpdate

#### 12.2.2.1 Interface Overview

```
public interface ForUpdate
```

An interface implemented by iterator classes whose instances will be used in a positioned update or delete statement (as parameter to a WHERE CURRENT OF clause). The class of every iterator object that is to be passed as a parameter to a WHERE CURRENT OF clause shall implement this interface.

#### 12.2.2.2 Methods

#### 12.2.2.2.1 getCursorName ()

```
public abstract String getCursorName ( ) throws SQLException
```

Get the name of the implicit SQL cursor used by this iterator.

In SQL, a result table is retrieved through a named cursor. The current row of a result can be updated or deleted using a positioned update or delete statement that references the cursor name.

SQLJ supports this SQL feature by providing the name of the implicit cursor used by an iterator. The current row of an iterator is also the current row of this implicit cursor.

NOTE 30 — If positioned update is not supported, then an SQLException is thrown.

## Returns

— The iterator's SQL cursor name.



 

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 12.2.3 sqlj.runtime.NamedIterator

```
public interface NamedIterator
    extends ResultSetIterator
```

An interface implemented by all iterators that employ a strategy of binding by name. All such iterators depend on the name of the columns of the data to which they are bound, as opposed to the order of the columns to which they are bound.

In addition to implementing this interface, classes that implement the NamedIterator interface shall provide:

— A public constructor that, when invoked, takes an RTResultSet object as an argument. If the construction of a named iterator results in an exception being thrown, it is assumed that the iterator automatically closes the underlying RTResultSet object. This only applies to exceptions thrown during construction.

— A named accessor method for each <java id> appearing in the <java pair list> of the <iterator declaration clause> that declared the current iterator. Each named accessor method uses as its name an exact case-matching copy of its <java id>. Using case-insensitive comparison, the name of the accessor method is equal to the name of its associated result column.

— Once `next()` has returned **false**, the behaviour of any named accessor method is implementation-dependent.

### 12.2.4 sqlj.runtime.PositionedIterator

#### 12.2.4.1 Interface Overview

```
public interface PositionedIterator
    extends ResultSetIterator
```

An interface implemented by all iterators that employ a by position binding strategy. All such iterators depend on the position of the columns of the data to which they are bound, as opposed to the names of the columns to which they are bound.

In addition to implementing this interface, classes that implement the PositionedIterator interface shall provide:

— A public constructor that, when invoked, takes an RTResultSet object as an argument. If the construction of a positioned iterator results in an exception being thrown, it is assumed that the iterator automatically closes the underlying RTResultSet object. This only applies to exceptions thrown during construction.

— A positioned accessor method for each column in the expected result. The name of the positioned accessor method for the *N*-th column will be getCol*N*.

#### 12.2.4.2 Methods

##### 12.2.4.2.1 endFetch ()

```
public abstract boolean endFetch ( ) throws SQLException
```

This method is used to determine the success of a `FETCH...INTO` statement; it returns **true** if the last attempt to fetch a row failed, and returns **false** if the last attempt was successful. Rows are attempted to be fetched when the `next()` method is called (which is called implicitly during the execution of a `FETCH...INTO` statement).

NOTE 31 — If `next()` has not yet been called, this method returns **true**.

## Returns

— If the iterator is not positioned on a row, then **true**; otherwise, **false**.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 12.2.5.3.9, “next ()”

### 12.2.5 sqlj.runtime.ResultSetIterator

#### 12.2.5.1 Interface Overview

```
public interface ResultSetIterator
```

An interface that defines the shared functionality of those objects used to iterate over the contents of an iterator.

#### 12.2.5.2 Variables

##### 12.2.5.2.1 ASENSITIVE

```
public static final int ASENSITIVE = 1;
```

Constant used by the “sensitivity” Java field, indicating that the iterator is defined to have an asensitive cursor.



 

#### 12.2.5.2.2 FETCH_FORWARD

```
public static final int FETCH_FORWARD = java.sql.ResultSet.FETCH_FORWARD;
```

Constant used by **sqlj.runtime.Scrollable.setFetchDirection**, **sqlj.runtime.ExecutionContext.getFetchDirection**, and **sqlj.runtime.ExecutionContext.setFetchDirection** to indicate that the rows in an iterator object will be processed in a forward direction, first-to-last.

#### 12.2.5.2.3 FETCH_REVERSE

```
public static final int FETCH_REVERSE = java.sql.ResultSet.FETCH_REVERSE;
```

Constant used by **sqlj.runtime.Scrollable.setFetchDirection**, **sqlj.runtime.ExecutionContext.getFetchDirection**, and **sqlj.runtime.ExecutionContext.setFetchDirection** to indicate that the rows in an iterator object will be processed in a reverse direction, last-to-first.

#### 12.2.5.2.4 FETCH_UNKNOWN

```
public static final int FETCH_UNKNOWN = java.sql.ResultSet.FETCH_UNKNOWN;
```

Constant used by **sqlj.runtime.Scrollable.setFetchDirection**, **sqlj.runtime.ExecutionContext.getFetchDirection**, and **sqlj.runtime.ExecutionContext.setFetchDirection** to indicate that the order in which rows in an iterator object will be processed is unknown.

#### 12.2.5.2.5 INSENSITIVE

```
public static final int INSENSITIVE = 2;
```

Constant used by the “sensitivity” Java field, indicating that the iterator is defined to have an insensitive cursor.

#### 12.2.5.2.6 SENSITIVE

```
public static final int SENSITIVE = 3;
```

Constant used by the “sensitivity” Java field, indicating that the iterator is defined to have a sensitive cursor.

### 12.2.5.3 Methods

NOTE 32 — Once method `isClosed()` has returned **true**, the behaviour of any other method on that iterator is implementation-dependent.

#### 12.2.5.3.1 clearWarnings ()

```
public abstract void clearWarnings ( ) throws SQLException
```

After this call, getWarnings returns null until a new warning is reported for this iterator.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 12.2.5.3.2 close ()

```
public abstract void close ( ) throws SQLException
```

Closes the iterator object, releasing any underlying resources. It is recommended that iterators be explicitly closed as soon as they are no longer needed, to allow for the immediate release of resources that are no longer needed.

NOTE 33 — If it is not already closed, an iterator is automatically closed when it is destroyed.

## Throws

— SQLException — if there is a problem closing the iterator

## See Also

— Subclause 12.2.5.3.8, “isClosed ()”

#### 12.2.5.3.3 getFetchSize ()

```
synchronized public int getFetchSize ( ) throws SQLException
```

Retrieves the number of rows that is the current fetch size for this iterator object. If this iterator object has not set a fetch size by calling the method **setFetchSize**, or has set a fetch size of 0 (zero), then the value returned is implementation-dependent.

## Returns

— The current fetch size for the iterator object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.



 

#### 12.2.5.3.4 getResultSet ()

```
public abstract ResultSet getResultSet ( ) throws SQLException
```

Returns the `java.sql.ResultSet` object associated with this iterator. The produced `java.sql.ResultSet` object shall have normal JDBC functionality, as defined in [JDBC] (in particular, SQL null values fetched with JDBC positional column access methods will not raise an SQLNullException). This method is provided to facilitate interoperability with JDBC.

If support for Feature J002, "ResultSetIterator access to JDBC ResultSet" is provided, then any synchronization between the iterator and the produced `java.sql.ResultSet` object is implementation-defined.

NOTE 34 — For maximum portability, this method should be invoked before the first `next()` method invocation on the iterator. Once the `java.sql.ResultSet` object has been produced, all operations to fetch data should be through the `java.sql.ResultSet` object.

## Returns

— A `java.sql.ResultSet` object for this iterator.

## Throws

— SQLException: *OLB-specific error — unsupported feature* — if support for Feature J002, "ResultSetIterator access to JDBC ResultSet", is not provided.

#### 12.2.5.3.5 getRow ()

```
synchronized public int getRow ( ) throws SQLException
```

Retrieves the current row number. The first row is number 1, the second is number 2, and so on.

## Returns

— If there is no current row, then 0 (zero); otherwise, the number of the current row.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 12.2.5.3.6 getSensitivity ()

```
synchronized public int getSensitivity ( ) throws SQLException
```

Retrieves the sensitivity of this iterator object. The sensitivity is determined by the <iterator declaration clause> and by the SQLJ runtime implementation that created the iterator object.

## Returns

— Case:

- If this iterator object was declared with the <predefined iterator with keyword> **sensitivity** and a corresponding <with value> **SENSITIVE**, and the SQLJ runtime that created this iterator object supports sensitive iterators, then `ResultSetIterator.SENSITIVE`.
- If this iterator object was declared with the <predefined iterator with keyword> **sensitivity** and a corresponding <with value> **INSENSITIVE**, and the SQLJ runtime that created this iterator object supports insensitive iterators, then `ResultSetIterator.INSENSITIVE`.
- If this iterator object was declared with the <predefined iterator with keyword> **sensitivity** and a corresponding <with value> **ASENSITIVE**, then `ResultSetIterator.ASENSITIVE`.
- Otherwise, an implementation-dependent value.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 12.2.5.3.7 getWarnings ()

```
public abstract SQLWarning getWarnings ( ) throws SQLException
```

The first warning reported by calls on this iterator is returned. Subsequent iterator warnings will be chained to this SQLWarning.

The warning chain is automatically cleared each time the iterator object is advanced to the next row.

NOTE 35 — This warning chain only covers warnings caused by iterator methods. Any warning caused by statement execution (such as fetching **OUT** parameters) will be chained on the ExecutionContext object.

## Returns

— If there are no errors, then null; otherwise, the first SQLWarning.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.



 

#### 12.2.5.3.8 isClosed ()

```
public abstract boolean isClosed ( ) throws SQLException
```

## Returns

— If this iterator has been closed, then **true**; otherwise, **false**

## Throws

— SQLException — if an error occurs determining the close status of the iterator.

## See Also

— Subclause 12.2.5.3.2, "close ()"

#### 12.2.5.3.9 next ()

```
public abstract boolean next ( ) throws SQLException
```

Advances the iterator to the next row. At the beginning, the iterator is positioned before the first row.

NOTE 36 — A **`FETCH...INTO`** statement performs an implicit invocation of `next()` on the iterator passed.

## Returns

— If there was a next row in the iterator, then **true**; otherwise, **false**.

## Throws

— SQLException — if an exception occurs while changing the position of the iterator

#### 12.2.5.3.10 setFetchSize (int)

```
synchronized public void setFetchSize (int rows ) throws SQLException
```

Gives the SQLJ runtime a hint as to the number of rows that should be fetched when more rows are needed from this iterator object. If the value specified is zero, then the runtime is free to choose an implementation-dependent fetch size.

## Parameters

— rows — the default fetch size for result sets generated from this iterator object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition, or the condition 0 (zero) ≤ rows ≤ **ECtxt**.`getMaxRows()` is not satisfied, where **ECtxt** is the ExecutionContext object that was used to create this iterator object.

## Conformance Rules

*None.*



 

### 12.2.6 sqlj.runtime.Scrollable

#### 12.2.6.1 Interface Overview

```
public interface Scrollable
```

This interface provides a set of methods that all scrollable iterator objects support. The effect of an update on a Scrollable iterator object is implementation-defined.

#### 12.2.6.2 Variables

None.

#### 12.2.6.3 Methods

##### 12.2.6.3.1 absolute (int)

```
public abstract boolean absolute ( int row ) throws SQLException
```

Moves the iterator object to the row with the given row number.

If the row number is positive, the iterator object moves to the row with the given row number with respect to its beginning. The first row is row 1, the second is row 2, and so on.

If the given row number is negative, the iterator object moves to an absolute row position with respect to its end. For example, calling `absolute(-1)` positions the iterator object on the last row, `absolute(-2)` indicates the next-to-last row, and so on.

An attempt to position the iterator object beyond its first or last row leaves the iterator object before or after its first or last row, respectively.

NOTE 37 — Calling `absolute(1)` is the same as calling `first()`. Calling `absolute(-1)` is the same as calling `last()`.

## Returns

— If the iterator object is on a row, then **true**; otherwise, **false**

## Throws

— SQLException — if the SQL-implementation raises an exception condition, or row is 0 (zero).

#### 12.2.6.3.2 afterLast ()

```
public abstract void afterLast ( ) throws SQLException
```

Moves the iterator object to immediately after its last row. Has no effect if the iterator object contains no rows.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 12.2.6.3.3 beforeFirst ()

```
public abstract void beforeFirst ( ) throws SQLException
```

Moves the iterator object to immediately before its first row. Has no effect if the iterator object contains no rows.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 12.2.6.3.4 first ()

```
public abstract boolean first ( ) throws SQLException
```

Moves the iterator object to its first row.

## Returns

— If the iterator object is on a row, then **true**; If there are no rows, then **false**.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.



 

#### 12.2.6.3.5 getFetchDirection ()

```
public abstract int getFetchDirection ( ) throws SQLException
```

Retrieves the direction for fetching rows for this iterator object. If this iterator object has not set a fetch direction by calling the method `setFetchDirection()`, then the value returned is the default specified in Subclause 12.2.6.3.13, "setFetchDirection (int)".

### Returns

— The fetch direction for this iterator object.

### Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 12.2.6.3.6 isAfterLast ()

```
public abstract boolean isAfterLast ( ) throws SQLException
```

Indicates whether the iterator object is after its last row.

### Returns

— If the iterator object is positioned after its last row, then **true**; otherwise **false**. Returns **false** when the iterator object contains no rows.

### Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 12.2.6.3.7 isBeforeFirst ()

```
public abstract boolean isBeforeFirst ( ) throws SQLException
```

Indicates whether the iterator object is before its first row.

### Returns

— If the iterator object is positioned before its first row, then **true**; otherwise **false**. Returns **false** when the iterator object contains no rows.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 12.2.6.3.8 isFirst ()

```
public abstract boolean isFirst ( ) throws SQLException
```

Indicates whether the iterator object is on its first row.

## Returns

— If the iterator object is positioned on its first row, then **true**; otherwise **false**. Returns **false** when the iterator object contains no rows.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 12.2.6.3.9 isLast ()

```
public abstract boolean isLast ( ) throws SQLException
```

Indicates whether the iterator object is on its last row.

NOTE 38 — Invocation of the method `isLast()` may be expensive, because the SQLJ driver might need to fetch ahead one row in order to determine whether the current row is the last row.

## Returns

— If the iterator object is positioned on its last row, then **true**; otherwise **false**. Returns **false** when the iterator object contains no rows.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 12.2.6.3.10 last ()

```
public abstract boolean last ( ) throws SQLException
```

Moves the iterator object to its last row.



 

## Returns

— If the iterator object is positioned on a row, then **true**; otherwise **false**. Returns **false** when the iterator object contains no rows.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 12.2.6.3.11 previous ()

```
public abstract boolean previous ( ) throws SQLException
```

Moves the iterator object to its previous row.

## Returns

— If the iterator object is positioned on a row, then **true**; if it is positioned before its first row or after its last row, then **false**. Returns **false** when the iterator object contains no rows.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 12.2.6.3.12 relative (int)

```
public abstract boolean relative ( int rows ) throws SQLException
```

Moves the iterator object the given number of rows, either positive or negative, from its current position. Attempting to move beyond its first or last row positions the iterator object before or after its first or last row, respectively. Invoking `relative(0)` is valid, but does not change the iterator object position.

## Returns

— If the iterator object is positioned on a row, then **true**; **false** otherwise. Returns **false** when the iterator object contains no rows.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 12.2.6.3.13 setFetchDirection (int)

```
public abstract void setFetchDirection (int direction ) throws SQLException
```

Gives the SQLJ runtime a hint as to the direction in which rows of this iterator object are processed. The default value is `sqlj.runtime.ResultSetIterator.FETCH_FORWARD`.

## Parameters

— direction — the initial direction for processing rows.

## Throws

— SQLException — if the SQL-implementation raises an exception condition, or the given direction is not one of `ResultSetIterator.FETCH_FORWARD`, `ResultSetIterator.FETCH_REVERSE`, or `ResultSetIterator.FETCH_UNKNOWN`.

## Conformance Rules

*None.*



 

# 12.3 SQLJ Runtime Classes

## 12.3.1 sqlj.runtime.AsciiStream

### 12.3.1.1 Class Overview

```
java.lang.Object
   |
   +--java.io.InputStream
           |
           +--java.io.FilterInputStream
                   |
                   +--sqlj.runtime.StreamWrapper
                           |
                           +--sqlj.runtime.AsciiStream
```

```
public class AsciiStream
    extends StreamWrapper
```

**AsciiStream** (sqlj.runtime.AsciiStream) is a class derived from java.io.InputStream. The octets comprising an AsciiStream object are interpreted as ASCII characters. In order to process an InputStream object as an input argument to an SQLJ executable clause, an SQLJ implementation has to know both its length and the way to interpret its octets. Therefore, an InputStream object cannot be passed directly, but rather shall be an instance of AsciiStream, BinaryStream, or UnicodeStream.

## See Also

— Subclause 12.3.2, “sqlj.runtime.BinaryStream”

— Subclause 12.3.7, “sqlj.runtime.UnicodeStream”

### 12.3.1.2 Constructors

#### 12.3.1.2.1 AsciiStream (InputStream)

```
public AsciiStream ( InputStream in )
```

Creates an ASCII-valued InputStream object with an uninitialized length.

NOTE 39 — The length Java field shall be set via a call to `setLength()` before an AsciiStream object is substituted for an input (or inout) parameter in an invocation of an SQL-statement.

## Parameters

— **IN** — the InputStream object to interpret as an AsciiStream object.

#### 12.3.1.2.2 AsciiStream (InputStream, int)

```
public AsciiStream ( InputStream in, int length )
```

Creates an ASCII-valued InputStream object of given length.

## Parameters

— **IN** — the InputStream object to interpret as an AsciiStream object.

— **length** — the length in octets of the AcsiiStream object.

### 12.3.2 sqlj.runtime.BinaryStream

#### 12.3.2.1 Class Overview

```
java.lang.Object
   |
   +--java.io.InputStream
           |
           +--java.io.FilterInputStream
                   |
                   +--sqlj.runtime.StreamWrapper
                           |
                           +--sqlj.runtime.BinaryStream

public class BinaryStream
    extends StreamWrapper
```

**BinaryStream** (sqlj.runtime.BinaryStream) is a class derived from java.io.InputStream. The octets comprising a BinaryStream object are not interpreted as characters. In order to process an InputStream object as an input argument to an SQLJ executable clause, an SQLJ implementation has to know both its length and the way to interpret its octets. Therefore, an InputStream object cannot be passed directly, but rather shall be an instance of AsciiStream, BinaryStream, or UnicodeStream.

## See Also

— Subclause 12.3.1, “sqlj.runtime.AsciiStream”

— Subclause 12.3.7, “sqlj.runtime.UnicodeStream”



 

#### 12.3.2.2 Constructors

##### 12.3.2.2.1 BinaryStream (InputStream)

```
public BinaryStream ( InputStream in )
```

Creates a Binary-valued InputStream object with an uninitialized length.

NOTE 40 — The length Java field shall be set via a call to `setLength()` before a BinaryStream object is substituted for an input (or inout) parameter in an invocation of an SQL-statement.

## Parameters

— **IN** — the InputStream object to interpret as a BinaryStream object.

##### 12.3.2.2.2 BinaryStream (InputStream, int)

```
public BinaryStream ( InputStream in, int length )
```

Creates a binary valued InputStream object of given length.

## Parameters

— **IN** — the InputStream object to interpret as a BinaryStream object.

— **length** — the length in octets of the BinaryStream object.

### 12.3.3 sqlj.runtime.DefaultRuntime

#### 12.3.3.1 Class Overview

```
java.lang.Object
   |
   +--sqlj.runtime.RuntimeContext
           |
           +--sqlj.runtime.DefaultRuntime

public class DefaultRuntime
    extends RuntimeContext
```

The DefaultRuntime class implements the expected runtime behavior defined by the abstract RuntimeContext class for most Java Virtual Machine environments.

#### 12.3.3.2 Constructors

##### 12.3.3.2.1 DefaultRuntime ()

```
public DefaultRuntime ( )
```

#### 12.3.3.3 Methods

##### 12.3.3.3.1 getDefaultConnection ()

```
public Connection getDefaultConnection ( )
```

The default data source defined in JNDI is used to establish the default connection. If no such data source is defined or the connection cannot be established, then null is returned.

## Returns

— If the default data source does not exist or cannot establish a connection, then null; otherwise, a default Connection object, as defined by the default data source.

## Overrides

— `getDefaultConnection()` in class RuntimeContext

## See Also

— Subclause 12.3.5.2.1, "DEFAULT_DATA_SOURCE"

##### 12.3.3.3.2 getLoaderForClass (Class)

```
public Loader getLoaderForClass ( Class forClass )
```

Creates and returns a default Loader object that uses the class loader of the given class.

## Parameters

— **forClass** — the class with which the resulting Loader object is to be associated.



 

## Returns

— A default Loader object for the given class.

## Overrides

— `getLoaderForClass()` in class RuntimeContext

## See Also

— Subclause 13.3.1, “sqlj.runtime.profile.DefaultLoader”

### 12.3.4 sqlj.runtime.ExecutionContext

#### 12.3.4.1 Class Overview

```
java.lang.Object
   |
   +--sqlj.runtime.ExecutionContext

public class ExecutionContext
    extends Object
```

An ExecutionContext object provides the execution context in which SQLJ executable clauses are performed. An execution context object contains a number of operations for execution control, execution status, and execution cancellation. Execution control operations modify the semantics of subsequent SQL-statements executed on this execution context object. Execution status operations describe the results of the last SQL-statement executed on this execution context object. Execution cancellation methods terminate the currently executing SQL-statement on this execution context object.

NOTE 41 — Concurrently executing SQL-statements are expected to use distinct execution context objects. The execution context class implementing the ExecutionContext interface is not expected to support multiple SQL-statements executing with the same execution context object. The client is responsible for ensuring the proper creation of distinct execution context objects where needed, or synchronizing the execution of operations on a particular execution context object. It is also assumed that generated calls to methods on this class appear within a synchronized block to avoid concurrent calls. Recursive SQL execution calls on the same connection context object are supported.

Without Feature J003, “Execution control”, if an ExecutionContext's Java fields have been set to anything other than their respective default values, with the following routines, and an attempt is made to register a statement with such an ExecutionContext, then an SQLException condition is thrown: *OLB-specific error — unsupported feature*.

— getMaxFieldSize

— setMaxFieldSize

— getMaxRows

— setMaxRows;

— getQueryTimeout

— setQueryTimeout

## See Also

— Subclause 12.2.1.3.5, “getExecutionContext ()”

### 12.3.4.2 Variables

#### 12.3.4.2.1 ADD_BATCH_COUNT

```
public static final int ADD_BATCH_COUNT = -3;
```

Constant possibly returned by `getUpdateCount` indicating that the last statement encountered was added to the existing statement batch rather than being executed.

## See Also

— Subclause 12.3.4.4.15, “getUpdateCount ()”

#### 12.3.4.2.2 AUTO_BATCH

```
public static final int AUTO_BATCH = 100;
```

Constant passed to `setBatchLimit` to indicate that implicit batch execution should be performed, and that the actual batch size is at the discretion of the SQLJ runtime implementation.

## See Also

— Subclause 12.3.4.4.21, “setBatchLimit (int)”

#### 12.3.4.2.3 EXEC_BATCH_COUNT

```
public static final int EXEC_BATCH_COUNT = -5;
```

Constant possibly returned by `getUpdateCount` indicating that the last execution was a statement batch execution.



 

## See Also

— Subclause 12.3.4.4.3, “executeBatch ()”

— Subclause 12.3.4.4.15, “getUpdateCount ()”

### 12.3.4.2.4 EXCEPTION_COUNT

```
public static final int EXCEPTION_COUNT = -2;
```

Constant possibly returned by getUpdateCount indicating that an exception was thrown before the last execution was successfully completed, or that no operation has yet been attempted on this execution context object.

## See Also

— Subclause 12.3.4.4.15, “getUpdateCount ()”

### 12.3.4.2.5 NEW_BATCH_COUNT

```
public static final int NEW_BATCH_COUNT = -6;
```

Constant possibly returned by `getUpdateCount` indicating that the last statement encountered was added to a new statement batch rather than being executed.

## See Also

— Subclause 12.3.4.4.15, “getUpdateCount ()”

### 12.3.4.2.6 QUERY_COUNT

```
public static final int QUERY_COUNT = -1;
```

Constant possibly returned by getUpdateCount indicating that the last execution produced a RTResultSet object or iterator.

## See Also

— Subclause 12.3.4.4.15, “getUpdateCount ()”

#### 12.3.4.2.7 UNLIMITED_BATCH

```
public static final int UNLIMITED_BATCH = -7;
```

Constant passed to `setBatchLimit` to indicate that no implicit batch execution should be performed upon reaching a certain batch size.

## See Also

— Subclause 12.3.4.4.21, “setBatchLimit (int)”

### 12.3.4.3 Constructors

#### 12.3.4.3.1 ExecutionContext ()

```
public ExecutionContext ( )
```

The default constructor for the ExecutionContext class.

### 12.3.4.4 Methods

#### 12.3.4.4.1 cancel ()

```
public void cancel ( ) throws SQLException
```

The `cancel()` method can be used by one thread to cancel an SQL-statement that is currently being executed by another thread using this execution context object. Note that this method has no effect if there is no RTStatement object currently being executed for this execution context object. If there is a pending statement batch on this execution context object, the statement batch is canceled and emptied.

## Throws

— SQLException — if unable to cancel

## See Also

— Subclause 12.3.4.4.5, “executeUpdate ()”

— Subclause 12.3.4.4.4, “executeQuery ()”



 

### 12.3.4.4.2 execute ()

```
public boolean execute ( ) throws SQLException
```

Prior to statement execution, if there is a pending statement batch on this execution context object and one or more of the following conditions is true:

— Batching is currently disabled on this execution context object.

— The currently registered RTStatement object is not batchable.

— The currently registered RTStatement object is not batch compatible with the pending statement batch.

then the statement batch is implicitly executed using `BatchContext.executeBatch()`.

If batching is currently enabled on this execution context object and the currently registered RTStatement object is batchable, then the statement is batched rather than executed. The pending statement batch is replaced by a statement batch that includes the currently registered RTStatement object, as returned by `RTStatement.getBatchContext()`. Note that in this case, the statement may not return side-channel `java.sql.ResultSet` objects. If the statement was added to the existing statement batch, then the update count is set to `ADD_BATCH_COUNT`. Otherwise, if the statement was added to a new statement batch, then the update count is set to `NEW_BATCH_COUNT`.

Otherwise, a generic execute is performed on the currently registered RTStatement object. If a new statement batch is created as a result of executing the current RTStatement object, the current statement batch (if any) is implicitly executed. Under some situations, a single SQL CALL statement might return multiple `java.sql.ResultSet` objects. The `execute()`, `getNextResultSet()`, and `getNextResultSet(int)` methods allow navigation through multiple `java.sql.ResultSet` objects.

The `execute()` method executes the currently registered RTStatement object and returns **true** if it produced any side-channel result sets, and otherwise returns **false**. The `getNextResultSet()` method or `getNextResultSet(int)` method is used to obtain the next `java.sql.ResultSet` object. When the RTStatement object is released, the update count is set to QUERY_COUNT.

NOTE 42 — This method is called by generated code. Most programs do not need to call it directly. Instead, they will use only `getNextResultSet()` or `getNextResultSet(int)` to navigate multiple `java.sql.ResultSet` objects.

If the current operation produces multiple `java.sql.ResultSet` objects, it is not released until all `java.sql.ResultSet` objects have been processed and `getNextResultSet()` or `getNextResultSet(int)` returns null. If this execution context object is used to execute an SQL-statement while `java.sql.ResultSet` objects are still pending from the previous SQL-statement, or if an RTStatement object execution completes while `java.sql.ResultSet` objects from a recursive call are still pending, the `java.sql.ResultSet` objects are closed and discarded, and resources are released.

If this operation also produces side-channel update counts, they are discarded.

If an error occurs during execution of the SQL-statement, the current RTStatement object is released and an SQLException is thrown. Subsequent calls to `getNextResultSet()` or `getNextResultSet(int)` will return null.

It is assumed that this method is called within a block that is synchronized on this execution context object. Furthermore, it is also assumed that the previous call to register and the subsequent call to release the current RTStatement object both appear within the same synchronized block.

## Returns

— If the statement produced a side-channel result set, then **true**; otherwise **false**.

## Throws

— SQLException — if an error occurs during the execution of the currently registered RTStatement object (for example, the expiration of the query timeout previously set by invoking `setQueryTimeout()` on this execution context object).

## See Also

— Subclause 12.3.4.4.2, "execute ()"

— Subclause 12.3.4.4.18, "registerStatement (ConnectionContext, Object, int)"

— Subclause 12.3.4.4.12, "getNextResultSet ()"

— Subclause 12.3.4.4.13, "getNextResultSet (int)"

### 12.3.4.4.3 executeBatch ()

```
public synchronized int[] executeBatch ( ) throws SQLException
```

Executes the pending statement batch contained in this execution context object and returns the result as an array of update counts. If no pending statement batch exists for this execution context object, null is returned.

Upon direct or exceptional return from this method, update count is set to `EXEC_BATCH_COUNT`. If this method returns successfully, the batch update counts of this execution context object are updated to reflect the return result.

Once this method is called, the statement batch is emptied even if the call results in an exception. If a new statement batch is created as a result of executing the current batch, the new batch is implicitly executed. Subsequent calls to this method return null until another batchable statement is added.

Note that exceptions returned by this method will generally be instances of `java.sql.BatchUpdateException`.

## Returns

— If no statement batch exists, then null; otherwise, an array of update counts containing one element for each command in the batch.

The array is ordered according to the order in which commands were inserted into the batch. Each element either contains a non-negative update count, or the value –2 as a generic success indicator, or the value –3 as a generic failure indicator. Failure may also be indicated by an array that has fewer elements than the number of commands in the batch. In this case, each element shall contain either a non-negative update count or the value –2 as a generic success indicator.

## Throws

— SQLException — if the SQL-implementation raises an exception condition while executing the statement batch.



 

#### 12.3.4.4.4 executeQuery ()

```
public RTResultSet executeQuery ( ) throws SQLException
```

Invokes the `executeQuery()` method on the currently registered RTStatement object. Prior to statement execution, if there is a pending statement batch on this execution context object then the statement batch is implicitly executed using `BatchContext.executeBatch()`. If a new statement batch is created as a result of executing the current statement, the new batch is implicitly executed. When the RTStatement object is released, the update count is set to QUERY_COUNT.

NOTE 43 — This method is called by generated code. Most programs do not need to call it directly.

It is assumed that this method is called within a block that is synchronized on this execution context object. Furthermore, it is also assumed that the previous call to register and the subsequent call to release the current RTStatement object both appear within the same synchronized block.

## Returns

— The result of calling executeQuery on the currently registered RTStatement object.

## Throws

— SQLException — if an error occurs during the execution of the given RTStatement object

## See Also

— Subclause 13.2.6.2.5, “executeRTQuery ()”

— Subclause 12.3.4.4.18, “registerStatement (ConnectionContext, Object, int)”

#### 12.3.4.4.5 executeUpdate ()

```
public int executeUpdate ( ) throws SQLException
```

Prior to statement execution, if there is a pending statement batch on this execution context object and any of the following conditions are true:

— Batching currently disabled on this execution context object.

— The currently registered statement is not batchable.

— The currently registered RTStatement object is not batch compatible with the pending statement batch.

then the statement batch is implicitly executed using `BatchContext.executeBatch()`.

If batching is currently enabled on this execution context object and the currently registered RTStatement object is batchable, then the statement is batched rather than executed. The pending statement batch is replaced by a statement batch which includes the currently registered RTStatement object, as returned by `RTStatement.getBatchContext()`. If the statement was added to the existing statement batch, update count is set to

`ADD_BATCH_COUNT`. Otherwise, if the statement was added to a new statement batch, update count is set to `NEW_BATCH_COUNT`.

Otherwise, this invokes the `executeUpdate()` method on the currently registered RTStatement object. If a new statement batch is created as a result of executing the current statement, the new batch is implicitly executed. When the RTStatement object is released, the update count will be updated accordingly.

NOTE 44 — This method is called by generated code. Most programs do not need to call it directly.

It is assumed that this method is called within a block that is synchronized on this execution context object. Furthermore, it is also assumed that the previous call to register and the subsequent call to release the current RTStatement object both appear within the same synchronized block.

## Returns

— The update count resulting from the execution of the currently registered RTStatement object.

## Throws

— SQLException — if an error occurs during the execution of the given RTStatement object.

## See Also

— Subclause 12.3.4.4.5, “executeUpdate ()”

— Subclause 12.3.4.4.18, “registerStatement (ConnectionContext, Object, int)”

### 12.3.4.4.6 getBatchLimit ()

```
synchronized public int getBatchLimit ( )
```

Returns the current batch limit that was set for this execution context object.

## Returns

— Case:

- If the maximum batch size is unlimited, then `UNLIMITED_BATCH`.
- If the maximum batch size is finite and implementation-dependent, then `AUTO_BATCH`;
- Otherwise, a maximum batch size *n*>0.



 

#### 12.3.4.4.7 getBatchUpdateCounts ()

```
public synchronized int[] getBatchUpdateCounts ( )
```

Returns an array of update counts containing one element for each command in the last statement batch to successfully complete execution. Returns null if no statement batch has completed execution.

## Returns

— If no statement batch has completed execution, then null; otherwise, an array of update counts resulting from the last statement batch executed.

The array is ordered according to the order in which commands were inserted into the batch. Each element either contains a non-negative update count, or the value –2 as a generic success indicator, or the value –3 as a generic failure indicator. Failure may also be indicated by an array that has fewer elements than the number of commands in the batch. In this case, each element shall contain either a non-negative update count or the value –2 as a generic success indicator.

#### 12.3.4.4.8 getFetchDirection ()

```
synchronized public int getFetchDirection ( ) throws SQLException
```

Retrieves the current fetch direction for scrollable iterator objects generated from this ExecutionContext object. If this ExecutionContext object has not set a fetch direction by calling `setFetchDirection()`, then the value returned is the default specified in Subclause 12.3.4.4.22, "setFetchDirection (int)".

## Returns

— The current fetch direction for scrollable iterator objects generated from this ExecutionContext object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 12.3.4.4.9 getFetchSize ()

```
synchronized public int getFetchSize ( ) throws SQLException
```

Retrieves the number of rows that is the current fetch size for iterator objects generated from this Execution-Context object. If this ExecutionContext object has not set a fetch size by calling **setFetchSize**, then the value returned is 0 (zero). If this ExecutionContext object has set a non-negative fetch size by calling the method **setFetchSize**, then the return value is the fetch size specified on **setFetchSize**.

## Returns

— The current fetch size for iterator objects generated from this ExecutionContext object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 12.3.4.4.10 getMaxFieldSize ()

```
public synchronized int getMaxFieldSize ( )
```

The maximum Java field size limit (in bytes) is the maximum amount of data returned for any column value for SQL-statements subsequently executed using this execute context object; it only applies to BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR columns. These columns can be fetched into Java String, Byte array, or Stream objects. The limit affects both **`OUT`** parameters and **`INOUT`** parameters passed, and the result returned from any SQLJ executable clause. If the limit is exceeded, the excess data is discarded.

By default, the maximum Java field size limit is zero (unlimited).

## Returns

— The current maximum Java field size limit; 0 (zero) means unlimited.

### 12.3.4.4.11 getMaxRows ()

```
public synchronized int getMaxRows ( )
```

The maximum rows limit is the maximum number of rows that any iterator or `java.sql.ResultSet` object returned by SQL-statements subsequently executed using this execution context object can contain. If the limit is exceeded, the excess rows are dropped.

By default, the max rows limit is zero (unlimited).

## Returns

— The current maximum rows limit; 0 (zero) means unlimited.



 

#### 12.3.4.4.12 getNextResultSet ()

```
public synchronized ResultSet getNextResultSet ( ) throws SQLException
```

This method effectively invokes `this.getNextResultSet (java.sql.Statement.CLOSE_CURRENT_RESULT)` to return the currently registered RTStatement's next `java.sql.ResultSet` object (if any).

## Returns

— If there are no further `java.sql.ResultSet` objects, then null; otherwise, the next side-channel result set.

## Throws

— SQLException — if an error occurs obtaining the next `java.sql.ResultSet` object

— SQLException: *OLB-specific error — unsupported feature* — if closeType is set to other than `java.sql.Statement.CLOSE_CURRENT_RESULT` and support for Feature J009, "Multiple Open ResultSets", is not provided

## See Also

— Subclause 12.3.4.4.2, "execute ()"

— Subclause 12.3.4.4.13, "getNextResultSet (int)"

— Subclause 13.2.6.2.29, "getMaxRows ()"

— Subclause 13.2.6.2.34, "getResultSet ()"

#### 12.3.4.4.13 getNextResultSet (int)

```
public synchronized ResultSet getNextResultSet ( int closeType ) throws SQLException
```

Moves to the currently registered RTStatement object's next `java.sql.ResultSet` object. The first time this method is called after an SQL-statement is executed, the first side-channel result set is returned (if any). Further calls to `getNextResultSet(int)` advance to and return subsequent `java.sql.ResultSet` objects of the currently registered RTStatement. `getNextResultSet(int)` returns null if there are no further `java.sql.ResultSet` objects; null is also returned if an SQL-statement has not yet been executed on this execution context object.

If the constant `java.sql.Statement.CLOSE_CURRENT_RESULT` is passed, then the `java.sql.ResultSet` object returned by the last call to `getResultSet()` against the currently registered RTStatement is closed. If the constant `java.sql.Statement.CLOSE_ALL_RESULTS` is passed, then all open `java.sql.ResultSet` objects previously obtained from the currently registered RTStatement are closed. If the constant `java.sql.Statement.KEEP_CURRENT_RESULT` is passed, then the last `java.sql.ResultSet` object obtained from the currently registered RTStatement is left open.

NOTE 45 — If the last SQL-statement executed on this execution context object produced multiple `java.sql.ResultSet` objects, its resources are not released until all `java.sql.ResultSet` objects have been processed and `getNextResultSet()` returns null. If this execution context object is used to execute an SQL-statement while `java.sql.ResultSet` objects are still pending from the previous operation, or if a statement execution completes while `java.sql.ResultSet` objects from a recursive call are still pending, the `java.sql.ResultSet` objects are closed and discarded, and resources are released.

If this operation also produces side-channel update counts, they are discarded.

If an error occurs during a call to `getNextResultSet(int)`, the current `java.sql.ResultSet` object is released and an SQLException is thrown. Subsequent calls to `getNextResultSet(int)` return null.

## Parameters

— closeType — one of the values `java.sql.Statement.CLOSE_CURRENT_RESULT`, `java.sql.Statement.CLOSE_ALL_RESULTS`, and `java.sql.Statement.KEEP_CURRENT_RESULT`

## Returns

— If there are no further `java.sql.ResultSet` objects, then null; otherwise, the next side-channel result set.

## Throws

— SQLException — if an error occurs obtaining the next `java.sql.ResultSet` object

## See Also

— Subclause 12.3.4.4.2, “execute ()”

— Subclause 12.3.4.4.12, “getNextResultSet ()”

— Subclause 12.2.5.3.4, “getResultSet ()”

— Subclause 13.2.6.2.29, “getMaxRows ()”

— Subclause 13.2.6.2.34, “getResultSet ()”

### 12.3.4.4.14 getQueryTimeout ()

```
public synchronized int getQueryTimeout ( )
```

The query timeout limit is the maxiumum number of seconds SQL-statements subsequently executed using this execution context object are permitted to take to complete. If execution of the SQL-statement exceeds the limit, an SQLException is thrown.

By default, the query timeout limit is zero (unlimited).



 

## Returns

— The current query timeout limit in seconds; 0 (zero) means unlimited.

### 12.3.4.4.15 getUpdateCount ()

```
public synchronized int getUpdateCount ( )
```

Returns the update count, defined as the number of rows updated by the last SQL-statement to complete execution using this execution context object. 0 (zero) is returned if the last SQL-statement was not a DML statement.

QUERY_COUNT is returned if the last SQL-statement created an iterator object or `java.sql.ResultSet` object.

EXCEPTION_COUNT is returned if an exception occurred before the last SQL-statement completed execution, or no operation has yet been attempted.

## Returns

— Case:

- If the last SQL-statement was batchable and was added as the first member of a new statement batch, then `NEW_BATCH_COUNT`.
- If the last SQL-statement was batchable and was added to the current statement batch, then `ADD_BATCH_COUNT`.
- If a statement batch has completed execution more recently than any unbatched statement, then `EXEC_BATCH_COUNT`.
- Otherwise, the number of rows updated by the last operation.

### 12.3.4.4.16 getWarnings ()

```
public synchronized SQLWarning getWarnings ( )
```

Returns the first warning reported by the last SQL-statement to complete execution using this execution context object. Subsequent warnings resulting from the same SQL-statement are chained to this SQLWarning. The SQLWarning chain returned represents those warnings that occured during the execution of the last SQL-statement and the subsequent binding of any output host variables.

NOTE 46 — If an iterator is being processed, then all warnings associated with iterator column reads are chained on the iterator object.

## Returns

— If no warnings occurred, then null; otherwise, the first SQLWarning.

### 12.3.4.4.17 isBatching ()

```
public synchronized boolean isBatching ( )
```

Returns **true** if batching is currently enabled for this execution context object, **false** if batching is disabled. Note that the value returned reflects only whether it is possible to batch statements, but not whether a pending statement batch exists.

## Returns

— If batching enabled, then **true**; otherwise, **false**.

### 12.3.4.4.18 registerStatement (ConnectionContext, Object, int)

```
public RTStatement registerStatement
      ( ConnectionContext connCtx, Object profileKey, int stmtNdx )
    throws SQLException
```

Creates, registers and returns an RTStatement object. This method is called by generated code. Most programs do not need to call it directly.

The RTStatement object is created by accessing the ConnectedProfile object within connection context object "connCtx" that has the key "profileKey". The RTStatement object at index "stmtNdx" in the ConnectedProfile object is created using the `getStatement()` method. If batching is currently enabled, then the current statement batch is passed as an additional argument to the `getStatement()` method. If there is no pending statement batch, then the current statement batch passed to `getStatement()` is `null`.

The RTStatement object created is registered and becomes the current RTStatement object of this execution context object.

For each of the maximum rows, maximum Java field size, and query timeout limits of this execution context object, if the limit has a non-default value, then the corresponding methods for setting these limits on the registered RTStatement object are invoked. An SQLException is thrown if the runtime class implementing RTStatement does not support changing the limit to a non-default value.

The given connection context object's execution context object is not used by this method.

Note that if this method throws an exception, no RTStatement object will be registered.

> NOTE 47 — It is assumed that this method is called within a block that is synchronized on this execution context object. Subsequent calls to execute and release the RTStatement object returned should also appear within the same synchronized block. If there is another RTStatement object currently registered on this execution context object, it is assumed that this method is a recursive call initiated by the currently registered RTStatement object. In such cases, state involving the currently registered RTStatement object is saved, and the RTStatement object returned by this method becomes the currently registered RTStatement object. Once the execution of this new RTStatement object has completed execution and and the object is released, the previous RTStatement object is restored as the currently registered RTStatement object.

## Parameters

— connCtx — the connection context object that contains the profile object that contains the RTStatement object to register



 

— profileKey — the key of the ConnectedProfile object within the connection context object

— stmtNdx — the zero-based index of the RTStatement object within the profile object to be registered

## Returns

— The newly-created and -registered RTStatement object.

## Throws

— SQLException — if there is another RTStatement object currently executing or if the maximum Java field size, maximum rows, or query timeout cannot be set on the registered RTStatement object

## See Also

— Subclause 12.3.4.4.19, “releaseStatement ()”

— Subclause 13.2.2.2.4, “getStatement (int, Map)”

— Subclause 12.2.1.3.3, “getConnectedProfile (Object)”

— Subclause 13.2.2.2.5, “getStatement (int, BatchContext, Map)”

### 12.3.4.4.19 releaseStatement ()

```
public void releaseStatement ( ) throws SQLException
```

Releases the currently registered RTStatement object, signaling that all execution related operations have completed. Once this method has been executed, `registerStatement` can be called again. The SQL warnings and update count are updated as reflected by the registered RTStatement object and the execution RTResultSet objects.

If the execution of the currently registered RTStatement object produced multiple `java.sql.ResultSet` objects and not all `java.sql.ResultSet` objects have been implicitly or explicitly closed, then this operation is a no-op. In such cases, this method is automatically called to release the RTStatement object once all `java.sql.ResultSet` objects have been processed and `getNextResultSet()` or `getNextResultSet(int)` returns null.

This method calls the `executeComplete()` method of the registered RTStatement object.

NOTE 48 — This method is called by generated code. Most programs do not need to call it directly.

It is assumed that this method is called within a block that is synchronized on this execution context object. Furthermore, it is also assumed that the previous call to register and the subsequent call to release the current RTStatement object both appear within the same synchronized block.

## Throws

— SQLException — if an error occurs retrieving the warnings

### 12.3.4.4.20 setBatching (boolean)

```
public synchronized void setBatching (boolean doBatch)
```

Enables or disables batching for statements executed on this execution context object. When batching is enabled, batchable statements that are registered with this execution context object will be added to a statement batch for deferred execution instead of being executed immediately. A statement batch can be executed explicitly using the `executeBatch()` command. Statement batches are also executed implicitly when a statement that cannot be added to the current statement batch is executed. If a statement being executed is batchable and compatible with the current statement batch, it is added to the batch.

When batching is disabled, statements are executed as usual. Subsequent statements are not considered for addition to the pending statement batch.

This method only affects statements encountered after it is called. It does not affect statements that have previously been or are currently being executed, nor does it affect the pending statement batch.

## Parameters

— doBatch — **true** if batching should be enabled, **false** if batching should be disabled

### 12.3.4.4.21 setBatchLimit (int)

```
public synchronized void setBatchLimit (int batchLimit)
```

Sets the maximum batch size. When batching is enabled and the maximum batch size is exceeded, implicit batch execution is performed.

The following remarks assume that batching is enabled.

— When the constant `UNLIMITED_BATCH` is specified, the maximum batch size is unlimited, and can not be exceeded. New ExecutionContext objects are always created with `UNLIMITED_BATCH`.

— When a positive `batchLimit` is specified, an implict batch execution will be performed whenever the number of batched statements reaches `batchLimit`.

— When the constant `AUTO_BATCH` is specified, the maximum batch size is finite but unspecified. Whenever a batch-compatible statement is added to a batch, the SQLJ runtime implementation may decide to do one of the following:

- Add the statement to the batch.
- Execute the current non-empty batch and create a new singleton batch that contains the statement.
- Add the statement to the current batch and execute the batch. As a special case of this situation, given an empty batch, the implementation may also simply go ahead and execute the statement.

— The implementation should reasonably avoid creating out-of-memory conditions due to implicit batching with `AUTO_BATCH`.



 

This method only affects statements encountered after it is called. It does not affect statements that have previously been or are currently being executed, nor does it affect the pending statement batch.

## Parameters

— batchLimit — `UNLIMITED_BATCH` if the maximum batch size is unlimited, `AUTO_BATCH` if the maximum batch size is finite and implementation dependent, or *n*>0 for a maximum batch size of *n*.

### 12.3.4.4.22 setFetchDirection (int)

```
public synchronized void setFetchDirection ( int direction ) throws SQLException
```

Gives the SQLJ runtime a hint as to the direction in which rows of scrollable iterator objects are processed. The hint applies only to scrollable iterator objects created using this ExecutionContext object. The default value is **sqlj.runtime.ResultSetIterator.FETCH_FORWARD**.

## Parameters

— direction — the initial fetch direction for scrollable iterator objects generated from this ExecutionContext object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition, or the given direction is not one of `ResultSetIterator.FETCH_FORWARD`, `ResultSetIterator.FETCH_REVERSE`, or `ResultSetIterator.FETCH_UNKNOWN`.

### 12.3.4.4.23 setFetchSize (int)

```
synchronized public void setFetchSize ( int rows ) throws SQLException
```

Gives the SQLJ runtime a hint as to the number of rows that should be fetched when more rows are needed. The number of rows specified affects only iterator objects created using this ExecutionContext object.

## Parameters

— rows — the fetch size for result sets associated with iterator objects whose initialization involves use of this ExecutionContext object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition, or the condition 0 (zero) ≤ `rows` ≤ `this.getMaxRows()` is not satisfied.

#### 12.3.4.4.24 setMaxFieldSize (int)

```
public synchronized void SetMaxFieldSize (int max)
```

The maximum Java field size limit (in bytes) is the maximum amount of data returned for any column value for SQL-statements subsequently executed using this execution context object; it only applies to BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR columns. These columns can be fetched into Java String, Byte array, or Stream objects. The limit affects both **`OUT`** parameters and **`INOUT`** parameters passed, and the result returned from any SQLJ executable clause. If the limit is exceeded, the excess data is discarded. For maximum portability, use values greater than 256.

By default, the maximum Java field size limit is zero (unlimited).

NOTE 49 — Without Feature J003, "Execution control", if MaxFieldSize is set to other than its default value and a subsequent attempt is made to register an RTStatement object with such an ExecutionContext object, then an SQLException condition is thrown: *OLB-specific error — unsupported feature*.

## Parameters

— max — the new maximum Java field size limit; zero means unlimited

#### 12.3.4.4.25 setMaxRows (int)

```
public synchronized void setMaxRows (int max)
```

The maximum rows limit is the maximum number of rows that any iterator or `java.sql.ResultSet` object returned by SQL-statements subsequently executed using this execution context object can contain. If the limit is exceeded, the excess rows are dropped.

By default, the maximum rows limit is zero (unlimited).

NOTE 50 — Without Feature J003, "Execution control", if MaxRows is set to other than its default value and a subsequent attempt is made to register an RTStatement object with such an ExecutionContext object, then an SQLException condition is thrown: *OLB-specific error — unsupported feature*.

## Parameters

— max — the new maximum rows limit; zero means unlimited

#### 12.3.4.4.26 setQueryTimeout (int)

```
public synchronized void setQueryTimeout (int seconds)
```

The query timeout limit is the maxiumum number of seconds SQL-statements subsequently executed using this execution context object are permitted to take to complete. If execution of the SQL-statement exceeds the limit, an SQLException is thrown.

By default, the query timeout limit is zero (unlimited).



 

NOTE 51 — Without Feature J003, "Execution control", if QueryTimeout is set to other than its default value and a subsequent attempt is made to register an RTStatement object with such an ExecutionContext object, then an SQLException condition is thrown: *OLB-specific error — unsupported feature*.

## Parameters

— seconds — the new query timeout limit in seconds; zero means unlimited.

### 12.3.5 sqlj.runtime.RuntimeContext

#### 12.3.5.1 Class Overview

```
java.lang.Object
   |
   +--sqlj.runtime.RuntimeContext

public abstract class RuntimeContext
    extends Object
```

The RuntimeContext class defines system specific services to be provided by the runtime environment. The RuntimeContext class is an abstract class the implementation of which might vary according to the Java Virtual Machine environment.

#### 12.3.5.2 Variables

##### 12.3.5.2.1 DEFAULT_DATA_SOURCE

```
public static final String DEFAULT_DATA_SOURCE = "jdbc/defaultDataSource";
```

The JNDI name of the data source used to create the default Connection object, *`jdbc/defaultDataSource`*.

## See Also

— Subclause 12.3.5.4.1, "getDefaultConnection ()"

#### 12.3.5.2.2 DEFAULT_RUNTIME

```
public static final String DEFAULT_RUNTIME = "sqlj.runtime.DefaultRuntime";
```

The fully-qualified class name of the default runtime implementation used if no other implementation has been defined for a Java Virtual Machine environment.

## See Also

— Subclause 12.3.3.2.1, "DefaultRuntime ()"

#### 12.3.5.2.3 PROPERTY_KEY

```
public static final String PROPERTY_KEY = "sqlj.runtime";
```

The key under which the RuntimeContext implementation class name is stored in the system properties.

### 12.3.5.3 Constructors

#### 12.3.5.3.1 RuntimeContext ()

```
public RuntimeContext ( )
```

The default constructor for the RuntimeContext class

### 12.3.5.4 Methods

#### 12.3.5.4.1 getDefaultConnection ()

```
public abstract Connection getDefaultConnection ( )
```

Returns the default Connection object, if one exists, or null otherwise.

NOTE 52 — Some environments might have an implicit Connection object available. For example, a Java Virtual Machine running in an SQL-environment might have an implicit Connection object associated with the current SQL-session.

If the default data source is defined in JNDI, then it is used to establish the default Connection object.



 

## Returns

— If no default Connection object exists, then null; otherwise, the default Connection object.

## See Also

— Subclause 12.3.5.2.1, "DEFAULT_DATA_SOURCE"

### 12.3.5.4.2 getLoaderForClass (Class)

```
public abstract Loader getLoaderForClass ( Class forClass )
```

Resources and classes loaded from this Loader object are found in the same location that the given class was found in.

NOTE 53 — The definition of location might vary depending on class loading and resolution semantics of the runtime implementation.

It is assumed that the argument substituted for forClass contains enough information for a Java Virtual Machine implementation to be able to determine the location in which to find related resources. Most Java Virtual Machine implementations will be able to use the given class's class loader (or the system class loader, if the class has no loader). However, some Java Virtual Machine implementations might need additional information to resolve resources. For example, a Java Virtual Machine running in an SQL-environment might use the schema in which the given class is located to search for related resources.

## Parameters

— forClass — the class with which the resulting Loader object is to be associated

## Returns

— A Loader object associated with a class.

### 12.3.5.4.3 getRuntime ()

```
public static RuntimeContext getRuntime ( )
```

Returns a RuntimeContext object resembling the runtime context object associated with the current Java Virtual Machine instance. Each Java Virtual Machine instance has a single unique runtime context object. Subsequent invocations of this method within the same Java Virtual Machine instance will return the same object. The appropriate RuntimeContext implementation is discovered by examining the value of the `RuntimeContext.PROPERTY_KEY` system property. If this property is set, it indicates the full name of a class that is able to be instantiated to create a runtime context object. If no such property is defined, or if access to this system property is not allowed, then the class given by `RuntimeContext.DEFAULT_RUNTIME` is used.

NOTE 54 — All runtime implementations shall be able to be constructed via the `Class.newInstance()` method. That is, they shall have a public niladic constructor.

## Returns

— The RuntimeContext object associated with the current Java Virtual Machine

## See Also

— Subclause 12.3.5.2.3, “PROPERTY_KEY”

— Subclause 12.3.5.2.2, “DEFAULT_RUNTIME”

### 12.3.6 sqlj.runtime.StreamWrapper

#### 12.3.6.1 Class Overview

```
java.lang.Object
   |
   +--java.io.InputStream
           |
           +--java.io.FilterInputStream
                   |
                   +--sqlj.runtime.StreamWrapper

public class StreamWrapper
    extends FilterInputStream
```

This class wraps a particular InputStream object. It also extends the InputStream class, delegating method invocations directly to the wrapped InputStream object for all methods. Additionally, it supports methods for specifying the length of the wrapper InputStream object, which allows it to be passed as an argument to the invocation of an SQL-statement.

## See Also

— java.io.InputStream — a standard Java class

#### 12.3.6.2 Constructors

##### 12.3.6.2.1 StreamWrapper (InputStream)

```
protected StreamWrapper ( InputStream in )
```

Creates a new StreamWrapper object using the octets in the given InputStream object. The length of the InputStream object is uninitialized.



 

## Parameters

— in — the InputStream object to wrap.

#### 12.3.6.2.2 StreamWrapper (InputStream, int)

```
protected StreamWrapper ( InputStream in, int length )
```

Creates a new StreamWrapper object using the octets in the given InputStream object. The length of the InputStream object is initialized to the given length value.

## Parameters

— in — the InputStream object to wrap.

— length — the length of the InputStream object in octets.

### 12.3.6.3 Methods

#### 12.3.6.3.1 getInputStream ()

```
public InputStream getInputStream ( )
```

Returns the InputStream object that is being wrapped by this StreamWrapper object.

## Returns

— The underlying InputStream object that is being wrapped.

#### 12.3.6.3.2 getLength ()

```
public int getLength()
```

Returns the length in octets of the wrapped InputStream object, as specified during construction or in the last call to `setLength()`.

## Returns

— The length in octets of the InputStream object.

#### 12.3.6.3.3 setLength (int)

```
public void setLength ( int length )
```

Sets the length Java field of the wrapped stream to be the given value. This does not affect the wrapped InputStream object, but will affect the number of octets read from it when it is passed as an argument to an invocation of an SQL-statement.

## Parameters

— length — the new length of the InputStream object in octets.

### 12.3.7 sqlj.runtime.UnicodeStream

#### 12.3.7.1 Class Overview

```
java.lang.Object
   |
   +--java.io.InputStream
           |
           +--java.io.FilterInputStream
                   |
                   +--sqlj.runtime.StreamWrapper
                           |
                           +--sqlj.runtime.UnicodeStream

public class UnicodeStream
    extends StreamWrapper
```

**UnicodeStream** (sqlj.runtime.UnicodeStream) is a class derived from java.io.InputStream. The octets comprising a UnicodeStream object are interpreted as Unicode characters. When an InputStream is passed as an argument to an invocation of an SQL-statement, both the length of the InputStream object and the way to interpret its octets shall be specified. Therefore, an InputStream object cannot be passed directly, but rather shall be an instance of AsciiStream, BinaryStream or UnicodeStream.

## See Also

— Subclause 12.3.1, “sqlj.runtime.AsciiStream”

— Subclause 12.3.2, “sqlj.runtime.BinaryStream”



 

#### 12.3.7.2 Constructors

##### 12.3.7.2.1 UnicodeStream (InputStream)

```
public UnicodeStream ( InputStream in )
```

Creates a Unicode-valued InputStream object with an uninitialized length.

NOTE 55 — The length Java field shall be set by a call to `setLength()` before a UnicodeStream object is substituted for an input (or inout) parameter in an invocation of an SQL-statement.

## Parameters

— in — the InputStream object to interpret as a UnicodeStream object.

##### 12.3.7.2.2 UnicodeStream (InputStream, int)

```
public UnicodeStream ( InputStream in, int length )
```

Creates a Unicode-valued InputStream object of given length.

## Parameters

— in — the InputStream object to interpret as a UnicodeStream object.

— length — the length in octets of the UnicodeStream object.

### 12.3.8 sqlj.runtime.CharacterStream

#### 12.3.8.1 Class Overview

```
java.lang.Object
   |
   +--java.io.Reader
          |
          +--java.io.FilterReader
```

```
                    |
                    +--sqlj.runtime.CharacterStream

public class CharacterStream
    extends FilterReader
```

A class derived from java.io.Reader whose instances contain Unicode data. When an instance of this class is passed as an input argument to an invocation of an SQL-statement, the length of the Reader object shall be specified. Therefore, an instance of the Reader class cannot be passed directly, but rather shall be an instance of CharacterStream.

### 12.3.8.2 Constructors

#### 12.3.8.2.1 CharacterStream (Reader)

```
public CharacterStream ( Reader in )
```

Creates an instance of CharacterStream with an uninitialized length.

NOTE 56 — The length Java field shall be set by a call to `setLength()` before use of a CharacterStream object as an input (or inout) parameter in an invocation of an SQL-statement.

## Parameters

— in — the Reader to interpret as a CharacterStream object.

#### 12.3.8.2.2 CharacterStream (Reader, int)

```
public CharacterStream ( Reader in, int length )
```

Creates an instance of CharacterStream of given length.

## Parameters

— in — the Reader object to interpret as a CharacterStream object.

— length — the length in characters of the CharacterStream object.



 

#### 12.3.8.3 Methods

##### 12.3.8.3.1 getReader ()

```
public Reader getReader ( )
```

Returns the underlying Reader object wrapped by the CharacterStream object.

## Returns

— The underlying Reader object that is being wrapped

##### 12.3.8.3.2 getLength ()

```
public int getLength()
```

Returns the length in characters of the wrapped Reader object, as specified during construction or in the last call to `setLength()`.

## Returns

— The length in characters of the Reader object

##### 12.3.8.3.3 setLength (int)

```
public void setLength ( int length )
```

Sets the length Java field of the wrapped Reader object to be the passed value. This does not affect the wrapped Reader object, but will affect the number of characters read from it when it is passed as an input argument to an invocation of an SQL-statement.

## Parameters

— length — the length of the Reader object in characters.

### 12.3.9 sqlj.runtime.SQLNullException

#### 12.3.9.1 Class Overview

```
java.lang.Object
   |
   +--java.lang.Throwable
           |
           +--java.lang.Exception
                   |
                   +--java.sql.SQLException
                           |
                           +--sqlj.runtime.SQLNullException
```

```
public class SQLNullException
    extends SQLException
```

The SQLNullException class is a subclass of SQLException that is used in the case that the SQL null value was attempted to be fetched into a Java host variable whose type is a Java primitive datatype. This exception is thrown when such a condition occurs.

The SQLSTATE value for every instance of SQLNullException is '22002' (*data exception — null value, no indicator parameter*).

#### 12.3.9.2 Constructors

##### 12.3.9.2.1 SQLNullException ()

```
public SQLNullException ( )
```

Create an SQLNullException object. The SQLState Java field is initialized to '22002', and the vendorCode Java field is set to the SQLException default.

## Conformance Rules

*None.*



 

# 13 Package sqlj.runtime.profile

## 13.1 Overview

The sqlj.runtime.profile package defines the classes and interfaces that enable binary portable SQLJ programs. It is distinguished from the package **sqlj.runtime** because it defines classes that are used by SQLJ runtime implementations, but are not otherwise visible to an SQLJ program.

## 13.2 SQLJ sqlj.runtime.profile Interfaces

### 13.2.1 sqlj.runtime.profile.BatchContext

#### 13.2.1.1 Interface Overview

```
public interface BatchContext
```

A batch context object is used to group statements that are to be submitted to the SQL-implementation for execution as a batch using a single round trip.

#### 13.2.1.2 Methods

##### 13.2.1.2.1 clearBatch ()

```
public abstract void clearBatch() throws SQLException
```

Removes all statements contained in this batch context object and releases all associated resources.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 13.2.1.2.2 executeBatch ()

```
public abstract int[] executeBatch ( ) throws SQLException
```

Executes the statements contained in this batch context object and returns the result as an array of update counts. The array is ordered according to the order in which commands were inserted into the batch. Note that exceptions returned by this method will generally be instances of `java.sql.BatchUpdateException`.

## Returns

— An array of update counts containing one element for each command in the batch. The array is ordered according to the order in which commands were inserted into the batch. Each element either contains a non-negative update count, or the value –2 as a generic success indicator, or the value –3 as a generic failure indicator. Failure may also be indicated by an array that has fewer elements than the number of commands in the batch. In this case, each element shall contain either a non-negative update count or the value –2 as a generic success indicator.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 13.2.1.2.3 setBatchLimit (int)

```
public abstract void setBatchLimit (int batchSize) throws SQLException
```

Sets the maximum batch size on this batch context object. When batching is enabled and the maximum batch size is exceeded, implicit batch execution is performed. The following remarks assume that batching is enabled.

— When the constant `ExecutionContext.UNLIMITED_BATCH` is specified, the maximum batch size is unlimited, and can not be exceeded. New BatchContext objects are always created with `UNLIMITED_BATCH`.

— When a positive batchLimit is specified, an implict batch execution will be performed, whenever the number of batched statements reaches `batchLimit`.

— When the constant `ExecutionContext.AUTO_BATCH` is specified, the maximum batch size is finite but unspecified.

## Parameters

— `batchLimit` — `ExecutionContext.UNLIMITED_BATCH` if the maximum batch size is unlimited, `ExecutionContext.AUTO_BATCH` if the maximum batch size is finite and implementation dependent, or $n > 0$ (zero) for a maximum batch size of $n$.

## Throws

— SQLException — if an invalid or unsupported batch size is specified



 

## 13.2.2 sqlj.runtime.profile.ConnectedProfile

### 13.2.2.1 Interface Overview

```
public interface ConnectedProfile
```

A ConnectedProfile object represents a profile object that has been attached to a particular `java.sql.Connection` object. Since it is attached to a Connection object, it is able to convert its contents into an executable statement object on the associated Connection object. The implementation of this object might be customized for the given data source, which allows it to use optimizations that circumvent the JDBC dynamic SQL model. Profile customization will typically involve implementation-dependent profile object transformations that allow more efficient SQL execution such a precompilation of SQL text or use of SQL-invoked procedures.

A ConnectedProfile object contains statements that correspond to entries at a particular index in the profile object. The profile's EntryInfo object at a particular index can be used to determine how the corresponding statement returned by a ConnectedProfile object will be executed at runtime. The statement returned need only respond to the `execute()` method indicated in the EntryInfo object.

A customization may also provide means for specifying the user identifier used for privilege checking. By default, the connection context user identifier of the Connection object associated with a ConnectedProfile object is used as the current user identifier for execution of all SQL-statements created by a connected profile. As an alternative, a customized user identifier can be provided during the customization of an SQL/OLB application as additional input to a customizer and included in a customized profile in an implementation-defined manner. At runtime, a registered Customization object can make the customized profile user identifier available to the customization-specific ConnectedProfile (and RTStatement) objects in an implementation-dependent manner, so that statements created by the ConnectedProfile use the customized profile user identifier as the current user identifier, instead of the connection context user identifier.

The profile's EntryInfo object at a particular index also characterizes the statement type. A statement can be either PREPARED or CALLABLE, the difference between the two being that CALLABLE statements are permitted to have **`OUT`** parameters whereas PREPARED statements will have only **`IN`** parameters.

All statements returned by a ConnectedProfile object conform to the following requirements:

— The operation performed shall be equivalent to the operation that would have been performed if using regular JDBC and the text of the SQL-statement directly.

— Any **`OUT`** parameters of the operation shall have been already registered for the statement returned (CALLABLE statements only). The profile object describes each parameter to the operation in terms of its Java class description, and provides additional SQL type information (*i.e.*, STRUCT, DISTINCT, JAVA_OBJECT) for Java classes that map to user-defined data types. It is up to the implementation to properly register the SQL type for this class description as needed for the particular JDBC (or implementation-dependent) driver used.

If the ConnectedProfile object is unable to create the desired statement, an exception is thrown. Note that a particular profile customization object might employ an “eager” verification algorithm in which all entries in the profile object are verified against the Connection object when a ConnectedProfile object is created, or a “lazy” verification algorithm in which statements are not verified until they are indexed via this method. It is up to the implementations of the Customization and ConnectedProfile interfaces to decide upon an appropriate verification strategy.

## See Also

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.2.14, “PREPARED_STATEMENT”

— Subclause 13.3.2.4.3, “getExecuteType ()”

### 13.2.2.2 Methods

#### 13.2.2.2.1 close ()

```
public abstract void close ( ) throws SQLException
```

Closes this ConnectedProfile object, releasing any resources associated with it. `close()` is called when the connection context object associated with the profile object is closed.

## Throws

— SQLException — if an error occurs while closing

#### 13.2.2.2.2 getConnection ()

```
public abstract Connection getConnection ( )
```

## Returns

— The Connection object with which this ConnectedProfile object was created

## See Also

— Subclause 12.2.1.3.3, “getConnectedProfile (Object)”

#### 13.2.2.2.3 getProfileData ()

```
public abstract ProfileData getProfileData ( )
```

The top level profile object that created this connected profile object can be retrieved by calling the `getProfile()` method on the resulting ProfileData object.



 

## Returns

— The ProfileData object associated with this ConnectedProfile object.

## See Also

— Subclause 12.2.1.3.3, "getConnectedProfile (Object)"

— Subclause 13.2.3.2.2, "getProfile (Connection, Profile)"

### 13.2.2.2.4 getStatement (int, Map)

```
public abstract RTStatement getStatement ( int ndx, java.util.Map typeMap )
       throws SQLException
```

If the profile EntryInfo object contains invalid information, then an SQLException condition is thrown: *OLB-specific error — invalid profile state*. The Map object provided in the typeMap parameter is passed to the returned RTStatement object in an implementation-defined manner.

## Parameters

— ndx — the index of the statement to return, zero-based

— typeMap — a java.util.Map object containing user-defined type mapping information of the connection context class that is associated with the statement to be executed.

## Returns

— A statement object representing the EntryInfo object at index ndx in the profile object, where ndx is zero-based.

## Throws

— SQLException — if an error occurs preparing the statement

### 13.2.2.2.5 getStatement (int, BatchContext, Map)

```
public abstract RTStatement getStatement ( int ndx, BatchContext batch,
                                           java.util.Map typeMap )
    throws SQLException
```

Returns a statement object representing the EntryInfo object at index ndx in the profile object, where ndx is zero-based. The Map object provided in the typeMap parameter is passed to the returned RTStatement object in an implementation-defined manner.

The passed batch context object is used by the statement to determine batch compatibility. If possible, the statement will be added to the passed batch for deferred execution via a call to `RTStatement.getBatchContext()`. If the passed batch is `null`, then the statement will create and return a new one-element batch containing itself when `getBatchContext()` is subsequently called on it.

If the profile EntryInfo object contains invalid information, then an SQLException is thrown: *OLB-specific error — invalid profile state*.

## Parameters

— ndx — the index of the statement to return, zero-based

— batch — a pending statement batch with which to merge, if possible. This batch may be null.

— typeMap — a java.util.Map object containing user-defined type mapping information of the connection context class that is associated with the statement to be executed.

## Returns

— A statement object representing the entry at index ndx in the profile object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.2.6.2.45, “isBatchCompatible ()”

— Subclause 13.2.6.2.8, “getBatchContext ()”

### 13.2.3 sqlj.runtime.profile.Customization

#### 13.2.3.1 Interface Overview

```
public interface Customization
    extends java.io.Serializable
```

A profile Customization object is a serializable object that maps a particular `java.sql.Connection` object and basic profile object into a customized ConnectedProfile object. Because both profile objects and Customization objects are serializable, new Customization objects can be added to profiles as needed anytime after the profile object has been created. This will most often happen during an “installation” phase after the application has been translated, but before the application is actually run.

Profiles might be customized in any number of ways. Some typical examples are:



 

— Transformation of SQL text into a format that allows more efficient execution on a particular data source. Precompilation and use of SQL-invoked procedures are examples of this.

— Batch verification and/or preparation of profile EntryInfo objects to avoid multiple data source round trips.

— Distributed and/or remote loading of custom EntryInfo objects.

— Custom type registration of data source-specific EntryInfo object parameters.

— Behavioral unification of multiple JDBC drivers with which an application is to be deployed.

— Specification of a user identifier to be used for privilege checking of embedded statements at runtime.

## See Also

— Subclause 13.3.3, "sqlj.runtime.profile.Profile"

### 13.2.3.2 Methods

#### 13.2.3.2.1 acceptsConnection (Connection)

```
public abstract boolean acceptsConnection ( Connection conn )
```

## Parameters

— conn — the `java.sql.Connection` object used in testing the ability to create a ConnectedProfile object.

## Returns

— If this Customization object can create a ConnectedProfile object for the given `java.sql.Connection` object, then **true**; otherwise, **false**.

#### 13.2.3.2.2 getProfile (Connection, Profile)

```
public abstract ConnectedProfile getProfile
     ( Connection conn, Profile baseProfile )
       throws SQLException
```

If the Profile object identified by baseProfile cannot be connected, then an exception is thrown. The exception might be the result of the Profile object identified by baseProfile containing entries that cannot be prepared and executed on the Connection object identified by conn. Depending on the implementation of the Customization object, verification of Profile object entries might occur when the Profile object identified by baseProfile is connected, or be deferred until an entry is directly accessed by the client.

## Parameters

— conn — input `java.sql.Connection` object

— baseProfile — input base Profile

## Returns

— A ConnectedProfile object for baseProfile on the given `java.sql.Connection` object.

## Throws

— SQLException — if the Profile object identified by baseProfile cannot be connected.

### 13.2.4 sqlj.runtime.profile.Loader

#### 13.2.4.1 Interface Overview

```
public interface Loader
```

A profile.Loader object is used as the context for profile object instantiation rather than a Java class loader object. This allows flexibility to runtime environments in which class Loader objects cannot be properly defined for all classes, and resource names would not otherwise be able to be resolved.

## See Also

— Subclause 13.3.1, “sqlj.runtime.profile.DefaultLoader”

#### 13.2.4.2 Methods

##### 13.2.4.2.1 getResourceAsStream (String)

```
public abstract InputStream getResourceAsStream ( String name )
```

Get an InputStream object on a given resource. Returns null if no resource with this name is found. This method is called when SerializedProfile objects are instantiated.

The way in which resources are located is determined solely by the Loader implementation.



 

## Parameters

— name — the name of the resource

## Returns

— If an InputStream object on the resource identified by the name parameter can be found, then the InputStream on that resource; otherwise null.

#### 13.2.4.2.2 loadClass (String)

```
public abstract Class loadClass ( String className )
       throws ClassNotFoundException
```

Requests the Loader object to load a class with the specified name. The `loadClass()` method is called when a profile object is instantiated and when a profile object is instantiated and the Java class of a TypeInfo object needs to be loaded for the first time as part of the instantiation process.

Loaders should use a hashtable or other cache to avoid defining classes with the same name multiple times.

## Parameters

— name — the fully qualified name of the desired Class.

## Returns

— The resulting Class.

## Throws

— ClassNotFoundException — if the Loader object cannot find a definition for the class

## See Also

— Subclause 13.3.3.3.11, “instantiate (Loader, InputStream)”

— Subclause 13.3.3.3.12, “instantiate (Loader, String)”

— Subclause 13.3.3.3.5, “getJavaType (String)”

— Subclause 13.3.3.3.6, “getJavaType (TypeInfo)”

### 13.2.5 sqlj.runtime.profile.RTResultSet

#### 13.2.5.1 Interface Overview

```
public interface RTResultSet
```

This interface defines the operations used for accessing an RTResultSet's data resulting from the execution of an SQL query described by a profile EntryInfo object. It is based strongly on the `java.sql.ResultSet` interface, and can be implemented using a `java.sql.ResultSet` object. In general, any method with the same name as one of those in the `java.sql.ResultSet` interface is intended to have the same behavior. Methods with new names are intended to have new behavior. Note, however, that all new methods can be implemented in terms of calls to other methods in the `java.sql.ResultSet` interface. The primary difference between this interface and the `java.sql.ResultSet` interface is the addition of getter methods that throw exceptions on fetch of null primitives, and the omission of named getters and result set metadata.

In an actual implementation of the SQLJ runtime, a class implementing the RTResultSet interface also maintains a runtime type map object. This type map is a java.util.Map object that contains type mapping information as specified in [JDBC]. It is provided to the RTResultSet object at the time of its creation in an implementation-defined manner, and is used for subsequent invocations of `getObject()`.

By partitioning new methods into a different namespace, it is possible for a JDBC driver to implement both the `java.sql.ResultSet` interface and the RTResultSet interface, allowing more efficient runtime performance in both the dynamic and static case.

The following tables describe the correspondence between some of the methods of the `java.sql.ResultSet` interface and methods of the RTResultSet interface.

**Table 11 — Methods retained from java.sql.ResultSet**

| Method Retained | |
|---|---|
| next() | |
| close() | |
| getArray(int) | |
| getBlob(int) | |
| getClob(int) | |
| getWarnings() | |
| clearWarnings() | |
| getBytes(int) | |
| getCursorName() | |
| getDate(int) | |



 

| Method Retained | |
|---|---|
| getTime(int) | |
| getTimestamp(int) | |
| getString(int) | |
| getRef(int) | |
| getURL(int) | |
| getSQLXML(int) | |
| findColumn(String) | |
| isClosed() | |

**Table 12 — Methods not retained from java.sql.ResultSet**

| Method Removed | Replacement Method |
|---|---|
| getMetaData() | |
| getArray(String) | |
| getBlob(String) | |
| getBoolean(int) get-Boolean(String) | getBooleanNoNull(int) |
| getByte(int) getByte(String) | getByteNoNull(int) |
| getCharacterStream(int) | getCharacterStreamWrapper(int) |
| getCharacterStream(String) | |
| getClob(String) | |
| getShort(int) getShort(String) | getShortNoNull(int) |
| getInt(int) getInt(String) | getIntNoNull(int) |
| getLong(int) getLong(String) | getLongNoNull(int) |
| getFloat(int) getFloat(String) | getFloatNoNull(int) |
| getDouble(int) getDou-ble(String) | getDoubleNoNull(int) |

| Method Removed | Replacement Method |
|---|---|
| getObject(int) getObject(String) | getObject(int, Class) |
| wasNull(int) | getBooleanWrapper(int) getByteWrapper(int) getShortWrapper(int) getIntWrapper(int) getLongWrapper(int) getFloatWrapper(int) getDoubleWrapper(int) |
| getBigDecimal(int,int) getBigDecimal(String,int) | getBigDecimal(int) |
| getAsciiStream(int) getAsciiStream(String) | getAsciiStreamWrapper(int) |
| getBinaryStream(int) getBinaryStream(String) | getBinaryStreamWrapper(int) |
| getUnicodeStream(int) getUnicodeStream(String) | getUnicodeStreamWrapper(int) |
| getString(String) | |
| getBytes(String) | |
| getDate(String) | |
| getTime(String) | |
| getTimestamp(String) | |
| getRef(String) | |
| getURL(String) | |
| getSQLXML(String) | |

**Table 13 — Additional methods unique to RTResultSet**

| | Additional Method |
|---|---|
| | getJDBCResultSet() |
| | isValidRow() |
| | getColumnCount() |

NOTE 57 — The getXXX(String) methods were omitted because int-based column lookup is generally more efficient. Moreover, when columns are looked up by name, the `findColumn()` method is used to find and cache the appropriate index before any getXXX calls are made.

### 13.2.5.2 Methods

#### 13.2.5.2.1 clearWarnings ()

```
public abstract void clearWarnings ( ) throws SQLException
```

After this call, getWarnings returns null until a new warning is reported for this iterator object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 13.2.5.2.2 close ()

```
public abstract void close ( ) throws SQLException
```

The `close()` method provides an immediate release of the runtime resources in the SQL-environment and the Java Virtual Machine associated with an RTResultSet object instead of waiting for this to happen when it is automatically destroyed by garbage collection.

NOTE 58 — An RTResultSet object is also automatically closed when it is destroyed by garbage collection.

#### 13.2.5.2.3 findColumn (String)

```
public abstract int findColumn ( String columnName ) throws SQLException
```

Map an RTResultSet object column name to an RTResultSet object column index. The index of the first column the name of which is a case-insensitive match of the given columnName is returned. If no such column is found, then an SQLException is thrown: *OLB-specific error — invalid column name*.

NOTE 59 — This method is called if and only if the profile EntryInfo object for the statement object that produced this RTResultSet object has a result set type with value NAMED_RESULT.

## Parameters

— columnName — the name of the column

## Returns

— The column index of the specified column

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.2.9, “NAMED_RESULT”

### 13.2.5.2.4 getArray (int)

```
public abstract java.sql.Array getArray ( int columnIndex ) throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a `java.sql.Array` object.

NOTE 60 — The implementation of the `java.sql.Array` interface is based on array locators. The accessibility of the ARRAY value through the methods of `java.sql.Array` is only guaranteed in the scope of the transaction in which the `getArray()` method was executed.

NOTE 61 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has `javaTypeName = java.sql.Array`. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”



 

#### 13.2.5.2.5 getAsciiStreamWrapper (int)

```
public abstract AsciiStream getAsciiStreamWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as an sqlj.runtime.AsciiStream object.

NOTE 62 — A column value can be retrieved as a stream of ASCII characters and then read in chunks from the stream. This method is particularly suitable for retrieving large LONGVARCHAR values. The driver will do any necessary conversion from the SQL-data's character set into ASCII.

NOTE 63 — All the data in the returned stream shall be read prior to getting the value of any other column. The next call to a get method implicitly closes the stream. Also, a stream might return 0 (zero) for `available()` whether there is data available or not.

NOTE 64 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = sqlj.runtime.AsciiStream. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value is an SQL null value, then Java null; otherwise, a Java AsciiStream object that delivers the value of the column identified by columnIndex as a stream of one-octet ASCII characters.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

#### 13.2.5.2.6 getBigDecimal (int)

```
public abstract BigDecimal getBigDecimal ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.math.BigDecimal object. Unlike the corresponding JDBC method, this method does not have a scale parameter. The value returned uses the default scale for the given column.

NOTE 65 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.math.BigDecimal. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

#### 13.2.5.2.7 getBinaryStreamWrapper (int)

```
public abstract BinaryStream getBinaryStreamWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as an sqlj.runtime.BinaryStream object. A column value can be retrieved as a stream of uninterpreted octets and then read in chunks from the stream. This method is particularly suitable for retrieving large binary strings.



 

NOTE 66 — All the data in the returned stream shall be read prior to getting the value of any other column. The next call to a get method implicitly closes the stream. Also, a stream might return 0 (zero) for `available()` whether there is data available or not.

NOTE 67 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = sqlj.runtime.BinaryStream. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value is an SQL null value, then the Java null; otherwise, a BinaryStream object that delivers the column value as a stream of uninterpreted octets.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.8 getBlob (int)

```
public abstract Blob getBlob ( int columnIndex ) throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.sql.Blob object.

NOTE 68 — The implementation of the java.sql.Blob interface is based on large object locators. The accessibility of the BLOB value through the methods of java.sql.Blob is only guaranteed in the scope of the transaction in which the getBlob method was executed.

NOTE 69 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.sql.Blob. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.9 getBooleanNoNull (int)

```
public abstract boolean getBooleanNoNull ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a Java boolean.

NOTE 70 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = boolean. Note that if the entry's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— The value of the column identified by columnIndex.



 

## Throws

— SQLNullException — if the value of the column indicated by columnIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, "getResultSetType ()"

— Subclause 13.3.2.4.8, "getResultSetInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

### 13.2.5.2.10 getBooleanWrapper (int)

```
public abstract Boolean getBooleanWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.lang.Boolean object.

NOTE 71 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.lang.Boolean. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then the columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, "getResultSetType ()"

— Subclause 13.3.2.4.8, "getResultSetInfo (int)"

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.11 getByteNoNull (int)

```
public abstract byte getByteNoNull ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a Java byte.

NOTE 72 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = byte. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— The value of the column identified by columnIndex.

## Throws

— SQLNullException — if the value of the column indicated by columnIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”



 

### 13.2.5.2.12 getBytes (int)

```
public abstract byte[] getBytes ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a Java byte[].

NOTE 73 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName=[byte. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.13 getByteWrapper (int)

```
public abstract Byte getByteWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.lang.Byte object.

NOTE 74 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.lang.Byte. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, "getResultSetType ()"

— Subclause 13.3.2.4.8, "getResultSetInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

### 13.2.5.2.14 getCharacterStreamWrapper (int)

```
public CharacterStream getCharacterStreamWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as an sqlj.runtime.CharacterStream object. A column value can be retrieved as a stream of Unicode characters and then read in chunks from the stream. This method is particularly suitable for retrieving large character strings. The driver will do any necessary conversion from the SQL character set into Unicode.

NOTE 75 — All the data in the returned CharacterStream object shall be read prior to getting the value of any other column. The next call to a get method implicitly closes the CharacterStream object. An invocation of `CharacterStream.available()` might return 0 (zero) whether there is data available or not.

NOTE 76 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = sqlj.runtime.CharacterStream. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.



 

## Returns

— If the value of the column identified by columnIndex is the SQL null value, then the Java null; otherwise, a CharacterStream object that delivers the value of the column identified by columnIndex as a stream of Unicode characters.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.15 getClob (int)

```
public abstract Clob getClob ( int columnIndex ) throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.sql.Clob object.

NOTE 77 — The implementation of the java.sql.Clob interface is based on large object locators. The accessibility of the CLOB value through the methods of java.sql.Clob is only guaranteed in the scope of the transaction in which the getClob method was executed.

NOTE 78 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.sql.Clob. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.16 getColumnCount ()

```
public abstract int getColumnCount ( ) throws SQLException
```

Determine the number of columns in this RTResultSet object. This is used to verify that the number of columns in the RTResultSet object match the number expected by a strongly typed iterator object.

NOTE 79 — This method can be implemented in JDBC using the `getColumnCount()` method of a `java.sql.ResultSet` object's MetaData object.

## Returns

— The number of columns in this RTResultSet object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 13.2.5.2.17 getCursorName ()

```
public abstract String getCursorName ( ) throws SQLException
```

Get the name of the implicit cursor used by this RTResultSet object.

In SQL, a result table is retrieved through a cursor that is named. The current row of a result can be updated or deleted using a positioned update/delete statement that references the cursor name.

JDBC drivers support this SQL feature by providing the name of the implicit cursor used by a `java.sql.ResultSet` object. The current row of a `java.sql.ResultSet` object is also the current row of this implicit cursor. This method is provided for interoperability with JDBC-based implementations.

NOTE 80 — If positioned update is not supported an SQLException is thrown.

NOTE 81 — This method is called only if the profile EntryInfo object for the statement that produced this RTResultSet object has a role with value POSITIONED.



 

## Returns

— The RTResultSet object's SQL cursor name.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.11, “getRole ()”

— Subclause 13.3.2.2.12, “POSITIONED”

### 13.2.5.2.18 getDate (int)

```
public abstract Date getDate ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.sql.Date object.

NOTE 82 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.sql.Date. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.19 getDoubleNoNull (int)

```
public abstract double getDoubleNoNull ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a Java double.

NOTE 83 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = double. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— The value of the column identified by columnIndex.

## Throws

— SQLNullException — if the value of the column indicated by columnIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”



 

### 13.2.5.2.20 getDoubleWrapper (int)

```
public abstract Double getDoubleWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.lang.Double object.

NOTE 84 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.lang.Double. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.21 getFloatNoNull (int)

```
public abstract float getFloatNoNull ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a Java float.

NOTE 85 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = float. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— The value of the column identified by columnIndex.

## Throws

— SQLNullException — if the value of the column indicated by columnIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.22 getFloatWrapper (int)

```
public abstract Float getFloatWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.lang.Float object.

NOTE 86 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.lang.Float. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.



 

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.23 getIntNoNull (int)

```
public abstract int getIntNoNull ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a Java int.

NOTE 87 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = int. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— The value of the column identified by columnIndex.

## Throws

— SQLNullException — if the value of the column indicated by columnIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.24 getIntWrapper (int)

```
public abstract Integer getIntWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.lang.Integer object.

NOTE 88 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.lang.Integer. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.25 getJDBCResultSet ()

```
public abstract ResultSet getJDBCResultSet ( )
       throws SQLException
```

Returns the `java.sql.ResultSet` object associated with this RTResultSet object. The returned ResultSet object shall have normal JDBC functionality, as defined by [JDBC] (in particular, primitive accessor methods



 

will not raise an SQLNullException when SQL null values are fetched). This method is provided to facilitate interoperability with JDBC.

If support for Feature J002, "ResultSetIterator access to JDBC ResultSet", is provided, then any synchronization between the RTResultSet object and the returned `java.sql.ResultSet` object is implementation-defined.

NOTE 89 — For maximum portability, this method should be invoked before the first `next()` method invocation on the RTResultSet object. Once the `java.sql.ResultSet` object has been produced, all operations to fetch data should be through that `java.sql.ResultSet` object.

## Returns

— A `java.sql.ResultSet` object representing this RTResultSet object.

## Throws

— SQLException: *OLB-specific error — unsupported feature* — if support for Feature J002, "ResultSetIterator access to JDBC ResultSet", is not provided

### 13.2.5.2.26 getLongNoNull (int)

```
public abstract long getLongNoNull ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a Java long.

NOTE 90 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = long. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— The value of the column identified by columnIndex.

## Throws

— SQLNullException — if the value of the column indicated by columnIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.27 getLongWrapper (int)

```
public abstract Long getLongWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.lang.Long object.

NOTE 91 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.lang.Long. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”



 

### 13.2.5.2.28 getObject (int, Class)

```
public abstract Object getObject
       ( int columnIndex, Class objectType )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.lang.Object. This method is used to read implementation-defined, user-defined data types with type SQL STRUCT, DISTINCT, JAVA_OBJECT, or OTHER.

The static type of the Java lvalue into which the object returned by the invocation of this method is assigned is passed as objectType. If the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has SQL Type STRUCT, DISTINCT, or JAVA_OBJECT, then the runtime type map **TM** of the RTResultSet object is non-null and has a map entry mapping the actual SQL type name to the Java class specified in the **Class** argument or to a subclass of that Java class. In this case, the result of `getObject()` is equivalent to the invocation of `ResultSet.getObject(columnIndex, TM)`, as defined in [JDBC]. If the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has SQL Type OTHER, then the runtime type map is null. An exception is thrown if the object returned is not assignable to an object with class objectType.

If the object type cannot be constructed or otherwise has invalid structure (such as an iterator whose named accessor methods cannot be determined), then an SQLException condition is thrown: *OLB-specific error — invalid class declaration*.

NOTE 92 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has SQL Type STRUCT, DISTINCT, JAVA_OBJECT, or OTHER. In such cases, the javaTypeName indicates the expected Java Class of the object; the class cannot be handled by any other getXXX method defined by this statement. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

— objectType — the class of the Java lvalue into which the returned value will be assigned

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.29 getRef (int)

```
public abstract Ref getRef ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.sql.Ref object.

NOTE 93 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.sql.Ref. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”



 

### 13.2.5.2.30 getShortNoNull (int)

```
public abstract short getShortNoNull ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a Java short.

NOTE 94 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = short. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— The value of the column identified by columnIndex.

## Throws

— SQLNullException — if the value of the column indicated by columnIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.31 getShortWrapper (int)

```
public abstract Short getShortWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.lang.Short object.

NOTE 95 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.lang.Short. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the

result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, "getResultSetType ()"

— Subclause 13.3.2.4.8, "getResultSetInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

### 13.2.5.2.32 getString (int)

```
public abstract String getString ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a Java String.

NOTE 96 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.lang.String. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.



 

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.33 getSQLXML(int)

```
public abstract java.sql.SQLXML getSQLXML ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a `java.sql.SQLXML` object.

NOTE 97 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = `java.sql.SQLXML`. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.34 getTime (int)

```
public abstract Time getTime ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.sql.Time object.

NOTE 98 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.sql.Time. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.35 getTimestamp (int)

```
public abstract Timestamp getTimestamp ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as a java.sql.Timestamp object.



 

NOTE 99 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.sql.Timestamp. Note that if the entry's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, is the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the column identified by columnIndex is an SQL null value, then the Java null; otherwise, the value of the column identified by columnIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, "getResultSetType ()"

— Subclause 13.3.2.4.8, "getResultSetInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

### 13.2.5.2.36 getUnicodeStreamWrapper (int)

```
public abstract UnicodeStream getUnicodeStreamWrapper ( int columnIndex )
       throws SQLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as an sqlj.runtime.UnicodeStream object. A column value can be retrieved as a stream of Unicode characters and then read in chunks from the stream. This method is particularly suitable for retrieving large character strings. The driver will do any necessary conversion from the SQL character set into Unicode.

NOTE 100 — All the data in the returned UnicodeStream object shall be read prior to getting the value of any other column. The next call to a get method implicitly closes the UnicodeStream object. An invocation of `UnicodeStream.available()` might return 0 (zero) whether there is data available or not.

NOTE 101 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = sqlj.runtime.UnicodeStream. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value is the SQL null value, then the Java null; otherwise, a Java UnicodeStream object that delivers the value of the column identified by columnIndex as a stream of two-octet Unicode characters.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.10, “getResultSetType ()”

— Subclause 13.3.2.4.8, “getResultSetInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.5.2.37 getURL (int)

```
public abstract java.net.URL getURL( int columnIndex )
       throws SQLException, java.net.MalformedURLException
```

Get the value of the column identified by columnIndex in the current row of this RTResultSet object as an `java.net.URL` object.

NOTE 102 — An invocation of this method is generated by the translator if and only if the result TypeInfo object for the current column in the profile EntryInfo object for the statement that produced this RTResultSet object has javaTypeName = java.net.URL. Note that if the EntryInfo object's resultSetType is POSITIONED_RESULT, then columnIndex can be used directly to find the result TypeInfo object. Otherwise, if the EntryInfo object's resultSetType is NAMED_RESULT, then the name of the current column shall be used to find the result TypeInfo object with the same name.

## Parameters

— columnIndex — the first column is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the result column identified by columnIndex is not the SQL null value, then the value of the result column identified by columnIndex; otherwise, the Java null.



 

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

— java.net.MalformedURLException — if the DATALINK URL value cannot be used to construct a java.net.URL object.

### 13.2.5.2.38 getWarnings ()

```
public abstract SQLWarning getWarnings ( ) throws SQLException
```

The first warning reported by calls on this iterator is returned. Subsequent iterator warnings will be chained to this SQLWarning.

The warning chain is automatically cleared each time a new row is read.

NOTE 103 — This warning chain only covers warnings caused by iterator methods. Any warning caused by statement execution (such as fetching **OUT** parameters) will be chained on the ExecutionContext object.

## Returns

— If there are no errors, then null; otherwise, the first SQLWarning.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 13.2.5.2.39 isClosed ()

```
public abstract boolean isClosed ( ) throws SQLException
```

Test to see if this RTResultSet object is closed.

## Returns

— If the RTResultSet object is closed, then **true**; otherwise, **false**.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 13.2.5.2.40 isValidRow ()

```
public abstract boolean isValidRow ( )
       throws SQLException
```

Returns **true** if the RTResultSet object is currently positioned on a row, **false** otherwise. In particular, **false** is returned if the RTResultSet object is currently positioned before its first row, or after its last row.

## Returns

— If the RTResultSet object is positioned on a row, then **true**; otherwise, **false**.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 13.2.5.2.41 next ()

```
public abstract boolean next ( ) throws SQLexception
```

An RTResultSet object is initially positioned before its first row; the first call to `next()` makes the first row the current row, the second call makes the second row the current row, *etc*.

If an InputStream object from the previous row is open, it is implicitly closed.

## Returns

— If the new current row is valid, then **true**; otherwise, **false**. If there are no more rows, then **false**.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 13.2.6 sqlj.runtime.profile.RTStatement

#### 13.2.6.1 Interface Overview

```
public interface RTStatement
```

This interface defines the operations used to execute an SQL-statement described by a profile EntryInfo object. It is based strongly on the `java.sql.Statement`, `PreparedStatement`, and `CallableStatement` interfaces, and can be implemented using one of these interfaces. In general, any method with the same name as



 

one of those in the `java.sql.Statement` interfaces (*i.e.*, Statement, PreparedStatement, and CallableStatement) is intended to have the same semantic behavior. Methods with new names are intended to have new behavior. Note, however, that all new methods can be implemented in terms of calls to other methods in the `java.sql.Statement` interfaces. The primary difference between this interface and the `java.sql.Statement` interfaces is the addition of getter methods that throw exceptions on fetch of null primitives, and a redefinition of statement close semantics.

In an actual implementation of the SQLJ runtime, a class implementing the RTStatement interface also maintains a runtime type map object. This type map is a java.util.Map object that contains type mapping information as specified in [JDBC]. It is provided to the RTStatement object at the time of its creation in an implementation-defined manner, and is used for subsequent invocations of `getObject()` and `setObject()`. It is also passed in an implementation-defined manner to any RTResultSet object created as a result of the execution of the RTStatement object.

By partitioning new methods into a different namespace, it is possible for a JDBC driver to implement both the `java.sql.Statement` interfaces and this interface, allowing more efficient runtime performance in both the dynamic and static case.

By default, the connection context user identifier of the connection context object associated with the ConnectedProfile object that created the RTStatement object is used for privilege checking during execution of an RTStatement object. If a customized profile user identifier has been provided during profile customization, then that identifier is used as the current user identifier during execution of an RTStatement object instead of the connection context user identifier.

The following tables describe the correspondence between some of the methods of the `java.sql.Statement` interfaces and methods of the RTStatement interface. Table 14, "Methods retained from java.sql.Statement", identifies methods that are retained from `java.sql.Statement`. Table 15, "Methods not retained from java.sql.Statement", identifies methods not retained from java.sql.Statement; most are simply removed, while one is replaced by a new method defined herein. Table 16, "Methods retained from java.sql.PreparedStatement", identifies methods that are retained from `java.sql.PreparedStatement`. Table 17, "Methods not retained from java.sql.PreparedStatement", identifies methods not retained from `java.sql.PreparedStatement`; some are simply removed, while several are replaced by new methods defined herein. Table 18, "Methods retained from java.sql.CallableStatement", identifies methods that are retained from `java.sql.CallableStatement`. Table 19, "Methods not retained from java.sql.CallableStatement", identifies methods not retained from `java.sql.CallableStatement`; some are simply removed, while several are replaced by new methods defined herein. Table 20, "Additional methods unique to RTStatement", identifies methods that are unique to RTStatement.

**Table 14 — Methods retained from java.sql.Statement**

| Method Retained | |
|---|---|
| cancel() | |
| getMaxFieldSize() | |
| setMaxFieldSize(int) | |
| getMaxRows() | |
| setMaxRows(int) | |

| Method Retained | |
|---|---|
| getMoreResults(int) | |
| getQueryTimeout() | |
| setQueryTimeout(int) | |
| getUpdateCount() | |
| getWarnings() | |
| clearWarnings() | |
| getResultSet() | |
| clearBatch() | BatchContext.clearBatch() |
| executeBatch() | BatchContext.executeBatch() |
| addBatch(String) | |

**Table 15 — Methods not retained from java.sql.Statement**

| Method Removed | Replacement Method |
|---|---|
| setEscapeProcessing(boolean) | |
| close() | executeComplete() |
| execute(String) | |
| executeQuery(String) | |
| executeUpdate(String) | |
| getMoreResults() | |
| setCursorName(String) | |

**Table 16 — Methods retained from java.sql.PreparedStatement**

| Method Retained | |
|---|---|
| addBatch() | getBatchContext() |
| execute() | |
| executeUpdate() | |



 

| Method Retained | |
|---|---|
| setArray(int, Array) | |
| setBigDecimal(int, BigDecimal) | |
| setBlob(int, Blob) | |
| setBoolean(int, boolean) | |
| setByte(int, byte) | |
| setBytes(int, byte[]) | |
| setClob(int, Clob) | |
| setDate(int, java.sql.Date) | |
| setDouble(int, double) | |
| setFloat(int, float) | |
| setInt(int, int) | |
| setLong(int, long) | |
| setObject(int, Object) | |
| setRef(int, Ref) | |
| setShort(int, short) | |
| setString(int, String) | |
| setTime(int, java.sql.Time) | |
| setTimestamp(int, java.sql.Timestamp) | |
| setURL(int, java.net.URL) | |
| setSQLXML(int, java.sql.SQLXML) | |

**Table 17 — Methods not retained from java.sql.PreparedStatement**

| Method Removed | Replacement Method |
|---|---|
| setNull(int, int) | setBooleanWrapper(int, Boolean) setByteWrapper(int, Byte) setDoubleWrapper(int, Double) setFloatWrapper(int, Float) setIntWrapper(int, Int) setLongWrapper(int, Long) setShortWrapper(int, Short) |
| setAsciiStream(int, InputStream) | setASCIIStreamWrapper(int, AsciiStream) |
| setBinaryStream(int, InputStream) | setBinaryStreamWrapper(int, BinaryStream) |
| setCharacterStream(int, Reader) | setCharacterStreamWrapper(int, CharacterStream) |
| setUnicodeStream(int, InputStream) | setUnicodeStreamWrapper(int, UnicodeStream) |
| clearParameters() | |
| setObject(int, Object, int, int) | |
| setObject(int, Object, int) | |
| executeQuery() | executeRTQuery() |

**Table 18 — Methods retained from java.sql.CallableStatement**

| Method Retained | |
|---|---|
| getBlob(int) | |
| getByte(int) | |
| getArray(int) | |
| getClob(int) | |
| getDate(int) | |
| getRef(int) | |
| getString(int) | |
| getTime(int) | |
| getTimestamp(int) | |

| Method Retained | |
|---|---|
| getURL(int) | |
| getSQLXML(int) | |

**Table 19 — Methods not retained from java.sql.CallableStatement**

| Method Removed | Replacement Method |
|---|---|
| registerOutParameter(int, int) | |
| registerOutParameter(int, int, int) | |
| getBoolean(int) | getBooleanNoNull(int) |
| getByte(int) | getByteNoNull(int) |
| getDouble(int) | getDoubleNoNull(int) |
| getFloat(int) | getFloatNoNull(int) |
| getInt(int) | getIntNoNull(int) |
| getLong(int) | getLongNoNull(int) |
| getShort(int) | getShortNoNull(int) |
| getObject(int) | getObject(int, Class) |
| wasNull() | getBooleanWrapper(int) getByteWrapper(int) getDoubleWrapper(int) getFloatWrapper(int) getIntWrapper(int) getLongWrapper(int) getShort-Wrapper(int) |

**Table 20 — Additional methods unique to RTStatement**

| | Additional Method |
|---|---|
| | getJDBCPreparedStatement() |
| | getJDBCCallableStatement() |
| | isBatchable() |
| | isBatchCompatible() |

NOTE 104 — Escape processing is handled by the implementation-dependent customization. By default, it is on, since the SQL strings stored in the profile EntryInfo object are in escaped syntax. However, a driver might remove the escape clauses before application runtime, in which case escape processing could be shut off by the statement implementation.

Execute methods that have an SQL String parameter are omitted, since the SQL string is known from the profile EntryInfo object.

The cursor name does not need to be set explicitly, since POSITIONED statements are handled by passing the iterator object itself.

The registerOutParameter methods are omitted, since the types of the **OUT** parameters are stored in the profile object and can be implicitly registered by the statement object implementation.

### 13.2.6.2 Methods

#### 13.2.6.2.1 cancel ()

```
public abstract void cancel ( ) throws SQLException
```

Cancel can be used by one thread to cancel an RTStatement object that is being executed by another thread.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 13.2.6.2.2 clearWarnings ()

```
public abstract void clearWarnings ( )
       throws SQLException
```

After this call, getWarnings returns null until a new warning is reported for this RTStatement object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 13.2.6.2.3 execute ()

```
public abstract boolean execute ( )
     throws SQLException
```

Some CALL statements return multiple results; the execute method handles these complex statements.

NOTE 105 — An invocation of this method is generated by the translator if and only if the execute type of the profile EntryInfo object for this RTStatement object has value EXECUTE and the role has a value of CALL.

## Returns

— If the statement was executed without raising an exception, then **true**; otherwise, **false**.



 

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.3, "getExecuteType ()"

— Subclause 13.3.2.2.5, "EXECUTE"

### 13.2.6.2.4 executeComplete ()

```
public abstract void executeComplete ( )
       throws SQLException
```

Called once the execution of this RTStatement object (and all the required gets) have been made. This is a guarantee that no further calls will be made to this RTStatement object by the codegen or runtime environment. Once `executeComplete()` has been called, further calls to any other method are implementation-dependent and might result in an SQLException being thrown. Additionally, if this RTStatement object has previously been added to a RTStatement object batch via `getBatchContext()`, then it should remain open and executable until either `BatchContext.executeBatch()` or `BatchContext.clearBatch()` has been called.

This method is distinguished from the JDBC `close()` method because, unlike the JDBC `close()` method, this method will not close any ResultSet objects that have been opened by this RTStatement object. If this RTStatement object is implemented using JDBC, then the underlying `java.sql.Statement` object should not be closed until all open RTResultSet objects have been explicitly closed, and the `executionComplete()` method has been called.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 13.2.6.2.5 executeRTQuery ()

```
public abstract RTResultSet executeRTQuery ( )
       throws SQLException
```

The prepared SQL query described by the profile EntryInfo object for this RTStatement object is executed and its RTResultSet object is returned. The runtime type map of the RTStatement object is passed to the newly-created RTResultSet object in an implementation-defined manner.

NOTE 106 — An invocation of this method is generated by the translator if and only if the execute type of the profile EntryInfo object for this RTStatement object has value EXECUTE_QUERY.

## Returns

— An RTResultSet object that contains the data produced by the query (never null)

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.3, “getExecuteType ()”

— Subclause 13.3.2.2.5, “EXECUTE”

### 13.2.6.2.6 executeUpdate ()

```
public abstract int executeUpdate ( )
       throws SQLException
```

Execute the SQL-statement described by the profile EntryInfo object for this RTStatement object.

NOTE 107 — An invocation of this method is generated by the translator if and only if the execute type of the profile EntryInfo object for this RTStatement object has value EXECUTE_UPDATE.

## Returns

— If the SQL-statement is INSERT, UPDATE or DELETE, then the number of rows affected by the SQL-statement; otherwise, 0 (zero).

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.3, “getExecuteType ()”

— Subclause 13.3.2.2.5, “EXECUTE”

### 13.2.6.2.7 getArray (int)

```
public abstract java.sql.Array getArray ( int parameterIndex ) throws SQLException
```

Get the value of the SQL ARRAY identified by parameterIndex as a `java.sql.Array` object.



 

NOTE 108 — The implementation of the `java.sql.Array` interface is based on array locators. The accessibility of the ARRAY value through the methods of `java.sql.Array` is only guaranteed in the scope of the transaction in which the `getArray()` method was executed.

NOTE 109 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement statement has the value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode OUT or INOUT, and `javaTypeName = java.sql.Array`.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"

— Subclause 13.3.2.4.13, "getStatementType ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.8 getBatchContext ()

```
public abstract BatchContext getBatchContext ( ) throws SQLException
```

Returns a batch context object that can be used to execute this RTStatement object as part of a batch of RTStatement objects. If this RTStatement object is compatible with the underlying batch context object as defined by `isBatchCompatible()`, it is added to the underlying batch context object. Otherwise a new batch context object is created which initially contains only this RTStatement object. Such a new batch context object is also created when the batch context object passed in the `getStatement(int,BatchContext)` method was `null`.

The result is undefined if this method is called on an RTStatement object that was not obtained by `getStatement(int,BatchContext)`, or if the RTStatement object is not batchable.

This method is called after all **`IN`** parameters and execution control attributes have been set, but before RTStatement object execution.

## Returns

— A batch context object that can be used to execute this RTStatement object as part of a batch of RTStatement objects.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.2.6.2.44, “isBatchable ()”

— Subclause 13.2.6.2.45, “isBatchCompatible ()”

— Subclause 13.2.2.2.5, “getStatement (int, BatchContext, Map)”

### 13.2.6.2.9 getBigDecimal (int)

```
public abstract BigDecimal getBigDecimal ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL NUMERIC parameter identified by parameterIndex as a java.math.BigDecimal. Unlike the corresponding JDBC method, this method does not have a scale parameter. The value returned uses the scale of the SQL data type of the given parameter.

NOTE 110 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.math.BigDecimal.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”



 

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.10 getBlob (int)

```
public abstract Blob getBlob ( int parameterIndex ) throws SQLException
```

Get the value of SQL BLOB parameter identified by parameterIndex as a java.sql.Blob object.

NOTE 111 — The implementation of the java.sql.Blob interface is based on large object locators. The accessibility of the BLOB value through the methods of java.sql.Blob is only guaranteed in the scope of the transaction in which the `getBlob()` method was executed.

NOTE 112 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has the value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode OUT or INOUT, and javaTypeName = java.sql.Blob.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.11 getBooleanNoNull (int)

```
public abstract boolean getBooleanNoNull ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL BOOLEAN parameter identified by parameterIndex as a Java boolean.

NOTE 113 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=boolean.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— The value of the parameter identified by parameterIndex.

## Throws

— SQLNullException — if the value of the parameter indicated by parameterIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"

— Subclause 13.3.2.4.13, "getStatementType ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.12 getBooleanWrapper (int)

```
public abstract Boolean getBooleanWrapper ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL BOOLEAN parameter identified by parameterIndex as a java.lang.Boolean.

NOTE 114 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.lang.Boolean.



 

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLNullException — if the parameter identified by parameterIndex has the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"

— Subclause 13.3.2.4.13, "getStatementType ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.13 getByteNoNull (int)

```
public abstract byte getByteNoNull ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL_TINYINT parameter identified by parameterIndex as a Java byte.

NOTE 115 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=byte.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— The value of the parameter identified by parameterIndex.

## Throws

— SQLNullException — if the value of the parameter indicated by parameterIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"

— Subclause 13.3.2.4.13, "getStatementType ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.14 getBytes (int)

```
public abstract byte[] getBytes ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL VARBINARY parameter identified by parameterIndex as an array of Java bytes.

NOTE 116 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=[byte.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"



 

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.15 getByteWrapper (int)

```
public abstract Byte getByteWrapper ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL TINYINT parameter identified by parameterIndex as a java.lang.Byte.

NOTE 117 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.lang.Byte.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc.*

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.16 getClob (int)

```
public abstract Clob getClob ( int parameterIndex ) throws SQLException
```

Get the value of the SQL CLOB parameter identified by parameterIndex as a java.sql.Clob object.

NOTE 118 — The implementation of the java.sql.Clob interface is based on large object locators. The accessibility of the CLOB value through the methods of java.sql.Clob is only guaranteed in the scope of the transaction in which the `getClob()` method was executed.

NOTE 119 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has the value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode OUT or INOUT, and javaTypeName = java.sql.Clob.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.17 getDate (int)

```
public abstract Date getDate ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL DATE parameter identified by parameterIndex as a java.sql.Date.

NOTE 120 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.sql.Date.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.



 

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"

— Subclause 13.3.2.4.13, "getStatementType ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.18 getDoubleNoNull (int)

```
public abstract double getDoubleNoNull ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL DOUBLE PRECISION parameter identified by parameterIndex as a Java double.

NOTE 121 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=double.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— The value of the parameter identified by parameterIndex.

## Throws

— SQLNullException — if the value of the parameter indicated by parameterIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.19 getDoubleWrapper (int)

```
public abstract Double getDoubleWrapper ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL DOUBLE PRECISION parameter identified by parameterIndex as a java.lang.Double.

NOTE 122 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.lang.Double.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.20 getFloatNoNull (int)

```
public abstract float getFloatNoNull ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL FLOAT parameter identified by parameterIndex as a Java float.

NOTE 123 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=float.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— The value of the parameter identified by parameterIndex.

## Throws

— SQLNullException — if the value of the parameter indicated by parameterIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.21 getFloatWrapper (int)

```
public abstract Float getFloatWrapper ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL FLOAT parameter identified by parameterIndex as a java.lang.Float.

NOTE 124 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.lang.Float.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"

— Subclause 13.3.2.4.13, "getStatementType ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.22 getIntNoNull (int)

```
public abstract int getIntNoNull ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL INTEGER parameter identified by parameterIndex as a Java int.

NOTE 125 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=int.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— The value of the parameter identified by parameterIndex.

## Throws

— SQLNullException — if the value of the parameter indicated by parameterIndex is the SQL null value



 

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.23 getIntWrapper (int)

```
public abstract Integer getIntWrapper ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL INTEGER parameter identified by parameterIndex as a java.lang.Integer.

NOTE 126 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.lang.Integer.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.24 getJDBCCallableStatement ()

```
public abstract CallableStatement getJDBCCallableStatement ( )
       throws SQLException
```

Returns a representation of this RTStatement object as a `java.sql.CallableStatement` object. Operations performed on the returned `CallableStatement` object affect the state of this RTStatement object as well.

## Returns

— A `java.sql.CallableStatement` object representing this RTStatement object.

## Throws

— SQLException — if this RTStatement object cannot be represented as a `java.sql.CallableStatement` object.

### 13.2.6.2.25 getJDBCPreparedStatement ()

```
public abstract PreparedStatement getJDBCPreparedStatement ( )
       throws SQLException
```

Returns a representation of this RTStatement object as a `java.sql.PreparedStatement` object. Operations performed on the returned `PreparedStatement` object affect the state of this RTStatement object as well.

## Returns

— A `java.sql.PreparedStatement` object representing this RTStatement object.

## Throws

— SQLException — if this RTStatement object cannot be represented as a `java.sql.PreparedStatement` object

### 13.2.6.2.26 getLongNoNull (int)

```
public abstract long getLongNoNull ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL BIGINT parameter identified by parameterIndex as a Java long.

NOTE 127 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=long.



 

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— The value of the parameter identified by parameterIndex.

## Throws

— SQLNullException — if the value of the parameter indicated by parameterIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"

— Subclause 13.3.2.4.13, "getStatementType ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.27 getLongWrapper (int)

```
public abstract Long getLongWrapper ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL BIGINT parameter identified by parameterIndex as a java.lang.Long.

NOTE 128 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.lang.Long.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.28 getMaxFieldSize ()

```
public abstract int getMaxFieldSize ( )
       throws SQLException
```

The maxFieldSize limit (in bytes) is the maximum amount of data returned for any column value; it only applies to binary string and character string (BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR) columns. These columns can be fetched into Java String, Byte array, or stream objects. If the limit is exceeded, the excess data is discarded. The default maxFieldSize is 0 (zero).

## Returns

— The maxFieldSize limit of this RTStatement object; 0 (zero) means unlimited.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 13.2.6.2.29 getMaxRows ()

```
public abstract int getMaxRows ( )
      throws SQLException
```

Returns the maximum number of rows that can be contained by a ResultSet object or by an RTResultSet object created by executing this RTStatement object. If this maxRows limit is exceeded, then the excess rows are dropped. The default maxRows value is 0 (zero).



 

## Returns

— The maxRows limit of this RTStatement object; 0 (zero) means unlimited.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 13.2.6.2.30 getMoreResults (int)

```
public abstract boolean getMoreResults ( int closeType )
       throws SQLException
```

Moves to an RTStatement's next result. It returns **true** if this result is a `java.sql.ResultSet` object. `getMoreResults(int)` also optionally closes `java.sql.ResultSet` objects obtained with `getResultSet()`. There are no more results if and only if `(!getMoreResults(int) && (getUpdateCount() == -1))`.

If the constant `java.sql.Statement.CLOSE_CURRENT_RESULT` is passed, then the `java.sql.ResultSet` object returned by the last call to `getResultSet()` against the currently registered RTStatement is closed. If the constant `java.sql.Statement.CLOSE_ALL_RESULTS` is passed, then all open `java.sql.ResultSet` objects previously obtained from the currently registered RTStatement are closed. If the constant `java.sql.Statement.KEEP_CURRENT_RESULT` is passed, then the last `java.sql.ResultSet` object obtained from the currently registered RTStatement is left open.

NOTE 129 — Invocation of this method occurs as a result of the <embedded SQL Java program> having invoked `getNextResultSet(int)` against the ExecutionContext for which this RTStatement is the currently registered RTStatement.

## Returns

— If the next result is a ResultSet object, then **true**; if it is an update count or there are no more results, then **false**.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 12.3.4.4.13, “getNextResultSet (int)”

— Subclause 13.2.6.2.3, “execute ()”

— Subclause 13.3.2.4.3, “getExecuteType ()”

— Subclause 13.3.2.2.5, “EXECUTE”

### 13.2.6.2.31 getObject (int, Class)

```
public abstract Object getObject
       ( int parameterIndex, Class objectType )
       throws SQLException
```

Get the value of the parameter identified by parameterIndex as a java.lang.Object object. This method is used to fetch implementation-defined instances of user-defined types with SQL Type STRUCT, DISTINCT, JAVA_OBJECT, or OTHER.

The objectType parameter gives the static type of the Java lvalue to which the value of the parameter indicated by parameterIndex is to be assigned. If the TypeInfo profile EntryInfo object for the parameter has SQL Type STRUCT, DISTINCT, or JAVA_OBJECT, then the runtime type map **TM** of the RTStatement object is non-null and has a map entry mapping the actual SQL type name to the Java class specified in the **Class** argument or to a subclass of that Java class. In this case, the result of getObject is equivalent to the invocation of getObject(columnIndex, **TM**), as defined in [JDBC]. If the TypeInfo profile EntryInfo object for the parameter has SQL Type OTHER, then the runtime type map is null. An exception is thrown if the object returned is not assignable to an lvalue with static type objectType.

If an object of type objectType cannot be constructed or otherwise has invalid structure (as would be the case with an iterator whose named accessor methods cannot be determined), then an SQLException condition is thrown: *OLB-specific error — invalid class declaration.*

NOTE 130 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode OUT or INOUT, and SQLType STRUCT, DISTINCT, JAVA_OBJECT, or OTHER. In such cases, the javaTypeName of the TypeInfo profile entry indicates the expected Java Class of the object; the class cannot be handled by any other getXXX method defined by this RTStatement object. Accordingly, this method is used as the catch-all for any unrecognized types.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— objectType — the class of the Java lvalue to which the value of the parameter indicated by parameterIndex is to be assigned.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the class of the object returned is not assignment compatible with the given objectType class, or if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”



 

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

#### 13.2.6.2.32 getQueryTimeout ()

```
public abstract int getQueryTimeout ( )
       throws SQLException
```

The queryTimeout limit is the number of seconds that the SQLJ runtime implementation will wait for an invocation of `execute()` to complete. If the limit is exceeded, an SQLException is thrown. The default queryTimeout is 0 (zero).

### Returns

— The queryTimeout limit of this RTStatement object in seconds; 0 (zero) means unlimited.

### Throws

— SQLException — if the SQL-implementation raises an exception condition.

#### 13.2.6.2.33 getRef (int)

```
public abstract Ref getRef ( int parameterIndex )
       throws SQLException
```

Get the value of an SQL REF parameter as a java.sql.Ref object.

NOTE 131 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.sql.Ref.

### Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

### Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.34 getResultSet ()

```
public abstract ResultSet getResultSet ( )
       throws SQLException
```

Returns the current result of this RTStatement object as a ResultSet object. It is only called once per result if using the `execute()` method.

NOTE 132 — Invocation of this method occurs as a result of the <embedded SQL Java program> having invoked `getNextResultSet(int)` against the ExecutionContext for which this RTStatement is the currently registered RTStatement.

## Returns

— If the result of this RTStatement object is an update count or there are no more results, then null; otherwise, the current result of this RTStatement object as a ResultSet object.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.2.6.2.3, “execute ()”

— Subclause 13.3.2.4.3, “getExecuteType ()”

— Subclause 13.3.2.2.5, “EXECUTE”



 

### 13.2.6.2.35 getShortNoNull (int)

```
public abstract short getShortNoNull ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL SMALLINT parameter identified by parameterIndex as a Java short.

NOTE 133 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=short.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— The value of the parameter identified by parameterIndex.

## Throws

— SQLNullException — if the value of the parameter indicated by parameterIndex is the SQL null value

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"

— Subclause 13.3.2.4.13, "getStatementType ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.36 getShortWrapper (int)

```
public abstract Short getShortWrapper ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL SMALLINT parameter identified by parameterIndex as a java.lang.Short.

NOTE 134 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.lang.Short.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"

— Subclause 13.3.2.4.13, "getStatementType ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.37 getString (int)

```
public abstract String getString ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL character string (CHAR, VARCHAR, or LONGVARCHAR) parameter identified by parameterIndex as a Java String.

NOTE 135 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT and javaTypeName=java.lang.String.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.



 

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.38 getSQLXML (int)

```
public abstract java.sql.SQLXML getSQLXML ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL XML parameter identified by parameterIndex as a `java.sql.SQLXML` object.

NOTE 136 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT and javaTypeName=`java.sql.SQLXML`.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is an SQL null value, then the Java null; otherwise, the value of the parameter identified by parameterIndex.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.39 getTime (int)

```
public abstract Time getTime ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL TIME parameter identified by parameterIndex as a java.sql.Time.

NOTE 137 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.sql.Time.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.2.2.3, “CALLABLE_STATEMENT”

— Subclause 13.3.2.4.13, “getStatementType ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.40 getTimestamp (int)

```
public abstract Timestamp getTimestamp ( int parameterIndex )
       throws SQLException
```

Get the value of the SQL TIMESTAMP parameter identified by parameterIndex as a java.sql.Timestamp.

NOTE 138 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.sql.Timestamp.



 

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.2.2.3, "CALLABLE_STATEMENT"

— Subclause 13.3.2.4.13, "getStatementType ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.41 getUpdateCount ()

```
public abstract int getUpdateCount ( )
       throws SQLException
```

Returns the current result of this RTStatement object as an update count; if the result is a ResultSet object or there are no more results, –1 is returned. It is only called once per result.

NOTE 139 — An invocation of this method is generated by the translator if and only if the execute type of the profile EntryInfo object for this RTStatement object has value EXECUTE.

## Returns

— If the current result of this RTStatement object is a ResultSet object or there are no more results, then –1; otherwise, the current result as an update count.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.2.6.2.3, “execute ()”

— Subclause 13.3.2.4.3, “getExecuteType ()”

— Subclause 13.3.2.2.5, “EXECUTE”

### 13.2.6.2.42 getURL ()

```
public abstract int getURL ( int parameterIndex )
       throws SQLException, java.net.MalformedURLException
```

Get the value of an SQL DATALINK parameter identified by parameterIndex as a java.net.URL object.

NOTE 140 — An invocation of this method is generated by the translator if and only if the statement type of the profile EntryInfo object for this RTStatement object has value CALLABLE_STATEMENT and the parameter TypeInfo object at parameterIndex in the EntryInfo object has mode=OUT or INOUT, and javaTypeName=java.net.URL.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL implementation raises an exception condition.

— java.net.MalformedURLException — if the DATALINK URL value cannot be used to construct a java.net.URL object.

### 13.2.6.2.43 getWarnings ()

```
public abstract SQLWarning getWarnings ( )
       throws SQLException
```

The first warning reported by invocations of methods on this Statement object is returned. A Statement object's execute methods clear its SQLWarning chain. Subsequent Statement warnings will be chained to this SQLWarning.

The warning chain is automatically cleared each time `execute()`, `executeRTQuery()`, or `executeUpdate()` is invoked on this RTStatement object.



 

NOTE 141 — If a ResultSet object is being processed, then any warnings associated with ResultSet reads will be chained on the ResultSet object and made available to the client on the associated iterator object.

## Returns

— If there is any outstanding SQLWarning, then the first SQLWarning; otherwise, null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.2.6.2.3, "execute ()"

— Subclause 13.3.2.4.3, "getExecuteType ()"

— Subclause 13.3.2.2.5, "EXECUTE"

### 13.2.6.2.44 isBatchable ()

```
public abstract boolean isBatchable ( ) throws SQLException
```

Returns **true** if this RTStatement object is able to be added to a statement batch for deferred execution, **false** otherwise. Batchable RTStatement objects are typically (but not exclusively) DDL, DML and invocations of SQL-invoked procedures with no **OUT** parameters. If this RTStatement object returns **OUT** parameters or produces one or more side-channel result sets, then **false** is returned.

If statement batching is not supported by the runtime implementation, this method returns **false**.

Use the method `isBatchCompatible()` to determine whether this RTStatement object is compatible with the batch context object passed when this RTStatement object was created.

This method is called after all **IN** parameters and execution control attributes have been set, but before RTStatement object execution.

## Returns

— If able to be batched, then **true**; otherwise, **false**.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.2.6.2.45, "isBatchCompatible ()"

— Subclause 13.2.6.2.8, “getBatchContext ()”

### 13.2.6.2.45 isBatchCompatible ()

```
public abstract boolean isBatchCompatible ( ) throws SQLException
```

Returns **true** if this RTStatement object is compatible with the underlying batch context object, and **false** otherwise.

The underlying batch context object is the batch context object that was passed to `ConnectedProfile.getStatement()` when this RTStatement object was created. If no such batch context object was passed, **false** is returned. The behavior of this method is undefined in the following cases.

— The RTStatement object was not obtained with `getStatement(int,BatchContext)`.

— The RTStatement object is not batchable.

In general, RTStatement objects with one or more **IN** parameters are only compatible with batch context objects that contain other instances of the same RTStatement object. RTStatement objects without **IN** parameters are only compatible with batch context objects that contain other RTStatement objects without **IN** parameters.

This method is called after all **IN** parameters and execution control attributes have been set, but before RTStatement object execution.

## Returns

— If compatible with the underlying batch context object, then **true**; otherwise, **false**.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.2.6.2.44, “isBatchable ()”

— Subclause 13.2.6.2.8, “getBatchContext ()”

— Subclause 13.2.2.2.5, “getStatement (int, BatchContext, Map)”

### 13.2.6.2.46 aetArray (int, Array)

```
public abstract void setArray ( int parameterIndex, java.sql.Array x ) throws SQLException
```

Set the parameter identified by parameterIndex to a `java.sql.Array` object.



 

NOTE 142 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName = java.sql.Array.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Returns

— If the value of the parameter identified by parameterIndex is not the SQL null value, then the value of the parameter identified by parameterIndex; otherwise, the Java null.

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, "getParamInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.47 setAsciiStreamWrapper (int, AsciiStream)

```
public abstract void setAsciiStreamWrapper ( int paramIndex, AsciiStream x )
       throws SQLException
```

Set the parameter identified by parameterIndex to an sqlj.runtime.AsciiStream value. The driver converts this to an SQL character string value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

If a very long ASCII character string is input to a character string parameter, it might be more practical to send it via a java.io.InputStream. JDBC reads the data from the stream as needed, until it reaches the end of the stream. The JDBC driver does any necessary conversion from ASCII to the SQL-data's character set.

NOTE 143 — The AsciiStream class implements java.io.InputStream, and adds a Java field, length, which is used to determine the number of octets in the stream. The AsciiStream class typically wraps a standard Java stream class or a custom subclass that implements the InputStream interface.

NOTE 144 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=sqlj.runtime.AsciiStream.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, "getParamInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

### 13.2.6.2.48 setBigDecimal (int, BigDecimal)

```
public abstract void setBigDecimal
        ( int parameterIndex, BigDecimal x )
      throws SQLException
```

Set the parameter identified by parameterIndex to a java.math.Bigdecimal value. The driver converts this to an SQL NUMERIC value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 145 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.math.BigDecimal.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, "getParamInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"



 

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.49 setBinaryStreamWrapper (int, BinaryStream)

```
public abstract void setBinaryStreamWrapper ( int paramIndex, BinaryStream x )
       throws SQLException
```

Set the parameter identified by parameterIndex to an sqlj.runtime.BinaryStream value. The driver converts this to an SQL binary string value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

If a very large binary value is input to a binary string parameter, it might be more practical to send it via a java.io.InputStream. JDBC will read the data from the stream as needed, until it reaches the end of the stream.

NOTE 146 — The BinaryStream class implements java.io.InputStream, and adds a Java field, length, which is used to determine the number of octets in the stream. The BinaryStream class typically wraps a standard Java stream class or a custom subclass that implements the InputStream interface.

NOTE 147 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=sqlj.runtime.BinaryStream.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.6.2.50 setBlob (int, Blob)

```
public abstract void setBlob ( int parameterIndex, Blob x ) throws SQLException
```

Set the parameter identified by parameterIndex to a Java Blob object.

NOTE 148 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName = java.sql.Blob.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, "getParamInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.51 setBoolean (int, boolean)

```
public abstract void setBoolean
       ( int parameterIndex, boolean x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a Java boolean value. The driver converts this to an SQL BOOLEAN value.

NOTE 149 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=boolean.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, "getParamInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"



 

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.52 setBooleanWrapper (int, Boolean)

```
public abstract void setBooleanWrapper
       ( int parameterIndex, Boolean x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.lang.Boolean value. The driver converts this to an SQL BOOLEAN value. If the given value is Java null, then the parameter is set to the SQL null value.

NOTE 150 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.lang.Boolean.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc.*

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.53 setByte (int, byte)

```
public abstract void setByte
     ( int parameterIndex, byte x )
     throws SQLException
```

Set the parameter identified by parameterIndex to a Java byte value. The driver converts this to an SQL TINYINT value.

NOTE 151 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=byte.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, "getParamInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"

— Subclause 13.3.6.4.3, "getMode ()"

### 13.2.6.2.54 setBytes (int, byte)

```
public abstract void setBytes
       ( int parameterIndex, byte x[] )
       throws SQLException
```

Set the parameter identified by parameterIndex to a Java array of bytes. The driver converts this to an SQL binary string (VARBINARY or LONGVARBINARY, depending on the argument's size relative to the driver's limits on VARBINARY values). If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 152 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=[byte.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, "getParamInfo (int)"



 

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.55 setByteWrapper (int, Byte)

```
public abstract void setByteWrapper
       ( int parameterIndex, Byte x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.lang.Byte value. The driver converts this to an SQL TINYINT value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 153 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.lang.Byte.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.56 setCharacterStreamWrapper (int, CharacterStream)

```
public void setCharacterStreamWrapper ( int columnIndex, CharacterStream x )
       throws SQLException
```

Set the parameter identified by parameterIndex to an sqlj.runtime.CharacterStream object. The driver converts this to an SQL LONGVARCHAR value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

If a very large Unicode value is input to a LONGVARCHAR parameter, it might be more practical to send it as an instance of java.io.Reader. JDBC will read the data from the stream as needed, until it reaches the end of the stream. The JDBC driver will do any necessary conversion from Unicode to the appropriate SQL character set.

NOTE 154 — The CharacterStream class implements java.io.Reader, and adds a Java field, length, which is used to determine the number of characters in the stream. The CharacterStream class typically wraps a standard Java Reader object or an instance of a custom subclass that implements the Reader interface.

NOTE 155 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and has javaTypeName = sqlj.runtime.CharacterStream.

## Parameters

— parameterIndex — the first column is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.6.2.57 setClob (int, Clob)

```
public abstract void setClob ( int parameterIndex, Clob x ) throws SQLException
```

Set the parameter identified by parameterIndex to a Java Clob object.

NOTE 156 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName = java.sql.Clob.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.



 

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.58 setDate (int, Date)

```
public abstract void setDate
       ( int parameterIndex, Date x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.sql.Date value. The driver converts this to an SQL DATE value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 157 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.sql.Date.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.59 setDouble (int, double)

```
public abstract void setDouble
```

```
( int parameterIndex, double x )
throws SQLException
```

Set the parameter identified by parameterIndex to a Java double value. The driver converts this to an SQL DOUBLE PRECISION value.

NOTE 158 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=double.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.60 setDoubleWrapper (int, Double)

```
public abstract void setDoubleWrapper
       ( int parameterIndex, Double x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.lang.Double value. The driver converts this to an SQL DOUBLE PRECISION value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 159 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.lang.Double.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex



 

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.61 setFloat (int, float)

```
public abstract void setFloat
       ( int parameterIndex, float x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a Java float value. The driver converts this to an SQL FLOAT value.

NOTE 160 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=float.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.62 setFloatWrapper (int, Float)

```
public abstract void setFloatWrapper
       ( int parameterIndex, Float x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.lang.Float value. The driver converts this to an SQL FLOAT value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 161 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.lang.Float.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.63 setInt (int, int)

```
public abstract void setInt
       ( int parameterIndex, int x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a Java int value. The driver converts this to an SQL INTEGER value.

NOTE 162 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=int.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.



 

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.64 setIntWrapper (int, Integer)

```
public abstract void setIntWrapper
       ( int parameterIndex, Integer x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.lang.Integer value. The driver converts this to an SQL INTEGER value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 163 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.lang.Integer.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.65 setLong (int, long)

```
public abstract void setLong
       ( int parameterIndex, long x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a Java long value. The driver converts this to an SQL BIGINT value.

NOTE 164 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=long.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.66 setLongWrapper (int, Long)

```
public abstract void setLongWrapper
       ( int parameterIndex, Long x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.lang.Long value. The driver converts this to an SQL BIGINT value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 165 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.lang.Long.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.



 

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.67 setMaxFieldSize (int)

```
public abstract void setMaxFieldSize ( int max )
       throws SQLException
```

The maxFieldSize limit (in bytes) is the maximum amount of data returned for any column value; it only applies to binary string and character string (BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR) columns. Such columns can be fetched into Java String, Byte array, or stream objects. If the limit is exceeded, the excess data is discarded.

## Parameters

— max — the new max column size limit; zero means unlimited

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

— SQLException: *OLB-specific error — unsupported feature* — if max is set to other than MaxFieldSize's default value and support for Feature J003, “Execution control”, is not provided

### 13.2.6.2.68 setMaxRows (int)

```
public abstract void setMaxRows ( int max )
       throws SQLException
```

Sets the maxRows limit of this RTStatement object. The maxRows limit is the maximum number of rows that can be contained by a ResultSet object or by an RTResultSet object created by executing this RTStatement object. If the limit is exceeded, then the excess rows are dropped.

## Parameters

— max — the new max row limit; zero means unlimited

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

— SQLException: *OLB-specific error — unsupported feature* — if max is set to other than MaxRow's default value and support for Feature J003, "Execution control", is not provided

### 13.2.6.2.69 setObject ()

```
public abstract void setObject
       ( int parameterIndex, Object x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a Java object value. If the TypeInfo object for this parameter in the profile EntryInfo object has SQL Type STRUCT, DISTINCT, or JAVA_OBJECT, then the runtime implementation uses this SQL Type, following the semantics described for the execution of `setObject()` in [JDBC]. Otherwise, the driver uses the type SQL OTHER. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

This method can also be used to pass implementation-defined user-defined data types, by using a Driver-specific Java type.

NOTE 166 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode IN or INOUT, and SQLType STRUCT, DISTINCT, JAVA_OBJECT, or OTHER. In such cases, the JavaTypeName indicates the expected Java Class of the object; the class cannot be handled by any other setXXX method defined by this RTStatement object. Accordingly, this method is also used as the catch-all for any unrecognized types.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, "getParamInfo (int)"

— Subclause 13.3.6.4.5, "getSQLType ()"

— Subclause 13.3.6.4.1, "getJavaTypeName ()"



 

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.70 setQueryTimeout (int)

```
public abstract void setQueryTimeout ( int seconds )
       throws SQLException
```

Sets the queryTimeout limit of this RTStatement object. The queryTimeout limit is the maximum number of seconds the SQLJ runtime implementation will wait for an invocation of `execute()` to complete. If the limit is exceeded, an SQLException is thrown.

## Parameters

— seconds — the new query timeout limit in seconds; zero means unlimited

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

— SQLException: *OLB-specific error — unsupported feature* — if seconds is set to other than QueryTimeout's default value and support for Feature J003, “Execution control”, is not provided

### 13.2.6.2.71 setRef (int, Ref)

```
public abstract void setRef
       ( int parameterIndex, Ref x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.sql.Ref value. The driver converts this to an SQL REF value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 167 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.sql.Ref.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.72 setShort (int, short)

```
public abstract void setShort
       ( int parameterIndex, short x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a Java short value. The driver converts this to an SQL SMALLINT value.

NOTE 168 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=short.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.73 setShortWrapper (int, Short)

```
public abstract void setShortWrapper
```



 

```
( int parameterIndex, Short x )
throws SQLException
```

Set the parameter identified by parameterIndex to a java.lang.Short value. The driver converts this to an SQL SMALLINT value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 169 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.lang.Short.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.74 setString (int, String)

```
public abstract void setString
       ( int parameterIndex, String x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.lang.String value. The driver converts this to an SQL character string value (CHARACTER VARYING or LONGVARCHAR, depending on the argument's size relative to the driver's limits on CHARACTER VARYING). If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 170 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.lang.String.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.75 setSQLXML (int, SQLXML)

```
public abstract void setSQLXML
       ( int parameterIndex, java.sql.SQLXML x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.sql.SQLXML value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 171 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=`java.sql.SQLXML`.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”



 

### 13.2.6.2.76 setTime (int, Time)

```
public abstract void setTime
       ( int parameterIndex, Time x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.sql.Time value. The driver converts this to an SQL TIME value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 172 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.sql.Time.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.77 setTimestamp (int, Timestamp)

```
public abstract void setTimestamp
       ( int parameterIndex, Timestamp x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.sql.Timestamp value. The driver converts this to an SQL TIMESTAMP value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 173 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.sql.Timestamp.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

— Subclause 13.3.6.4.3, “getMode ()”

### 13.2.6.2.78 setUnicodeStreamWrapper (int, UnicodeStream)

```
public abstract void setUnicodeStreamWrapper ( int paramIndex, UnicodeStream x )
       throws SQLException
```

Set the parameter identified by parameterIndex to an `sqlj.runtime.UnicodeStream` value. The driver converts this to an SQL character string value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

If a very large Unicode string value is input to an SQL character string parameter, it might be more practical to send it via a java.io.InputStream. JDBC will read the data from the stream as needed, until it reaches end of file. The JDBC driver will do any necessary conversion from Unicode to the appropriate SQL character set.

NOTE 174 — The UnicodeStream class implements java.io.InputStream, and adds a Java field, length, which is used to determine the number of octets in the stream. The UnicodeStream class typically wraps a standard Java stream class or a custom subclass that implements the InputStream interface.

NOTE 175 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=sqlj.runtime.UnicodeStream.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.



 

## See Also

— Subclause 13.3.2.4.6, “getParamInfo (int)”

— Subclause 13.3.6.4.5, “getSQLType ()”

— Subclause 13.3.6.4.1, “getJavaTypeName ()”

### 13.2.6.2.79 setURL (int, URL)

```
public abstract void setURL
       ( int parameterIndex, java.net.URL x )
       throws SQLException
```

Set the parameter identified by parameterIndex to a java.net.URL value. The driver converts this to an SQL DATALINK value. If the given value is Java null, then the parameter identified by parameterIndex is set to the SQL null value.

NOTE 176 — An invocation of this method is generated by the translator if and only if the parameter TypeInfo object at parameterIndex in the profile EntryInfo object for this RTStatement object has mode=IN or INOUT, and javaTypeName=java.net.URL.

## Parameters

— parameterIndex — the first parameter is 1 (one), the second is 2, *etc*.

— x — the value of the parameter identified by parameterIndex

## Throws

— SQLException — if the SQL-implementation raises an exception condition.

### 13.2.7 sqlj.runtime.profile.SerializedProfile

#### 13.2.7.1 Interface Overview

```
public interface SerializedProfile
```

A class implementing the SerializedProfile interface is able to provide an InputStream object from which a SerializedProfile object can be read. Instances of the SerializedProfile interface can be loaded and used by the `Profile.instantiate()` method. This object provides a hook by which profile objects can be loaded by non-standard means.

NOTE 177 — As an example of where this was found useful, it was discovered that a particular version of a web browser did not support loading of a serialized object as an applet resource. In this case, the SerializedProfile object was encoded as a static string on a class implementing SerializedProfile object, and the class packaged with the applet in place of the original SerializedProfile object.

## See Also

— Subclause 13.3.3.3.11, “instantiate (Loader, InputStream)”

— Subclause 13.3.3.3.12, “instantiate (Loader, String)”

### 13.2.7.2 Methods

#### 13.2.7.2.1 getProfileAsStream ()

```
public abstract InputStream getProfileAsStream ( )
       throws IOException
```

Returns an InputStream object from which a SerializedProfile object can be read. The first object on the InputStream object returned is expected to be a SerializedProfile object.

## Returns

— An InputStream object containing a SerializedProfile object.

## Throws

— IOException — if the stream could not be created

## Conformance Rules

*None.*



 

## 13.3 SQLJ sqlj.runtime.profile Classes

### 13.3.1 sqlj.runtime.profile.DefaultLoader

#### 13.3.1.1 Class Overview

```
java.lang.Object
   |
   +--sqlj.runtime.profile.DefaultLoader

public class DefaultLoader
    extends Object
    implements Loader
```

The default profile.Loader implementation. The DefaultLoader class provides methods that implement the Loader interface by deferring to a wrapped class Loader argument.

#### 13.3.1.2 Constructors

##### 13.3.1.2.1 DefaultLoader (ClassLoader)

```
public DefaultLoader ( ClassLoader loader )
```

Creates a default profile.Loader object the implementation of which will defer to the given class loader. If the given Loader object is Java null, the system Loader object is used instead.

## Parameters

— loader — the class Loader object to use for loading classes and resources; if the system ClassLoader object should be used, then null