



BSI Standards Publication

Information technology — Database languages SQL

Part 10: Object language bindings (SQL/OLB)

National foreword

This British Standard is the UK implementation of ISO/IEC 9075-10:2023.

The UK participation in its preparation was entrusted to Technical Committee IST/40, Data management and interchange.

A list of organizations represented on this committee can be obtained on request to its committee manager.

Contractual and legal considerations

This publication has been prepared in good faith, however no representation, warranty, assurance or undertaking (express or implied) is or will be made, and no responsibility or liability is or will be accepted by BSI in relation to the adequacy, accuracy, completeness or reasonableness of this publication. All and any such responsibility and liability is expressly disclaimed to the full extent permitted by the law.

This publication is provided as is, and is to be used at the recipient's own risk.

The recipient is advised to consider seeking professional guidance with respect to its use of this publication.

This publication is not intended to constitute a contract. Users are responsible for its correct application.

© The British Standards Institution 2023
Published by BSI Standards Limited 2023

ISBN 978 0 539 21711 7

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 30 June 2023.

Amendments/corrigenda issued since publication

Date	Text affected
------	---------------

**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 9075-10:2023(E)

AVA Documents
Document Sharing Hub
© ISO/IEC 2023



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2023

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword.....	xviii
Introduction.....	xx
1 Scope.....	1
2 Normative references.....	2
3 Terms and definitions.....	3
4 Concepts.....	6
4.1 Notations and conventions.....	6
4.1.1 Notations.....	6
4.1.2 First term usage.....	6
4.1.3 Java-related conventions.....	6
4.1.4 Specification of translator-generated classes.....	7
4.2 Character strings.....	8
4.2.1 Unicode support.....	8
4.3 Embedded syntax.....	8
4.4 Introduction to SQLJ.....	8
4.4.1 Overview.....	8
4.4.2 SQL constructs.....	9
4.4.3 SQLJ clauses.....	9
4.4.4 Binary portability.....	10
4.4.4.1 Binary portability requirements.....	10
4.4.4.2 Components of binary portable applications.....	11
4.4.5 Profile overview.....	11
4.4.5.1 Introduction to profiles.....	11
4.4.5.2 EntryInfo overview.....	12
4.4.5.3 TypeInfo overview.....	14
4.4.5.4 SQLJ datatype properties.....	16
4.4.6 Host variables.....	18
4.4.7 Host expressions.....	18
4.4.8 Connection contexts.....	18
4.4.9 Default connection context.....	18
4.4.10 Schema checking using exemplar schemas.....	19
4.4.11 Using multiple SQLJ contexts and connections.....	19
4.4.12 Dynamic SQL and JDBC/SQLJ Connection interoperability.....	20
4.4.12.1 Introduction to connection interoperability.....	20
4.4.12.2 Creating an SQLJ ConnectionContext from a java.sql.Connection object.....	20
4.4.12.3 Obtaining a java.sql.Connection object from an SQLJ ConnectionContext.....	20
4.4.12.4 Connection sharing.....	20
4.4.12.5 Connection resource management.....	20

4.4.13	SQL execution control and status.	21
4.4.14	Iterators.	21
4.4.15	Input and output assignability.	22
4.4.16	Multiple java.sql.ResultSet objects from SQL-invoked procedure calls.	36
4.4.16.1	Introduction to multiple result sets.	36
4.4.16.2	Resource management with multiple results.	36
4.4.17	JDBC/SQLJ ResultSet interoperability.	36
4.4.17.1	Introduction to interoperability.	36
4.4.17.2	Creating an SQLJ iterator from a java.sql.ResultSet object.	36
4.4.17.3	Obtaining a java.sql.ResultSet object from an SQLJ iterator object.	37
4.4.17.4	Obtaining a java.sql.ResultSet object from an untyped iterator object.	37
4.4.17.5	Iterator and java.sql.ResultSet object resource management.	37
4.4.18	Multi-threading considerations.	38
4.4.19	User-defined data types.	38
4.4.20	Batch updates.	39
4.4.20.1	Introduction to batch updates.	39
4.4.20.2	Batchable statements and batch compatibility.	39
4.4.20.3	Statement batching API.	39
4.4.20.4	Execution status and update counts.	40
4.4.20.5	Program semantics and exceptions.	40
4.4.20.6	Batch cancellation and disabling.	41
4.4.20.7	Specification of a batching limit.	41
4.4.21	SQLJ language elements.	42
4.4.21.1	Introduction to SQLJ language elements.	42
4.4.21.2	<cursor name>.	42
4.4.21.3	SQL-schema, SQL-data, and SQL-transaction statements.	43
4.4.21.4	<SQL dynamic statement>.	43
4.4.21.5	<SQL connection statement>.	43
4.4.21.6	<host variable definition>.	43
4.4.21.7	<embedded exception declaration>.	43
4.4.21.8	<SQL diagnostics statement>.	44
4.4.21.9	Cursor declaration.	44
4.4.21.10	Input parameters to SQL-statements.	44
4.4.21.11	Extracting column values from SQLJ iterators.	44
4.4.21.12	<open statement> and cursors.	44
4.4.22	SQLJ, JDBC, and SQLExceptions and SQLWarnings.	45
4.4.23	Profile generation and naming.	45
4.4.24	SQLJ application packaging.	46
4.4.25	Profile customizer interface.	46
4.4.26	Customization interface.	47
4.4.26.1	Introduction to customization interfaces.	47
4.4.26.2	Customization usage.	47
4.4.26.3	Customization registration.	48
5	Lexical elements.	50
5.1	<SQL terminal character>.	50
5.2	<token> and <separator>.	51
6	Scalar expressions.	52

6.1	<value specification> and <target specification>.....	52
7	Additional common rules.....	53
7.1	Invoking an SQL-invoked routine.....	53
8	Data manipulation.....	54
8.1	<fetch statement>.....	54
8.2	<select statement: single row>.....	57
8.3	<delete statement: positioned>.....	61
8.4	<update statement: positioned>.....	63
9	Control statements.....	65
9.1	<call statement>.....	65
9.2	<compound statement>.....	67
9.3	<assignment statement>.....	69
10	Transaction management.....	71
10.1	<set transaction statement>.....	71
10.2	<savepoint statement>.....	72
10.3	<release savepoint statement>.....	73
10.4	<commit statement>.....	74
10.5	<rollback statement>.....	75
11	Embedded SQL.....	76
11.1	<embedded SQL host program>.....	76
11.2	<embedded SQL Java program>.....	78
12	SQLJ reserved names.....	79
12.1	Introduction to reserved names.....	79
12.2	Naming runtime library components.....	79
12.3	Temporary variable names.....	79
12.4	Class and resource file names.....	79
12.4.1	Introduction.....	79
12.4.2	Generated classes.....	79
12.4.3	Resource files and profiles.....	80
13	Common subelements.....	81
13.1	<modifiers>.....	81
13.2	<java class name>.....	82
13.3	<java id>.....	83
13.4	<java datatype>.....	84
13.5	<java constant expression>.....	85
13.6	<embedded Java expression>.....	86
13.7	<implements clause>.....	89
13.8	<declaration with clause>.....	90
14	<SQLJ specific clause> and contents.....	93
14.1	<SQLJ specific clause>.....	93
14.2	<connection declaration clause>.....	94
14.3	Generated connection class.....	95
14.4	<iterator declaration clause>.....	100
14.5	<positioned iterator>.....	102
14.6	Generated positioned iterator class.....	103

14.7	<named iterator>.....	105
14.8	Generated named iterator class.....	106
14.9	<executable clause>.....	108
14.10	<context clause>.....	114
14.11	<statement clause>.....	115
14.12	<assignment clause>.....	116
14.13	<query clause>.....	118
14.14	<function clause>.....	122
14.15	<iterator conversion clause>.....	124
15	Package sqlj.runtime.....	127
15.1	Overview.....	127
15.2	SQLJ runtime interfaces.....	127
15.2.1	sqlj.runtime.ConnectionContext.....	127
15.2.1.1	Interface overview.....	127
15.2.1.2	Variables.....	128
15.2.1.2.1	CLOSE_CONNECTION.....	128
15.2.1.2.2	KEEP_CONNECTION.....	128
15.2.1.3	Methods.....	128
15.2.1.3.1	close ()......	128
15.2.1.3.2	close (boolean).....	129
15.2.1.3.3	getConnectedProfile (Object).....	129
15.2.1.3.4	getConnection ()......	130
15.2.1.3.5	getExecutionContext ()......	130
15.2.1.3.6	getTypeMap ()......	130
15.2.1.3.7	isClosed ()......	131
15.2.2	sqlj.runtime.ForUpdate.....	131
15.2.2.1	Interface overview.....	131
15.2.2.2	Methods.....	131
15.2.2.2.1	getCursorName ()......	131
15.2.3	sqlj.runtime.NamedIterator.....	131
15.2.4	sqlj.runtime.PositionedIterator.....	132
15.2.4.1	Interface overview.....	132
15.2.4.2	Methods.....	132
15.2.4.2.1	endFetch ()......	132
15.2.5	sqlj.runtime.ResultSetIterator.....	133
15.2.5.1	Interface overview.....	133
15.2.5.2	Variables.....	133
15.2.5.2.1	ASENSITIVE.....	133
15.2.5.2.2	FETCH_FORWARD.....	133
15.2.5.2.3	FETCH_REVERSE.....	133
15.2.5.2.4	FETCH_UNKNOWN.....	133
15.2.5.2.5	INSENSITIVE.....	133
15.2.5.2.6	SENSITIVE.....	134
15.2.5.3	Methods.....	134
15.2.5.3.1	Note about methods.....	134
15.2.5.3.2	clearWarnings ()......	134
15.2.5.3.3	close ()......	134

15.2.5.3.4	getFetchSize ()	134
15.2.5.3.5	getResultSet ()	135
15.2.5.3.6	getRow ()	135
15.2.5.3.7	getSensitivity ()	135
15.2.5.3.8	getWarnings ()	136
15.2.5.3.9	isClosed ()	136
15.2.5.3.10	next ()	137
15.2.5.3.11	setFetchSize (int)	137
15.2.6	sqlj.runtime.Scrollable	138
15.2.6.1	Interface overview	138
15.2.6.2	Variables	138
15.2.6.3	Methods	138
15.2.6.3.1	absolute (int)	138
15.2.6.3.2	afterLast ()	138
15.2.6.3.3	beforeFirst ()	138
15.2.6.3.4	first ()	139
15.2.6.3.5	getFetchDirection ()	139
15.2.6.3.6	isAfterLast ()	139
15.2.6.3.7	isBeforeFirst ()	140
15.2.6.3.8	isFirst ()	140
15.2.6.3.9	isLast ()	140
15.2.6.3.10	last ()	140
15.2.6.3.11	previous ()	141
15.2.6.3.12	relative (int)	141
15.2.6.3.13	setFetchDirection (int)	141
15.3	SQLJ runtime classes	143
15.3.1	sqlj.runtime.AsciiStream	143
15.3.1.1	Class overview	143
15.3.1.2	Constructors	143
15.3.1.2.1	AsciiStream (InputStream)	143
15.3.1.2.2	AsciiStream (InputStream, int)	143
15.3.2	sqlj.runtime.BinaryStream	144
15.3.2.1	Class overview	144
15.3.2.2	Constructors	144
15.3.2.2.1	BinaryStream (InputStream)	144
15.3.2.2.2	BinaryStream (InputStream, int)	144
15.3.3	sqlj.runtime.DefaultRuntime	145
15.3.3.1	Class overview	145
15.3.3.2	Constructors	145
15.3.3.2.1	DefaultRuntime ()	145
15.3.3.3	Methods	145
15.3.3.3.1	getDefaultConnection ()	145
15.3.3.3.2	getLoaderForClass (Class)	145
15.3.4	sqlj.runtime.ExecutionContext	146
15.3.4.1	Class overview	146
15.3.4.2	Variables	147
15.3.4.2.1	ADD_BATCH_COUNT	147

15.3.4.2.2	AUTO_BATCH.	147
15.3.4.2.3	EXEC_BATCH_COUNT.	147
15.3.4.2.4	EXCEPTION_COUNT.	147
15.3.4.2.5	NEW_BATCH_COUNT.	147
15.3.4.2.6	QUERY_COUNT.	148
15.3.4.2.7	UNLIMITED_BATCH.	148
15.3.4.3	Constructors.	148
15.3.4.3.1	ExecutionContext ().	148
15.3.4.4	Methods.	148
15.3.4.4.1	cancel ().	148
15.3.4.4.2	execute ().	149
15.3.4.4.3	executeBatch ().	150
15.3.4.4.4	executeQuery ().	151
15.3.4.4.5	executeUpdate ().	151
15.3.4.4.6	getBatchLimit ().	152
15.3.4.4.7	getBatchUpdateCounts ().	152
15.3.4.4.8	getFetchDirection ().	153
15.3.4.4.9	getFetchSize ().	153
15.3.4.4.10	getMaxFieldSize ().	153
15.3.4.4.11	getMaxRows ().	154
15.3.4.4.12	getNextResultSet ().	154
15.3.4.4.13	getNextResultSet (int).	154
15.3.4.4.14	getQueryTimeout ().	155
15.3.4.4.15	getUpdateCount ().	156
15.3.4.4.16	getWarnings ().	156
15.3.4.4.17	isBatching ().	156
15.3.4.4.18	registerStatement (ConnectionContext, Object, int).	157
15.3.4.4.19	releaseStatement ().	158
15.3.4.4.20	setBatching (boolean).	158
15.3.4.4.21	setBatchLimit (int).	159
15.3.4.4.22	setFetchDirection (int).	159
15.3.4.4.23	setFetchSize (int).	160
15.3.4.4.24	setMaxFieldSize (int).	160
15.3.4.4.25	setMaxRows (int).	160
15.3.4.4.26	setQueryTimeout (int).	161
15.3.5	sqlj.runtime.RuntimeContext.	161
15.3.5.1	Class overview.	161
15.3.5.2	Variables.	161
15.3.5.2.1	DEFAULT_DATA_SOURCE.	161
15.3.5.2.2	DEFAULT_RUNTIME.	161
15.3.5.2.3	PROPERTY_KEY.	162
15.3.5.3	Constructors.	162
15.3.5.3.1	RuntimeContext ().	162
15.3.5.4	Methods.	162
15.3.5.4.1	getDefaultConnection ().	162
15.3.5.4.2	getLoaderForClass (Class).	162
15.3.5.4.3	getRuntime ().	163

15.3.6	sqlj.runtime.StreamWrapper.	163
15.3.6.1	Class overview.	163
15.3.6.2	Constructors.	164
15.3.6.2.1	StreamWrapper (InputStream).	164
15.3.6.2.2	StreamWrapper (InputStream, int).	164
15.3.6.3	Methods.	164
15.3.6.3.1	getInputStream ().	164
15.3.6.3.2	getLength ().	164
15.3.6.3.3	setLength (int).	165
15.3.7	sqlj.runtime.UnicodeStream.	165
15.3.7.1	Class overview.	165
15.3.7.2	Constructors.	165
15.3.7.2.1	UnicodeStream (InputStream).	165
15.3.7.2.2	UnicodeStream (InputStream, int).	166
15.3.8	sqlj.runtime.CharacterStream.	166
15.3.8.1	Class overview.	166
15.3.8.2	Constructors.	166
15.3.8.2.1	CharacterStream (Reader).	166
15.3.8.2.2	CharacterStream (Reader, int).	166
15.3.8.3	Methods.	167
15.3.8.3.1	getReader ().	167
15.3.8.3.2	getLength ().	167
15.3.8.3.3	setLength (int).	167
15.3.9	sqlj.runtime.SQLNullException.	167
15.3.9.1	Class overview.	167
15.3.9.2	Constructors.	168
15.3.9.2.1	SQLNullException ().	168
16	Package sqlj.runtime.profile.	169
16.1	Overview.	169
16.2	SQLJ sqlj.runtime.profile interfaces.	169
16.2.1	sqlj.runtime.profile.BatchContext.	169
16.2.1.1	Interface overview.	169
16.2.1.2	Methods.	169
16.2.1.2.1	clearBatch ().	169
16.2.1.2.2	executeBatch ().	169
16.2.1.2.3	setBatchLimit (int).	170
16.2.2	sqlj.runtime.profile.ConnectedProfile.	170
16.2.2.1	Interface overview.	170
16.2.2.2	Methods.	171
16.2.2.2.1	close ().	171
16.2.2.2.2	getConnection ().	171
16.2.2.2.3	getProfileData ().	172
16.2.2.2.4	getStatement (int, Map).	172
16.2.2.2.5	getStatement (int, BatchContext, Map).	173
16.2.3	sqlj.runtime.profile.Customization.	173
16.2.3.1	Interface overview.	173
16.2.3.2	Methods.	174

16.2.3.2.1	acceptsConnection (Connection).....	174
16.2.3.2.2	getProfile (Connection, Profile).....	174
16.2.4	sqlj.runtime.profile.Loader.....	175
16.2.4.1	Interface overview.....	175
16.2.4.2	Methods.....	175
16.2.4.2.1	getResourceAsStream (String).....	175
16.2.4.2.2	loadClass (String).....	175
16.2.5	sqlj.runtime.profile.RTResultSet.....	176
16.2.5.1	Interface overview.....	176
16.2.5.2	Methods.....	179
16.2.5.2.1	clearWarnings ()......	179
16.2.5.2.2	close ()......	179
16.2.5.2.3	findColumn (String).....	179
16.2.5.2.4	getArray (int).....	180
16.2.5.2.5	getAsciiStreamWrapper (int).....	180
16.2.5.2.6	getBigDecimal (int).....	181
16.2.5.2.7	getBinaryStreamWrapper (int).....	182
16.2.5.2.8	getBlob (int).....	183
16.2.5.2.9	getBooleanNotNull (int).....	183
16.2.5.2.10	getBooleanWrapper (int).....	184
16.2.5.2.11	getByteNotNull (int).....	185
16.2.5.2.12	getBytes (int).....	185
16.2.5.2.13	getByteWrapper (int).....	186
16.2.5.2.14	getCharacterStreamWrapper (int).....	187
16.2.5.2.15	getClob (int).....	188
16.2.5.2.16	getColumnCount ()......	188
16.2.5.2.17	getCursorName ()......	189
16.2.5.2.18	getDate (int).....	189
16.2.5.2.19	getDoubleNotNull (int).....	190
16.2.5.2.20	getDoubleWrapper (int).....	191
16.2.5.2.21	getFloatNotNull (int).....	191
16.2.5.2.22	getFloatWrapper (int).....	192
16.2.5.2.23	getIntNotNull (int).....	193
16.2.5.2.24	getIntWrapper (int).....	193
16.2.5.2.25	getJDBCResultSet ()......	194
16.2.5.2.26	getLongNotNull (int).....	194
16.2.5.2.27	getLongWrapper (int).....	195
16.2.5.2.28	getObject (int, Class).....	196
16.2.5.2.29	getRef (int).....	197
16.2.5.2.30	getShortNotNull (int).....	197
16.2.5.2.31	getShortWrapper (int).....	198
16.2.5.2.32	getString (int).....	199
16.2.5.2.33	getSQLXML(int).....	199
16.2.5.2.34	getTime (int).....	200
16.2.5.2.35	getTimestamp (int).....	201
16.2.5.2.36	getUnicodeStreamWrapper (int).....	201
16.2.5.2.37	getURL (int).....	202

16.2.5.2.38	getWarnings ()	203
16.2.5.2.39	isClosed ()	203
16.2.5.2.40	isValidRow ()	203
16.2.5.2.41	next ()	204
16.2.6	sqlj.runtime.profile.RTStatement	204
16.2.6.1	Interface overview	204
16.2.6.2	Methods	209
16.2.6.2.1	cancel ()	209
16.2.6.2.2	clearWarnings ()	209
16.2.6.2.3	execute ()	210
16.2.6.2.4	executeComplete ()	210
16.2.6.2.5	executeRTQuery ()	210
16.2.6.2.6	executeUpdate ()	211
16.2.6.2.7	getArray (int)	211
16.2.6.2.8	getBatchContext ()	212
16.2.6.2.9	getBigDecimal (int)	213
16.2.6.2.10	getBlob (int)	213
16.2.6.2.11	getBooleanNotNull (int)	214
16.2.6.2.12	getBooleanWrapper (int)	215
16.2.6.2.13	getByteNotNull (int)	215
16.2.6.2.14	getBytes (int)	216
16.2.6.2.15	getByteWrapper (int)	217
16.2.6.2.16	getClob (int)	217
16.2.6.2.17	getDate (int)	218
16.2.6.2.18	getDoubleNotNull (int)	219
16.2.6.2.19	getDoubleWrapper (int)	219
16.2.6.2.20	getFloatNotNull (int)	220
16.2.6.2.21	getFloatWrapper (int)	221
16.2.6.2.22	getIntNotNull (int)	221
16.2.6.2.23	getIntWrapper (int)	222
16.2.6.2.24	getJDBCCallableStatement ()	223
16.2.6.2.25	getJDBCPreparedStatement ()	223
16.2.6.2.26	getLongNotNull (int)	223
16.2.6.2.27	getLongWrapper (int)	224
16.2.6.2.28	getMaxFieldSize ()	225
16.2.6.2.29	getMaxRows ()	225
16.2.6.2.30	getMoreResults (int)	225
16.2.6.2.31	getObject (int, Class)	226
16.2.6.2.32	getQueryTimeout ()	227
16.2.6.2.33	getRef (int)	228
16.2.6.2.34	getResultSet ()	228
16.2.6.2.35	getShortNotNull (int)	229
16.2.6.2.36	getShortWrapper (int)	229
16.2.6.2.37	getString (int)	230
16.2.6.2.38	getSQLXML (int)	231
16.2.6.2.39	getTime (int)	231
16.2.6.2.40	getTimestamp (int)	232

16.2.6.2.41	getUpdateCount ()	233
16.2.6.2.42	getURL ()	233
16.2.6.2.43	getWarnings ()	234
16.2.6.2.44	isBatchable ()	234
16.2.6.2.45	isBatchCompatible ()	235
16.2.6.2.46	setArray (int, Array)	236
16.2.6.2.47	setAsciiStreamWrapper (int, AsciiStream)	236
16.2.6.2.48	setBigDecimal (int, BigDecimal)	237
16.2.6.2.49	setBinaryStreamWrapper (int, BinaryStream)	238
16.2.6.2.50	setBlob (int, Blob)	238
16.2.6.2.51	setBoolean (int, boolean)	239
16.2.6.2.52	setBooleanWrapper (int, Boolean)	239
16.2.6.2.53	setByte (int, byte)	240
16.2.6.2.54	setBytes (int, byte)	241
16.2.6.2.55	setByteWrapper (int, Byte)	241
16.2.6.2.56	setCharacterStreamWrapper (int, CharacterStream)	242
16.2.6.2.57	setClob (int, Clob)	242
16.2.6.2.58	setDate (int, Date)	243
16.2.6.2.59	setDouble (int, double)	244
16.2.6.2.60	setDoubleWrapper (int, Double)	244
16.2.6.2.61	setFloat (int, float)	245
16.2.6.2.62	setFloatWrapper (int, Float)	245
16.2.6.2.63	setInt (int, int)	246
16.2.6.2.64	setIntWrapper (int, Integer)	247
16.2.6.2.65	setLong (int, long)	247
16.2.6.2.66	setLongWrapper (int, Long)	248
16.2.6.2.67	setMaxFieldSize (int)	248
16.2.6.2.68	setMaxRows (int)	249
16.2.6.2.69	setObject ()	249
16.2.6.2.70	setQueryTimeout (int)	250
16.2.6.2.71	setRef (int, Ref)	250
16.2.6.2.72	setShort (int, short)	251
16.2.6.2.73	setShortWrapper (int, Short)	251
16.2.6.2.74	setString (int, String)	252
16.2.6.2.75	setSQLXML (int, SQLXML)	253
16.2.6.2.76	setTime (int, Time)	253
16.2.6.2.77	setTimestamp (int, Timestamp)	254
16.2.6.2.78	setUnicodeStreamWrapper (int, UnicodeStream)	254
16.2.6.2.79	setURL (int, URL)	255
16.2.7	sqlj.runtime.profile.SerializedProfile	256
16.2.7.1	Interface overview	256
16.2.7.2	Methods	256
16.2.7.2.1	getProfileAsStream ()	256
16.3	SQLJ sqlj.runtime.profile classes	257
16.3.1	sqlj.runtime.profile.DefaultLoader	257
16.3.1.1	Class overview	257
16.3.1.2	Constructors	257

16.3.1.2.1	DefaultLoader (ClassLoader).....	257
16.3.1.3	Methods.....	257
16.3.1.3.1	getResourceAsStream (String).....	257
16.3.1.3.2	loadClass (String).....	258
16.3.2	sqlj.runtime.profile.EntryInfo.....	258
16.3.2.1	Class overview.....	258
16.3.2.2	Variables.....	258
16.3.2.2.1	BLOCK.....	258
16.3.2.2.2	CALL.....	259
16.3.2.2.3	CALLABLE_STATEMENT.....	259
16.3.2.2.4	COMMIT.....	259
16.3.2.2.5	EXECUTE.....	259
16.3.2.2.6	EXECUTE_QUERY.....	260
16.3.2.2.7	EXECUTE_UPDATE.....	260
16.3.2.2.8	ITERATOR_CONVERSION.....	260
16.3.2.2.9	NAMED_RESULT.....	260
16.3.2.2.10	NO_RESULT.....	261
16.3.2.2.11	OTHER.....	261
16.3.2.2.12	POSITIONED.....	261
16.3.2.2.13	POSITIONED_RESULT.....	261
16.3.2.2.14	PREPARED_STATEMENT.....	262
16.3.2.2.15	QUERY.....	262
16.3.2.2.16	QUERY_FOR_UPDATE.....	262
16.3.2.2.17	RELEASE_SAVEPOINT.....	262
16.3.2.2.18	ROLLBACK.....	263
16.3.2.2.19	SAVEPOINT.....	263
16.3.2.2.20	SET_TRANSACTION.....	263
16.3.2.2.21	SINGLE_ROW_QUERY.....	263
16.3.2.2.22	STATEMENT.....	264
16.3.2.2.23	UNTYPED_SELECT.....	264
16.3.2.2.24	VALUES.....	264
16.3.2.3	Constructors.....	264
16.3.2.3.1	EntryInfo ()......	264
16.3.2.4	Methods.....	264
16.3.2.4.1	executeTypeToString (int).....	264
16.3.2.4.2	getDescriptor ()......	265
16.3.2.4.3	getExecuteType ()......	265
16.3.2.4.4	getLineNumber ()......	266
16.3.2.4.5	getParamCount ()......	266
16.3.2.4.6	getParamInfo (int).....	266
16.3.2.4.7	getResultSetCount ()......	266
16.3.2.4.8	getResultSetInfo (int).....	267
16.3.2.4.9	getResultSetName ()......	267
16.3.2.4.10	getResultSetType ()......	268
16.3.2.4.11	getRole ()......	268
16.3.2.4.12	getSQLString ()......	269
16.3.2.4.13	getStatementType ()......	269

16.3.2.4.14	getTransactionDescriptor ()	270
16.3.2.4.15	isDefinedRole (int)	270
16.3.2.4.16	isValidDescriptor (Object, int)	270
16.3.2.4.17	isValidExecuteType (int)	271
16.3.2.4.18	isValidResultSetType (int)	271
16.3.2.4.19	isValidRole (int)	271
16.3.2.4.20	isValidStatementType (int)	272
16.3.2.4.21	resultSetTypeToString (int)	272
16.3.2.4.22	roleToString (int)	273
16.3.2.4.23	statementTypeToString (int)	273
16.3.2.4.24	validateObject ()	274
16.3.3	sqlj.runtime.profile.Profile	274
16.3.3.1	Class overview	274
16.3.3.2	Constructors	274
16.3.3.2.1	Profile (Loader)	274
16.3.3.3	Methods	275
16.3.3.3.1	deregisterCustomization (Customization)	275
16.3.3.3.2	getConnectedProfile (Connection)	275
16.3.3.3.3	getContextName ()	276
16.3.3.3.4	getCustomizations ()	276
16.3.3.3.5	getJavaType (String)	276
16.3.3.3.6	getJavaType (TypeInfo)	277
16.3.3.3.7	getLoader ()	277
16.3.3.3.8	getProfileData ()	277
16.3.3.3.9	getProfileName ()	277
16.3.3.3.10	getTimestamp ()	278
16.3.3.3.11	instantiate (Loader, InputStream)	278
16.3.3.3.12	instantiate (Loader, String)	279
16.3.3.3.13	registerCustomization (Customization)	280
16.3.3.3.14	registerCustomization (Customization, Customization)	280
16.3.3.3.15	replaceCustomization (Customization, Customization)	281
16.3.4	sqlj.runtime.profile.ProfileData	281
16.3.4.1	Class overview	281
16.3.4.2	Constructors	281
16.3.4.2.1	ProfileData ()	281
16.3.4.3	Methods	282
16.3.4.3.1	getEntryInfo (int)	282
16.3.4.3.2	getProfile ()	282
16.3.4.3.3	getSourceFile ()	282
16.3.4.3.4	size ()	282
16.3.5	sqlj.runtime.profile.SetTransactionDescriptor	283
16.3.5.1	Class overview	283
16.3.5.2	Variables	283
16.3.5.2.1	READ_NONE	283
16.3.5.2.2	READ_ONLY	283
16.3.5.2.3	READ_WRITE	283
16.3.5.3	Constructors	284

16.3.5.3.1	SetTransactionDescriptor (int, int).....	284
16.3.5.4	Methods.....	284
16.3.5.4.1	getAccessMode ()......	284
16.3.5.4.2	getIsolationLevel ()......	284
16.3.6	sqlj.runtime.profile.TypeInfo.....	285
16.3.6.1	Class overview.....	285
16.3.6.2	Variables.....	285
16.3.6.2.1	IN.....	285
16.3.6.2.2	INOUT.....	285
16.3.6.2.3	OUT.....	285
16.3.6.3	Constructors.....	286
16.3.6.3.1	TypeInfo ()......	286
16.3.6.4	Methods.....	286
16.3.6.4.1	getJavaTypeName ()......	286
16.3.6.4.2	getMarkerIndex ()......	286
16.3.6.4.3	getMode ()......	287
16.3.6.4.4	getName ()......	287
16.3.6.4.5	getSQLType ()......	287
16.3.6.4.6	getSQLTypeName ()......	288
16.3.6.4.7	isValidMode (int).....	288
16.3.6.4.8	isValidSQLType (int).....	289
16.3.6.4.9	modeToString (int).....	289
16.3.6.4.10	SQLTypeToString (int).....	290
16.3.6.4.11	validateObject ()......	290
17	sqlj.runtime.profile.util.ProfileCustomizer.....	291
17.1	Interface overview.....	291
17.2	Methods.....	293
17.2.1	acceptsConnection (Connection).....	293
17.2.2	customize (Profile, Connection, ErrorLog).....	293
18	Status codes.....	295
18.1	SQLSTATE.....	295
19	Conformance.....	296
19.1	Claims of conformance to SQL/OLB.....	296
19.2	Additional conformance requirements for SQL/OLB.....	296
19.3	Implied feature relationships of SQL/OLB.....	296
Annex A	(informative) SQL conformance summary.....	297
Annex B	(informative) Implementation-defined elements.....	300
Annex C	(informative) Implementation-dependent elements.....	304
Annex D	(informative) SQL optional feature taxonomy.....	308
Annex E	(informative) Deprecated features.....	310
Annex F	(informative) Incompatibilities with ISO/IEC 9075:2016.....	311
Annex G	(informative) Defect Reports not addressed in this edition of this document.....	312
	Bibliography.....	313

Index..... 314

Tables

Table	Page
1 Association of roles with SQLJ <executable clause>s.	13
2 SQLJ type properties.	16
3 SQLJ output assignability (data type group 1).	23
4 SQLJ output assignability (data type group 2).	25
5 SQLJ output assignability (data type group 3).	26
6 SQLJ output assignability (data type group 4).	28
7 SQLJ input assignability (data type group 1).	30
8 SQLJ input assignability (data type group 2).	31
9 SQLJ input assignability (data type group 3).	33
10 SQLJ input assignability (data type group 4).	34
11 Methods retained from java.sql.ResultSet.	177
12 Methods not retained from java.sql.ResultSet.	177
13 Additional methods unique to ResultSet.	179
14 Methods retained from java.sql.Statement.	205
15 Methods not retained from java.sql.Statement.	206
16 Methods retained from java.sql.PreparedStatement.	206
17 Methods not retained from java.sql.PreparedStatement.	207
18 Methods retained from java.sql.CallableStatement.	208
19 Methods not retained from java.sql.CallableStatement.	208
20 Additional methods unique to PreparedStatement.	209
21 Customize Result Interpretation.	292
22 SQLSTATE class and subclass codes.	295
23 Implied feature relationships of SQL/OLB.	296
A.1 Feature definitions outside of Conformance Rules.	297
D.1 Feature taxonomy for optional features.	308

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.

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 or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC have not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and <https://patents.iec.ch>. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

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

For an explanation of the voluntary nature of standards, 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 www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

This fifth edition cancels and replaces the fourth edition (ISO/IEC 9075-10:2016), which has been technically revised. It also incorporates the Technical Corrigendum ISO/IEC 9075-10:2016/Cor.1:2022.

The main changes are as follows:

- improve the presentation and accuracy of the summaries of implementation-defined and implementation-dependent aspects of this document;
- introduction of several digital artifacts;
- alignment with updated ISO house style and other guidelines for creating standards.

This fifth edition of ISO/IEC 9075-10 is designed to be used in conjunction with the following editions of other parts of the ISO/IEC 9075 series, all published in 2023:

- ISO/IEC 9075-1, sixth edition;
- ISO/IEC 9075-2, sixth edition;
- ISO/IEC 9075-3, sixth edition;

- ISO/IEC 9075-4, seventh edition;
- ISO/IEC 9075-9, fifth edition;
- ISO/IEC 9075-11, fifth edition;
- ISO/IEC 9075-13, fifth edition;
- ISO/IEC 9075-14, sixth edition;
- ISO/IEC 9075-15, second edition;
- ISO/IEC 9075-16, first edition.

A list of all parts in the ISO/IEC 9075 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

The organization of this document is as follows:

- 1) [Clause 1, “Scope”](#), specifies the scope of this document.
- 2) [Clause 2, “Normative references”](#), identifies additional standards and publicly-available specifications that, through reference in this document, constitute provisions of this document.
- 3) [Clause 3, “Terms and definitions”](#), defines the terms and definitions used in this document.
- 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 rules”](#), specifies the rules for assignments that retrieve data from or store data into SQL-data, and formation rules for set operations.
- 8) [Clause 8, “Data manipulation”](#), defines the data manipulation statements.
- 9) [Clause 9, “Control statements”](#), defines the SQL-control statements.
- 10) [Clause 10, “Transaction management”](#), defines the SQL-transaction management statements.
- 11) [Clause 11, “Embedded SQL”](#), defines the host language embeddings.
- 12) [Clause 12, “SQLJ reserved names”](#), defines the reserved names for SQLJ.
- 13) [Clause 13, “Common subelements”](#), defines the commonly used subelements for SQLJ.
- 14) [Clause 14, “<SQLJ specific clause> and contents”](#), defines the syntax and rules for SQLJ constructs.
- 15) [Clause 15, “Package sqlj.runtime”](#), specifies the SQLJ runtime package.
- 16) [Clause 16, “Package sqlj.runtime.profile”](#), specifies the SQLJ runtime profile package.
- 17) [Clause 17, “sqlj.runtime.profile.util.ProfileCustomizer”](#), specifies the SQLJ profile customizer class.
- 18) [Clause 18, “Status codes”](#), defines SQLSTATE values related to Object Language Bindings.
- 19) [Clause 19, “Conformance”](#), defines the criteria for conformance to this document.
- 20) [Annex A, “SQL conformance summary”](#), is an informative Annex. It summarizes the conformance requirements of the SQL language.
- 21) [Annex B, “Implementation-defined elements”](#), is an informative Annex. It lists those features for which the body of this document states that the syntax, the meaning, the returned results, the effect on SQL-data and/or schemas, or other aspect is partly or wholly implementation-defined.
- 22) [Annex C, “Implementation-dependent elements”](#), is an informative Annex. It lists those features for which the body of this document states that the syntax, the meaning, the returned results, the effect on SQL-data and/or schemas, or other aspect is partly or wholly implementation-dependent.
- 23) [Annex D, “SQL optional feature taxonomy”](#), is an informative Annex. It identifies the optional features of the SQL language specified in this document by an identifier and a short descriptive name. This taxonomy is used to specify conformance.
- 24) [Annex E, “Deprecated features”](#), is an informative Annex. It lists features that the responsible Technical Committee intends not to include in a future edition of this document.

- 25) [Annex F, “Incompatibilities with ISO/IEC 9075:2016”](#), is an informative Annex. It lists incompatibilities with the previous edition of this document.
- 26) [Annex G, “Defect Reports not addressed in this edition of this document”](#), is an informative Annex. It describes the Defect Reports that were known at the time of publication of this document. Each of these problems is a problem carried forward from the previous edition of the ISO/IEC 9075 series. No new problems have been created in the drafting of this edition of this document.

In the text of this document, Clauses and Annexes begin new odd-numbered pages. Any resulting blank space is not significant.

All Clauses of this document are normative.

Information technology — Database language 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 document defines similar features of Database language SQL that support embedding of SQL-statements into programs written in the Java¹ programming language. The embedding of SQL into Java is commonly known as “SQLJ”. This document 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 document, the terms “SQLJ” and “SQL/OLB” are used synonymously.

¹ Java™ is a registered trademark of Oracle Corporation and/or its affiliates.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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.

ISO/IEC 9075-1, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*

ISO/IEC 9075-2, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*

ISO/IEC 9075-4, *Information technology — Database languages — SQL — Part 4: Persistent Stored Modules (SQL/PSM)*

The Unicode Consortium. *The Unicode Standard* (Information about the latest version of the Unicode standard can be found by using the “Latest Version” link on the “Enumerated Versions of The Unicode Standard” page.) [online]. Mountain View, California, USA: The Unicode Consortium, Available at <https://www.unicode.org/versions/enumeratedversions.html>

Java Community Process. *The Java™ Language Specification* [online]. Java SE 13 Edition. Redwood Shores, California, USA: Oracle, Available at <https://docs.oracle.com/javase/specs/jls/se13/jls13.pdf>

Java Community Process. *JDBC™ 4.3 Specification* [online]. Edition 4.3. Redwood Shores, California, USA: Oracle, Available at https://download.oracle.com/otn-pub/jcp/jdbc-4_3-mrel3-eval-spec/jdbc4.3-fr-spec.pdf

Java Community Process. *Java Naming and Directory Interface™* [online]. Redwood Shores, California, USA: Oracle, Available at <https://docs.oracle.com/javase/8/docs/technotes/guides/jndi/index.html>

Java Community Process. *The JavaBeans™ 3.0 Specification* [online]. Edition 3.0. Redwood Shores, California, USA: Oracle, Available at https://download.oracle.com/otndocs/jcp/ejb-3_0-pr-spec-oth-JSpec/

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9075-1, ISO/IEC 9075-2, and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at <https://www.iso.org/obp>
- IEC Electropedia: available at <https://www.electropedia.org/>

3.1

accessor method

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

Note 1 to entry: An *accessor method* is either a *named accessor method* (3.10) or a *positioned accessor method* (3.14).

3.2

customization

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

Note 1 to entry: 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.

3.3

generated connection class

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

Note 1 to entry: 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.

3.4

generated iterator class

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

Note 1 to entry: A *generated iterator class* is either a *generated named iterator class* (3.5) or a *generated positioned iterator class* (3.6).

3.5

generated named iterator class

generated iterator class (3.4) with a signature that is produced as a side effect of the inclusion of an <iterator declaration clause> that contains an <iterator spec declaration> of <named iterator>, and it specifies named accessor methods

3.6

generated positioned iterator class

generated iterator class (3.4) with a signature that is produced as a side effect of the inclusion of an <iterator declaration clause> that contains an <iterator spec declaration> of <positioned iterator>, and it specifies positioned accessor methods

3.7

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

3.8

implementation-specific, adj.

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

3.9

installation

<of an SQLJ application> implementation-defined, and possibly empty, phase that includes anything other than *SQLJ translation* (3.19) and *customization* (3.2) needed prior to the SQLJ application being able to execute against its target SQL-implementation

3.10

named accessor method

accessor method (3.1) that is declared as the result of an <iterator declaration clause> containing an <iterator spec declaration> of <named iterator>

Note 1 to entry: A named accessor method derives both its name and its result datatype from its defining <named iterator> clause.

3.11

Java primitive datatype

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

Note 1 to entry: For interoperability with JDBC, the Java primitive datatype char is intentionally omitted from this list.

3.12

l-valued expression

Java expression that is allowed to appear as the *LeftHandSide* of a Java *assignment*, as defined in *Java*

Note 1 to entry: 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.

3.13

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 1 to entry: The serialized objects can then be accessed for additional processing by a customizer or by the SQLJ runtime system.

3.14

positioned accessor method

accessor method (3.1) that is declared as the result of an <iterator declaration clause> containing an <iterator spec declaration> of <named iterator>

Note 1 to entry: A *named accessor method* (3.10) derives both its name and its result datatype from its defining <named iterator> clause.

3.15

profile file

file containing one or more profiles generated as a result of an *SQLJ translation* (3.19)

3.16

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 1 to entry: 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.

3.17

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

3.18

SQLJ file

file with the suffix `'.sqlj'` (or file type `'sqlj'`) containing Java source language or SQLJ constructs, that is input to *SQLJ translation* (3.19)

3.19

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

3.20

SQLJ translation unit

collection of user-defined classes and property files defined or referenced by a given *SQLJ file* (3.18), and the translator-generated classes and profile files that result after that *SQLJ file* (3.18) is processed by an SQLJ translator, and its resulting Java source file is compiled

4 Concepts

This Clause modifies Clause 4, “Concepts”, in ISO/IEC 9075-2.

4.1 Notations and conventions

This Subclause modifies Subclause 4.1, “Notations and conventions”, in ISO/IEC 9075-2.

4.1.1 Notations

This Subclause modifies Subclause 4.1.1, “Notations”, in ISO/IEC 9075-2.

The notations used in this document are defined in ISO/IEC 9075-1.

The syntax defined in this document is available from the ISO website as a “digital artifact”. See <https://standards.iso.org/iso-iec/9075/-10/ed-5/en/> to download digital artifacts for this document. To download the syntax defined in a plain-text format, select the file named [ISO_IEC_9075-10\(E\)_OLB.bnf.txt](#). To download the syntax defined in an XML format, select the file named [ISO_IEC_9075-10\(E\)_OLB.bnf.xml](#).

4.1.2 First term usage

In addition, bolding is used when a term is first introduced in this document.

4.1.3 Java-related conventions

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*.

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.

The following denotations also hold throughout this document.

- 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 an <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.

4.1.4 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 13, “Common subelements”, and Clause 14, “<SQLJ specific clause> and contents”, are specified in terms of the following attributes.

- **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 4.4.5.2, “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>.

4.2 Character strings

This Subclause modifies *Subclause 4.3, “Character strings”*, in ISO/IEC 9075-2.

4.2.1 Unicode support

Java relies on the Unicode character set (also known as [ISO/IEC 10646](#), *Universal Multi-Octet Coded Character Set (UCS)*; see [The Unicode® Standard](#)) for String data and for identifiers. That allows Java to represent most character data in a uniform way. [ISO/IEC 9075-2](#) 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 ([IA018](#)). 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 [ISO/IEC 9075-2](#) 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.

4.3 Embedded syntax

This Subclause modifies *Subclause 4.31, “Embedded syntax”*, in ISO/IEC 9075-2.

Augment the [1st paragraph](#) by adding “except <embedded SQL Java program>s” as a restriction for <embedded SQL host program>s in the last sentence of the lead text of the Rule.

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.

Insert after the [1st paragraph](#): An SQL-implementation may reserve portions of the name space for the names of procedures, subroutines, program variables, branch labels, <SQL-client module definition>s, or <externally-invoked procedure>s for <embedded SQL host program>s other than <embedded SQL Java program>s; if a portion of the name space is so reserved, the portion reserved is implementation-defined ([IA051](#)).

4.4 Introduction to SQLJ

4.4.1 Overview

[ISO/IEC 9075-2](#) 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 document 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 language 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.

4.4.2 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

4.4.3 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 valid 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 upper-case, and write Java tokens in lower-case or mixed case. Throughout this document, we use the lower-case **null** to represent the Java “null” value, and the upper-case **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.

4.4.4 Binary portability

4.4.4.1 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 through 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 15, “Package sqlj.runtime”](#), [Clause 16, “Package sqlj.runtime.profile”](#), and [Clause 17, “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 ([UA014](#)) SQL execution is accom-

plished by implementing runtime library components. The calls made to these components by each <SQLJ specific clause> are defined in order to allow a standardized SQL/OLB implementation.

Given the portability framework, SQL/OLB 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. Instead, the above list specifies the preconditions that shall be satisfied to enable reliable addition of custom SQL execution components to an existing SQLJ application.

4.4.4.2 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.

- SQLJ standard runtime classes.
- For each translator **T** used to translate units in the application:
 - runtime classes used by translator-generated code (if any);
 - for each unit translated by **T**:
 - user-defined classes (which might contain executable SQLJ clauses),
 - user-defined property files containing maps for user-defined data types (if any),
 - classes generated by **T** as a result of declarative SQLJ clauses,
 - profiles generated by **T**,
 - auxiliary helper classes generated by the **T** (if any).
- 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-implement-ation 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.

4.4.5 Profile overview

4.4.5.1 Introduction to profiles

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 translation 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 SQL/OLB implementation

that supports an infrastructure that emulates the JDBC call interfaces and semantics will automatically be able to support SQLJ.

In addition, many SQL/OLB 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 SQL/OLB 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 (UA015) *customization objects*. A customization object is an implementation-dependent (UA015) 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 (UA017) “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.

4.4.5.2 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 here.

— 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 whitespace, 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	<i>all other operations</i>
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 1 — 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>, **HE_i**, the *i*-th TypeInfo object, describes **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 $1 \text{ (one)} \leq i \leq k$. The TypeInfo object returned has the mode of the <parameter mode> of **HE_i**. It has Java type name corresponding to the name of the type of **HE_i**. If **HE_i** is a <simple variable>, then the TypeInfo object returned has the same name as that of the <simple variable>. Otherwise, if **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 (negative one).

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.

4.4.5.3 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 [Java](#); 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 2 — 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 an <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 (negative one) if this TypeInfo object describes a result set column.

4.4.5.4 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	getBooleanNotNull	setBoolean
byte	TINYINT	getByteNotNull	setByte
short	SMALLINT	getShortNotNull	setShort
int	INTEGER	getIntNotNull	setInt
long	BIGINT	getLongNotNull	setLong
float	REAL	getFloatNotNull	setFloat
double	DOUBLE	getDoubleNotNull	setDouble
java.lang.Boolean	BOOLEAN	getBooleanWrapper	setBooleanWrapper
java.lang.Byte	TINYINT	getByteWrapper	setByteWrapper
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 type name	java.sql.Types value	getter method	setter method
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	setCharacterStream-Wrapper
sqlj.runtime.Uni-codeStream ²	OTHER	getUnicodeStream ³	setUnicodeStream
java.sql.Ref	REF	getRef	setRef
java.sql.SQLXML	SQLXML	getSQLXML	setSQLXML
<i>any other class</i> ⁵	STRUCT, DIS-TINCT, JAVA_OBJECT, OTHER	getObject	setObject

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 4.4.5.3, “TypeInfo overview”](#).

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

4.4.6 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.

4.4.7 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.

4.4.8 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 RFC 2368, and RFC 3986 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 3 — The connection context object implicitly specifies an SQL-implementation, and the default catalog and schema, as per a `java.sql.Connection` object.

4.4.9 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 (ID024). 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 multi-threaded 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 (ID023). JNDI is the Java Naming Directory Interface, defined in JNDI.

4.4.10 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 (IW160) 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.

4.4.11 Using multiple SQLJ contexts and connections

SQLJ supports concurrent connections to multiple SQL-servers. SQLJ models an 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 a `#sql` statement allows type checking across multiple SQL-servers at translate time.

4.4.12 Dynamic SQL and JDBC/SQLJ Connection interoperability

4.4.12.1 Introduction to 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.

4.4.12.2 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.

4.4.12.3 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](#).

4.4.12.4 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.

4.4.12.5 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 (UA016).

4.4.13 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 4 — 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 14.9, “<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.

Runtime support of the `sqlj.runtime.ExecutionContext` class methods **setMaxRows**, **setMaxFieldSize**, and **setQueryTimeout** if invoked to set an `ExecutionContext` object’s corresponding underlying values to anything other than their default values is implementation-defined (IA052).

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.

4.4.14 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.

4.4.15 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 (data type group 1)”, Table 4, “SQLJ output assignability (data type group 2)”, Table 5, “SQLJ output assignability (data type group 3)”, or Table 6, “SQLJ output assignability (data type group 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 (data type group 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 (data type group 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 (data type group 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 (data type group 1)

	java.sql.Types constants							
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

	java.sql.Types constants							
Java Data Types and Classes	TI	SI	IN	BI	RL	FL	DB	DC
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								
<i>any other class/interface</i>								
TI: corresponds to TINYINT SI: corresponds to SMALLINT IN: corresponds to INTEGER BI: corresponds to BIGINT RL: corresponds to REAL FL: corresponds to FLOAT DB: corresponds to DOUBLE DC corresponds to DECIMAL								

Table 4 — SQLJ output assignability (data type group 2)

	java.sql.Types constants							
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.Timestamp			x	x	x			
sqlj.runtime.ASCIIStream			x	x	x		x	x

	java.sql.Types constants							
Java Data Types and Classes	NU	BO	CH	VC	LC	CL	BI	VB
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								
<i>any other class/interface</i>								
NU: corresponds to NUMERIC BO: corresponds to BOOLEAN CH: corresponds to CHAR VC: corresponds to VARCHAR LC: corresponds to LONGVARCHAR CL: corresponds to CLOB BI: corresponds to BINARY VB: corresponds to VARBINARY								

Table 5 — SQLJ output assignability (data type group 3)

	java.sql.Types constants							
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