

Australian Standard™

**Information technology—Database
languages—SQL**

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



This Australian Standard was prepared by Committee IT-027, Data Management and Interchange. It was approved on behalf of the Council of Standards Australia on 16 May 2005. This Standard was published on 16 June 2005.

The following are represented on Committee IT-027:

Association of Superannuation Funds of Australia
Australia Post
Australia Bankers Association
Australian Bureau of Statistics
Australian Computer Society
Australian Customs Service
Australian Electoral Commission
Australian Electric and Electronic Manufacturers Association
Australian Industry Group
Australian Information Industry Association
Australian Institute of Health and Welfare
Australian Taxation Office
Centrelink
Department of Defence
Department of Immigration, Multicultural and Indigenous Affairs
Health Insurance Commission
Data Management Association Australia

Keeping Standards up-to-date

Standards are living documents which reflect progress in science, technology and systems. To maintain their currency, all Standards are periodically reviewed, and new editions are published. Between editions, amendments may be issued. Standards may also be withdrawn. It is important that readers assure themselves they are using a current Standard, which should include any amendments which may have been published since the Standard was purchased.

Detailed information about Standards can be found by visiting the Standards Web Shop at www.standards.com.au and looking up the relevant Standard in the on-line catalogue.

Alternatively, the printed Catalogue provides information current at 1 January each year, and the monthly magazine, *The Global Standard*, has a full listing of revisions and amendments published each month.

Australian Standards™ and other products and services developed by Standards Australia are published and distributed under contract by SAI Global, which operates the Standards Web Shop.

We also welcome suggestions for improvement in our Standards, and especially encourage readers to notify us immediately of any apparent inaccuracies or ambiguities. Contact us via email at mail@standards.org.au, or write to the Chief Executive, Standards Australia, GPO Box 5420, Sydney, NSW 2001.

This Standard was issued in draft form for comment as DR 05148.

Australian Standard™

Information technology—Database languages—SQL

Part 10: Object Language Bindings (SQL/OLB)

First published as AS ISO/IEC 9075.10—2005.

COPYRIGHT

© Standards Australia

All rights are reserved. No part of this work may be reproduced or copied in any form or by any means, electronic or mechanical, including photocopying, without the written permission of the publisher.

Published by Standards Australia GPO Box 5420, Sydney, NSW 2001, Australia

ISBN 0 7337 6769 9

PREFACE

This Standard was prepared by the Standards Australia Committee IT-027, Data Management and Interchange.

This Standard is identical with, and has been reproduced from ISO/IEC 9075-10:2003, *Information technology—Database languages—SQL—Part 10: Object Language Bindings (SQL/OLB)*.

The objective of this Standard is to provide database designers, administrators and developers extension definitions of database language SQL to support embedding of SQL statements into the programs written in the Java(tm) programming language.

This Standard is Part 10 of AS 9075, *Information technology—Database languages—SQL*, which is published in parts as follows:

- Part 1: Framework (SQL/Framework)
- Part 2: Foundation (SQL/Foundation)
- Part 3: Call-Level Interface (SQL/CLI)
- Part 4: Persistent Stored Modules (SQL/PSM)
- Part 5: Host Language Bindings (SQL/Bindings)
- Part 9: Management of External Data (SQL/MED)
- Part 10: Object Language Bindings (SQL/OLB) (this Standard)
- Part 11: Information and Definition Schemas (SQL/Schemata)
- Part 13: SQL Routines and Types Using the Java™ Programming Language (SQL/JRT)
- Part 14: XML-Related Specifications (SQL/XML)

As this Standard is reproduced from an international standard, the following applies:

- (a) Its number appears on the cover and title page while the international standard number appears only on the cover.
- (b) In the source text ‘this part of ISO/IEC 9075’ should read ‘this Australian Standard’.
- (c) A full point substitutes for a comma when referring to a decimal marker.

References to International Standards should be replaced by references to Australian or Australian/New Zealand Standards, as follows:

<i>Reference to International Standard</i>		<i>Australian Standard</i>	
ISO/IEC		AS	
9075	Information technology—Database languages—SQL	9075	Information technology—Database languages—SQL
9075-1	Part 1: Framework (SQL/Framework)	9075.1	Part 1: Framework (SQL/Framework)
9075-2	Part 2: Foundation (SQL/Foundation)	9075.2	Part 2: Foundation (SQL/Foundation)
9075-4	Part 4: Persistent Stored Modules (SQL/PSM)	9075.4	Part 4: Persistent Stored Modules (SQL/PSM)
9075-9	Part 9: Management of External Data (SQL/MED)	9075.9	Part 9: Management of External Data (SQL/MED)
9075-11	Part 11: Information and Definition Schemas (SQL/Schemas)	9075.11	Part 11: Information and Definition Schemas (SQL/Schemas)

Only referenced documents that have been adopted as Australian or Australian/New Zealand Standards have been listed.

CONTENTS

Page

1	Scope.....	1
2	Normative references.....	3
2.1	JTC 1 standards.....	3
2.2	Other international standards.....	3
3	Definitions, notations, and conventions.....	5
3.1	Definitions.....	5
3.1.1	Definitions provided in Part 10.....	5
3.2	Notation.....	6
3.3	Conventions.....	7
3.3.1	Use of terms.....	7
3.3.1.1	Other terms.....	7
3.3.2	Specification of translator-generated classes.....	8
4	Concepts.....	11
4.1	Introduction.....	11
4.2	Embedded syntax.....	11
4.3	SQL constructs.....	12
4.4	Character strings.....	12
4.4.1	Unicode support.....	12
4.4.2	Character sets.....	13
4.5	Host variables.....	13
4.6	Host expressions.....	13
4.7	SQLJ clauses.....	14
4.8	Database connection context.....	14
4.9	Default connection context.....	15
4.10	Schema checking using exemplar schemas.....	15
4.11	Using multiple SQLJ contexts and connections.....	16
4.12	Dynamic SQL and JDBC/SQLJ Connection interoperability.....	16
4.12.1	Creating an SQLJ ConnectionContext from a JDBC connection.....	16
4.12.2	Obtaining a JDBC connection from an SQLJ ConnectionContext.....	17
4.12.3	Connection sharing.....	17
4.12.4	Connection resource management.....	17
4.13	SQL execution control and status.....	18
4.14	Iterators.....	19
4.15	Input and output assignability.....	20

4.16	Calls to stored procedures and functions.	31
4.17	Multiple JDBC ResultSet objects from stored procedure calls.	31
4.17.1	Resource management with multiple results.	32
4.18	JDBC/SQLJ ResultSet interoperability.	32
4.18.1	Creating an SQLJ iterator from a JDBC ResultSet object.	32
4.18.2	Obtaining a JDBC ResultSet object from an SQLJ iterator object.	33
4.18.3	Obtaining a JDBC ResultSet object from an untyped iterator object.	33
4.18.4	Iterator and JDBC ResultSet resource management.	33
4.19	Multi-threading considerations.	34
4.20	User-defined data types.	34
4.21	Batch updates.	35
4.21.1	Batchable statements and batch compatibility.	35
4.21.2	Statement batching API.	36
4.21.3	Execution status and update counts.	36
4.21.4	Program semantics and exceptions.	37
4.21.5	Batch cancellation and disabling.	38
4.21.6	Specification of a batching limit.	38
4.22	SQLJ language elements.	39
4.22.1	<cursor name>.	39
4.22.2	SQL schema, data, and transaction statements.	40
4.22.3	<SQL dynamic statement>.	40
4.22.4	<SQL connection statement>.	40
4.22.5	<host variable definition>.	40
4.22.6	<embedded exception declaration>.	41
4.22.7	<SQL diagnostics statement>.	41
4.22.8	Cursor declaration.	41
4.22.9	Input parameters to SQL statements.	42
4.22.10	Extracting column values from SQLJ iterators.	42
4.22.11	<open statement> and cursors.	42
5	Lexical elements.	43
5.1	<SQL terminal character>.	43
5.2	<token> and <separator>.	44
6	Scalar expressions.	45
6.1	<value specification> and <target specification>.	45
7	Additional common elements.	47
7.1	<routine invocation>.	47
8	Embedded SQL.	49
8.1	<embedded SQL host program>.	49
8.2	<embedded SQL Java program>.	52
9	Binary portability.	53
9.1	Components of binary portable applications.	53

9.2	Naming runtime components.	54
9.3	Binary portability requirements.	54
9.4	Profile overview.	55
9.5	Profile generation and naming.	56
9.5.1	Example.	57
9.6	SQLJ application packaging.	57
9.6.1	Example.	58
9.7	Profile customization overview.	58
9.7.1	Profile customization process.	59
9.7.2	Profile customization utilities.	59
9.7.3	Profile customizer interface.	60
9.8	Customization interface.	60
9.8.1	Customization usage.	61
9.8.2	Customization registration.	62
9.9	EntryInfo overview.	63
9.10	TypeInfo overview.	65
9.11	SQLJ datatype properties.	67
10	SQLJ grammar constructs.	71
10.1	SQLJ reserved names.	71
10.1.1	Temporary variable names.	71
10.1.2	Class and resource file names.	71
10.1.2.1	Internal classes.	72
10.1.2.2	Resource files and profiles.	72
10.2	Common subelements.	73
10.2.1	<modifiers>.	73
10.2.2	<java class name>.	73
10.2.3	<java id>.	74
10.2.4	<java datatype>.	75
10.2.5	<java constant expression>.	75
10.2.6	<embedded Java expression>.	76
10.2.7	<implements clause>.	78
10.2.8	<declaration with clause>.	79
11	<SQLJ specific clause> and contents.	83
11.1	<SQLJ specific clause>.	83
11.2	<connection declaration clause>.	84
11.3	Generated connection class.	85
11.4	<iterator declaration clause>.	90
11.5	<positioned iterator>.	92
11.6	Generated positioned iterator class.	93
11.7	<named iterator>.	96
11.8	Generated named iterator class.	97
11.9	<executable clause>.	99

11.10	<context clause>.....	106
11.11	<statement clause>.....	108
11.12	<delete statement: positioned>.....	110
11.13	<update statement: positioned>.....	112
11.14	<select statement: single row>.....	114
11.15	<fetch statement>.....	118
11.16	<assignment statement>.....	122
11.17	<savepoint statement>.....	124
11.18	<release savepoint statement>.....	125
11.19	<commit statement>.....	126
11.20	<rollback statement>.....	127
11.21	<set transaction statement>.....	128
11.22	<call statement>.....	129
11.23	<assignment clause>.....	131
11.24	<query clause>.....	133
11.25	<function clause>.....	139
11.26	<iterator conversion clause>.....	142
11.27	<compound statement>.....	145
12	Package sqlj.runtime.....	147
12.1	SQLJ runtime interfaces.....	147
12.1.1	sqlj.runtime.ConnectionContext.....	147
12.1.1.1	Variables.....	148
12.1.1.1.1	CLOSE_CONNECTION.....	148
12.1.1.1.2	KEEP_CONNECTION.....	149
12.1.1.2	Methods.....	149
12.1.1.2.1	close ()......	149
12.1.1.2.2	close (boolean).....	149
12.1.1.2.3	getConnectedProfile (Object).....	150
12.1.1.2.4	getConnection ()......	151
12.1.1.2.5	getExecutionContext ()......	151
12.1.1.2.6	getTypeMap ()......	151
12.1.1.2.7	isClosed ()......	152
12.1.2	sqlj.runtime.ForUpdate.....	152
12.1.2.1	Methods.....	152
12.1.2.1.1	getCursorName ()......	152
12.1.3	sqlj.runtime.NamedIterator.....	153
12.1.4	sqlj.runtime.PositionedIterator.....	153
12.1.4.1	Methods.....	154
12.1.4.1.1	endFetch ()......	154
12.1.5	sqlj.runtime.ResultSetIterator.....	154
12.1.5.1	Variables.....	154
12.1.5.1.1	ASENSITIVE.....	154

12.1.5.1.2	FETCH_FORWARD.....	155
12.1.5.1.3	FETCH_REVERSE.....	155
12.1.5.1.4	FETCH_UNKNOWN.....	155
12.1.5.1.5	INSENSITIVE.....	155
12.1.5.1.6	SENSITIVE.....	155
12.1.5.2	Methods.....	156
12.1.5.2.1	clearWarnings ()......	156
12.1.5.2.2	close ()......	156
12.1.5.2.3	getFetchSize ()......	156
12.1.5.2.4	getResultSet ()......	157
12.1.5.2.5	getRow ()......	157
12.1.5.2.6	getSensitivity ()......	158
12.1.5.2.7	getWarnings ()......	158
12.1.5.2.8	isClosed ()......	159
12.1.5.2.9	next ()......	159
12.1.5.2.10	setFetchSize (int).....	160
12.1.6	sqlj.runtime.Scrollable.....	161
12.1.6.1	Variables.....	161
12.1.6.2	Methods.....	161
12.1.6.2.1	absolute (int).....	161
12.1.6.2.2	afterLast ()......	162
12.1.6.2.3	beforeFirst ()......	162
12.1.6.2.4	first ()......	162
12.1.6.2.5	getFetchDirection ()......	162
12.1.6.2.6	isAfterLast ()......	163
12.1.6.2.7	isBeforeFirst ()......	163
12.1.6.2.8	isFirst ()......	164
12.1.6.2.9	isLast ()......	164
12.1.6.2.10	last ()......	164
12.1.6.2.11	previous ()......	165
12.1.6.2.12	relative (int).....	165
12.1.6.2.13	setFetchDirection (int).....	166
12.2	SQLJ Runtime Classes.....	167
12.2.1	sqlj.runtime.AsciiStream.....	167
12.2.1.1	Constructors.....	167
12.2.1.1.1	AsciiStream (InputStream).....	167
12.2.1.1.2	AsciiStream (InputStream, int).....	168
12.2.2	sqlj.runtime.BinaryStream.....	168
12.2.2.1	Constructors.....	169
12.2.2.1.1	BinaryStream (InputStream).....	169
12.2.2.1.2	BinaryStream (InputStream, int).....	169
12.2.3	sqlj.runtime.DefaultRuntime.....	169

12.2.3.1	Constructors.	170
12.2.3.1.1	DefaultRuntime ().	170
12.2.3.2	Methods.	170
12.2.3.2.1	getDefaultConnection ().	170
12.2.3.2.2	getLoaderForClass (Class).	170
12.2.4	sqlj.runtime.ExecutionContext.	171
12.2.4.1	Variables.	172
12.2.4.1.1	ADD_BATCH_COUNT.	172
12.2.4.1.2	AUTO_BATCH.	172
12.2.4.1.3	EXEC_BATCH_COUNT.	172
12.2.4.1.4	EXCEPTION_COUNT.	173
12.2.4.1.5	NEW_BATCH_COUNT.	173
12.2.4.1.6	QUERY_COUNT.	173
12.2.4.1.7	UNLIMITED_BATCH.	174
12.2.4.2	Constructors.	174
12.2.4.2.1	ExecutionContext ().	174
12.2.4.3	Methods.	174
12.2.4.3.1	cancel ().	174
12.2.4.3.2	execute ().	175
12.2.4.3.3	executeBatch ().	176
12.2.4.3.4	executeQuery ().	177
12.2.4.3.5	executeUpdate ().	177
12.2.4.3.6	getBatchLimit ().	178
12.2.4.3.7	getBatchUpdateCounts ().	179
12.2.4.3.8	getFetchDirection ().	179
12.2.4.3.9	getFetchSize ().	179
12.2.4.3.10	getMaxFieldSize ().	180
12.2.4.3.11	getMaxRows ().	180
12.2.4.3.12	getNextResultSet ().	181
12.2.4.3.13	getNextResultSet (int).	181
12.2.4.3.14	getQueryTimeout ().	182
12.2.4.3.15	getUpdateCount ().	183
12.2.4.3.16	getWarnings ().	183
12.2.4.3.17	isBatching ().	183
12.2.4.3.18	registerStatement (ExecutionContext, Object, int).	184
12.2.4.3.19	releaseStatement ().	185
12.2.4.3.20	setBatching (boolean).	186
12.2.4.3.21	setBatchLimit (int).	186
12.2.4.3.22	setFetchDirection (int).	187
12.2.4.3.23	setFetchSize (int).	187
12.2.4.3.24	setMaxFieldSize (int).	188
12.2.4.3.25	setMaxRows (int).	188

12.2.4.3.26	setQueryTimeout (int).	189
12.2.5	sqlj.runtime.RuntimeContext.	189
12.2.5.1	Variables.	189
12.2.5.1.1	DEFAULT_DATA_SOURCE.	189
12.2.5.1.2	DEFAULT_RUNTIME.	190
12.2.5.1.3	PROPERTY_KEY.	190
12.2.5.2	Constructors.	190
12.2.5.2.1	RuntimeContext ().	190
12.2.5.3	Methods.	190
12.2.5.3.1	getDefaultConnection ().	190
12.2.5.3.2	getLoaderForClass (Class).	191
12.2.5.3.3	getRuntime ().	191
12.2.6	sqlj.runtime.StreamWrapper.	192
12.2.6.1	Constructors.	192
12.2.6.1.1	StreamWrapper (InputStream).	192
12.2.6.1.2	StreamWrapper (InputStream, int).	193
12.2.6.2	Methods.	193
12.2.6.2.1	getInputStream ().	193
12.2.6.2.2	getLength ().	193
12.2.6.2.3	setLength (int).	194
12.2.7	sqlj.runtime.UnicodeStream.	194
12.2.7.1	Constructors.	195
12.2.7.1.1	UnicodeStream (InputStream).	195
12.2.7.1.2	UnicodeStream (InputStream, int).	195
12.2.8	sqlj.runtime.CharacterStream.	195
12.2.8.1	Constructors.	196
12.2.8.1.1	CharacterStream (Reader).	196
12.2.8.1.2	CharacterStream (Reader, int).	196
12.2.8.2	Methods.	196
12.2.8.2.1	getReader ().	196
12.2.8.2.2	getLength ().	197
12.2.8.2.3	setLength (int).	197
12.2.9	sqlj.runtime.SQLNullException.	197
12.2.9.1	Constructors.	198
12.2.9.1.1	SQLException ().	198
13	Package sqlj.runtime.profile.	199
13.1	SQLJ sqlj.runtime.profile Interfaces.	199
13.1.1	sqlj.runtime.profile.BatchContext.	199
13.1.1.1	Methods.	199
13.1.1.1.1	clearBatch ().	199
13.1.1.1.2	executeBatch ().	199
13.1.1.1.3	setBatchLimit (int).	200

13.1.2	sqlj.runtime.profile.ConnectedProfile.	200
13.1.2.1	Methods.	202
13.1.2.1.1	close ().	202
13.1.2.1.2	getConnection ().	202
13.1.2.1.3	getProfileData ().	202
13.1.2.1.4	getStatement (int, Map).	203
13.1.2.1.5	getStatement (int, BatchContext, Map).	203
13.1.3	sqlj.runtime.profile.Customization.	204
13.1.3.1	Methods.	205
13.1.3.1.1	acceptsConnection (Connection).	205
13.1.3.1.2	getProfile (Connection, Profile).	205
13.1.4	sqlj.runtime.profile.Loader.	206
13.1.4.1	Methods.	206
13.1.4.1.1	getResourceAsStream (String).	206
13.1.4.1.2	loadClass (String).	207
13.1.5	sqlj.runtime.profile.RTResultSet.	207
13.1.5.1	Methods.	211
13.1.5.1.1	clearWarnings ().	211
13.1.5.1.2	close ().	211
13.1.5.1.3	findColumn (String).	211
13.1.5.1.4	getArray (int).	212
13.1.5.1.5	getAsciiStreamWrapper (int).	213
13.1.5.1.6	getBigDecimal (int).	214
13.1.5.1.7	getBinaryStreamWrapper (int).	214
13.1.5.1.8	getBlob (int).	215
13.1.5.1.9	getBooleanNotNull (int).	216
13.1.5.1.10	getBooleanWrapper (int).	217
13.1.5.1.11	getByteNotNull (int).	218
13.1.5.1.12	getBytes (int).	219
13.1.5.1.13	getByteWrapper (int).	219
13.1.5.1.14	getCharacterStreamWrapper (int).	220
13.1.5.1.15	getClob (int).	221
13.1.5.1.16	getColumnCount ().	222
13.1.5.1.17	getCursorName ().	222
13.1.5.1.18	getDate (int).	223
13.1.5.1.19	getDoubleNotNull (int).	224
13.1.5.1.20	getDoubleWrapper (int).	225
13.1.5.1.21	getFloatNotNull (int).	225
13.1.5.1.22	getFloatWrapper (int).	226
13.1.5.1.23	getIntNotNull (int).	227
13.1.5.1.24	getIntWrapper (int).	228
13.1.5.1.25	getJDBCResultSet ().	229

13.1.5.1.26	getLongNotNull (int).....	229
13.1.5.1.27	getLongWrapper (int).....	230
13.1.5.1.28	getObject (int, Class).....	231
13.1.5.1.29	getRef (int).....	232
13.1.5.1.30	getShortNotNull (int).....	233
13.1.5.1.31	getShortWrapper (int).....	234
13.1.5.1.32	getString (int).....	234
13.1.5.1.33	getTime (int).....	235
13.1.5.1.34	getTimestamp (int).....	236
13.1.5.1.35	getUnicodeStreamWrapper (int).....	237
13.1.5.1.36	getURL (int).....	238
13.1.5.1.37	getWarnings ()......	238
13.1.5.1.38	isClosed ()......	239
13.1.5.1.39	isValidRow ()......	239
13.1.5.1.40	next ()......	240
13.1.6	sqlj.runtime.profile.RTStatement.....	240
13.1.6.1	Methods.....	245
13.1.6.1.1	cancel ()......	245
13.1.6.1.2	clearWarnings ()......	246
13.1.6.1.3	execute ()......	246
13.1.6.1.4	executeComplete ()......	246
13.1.6.1.5	executeRTQuery ()......	247
13.1.6.1.6	executeUpdate ()......	248
13.1.6.1.7	getArray (int).....	248
13.1.6.1.8	getBatchContext ()......	249
13.1.6.1.9	getBigDecimal (int).....	250
13.1.6.1.10	getBlob (int).....	250
13.1.6.1.11	getBooleanNotNull (int).....	251
13.1.6.1.12	getBooleanWrapper (int).....	252
13.1.6.1.13	getByteNotNull (int).....	253
13.1.6.1.14	getBytes (int).....	254
13.1.6.1.15	getByteWrapper (int).....	254
13.1.6.1.16	getClob (int).....	255
13.1.6.1.17	getDate (int).....	256
13.1.6.1.18	getDoubleNotNull (int).....	257
13.1.6.1.19	getDoubleWrapper (int).....	257
13.1.6.1.20	getFloatNotNull (int).....	258
13.1.6.1.21	getFloatWrapper (int).....	259
13.1.6.1.22	getIntNotNull (int).....	260
13.1.6.1.23	getIntWrapper (int).....	261
13.1.6.1.24	getJDBCCallableStatement ()......	261
13.1.6.1.25	getJDBCPreparedStatement ()......	262

13.1.6.1.26	getLongNotNull (int).....	262
13.1.6.1.27	getLongWrapper (int).....	263
13.1.6.1.28	getMaxFieldSize ()......	264
13.1.6.1.29	getMaxRows ()......	264
13.1.6.1.30	getMoreResults (int).....	265
13.1.6.1.31	getObject (int, Class).....	265
13.1.6.1.32	getQueryTimeout ()......	267
13.1.6.1.33	getRef (int).....	267
13.1.6.1.34	getResultSet ()......	268
13.1.6.1.35	getShortNotNull (int).....	268
13.1.6.1.36	getShortWrapper (int).....	269
13.1.6.1.37	getString (int).....	270
13.1.6.1.38	getTime (int).....	271
13.1.6.1.39	getTimestamp (int).....	272
13.1.6.1.40	getUpdateCount ()......	272
13.1.6.1.41	getURL ()......	273
13.1.6.1.42	getWarnings ()......	274
13.1.6.1.43	isBatchable ()......	274
13.1.6.1.44	isBatchCompatible ()......	275
13.1.6.1.45	getArray (int, Array).....	276
13.1.6.1.46	setAsciiStreamWrapper (int, AsciiStream).....	277
13.1.6.1.47	setBigDecimal (int, BigDecimal).....	277
13.1.6.1.48	setBinaryStreamWrapper (int, BinaryStream).....	278
13.1.6.1.49	setBlob (int, Blob).....	279
13.1.6.1.50	setBoolean (int, boolean).....	279
13.1.6.1.51	setBooleanWrapper (int, Boolean).....	280
13.1.6.1.52	setByte (int, byte).....	281
13.1.6.1.53	setBytes (int, byte).....	282
13.1.6.1.54	setByteWrapper (int, Byte).....	282
13.1.6.1.55	setCharacterStreamWrapper (int, CharacterStream).....	283
13.1.6.1.56	setClob (int, Clob).....	284
13.1.6.1.57	setDate (int, Date).....	284
13.1.6.1.58	setDouble (int, double).....	285
13.1.6.1.59	setDoubleWrapper (int, Double).....	286
13.1.6.1.60	setFloat (int, float).....	287
13.1.6.1.61	setFloatWrapper (int, Float).....	287
13.1.6.1.62	setInt (int, int).....	288
13.1.6.1.63	setIntWrapper (int, Integer).....	289
13.1.6.1.64	setLong (int, long).....	289
13.1.6.1.65	setLongWrapper (int, Long).....	290
13.1.6.1.66	setMaxFieldSize (int).....	291
13.1.6.1.67	setMaxRows (int).....	291

13.1.6.1.68	setObject ().	292
13.1.6.1.69	setQueryTimeout (int).	292
13.1.6.1.70	setRef (int, Ref).	293
13.1.6.1.71	setShort (int, short).	294
13.1.6.1.72	setShortWrapper (int, Short).	294
13.1.6.1.73	setString (int, String).	295
13.1.6.1.74	setTime (int, Time).	296
13.1.6.1.75	setTimestamp (int, Timestamp).	296
13.1.6.1.76	setUnicodeStreamWrapper (int, UnicodeStream).	297
13.1.6.1.77	setURL (int, URL).	298
13.1.7	sqlj.runtime.profile.SerializedProfile.	298
13.1.7.1	Methods.	299
13.1.7.1.1	getProfileAsStream ().	299
13.2	SQLJ sqlj.runtime.profile Classes.	300
13.2.1	sqlj.runtime.profile.DefaultLoader.	300
13.2.1.1	Constructors.	300
13.2.1.1.1	DefaultLoader (ClassLoader).	300
13.2.1.2	Methods.	300
13.2.1.2.1	getResourceAsStream (String).	300
13.2.1.2.2	loadClass (String).	301
13.2.2	sqlj.runtime.profile.EntryInfo.	302
13.2.2.1	Variables.	302
13.2.2.1.1	BLOCK.	302
13.2.2.1.2	CALL.	302
13.2.2.1.3	CALLABLE_STATEMENT.	303
13.2.2.1.4	COMMIT.	303
13.2.2.1.5	EXECUTE.	303
13.2.2.1.6	EXECUTE_QUERY.	304
13.2.2.1.7	EXECUTE_UPDATE.	304
13.2.2.1.8	ITERATOR_CONVERSION.	304
13.2.2.1.9	NAMED_RESULT.	305
13.2.2.1.10	NO_RESULT.	305
13.2.2.1.11	OTHER.	305
13.2.2.1.12	POSITIONED.	306
13.2.2.1.13	POSITIONED_RESULT.	306
13.2.2.1.14	PREPARED_STATEMENT.	306
13.2.2.1.15	QUERY.	307
13.2.2.1.16	QUERY_FOR_UPDATE.	307
13.2.2.1.17	RELEASE_SAVEPOINT.	307
13.2.2.1.18	ROLLBACK.	308
13.2.2.1.19	SAVEPOINT.	308
13.2.2.1.20	SET_TRANSACTION.	308

13.2.2.1.21	SINGLE_ROW_QUERY.....	309
13.2.2.1.22	STATEMENT.....	309
13.2.2.1.23	UNTYPED_SELECT.....	309
13.2.2.1.24	VALUES.....	309
13.2.2.2	Constructors.....	310
13.2.2.2.1	EntryInfo ()......	310
13.2.2.3	Methods.....	310
13.2.2.3.1	executeTypeToString (int).....	310
13.2.2.3.2	getDescriptor ()......	310
13.2.2.3.3	getExecuteType ()......	311
13.2.2.3.4	getLineNumber ()......	312
13.2.2.3.5	getParamCount ()......	312
13.2.2.3.6	getParamInfo (int).....	312
13.2.2.3.7	getResultSetCount ()......	313
13.2.2.3.8	getResultSetInfo (int).....	313
13.2.2.3.9	getResultSetName ()......	314
13.2.2.3.10	getResultSetType ()......	314
13.2.2.3.11	getRole ()......	315
13.2.2.3.12	getSQLString ()......	316
13.2.2.3.13	getStatementType ()......	316
13.2.2.3.14	getTransactionDescriptor ()......	316
13.2.2.3.15	isDefinedRole (int).....	317
13.2.2.3.16	isValidDescriptor (Object, int).....	317
13.2.2.3.17	isValidExecuteType (int).....	318
13.2.2.3.18	isValidResultSetType (int).....	318
13.2.2.3.19	isValidRole (int).....	319
13.2.2.3.20	isValidStatementType (int).....	319
13.2.2.3.21	resultSetTypeToString (int).....	320
13.2.2.3.22	roleToString (int).....	320
13.2.2.3.23	statementTypeToString (int).....	321
13.2.2.3.24	validateObject ()......	321
13.2.3	sqlj.runtime.profile.Profile.....	321
13.2.3.1	Constructors.....	322
13.2.3.1.1	Profile (Loader).....	322
13.2.3.2	Methods.....	323
13.2.3.2.1	deregisterCustomization (Customization).....	323
13.2.3.2.2	getConnectedProfile (Connection).....	323
13.2.3.2.3	getContextName ()......	324
13.2.3.2.4	getCustomizations ()......	324
13.2.3.2.5	getJavaType (String).....	324
13.2.3.2.6	getJavaType (TypeInfo).....	325
13.2.3.2.7	getLoader ()......	326

13.2.3.2.8	getProfileData ().	326
13.2.3.2.9	getProfileName ().	326
13.2.3.2.10	getTimestamp ().	326
13.2.3.2.11	instantiate (Loader, InputStream).	327
13.2.3.2.12	instantiate (Loader, String).	327
13.2.3.2.13	registerCustomization (Customization).	329
13.2.3.2.14	registerCustomization (Customization, Customization).	329
13.2.3.2.15	replaceCustomization (Customization, Customization).	330
13.2.4	sqlj.runtime.profile.ProfileData.	330
13.2.4.1	Constructors.	331
13.2.4.1.1	ProfileData ().	331
13.2.4.2	Methods.	331
13.2.4.2.1	getEntryInfo (int).	331
13.2.4.2.2	getProfile ().	331
13.2.4.2.3	getSourceFile ().	332
13.2.4.2.4	size ().	332
13.2.5	sqlj.runtime.profile.SetTransactionDescriptor.	332
13.2.5.1	Variables.	333
13.2.5.1.1	READ_NONE.	333
13.2.5.1.2	READ_ONLY.	333
13.2.5.1.3	READ_WRITE.	333
13.2.5.2	Constructors.	334
13.2.5.2.1	SetTransactionDescriptor (int, int).	334
13.2.5.3	Methods.	334
13.2.5.3.1	getAccessMode ().	334
13.2.5.3.2	getIsolationLevel ().	335
13.2.6	sqlj.runtime.profile.TypeInfo.	335
13.2.6.1	Variables.	335
13.2.6.1.1	IN.	335
13.2.6.1.2	INOUT.	336
13.2.6.1.3	OUT.	336
13.2.6.2	Constructors.	336
13.2.6.2.1	TypeInfo ().	336
13.2.6.3	Methods.	337
13.2.6.3.1	getJavaTypeName ().	337
13.2.6.3.2	getMarkerIndex ().	337
13.2.6.3.3	getMode ().	338
13.2.6.3.4	getName ().	338
13.2.6.3.5	getSQLType ().	339
13.2.6.3.6	getSQLTypeName ().	339
13.2.6.3.7	isValidMode (int).	340
13.2.6.3.8	isValidSQLType (int).	340

13.2.6.3.9	modeToString (int).....	341
13.2.6.3.10	SQLTypeToString (int).....	341
13.2.6.3.11	validateObject ()......	342
14	sqlj.runtime.profile.util.ProfileCustomizer.....	343
14.1	Methods.....	345
14.1.1	acceptsConnection (Connection).....	345
14.1.2	customize (Profile, Connection, ErrorLog).....	345
15	Status codes.....	347
15.1	SQLSTATE.....	347
16	Conformance.....	349
16.1	Claims of conformance to SQL/OLB.....	349
16.2	Additional conformance requirements for SQL/OLB.....	349
16.3	Implied feature relationships of SQL/OLB.....	349
Annex A	SQL Conformance Summary.....	351
Annex B	Implementation-defined elements.....	353
Annex C	Implementation-dependent elements.....	357
Annex D	SQL feature taxonomy.....	361
Annex E	SQLJ tutorial.....	363
E.1	Design goals.....	363
E.2	Advantages of SQLJ over JDBC.....	363
E.3	Consistency with existing embedded SQL languages.....	364
E.4	Examples.....	365
E.4.1	Host variables.....	365
E.4.2	Host expressions.....	365
E.4.3	SQLJ clauses.....	366
E.4.4	Database connection context.....	366
E.4.5	Default connection context.....	366
E.4.6	Iterators.....	367
E.4.6.1	Positional bindings to columns.....	367
E.4.6.2	Named bindings to columns.....	368
E.4.6.3	Providing names for columns of queries.....	369
E.4.7	Calls to stored procedures and functions.....	369
E.4.8	Using multiple SQLJ contexts and connections.....	370
E.4.9	SQL execution control and status.....	371
E.4.10	Multiple JDBC ResultSets from stored procedure calls.....	372
E.4.11	Creating an SQLJ iterator object from a JDBC ResultSet object.....	372
E.4.12	Obtaining a JDBC ResultSet object from an untyped iterator object.....	373
E.4.13	Working with user-defined types.....	373
E.4.14	Example program.....	375
E.4.15	Host variable definition.....	376

Index.....377

TABLES

Page

1	SQLJ output assignability.	20
2	SQLJ input assignability.	26
3	Association of roles with SQLJ <executable clause>s.	63
4	SQLJ type properties.	67
5	Methods retained from java.sql.ResultSet.	208
6	Methods not retained from java.sql.ResultSet.	209
7	Additional methods unique to RTResultSet.	210
8	Methods retained from java.sql.Statement.	241
9	Methods not retained from java.sql.Statement.	242
10	Methods retained from java.sql.PreparedStatement.	242
11	Methods not retained from java.sql.PreparedStatement.	243
12	Methods retained from java.sql.CallableStatement.	244
13	Methods not retained from java.sql.CallableStatement.	244
14	Additional methods unique to RTStatement.	245
15	Customize Result Interpretation.	344
16	SQLSTATE class and subclass values.	347
17	Implied feature relationships of SQL/OLB.	349
18	Feature taxonomy for optional features.	361

INTRODUCTION

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

- 1) [Clause 1, “Scope”](#), specifies the scope of this part of ISO/IEC 9075.
- 2) [Clause 2, “Normative references”](#), identifies additional standards and publically-available specifications that, through reference in this part of ISO/IEC 9075, constitute provisions of this part of ISO/IEC 9075.
- 3) [Clause 3, “Definitions, notations, and conventions”](#), defines the notations and conventions used in this part of ISO/IEC 9075.
- 4) [Clause 4, “Concepts”](#), presents concepts used in the definition of the Object Language Bindings.
- 5) [Clause 5, “Lexical elements”](#), defines the lexical elements of the language.
- 6) [Clause 6, “Scalar expressions”](#), defines the elements of the language that produce scalar values.
- 7) [Clause 7, “Additional common elements”](#), defines additional language elements that are used in various parts of the language.
- 8) [Clause 8, “Embedded SQL”](#), defines the host language embeddings.
- 9) [Clause 9, “Binary portability”](#), provides an overview of the binary portability and profile customization requirements for SQLJ.
- 10) [Clause 10, “SQLJ grammar constructs”](#), defines the BNF conventions, common subelements, and datatype mappings for SQLJ.
- 11) [Clause 11, “<SQLJ specific clause> and contents”](#), defines the syntax and rules for SQLJ constructs.
- 12) [Clause 12, “Package sqlj.runtime”](#), specifies the SQLJ runtime package.
- 13) [Clause 13, “Package sqlj.runtime.profile”](#), specifies the SQLJ runtime profile package.
- 14) [Clause 14, “sqlj.runtime.profile.util.ProfileCustomizer”](#), specifies the SQLJ profile customizer class.
- 15) [Clause 15, “Status codes”](#), defines SQLSTATE values related to Object Language Bindings.
- 16) [Clause 16, “Conformance”](#), defines the criteria for conformance to this part of ISO/IEC 9075.
- 17) [Annex A, “SQL Conformance Summary”](#), is an informative Annex. It summarizes the conformance requirements of the SQL language.
- 18) [Annex B, “Implementation-defined elements”](#), is an informative Annex. It lists those features for which the body of this part of ISO/IEC 9075 states that the syntax, the meaning, the returned results, the effect on SQL-data and/or schemas, or any other behavior is partly or wholly implementation-defined.
- 19) [Annex C, “Implementation-dependent elements”](#), is an informative Annex. It lists those features for which the body of this part of ISO/IEC 9075 states that the syntax, the meaning, the returned results, the effect on SQL-data and/or schemas, or any other behavior is partly or wholly implementation-dependent.

- 20) [Annex D, “SQL feature taxonomy”](#), is an informative Annex. It identifies features of the SQL language specified in this part of ISO/IEC 9075 by a numeric identifier and a short descriptive name. This taxonomy is used to specify conformance and might be used to develop other profiles involving the SQL language.
- 21) [Annex E, “SQLJ tutorial”](#), is an informative Annex. It contains tutorial information about the features of the SQL language that are specified in this part of ISO/IEC 9075.

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

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

AUSTRALIAN STANDARD

Information technology — Database languages — SQL —

Part 10: Object Language Bindings (SQL/OLB)

1 Scope

This part of ISO/IEC 9075 defines extensions of Database language SQL to support embedding of SQL statements into programs written in the Java™ programming language (Java is a registered trademark of Sun Microsystems, Inc.). The embedding of SQL into Java is commonly known as “SQLJ”. This part of ISO/IEC 9075 specifies the syntax and semantics of SQLJ, as well as mechanisms to ensure binary portability of resulting SQLJ applications. In addition, it specifies a number of Java packages and their contained classes (including methods).

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

This page intentionally left blank.

2 Normative references

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

2.1 JTC 1 standards

[ISO646] ISO/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange*.

[Framework] ISO/IEC 9075-1:2003, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*.

[Foundation] ISO/IEC 9075-2:2003, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*.

[PSM] ISO/IEC 9075-4:2003, *Information technology — Database languages — SQL — Part 4: Persistent Stored Modules (SQL/PSM)*.

[MED] ISO/IEC 9075-9:2003, *Information technology — Database languages — SQL — Part 9: Management of External Data (SQL/MED)*.

[Schemata] ISO/IEC 9075-11:2003, *Information technology — Database languages — SQL — Part 11: Information and Definition Schemas (SQL/Schemata)*.

[UCS] ISO/IEC 10646-1:2000, *Information technology — Universal Multi-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*.

[UCSuppl] ISO/IEC 10646-2:2000, *Information technology — Universal Multi-Octet Coded Character Set (UCS) — Part 2: Supplementary Planes*.

[Ordering] ISO/IEC 14651:2001, *Information technology — International string ordering and comparison — Method for comparing character strings and description of the common template tailorable ordering*.

2.2 Other international standards

[Unicode 3.0] The Unicode Consortium, *The Unicode Standard, Version 3.0*, Reading, MA, Addison-Wesley Developers Press, 2000. ISBN 0-201-61633-5.

[Unicode 3.1] The Unicode Consortium, *The Unicode Standard, Version 3.1.0, Unicode Standard Annex #27: Unicode 3.1 (which amends The Unicode Standard, Version 3.0)*. 2001-03-23
<http://www.unicode.org/unicode/reports/tr27/>

[Unicode15] Davis, Mark and Dürst, Martin. *Unicode Standard Annex #15, Unicode Normalization Forms, Version 21.0*, 2001-03-23. The Unicode Consortium.

<http://www.unicode.org/unicode/reports/tr15/tr15-21.html>

[Unicode19] Davis, Mark. *Unicode Standard Annex #19, UTF-32, Version 8.0*, 2001-03-23. The Unicode Consortium.

<http://www.unicode.org/unicode/reports/tr19/tr19-8.html>

[Unicode10] Davis, Mark and Whistler, Ken. *Unicode Technical Standard #10, Unicode Collation Algorithm, Version 8.0*, 2001-03-23. The Unicode Consortium.

<http://www.unicode.org/unicode/reports/tr10/tr10-8.html>

[Java] *The Java™ Language Specification, Second Edition*, Bill Joy (Editor), Guy Steele, James Gosling, and Gilad Bracha, Addison-Wesley, 2000, ISBN 0-201-31008-2

[JDBC] *JDBC™ 3.0 Specification, Final Release*, John Ellis & Linda Ho with Maydene Fisher, Sun Microsystems, Inc., October, 2001

[JNDI] *Java Naming and Directory Interface™ Application Programming Interface (JNDI API)*, Sun Microsystems, Inc., July 14, 1999.

[JavaBeans] *The JavaBeans™ 1.01 Specification*

<http://java.sun.com/products/javabeans/docs/spec.html>

[JDBCtutorial] *JDBC™ API Tutorial and Reference, Second Edition: Universal Data Access for the Java™ 2 Platform*, Seth White, Maydene Fisher, Rick Cattell, Graham Hamilton, Mark Hapner, Addison Wesley, Reading, Massachusetts, 1999, ISBN: 0-201-43328-1

[RFC2368] RFC 2368, *The mailto URL scheme*, P. Hoffman, L. Masinter, J. Zawinski, July, 1998

<http://www.ietf.org/rfc/rfc2368.txt>

[RFC2396] RFC 2396, *Uniform Resource Identifiers (URI): Generic Syntax*, T. Berners-Lee, R. Fielding, L. Masinter, August, 1998

<http://www.ietf.org/rfc/rfc2396.txt>

3 Definitions, notations, and conventions

3.1 Definitions

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

3.1.1 Definitions provided in Part 10

For the purposes of this part of ISO/IEC 9075, the following definitions apply:

- 3.1.1.1 accessor method:** A method that, when invoked, accesses column data returned by a result set iterator object. An *accessor method* is either a *named accessor method* or a *positioned accessor method*. A named accessor method is declared as the result of an <iterator declaration clause> containing an <iterator spec declaration> of <named iterator>. A named accessor method derives both its name and its result datatype from its defining <named iterator> clause. A positioned accessor method is declared as the result of an <iterator declaration clause> containing an <iterator spec declaration> of <positioned iterator>. A positioned accessor method derives its result datatype from its defining <positioned iterator> clause.
- 3.1.1.2 customization:** An implementation-specific process of tailoring an SQLJ application's embedded SQL to run against a target SQL-implementation. This frequently involves creating new versions of some of the object instances stored in the SQLJ translation-generated profile, to create customized profile object instances.
- 3.1.1.3 generated connection class:** A class whose methods, when invoked, maintain a named database connection. The signature of this class is produced as a side effect of the direct inclusion of a <connection declaration clause> in a Java application.
- 3.1.1.4 generated iterator class:** A class whose methods, when invoked, provide access to the rows and columns of SQL queries associated with result set iterators. A *generated iterator class* is either a *generated named iterator class* or a *generated positioned iterator class*. The signature of a generated named iterator class is produced as a side effect of the inclusion of a <iterator declaration clause> that contains an <iterator spec declaration> of <named iterator>, and it specifies named accessor methods. The signature of a generated positioned iterator class is produced as a side effect of the inclusion of a <iterator declaration clause> that contains an <iterator spec declaration> of <positioned iterator>, and it specifies positioned accessor methods.
- 3.1.1.5 getter method:** A method defined on objects of either the `RTStatement` or `RTResultSet` class or a subclass of such a class, and that when invoked populates host variables of a given datatype when those host variables appear as bind variables.
- 3.1.1.6 implementation-specific:** Possibly differing between SQL-implementations, but provided as part of each particular SQL-implementation.

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

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

NOTE 1 — For interoperability with JDBC, the Java primitive datatype char is intentionally omitted from this list.

3.1.1.9 l-valued expression: A Java expression that is allowed to appear as the *LeftHandSide* of a Java *assignment*, as defined in [Java]. For example, an l-valued expression may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access or an array access.

3.1.1.10 profile: A 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. The serialized objects can then be accessed for additional processing by a customizer or by the SQLJ runtime system.

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

3.1.1.12 setter method: A 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 database.

3.1.1.13 SQLJ translation: The 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.

3.2 Notation

This Subclause modifies Subclause 3.2, “Notation”, in ISO/IEC 9075-2.

Insert this paragraph All keywords and SQL <token>s contained within a <statement spec clause> or an <assignment spec clause> (*i.e.*, those productions immediately contained between the <left brace> and <right brace> of an enclosing <SQLJ specific clause>) are case-insensitive except for:

- Any token appearing within the <expression> of an <embedded Java expression> is a Java token and therefore is case-sensitive.
- Any SQL <token> denoting an SQL <delimited identifier> is case-sensitive.

Insert this paragraph All other keywords and tokens are case-sensitive as specified by [Java].

NOTE 2 — If a <query clause> is assigned to a <named iterator>, column names in the <query clause>s select list that differ only in the case of one or more characters should use the SQL AS clause to avoid ambiguity, even if one or both of those column names are specified using <delimited identifier>s.

3.3 Conventions

This Subclause modifies Subclause 3.3, “Conventions”, in ISO/IEC 9075-2.

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

3.3.1 Use of terms

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

3.3.1.1 Other terms

This Subclause modifies Subclause 3.3.1.1, “Other terms”, in ISO/IEC 9075-2.

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

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

- An instance of a class that implements the interface *java type*.
- An instance of a class that implements an interface that extends the interface *java type*.
- An instance of a subclass of such class.

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

- A *connection context object* is an instance of a class that is generated by an SQLJ translator as the result of processing a <connection declaration clause>. The generated class implements the interface `sqlj.runtime.ConnectionContext`.
- An *execution context object* is an instance of the class `sqlj.runtime.ExecutionContext`.
- A *named iterator* is an instance of a class that is generated by an SQLJ translator as the result of processing an <iterator declaration clause> that contains a <named iterator>. The generated class implements the interface `sqlj.runtime.NamedIterator`.
- A *positioned iterator* is an instance of a class that is generated by an SQLJ translator as the result of processing a <iterator declaration clause> that contains a <positioned iterator>. The generated class implements the interface `sqlj.runtime.PositionedIterator`.
- An *iterator* is either a named iterator or a positioned iterator.

3.3.2 Specification of translator-generated classes

The conventions used in this part of ISO/IEC 9075 are defined in ISO/IEC 9075-1, with the following additions. Descriptions of translator-generated classes and their relationships to syntax elements contained in <executable spec clause> specified in the Subclauses of [Clause 10, “SQLJ grammar constructs”](#), and [Clause 11, “<SQLJ specific clause> and contents”](#), are specified in terms of:

- **Function**

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

- **Signature**

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

- **Definitions and Rules**

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

- **Profile EntryInfo Properties**

Defines the properties of the profile EntryInfo object created for this syntax element, if any. If an EntryInfo Java field is not listed explicitly in this heading, it defaults to the value described in [Subclause 9.9, “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.

NOTE 3 — The Profile EntryInfo Properties, Binary Composition, and Code Generation headings are intended for implementations of SQLJ translators and/or runtime components. They can be skipped by those interested only in the SQLJ programming language.

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

This page intentionally left blank.

4 Concepts

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

4.1 Introduction

ISO/IEC 9075-2 specifies *embedded SQL* for the programming languages: Ada, C, COBOL, Fortran, MUMPS, Pascal, and PL/I. This part of ISO/IEC 9075 provides a similar embedding, *<embedded SQL Java program>*, tailored for the Java programming language.

SQLJ comprises a set of clauses that extend Java programs to include SQL constructs. An SQLJ *translator* is a utility that transforms those *SQLJ clauses* into standard Java code that accesses the database through a call interface. The output of an SQLJ translator is a generated Java source program that can then be compiled by any Java compiler. Java programs containing embedded SQL are amenable to static analysis of SQL statements for the purposes of syntax checking, type checking and schema validation.

SQLJ is compatible with a range of SQLJ translation approaches. In a client environment, SQLJ can be processed by a standalone SQLJ-to-Java translator or by a Java integrated development environment (IDE), to emit standard Java code with calls to any appropriate database API. If Java is supported as a *stored procedure language* (in an RDBMS server), SQLJ translation can be integrated with Java and SQL compilation for ease-of-use and performance.

4.2 Embedded syntax

This Subclause modifies Subclause 4.23, “Embedded syntax”, in ISO/IEC 9075-2.

Replace 1st paragraph An *<embedded SQL host program>* (*<embedded SQL Ada program>*, *<embedded SQL C program>*, *<embedded SQL COBOL program>*, *<embedded SQL Fortran program>*, *<embedded SQL MUMPS program>*, *<embedded SQL Pascal program>*, *<embedded SQL PL/I program>*, or *<embedded SQL Java program>*) is a compilation unit that consists of programming language text and SQL text. The programming language text shall conform to the requirements of a specific standard programming language. The SQL text shall consist of one or more *<embedded SQL statement>*s and, optionally, one or more *<embedded SQL declare section>*s, as defined in this International Standard. This allows database applications to be expressed in a hybrid form in which SQL-statements are embedded directly in a compilation unit. For all *<embedded SQL host program>*s other than *<embedded SQL Java program>*s, such a hybrid compilation unit is defined to be equivalent to a standard compilation unit in which the SQL-statements have been replaced by standard procedure or subroutine calls of *<externally-invoked procedure>*s in a separate SQL-client module, and in which each has been removed and the declarations contained therein have been suitably transformed into standard host-language syntax. 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 standard compilation unit in which the SQL statements have been replaced by use of Java classes whose methods, when invoked, make use of JDBC.

Replace 3rd paragraph An implementation may reserve a portion of the name space in the <embedded SQL host program>. For all <embedded SQL host program>s except <embedded SQL Java program>s, name space may be reserved for the names of procedures or subroutines that are generated to replace SQL-statements and for program variables and branch labels that may be generated as required to support the calling of these procedures or subroutines; whether this reservation is made is implementation-defined. They may similarly reserve name space for the <module name> and <procedure name>s of the generated <SQL-client module definition> that may be associated with the resulting standard compilation unit. The portion of the name space to be so reserved, if any, is implementation-defined. When the <embedded SQL host program> immediately contains an <embedded SQL Java program>, name space is reserved for variable names, internal class names, and resource file names. See [Subclause 10.1, “SQLJ reserved names”](#), for further details.

4.3 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 stored procedures: *e.g.*, CALL MYPROC(:x, :y, :z)
- Invocations of stored functions: *e.g.*, VALUES(MYFUN(:x))
- Assignment statement: SET

4.4 Character strings

This Subclause modifies [Subclause 4.2, “Character strings”](#), in ISO/IEC 9075-2.

4.4.1 Unicode support

Java relies on the Unicode character set [Unicode] (also known as ISO/IEC 10646-1, *Universal Multi-Octet Coded Character Set (UCS)*; see [UCS]) for String data and for identifiers. That allows Java to represent most character data in a uniform way. [Foundation] defines support for Unicode through its UTF8, UTF16, and UTF32 character sets, which represent different encodings for Unicode character data. When character data is moved between an SQL-server and an SQL/OLB host program, an SQL/OLB implementation that provides SQL character set support for UTF8, UTF16, and/or UTF32 is required to support implicit conversion between Java string data and the supported Unicode encodings. Any support for implicit conversions to and from character sets other than Unicode is implementation-defined. Because of Java's reliance on Unicode as an internal

representation for character data, the SQL/OLB specification does not define support for host variables that hold character data based on character sets other than Unicode.

The rules in [Foundation] 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.

4.4.2 Character sets

This Subclause modifies Subclause 4.2.7, “Character sets”, in ISO/IEC 9075-2.

- Augment 1st bullet If <embedded SQL host program> immediately contains an <embedded SQL Java program>, then <SQL special character> shall include the <number sign> (#) in addition to the characters that it otherwise required to contain.

4.5 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.

4.6 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 legal Java expression that shall result in either: a single value (in the case of IN or INOUT mode) **and/or** the site of 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.

4.7 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 database.

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

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

In an SQLJ executable clause, the tokens that appear inside of the curly braces are SQL <token>s and <separator>s, except for the host variables. All host variables 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 curly braces of an SQLJ executable clause.

In general, SQL <token>s are case-insensitive (except for identifiers delimited by double quotes), and can be written in upper, lower, or mixed case. Java tokens, however, are case-sensitive. For clarity in examples, we write case-insensitive SQL <token>s in uppercase, and write Java tokens in lowercase or mixed case.

Throughout this document, we use the lowercase **null** to represent the Java “null” value, and the uppercase **NULL** to represent the SQL null value.

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

4.8 Database connection context

Each SQLJ executable clause requires, either explicitly or implicitly, a *connection context object* that designates the database connection at which the SQL operation specified in that clause will be executed. The connection context object is an optional expression, delimited by square brackets, that immediately follows the token **#sql**.

The connection context object designates a database 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 JDBC Connection, a URL (see [RFC2368], and [RFC2396] for more details about URLs), or a URL and information such as a user name and password, a connection context class has methods for identifying and, if necessary, opening a connection to a database. At run time, an SQLJ program shall use one of those methods to establish a database connection before any SQLJ clauses are executed.

NOTE 4 — The connection context object implicitly specifies a database and associated schemas, and the default catalog and schema, as per a JDBC Connection object.

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. Portable applications should always use explicit connection contexts.

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

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

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

If an SQLJ program is executing inside a database as a “stored procedure” (or is otherwise executing in an environment that automatically provides a connection context), calls to method **ConnectionContext.getDefaultContext** always return an object representing the schema 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 database 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 connection used is implementation-defined.

NOTE 5 — “JNDI” is the Java Naming Directory Interface, defined in [JNDI].

4.10 Schema checking using exemplar schemas

At SQLJ translation time, a connection context class plays a different role. It symbolizes the “type” of database schema to which the SQLJ program will connect at run time. The notion of the “type of a database schema” is informal. It includes the names and privileges associated with tables and views, the “shapes” of their rows, stored programs, and so forth. The type of a schema is symbolized by an *exemplar schema*, which is simply a database schema that contains the tables, views, programs, and privileges that would be required in order for the SQL operations 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 shall provide 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 database 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, stored functions, and stored procedures with the same names and types, and having privileges set appropriately.

If no appropriate exemplar schema is available, or if it is inconvenient to connect to a database during SQLJ program development, then the programmer is permitted to omit the exemplar schema for a connection type. SQLJ clauses to be executed on connections of that type will not then be schema checked at SQLJ translation time, and will instead be checked later at installation or customization time.

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

4.11 Using multiple SQLJ contexts and connections

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

4.12 Dynamic SQL and JDBC/SQLJ Connection interoperability

The SQLJ language provides direct support for SQL operations 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 operations from an SQLJ program, use JDBC. A `ConnectionContext` object contains a JDBC Connection object that can be used to create JDBC Statement objects needed for dynamic SQL operations.

4.12.1 Creating an SQLJ ConnectionContext from a JDBC connection

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

4.12.2 Obtaining a JDBC connection from an SQLJ ConnectionContext

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

4.12.3 Connection sharing

An SQLJ ConnectionContext always contains a JDBC Connection and relies upon it to provide communication with the underlying database 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 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 JDBC connection has auto commit mode on by default, an SQLJ connection context requires the auto commit mode to be specified explicitly upon construction.

4.12.4 Connection resource management

Calling the **close** method of a connection context object causes the associated JDBC Connection object and the underlying database connection to be closed. Since connection contexts are permitted to share the underlying database connection with other connection contexts and/or JDBC connections, it might not be desirable to close the underlying database connection when a connection context is closed. A programmer might wish to release the resources maintained by the connection context (*e.g.*, statement handles) without actually closing the underlying database connection. To this end, connection context classes also support a **close** method that takes as a boolean argument indicating whether or not to close the underlying database connection. Pass the constant **CLOSE_CONNECTION** if the database connection should be closed, and **KEEP_CONNECTION** if it should be retained. The variant of **close** that takes no arguments is a shorthand for calling **close(CLOSE_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 database connection for other JDBC and SQLJ objects that might be using it. Note that if no other JDBC or SQLJ objects are using the connection, then the database connection will also be closed and reclaimed by the garbage collection process.

Both SQLJ connection context objects and JDBC 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 JDBC connection associated with it. However, it is not sufficient to close only the JDBC connection returned by the **getConnection** method of a connection context. This is because the **close** method of a JDBC connection will not cause the containing connection context to be closed, and therefore, resources maintained by the connection context will not be released until it is garbage-collected.

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

4.13 SQL execution control and status

The execution semantics of SQL operations 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 operations. The **getXXX** and **setXXX** methods read and change the XXX value. Once set, they affect all SQL operations 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 operation to complete.

NOTE 6 — Runtime support of the above `ExecutionContext` Java fields, if set to anything other than their default values, is not part of Core SQLJ. It is implementation-defined whether or not such support is provided. If an implementation does not support an assigned setting, then an attempt to register a statement with the `ExecutionContext` causes an `SQLException` condition to be thrown: *OLB-specific error — unsupported feature*.

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

- **UpdateCount** specifies the number of rows updated, inserted, or deleted during the last operation.
- **SQLWarnings** describes any warnings that occurred during the last operation.

An execution context is associated either explicitly or implicitly with each executable SQL operation appearing in an SQLJ program. An execution context can be supplied explicitly as an argument to each SQL operation.

If explicit execution contexts are used, each SQL operation can be executed using a different execution context object. If an explicit connection context is also being used, both are allowed to be passed as arguments to the SQL operation.

If an execution context is not supplied explicitly as an argument to an SQL operation, a default execution context is used implicitly. The default execution context for a particular SQL operation is obtained via the `getExecutionContext()` method of the connection context used in the operation.

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.

4.14 Iterators

A capability central to SQL is the ability to execute queries that retrieve a “result set” of rows from the database. 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 is permitted to 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 a database 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 database schema.

4.15 Input and output assignability

An SQL type *ST* is *SQLJ output assignable* to a Java class or primitive type *JT* if Table 1, “SQLJ output assignability”, 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 1, “SQLJ output assignability”, (*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 1, “SQLJ output assignability”, 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 1, “SQLJ output assignability”, 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 1 — SQLJ output assignability

	java.sql.Types constants¹							
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

	java.sql.Types constants ¹							
float	x	x	x	x	x	x	x	x
double	x	x	x	x	x	x	x	x
java.lang.Boolean	x	x	x	x	x	x	x	x
java.lang.Byte	x	x	x	x	x	x	x	x
java.lang.Short	x	x	x	x	x	x	x	x
java.lang.Integer	x	x	x	x	x	x	x	x
java.lang.Long	x	x	x	x	x	x	x	x
java.lang.Float	x	x	x	x	x	x	x	x
java.lang.Double	x	x	x	x	x	x	x	x
java.lang.String	x	x	x	x	x	x	x	x
java.math.BigDecimal	x	x	x	x	x	x	x	x
byte[]								
java.sql.Array								
java.sql.Blob								
java.sql.Clob								
java.sql.Date								
java.sql.Ref								
java.sql.Time								
java.sql.Timestamp								
sqlj.runtime.AsciiStream								
sqlj.runtime.BinaryStream								
sqlj.runtime.CharacterStream								
sqlj.runtime.UnicodeStream								
java.net.URL								
<i>any other class/interface</i>								

	java.sql.Types constants ¹							
Java Data Types and Classes	NU	BT	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			

	java.sql.Types constants ¹							
sqlj.runtime.AsciiStream			x	x	x		x	x
sqlj.runtime.BinaryStream							x	x
sqlj.runtime.CharacterStream			x	x	x		x	x
sqlj.runtime.UnicodeStream			x	x	x		x	x
java.net.URL			x	x	x			
<i>any other class/interface</i>								
Java Data Types and Classes	LB	BL	DT	TM	TS	RF	DS	ST
boolean							x	x
byte							x	x
short							x	x
int							x	x
long							x	x
float							x	x
double							x	x
java.lang.Boolean							x	x
java.lang.Byte							x	x
java.lang.Short							x	x
java.lang.Integer							x	x
java.lang.Long							x	x
java.lang.Float							x	x
java.lang.Double							x	x
java.lang.String	x		x	x	x		x	x
java.math.BigDecimal							x	x
byte[]	x						x	x
java.sql.Array								

	java.sql.Types constants ¹							
java.sql.Blob		x					x	x
java.sql.Clob							x	x
java.sql.Date			x		x		x	x
java.sql.Ref						x		
java.sql.Time				x	x		x	x
java.sql.Timestamp			x		x		x	x
sqlj.runtime.AsciiStream	x						x	x
sqlj.runtime.BinaryStream	x						x	x
sqlj.runtime.CharacterStream	x						x	x
sqlj.runtime.UnicodeStream	x						x	x
java.net.URL							x	x
<i>any other class/interface</i>							x	x
Java Data Types and Classes	JO	OT	DL					
boolean								
byte								
short								
int								
long								
float								
double								
java.lang.Boolean								
java.lang.Byte								
java.lang.Short								
java.lang.Integer								
java.lang.Long								

	java.sql.Types constants ¹							
java.lang.Float								
java.lang.Double								
java.lang.String			x					
java.math.BigDecimal								
byte[]								
java.sql.Array				x				
java.sql.Blob								
java.sql.Clob								
java.sql.Date								
java.sql.Ref								
java.sql.Time								
java.sql.Timestamp								
sqlj.runtime.AsciiStream								
sqlj.runtime.BinaryStream								
sqlj.runtime.CharacterStream								
sqlj.runtime.UnicodeStream								
java.net.URL			x					
any other class/interface	x	x						
¹ where: TI corresponds to TINYINT, SI to SMALLINT, IN to INTEGER, BI to BIGINT, RL to REAL, FL to FLOAT, DB to DOUBLE, DC to DECIMAL, NU to NUMERIC, BT to BIT, CH to CHAR, VC to VARCHAR, LC to LONGVARCHAR, CL to CLOB, BI to BINARY, VB to VARBINARY, LB to LONGVARBINARY, BL to BLOB, DT to DATE, TM to TIME, TS to TIMESTAMP, RF to REF, DS to DISTINCT, ST to STRUCT, JO to JAVA_OBJECT, DL to DATALINK, and AY to ARRAY, and OT to OTHER								

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

- If the `java.sql.Types` value of *ST* is either DISTINCT or STRUCT, and *JT* is not one of the Java classes or primitive types identified in the first column of Table 2, “SQLJ input assignability”, (i.e., “any

other class/interface” applies), then the user-defined type map that is associated with the connection context class of the SQLJ clause for which input assignability is checked shall specify a Java class or primitive type *JT* that corresponds to *ST*.

- If the `java.sql.Types` value of *ST* is DISTINCT, and *JT* is one of the Java classes or primitive types identified in the first column of Table 2, “SQLJ input assignability”, then there exists an SQL type *ST1*, where *ST1* is either the representation type of *ST*, or a transform group has been specified for *ST* in the connection context class of the SQLJ clause for which input assignability is checked, and *ST1* is the input parameter type of the to-sql transform function or method of that transform group. *JT* shall be SQLJ input assignable to *ST1*.
- If the `java.sql.Types` value of *ST* is STRUCT, and *JT* is one of the Java classes or primitive types identified in the first column of Table 2, “SQLJ input assignability”, then a transform group has been specified for *ST* in the connection context class of the SQLJ clause for which input assignability is checked, and *JT* is SQLJ input assignable to the input parameter type of the to-sql transform function or method of that transform group.

Table 2 — SQLJ input assignability

	java.sql.Types constants¹							
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.sql.Types constants ¹							
java.lang.String	x	x	x	x	x	x	x	x
java.math.BigDecimal	x	x	x	x	x	x	x	x
byte[]								
java.sql.Array								
java.sql.Blob								
java.sql.Clob								
java.sql.Date								
java.sql.Ref								
java.sql.Time								
java.sql.Timestamp								
sqlj.runtime.AsciiStream								
sqlj.runtime.BinaryStream								
sqlj.runtime.CharacterStream								
sqlj.runtime.UnicodeStream								
java.net.URL								
<i>any other class/interface</i>								
Java Data Types and Classes	NU	BT	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.sql.Types constants ¹							
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	x
sqlj.runtime.BinaryStream							x	x
sqlj.runtime.CharacterStream			x	x	x	x	x	x
sqlj.runtime.UnicodeStream			x	x	x	x	x	x
java.net.URL			x	x	x			
<i>any other class/interface</i>								
Java Data Types and Classes	LB	BL	DT	TM	TS	RF	DS	ST
boolean							x	x
byte							x	x