A
JDBC Reference Information

This appendix contains detailed Java Database Connectivity (JDBC) reference information, including the following topics:

- Supported SQL-JDBC Data Type Mappings
- Supported SQL and PL/SQL Data Types
- About Using PL/SQL Types
- Using Embedded JDBC Escape Syntax
- Oracle JDBC Notes and Limitations

A.1 Supported SQL-JDBC Data Type Mappings

This section lists all the possible Java types to which a given SQL data type can have a valid mapping.

Oracle JDBC drivers will support these nondefault mappings. For example, to materialize SQL `CHAR` data in an `oracle.sql.CHAR` object, use the `getCHAR` method. To materialize it as a `java.math.BigDecimal` object, use the `getBigDecimal` method.

> **Note:**
>
> The classes, where `oracle.jdbc.OracleData` appears in italic, can be generated by Oracle JVM Web Services Call-Out Utility.

Table A-1 Valid SQL Data Type-Java Class Mappings

SQL data type	Java types
`BOOLEAN`	`boolean`, `java.lang.Boolean`, `oracle.sql.BOOLEAN`
`CHAR`, `VARCHAR2`, `LONG`	`java.lang.String`
	`oracle.sql.CHAR`

Table A-1 (Cont.) Valid SQL Data Type-Java Class Mappings

SQL data type	Java types
NUMBER	boolean
	char
	byte
	short
	int
	long
	float
	double
	java.lang.Byte
	java.lang.Short
	java.lang.Integer
	java.lang.Long
	java.lang.Float
	java.lang.Double
	java.math.BigDecimal
	oracle.sql.NUMBER
BINARY_INTEGER	boolean
	char
	byte
	short
	int
	long
BINARY_FLOAT	oracle.sql.BINARY_FLOAT
BINARY_DOUBLE	oracle.sql.BINARY_DOUBLE
DATE	oracle.sql.DATE
RAW	oracle.sql.RAW
BLOB	oracle.jdbc.OracleBlob
CLOB	oracle.jdbc.OracleClob
BFILE	oracle.sql.BFILE
ROWID	oracle.sql.ROWID
TIMESTAMP	oracle.sql.TIMESTAMP
TIMESTAMP WITH TIME ZONE	oracle.sql.TIMESTAMPTZ
TIMESTAMP WITH LOCAL TIME ZONE	oracle.sql.TIMESTAMPLTZ
ref cursor	java.sql.ResultSet
user defined named types, abstract data types	oracle.jdbc.OracleStruct
opaque named types	oracle.jdbc.OracleOpaque
nested tables and VARRAY named types	oracle.jdbc.OracleArray
references to named types	oracle.jdbc.OracleRef

> **Note:**
>
> - The type `UROWID` is not supported.
> - The `oracle.sql.Datum` class is abstract. The value passed to a parameter of type `oracle.sql.Datum` must be of the Java type corresponding to the underlying SQL type. Likewise, the value returned by a method with return type `oracle.sql.Datum` must be of the Java type corresponding to the underlying SQL type.

A.2 Supported SQL and PL/SQL Data Types

The tables in this section list SQL and PL/SQL data types, and Oracle JDBC driver support for them. The following table describes Oracle JDBC driver support for SQL data types.

Table A-2 Support for SQL Data Types

SQL Data Type	Supported by JDBC Drivers?
BFILE	yes
BLOB	yes
CHAR	yes
CLOB	yes
DATE	yes
NCHAR	no[1]
NCHAR VARYING	no
NUMBER	yes
NVARCHAR2	yes[2]
RAW	yes
REF	yes
ROWID	yes
UROWID	no
VARCHAR2	yes

1 The NCHAR type is supported indirectly. There is no corresponding java.sql.Types type, but if your application calls the formOfUse(NCHAR) method, then this type can be accessed.
2 In JSE 6, the NVARCHAR2 type is supported directly. In J2SE 5.0, the NVARCHAR2 type is supported indirectly. There is no corresponding java.sql.Types type, but if your application calls the formOfUse(NCHAR) method, then this type can be accessed.

The following table describes Oracle JDBC support for the ANSI-supported SQL data types.

Table A-3 Support for ANSI-92 SQL Data Types

ANSI-Supported SQL Data Type	Supported by JDBC Drivers?
CHARACTER	yes
DEC	yes
DECIMAL	yes

Table A-3 (Cont.) Support for ANSI-92 SQL Data Types

ANSI-Supported SQL Data Type	Supported by JDBC Drivers?
DOUBLE PRECISION	yes
FLOAT	yes
INT	yes
INTEGER	yes
NATIONAL CHARACTER	no
NATIONAL CHARACTER VARYING	no
NATIONAL CHAR	yes
NATIONAL CHAR VARYING	no
NCHAR	yes
NCHAR VARYING	no
NUMERIC	yes
REAL	yes
SMALLINT	yes
VARCHAR	yes

The following table describes Oracle JDBC driver support for SQL User-Defined types.

Table A-4 Support for SQL User-Defined Types

SQL User-Defined type	Supported by JDBC Drivers?
OPAQUE	yes
Reference types	yes
Object types (`JAVA_OBJECT`)	yes
Nested table types and VARRAY types	yes

The following table describes Oracle JDBC driver support for PL/SQL data types. The PL/SQL data types include the following categories:

- Scalar types
- Scalar character types, which includes `DATE` data type
- Composite types
- Reference types
- Large object (LOB) types

Table A-5 Support for PL/SQL Data Types

PL/SQL Data Type	Supported by JDBC Drivers?
Scalar Types:	
BINARY INTEGER	yes
DEC	yes
DECIMAL	yes

Table A-5 (Cont.) Support for PL/SQL Data Types

PL/SQL Data Type	Supported by JDBC Drivers?
DOUBLE PRECISION	yes
FLOAT	yes
INT	yes
INTEGER	yes
NATURAL	yes
NATURAL*n*	no
NUMBER	yes
NUMERIC	yes
PLS_INTEGER	yes
POSITIVE	yes
POSITIVE*n*	no
REAL	yes
SIGNTYPE	yes
SMALLINT	yes
BOOLEAN	yes
Scalar Character Types:	
CHAR	yes
CHARACTER	yes
LONG	yes
LONG RAW	yes
NCHAR	no (see Note)
NVARCHAR2	no (see Note)
RAW	yes
ROWID	yes
STRING	yes
UROWID	no
VARCHAR	yes
VARCHAR2	yes
DATE	yes
Composite Types:	
RECORD	no
TABLE	no
VARRAY	yes
Reference Types:	
REF CURSOR types	yes
object reference types	yes
LOB Types:	
BFILE	yes
BLOB	yes

Table A-5 (Cont.) Support for PL/SQL Data Types

PL/SQL Data Type	Supported by JDBC Drivers?
CLOB	yes
NCLOB	yes

Note:

- The types `NATURAL`, `NATURAL`*n*, `POSITIVE`, `POSITIVE`*n*, and `SIGNTYPE` are subtypes of `BINARY INTEGER`.
- The types `DEC`, `DECIMAL`, `DOUBLE PRECISION`, `FLOAT`, `INT`, `INTEGER`, `NUMERIC`, `REAL`, and `SMALLINT` are subtypes of `NUMBER`.
- The types `NCHAR` and `NVARCHAR2` are supported indirectly. There is no corresponding `java.sql.Types` type, but if your application calls `formOfUse(NCHAR)`, then these types can be accessed.

Related Topics

- NCHAR, NVARCHAR2, NCLOB and the defaultNChar Property

A.3 About Using PL/SQL Types

Starting from Oracle Database 12*c* Release 1 (12.1), you can map schema-level PL/SQL types as generic `java.sql.Struct` type and PL/SQL collections as `java.sql.Array` types. So, instead of creating schema-level types that are mapped to PL/SQL package types for binding, you can describe and bind PL/SQL types using only the JDBC APIs.

For example, you can call the `Connection.createStruct(type_name)` method to first create a descriptor that can be used to describe a PL/SQL type and then to create a new `STRUCT` representation of this type on the client. In Oracle Database 12*c* Release 1 (12.1), you can reuse this API by specifying `type_name` as `"schema.package.typename"` or `"package.typename"`.

All PL/SQL package types are mapped to a system-wide unique name that can be used by JDBC to retrieve the server-side type metadata. The name is in the following form:

```
[SCHEMA.]<PACKAGE>.<TYPE>
```

Note:

If the schema is the same as the package name, and if there is a type with the same name as the PL/SQL type, then it will not be able to identify an object with the two part name format, that is, <package>.<type>. In such cases, you must use three part names <schema>.<package>.<type>.

The following code snippet explains how to bind types declared in PL/SQL packages:

```
/*
---------------------------
# Perform the following SQL operations prior to running this sample
---------------------------
conn HR/<password>;
create or replace package TEST_PKG is
   type V_TYP is varray(10) of varchar2(200);
   type R_TYP is record(c1 pls_integer, c2 varchar2(100));
   procedure VARR_PROC(p1 in V_TYP, p2 OUT V_TYP);
   procedure REC_PROC(p1 in R_TYP, p2 OUT R_TYP);
end;
/
create or replace package body TEST_PKG is
 procedure VARR_PROC(p1 in V_TYP, p2 OUT V_TYP) is
  begin
    p2 := p1;
  end;
  procedure REC_PROC(p1 in R_TYP, p2 OUT R_TYP) is
  begin
    p2 := p1;
  end;
end;
/
*/

import java.sql.Array;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Struct;
import java.sql.Types;

import oracle.jdbc.OracleConnection;
public class PLSQLTypesSample
{
  public static void main(String[] args) throws SQLException
  {
    System.out.println("begin...");
    Connection conn = null;
    oracle.jdbc.pool.OracleDataSource ods = new oracle.jdbc.pool.OracleDataSource();
    ods.setURL("jdbc:oracle:oci:localhost:5521:orcl");
    ods.setUser("HR");
    ods.setPassword("hr");
    //get connection
    conn = ods.getConnection();

    //call procedure TEST_PKG.VARR_PROC
    CallableStatement cstmt = null;
    try {
      cstmt = conn.prepareCall("{ call TEST_PKG.VARR_PROC(?,?) }");
      //PLSQL VARRAY type binding
      Array arr = ((OracleConnection)conn).createArray("TEST_PKG.V_TYP", new String[]
{"A", "B"});
      cstmt.setArray(1, arr);
      cstmt.registerOutParameter(2, Types.ARRAY, "TEST_PKG.V_TYP");
      cstmt.execute();
      //get PLSQL VARRAY type out parameter value
      Array outArr = cstmt.getArray(2);
      //...
    }
    catch( Exception e) {
      e.printStackTrace();
```

```
    }finally {
      if (cstmt != null)
        cstmt.close();
    }

    //call procedure TEST_PKG.REC_PROC
    try {
      cstmt = conn.prepareCall("{ call TEST_PKG.REC_PROC(?,?) }");
      //PLSQL RECORD type binding
      Struct struct = conn.createStruct("TEST_PKG.R_TYP", new Object[]{12345, "B"});
      cstmt.setObject(1, struct);
      cstmt.registerOutParameter(2, Types.STRUCT, "TEST_PKG.R_TYP");
      cstmt.execute();
      //get PLSQL RECORD type out parameter value
      Struct outStruct = (Struct)cstmt.getObject(2);
      //...
    }
    catch( Exception e) {
      e.printStackTrace();
    }finally {
      if (cstmt != null)
        cstmt.close();
    }

    if (conn != null)
      conn.close();

    System.out.println("done!");
  }
}
```

Creating Java level objects for each row using %ROWTYPE Attribute

You can create Java-level objects using the `%ROWTYPE` attribute. In this case, each row of the table is created as a `java.sql.Struct` object. For example, if you have a package `pack1` with the following specification:

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about the `%ROWTYPE` attribute

```
CREATE OR REPLACE PACKAGE PACK1 AS
  TYPE EMPLOYEE_ROWTYPE_ARRAY IS TABLE OF EMPLOYEES%ROWTYPE;
END PACK1;
/
```

The following code snippet shows how you can retrieve the value of the `EMPLOYEE_ROWTYPE_ARRAY` array using JDBC APIs:

This example returns a `java.sql.Array` of `java.sql.Struct` objects, where every `Struct` element represents one row of the `EMPLOYEES` table.

Example A-1 Creating Struct Objects for Database Table Rows

```
CallableStatement cstmt = conn.prepareCall("BEGIN SELECT * BULK COLLECT INTO :1 FROM
EMPLOYEE; END;");
cstmt.registerOutParameter(1,OracleTypes.ARRAY, "PACK1.EMPLOYEE_ROWTYPE_ARRAY");
```

```
cstmt.execute();
Array a = cstmt.getArray(1);
```

A.4 Using Embedded JDBC Escape Syntax

Oracle JDBC drivers support some embedded JDBC escape syntax, which is the syntax that you specify between curly braces. The current support is basic.

> **Note:**
>
> JDBC escape syntax was previously known as SQL92 Syntax or SQL92 escape syntax.

This section describes the support offered by the drivers for the following constructs:

- Time and Date Literals
- Scalar Functions
- LIKE Escape Characters
- MATCH_RECOGNIZE Clause
- Outer Joins
- Function Call Syntax

Where driver support is limited, these sections also describe possible workarounds.

Disabling Escape Processing

The processing for JDBC escape syntax is enabled by default, which results in the JDBC driver performing escape substitution before sending the SQL code to the database. If you want the driver to use regular Oracle SQL syntax, which is more efficient than JDBC escape syntax processing, then use this statement:

```
stmt.setEscapeProcessing(false);
```

A.4.1 Time and Date Literals

Databases differ in the syntax they use for date, time, and timestamp literals. JDBC supports dates and times written only in a specific format. This section describes the formats you must use for date, time, and timestamp literals in SQL statements.

A.4.1.1 Date Literals

The JDBC drivers support date literals in SQL statements written in the format:

```
{d 'yyyy-mm-dd'}
```

Where `yyyy-mm-dd` represents the year, month, and day. For example:

```
{d '1995-10-22'}
```

The JDBC drivers will replace this escape clause with the equivalent Oracle representation: "22 OCT 1995".

The following code snippet contains an example of using a date literal in a SQL statement.

```
// Connect to the database
// You can put a database name after the @ sign in the connection URL.
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:oci:@");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();

// Create a Statement
Statement stmt = conn.createStatement ();

// Select the first name column from the employees table where the hire date is
Jan-23-1982
ResultSet rset = stmt.executeQuery
                 ("SELECT first_name FROM employees WHERE hire_date = {d '1982-01-23'}");

// Iterate through the result and print the employee names
while (rset.next ())
   System.out.println (rset.getString (1));
```

A.4.1.2 Time Literals

The JDBC drivers support time literals in SQL statements written in the format:

```
{t 'hh:mm:ss'}
```

where, `hh:mm:ss` represents the hours, minutes, and seconds. For example:

```
{t '05:10:45'}
```

The JDBC drivers will replace this escape clause with the equivalent Oracle representation: "05:10:45".

If the time is specified as:

```
{t '14:20:50'}
```

Then the equivalent Oracle representation would be "14:20:50", assuming the server is using a 24-hour clock.

This code snippet contains an example of using a time literal in a SQL statement.

```
ResultSet rset = stmt.executeQuery
                 ("SELECT first_name FROM employees WHERE hire_date = {t '12:00:00'}");
```

A.4.1.3 Timestamp Literals

The JDBC drivers support timestamp literals in SQL statements written in the format:

```
{ts 'yyyy-mm-dd hh:mm:ss.f...'}
```

where `yyyy-mm-dd hh:mm:ss.f...` represents the year, month, day, hours, minutes, and seconds. The fractional seconds portion (`.f...`) is optional and can be omitted. For example: `{ts '1997-11-01 13:22:45'}` represents, in Oracle format, NOV 01 1997 13:22:45.

This code snippet contains an example of using a timestamp literal in a SQL statement.

```
ResultSet rset = stmt.executeQuery
    ("SELECT first_name FROM employees WHERE hire_date = {ts '1982-01-23 12:00:00'}");
```

Mapping SQL DATE Data type to Java

Oracle Database 8*i* and earlier versions did not support `TIMESTAMP` data, but Oracle `DATE` data used to have a time component as an extension to the SQL standard. So, Oracle Database 8*i* and earlier versions of JDBC drivers mapped `oracle.sql.DATE` to `java.sql.Timestamp` to preserve the time component. Starting with Oracle Database 9.0.1, `TIMESTAMP` support was included and 9*i* JDBC drivers started mapping `oracle.sql.DATE` to `java.sql.Date`. This mapping was incorrect as it truncated the time component of Oracle `DATE` data. To overcome this problem, Oracle Database 11*g* Release 1 introduced a new flag `mapDateToTimestamp`. The default value of this flag is `true`, which means that by default the drivers will correctly map `oracle.sql.DATE` to `java.sql.Timestamp`, retaining the time information. If you still want the incorrect but 10*g* compatible `oracle.sql.DATE` to `java.sql.Date` mapping, then you can get it by setting the value of `mapDateToTimestamp` flag to `false`.

Note:

- Since Oracle Database 11*g*, if you have an index on a DATE column to be used by a SQL query, then to obtain faster and accurate results, you must use the `setObject` method in the following way:

  ```
  Date d = parseIsoDate(val);
  Timestamp t = new Timestamp(d.getTime());
  stmt.setObject(pos, new oracle.sql.DATE(t, (Calendar)UTC_CAL.clone()));
  ```

  This is because if you use the `setDate` method, then the time component of the Oracle DATE data will be lost and if you use the `setTimestamp` method, then the index on the DATE column will not be used.
- To overcome the problem of `oracle.sql.DATE` to `java.sql.Date` mapping, Oracle Database 9.2 introduced a flag, `V8Compatible`. The default value of this flag was `false`, which allowed the mapping of Oracle `DATE` data to `java.sql.Date` data. But, users could retain the time component of the Oracle `DATE` data by setting the value of this flag to `true`. This flag is desupported since 11*g* because it controlled Oracle Database 8*i* compatibility, which is no longer supported.

A.4.2 Scalar Functions

Oracle JDBC drivers do not support all scalar functions. To find out which functions the drivers support, use the following methods supported by the Oracle-specific `oracle.jdbc.OracleDatabaseMetaData` class and the standard Java `java.sql.DatabaseMetadata` interface:

- `getNumericFunctions()`

  Returns a comma-delimited list of math functions supported by the driver. For example, `ABS`, `COS`, `SQRT`.
- `getStringFunctions()`

  Returns a comma-delimited list of string functions supported by the driver. For example, `ASCII`, `LOCATE`.
- `getSystemFunctions()`

Returns a comma-delimited list of system functions supported by the driver. For example, `DATABASE`, `USER`.

- `getTimeDateFunctions()`

  Returns a comma-delimited list of time and date functions supported by the driver. For example, `CURDATE`, `DAYOFYEAR`, `HOUR`.

> **Note:**
>
> Oracle JDBC drivers support `fn`, the function keyword.

A.4.3 LIKE Escape Characters

The characters `%` and _ have special meaning in SQL `LIKE` clauses. You use `%` to match zero or more characters and _ to match exactly one character. If you want to interpret these characters literally in strings, then you precede them with a special escape character. For example, if you want to use ampersand (`&`) as the escape character, then you identify it in the SQL statement as:

```
Statement stmt = conn.createStatement ();

// Select the empno column from the emp table where the ename starts with '_'
ResultSet rset = stmt.executeQuery
          ("SELECT empno FROM emp WHERE ename LIKE '&_%' {ESCAPE '&'}");

// Iterate through the result and print the employee numbers
while (rset.next ())
   System.out.println (rset.getString (1));
```

> **Note:**
>
> If you want to use the backslash character (\) as an escape character, then you must enter it twice, that is, \\. For example:
>
> ```
> ResultSet rset = stmt.executeQuery("SELECT empno FROM emp
>                 WHERE ename LIKE '_%' {escape '\\'}");
> ```

A.4.4 MATCH_RECOGNIZE Clause

The `?` character is used as a token in `MATCH_RECOGNIZE` clause in Oracle Database 11*g* and later versions. As the JDBC standard defines the `?` character as a parameter marker, the JDBC Driver and the Server SQL Engine cannot distinguish between different uses of the same token.

In earlier versions of JDBC Driver, if you want to interpret the `?` character as a `MATCH_RECOGNIZE` token and not as a parameter marker, then you must use a `Statement` instead of a `PreparedStatement` and disable escape processing. However, starting from Oracle Database 12*c* Release 1 (12.1.0.2), you can use the `'{\\ ... \\}'` syntax while using the `?` character, so that the JDBC driver does not process it as a parameter marker and allows the SQL engine to process it. The following code snippet shows how to use the `'{\\ ... \\}'` syntax:

```
String sql =
  "select T.firstW, T.lastZ, ? " +  // use of parameter marker
  "from tkpattern_S11 " +
  "MATCH_RECOGNIZE ( " +
  "    MEASURES A.c1 as firstW, last(Z.c1) as lastZ " +
  "    ALL MATCHES " +
  "    {\\ PATTERN(A? X*? Y+? Z??)\\} " +  // use of escape sequence
  "    DEFINE " +
  "        X as X.c2 > prev(X.c2), " +
  "        Y as Y.c2 < prev(Y.c2), " +
  "        Z as Z.c2 > prev(Z.c2)" +
  ") as T";
PreparedStatement ps = conn.prepareStatatement(sql);
ps.setString(1, "test");
ResultSet rs = ps.executeQuery();
```

Related Topics

- Using Embedded JDBC Escape Syntax

A.4.5 Outer Joins

Oracle JDBC drivers do not support the outer join syntax. The workaround is to use Oracle outer join syntax:

Instead of:

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
     ("SELECT ename, dname
       FROM {OJ dept LEFT OUTER JOIN emp ON dept.deptno = emp.deptno}
       ORDER BY ename");
```

Use Oracle SQL syntax:

```
Statement stmt = conn.createStatement ();
ResultSet rset = stmt.executeQuery
     ("SELECT ename, dname
       FROM emp b, dept a WHERE a.deptno = b.deptno(+)
       ORDER BY ename");
```

A.4.6 Function Call Syntax

Oracle JDBC drivers support the following procedure and function call syntax:

Procedure calls:

```
{ call procedure_name (argument1, argument2,...) }
```

Function calls:

```
{ ? = call procedure_name (argument1, argument2,...) }
```

A.4.7 JDBC Escape Syntax to Oracle SQL Syntax Example

You can write a simple program to translate JDBC escape syntax to Oracle SQL syntax. The following program prints the comparable Oracle SQL syntax for statements using JDBC escape syntax for function calls, date literals, time literals, and timestamp literals. In the program, the `oracle.jdbc.OracleSql` class `parse()` method performs the conversions.

```
public class Foo
{
   static oracle.jdbc.OracleDriver driver = new oracle.jdbc.OracleDriver();
   public static void main (String args[]) throws Exception
   {
      show ("{call foo(?, ?)}");
      show ("{? = call bar (?, ?)}");
      show ("{d '1998-10-22'}");
      show ("{t '16:22:34'}");
      show ("{ts '1998-10-22 16:22:34'}");
   }

   public static void show (String s) throws Exception
   {
      System.out.println (s + " => " +
         driver.processSqlEscapes(s));
   }
}
```

The following code is the output that prints the comparable SQL syntax.

```
{call foo(?, ?)} => BEGIN foo(:1, :2); END;
{? = call bar (?, ?)} => BEGIN :1 := bar (:2, :3); END;
{d '1998-10-22'} => TO_DATE ('1998-10-22', 'YYYY-MM-DD')
{t '16:22:34'} => TO_DATE ('16:22:34', 'HH24:MI:SS')
{ts '1998-10-22 16:22:34'} => TO_TIMESTAMP ('1998-10-22 16:22:34', 'YYYY-MM-DD
HH24:MI:SS.FF')
```

A.5 Oracle JDBC Notes and Limitations

The following limitations exist in the Oracle JDBC implementation, but all of them are either insignificant or have easy workarounds. This section covers the following topics:

- CursorName
- JDBC Outer Join Escapes
- IEEE 754 Floating Point Compliance
- Catalog Arguments to DatabaseMetaData Calls
- SQLWarning Class
- Executing DDL Statements
- Binding Named Parameters

A.5.1 CursorName

Oracle JDBC drivers do not support the `getCursorName` and `setCursorName` methods, because there is no convenient way to map them to Oracle constructs. Oracle recommends using `ROWID` instead.

Related Topics

- Oracle ROWID Type

A.5.2 JDBC Outer Join Escapes

Oracle JDBC drivers do not support JDBC outer join escapes. Use Oracle SQL syntax with + instead.

Related Topics

- Using Embedded JDBC Escape Syntax

A.5.3 IEEE 754 Floating Point Compliance

The arithmetic for the Oracle `NUMBER` type does not comply with the IEEE 754 standard for floating-point arithmetic. Therefore, there can be small disagreements between the results of computations performed by Oracle and the same computations performed by Java.

Oracle stores numbers in a format compatible with decimal arithmetic and guarantees 38 decimal digits of precision. It represents zero, minus infinity, and plus infinity exactly. For each positive number it represents, it represents a negative number of the same absolute value.

It represents every positive number between 10^{-30} and $(1 – 10^{-38}) * 10^{126}$ to full 38-digit precision.

A.5.4 Catalog Arguments to DatabaseMetaData Calls

Certain `DatabaseMetaData` methods define a `catalog` parameter. This parameter is one of the selection criteria for the method. Oracle does not have multiple catalogs, but it does have packages.

Related Topics

- About Reporting DatabaseMetaData TABLE_REMARKS

A.5.5 SQLWarning Class

The `java.sql.SQLWarning` class provides information about a database access warning. Warnings typically contain a description of the warning and a code that identifies the warning. Warnings are silently chained to the object whose method caused it to be reported. Oracle JDBC drivers generally do not support `SQLWarning`. As an exception to this, scrollable result set operations do generate SQL warnings, but the `SQLWarning` instance is created on the client, not in the database.

Related Topics

- About Processing SQL Exceptions

A.5.6 Executing DDL Statements

You must execute Data Definition Language (DDL) statements with `Statement` objects. If you use `PreparedStatements` objects or `CallableStatements` objects, then the DDL statement takes effect only on the first execution. This can cause unexpected behavior if the SQL statements are in a statement cache.

A.5.7 Binding Named Parameters

Binding by name is not supported when using the `set`*`XXX`* methods. Under certain circumstances, previous versions of Oracle JDBC drivers have allowed binding statement variables by name when using the `set`*`XXX`* methods. In the following statement, the named variable `EmpId` would be bound to the integer `314159`.

```
PreparedStatement p = conn.prepareStatement
  ("SELECT name FROM emp WHERE id = :EmpId");
  p.setInt(1, 314159);
```

This capability to bind by name using the `setXXX` methods is not part of the JDBC specification, and Oracle does not support it. The JDBC drivers can throw a `SQLException` or produce unexpected results. Starting from Oracle Database 10*g* JDBC drivers, bind by name is supported using the `setXXXAtName` methods.

The bound values are not copied by the drivers until you call the `execute` method. So, changing the bound value before calling the `execute` method could change the bound value. For example, consider the following code snippet:

```
PreparedStatement p;
.......
Date d = new Date(1181676033917L);
p.setDate(1, d);
d.setTime(0);
p.executeUpdate();
```

This code snippet inserts `Date(0)` in the database instead of `Date(1181676033917L)` because the bound values are not copied by JDBC driver implementation for performance reasons.

Related Topics

- Interface oracle.jdbc.OracleCallableStatement
- Interface oracle.jdbc.OraclePreparedStatement