23
Performance Extensions

This chapter describes the Oracle performance extensions to the Java Database Connectivity (JDBC) standard.

This chapter covers the following topics:

- Update Batching
- Additional Oracle Performance Extensions

> **Note:**
>
> Oracle update batching was deprecated in Oracle Database 12*c* Release 1 (12.1). Starting with Oracle Database 12*c* Release 2 (12.2), Oracle update batching is a no operation code (no-op). This means that if you implement Oracle update batching in your application, using the current release of Oracle JDBC driver, then the specified batch size is not set, and it results in a batch size of 1. With this batch setting, your application processes one row at a time. Oracle strongly recommends that you use the standard JDBC batching.

23.1 Update Batching

This section covers the following topics:

- Overview of Update Batching
- Standard Update Batching
- Premature Batch Flush

23.1.1 Overview of Update Batching

Update batching is especially useful with prepared statements, when you are repeating the same statement with different bind variables.

You can reduce the number of round-trips to the database and improve the application performance, by grouping multiple `UPDATE`, `DELETE`, or `INSERT` statements into a single batch, and then having the whole batch sent to the database and processed in one trip. This is referred to as 'update batching'.

> **Note:**
>
> - The JDBC 2.0 specification refers to 'update batching' as 'batch updates'.
> - To adhere to the JDBC 2.0 standard, Oracle implementation of standard update batching supports callable statements without `OUT` parameters, generic statements, and prepared statements. You can migrate standard update batching into an Oracle JDBC application without difficulty. However, the Oracle implementation of standard update batching does not implement true batching for generic statements and callable statements and you will see performance improvement for only `PreparedStatement` objects.

23.1.2 Standard Update Batching

JDBC standard update batching depends on explicitly adding statements to the batch using an `addBatch` method and explicitly processing the batch using an `executeBatch` method.

> **Note:**
>
> Disable auto-commit mode when you use update batching. In case an error occurs while you are processing a batch, this provides you the option of committing or rolling back the operations that ran successfully prior to the error.

23.1.2.1 About Adding Operations to the Batch

When any statement object is first created, its statement batch is empty. Use the standard `addBatch` method to add an operation to the statement batch. This method is specified in the standard `java.sql.Statement`, `PreparedStatement`, and `CallableStatement` interfaces, which are implemented by the `oracle.jdbc.OracleStatement`, `OraclePreparedStatement`, and `OracleCallableStatement` interfaces, respectively.

> **Note:**
>
> Starting from Oracle Databse Release 23ai, when you use the `addBatch` method, the JDBC driver starts a pipeline with the Database, which results in improved response time and throughput. See Support for Pipelined Database Operations.

For a `Statement` object, the `addBatch` method takes a Java `String` with a SQL operation as input. For example:

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

stmt.addBatch("INSERT INTO emp VALUES(1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO dept VALUES(260, 'Sales')");
stmt.addBatch("INSERT INTO emp_dept VALUES(1000, 260)");
...
```

At this point, three operations are in the batch.

For prepared statements, update batching is used to batch multiple runs of the same statement with different sets of bind parameters. For a `PreparedStatement` or `OraclePreparedStatement` object, the `addBatch` method takes no input. It simply adds the operation to the batch using the bind parameters last set by the appropriate `setXXX` methods. This is also true for `CallableStatement` or `OracleCallableStatement` objects, but remember that in the Oracle implementation of standard update batching, you will probably see no performance improvement in batching callable statements.

For example:

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();
...
```

At this point, two operations are in the batch.

Because a batch is associated with a single prepared statement object, you can batch only repeated runs of a single prepared statement, as in this example.

23.1.2.2 About Processing the Batch

To process the current batch of operations, use the `executeBatch` method of the statement object. This method is specified in the standard `Statement` interface, which is extended by the standard `PreparedStatement` and `CallableStatement` interfaces.

Following is an example that repeats the prepared statement `addBatch` calls shown previously and then processes the batch:

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();

int[] updateCounts = pstmt.executeBatch();
...
```

If you add too many operations to a batch by calling the `addBatch` method several times and create a very large batch (for example, with more than or equal to 100,000 rows), then while calling the `executeBatch` method on the whole batch, you may face severe performance problems in terms of memory. To avoid this issue, the earlier JDBC drivers used to transparently break up the large batches into smaller internal batches to make a round-trip to the server for each internal batch. This made your application slightly slower because of each round-trip overhead, but optimized memory significantly. However, if each bound row was very large in size (for example, more than about 1MB each or so), then this process could impact

the overall performance negatively because, in such a case, the performance gained in terms of memory was less than the performance lost in terms of time.

Starting from Oracle Database 23ai Release, the batches are no longer split, unless you configure the `CONNECTION_PROPERTY_MAX_BATCH_MEMORY` property. This property configures the maximum amount of memory allocated by the `executeBatch` and `executeLargeBatch` methods, when you convert Java bind values to SQL data types. The value of this property is measured in bytes. The default value is 0, which means that there is no limit on the maximum amount of memory allocated.

> **See Also:**
>
> Oracle JDBC Java API Reference

23.1.2.3 Row Count per Iteration for Array DMLs

Starting from Oracle Database 12*c* Release 1 (12.1), the `executeBatch` method has been improved so that it returns an int array of size that is the same as the number of records in the batch and each item in the return array is the number of database table rows affected by the corresponding record of the batch. For example, if the batch size is 5, then the `executeBatch` method returns an array of size 5. In case of an error in between execution of the batch, the `executeBatch` method cannot return a value, instead it throws a `BatchUpdateException`. In this case, the exception itself carries an int array of size `n` as its data, where `n` is the number of successful record executions. For example, if the batch is of size 5 and the error occurs at the 4th record, then the `BatchUpdateException` has an array of size 3 (3 records executed successfully) and each item in the array represents how many rows were affected by each of them.

23.1.2.4 About Committing the Changes in the Oracle Implementation of Standard Batching

After you process the batch, you must still commit the changes, presuming auto-commit is disabled as recommended.

Calling `commit`, commits nonbatched operations and batched operations for statement batches that have been processed, but for the Oracle implementation of standard batching, has no effect on pending statement batches that have *not* been processed.

23.1.2.5 About Clearing the Batch

To clear the current batch of operations instead of processing it, use the `clearBatch` method of the statement object. This method is specified in the standard `Statement` interface, which is extended by the standard `PreparedStatement` and `CallableStatement` interfaces.

Keep the following things in mind:

- When a batch is processed, operations are performed in the order in which they were batched.
- After calling `addBatch`, you must call either `executeBatch` or `clearBatch` before a call to `executeUpdate`, otherwise there will be a SQL exception.
- A `clearBatch` or `executeBatch` call resets the statement batch to empty.

- The statement batch is not reset to empty if the connection receives a `ROLLBACK` request. You must explicitly call `clearBatch` to reset it.
- Invoking `clearBatch` method after a rollback works for all releases.
- An `executeBatch` call closes the current result set of the statement object, if one exists.
- Nothing is returned by the `clearBatch` method.

Following is an example that repeats the prepared statement `addBatch` calls shown previously but then clears the batch under certain circumstances:

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();

if (...condition...)
{
   int[] updateCounts = pstmt.executeBatch();
   ...
}
else
{
   pstmt.clearBatch();
   ...
}
```

23.1.2.6 Update Counts in the Oracle Implementation of Standard Batching

If a statement batch is processed successfully, then the integer array, or update counts array, returned by the statement `executeBatch` call will always have one element for each operation in the batch. In the Oracle implementation of standard update batching, the values of the array elements are as follows:

- For a prepared statement batch, the array contains the actual update counts indicating the number of rows affected by each operation.
- For a generic statement batch, the array contains the actual update counts indicating the number of rows affected by each operation. The actual update counts can be provided only in the case of generic statements in the Oracle implementation of standard batching.
- For a callable statement batch, the array contains the actual update counts indicating the number of rows affected by each operation.

In your code, upon successful processing of a batch, you should be prepared to handle either `-2`, `1`, or true update counts in the array elements. For a successful batch processing, the array contains either all `-2`, `1`, or all positive integers.

Example 23-1 illustrates the use of standard update batching.

Example 23-1 Standard Update Batching

This example combines the sample fragments in the previous sections, accomplishing the following steps:

1. Disabling auto-commit mode, which you should always perform when using update batching
2. Creating a prepared statement object
3. Adding operations to the batch associated with the prepared statement object
4. Processing the batch
5. Committing the operations from the batch

```
conn.setAutoCommit(false);

PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");
pstmt.addBatch();

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();

int[] updateCounts = pstmt.executeBatch();

conn.commit();

pstmt.close();
...
```

You can process the update counts array to determine if the batch processed successfully.

23.1.2.7 Error Handling in the Oracle Implementation of Standard Batching

If any one of the batched operations fails to complete successfully or attempts to return a result set during an `executeBatch` call, then the processing stops and a `java.sql.BatchUpdateException` is generated.

After a batch exception, the update counts array can be retrieved using the `getUpdateCounts` method of the `BatchUpdateException` object. This returns an `int` array of update counts, just as the `executeBatch` method does. In the Oracle implementation of standard update batching, contents of the update counts array are as follows, after a batch is processed:

- For a prepared statement batch, in case of an error in between execution of the batch, the `executeBatch` method cannot return a value, instead it throws a `BatchUpdateException`. In this case, the exception itself carries an int array of size `n` as its data, where `n` is the number of successful record executions. For example, if the batch is of size 5 and the error occurs at the 4th record, then the `BatchUpdateException` has an array of size 3 (3 records executed successfully) and each item in the array represents how many rows were affected by each of them.
- For a generic statement batch or callable statement batch, the update counts array is only a partial array containing the actual update counts up to the point of the error. The actual update counts can be provided because Oracle JDBC cannot use true batching for generic and callable statements in the Oracle implementation of standard update batching.

  For example, if there were 20 operations in the batch, the first 13 succeeded, and the 14th generated an exception, then the update counts array will have 13 elements, containing actual update counts of the successful operations.

  You can either commit or roll back the successful operations in this situation, as you prefer.

In your code, upon failed processing of a batch, you should be prepared to handle either `-3` or true update counts in the array elements when an exception occurs. For a failed batch processing, you will have either a full array of `-3` or a partial array of positive integers.

23.1.2.8 About Intermixing Batched Statements and Nonbatched Statements

You cannot call `executeUpdate` for regular, nonbatched processing of an operation if the statement object has a pending batch of operations.

However, you can intermix batched operations and nonbatched operations in a single statement object if you process nonbatched operations either prior to adding any operations to the statement batch or after processing the batch. Essentially, you can call `executeUpdate` for a statement object only when its update batch is empty. If the batch is non-empty, then an exception will be generated.

For example, it is valid to have a sequence, such as the following:

```
...
PreparedStatement pstmt =
          conn.prepareStatement("INSERT INTO employees VALUES(?, ?)");

pstmt.setInt(1, 2000);
pstmt.setString(2, "Milo Mumford");

int scount = pstmt.executeUpdate();   // OK; no operations in pstmt batch

pstmt.setInt(1, 3000);
pstmt.setString(2, "Sulu Simpson");
pstmt.addBatch();                    // Now start a batch

pstmt.setInt(1, 4000);
pstmt.setString(2, "Stan Leland");
pstmt.addBatch();

int[] bcounts = pstmt.executeBatch();

pstmt.setInt(1, 5000);
pstmt.setString(2, "Amy Feiner");

int scount = pstmt.executeUpdate();   // OK; pstmt batch was executed
...
```

Intermixing nonbatched operations on one statement object and batched operations on another statement object within your code is permissible. Different statement objects are independent of each other with regard to update batching operations. A `COMMIT` request will affect all nonbatched operations and all successful operations in processed batches, but will not affect any pending batches.

23.1.2.9 Limitations in the Oracle Implementation of Standard Batching

This section discusses the limitations and implementation details regarding the Oracle implementation of standard update batching.

In Oracle JDBC applications, update batching is intended for use with prepared statements that are being processed repeatedly with different sets of bind values.

The Oracle implementation of standard update batching does not implement true batching for generic statements and callable statements. Even though Oracle JDBC supports the use of

standard batching for `Statement` and `CallableStatement` objects, you are unlikely to see performance improvement.

23.1.3 Premature Batch Flush

Premature batch flush happens due to a change in cached metadata. Cached metadata can be changed due to various reasons, such as the following:

- The initial bind was null and the following bind is not null.
- A scalar type is initially bound as string and then bound as scalar type or the reverse.

The premature batch flush count is summed to the return value of the next `executeUpdate` or `sendBatch` method.

The old functionality lost all these batch flush values which can be obtained now. To switch back to the old functionality, you can set the `AccumulateBatchResult` property to `false`, as follows:

```
java.util.Properties info = new java.util.Properties();
info.setProperty("user", "HR");
info.setProperty("passwd", "hr");
// other properties
...

// property: batch flush type
info.setProperty("AccumulateBatchResult", "false");

OracleDataSource ods = new OracleDataSource();
ods.setConnectionProperties(info);
ods.setURL("jdbc:oracle:oci:@");
Connection conn = ods.getConnection();
```

> **Note:**
>
> The `AccumulateBatchResult` property is set to `true` by default.

Example 23-2 illustrates premature batch flushing.

Example 23-2 Premature Batch Flushing

```
((OraclePreparedStatement)pstmt).setExecuteBatch (2);

pstmt.setNull(1, OracleTypes.NUMBER);
pstmt.setString(2, "test11");
int count = pstmt.executeUpdate(); // returns 0

/*
* Premature batch flush happens here.
*/
pstmt.setInt(1, 22);
pstmt.setString(2, "test22");
int count = pstmt.executeUpdate(); // returns 0

pstmt.setInt(1, 33);
pstmt.setString(2, "test33");
/*
*  returns 3 with the new batching scheme where as,
```

```
*  returns 2 with the old batching scheme.
*/
int count = pstmt.executeUpdate();
```

23.2 Additional Oracle Performance Extensions

In addition to update batching, Oracle JDBC drivers support the extensions discussed in this chpter, which improve performance by reducing round-trips to the database:

- Prefetching rows

  This reduces round-trips to the database by fetching multiple rows of data each time data is fetched. The extra data is stored in client-side buffers for later access by the client. The number of rows to prefetch can be set as desired.

- Specifying column types

  This avoids an inefficiency in the standard JDBC protocol for performing and returning the results of queries.

- Suppressing database metadata `TABLE_REMARKS` columns

  This avoids an expensive outer join operation.

Oracle provides several extensions to connection properties objects to support these performance extensions. These extensions enable you to set the `remarksReporting` flag and default values for row prefetching and update batching.

This section covers the following topics:

- Oracle Row-Prefetching Limitations
- About Defining Column Types
- About Reporting DatabaseMetaData TABLE_REMARKS

23.2.1 Oracle Row-Prefetching Limitations

There is no maximum prefetch setting. The default value is 10. Larger or smaller values may be appropriate depending on the number of rows and columns expected from the query. You can set the default connection row-prefetch value using a `Properties` object.

When a statement object is created, it receives the default row-prefetch setting from the associated connection. Subsequent changes to the default connection row-prefetch setting will have no effect on the statement row-prefetch setting.

If a column of a result set is of data type `LONG`, `LONG RAW` or `LOB`s returned through the data interface, that is, the streaming types, then JDBC changes the statement row-prefetch setting to 1, even if you never actually read a value of either of these types.

Setting the prefetch size can affect the performance of an application. Increasing the prefetch size will reduce the number of round-trips required to get all the data, but will increase memory usage. This will depend on the number and size of the columns in the query and the number of rows expected to be returned. It will also depend on the memory and CPU loading of the JDBC client machine. The optimum for a standalone client application will be different from a heavily loaded application server. The speed and latency of the network connection should also be considered.

> **Note:**
>
> Starting from Oracle Database 11*g* Release 1, the Thin driver can fetch the first `prefetch_size` number of rows from the server in the very first round-trip. This saves one round-trip in SELECT statements.

If you are migrating an application from earlier releases of Oracle JDBC drivers to 10*g* Release 1 (10.1) or later releases of Oracle JDBC drivers, then you should revisit the optimizations that you had done earlier, because the memory usage and performance characteristics may have changed substantially.

A common situation that you may encounter is, say, you have a query that selects a unique key. The query will return only zero or one row. Setting the prefetch size to 1 will decrease memory and CPU usage and cannot increase round-trips. However, you must be careful to avoid the error of requesting an extra fetch by writing `while(rs.next())` instead of `if(rs.next())`.

If you are using the JDBC Thin driver, then use the `useFetchSizeWithLongColumn` connection property, because it will perform `PARSE`, `EXECUTE`, and `FETCH` in a single round-trip.

Tuning of the prefetch size should be done along with tuning of memory management in your JVM under realistic loads of the actual application.

> **Note:**
>
> - Do not mix the JDBC 2.0 fetch size application programming interface (API) and the Oracle row-prefetching API in your application. You can use one or the other, but not both.
> - Be aware that setting the Oracle fetch size value can affect not only queries, but also explicitly refetching rows in a result set through the result set `refreshRow` method, which is relevant for scroll-sensitive/read-only, scroll-sensitive/updatable, and scroll-insensitive/updatable result sets, and the window size of a scroll-sensitive result set, affecting how often automatic refetches are performed. However, the Oracle fetch size value will be overridden by any setting of the fetch size.

23.2.2 About Defining Column Types

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), the `defineColumnType` method is deprecated.

The implementation of `defineColumnType` changed significantly since Oracle Database 10*g*. Previously, `defineColumnType` was used both as a performance optimization and to force data type conversion. In previous releases, all of the drivers benefited from calls to `defineColumnType`. Starting from Oracle Database 10*g*, the JDBC Thin driver no longer needs

the information provided. The JDBC Thin driver achieves maximum performance without calls to `defineColumnType`. The JDBC Oracle Call Interface (OCI) and server-side internal drivers still get better performance when the application uses `defineColumnType`.

If your code is used with both the JDBC Thin and OCI drivers, you can disable the `defineColumnType` method when using the Thin driver by setting the connection property `disableDefineColumnType` to `true`. Doing this makes `defineColumnType` have no effect. Do not set this connection property to `true` when using the JDBC OCI or server-side internal drivers.

You can also use `defineColumnType` to control how much memory the client-side allocates or to limit the size of variable-length data.

Follow these general steps to define column types for a query:

1. If necessary, cast your statement object to `OracleStatement`, `OraclePreparedStatement`, or `OracleCallableStatement`, as applicable.
2. If necessary, use the `clearDefines` method of your `Statement` object to clear any previous column definitions for this `Statement` object.
3. On each column, call the `defineColumnType` method of your `Statement` object, passing it these parameters:
   - Column index (integer)
   - Type code (integer)

     Use the `static` constants of the `java.sql.Types` class or `oracle.jdbc.OracleTypes` class, such as `Types.INTEGER`, `Types.FLOAT`, `Types.VARCHAR`, `OracleTypes.VARCHAR`, and `OracleTypes.ROWID`. Type codes for standard types are identical in these two classes.
   - Type name (string)

     For structured objects, object references, and arrays, you must also specify the type name. For example, `Employee`, `EmployeeRef`, or `EmployeeArray`.
   - Maximum field size (integer)

     Optionally specify a maximum data length for this column.

     You cannot specify a maximum field size parameter if you are defining the column type for a structured object, object reference, or array. If you try to include this parameter, it will be ignored.
   - Form of use (short)

     Optionally specify a form of use for the column. This can be `OraclePreparedStatement.FORM_CHAR` to use the database character set or `OraclePreparedStatement.FORM_NCHAR` to use the national character set. If this parameter is omitted, the default is `FORM_CHAR`.

   For example, assuming `stmt` is an Oracle statement, use:

   ```
   stmt.defineColumnType(column_index, typeCode);
   ```

   If the column is `VARCHAR` or equivalent and you know the length limit:

   ```
   stmt.defineColumnType(column_index, typeCode, max_size);
   ```

   For an `NVARCHAR` column where the original maximum length is desired and conversion to the database character set is requested:

```
stmt.defineColumnType(column_index, typeCode, 0,
   OraclePreparedStatement.FORM_CHAR );
```

For structured object, object reference, and array columns:

```
stmt.defineColumnType(column_index, typeCode, typeName);
```

Set a maximum field size if you do not want to receive the full default length of the data. Calling the `setMaxFieldSize` method of the standard JDBC `Statement` class sets a restriction on the amount of data returned. Specifically, the size of the data returned will be the minimum of the following:

- The maximum field size set in `defineColumnType`
- The maximum field size set in `setMaxFieldSize`
- The natural maximum size of the data type

After you complete these steps, use the `executeQuery` method of the statement to perform the query.

> **Note:**
>
> It is no longer necessary to specify a data type for each column of the expected result set.

The following example illustrates the use of this feature. It assumes you have imported the `oracle.jdbc.*` interfaces.

Example 23-3 Defining Column Types

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:@localhost:5221:orcl");
ods.setUser("HR");
ods.setPassword("hr");
Connection conn = ods.getConnection();

Statement stmt = conn.createStatement();
// Allocate only 2 chars for this column (truncation will happen)
((OracleStatement)stmt).defineColumnType(1, Types.VARCHAR, 2);
ResultSet rset = stmt.executeQuery("select ename from emp");
while(rset.next() )
  System.out.println(rset.getString(1));
stmt.close();
```

As this example shows, you must cast the `Statement` object, `stmt`, to `OracleStatement` in the invocation of the `defineColumnType` method. The `createStatement` method of the connection returns an object of type `java.sql.Statement`, which does not have the `defineColumnType` and `clearDefines` methods. These methods are provided only in the `OracleStatement` implementation.

The define-extensions use JDBC types to specify the desired types. The allowed define types for columns depend on the internal Oracle type of the column.

All columns can be defined to their natural JDBC types. In most cases, they can be defined to the `Types.CHAR` or `Types.VARCHAR` type code.

The following table lists the valid column definition arguments that you can use in the `defineColumnType` method.

Table 23-1 Valid Column Type Specifications

If the column has Oracle SQL type:	You can use defineColumnType to define it as:
NUMBER, VARNUM	BIGINT, TINYINT, SMALLINT, INTEGER, FLOAT, REAL, DOUBLE, NUMERIC, DECIMAL, CHAR, VARCHAR
CHAR, VARCHAR2	CHAR, VARCHAR
LONG	CHAR, VARCHAR, LONGVARCHAR
LONGRAW	LONGVARBINARY, VARBINARY, BINARY
RAW	VARBINARY, BINARY
DATE	DATE, TIME, TIMESTAMP, CHAR, VARCHAR
ROWID	ROWID
BLOB	VARBINARY, BINARY
CLOB	LONG, CHAR, VARCHAR

It is always valid to use `defineColumnType` with the original data type of the column.

23.2.3 About Reporting DatabaseMetaData TABLE_REMARKS

The `getColumns`, `getProcedureColumns`, `getProcedures`, and `getTables` methods of the database metadata classes are slow if they must report `TABLE_REMARKS` columns, because this necessitates an expensive outer join. For this reason, the JDBC driver does *not* report `TABLE_REMARKS` columns by default.

You can enable `TABLE_REMARKS` reporting by passing a `true` argument to the `setRemarksReporting` method of an `OracleConnection` object.

Equivalently, instead of calling `setRemarksReporting`, you can set the `remarksReporting` Java property if you use a Java `Properties` object in establishing the connection.

If you are using a standard `java.sql.Connection` object, you must cast it to `OracleConnection` to use `setRemarksReporting`.

The following code snippet illustrates how to enable `TABLE_REMARKS` reporting:

```
((oracle.jdbc.OracleConnection)conn ).setRemarksReporting(true);
```

Here, `conn` is the name of your standard `Connection` object, the following statement enables `TABLE_REMARKS` reporting:

Considerations for getColumns

By default, the `getColumns` method does not retrieve information about the columns if a synonym is specified. To enable the retrieval of information if a synonym is specified, you must call the `setIncludeSynonyms` method on the connection as follows:

```
((oracle.jdbc.OracleConnection)conn ).setIncludeSynonyms(true)
```

This will cause all subsequent `getColumns` method calls on the connection to include synonyms. This is similar to `setRemarksReporting`. Alternatively, you can set the `includeSynonyms` connection property. This is similar to the `remarksReporting` connection property.

However, bear in mind that if `includeSynonyms` is `true`, then the name of the object returned in the `table_name` column will be the synonym name, if a synonym exists. This is true even if you pass the table name to `getColumns`.

Considerations for getProcedures and getProcedureColumns Methods

According to JDBC versions 1.1 and 1.2, the methods `getProcedures` and `getProcedureColumns` treat the `catalog`, `schemaPattern`, `columnNamePattern`, and `procedureNamePattern` parameters in the same way. In the Oracle definition of these methods, the parameters are treated differently:

- `catalog`

  Oracle does not have multiple catalogs, but it does have packages. Consequently, the `catalog` parameter is treated as the package name. This applies both on input, which is the `catalog` parameter, and the output, which is the `catalog` column in the returned `ResultSet`. On input, the construct `" "`, which is an empty string, retrieves procedures and arguments without a package, that is, standalone objects. A `null` value means to drop from the selection criteria, that is, return information about both standalone and packaged objects. That is, it has the same effect as passing in the percent sign (`%`). Otherwise, the `catalog` parameter should be a package name pattern, with SQL wild cards, if desired.

- `schemaPattern`

  All objects within Oracle database must have a schema, so it does not make sense to return information for those objects without one. Thus, the construct `" "`, which is an empty string, is interpreted on input to mean the objects in the current schema, that is, the one to which you are currently connected. To be consistent with the behavior of the `catalog` parameter, `null` is interpreted to drop the schema from the selection criteria. That is, it has the same effect as passing in `%`. It can also be used as a pattern with SQL wild cards.

- `procedureNamePattern` and `columnNamePattern`

  The empty string (" ") does not make sense for either parameter, because all procedures and arguments must have names. Thus, the construct `" "` will raise an exception. To be consistent with the behavior of other parameters, `null` has the same effect as passing in percent sign (`%`).