Introduction to Fetching Data from an Oracle Database using Java and JDBC part 1

by Eric Jenkinson on August 9, 2010
Categories: Java,JDBC
Tagged: Java, java.sql, JDBC, oracle.jdbc, Statement

In this three part series of posts we will look at fetching data from an Oracle Database using Java and JDBC. In this first post we look at the basics of the `Statement` interface and how to process simple queries. The other two articles will look at the`PreparedStatement` and the `CallableStatement`.

Note: The example programs presented in this series of post make use of the HR demonstration schema.

Using simple SQL statement such as the one below we will build a Java program using JDBC to execute the SQL and print out the results. This post will focus on the `java.sql.Statement` interface.

```
SQL> select employee_id, first_name, last_name, hire_date from employees;

EMPLOYEE_ID FIRST_NAME           LAST_NAME                 HIRE_DATE
----------- -------------------- ------------------------- ---------
        198 Donald               OConnell                  21-JUN-07
        199 Douglas              Grant                     13-JAN-08
        200 Jennifer             Whalen                    17-SEP-03
        201 Michael              Hartstein                 17-FEB-04
        202 Pat                  Fay                       17-AUG-05
        203 Susan                Mavris                    07-JUN-02
        204 Hermann              Baer                      07-JUN-02
        205 Shelley              Higgins                   07-JUN-02
        206 William              Gietz                     07-JUN-02
        100 Steven               King                      17-JUN-03
        101 Neena                Kochhar                   21-SEP-05

< cut for clarity >

        197 Kevin                Feeney                    23-MAY-06

107 rows selected.

SQL>
```

A Statement object is used to send and execute SQL statements on a given connection. There are three types of `Statement`objects in the package `java.sql` each specialized in a particular type of SQL statement.

`Statement` - SQL statements with no input (bind values) parameters.
`PreparedStatement` – preparsed SQL statements with or without input (bind values) parameters. Extends `Statement`.
`CallableStatement` – execute and retrieve data from stored procedures. Extends `PreparedStatement`.

Below is a Java program that will process the same query presented above.

```java
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class FetchRows1 {

  public static void main(String[] args) {

    try {
      // create the Oracle DataSource and set the URL
      OracleDataSource ods = new OracleDataSource();
      ods.setURL("jdbc.oracle.thin:hr/password@ora1:1521/orcl");

      // connect to the database and turn off auto commit
      OracleConnection ocon = (OracleConnection)ods.getConnection();
      ocon.setAutoCommit(false);

      // create the statement and execute the query
      Statement stmt = ocon.createStatement();
      ResultSet rset = stmt.executeQuery("select employee_id, first_name, last_name, hire_date

      // print out the results
      while(rset.next()) {
        System.out.println(rset.getInt(1) + ", " +
                           rset.getString(2)  + ", " +
                           rset.getString(3)  + ", " +
                           rset.getDate(4));
      }

    } catch (SQLException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

An SQL statement is executed within the context of a `Connection` so we need to use one of the methods provided by the`Connection` object to create the `Statement`. The `Connection` object provides three methods in which to create a Statement object.

```
Statement createStatement()
```

Returns a Statement object that will generate `ResultSet` objects that have a forward only cursor and that are read only.

```
Statement createStatement(int rsType, int rsConcurrency);
```

Returns a Statement object that will generate `ResultSet` objects of the given type and concurrency. Valid rsTypes are:`ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE` and`ResultSet.TYPE_SCROLL_SENSITIVE`. Valid `rsConcurrency` values are `ResultSet.CONCUR_READ_ONLY` and`ResultSet.CONCUR_UPDATABLE`.

```
Statement createStatement(int rstType, int rsConcurrency, int rsHoldability);
```

Returns a Statement object that will generate `ResultSet` objects of the given type, concurrency and `holdability`. The valid values for `rsHoldability` are `ResultSet.HOLD_CURSORS_OVER_COMMIT` and`ResultSet.CLOSE_CURSORS_AT_COMMIT`.

How the `ResultSet` is ultimately going to be used will determine the proper `createStatement` method to call. In the example program a forward only `ResultSet` was all that was needed.

After creating the Statement object `stmt` the code then executes the query by calling the `executeQuery` method passing in a String for the query. The `ResultSet` object returned contains the results of the query.

Next the program traverses through the `ResultSet` using the next() method. Each column is printed separated by commas. Notice that getter methods for a specific data type are used.

```
SQL> describe employees
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 EMPLOYEE_ID                               NOT NULL NUMBER(6)
 FIRST_NAME                                         VARCHAR2(20)
 LAST_NAME                                 NOT NULL VARCHAR2(25)
 EMAIL                                     NOT NULL VARCHAR2(25)
 PHONE_NUMBER                                       VARCHAR2(20)
 HIRE_DATE                                 NOT NULL DATE
 JOB_ID                                    NOT NULL VARCHAR2(10)
 SALARY                                             NUMBER(8,2)
 COMMISSION_PCT                                     NUMBER(2,2)
 MANAGER_ID                                         NUMBER(6)
 DEPARTMENT_ID                                      NUMBER(4)

SQL>
```

The query is to return the EMPLOYEE_ID, FIRST_NAME, LAST_NAME and HIRE_DATE so the data types returned in the`ResultSet` are `NUMBER`, `VARCHAR2`, `VARCHAR2` and `DATE`. The Oracle data types are not present in JDBC so we will have to use Integer, String and Date.

The number passed in each getter method is the column index in the result set. The column index of a `ResultSet` begins with 1. The code could have been written using the actual column names as below to make the code more readable.

```
// print out the results
while(rset.next()) {
  System.out.println(rset.getInt("EMPLOYEE_ID") + ", " +
                     rset.getString("FIRST_NAME")  + ", " +
                     rset.getString("LAST_NAME")  + ", " +
                     rset.getDate("HIRE_DATE"));
}
```

The Oracle JDBC drivers include `Statement` and `ResultSet` extensions that are tailored to the Oracle Database. For example `Statement`, `PreparedStatement`, `CallableStatement` and `ResultSet` can be replaced with`OracleStatement`, `OraclePreparedStatement`, OracleCallableStatement and `OracleResultSet` respectively which are include in the package `oracle.jdbc`.

Below is the program converted to using the Oracle JDBC extensions.

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
import oracle.jdbc.OracleStatement;
import oracle.jdbc.OracleResultSet;
import java.sql.SQLException;

public class FetchRows2 {

  public static void main(String[] args) {

    try {
      // create the Oracle DataSource and set the URL
      OracleDataSource ods = new OracleDataSource();
      ods.setURL("jdbc:oracle:thin:hr/hr@ora1:1521/orcl");

      // connect to the database and turn off auto commit
      OracleConnection ocon = (OracleConnection)ods.getConnection();
      ocon.setAutoCommit(false);

      // create the statement and execute the query
      OracleStatement stmt = (OracleStatement)ocon.createStatement();
      OracleResultSet rset = (OracleResultSet)stmt.executeQuery("select employee_id, first_name

      // print out the results
      while(rset.next()) {
        System.out.println(rset.getNUMBER("EMPLOYEE_ID").intValue() + ", " +
                           rset.getCHAR("FIRST_NAME")  + ", " +
                           rset.getCHAR("LAST_NAME")  + ", " +
                           rset.getDATE("HIRE_DATE").dateValue());
      }

    } catch (SQLException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Using the Oracle extensions requires casts when obtaining the `OracleStatement` and when receiving the`OracleResultSet` after executing the `OracleStatement`. Also note that the `OracleResultSet` has getter methods have names that match Oracle column types. With some of these extensions such as `getCHAR()` and `getNUMBER()` data conversion is not necessary.

Misuse of the Statement object

A common misuse of the Statement object is to use the Statement object to process a query multiple times with different values in the where clause. Below is an example.

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class FetchRows3 {

  public static void main(String[] args) {

    try {
      // create the Oracle DataSource and set the URL
      OracleDataSource ods = new OracleDataSource();
      ods.setURL("jdbc:oracle:thin:hr/hr@ora1:1521/orcl");

      // connect to the database and turn off auto commit
      OracleConnection ocon = (OracleConnection)ods.getConnection();
      ocon.setAutoCommit(false);

      // create the statement
      Statement stmt = ocon.createStatement();;
      ResultSet rset;
      String sqlStr;

      for(int i = 100; i <= 206; i++) {

        // build the query
        sqlStr = "select employee_id, first_name, last_name, hire_date from employees where emp

        // execute the new query
        rset = stmt.executeQuery(sqlStr);

        // process the result set
        while(rset.next()) {
          System.out.println(rset.getInt("EMPLOYEE_ID") + ", " +
                             rset.getString("FIRST_NAME")  + ", " +
                             rset.getString("LAST_NAME")  + ", " +
                             rset.getDate("HIRE_DATE"));
        }
      }

    } catch (SQLException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

This program returns the same results as the programs presented earlier but it does so in a very inefficient manner. The program generates and executes 107 different SQL statements.

While the code only shows the line `rset = stmt.executeQuery(sqlStr);` is executed 107 times, Oracle sees the following:

```
select employee_id, first_name, last_name, hire_date from employees where employee_id = 100
select employee_id, first_name, last_name, hire_date from employees where employee_id = 101
select employee_id, first_name, last_name, hire_date from employees where employee_id = 102
…
select employee_id, first_name, last_name, hire_date from employees where employee_id = 206
```

Each of those statements could result in a hard parse depending on the database parameter `CURSOR_SHARING` which defaults to `EXACT`. You can see this in the database by taking look at `V$SQL`.

```
SQL> set linesize 130
SQL> set pagesize 999
SQL> select sql_text
  2  from v$sql
  3  where sql_text like 'select employee_id, first_name, last_name, hire_date from employees w

SQL_TEXT
--------------------------------------------------------------------------------------------------
select employee_id, first_name, last_name, hire_date from employees where employee_id = 120
select employee_id, first_name, last_name, hire_date from employees where employee_id = 161

< … cut for clarity … >
```

```

select employee_id, first_name, last_name, hire_date from employees where employee_id = 190
select employee_id, first_name, last_name, hire_date from employees where employee_id = 167

107 rows selected.

SQL>
```

Each of those statements required a hard parse and the generation of an execution plan even though they only differed in the literal value for employee_id. Since the database is forced to hard parse every statement sent by the program CPU utilization will increase and other applications including this one may be forced to wait for the shared pool to become available.

If you have the need to execute multiple SQL statements that differ only in literal values then you should use the`PreparedStatement` which is the topic of the next post in this series.