Accessing Oracle from Java

Salman Khan

Introduction

You're writing a Java program and you want to access a database. What should you use? JDBC and/or SQLJ. These are both APIs (or drivers or packages... whatever you want to call them) used to access a database. Why two API's rather than one? The simple answer is that each one is better for specific situations. Actually, you only need JDBC, but as we'll see later on in this paper, you'll want to use SQLJ as well.

JDBC is an vendor-independent standard developed by Javasoft which was modeled after ODBC (developed by Microsoft). It stands for Java Database Connectivity. JDBC is today the *de facto* method for accessing a database from Java. Several vendors, including Oracle, Intersolv, and Weblogic develop several types (the various Oracle drivers will be explained in more detail a few paragraphs from now) of JDBC drivers. Oracle's, of course, are the best (I'm the product manager for Oracle's JDBC drivers so it is my duty to brainwash you).

SQLJ is a new standard for embedding SQL directly into Java programs. It is the product of a joint standards effort by Oracle, IBM, Sybase, Tandem, and Informix. For those you who are familiar with Pro*C (embedded SQL in C), you can view SQLJ as Pro*Java. Why isn't it called Pro*Java then? Pro*anything implies an Oracle-specific technology (Pro*prietary?) while SQLJ is an open standard. As we'll see later in this paper, SQLJ simplifies the task of writing, managing, and debugging database-backed Java applications to the point that it is almost pleasurable.

JDBC

The guts of a JDBC program

Before delving into JDBC any deeper, it's important that you take a look and understand how a simple JDBC program works.

```
import java.sql.*;
public class JDBCExample {
  public static void main(String args[]) throws SQLException {
    DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
    Connection conn=
     DriverManager.getConnection("jdbc:oracle:thin:@dlsun137:5521:sol1",
                                 "scott", "tiger");
    Statement stmt = conn.createStatement();
    ResultSet rs =
     stmt.executeQuery("select ename, empno, sal from emp");
    while(rs.next()){
      String name= rs.getString(1);
      int number = rs.getInt(2);
      double salary = rs.getDouble(3);
      System.out.println(name+" "+number+" "+salary);
      }
    rs.close();
    conn.close();
  }
}
```

The above example is a complete Java program. If you understand how it works, you're well on your way to becoming a JDBC expert. The first line in the main method of the is:

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

All you really have to know is that this call lets JDBC know that you are using Oracle's drivers.

The next call is:

```
Connection conn = DriverManager.getConnection(
                "jdbc:oracle:oci8:@server:1521:orcl",“scott”, “tiger”);
```

As you may have guessed, this call creates a connection `conn` to a database. The `jdbc:oracle:oci8` part of the *connect string* says to use an oracle OCI-based (what an OCI-based driver is will be explained in the next section) JDBC driver to establish the connection and communicate with the database. The `@server:1521:orcl` part of the string specifies to connect to the `orcl` database instance on machine `server` whose listener is on port `1521`. Finally, the connection is being made to the schema `scott` with the password `tiger`.

Now that the connection is made, we're ready to make SQL queries to the database. In the example, a query is made by the line:

```
ResultSet rs = stmt.executeQuery(“select ename, empno, sal from emp”);
```

The query string "select ename, empno, sal, from emp" queries the ename, empno, and sal columns from the table emp. The executeQuery method of the connection object passes the query string to SQL. If you were to type this string directly to a SQLPlus or Server Manager prompt you would get something like:

```
ENAME             EMPNO             SAL
---------------   ---------------   ---------------
SCOTT             12                12000
ADAM              2                 10000
.                 .                 .
.                 .                 .
.                 .                 .
LESLIE            9                 15000
```

In Java, this "result" will be stored in a `ResultSet` object (`rs` in this example). The data in this "result" is accessed by the following code:

```
while(rs.next())
{
  String name = rs.getString(1);
  int number =  rs.getInt(2);
  double salary = rs.getDouble(3);
  System.out.println(name+” “+number+” “+salary);
}
```

The `while` loop loops through the "result" `rs` row by row. If there are any rows left, the call to `rs.next()` returns `true` and "points" `rs` to the next row. Otherwise, `rs.next()` would return `false` and the `while` loop would terminate. So, after the first call to `rs.next()`, the result set would point to the row:

```
SCOTT              12                  12000
```

and when the result set points to the row:

```
LESLIE             9                   15000
```

the next call to `rs.next()` would return false.

Once in a given row, the `getXXX` methods of the result set can be used to actually access the SQL data (where XXX can be most any Java type). In the above code, the methods `getString`, `getInt`, and `getDouble` are used. The XXX part of the `getXXX` lets JDBC know what Java type to map the corresponding SQL type to. In the above example, the data in the `ename` column is accessed using the call `rs.getString(1)`. This call gets the data in the first column of the result set and maps it to a Java `String`. This is a natural mapping for the `ename` column because in SQL, the data in the column would be of type `VARCHAR2`. Likewise, the data in the `empno` column is accessed using the call `rs.getInt(2)`. This call gets the data from the second column of the result set and maps it to a Java `int`. Once again, this makes sense since the corresponding SQL data would be of type `NUMBER`. Finally, the call `rs.getDouble(3)` gets the data from the third column (which would be sal in this example) and maps it to a Java double.

The while loop in this example outputs the following:

```
SCOTT 12 12000
ADAM 2 10000
     .
     .
     .
LESLIE 9 15000
```

The last lines of the program, `rs.close()` and `conn.close()`, lets JDBC know that you are done with the JDBC result set and connection and that the database connection and corresponding session should be ended.

As I said before, if you have a decent understanding of the above code example, you are ready to write JDBC. Everything else in this paper from here on out is just icing on the cake.

Oracle's JDBC Drivers

Now that you have a reasonable understanding of what JDBC is and how it can be used, let's focus on Oracle's JDBC offering. Oracle offers three types of JDBC drivers: a 100% Java thin driver, an OCI-based fat driver, and a server-side (a.k.a. KPRB) driver. All three of these drivers currently comply with the JDBC 1.2.2 specification and are well on their way to complying with JDBC 2.0 as well. In addition, the drivers are compatible with JDK 1.0.2, and 1.1.6. The thin and fat JDBC drivers can be used against Oracle7.2 and higher databases. Finally, all three drivers use identical API's, thus providing maximum flexibility in how JDBC applications can be deployed.

As already mentioned, Oracle's thin driver is a 100% Java implementation. Because of this, it is ideal for applications that need to be 100% Java such as applets. Another benefit of the thin driver is that, at only 300K (150K compressed), it is extremely small in size making it even more attractive for use in downloadable applications such as applets. In terms of performance, the thin driver is slightly slower than its OCI-based big brother (this should be expected since the OCI-based driver does a lot of the data marshaling in C).

Oracle's fat driver is a JDBC layer implemented on top of the Oracle Call Interface (OCI). OCI is a C library for accessing Oracle databases. Because the fat driver needs OCI to be pre-installed, it cannot be used for 100% Java and/or downloadable applications. This driver is ideal, however, for use in middle-tier applications such as Java servlets.

Oracle's server-side driver is the new addition to the family. Now that Oracle8*i* allows Java applications to run inside the database, these applications need some way to actually access the data in the database (the main reason to run Java in the database is if the program deals a lot with data). The server-side drivers were created with this use in mind. Because these drivers "know" that the code is in the database, they don't have to explicitly open up a connection. Also, they completely avoid latency due to the network or interprocess communication by directly accessing SQL data (as you may already know, Java in Oracle8*i* runs in the same memory and process space as the database instance).

As touched on earlier in this section, all three of the aforementioned drivers use the same API. This gives Java/JDBC developers a great deal of flexibility in how they deploy their Java applications. For example, say a client-server application is written using the JDBC-OCI driver. In this case the client would be the combination of the Java application and the driver and the server would be an Oracle database. Then Oracle8*i* comes out, and the developer realizes that she might get significant performance gains if she runs the data-intensive pieces of her application within the database. This task would literally be as simple as loading the data intensive Java classes into the database and changing the lines that establish connections to:

```
Connection conn =
     new oracle.jdbc.driver.OracleDriver().defaultConnection();
```

Likewise, the developer could instead decide that she would like to deploy her application as a Java applet. In this case, the only JDBC modification she would have to make is to specify to use the thin rather than OCI-based driver in the connect string. In today's world, this flexibility cannot be overemphasized. It allows developers to be confident that their application can quickly and easily be adapted to the "deployment configuration of the month."

JDBC is not perfect

Before you call your mother and let her know about all the virtues of JDBC (which have been so expertly taught to you) let me warn you that it does have some drawbacks. First and foremost, errors in SQL statements can only be caught at runtime. This is because the Java compiler used to compile the JDBC code has no knowledge of SQL syntax or semantics. Second, writing complex SQL statements in JDBC can get messy. This is illustrated in the following code:

```
String name="SALMAN";
int id=37;
float salary=2500;
PreparedStatement stmt = conn.prepareStatement
       ("insert into emp (ename, empno, sal) values (?, ?, ?)");
stmt.setString (1, name);
stmt.setInt(2, id);
stmt.setFloat(3, salary);
stmt.execute ();
stmt.close ();
```

This first three lines just define three Java variables `name`, `id`, and `salary` whose values are `"SALMAN"`, `37`, and `20000` respectively. The next line then defines something call a prepared statement. A prepared statement should be used whenever values within a SQL statement need to by dynamically set (the same prepared statement can be used over and over with different variable values). The `"?"`s in this prepared statement can be viewed as placeholders for Java variables. The `"?"`s are then given values in the lines:

```
stmt.setString (1, name);
stmt.setInt(2, id);
stmt.setFloat(3, salary);
```

The first `"?"` in the prepared statement is set to the Java variable `name` (which holds the String `"SALMAN"`). The second `"?"` is set to the `int` variable `id` (has a value of 37115) . Finally, the third `"?"` is set to the `float` variable `salary` (which has a value of 20000). This prepared statement example, as promised, is messy and the SQL statement that it executes is not even that complicated. I'm sure you could imagine the complexity of multi-line SQL calls which use many more Java variables.

This example begs the question, "is there anything you can do to make SQL queries in Java that 1) are easier to write and read and 2) that can be debugged before runtime?"
ANSWER: Use SQLJ

SQLJ

To give you a very quick feel for what SQLJ looks like and why it is so good, let's take a look at a snippet of SQLJ that performs the exact same SQL call as the "messy" JDBC example of the last section.

```
String name="SALMAN";
int id=37;
float salary=20000;
#sql {insert into emp (ename, empno, sal) values (:name,:id,:salary)};
```

At a first glance, it is easy to see that this SQLJ code is significantly simpler that the JDBC equivalent. This mainly stems from the fact that SQLJ allows for Java bind variables and expressions (preceded by a ":") to be embedded within SQL statements. In the above example the Java variables `name`, `id`, and `salary` are directly embedded in the SQL statement.

SQLJ also addresses the issue of debugging SQL statements before runtime. Because SQLJ code is not pure Java syntax, it needs to be run through a translator before it can be compiled to Java bytecode (this is outlined in more detail in the next section). This translation step allows embedded SQL statements to be analyzed both semantically and syntactically, thus catching SQL errors at translate time rather than runtime (as is the case with JDBC).

At this point you're probably thinking, "If SQLJ addresses both of the drawbacks of JDBC, why do I even need to know about JDBC?" There are several reasons why you should know about both. First, they are complementary APIs. SQLJ and JDBC can and should be used together. JDBC must be used for dynamic SQL (when the SQL statements are generated dynamically at runtime) while SQLJ is ideal for static SQL (when the shape of the SQL query is known at application development time).

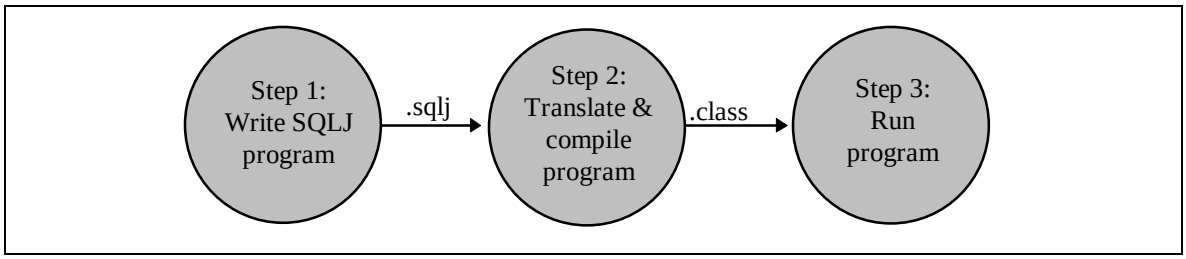
Second, Oracle's current incarnation of SQLJ runs on top of JDBC. This allows SQLJ to be deployed in any configuration as JDBC (thin, fat, or server-side) but it also means that the SQLJ developer needs to know about the various types of JDBC drivers.

Developing a SQLJ Application

There are three steps to build a SQLJ application:

1. Write the SQLJ program
2. Translate the SQLJ program.
3. Run the SQLJ program.

(Take a minute to and try to memorize these steps. It may seem overwhelming at first, but I think you're smart enough to handle this level of complexity.)



Writing a SQLJ program

Just as we did with JDBC, let us take a look at the pieces of an actual SQLJ program:

```
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;
#sql iterator MyIter (String ename, int empno, float sal);

public class MyExample
{
   public static void main (String args[]) throws SQLException
   {
      Oracle.connect
         ("jdbc:oracle:thin:@oow11:5521:sol2", "scott", "tiger");

      #sql { insert into emp (ename, empno, sal)
             values ('SALMAN', 32, 20000) };
      MyIter iter;

      #sql iter={ select ename, empno, sal from emp };
      while (iter.next()) {
         System.out.println
               (iter.ename()+" "+iter.empno()+" "+iter.sal());
            }
   }
}
```

The first line of the main method is:

```
Oracle.connect("jdbc:oracle:thin:@oow11:5521:sol2", "scott", "tiger");
```

This call establishes a default database connection for SQLJ to use.

Now we're at the exciting part! We're ready to write a simple, easier-than-JDBC, SQLJ clause. The first clause is:

```
#sql { insert into emp (ename, empno, sal) values
        ('SALMAN', 32, 20000)};
```

This SQLJ code is doing exactly what you think it is; it's inserting a row into the emp table.

The next clause is:

```
#sql miter={select ename, empno, sal from emp};
```

As you may have guessed, this statement is simply making a `select` query. So now you ask, "Wait! I see a query, but I don't see a result set to put the data into. Where does the data from the query go?" For your information, we don't use result sets in SQLJ; we use *iterators.* You should visualize an iterator the same way as a result set. The main difference between the two is that a given iterator can only store the output of a specific type of query while a result set can store the output of any query. Because of this, iterator types need to be defined for different types of queries. In the above example, the following line defines an iterator type called `MyIter`:

```
#sql iterator MyIter (String ename, int empno, float sal);
```

By looking at this definition, we can see that this type of iterator can store results whose first column can be mapped to a Java `String`, whose second column can be mapped to a Java `int`, and whose third row can be mapped to a Java `float`. This definition also names these three rows `ename`, `empno`, and `sal` respectively (`MyIter` is a *named iterator*. See the documentation to learn about positional iterators). An instance of `MyIter` is then created by the line:

```
MyIter miter;
```

Finally, the miter instance is populated with data in the select query.

Now that we have a populated iterator, we need to actually access the data in it. This happens in the following snippet of code:

```
while (miter.next())
  {
    System.out.println(miter.ename()+" "+miter.empno()+" "+miter.sal());
      }
```

This `while` loop pretty much does the same thing as the `while` loop in the JDBC example. The `next()` method of an iterator plays the same role the `next()` method of a result set. If the iterator is not at the last row of the data, it returns `true` and points the interator to the next row. Otherwise, it returns `false`. How the data is accessed from a given row in an iterator is slightly different then how it is done for a result set. The iterator in the example has methods which correspond to the names of the columns in the iterator. By calling these methods, the data in that column is returned. As you may remember, we had to use the `getXXX()` methods of a result set object to do the same thing.

At this point, I believe congratulations are in order. Assuming that you have diligently read the last few paragraphs, you are ready to run home an be a SQLJ-head (Be proud! I consider myself a SQLJ-head so you're in excellent company). The rest of the paper is downhill from here.

Translating and Compiling a SQLJ program

Let's take a look at how difficult it is to translate and compile a SQLJ program:

```
>sqlj MyExample.sqlj
```

There are a host of options which can be specified in addition to the arguments given above, but the above call is enough.

Translating and compiling a SQLJ application is so simple that it almost doesn't deserve its own section. The only reason why it does is that what the translator does for you is really quite interesting.

While translating a .sqlj file into a .java file., the SQLJ translator actually checks to see if your embedded-SQL makes sense. It analyzes your code from a syntactic and semantic point of view. It can even check to see if the tables, columns, and Java types that you are using are appropriate. How does it do this? It actually connects to an exemplar (fancy word for example) database schema. This schema doesn't have to have identical data to the one that the program will eventually run against, but the tables in it should have the same shapes and names (If you want the translator to connect to an exemplar database, you have to specify a URL of a database to connect to along with the appropriate username and password).

Besides translate-time SQL checking, the translator also generates a serialized object file (ends with the .ser extension). This file contains a description of the SQL queries made in the SQLJ program. It has two purposes. First, the .java files generated from the SQLJ translator make calls to a Java library called the SQLJ runtime. The classes in this library use the serialized object file to figure out what JDBC calls to make. Second, the serialized object file can be used to customize the SQLJ program for a specific vendors database (this is done automatically by the SQLJ translator) .

Compiling the .java files generated by the translator is taken care of by the SQLJ translator, and is, thus, transparent to most SQLJ users. Nothing fancy is happening behind the scenes. The JDK javac compiler is run on all of the .java files. The compilation process generates Java bytecode so, after this step, the SQLJ program can be run with the simple call:

```
>java MyExample
```

Now your SQLJ program is a lean, mean, Oracle-database-accessing, machine!

Conclusion

How does a Java application access a database? It can use JDBC and / or SQLJ. JDBC is the *de facto* API for accessing databases from Java. It provides a rich set of interfaces for passing SQL statements to databases and manipulating SQL data in Java. SQLJ is embedded SQL in Java. SQLJ simplifies writing, reading, and debugging SQL calls from Java.

JDBC and SQLJ are complementary (i.e. it isn't an either/or proposition). JDBC is required for dynamic SQL while SQLJ is preferable for static SQL. Also, Oracle's current implementation of SQLJ runs on top of JDBC.

Oracle offers three different JDBC drivers: thin, fat, and server-side. The thin driver is 100% Java and is ideal for Java applets or any Java applications which need to be completely platform independent. The fat driver is written on top of OCI. Because of this, it is more appropriate for middle-tier environments such as application servers. Now that Oracle8*i* allows Java programs to run inside the database, these programs need a way to access data. This functionality is provided by Oracle's server-side (also know as KPRB) driver. Because SQLJ runs on top of JDBC, SQLJ programs can be deployed in thin, fat, or server-side configurations as well. Also, all three drivers implement the same API, thus providing maximum flexibility in how Java applications are deployed and re-deployed.

Last, but not least, everything mentioned in this paper (everything being JDBC and SQLJ) is standards based and is not an Oracle proprietary solution.

We, my friend, are now at the end of our accessing-databases-from-Java journey. It's time to get off the train, find our children and reflect on where we've been and where we're going. If you're like me, JDBC/SQLJ will change your life forever and provide you with that little beacon of hope which is so often needed in this cold and heartless world. So remember, when life gets rough ... use JDBC and SQLJ!