Resources

- Database Systems: The Complete Book by Hector Garcia, Jeff Ullman, and Jennifer Widom.
- A First Course in Database Systems by Jeff Ullman and Jennifer Widom.
- Gradiance SQL Tutorial.

Introduction to JDBC

This document illustrates the basics of the JDBC (Java Database Connectivity) API (Application Program Interface). Here, you will learn to use the basic JDBC API to create tables, insert values, query tables, retrieve results, update tables, create prepared statements, perform transactions and catch exceptions and errors.

This document draws from the official Sun tutorial on JDBC Basics.

Overview

Call-level interfaces such as JDBC are programming interfaces allowing external access to SQL database manipulation and update commands. They allow the integration of SQL calls into a general programming environment by providing library routines which interface with the database. In particular, Java based JDBC has a rich collection of routines which make such an interface extremely simple and intuitive.

Here is an easy way of visualizing what happens in a call level interface: You are writing a normal Java program. Somewhere in the program, you need to interact with a database. Using standard library routines, you open a connection to the database. You then use JDBC to send your SQL code to the database, and process the results that are returned. When you are done, you close the connection.

Such an approach has to be contrasted with the precompilation route taken with Embedded SQL. The latter has a precompilation step, where the embedded SQL code is converted to the host language code(C/C++). Call-level interfaces do not require precompilation and thus avoid some of the problems of Embedded SQL. The result is

increased portability and a cleaner client-server relationship.

Establishing A Connection

The first thing to do, of course, is to install Java, JDBC and the DBMS on your working machines. Since we want to interface with an Oracle database, we would need a driver for this specific database as well. Fortunately, we have a responsible administrator who has already done all this for us on the Leland machines.

As we said earlier, before a database can be accessed, a connection must be opened between our program(client) and the database(server). This involves two steps:

- **Load the vendor specific driver**

  Why would we need this step? To ensure portability and code reuse, the API was designed to be as independent of the version or the vendor of a database as possible. Since different DBMS's have different behavior, we need to tell the driver manager which DBMS we wish to use, so that it can invoke the correct driver.

  An Oracle driver is loaded using the following code snippet:

  ```
      Class.forName("oracle.jdbc.driver.OracleDriver")
  ```

- **Make the connection**

  Once the driver is loaded and ready for a connection to be made, you may create an instance of a Connection object using:

  ```
  Connection con = DriverManager.getConnection(
     "jdbc:oracle:thin:@dbaprod1:1544:SHR1_PRD", username, passwd);
  ```

  Okay, lets see what this jargon is. The first string is the URL for the database including the protocol *(jdbc)*, the vendor *(oracle)*, the driver *(thin)*, the server *(dbaprod1)*, the port number *(1521)*, and a server instance *(SHR1_PRD)*. The username and passwd are *your* username and password, the same as you would enter into *SQLPLUS* to access your account.

That's it! The connection returned in the last step is an open connection which we will use to pass SQL statements to the database. In this code snippet, `con` is an open connection, and we will use it below. **Note:***The values mentioned above are valid for our (Leland) environment. They would have different values in other environments.*

Creating JDBC Statements

A JDBC `Statement` object is used to send your SQL statements to the DBMS, and should not to be confused with an SQL statement. A JDBC `Statement` object is associated with an open connection, and not any single SQL Statement. You can think of a JDBC `Statement` object as a channel sitting on a connection, and passing one or more of your SQL statements (which you ask it to execute) to the DBMS.

An active connection is needed to create a `Statement` object. The following code snippet, using our `Connection` object `con`, does it for you:

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

At this point, a `Statement` object exists, but it does not have an SQL statement to pass on to the DBMS. We learn how to do that in a following section.

Creating JDBC PreparedStatement

Sometimes, it is more convenient or more efficient to use a `PreparedStatement` object for sending SQL statements to the DBMS. The main feature which distinguishes it from its superclass `Statement`, is that unlike `Statement`, it is given an SQL statement right when it is created. This SQL statement is then sent to the DBMS right away, where it is compiled. Thus, in effect, a `PreparedStatement` is associated as a channel with a connection and a compiled SQL statement.

The advantage offered is that if you need to use the same, or similar query with different parameters multiple times, the statement can be compiled and optimized by the DBMS just once. Contrast this with a use of a normal `Statement` where each use of the same SQL statement requires a compilation all over again.

`PreparedStatements` are also created with a `Connection` method. The following snippet shows how to create a parameterized SQL statement with three input parameters:

```
   PreparedStatement prepareUpdatePrice = con.prepareStatement(
      "UPDATE Sells SET price = ? WHERE bar = ? AND beer = ?");
```

Before we can execute a `PreparedStatement`, we need to supply values for the parameters. This can be done by calling one of the `setXXX` methods defined in the class `PreparedStatement`. Most often used methods are `setInt, setFloat, setDouble, setString` etc. You can set these values before each execution of the prepared statement.

Continuing the above example, we would write:

```
   prepareUpdatePrice.setInt(1, 3);
   prepareUpdatePrice.setString(2, "Bar Of Foo");
   prepareUpdatePrice.setString(3, "BudLite");
```

Executing CREATE/INSERT/UPDATE Statements

Executing SQL statements in JDBC varies depending on the ``intention'' of the SQL statement. DDL (data definition language) statements such as table creation and table alteration statements, as well as statements to update the table contents, are all executed using the method `executeUpdate`. Notice that these commands change the state of the database, hence the name of the method contains ``Update''.

The following snippet has examples of `executeUpdate` statements.

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

   stmt.executeUpdate("CREATE TABLE Sells " +
```

```
      "(bar VARCHAR2(40), beer VARCHAR2(40), price REAL)" );
   stmt.executeUpdate("INSERT INTO Sells " +
      "VALUES ('Bar Of Foo', 'BudLite', 2.00)" );

   String sqlString = "CREATE TABLE Bars " +
      "(name VARCHAR2(40), address VARCHAR2(80), license INT)" ;
   stmt.executeUpdate(sqlString);
```

Since the SQL statement will not quite fit on one line on the page, we have split it into two strings concatenated by a plus sign(+) so that it will compile. Pay special attention to the space following "INSERT INTO Sells" to separate it in the resulting string from "VALUES". Note also that we are reusing the same `Statement` object rather than having to create a new one.

When `executeUpdate` is used to call DDL statements, the return value is always zero, while data modification statement executions will return a value greater than or equal to zero, which is the number of tuples affected in the relation.

While working with a `PreparedStatement`, we would execute such a statement by first plugging in the values of the parameters (as seen above), and then invoking the `executeUpdate` on it.

```
      int n = prepareUpdatePrice.executeUpdate() ;
```

Executing SELECT Statements

As opposed to the previous section statements, a query is expected to return a set of tuples as the result, and not change the state of the database. Not surprisingly, there is a corresponding method called `executeQuery`, which returns its results as a `ResultSet` object:

```
   String bar, beer ;
   float price ;

   ResultSet rs = stmt.executeQuery("SELECT * FROM Sells");
   while ( rs.next() ) {
      bar = rs.getString("bar");
      beer = rs.getString("beer");
      price = rs.getFloat("price");
      System.out.println(bar + " sells " + beer + " for " + price + " Dollars
   }
```

The bag of tuples resulting from the query are contained in the variable `rs` which is an instance of `ResultSet`. A set is of not much use to us unless we can access each row and the attributes in each row. The `ResultSet` provides a cursor to us, which can be used to access each row in turn. The cursor is initially set *just before* the first row. Each invocation of the method `next` causes it to move to the next row, if one exists and return `true`, or return `false` if there is no remaining row.

We can use the `getXXX` method of the appropriate type to retrieve the attributes of a row. In the previous example, we used `getString` and `getFloat` methods to access the column values. Notice that we provided the name of the column whose value is desired as a parameter to the method. Also note that the `VARCHAR2` type `bar, beer` have been converted to Java `String`, and the `REAL` to Java `float`.

Equivalently, we could have specified the column number instead of the column name, with the same result. Thus the relevant statements would be:

```
bar = rs.getString(1);
price = rs.getFloat(3);
beer = rs.getString(2);
```

While working with a `PreparedStatement`, we would execute a query by first plugging in the values of the parameters, and then invoking the `executeQuery` on it.

```
ResultSet rs = prepareUpdatePrice.executeQuery() ;
```

Notes on Accessing ResultSet

JDBC also offers you a number of methods to find out where you are in the result set using `getRow`, `isFirst`, `isBeforeFirst`, `isLast`, `isAfterLast`.

There are means to make scroll-able cursors allow free access of any row in the result set. By default, cursors scroll forward only and are read only. When creating a `Statement` for a `Connection`, you can change the type of `ResultSet` to a more flexible scrolling or updatable model:

```
Statement stmt = con.createStatement(
   ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM Sells");
```

The different options for types are `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, and `TYPE_SCROLL_SENSITIVE`. You can choose whether the cursor is read-only or updatable using the options `CONCUR_READ_ONLY`, and `CONCUR_UPDATABLE`. With the default cursor, you can scroll forward using `rs.next()`. With scroll-able cursors you have more options:

```
rs.absolute(3);          // moves to the third tuple
rs.previous();           // moves back one tuple (tuple 2)
rs.relative(2);          // moves forward two tuples (tuple 4)
rs.relative(-3);         // moves back three tuples (tuple 1)
```

There are a great many more details to the scroll-able cursor feature. Scroll-able cursors, though useful for certain applications, are extremely high-overhead, and should be used with restraint and caution. More information can be found at the New Features in the JDBC 2.0 API, where you can find a more detailed tutorial on the cursor manipulation techniques.

Transactions

JDBC allows SQL statements to be grouped together into a single transaction. Thus, we can ensure the ACID (Atomicity, Consistency, Isolation, Durability) properties using JDBC transactional features.

Transaction control is performed by the `Connection` object. When a connection is created, by default it is in the auto-commit mode. This means that each individual SQL statement is treated as a transaction by itself, and will be committed as soon as it's execution finished. (This is not exactly precise, but we can gloss over this subtlety for

most purposes).

We can turn off auto-commit mode for an active connection with :

```
con.setAutoCommit(false) ;
```

and turn it on again with :

```
con.setAutoCommit(true) ;
```

Once auto-commit is off, no SQL statements will be committed (that is, the database will not be permanently updated) until you have explicitly told it to commit by invoking the `commit()` method:

```
con.commit() ;
```

At any point before commit, we may invoke `rollback()` to rollback the transaction, and restore values to the last commit point (before the attempted updates).

Here is an example which ties these ideas together:

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.executeUpdate("INSERT INTO Sells VALUES('Bar Of Foo', 'BudLite', 1
con.rollback();
stmt.executeUpdate("INSERT INTO Sells VALUES('Bar Of Joe', 'Miller', 2.
con.commit();
con.setAutoCommit(true);
```

Lets walk through the example to understand the effects of various methods. We first set auto-commit off, indicating that the following statements need to be considered as a unit. We attempt to insert into the `Sells` table the `('Bar Of Foo', 'BudLite', 1.00)` tuple. However, this change has not been made final (committed) yet. When we invoke `rollback`, we cancel our insert and in effect we remove any intention of inserting the above tuple. Note that `Sells` now is still as it was before we attempted the insert. We then attempt another insert, and this time, we commit the transaction. It is only now that `Sells` is now permanently affected and has the new tuple in it. Finally, we reset the connection to auto-commit again.

We can also set transaction isolation levels as desired. For example, we can set the transaction isolation level to `TRANSACTION_READ_COMMITTED`, which will not allow a value to be accessed until after it has been committed, and forbid dirty reads. There are five such values for isolation levels provided in the `Connection` interface. By default, the isolation level is serializable. JDBC allows us to find out the transaction isolation level the database is set to (using the `Connection` method `getTransactionIsolation`) and set the appropriate level (using the `Connection` method `setTransactionIsolation` method).

Usually `rollback` will be used in combination with Java's exception handling ability to recover from (un)predictable errors. Such a combination provides an excellent and easy mechanism for handling data integrity. We study error handling using JDBC in the next section.

Handling Errors with Exceptions

The truth is errors always occur in software programs. Often, database programs are critical applications, and it is imperative that errors be caught and handled gracefully. Programs should recover and leave the database in a consistent state. Rollback-s used in conjunction with Java exception handlers are a clean way of achieving such a requirement.

The client(program) accessing a server(database) needs to be aware of any errors returned from the server. JDBC give access to such information by providing two levels of error conditions: `SQLException` and `SQLWarning`. `SQLExceptions` are Java exceptions which, if not handled, will terminate the application. `SQLWarnings` are subclasses of `SQLException`, but they represent nonfatal errors or unexpected conditions, and as such, can be ignored.

In Java, statements which are expected to ``throw'' an exception or a warning are enclosed in a `try` block. If a statement in the `try` block throws an exception or a warning, it can be ``caught'' in one of the corresponding `catch` statements. Each `catch` statement specifies which exceptions it is ready to ``catch''.

Here is an example of catching an `SQLException`, and using the error condition to rollback the transaction:

```
try {
   con.setAutoCommit(false) ;
   stmt.executeUpdate("CREATE TABLE Sells (bar VARCHAR2(40), " +
                      "beer VARHAR2(40), price REAL)") ;
   stmt.executeUpdate("INSERT INTO Sells VALUES " +
                      "('Bar Of Foo', 'BudLite', 2.00)") ;
   con.commit() ;
   con.setAutoCommit(true) ;

}catch(SQLException ex) {
   System.err.println("SQLException: " + ex.getMessage()) ;
   con.rollback() ;
   con.setAutoCommit(true) ;
}
```

In this case, an exception is thrown because `beer` is defined as `VARHAR2` which is a mis-spelling. Since there is no such data type in our DBMS, an `SQLException` is thrown. The output in this case would be:

```
Message:  ORA-00902: invalid datatype
```

Alternatively, if your datatypes were correct, an exception might be thrown in case your database size goes over space quota and is unable to construct a new table. `SQLWarnings` can be retrieved from `Connection` objects, `Statement` objects, and `ResultSet` objects. Each only stores the most recent `SQLWarning`. So if you execute another statement through your `Statement` object, for instance, any earlier warnings will be discarded. Here is a code snippet which illustrates the use of `SQLWarnings`:

```
ResultSet rs = stmt.executeQuery("SELECT bar FROM Sells") ;
SQLWarning warn = stmt.getWarnings() ;
if (warn != null)
   System.out.println("Message: " + warn.getMessage()) ;
SQLWarning warning = rs.getWarnings() ;
if (warning != null)
   warning = warning.getNextWarning() ;
```

```
if (warning != null)
   System.out.println("Message: " + warn.getMessage()) ;
```

`SQLWarnings` (as opposed to `SQLExceptions`) are actually rather rare -- the most common is a `DataTruncation` warning. The latter indicates that there was a problem while reading or writing data from the database.

Sample Code and Compilation Instructions

Hopefully, by now you are familiar enough with JDBC to write serious code. Here is a simple program which ties all the ideas in the tutorial together.

We have a few more pieces of sample code written by Craig Jurney at ITSS for educational purposes. Feel free to use sample code as a guideline or even a skeleton for code that you write in the future, but make a note that you were basing your solution on provided code.

SQLBuilder.java - Creation of a Relation

SQLLoader.java - Insertion of Tuples

SQLRunner.java - Processes Queries

SQLUpdater.java - Updating Tuples

SQLBatchUpdater.java - Batch Updating

SQLUtil.java - JDBC Utility Functions

Don't forget to use `source /usr/class/cs145/all.env`, which will correctly set your classpath. By adding this to your global classpath you simplify commands. For example, you can say:

```
elaine19:~$ javac SQLBuilder.java
elaine19:~$ java SQLBuilder
```

instead of:

```
elaine19:~$ javac SQLBuilder.java
elaine19:~$ java -classpath /usr/pubsw/apps/oracle/8.1.5/jdbc/lib/classes111.
```

There are static final values in each of the .java files for USERNAME and PASSWORD. These must be changed to your own username and your own password so that you can access the database.

This document was written originally by Nathan Folkert for Prof. Jennifer Widom's CS145 class, Spring 2000. Subsequently, it was hacked by Mayank Bawa for Prof. Jeff Ullman's CS145 class, Fall 2000. Jim Zhuang made a minor update for Summer 2005. Thanks to Matt Laue for typo correction.