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 on a database system from a Java program. This document draws from the official Sun tutorial on JDBC Basics.

Overview

Call-level interfaces such as JDBC are programming interfaces allowing external programs to access SQL databases. They allow the execution of SQL commands within 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. For your convenience, all of the code for this article is included in the accessDatabase.java file.

Establishing A Connection

As we said earlier, before a database can be accessed, a connection must be opened between our Java 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 JDBC API was designed to be as independent of the version or the vendor of a database as possible. The differences between different DBMS's are encapsulated by each DBMS's driver, and we need to tell the Java DriverManager the correct driver to load. A MySQL driver is loaded using the following code:

  ```
  Class.forName("com.mysql.jdbc.Driver");
  ```

- **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:mysql://localhost:3306/DatabaseName", username, passwd
);
```

Okay, let's see what this jargon is. The first string is the URL for the database including the protocol (jdbc), the vendor (mysql), and the server port (//localhost:3306/) and your database instance name (DatabaseName). Of course, you need to replace DatabaseName with the name of your database. The username and passwd are your username and password, the same as you would enter into MySQL 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.

Creating JDBC Statements

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

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.

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 VARCHAR(40), beer VARCHAR(40), price REAL)"
);
stmt.executeUpdate(
    "INSERT INTO Sells VALUES('Bar Of Foo', 'BudLite', 2.00)"
);
String sqlString =
    "CREATE TABLE Bars(name VARCHAR(40), address VARCHAR(80), license INT)";
stmt.executeUpdate(sqlString);
```

Note here that we are reusing the same `Statement` object rather than having to create a new one.

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

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 VARCHAR 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);
```

Creating JDBC PreparedStatement

It is often safer or more efficient to use a `PreparedStatement` object for sending SQL statements to the DBMS, especially when the SQL statement needs to include input from users. The main feature which distinguishes it from its superclass `Statement` is that (1) it is given an SQL statement right when it is created and (2) the SQL statement can be parameterized, so that the same `PreparedStatement` can be repeatedly used for queries with different parameters that are provided by users.

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

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

This SQL statement is then sent to the DBMS *right away*, where it is compiled. 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.

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:

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

Setting parameters *separately* from the main SQL statement by explicitly specifying the their types makes `PreparedStatement` much safer than plain `Statement` in terms of security. In fact, if you need to use a user input as part of your SQL statement, it is an industry-wide convention that you ***MUST*** use `PreparedStatement`, not plain `Statement`, to protect your application against many attacks, such as SQL injection.

Once we finish plugging in the values of the parameters (as seen above), and we execute the statement by invoking the `executeUpdate` on it.

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

Similarly, if our `PreparedStatement` was created with a SQL SELECT statement, not an UPDATE statement, then we would execute such a statement by invoking the `executeQuery` on it, like the following:

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

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. Rollbacks 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 gives 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`:

```
try {
    stmt.executeUpdate(
        "CREATE TABLE Sells (bar VARCHAR(40), beer VARCHAR(40), price REAL)"
    );
    stmt.executeUpdate(
        "INSERT INTO Sells VALUES('Bar Of Foo', 'BudLite', 2.00)"
    );
} catch (SQLException ex) {
    System.err.println("SQLException: " + ex.getMessage());
}
```

In this case, an exception is thrown because beer is defined as VARCHAR which is a misspelling. Since there is no such data type in our DBMS, an `SQLException` is thrown. 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) System.out.println("Message: " + warning.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.

Importance of Closing Everything

When you are done with using your `Connection`, you need to explicitly close it by calling its `close()` method in order to release any other database resources (cursors, handles, etc) the connection may be holding on to.

Actually, the safe pattern in Java is to close your `ResultSet`, `Statement`, and `Connection` (in that order) in a finally block when you are done with them, something like that:

```
Connection conn = null;
PreparedStatement ps = null;
ResultSet rs = null;

try {
    // Do stuff
    ...

} catch (SQLException ex) {
    // Exception handling stuff
    ...
} finally {
    if (rs != null) {
        try {
            rs.close();
        } catch (SQLException e) { /* ignored */}
    }
    if (ps != null) {
        try {
            ps.close();
        } catch (SQLException e) { /* ignored */}
    }
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) { /* ignored */}
    }
}
```

The finally block can be slightly improved into (to avoid the null check):

```
} finally {
    try { rs.close(); } catch (Exception e) { /* ignored */ }
    try { ps.close(); } catch (Exception e) { /* ignored */ }
    try { conn.close(); } catch (Exception e) { /* ignored */ }
}
```

Finishing Remarks

Hopefully, by now you are familiar enough with JDBC to write serious code. Here is the accessDatabase.java which provides the code that ties all the ideas in the tutorial together. The program connects to the database "CS144" using the username "cs144" with empty password.

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. This document was most recently edited by Junghoo Cho for CS144 class in Spring 2007.