Database Connectivity: JDBC

Database Connecitivity Basics

Application-level database connectivity:

- Host language (Java, C/C++, Perl, PHP, etc)
- Target DBMS (Oracle, MS SQL server, IBM DB2, MySQL, etc)
- Client — Server environment
  - Client: application program
  - Server: DBMS

General structure:

- Load database driver/database support functionality
- form an SQL statement
- connect to the DBMS
- pass SQL statement to the DBMS
- recieve result
- close connection

JDBC

JDBC originally, an abbreviation for Java Database Connectivity is the database connectivity package for Java.

JDBC driver

On CSL, JDBC driver is located in the

```
/app/oracle/product/10.2.0/client_1/jdbc/lib
```

directory (i.e., it is part of the Oracle software package).

To successfully compile Java programs using JDBC, this directory needs to be added to your `CLASSPATH` environment variable.

in bash the command is:

```
export CLASSPATH=/app/oracle/product/10.2.0/client_1/jdbc/lib/\*.jar
```

in tcsh/csh the command is:

```
setenv CLASSPATH /app/oracle/product/10.2.0/client_1/jdbc/lib/\*.jar
```

JDBC Package

The JDBC package name is **java.sql**. When writing Java applications, which include JDBC connectivity, add to your import section

```
import java.sql.*
```

Loading the database driver

First task in any JDBC application is loading the JDBC driver for the right DBMS. This is achieved using `Class.forName(String Name)` method. The argument passed to the `Class.forName` method is the name of the JDBC driver for a specific DBMS server.

DBMS	Driver name
Oracle	`oracle.jdbc.driver.OracleDriver`
MySQL	`com.mysql.jdbc.driver`
Microsoft SQL Server	`com.microsoft.jdbc.sqlserver.SQLServerDriver`
IBM DB2	`COM.ibm.db2.jdbc.app.DB2Driver`

`Class.forName()` invokation must be inside Java's `try–catch` block.

Example. The following code loads Oracle's JDBC driver or, if unsuccessful, reports an error.

```
try{
    Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch (ClassNotFoundException ex)
{
    System.out.println("Driver not found");
};
```

Establishing a Connection

JDBC package contains a `Connection` class representing client-server connections between the client Java applications and the DBMS servers. An instance of the `Connection` class will be created via the following driver manager call:

```
Connection conn = DriverManager.getConnection(url, userId, password);
```

Here,

url is a connection URL specifying location and connection port for the database. See below for syntax.

userId is the DBMS user login account.

password is the password for the DBMS account of the user `userId`.

Connection URL

Connection url has the following syntax:

```
<driver>:<dbms>:<connection-type>@<host>:<port>:<server-name>
```

For CSL machines, use the following line value for the connection url:

```
jdbc:oracle:thin:@hercules.csc.calpoly.edu:1522:ora10g
```

Example. The following code establishes the connection to our Oracle server for one of the student accounts:

```
Connection conn = null;
String url = "jdbc:oracle:thin:@hercules.csc.calpoly.edu:1522:ora10g";
String user ="ST44";
String password="empty";
try {
      conn = DriverManager.getConnection(url, user, password);
     }
      catch (Exception ex)
    {
      System.out.println("Could not open connection");
    };
```

Statements

Work with a `Connection` object within a Java program is straghtforward: SQL statements are created and passed via the connection, results are recieved in return. There are three classes for SQL statements:

Statement: general use statement class. Used for creation and execution of SQL statements, typically, once during the run of the program.

PreparedStatement: statement class to be used in the following cases:

- an sequence of similar SQL statements, different only in values of some parameters needs to be executed;
- a single time-consuming SQL statement needs to be executed, possibly multiple times.

SQL statements represented by instances of `PreparedStatement` class are pre-compiled and thus may be more efficiently executed.

CallableStatement: statement class for execution of stored SQL (PL/SQL) procedures.

Instances of each class are obtained by invoking (see below) methods from the `Connection` class.

JDBC distinguishes two types of SQL statements:

Non-Query SQL statements: All DDL and DML statements, which do NOT return relational tables.

Queries: SELECT statements and their combinations, which return relational tables.

Because queries return tables while non-queries return only exit status, different methods are used to pass these two types of SQL statements.

Class Statement

Obtaining Instances. Instances of class `Statement` can be obtained by invoking the `createStatement()` method of the `Connection` class:

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

Executing non-queries. Non-queries (a.k.a. updates) are executed using method `executeUpdate()` of class `Statement`. While, several method signatures exist, the method call to be used under most stanard circumstances is:

```
int executeUpdate(String sql) throws SQLException
```

The method returns the number of rows affected by the update, 0 if a DDL statement (CREATE TABLE, etc.) was executed.

For example, the following sequence executes two statements: a table `Employees` is created and a record is inserted into it.

```
 String sql = "CREATE TABLE Employee" +
              "(Id INT PRIMARY KEY, " +
              "Name CHAR(30), Salary INT)";
try{
 s.executeUpdate(sql);
 s.executeUpdate("INSERT INTO Employee VALUES(1,'John Smith', 30000");
    } catch (SQLException e) { }
```

Note, that the `Statement` instance is **reusable**. Generally speaking, in order to execute a sequence of SQL statements, you only need to create one `Statement` instance.

Executing SQL queries. Use method `executeQuery()` of class `Statement`. The method returns an instance of the class `ResultSet`, which is discussed below.

```
try{
 ResultSet result = s.executeQuery("SELECT * FROM Employees WHERE Name='Jones'");
    } catch (SQLException e) { }
```

Class PreparedStatement

`PreparedStatement` should be used when the same query with possibly different parameters is to be executed multiple times in the course of a program.

Instances of `PreparedStatement` are created with an SQL statement, possibly with parameter placeholders associated with them, and that association cannot change.

Obtaining instances. Instances of class `Statement` can be obtained by invoking the `preparedStatement()` method of the `Connection` class:

```
String sql = "INSERT INTO Employee VALUES(2, 'Bob Brown', 40000)";
PreparedStatement s = conn.createStatement(sql);
```

This code creates a prepared SQL statement associated with the SQL statement `INSERT INTO Employee VALUES(2, 'Bob Brown', 40000)`.

Paremeterized Prepared Statements. The text of the SQL code for the `PreparedStatement` instance can contain `'?'` symbols: one symbol per input parameter to the query. For example, in order to create a generic parameterized `INSERT` statement for the `Employee` table, we can do the following:

```
String sql1 = "INSERT INTO Employee VALUES(?,?,?)";
PreparedStatement s1 = conn.createStatement(sql1);
```

`createStatement()` method parses the input SQL string and identifies locations of all parameters. Each parameter will get a number (starting with 1).

Setting Parameter Values. `PreparedStatement` uses the following methods to set values for parameters (note: all methods are `void` and throw `SQLException`):

Method	Explanation
`setNull(int parIndex, int jdbcType)`	sets parameter `parIndex` to null
`setBoolean(int parIndex, boolean x)`	sets parameter `parIndex` to a boolean value `x`
`setByte(int parIndex, byte x)`	sets parameter `parIndex` to a byte value `x`
`setInt(int parIndex, int x)`	sets parameter `parIndex` to an integet value `x`
`setLong(int parIndex, long x)`	sets parameter `parIndex` to a long integer value `x`
`setFloat(int parIndex, float x)`	sets parameter `parIndex` to a floating point value `x`
`setDouble(int parIndex, double x)`	sets parameter `parIndex` to a double precision value `x`
`setString(int parIndex, String x)`	sets parameter `parIndex` to a string value `x`
`setDate(int parIndex, java.sql.Date x)`	sets parameter `parIndex` to a date value `x`
`setTime(int parIndex, java.sql.Time x)`	sets parameter `parIndex` to a time value `x`
`clearParameters()`	clears the values of all parameters

Executing non-query statements. `PreparedStatement` class has the `executeUpdate()` method to execute non-query SQL statements. This method takes no input arguments (since the SQL statement is already prepared).

The following example shows how parameters are set up and prepared updates are executed:

```
try{
     s1.setInt(1,3);                //set first column value for the INSERT statement (ID)
     s1.setString('Mary Williams'); //set second column value  (Name)
     s1.setInt(45000);              //set third column value (Salary)

     s1.executeUpdate();           //execute INSERT INTO Employee VALUES(3,'Mary Williams,45000')
    } catch (SQLException e) {}
```

Executing queries. To execute queries, use `executeQuery()` method, which also does not take any arguments. This method returns an instance of `ResultSet`.

The following code fragment shows the preparation and execution of SELECT statements which select rows of the `Employee` table by salary.

```
try{
  string sql2 = "SELECT * FROM Employee WHERE Salary > ?";
  PreparedStatement s2 = conn.prepareStatement(sql2);

  s2.setString(1,35000);
  ResultSet result = s2.executeQuery();  //execute SELECT * FROM Employee WHERE Salary > 35000

  s2.clearParameters();                  // clear all parameters

  s2.setString(1,42000);
  result = s2.executeQuery();  //execute SELECT * FROM Employee WHERE Salary > 42000
   } catch (SQLException e) {}
```

Working with output: Class ResultSet

Results of SELECT statements (and other SQL statements that return tables) are stored in instances of the `ResultSet` class.

An instance of the `ResultSet` maintains a *cursor* which points to the currently observed record (tuple) in the returned table. The following methods can be used to navigate a `ResultSet` object:

Method	Explanation
`boolean next()`	move cursor to the next record.
`boolean previous()`	move cursor to the previous record.
`boolean first()`	move cursor in front of the first record.
`boolean last()`	move cursor to the last row of the cursor.
`boolean absolute(int row)`	move cursor to the record number `row`.
`boolrsn relative(int rows)`	move cursor `rows` records from the current position.
`boolean isLast()`	`true` if the cursor is on the last row.
`void close()`	close the cursor, release JDBC resources.
`boolean wasNull()`	`true` if the last column read had `null` value
`int findColumn(String columnName)`	returns the column number given the name of the column

In addition to these methods, a collection of `get` methods is associated with `ResultSet` class. Each `get` method retrieves one value from the current record (tuple) in the cursor. There are two families of `get` methods: one family retrieves values by column number, the other — by column name.

get by Column Number	get by Column Name	Explanation
`String getString(int colIndex)`	`String getString(String colName)`	retrieve a string value
`boolean getBoolean(int colIndex)`	`boolean getBoolean(String colName)`	retrieve a boolean value
`byte getByte(int colIndex)`	`byte getByte(String colName)`	retrieve a byte value
`short getShort(int colIndex)`	`short getShort(String colName)`	retrieve a short integer value
`int getInt(int colIndex)`	`int getInt(String colName)`	retrieve an integer value
`float getFloat(int colIndex)`	`float getFloat(String colName)`	retrieve a floating point value
`double getDouble(int colIndex)`	`double getDouble(String colName)`	retrieve a double precision value
`java.sql.Date getDate(int colIndex)`	`java.sql.Date getDate(String colName)`	retrieve a string value

Example. The code fragment below prints out the values of the `Name` column from the `Employee` table returned from the query.

```
Statement query = conn.createStatement();
ResultSet result = query.executeQuery("SELECT * FROM Employee WHERE Salary > 27000");

boolean f = result.next();      // original position of the cursor - before first record
while (f)
 {
    String s = result.getString("Name");
    System.out.println(s);
    f=result.next();
 }
```

Types of ResultSet instances

`ResultSet` instances can be of one of three types:

- `TYPE_FORWARD_ONLY`: the result set is non-scrollable, the cursor can be moved using only the `next()` and `last()` methods (no methods that go back can be used) (default).
- `TYPE_SCROLL_INSENSITIVE`: the result set is scrollable, i.e., the cursor can be moved both forward (`next()`, `last()`) **and backwards** (`previous()`, `first()`). Also, the result set typically does not change in response to changes in the database.
- `TYPE_SCROLL_SENSITIVE`: the result set is scrollable, i.e., the cursor can be moved both forward (`next()`, `last()`) **and backwards** (`previous()`, `first()`). Also, the result set typically changes if the data in the underlying database changes.

In addition, the result set may, or may not be updatable. This is controlled by the concurrency setting:

- `CONCUR_READ_ONLY`: the result set is read-only, no programmatic updates are allowed (default).
- `CONCUR_UPDATABLE`: the result set can be updated programmatically.

The type of the `ResultSet` instance to be returned by `executeQuery()` statements can be selected at the creation time of the SQL statement object:

- Class `Statement`: default result set type set by the `createStatement()` method is `TYPE_FORWARD_ONLY`, and the concurrency setting is `CONCUR_READ_ONLY`.

  To create a statement with a different type of result set use createStatement(int Scrollable, int Concur)

Here, `Scrollable` is the scrollability type (one of `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, `TYPE_SCROLL_SENSITIVE`) and `Concur` is the concurrency setting (one of `CONCUR_READ_ONLY`, `CONCUR_UPDATABLE`)[1].

JDBC Types

JDBC package contains a number of JDBC type constants that are used to pass to the DBMS server the information about the type of various arguments. All constant declarations are made as as below:

```
public static final int NULL = 0;
```

The full list of types names, their ids and the appropriate Java types (for type conversion purposes) is below:

JDBC type	Constant Value	matching Java Type
NULL	0	N/A
OTHER	1111	N/A
BIT	-7	boolean
TINYINT	-6	byte
SMALLINT	5	short
INTEGER	4	int
BIGINT	-5	long
FLOAT	6	double
REAL	7	float
DOUBLE	8	double
CHAR	1	String
VARCHAR	12	String
LONGVARCHAR	-1	String
DATE	91	java.sql.Date
TIME	92	java.sql.Time
TIMESTAMP	93	java.sql.Timestamp
BINARY	-2	byte[]
VARBINARY	-3	byte[]
LONGVARBINARY	-4	byte[]
JAVA_OBJECT	2000	underlying Java class
DISTINCT	2001	N/A
STRUCT	2002	Struct
ARRAY	2003	Array
BLOB	2004	Blob
CLOB	2005	Clob
REF	2006	Ref
DATALINK	70	java.net.URL
BOOLEAN	16	boolean

[1] For the purposes of this course, we can live with the default concurrency setting, and do not even need to know about it. However, `Connection` class does not have a version of `createStatement()` that sets only the scrollability setting, hence, a brief description of the second argument is needed.