Java Interface to Databases (JDBC)

Table of Content:
1 Basic Principles
2 Installing Connection
3 Working with a Database

1 Basic Principles
Basically, the JDBC API is a software library providing a number of Java classes and interfaces that allows programmers to:

- Establish a connection to a wide range of different Relational Database Management Systems (RDBMS). Note, that each major RDBMS, such as Oracle, Microsoft SQL, or MySQL is supported.

- Access a database within a particular system.

- Manipulate the structure of a database by creating, deleting, or altering relations from that database.

- Manipulate the content of a database by inserting, deleting, or updating records in database relations.

- Retrieve the content from a database by querying the database.

Basically, the JDBC operates with two main classes:

- **DriverManager class** operates with a library of drivers for different DBMS implementations. The DriverManager class loads requested drivers, physically installs connection to a database and return an instance of a data class "Connection".

- An **instance of the class "Connection"** represent a single connection to a particular database. All the communication to the database is carryed out via this object.
2 Installing Connection

Establishing JDBC Connection means obtaining a correct instance of so-called "Connection" class. Usually, establishing a connection is carried out in two steps:

- loading an appropriate JDBC driver for the installed RDBMS.
- installing connection and getting reference to the Connection object

Loading a JDBC driver is very simple and involves just a single line of Java code.

```
... try {
    Class.forName("com.mysql.jdbc.Driver");
} catch(ClassNotFoundException exc){exc.printStackTrace();}
...
```

This line of code just notifies the DriverManager which particular Java class should be loaded as a JDBC driver class.

Please recollect that almost any modern DBMS supports JDBC. Primitively speaking, there are JDBC drivers for each implementation of DBMS. For example, we can load JDBC driver for MySQL DBMS.

```
... try {
    Class.forName("com.mysql.jdbc.Driver");
} catch(ClassNotFoundException exc){exc.printStackTrace();}
...
```
The next step in establishing a database connection is a message to loaded driver requesting actual connection to the RDBMS. The operation is carried out by sending message "getConnection" to the driver manager. Note that "DriverManager" returns a "Connection" instance that is used for further processing the database.

```java
try {
    Connection connection_;
    String dbms = "jdbc:mysql://" + host + "/" + db;
    connection_ = DriverManager.getConnection(dbms, username, password);
}
```

Method "getConnection()" accepts three arguments:

3. a so-called Database URL, which encoded using standard URL syntax (protocol + host + object). The protocol part starts always with "jdbc:" followed by the name of the RDBMS (in our case "mysql") and terminated with "/" symbols. Thus, the protocol part in our example is "jdbc:mysql://". The host part identifies a server where the DBMS is running. In our case (Servlets & DBMS on the same computer) "localhost" can be used to identify the host. Finally, the name of a particular database must be supplied proceeded with the slash character. In our case this would be "/example".

4. A registered username that has the proper privileges for manipulating the database.

5. A password valid for the username.

3 Working with a Database
In order to actually work with a database, a special "Statement" class is used. In order to create an instance of such "Statement" class, a message "createStatement" is sent to the previously created instance of JDBC connection.

```java
try {
    Statement statement = connection_.createStatement();
}
catch(SQLException exc) {
    exc.printStackTrace();
}
```
If an error occurs during the execution of the createStatement() method a SQLException will be thrown. Instances of the **Statement Class** provides a public interface to insert, update, or retrieve data from a database. Depending on a particular database operation, an appropriate method should be invoked. For instance,

- **executeUpdate()** can be used to insert data into a relation
- **executeQuery()** can be used to retrieve data from a database

Instances of the Statement Class provides a public interface to insert, update, or retrieve data from a database. Depending on a particular database operation, an appropriate method should be invoked. For instance,

- **executeUpdate()** can be used to insert data into a relation
- **executeQuery()** can be used to retrieve data from a database

```java
try {
    String insert_sql_stmt = "INSERT INTO " + table + " VALUES(" + values + ");
    statement.executeUpdate(insert_sql_stmt);
} catch(SQLException exc){exc.printStackTrace();} ...
```

Other methods of the "statement" class can be also applied to its instances. **Attention!** A programmer should notify the instance about intention to use any other method by setting parameters of the "executeStatement" message. For example, if we need to retrieve the keys automatically generated by the "executeUpdate" statement, we need to pass the "Statement.RETURN_GENERATED_KEYS" argument in advance.

```java
try {
    String sql = "INSERT INTO " + table + " VALUES(" + values + ");
    statement.executeUpdate(sql, Statement.RETURN_GENERATED_KEYS);
    ResultSet keys = statement.getGeneratedKeys();
} catch(SQLException exc){exc.printStackTrace();} ...
```

Similarly, to retrieve data from a database we need to obtain an instance of the Statement class, and then to invoke **executeQuery()** method on this instance. This method takes a string containing SQL source as an argument.

```java
try {
    String sql = "SELECT ...";
    ResultSet query_result = statement.executeQuery(sql);
} ...
```
Note, that the "sql" argument should contain a valid SQL Select statement. The `executeQuery()` method returns an instance of the `ResultSet` class. Generally, execution of any JDBC statement that returns data from a database, results in an instance of the `ResultSet` class. This instances may be seen as a number of rows (tuples) that hold the current results. The number and type of columns in this object corresponds to the number and types of columns returned as the result from the database system.

Consider the following sample database:

```java
Customer(cn, cname, ccity);
Product(pn, pname, pprice);
Transaction(cn, pn, tdate, tqnt);
```

```java
try {
    String sql = "SELECT * FROM Customer;";
    ResultSet query_result = statement.executeQuery(sql);
}
```

The "executeQuery" command will result in obtaining an instance of the `ResultSet` class which will hold all tuples from the Customer table as rows, each row will contain 3 values: "cn", "cname" and "ccity". Normally, the SQL statement explicitly defines the "ResultSet" internal structure.

Consider the following sample database:

```java
Customer(cn, cname, ccity);
Product(pn, pname, pprice);
Transaction(cn, pn, tdate, tqnt);
```

```java
try {
    String sql = "SELECT cname, pname, qnt;";
    sql = sql + " FROM Customer, Product, Transaction;"
    sql = sql + " where Customer.ccity = "Graz" And";
    sql = sql + " Customer.cn = Transaction.cn And";
    sql = sql + " Transaction.pn = Product.pn";
    ResultSet query_result = statement.executeQuery(sql);
}
```

The "executeQuery" command will result in obtaining an instance of the `ResultSet` class populated with a number of rows. Each row contains 3 values: "cname", "pname" and "qnt". Basically, an instance of the `ResultSet` class is an iterator over the rows it keeps. There is always the current row, and we can obtain only the data from the current row. If we want to move the cursor to the next row we need to invoke the `next()` method. At the beginning, the current row is set before the first row of the result, hence before obtaining data from the first row, the `next()` method should be invoked.
```java
try {
    String sql = "SELECT cname, pname, qnt"
    sql = sql + " FROM Customer, Product, Transaction"
    sql = sql + " where Customer.ccity = "Graz" And"
    sql = sql + " Customer.cn = Transaction.cn And"
    sql = sql + " Transaction.pn = Product.pn"
    ResultSet query_result = statement.executeQuery(sql);
    while(query_result.next()) {
        ...
    }
}
```

Finally, you may access the online application, which provides a Web interface to access and manipulate the context of the "example" database at [http://coronet.iicm.edu/shop/](http://coronet.iicm.edu/shop/). Note, that this Web application is implemented by means of the Java Servlet technology.

Normally, the SQL statement explicitly defines the "ResultSet" internal structure.

Consider the following sample database:

- **Customer(cn,cname,ccity)**
- **Product(pn,pname,pprice)**
- **Transaction(cn,pn,tdate,tqnt)**

The "executeQuery" may contain parameters received via an HTTP GET request.

```java
response.setContentType("text/html");
PrintWriter writer = response.getWriter();
String customerCity = request.getParameter("customerCity");
try {
    String sql = "SELECT cname, pname, qnt"
    sql = sql + " FROM Customer, Product, Transaction"
    sql = sql + " where Customer.ccity = " + customerCity + " And"
    sql = sql + " Customer.cn = Transaction.cn And"
    sql = sql + " Transaction.pn = Product.pn"
    ResultSet query_result = statement.executeQuery(sql);
    while(query_result.next()) {
        ...
    }
}
```

To comprehend the basic principles of "SQL Injection", suppose the user pass a string like this:

`1',1','1'); DELETE FROM Transaction;` as a value for the parameter customerID (cn). Obviously the string insert_sql will look as:

`INSERT INTO Customer VALUES('1','1','1'); DELETE FROM Transaction; '...` ...

```java
response.setContentType("text/html");
PrintWriter writer = response.getWriter();
String cn = request.getParameter("customerID");
String customerName = request.getParameter("customerName");
String customerCity = request.getParameter("customerCity");
```
try
  {String insert_sql = "INSERT INTO Customer ";
insert_sql += "VALUES('" + cn + "," + customerName + "," + customerCity + ")"; 
statement = connection.createStatement();
statement.executeUpdate(insert_sql);
...

All the tuples of the relation "Transaction" will be deleted!

To prevent such SQL injection, so-called prepared statements can be used. In this, 
case, SQL query is not defined as a string containing the source text. SQL query is 
a prepared statement with a number of parameters that can be set at run-time.
...

Consider the following sample database:
Customer(cn, cname, ccity);
Product(pn, pname, pprice);
Transaction(cn, pn, tdate, tqnt);
...

response.setContentType("text/html");
PrintWriter writer = response.getWriter();
String cn = request.getParameter("customerID");
String customerName = request.getParameter("customerName");
String customerCity = request.getParameter("customerCity");
try
  {String insert_sql = "INSERT INTO Customer ":
insert_sql += "VALUES(?, ?, ?)";
PreparedStatement statement =
connection.prepareStatement(insert_sql);
statement.setString(1, cn);
statement.setString(2, customerName);
statement.setString(3, customerCity);
statement.executeUpdate();

Once when we set a current row of the ResultSet, we can retrieve values by means 
of a number of methods. The methods correspond to a column type. Thus, to 
retrieve the value of a string column, we invoke a getString() method. Similarly, 
to retrieve an integer value we simply invoke a getInt() method.
...

response.setContentType("text/html");
PrintWriter writer = response.getWriter();
String customerCity = request.getParameter("customerCity");
try {String sql = "SELECT cname, pname, qnt"
sql = sql + " FROM Customer, Product, Transaction"
sql = sql + " where Customer.ccity = " + customerCity + " And"
sql = sql + " Customer.cn = Transaction.cn And"
sql = sql + " Transaction.pn = Product.pn"
ResultSet query_result = statement.executeQuery(sql);
while(query_result.next())
  { String customerName = query_result.getString("cname");
  String productTitle = query_result.getString("pname");
  int productQuantity = query_result.getInt("qnt"); ...}
...

Once when we set a current row of the ResultSet, we can retrieve values by 
means of a number of methods. The methods correspond to a column type. Thus, to retrieve the value of a string column, we invoke a getString() method. Similarly, to retrieve an integer value we simply invoke a getInt() method.
try {
    String sql = "SELECT cname, pname, qnt";
    sql = sql + " FROM Customer, Product, Transaction";
    sql = sql + " where Customer.ccity = "Graz" And";
    sql = sql + " Customer.cn = Transaction.cn And";
    sql = sql + " Transaction.pn = Product.pn";
    ResultSet query_result = statement.executeQuery(sql);
    while(query_result.next())
    {
        String customerName = query_result.getString("cname");
        String productTitle = query_result.getString("pname");
        int productQuantity = query_result.getInt("cid");
        ...
    }
    ...
}