Acesso a banco de dados através de JDBC

Prof. Pasteur Ottoni de Miranda Junior

Disponível em www.pasteurjr.blogspot.com

4.1-Configurações iniciais

Para rodar JDBC em sua máquina faça:

1-Instalar Java e JDBC. Faça o download e instale a última versão do JDK (Java Development Kit): obtém-se tanto o Java como o JDBC.
2-Instalar um driver. Cada driver tem suas instruções específicas de instalação. Por exemplo, os drivers do Oracle, do MySQL, do Interbase, etc.
3-Instalar o DBMS.
4- Criar um banco de dados chamado COFFEEBREAK.

No cabeçalho de cada arquivo Java onde haverá acesso a banco de dados, o seguinte Import deve ser colocado:

import java.sql.*;

Após a declaração de qualquer método onde haja acesso a banco de dados, as seguintes exceções devem ser levantadas;

throws SQLException, ClassNotFoundException

4.2-Estabelecendo uma conexão

4.2.1-Carregando Drivers

Carregar o driver a ser utilizado é muito simples, por exemplo:

Para Oracle:

```
DriverManager.registerDriver( new oracle.jdbc.driver.OracleDriver());
ou ainda Class.forName("oracle.jdbc.driver.OracleDriver");
```

Para Interbase:

```
DriverManager.registerDriver(new interbase.interclient.Driver()); ou
ainda Class.forName("interbase.interclient.Driver ");
```

Para MySQL:

```
DriverManager.registerDriver(new org.gjt.mm.mysql.Driver ()); ou ainda
Class.forName("org.gjt.mm.mysql.Driver");
```

Se seu driver não for nenhum dos acima, consulte a documentação dele. Se por exemplo o nome da classe for `jdbc.DriverXYZ` , você deve carregar o driver com a seguinte linha de código:

```
Class.forName("jdbc.DriverXYZ");
```

4.2.2-Fazendo a conexão

O segundo passo em estabelecer uma conexão é conectar o driver apropriadamente ao DBMS. A seguinte linha de código ilustra a idéia geral:

```
Connection con = DriverManager.getConnection(url,
                    "myLogin", "myPassword");
```

`url` é o caminho par acesso ao driver. Temos:

Para Oracle:

```
"jdbc:oracle:thin:@endereço IP:PORTA:NOMEBANCO "
```

Por exemplo:

```
"jdbc:oracle:thin:@192.168.2.6:1521:DBSA"
```

Para Interbase:

```
"jdbc:interbase://ENDEREÇO IP DO HOST/CAMINHO DIRETORIO DO BANCO
/NOMEDOARQUIVO GDB DO INTERBASE"
```

Por exemplo:

```
"jdbc:interbase://localhost/C:/diversos/projetointerclass/gdb/cnw.gdb"
```

Para MySQL:

```
"jdbc:mysql://endereço ip do host/nome do banco"
```

Por exemplo:

```
"jdbc:mysql://200.238.233.20/raw"
```

`myLogin e myPassword` são nome de usuário e senha respectivamente.

4.3- Configurando tabelas

Criar a seguinte tabela: COFFEES (cafés)

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

4.3.1-Criando comandos JDBC

O objeto `Statement` é o responsável por enviar o comando SQL ao DBMS. Basta criar um objeto `Statement` e executá-lo, fonecendo ao método apropriado o comando SQL a ser enviado. Para um comando `SELECT,` o método a ser utilizado é o `executeQuery` . Para comandos que criam ou modificam tabelas, o método a ser utilizado é o `executeUpdate` .

É necessária uma instância de um objeto connection ativo para criar um objeto `Statement`. No exemplo a seguir, usamos o objeto `con` da classe `Connection` para criar o objeto `stmt` da classe `Statement`:

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

Neste ponto, `stmt` existe, mas não tem um comando SQL para passar ao DBMS. Para criar, por exemplo, a tabela COFFEE , usamos `executeUpdate` passando o comando SQL CREATE TABLE como parâmetro:

```
stmt.executeUpdate("CREATE TABLE COFFEES " +
    "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +
    "SALES INTEGER, TOTAL INTEGER)");
```

4.3.2-Executando comandos

Usamos o método `executeUpdate` porque o comando SQL passado como parâmetro é uma DDL (data definition language). Comandos que criam, alteram, atualizam ou excluem uma tabela são exemplos de DDL e são executados com o método `executeUpdate` . O método mais frequentemente utilizado para executar comandos SQL é o `executeQuery` . Este método é utilizado para executar comandos SELECT.

4.3.3– Inserindo dados numa tabela

Vamos inserir dados na tabela COFFEE criada anteriormente utilizando o método executeUpdate e o comando DDL INSERT:

```
Statement stmt = con.createStatement();
stmt.executeUpdate(
    "INSERT INTO COFFEES " +
    "VALUES ('Colombian', 101, 7.99, 0, 0)");
```

Os comandos seguintes inserem mais valores

```
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast', 49, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO COFFEES " +
    "VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

4.3.4-Recuperando valores da tabela

O JDBC retorna resultados de um SELECT em um objeto a `ResultSet`, assim precisamos declarar uma instância da classe `ResultSet` para conter nossos resultados. O código a seguir mostra a declaração de um objeto `ResultSet rs` e a atribuição do resultado da query no mesmo:

```
ResultSet rs = stmt.executeQuery(
    "SELECT COF_NAME, PRICE FROM COFFEES");
```

4.3.5-Usando o método next

A variável `rs`, que é uma instância de `ResultSet`, contém o resultado da pesquisa realizada com o SELECT. Para acessar os campos da tabela COFFEE temos que navegar por cada registro e recuperar os campos de acordo com seus respectivos tipos. Isto pode ser feito através dos métodos `next` de `ResultSet`. Ele move o chamado cursor para o próximo registro e o torna apto a ser manipulado. Uma vez que o cursor está inicialmente posicionado acima do primeiro registro, a primeira chamada ao método `next` move o cursor para o primeiro registro. Chamadas sucessivas do método `next` movem o cursor do topo para o fim da lista de registros recuperados. O método `next` retorna também um valor booleano que, enquanto houver registros a serem percorridos, retorna true. Quando o final da lista de registros é atingido, retorna false.

4.3.6 –Usando os métodos getXXX

Os métodos `getXXX` pertencentes à classe ResultSet são utilizados para recuperar o valor de cada coluna da tabela. Por exemplo, a primeira coluna da tabela COFFEE é `COF_NAME`, que guarda um valor do tipo `VARCHAR` . O método para recuperar um `VARCHAR` é `getString`. A segunda coluna é um `FLOAT` , cujo método para recuperar é `getFloat` . O seguinte código acessa os valores guardados no registro atual de `rs` e imprime uma linha com o nome seguido por 3 espaços e o preço. Cada vez que o método `next` é invocado, a próxima linha se torna a atual, e o loop continua até que não haja mais registros em `rs`.

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String s = rs.getString("COF_NAME");
    float n = rs.getFloat("PRICE");
    System.out.println(s + "   " + n);
}
```

A saída será a seguinte:

```
Colombian   7.99
French_Roast   8.99
Espresso   9.99
Colombian_Decaf   8.99
French_Roast_Decaf   9.99
```

O comando

```
String s = rs.getString("COF_NAME");
```

recupera em rs, via o método `getString`, o valor do campo COF_NAME. A situação é similar com o método `getFloat`.

Existem duas formas de identificar a coluna a ser recuperada pelo método `getXXX`. Uma forma é passando o nome do campo como parâmetro, conforme mostrado acima. A outra consiste em dar o índice da coluna, sendo 1 para o primeiro campo, 2 para o segundo e assim por diante, como abaixo:

```
String s = rs.getString(1);
float n = rs.getFloat(2);
```

A tabela abaixo mostra quais métodos podem ser legalmente usados para recuperar tipos SQL e, mais importante, quais métodos são recomendados para recuperar os vários tipos

SQL. O método getString pode ser utilizado para recuperar qualquer tipo SQL. Obviamente este retorno deve ser feito em uma variável string.

Tabela – Uso dos métodos ResultSet.getXXX para recuperar tipos JDBC

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x
getBytes														X	X	x			
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimestamp											x	x	x				x	x	X
getAsciiStream											x	x	X	x	x	x			
getUnicodeStream											x	x	X	x	x	x			
getBinaryStream														x	x	X			
getObject	x	x	x	x	X	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Um "x" indica que o método `getXXX` pode legalmente ser utilizado para recuperar o tipo JDBC.

Um " *X* " indica que o método `getXXX` é recomendado para recuperar o tipo JDBC.

4.3.7-Atualizando tabelas

Para modificar registros de uma tabela, utilizamos o comando UPDATE do SQL, via o método executeUpdate, como no exemplo abaixo:

```
String updateString = "UPDATE COFFEES " +
        "SET SALES = 75 " +
        "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
```

Para selecionar o registro alterado, recuperar os valores das colunas `COF_NAME` and `SALES`, e imprimi-los:

```
String query = "SELECT COF_NAME, SALES FROM COFFEES " +
               "WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
               String s = rs.getString("COF_NAME");
               int n = rs.getInt("SALES");
               System.out.println(n + " pounds of " + s +
                                                  " sold this week.");
}
```

Isto imprime o seguinte:

```
75 pounds of Colombian sold this week.
```

Para atualizar a coluna `TOTAL`, pela adição da quantidade semanal e depois imprimi-la, fazemos:

```
String updateString = "UPDATE COFFEES " +
                      "SET TOTAL = TOTAL + 75 " +
                      "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
String query = "SELECT COF_NAME, TOTAL FROM COFFEES " +
                                    "WHERE COF_NAME LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
               String s = rs.getString(1);
               int n = rs.getInt(2);
               System.out.println(n + " pounds of " + s + " sold to
date.");
}
```

Note que neste exemplo utilizamos o índice da coluna (1=COF_NAME, 2=TOTAL).

4.4 Usando Comandos preparados

Um objeto PreparedStatement pode ser utilizado para enviar comandos a uma base de dados. Ele é derivado de statement. Se um objeto `Statement` vai ser usado muitas vezes, a utilização de um objeto `PreparedStatement` reduz o tempo de resposta, porque ele é precompilado, ou seja, o comando SQL não precisa ser compilado, apenas rodado.

4.4.1- Criando um objeto PreparedStatement

Tal qual com objetos `Statement`, objetos `PreparedStatement` são criados a partir de um método de `Connection` :

```
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
```

As interrogações contêm valores a serem passados como parâmetro, antes da execução do PreparedStatement. Para tal , utilizam-se os métodos `setXXX` da classe PreparedStatemet, onde XXX corresponde ao tipo a ser fornecido. Se for inteiro, temos para o exemplo acima:

```
updateSales.setInt(1, 75);
```

O primeiro parâmetro corresponde ao índice do valor a ser fornecido para substituir as interrogações. O segundo corresponde ao valor. Para substituir a segunda interrogação usamos:

```
updateSales.setString(2, "Colombian");
```

Depois de fornecidos os valores das interrogações, executa-se o comando através do método executeUpdate sem parâmetro:

```
PreparedStatement updateSales = con.prepareStatement(
        "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ? ");
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate():
```

Repare que utilizamos o comando UPDATE com interrogações e com PreparedStatement porque desta forma podemos diminuir o tempo para inserção, já que nas próximas inserções o comando SQL já terá sido compilado. Assim sendo, basta executá-lo nas vezes seguintes passando os novos valores a serem atualizados.

4.4.2-Usando um Loop para ajustar valores

O fragmento de código a seguir demonstra o uso de um loop `for` para ajustar valores de parâmetros no objeto `PreparedStatement`.

```
PreparedStatement updateSales;
String updateString = "update COFFEES " +
                      "set SALES = ? where COF_NAME like ?";
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
                     "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
                updateSales.setInt(1, salesForWeek[i]);
                updateSales.setString(2, coffees[i]);
                updateSales.executeUpdate();
        }
```

4.4.3-Retorno de valores do método executeUpdate

O valor de retorno do método executeUpdate é um inteiro correspondente ao número de linhas que foram atualizadas.

4.5-Utilizando Stored Procedures

4.5.1-Criando uma Stored Procedure

Vamos criar a seguinte Stored Procedure:

```
create procedure SHOW_COFFES
as
select COFFEES.COF_NAME
from  COFFEES
where order by COF_NAME
```

As linhas seguintes criam a Stored Procedure via método executeUpdate já visto:

```
String createProcedure = "create procedure SHOW_SUPPLIERS " +
                         "as " +
                         "select COFFEES.COF_NAME " +
```

```
                          "from COFFEES "
                           +
                          "order by COF_NAME";
Statement stmt = con.createStatement();
stmt.executeUpdate(createProcedure);
```

A stored procedure `SHOW_SUPPLIERS` será compilada e guardada no banco de dados.

3.5.2-Chamando uma Stored Procedure

O primeiro passo é criar um objeto `CallableStatement,` o qual é também criado com um objeto Connection já aberto. A primeira linha de código abaixo cria uma chamada à stored procedure `SHOW_SUPPLIERS` usando a conexão `con` . A chamada é feita colocando-se o call + nome da stored procedure entre chaves, como mostrado abaixo:

```
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

Repare que utilizamos o método executeQuery , pois o retorno da stored procedure provém de um comando SELECT.

4.6-Utilizando transações

4.6.1-Desabilitando o modo Auto-commit

Quando uma conexão é criada, por default ela está no modo auto-commit, ou seja, cada comando SQL é tratado como uma transação individual e será automaticamente submetido (commited) após sua execução. Para permitir que mais comandos SQL sejam agrupados em uma única transação deve-se desabilitar o modo auto-commit, como mostrado abaixo:

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

4.6.2- Submetendo(commiting) uma transação

Se o modo auto-commit for desabilitado, nenhum comando SQL será submetido até que a chamada ao método commit seja explicitamente feita. Todos os comandos realizados antes do commit serão incluídos na transação atual e submetidos. O exemplo a seguir ilustra isto:

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
```

```
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

Neste exemplo, o modo auto-commit é desabilitado para a conexão `con`, o que significa que os dois comandos preparados `updateSales` and `updateTotal` serão submetidos (commited) juntos quando o método commit for chamado. Quando o método commit for chamado, todas as alterações efetuadas no banco de dados serão feitas permanentes..

A última linha retorna a conexão ao modo auto-commit.

4.6.3-Usando o método rollback

Se houver algum problema com a transação, pode-se invocar o método rollback, que retorna o banco de dados ao estado anterior ao da execução dos comandos dentro da transação. Este é um método de Connection, invocado normalmente dentro de blocos catch, uma vez que os comandos da transação tenham sido colocados dentro de blocos try.

```
con.rollback();
```