



Curso de Java

aula 12

Prof. Anderson Henrique

JDBC – (Java Database Connectivity)

É a tecnologia usada pelos programas Java para se comunicarem com os bancos de dados. E não existe sistema corporativo sem um banco de dados.

A maioria das empresas armazena dados em vários arquivos. P.ex.: arquivos de folhas de pagamento, contas a receber, contas a pagar, inventário e outros.

Um grupo organizado de arquivos relacionados é o que chamamos de Banco de Dados.

E os programas projetados para criar e gerenciar banco de dados são chamados de SGDB's.

Os bancos de dados mais populares são os relacionais, e a linguagem utilizada para incluir, alterar, excluir e listar informações em um banco de dados relacional é o SQL (Structure Query Language) pré-requisito.

Um banco de dados relacional armazena dados em tabelas, essas tabelas por sua vez são compostas de linhas (registros) e colunas (atributos).

SGDB's (Sistema de Gerenciamento de Banco de Dados)








Os programas Java se comunicam e manipulam informações de um banco de dados através do driver JDBC. O JDBC nada mais é do que uma biblioteca Java para se conectar com o banco de dados (db).

Todo gerenciador de banco de dados fornece um drive JDBC para que as nossas aplicações possam interagir com aquele banco. Utilizaremos o PostgreSQL no nosso exemplo a seguir:

Banco de Dados

PESSOA

 CODIGO (PK)	NUMBER(38,0)
 NOME	VARCHAR2(100)
 SEXO	CHAR(1)
 EMAIL	VARCHAR2(150)
 DATA_NASCIMENTO	DATE

Exceto o campo data_nascimento, deixaremos de fora da tabela.

Utilizando o drive JDBC

Crie um novo projeto Java com o nome aula13, crie um pacote nesse projeto com o nome: br.com.treinacom.java.jdbc, crie uma classe Java principal com o nome: AcessoBanco.

```
public class AcessoBanco{  
    public static void main(String[ ] args) throws Exception{  
        String url = "jdbc:postgresql://localhost:5432/pessoa";  
        String user = "postgres";  
        String pass = "root";  
        String driver = "org.postgresql.Driver";
```

```
String sql = "select * from pessoa";  
Class.forName(driver).newInstance();  
try(Connection conn = DriverManager.getConnection(url, user, pass)){  
    PreparedStatement stm = conn.prepareStatement(sql);  
    ResultSet rs = stm.executeQuery( ){  
        while(rs.next( ){  
            String s = rs.getString("codigo")  
            + "; " + rs.getString("nome")  
            + "; " + rs.getString("sexo")  
            + "; " + rs.getString("email");  
            System.out.println(s);  
        }  
    }  
}
```


JDBC – Insert e Batch (inserir)

Conexão com Bancos de Dados via JDBC

DB2	<code>jdbc:db2:servidor:porta/banco</code>
Oracle	<code>jdbc:oracle:thin:@servidor:porta:banco</code>
PostgreSQL	<code>jdbc:postgresql://servidor:porta/banco</code>
SQLServer	<code>jdbc:sqlserver://servidor:porta;database=banco</code>
MySQL	<code>jdbc:mysql://servidor:porta/banco</code>
Derby	<code>jdbc:derby://servidor:porta/banco</code>
SyBase	<code>jdbc:Sybase:Tds:servidor:porta/banco</code>

Vamos Inserir informações no Banco de Dados, crie uma classe Java principal chamada IncluirDados.

```
public class IncluirDados{  
    public static void main(String[ ] args) throws Exception{  
        Class.forName("org.postgresql.Driver");  
        String sql = "insert into pessoa values (5, 'Caio', 'M',  
'caio@gmail.com', '2006-06-25')";  
        String url = "jdbc:postgresql://localhost:5432/postgres";  
        try(Connection conn = DriverManager.getConnection(url,  
"postgres", "root")){  
            PreparedStatement stm = conn.prepareStatement(sql)){  
                stm.executeUpdate( );  
            }  
        }  
    }  
}
```

```
        }  
    }  
}
```

Este programa serve apenas para inserir um registro por vez, sendo que o método `prepareStatement` permite preparar diversas instruções sql.

Vamos alterar o código, no lugar dos valores a serem inseridos, utilizaremos uma interrogação .

Ex.:

```
String sql = "insert into pessoa values (?, ?, ?, ?)";
```

Vamos criar um array de String com diversas pessoas.

```
String[ ] pessoas = {"Sandra", "Beatriz", "Juliana", "Fatima", "Veranda"};
```

Dentro do bloco **try**, vamos criar um laço que percorrerá todo o array, inserindo as pessoas na tabela do nosso banco.

```
stm.setInt(1, i + npessoas);  
stm.setString(2, pessoas[i]);  
stm.setString(3, "F");  
stm.setString(4, pessoas[i].toLowerCase( )+"@gmail.com");  
stm.addBatch( );  
stm.executeBatch( );
```

Incluímos um conjunto de requisições a serem executadas uma única vez, utilizando o método **addBatch()**, e no final executamos utilizando o método **executeBatch()**, poupando assim recursos do nosso computador (aproveitando a mesma conexão com banco de dados).

Agora, vamos recuperar todos os registros aproveitando a mesma conexão com o banco de dados, pois, um dos recursos mais caros em termo de consumo e de tempo é estabelecer conexões com bancos de dados.

Vamos remover o nosso PreparedStatement do nosso bloco **try** e inserir em outro bloco **try** interno.

```
try(Connection conn = DriverManager.getConnection(url, "postgres",  
"root")){  
    try(PreparedStatement stm = conn.prepareStatement(sql)){  
        stm.setInt(1, i + npessoas);  
        stm.setString(2, pessoas[i]);  
        stm.setString(3, "F");  
        stm.setString(4, pessoas[i].toLowerCase( )+"@gmail.com");  
        stm.addBatch( );  
        stm.executeBatch( );  
    } catch(SQLException e){ }
```

```
String sql2 = "select nome, email from pessoa";  
try(PreparedStatement stm2 = conn.prepareStatement(sql2);  
ResultSet rs = stm2.executeQuery( )){  
    while(rs.next( )){  
        System.out.println(rs.getString(1) + ":" + rs.getString(2));  
    }  
}  
}
```

JDBC – CRUD (Create, Read, Update e Delete)

Vamos criar a classe que vai realizar as operações (DML) no nosso banco de dados. Crie uma classe java chamada Conta e ContaCRUD, no mesmo pacote Java.

Ex.:

```
public class Conta{  
    int numero;  
    String cliente;  
    double saldo;
```

Crie o método **construtor** da classe Conta, que permite atribuir valores as propriedades.

Ex.:

```
public Conta(int numero, String cliente, double saldo){  
    this.numero = numero;  
    this.cliente = cliente;  
    this.saldo = saldo;  
}
```

Faça o @Override do método toString, que permitirá imprimir as propriedades do objeto Conta.

Ex.:

@Override

```
public String toString( ){  
    return numero + ", " + cliente + ", " + saldo;  
}
```

Com a classe **Conta** finalizada, vamos trabalhar na classe **ContaCRUD**. Implementar o método que permite inserir registros na tabela do banco.

```
public void criar(Connection conn, Conta conta) throws SQLException{  
    String sql = "insert into conta values (?, ?, ?)";  
    try(PreparedStatement stm = conn.prepareStatement(sql)){  
        stm.setInt(1, conta.numero);  
        stm.setString(2, conta.cliente);  
        stm.setDouble(3, conta.saldo);  
        stm.executeUpdate( );  
    }  
}
```

Implementar o método que permite ler os dados da tabela do banco

```
public List<Conta> ler(Connection conn) throws SQLException{  
    List<Conta> lista = new ArrayList<>( );  
    String sql = "select numero, cliente, saldo from conta";  
    try(PreparedStatement stm = conn.prepareStatement(sql);  
        ResultSet rs = stm.executeQuery( )){  
        while(rs.next( )){  
            lista.add(new Conta(rs.getInt(1), rs.getString(2),  
rs.getDouble(3)));  
        }  
    }  
    return lista;  
}
```

Agora, vamos implementar o método que altera um registro na tabela do banco

```
public void alterar(Connection conn, Conta conta) throws SQLException{  
    String sql = "update conta set cliente=?, saldo=? where numero =?";  
    try(PreparedStatement stm = conn.prepareStatement(sql)){  
        stm.setString(1, conta.cliente);  
        stm.setDouble(2, conta.saldo);  
        stm.setInt(3, conta.numero);  
        stm.executeUpdate( );  
    }  
}
```

Por último, vamos implementar o método que permite excluir um registro na tabela do banco.

```
public void excluir(Connection conn, Conta conta) throws SQLException{  
    String sql = "delete from conta where numero=?";  
    try(PreparedStatement stm = conn.prepareStatement(sql)){  
        stm.setInt(1, conta.numero);  
        stm.executeUpdate( );  
    }  
}
```

Agora vamos testar todos os métodos que foram criados no nosso método principal da classe.

```
public static void main(String[ ] args){  
    Class.forName("org.postgresql.Driver");  
    String url = "jdbc:postgresql://localhost:5432/postgres";  
    try(Connection conn = DriverManager.getConnection(url, "postgres",  
"root")){  
        ContaCRUD crud = new ContaCRUD( );  
        Conta conta1 = new Conta(1, "Luana", 1_000.00);  
        Conta conta2 = new Conta(2, "Paulo", 3_000.00);  
        Conta conta3 = new Conta(1, "Joana", 5_000.00);  
  
        crud.criar(conn, conta1); //insere registro  
        crud.criar(conn, conta2);  
        crud.criar(conn, conta3);  
    }  
}
```

```
conta1.saldo = 9_000.00;  
crud.alterar(conn, conta1); //alterar registro
```

```
crud.excluir(conn, conta3); ;;excluir registro
```

```
List<Conta> contas = crud.ler(conn); //listar registros  
for(Conta conta : contas){  
    System.out.println(conta);  
}
```

```
}
```

```
}
```

```
}
```

JDBC – Transation, Commit, Rollback

Aprendemos as rotinas de criar, ler, atualizar e excluir, mas agora precisamos criar um método para transferir dinheiro de uma conta para outra.

Precisamos verificar se a conta de origem tem o saldo para realizar a transferência, e se tiver você deverá retirar o dinheiro da conta origem e depositar na conta destino.

Mas imagine! Se a transferência falhar depois de retirar o dinheiro da conta de origem? Para resolver este tipo de problema é que existe a transação.

Uma transação entende que o conjunto de operações de banco de dados deve ser tratada como se fosse uma única operação. E se uma dessas operações falhar, então todas as outras serão desfeitas e a transação deve ser revertida.

Mas, se todas as operações forem bem sucedidas, então a transação deve ser confirmada.

Vamos criar um método de transferência de valores entre contas.

Ex.:

```
public void transferir(Connection conn, Conta origem, Conta destino, double valor) throws SQLException {  
    if(origem.saldo >= valor){  
        try{  
            conn.setAutoCommit(false); //trata operações  
            origem.saldo -= valor;  
            alterar(conn, origem);  
        }  
    }  
}
```

```
destino.saldo += valor;  
alterar(conn, destino);  
conn.commit( ); //confirma transação  
}catch(Exception e){  
    conn.rollback( ); //desfaz transação  
}  
}  
}
```

Vamos testar este método no nosso método **main** (principal).

Prosseguiremos no próximo slide... Com JEE (JSP e Servlets)

Professor: Anderson Henrique

Programador nas Linguagens Java e PHP

