

53 45 52 45 49 20 46 49 45 4c 20
41 4f 53 20 50 52 45 43 45 49 54
4f 53 20 44 41 20 48 4f 4e 52 41
20 45 20 44 41 20 43 49 c3 8a 4e
43 49 41 2c 20 50 52 4f 4d 4f 56
45 4e 44 4f 20 4f 20 55 53 4f 20
45 20 4f 20 44 45 53 45 4e 56 4f
4c 56 49 4d 45 4e 54 4f 20 44 41
20 49 4e 46 4f 52 4d c3 81 54 49
43 41 20 45 4d 20 42 45 4e 45 46
c3 8d 43 49 4f 20 44 4f 20 43 49
44 41 44 c3 83 4f 20 45 20 44 41
20 53 4f 43 49 45 44 41 44 45 2e

RESIDÊNCIA DE SOFTWARE

**CAPACITAR
TREINAR
EMPREGAR**

TRANSFORMAR



JAVA I

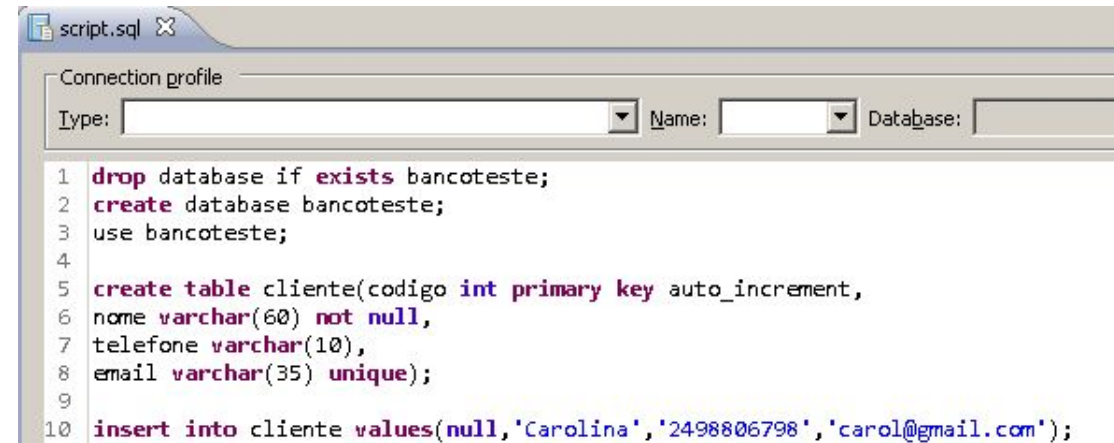
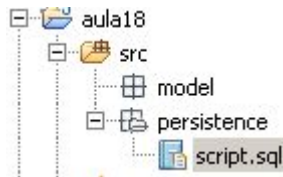
JDBC

12/08/2020

O JDBC é uma API escrita em Java que serve como uma ponte entre nossos programas e o banco de dados, foi desenvolvida com a intenção de padronizar o acesso a diferentes bancos de dados, dando maior flexibilidade aos sistemas. A biblioteca da JDBC localizada no pacote **java.sql** provê um conjunto de **interfaces**. Para implementar essas interfaces precisamos de classes concretas, que irão fazer a ponte entre o código cliente que usa a API JDBC e o banco de dados. Esse conjunto de classes recebe o nome de **driver**. A implementação das classes fica por conta do fabricante do banco de dados.

Criando o banco de dados no mysql e a tabela como exemplo

- Criar um novo projeto no eclipse
- Criar um pacote com o nome model e outro com o nome persistence
- Criar o script sql: File-New-Other-SQL Development-SQLFile



```
1 drop database if exists bancoteste;
2 create database bancoteste;
3 use bancoteste;
4
5 create table cliente(codigo int primary key auto_increment,
6 nome varchar(60) not null,
7 telefone varchar(10),
8 email varchar(35) unique);
9
10 insert into cliente values(null,'Carolina','2498806798','carol@gmail.com');
```

Instalar o driver para conexão do mysql

baixar do link <http://www.mysql.com/downloads/connector/j/>

Descompactar o arquivo com driver e copiar o arquivo **.jar** para o diretório workspace do eclipse para dentro da pasta aula17

Criar a conexão com o banco

```
ConnectionFactory.java
package persistence;

import java.sql.Connection;

public class ConnectionFactory {
    String urlConexao = "jdbc:mysql://localhost:3306/bancoteste";
    String usuario = "root";
    String senha = "";
    Connection connection;

    public Connection getConnection() {
        System.out.println("Conectando ao banco");
        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection = DriverManager
                .getConnection(urlConexao, usuario, senha);

            if (connection != null) {
                System.out.println("Conectado ao banco !!");
            } else {
                System.out.println("Não foi possível conectar");
            }
            return connection;
        } catch (ClassNotFoundException e) {
            System.out.println("Driver não encontrado");
            return null;
        }

        catch (SQLException e) {
            System.out.println("Não foi possível conectar");
            return null;
        }
    }
}
```

importar do pacote **java.sql**

- Endereço IP, porta e nome da base de dados
- Usuário do banco
- Senha do usuário.

A classe responsável pela criação de uma conexão JDBC é a **DriverManager** do pacote **java.sql**.

A url de conexão, o usuário e a senha devem ser passados ao método **getConnection()** para que ele possa retornar uma conexão. Uma exceção do tipo **SQLException** é repassada por **getConnection** por isto temos que tratar com **try/catch**. Nas versões anteriores ao Java 6 é necessário incluir a linha **Class.forName("com.mysql.jdbc.Driver")** uma exceção do tipo **ClassNotFoundException** será lançada.

Design patterns são padrões utilizados em sistemas para melhorar a organização interna do código e facilitar sua manutenção e extensão. O pattern **Factory** implementa uma fábrica de objetos, abstraindo e isolando o modo de criação dos objetos. A classe **ConnectionFactory** implementa o pattern **Factory**.

Criar a classe **TestaConexao**

```
TestaConexao.java
package persistence;

import java.sql.Connection;

public class TestaConexao {
    public static void main(String[] args) {
        Connection connection = new ConnectionFactory().getConnection();
    }
}
```

Ao executar o código recebemos a seguinte mensagem:

```
java.lang.RuntimeException: java.sql.SQLException: No suitable driver found for com.mysql.jdbc.Driver
```



No Eclipse clicar com o botão direito no **.jar Build Path-Add to Build Path**

Build Path

Add to Build Path

DAO (Data Access Object)

O DAO é um design pattern para acesso a dados com todas as características para acesso e manipulação de um banco de dados. Geralmente, temos um DAO para cada objeto do domínio do sistema como por exemplo Pessoa, Produto Cliente, e outros.

Criar a classe **Cliente** no pacote **model**

```
Cliente.java ✕
package model;
public class Cliente {
    private Integer codigo;
    private String nome;
    private String telefone;
    private String email;

    public Cliente() {
    }

    public Cliente(Integer codigo, String nome, String telefone, String email) {
        this.codigo = codigo;
        this.nome = nome;
        this.telefone = telefone;
        this.email = email;
    }

    public Integer getCodigo() {
        return codigo;
    }

    public String getNome() {
        return nome;
    }

    public String getTelefone() {
        return telefone;
    }

    public String getEmail() {
        return email;
    }
}
```

Classe Java Beans

Uma classe é considerada **Java Beans** quando possuem o construtor sem argumentos e os métodos getters e setters!

Criar a classe **ClienteDao** no pacote **persistence**

ClienteDao.java

Não esquecer dos imports

```
package persistence;

import java.sql.Connection;

public class ClienteDao {
    private Connection connection;

    public ClienteDao() {
        connection = new ConnectionFactory().getConnection();
    }

    public void adiciona(Cliente cliente) {
        String sql = "insert into cliente values(null,?,?,?)";
        try {
            PreparedStatement stmt = connection.prepareStatement(sql);
            stmt.setString(1, cliente.getNome());
            stmt.setString(2, cliente.getTelefone());
            stmt.setString(3, cliente.getEmail());

            stmt.execute();
            stmt.close();
            connection.close();
        }

        catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

As cláusulas são executadas em um banco de dados através da interface **PreparedStatement**.

Para receber um **PreparedStatement** relativo à conexão, basta chamar o método **prepareStatement**, passando como argumento o comando SQL com os valores vindos de variáveis preenchidos com uma interrogação.

Os comandos **Select**, **Insert**, **Update** e **Delete** tem uma parte fixa e uma parte variável. No exemplo:

insert	into	cliente	values
(null,"Carla","98986787","abcd@abcd.com.br");			

a parte que varia acima são os dados.

A maior parte dos bancos de dados SQL trabalha melhor se, em vez de ficarmos montando sempre consultas SQL diferentes, passarmos uma consulta FIXA e variar só os dados.

O **PreparedStatement** sempre entende:

insert into cliente values(null,?,?,?)

O símbolo de **?** pode ser os dados de Carla, Maria, João ou quem precisarmos inserir. Os dados que variam, não o nome das colunas.

Os parâmetros foram definidos através do caractere **"?"**. Antes de executar a query, é necessário determinar os valores dos parâmetros. Essa tarefa pode ser realizada através do método **setString()**, que recebe a posição do parâmetro que começa com 1 no código SQL e o valor correspondente do parâmetro. Temos outros métodos como **setBoolean**, **setInt**, **setDouble** para cada tipo de dados.

Criar o pacote **control** e a classe **ControleCliente** dentro deste pacote

```
ControleCliente.java ✕
package control;

import persistence.ClienteDao;

public class ControleCliente {

    public void adicionaCliente(Cliente cliente) {
        ClienteDao dao = new ClienteDao();
        dao.adiciona(cliente);
        System.out.println("Cliente Gravado com Sucesso");
    }
}
```

O control define o comportamento da aplicação, é ele que interpreta as ações do usuário (View) e as mapeia para chamadas do modelo (Model).

Criar a classe **TestaCliente** no pacote **persistence** e fazer a inserção do cliente no banco

```
*TestaCliente.java ✕
package persistence;

import control.ControleCliente;

public class TestaCliente {
    public static void main(String[] args) {
        ControleCliente controleCliente = new ControleCliente();
        Cliente cliente = new Cliente(1, "XPTO LTDA", "2231-0990",
            "xpto@xpto.com.br");
        controleCliente.adicionaCliente(cliente);
    }
}
```

Adicionar o método alterar em ClienteDao

```
public void alterar(Cliente cliente) {
    String sql = "update cliente set nome=?,telefone=?,email=? where codigo=?";
    try {

        PreparedStatement stmt = connection.prepareStatement(sql);
        stmt.setString(1, cliente.getNome());
        stmt.setString(2, cliente.getTelefone());
        stmt.setString(3, cliente.getEmail());
        stmt.setInt(4, cliente.getCodigo());
        stmt.execute();
        stmt.close();
        connection.close();

    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

Adicionar o método alterarCliente em ControleCliente

```
public void alterarCliente(Cliente cliente) {
    ClienteDao dao = new ClienteDao();
    dao.alterar(cliente);
    System.out.println("Cliente Alterado com Sucesso");
}
```

Alterar o método TestaCliente

```
TestaCliente.java
package persistence;

import control.ControleCliente;

public class TestaCliente {
    public static void main(String[] args) {
        ControleCliente controleCliente = new ControleCliente();

        Cliente cliente = new Cliente(1, "ABC LTDA", "2221-1090",
            "abc@abc.com.br");
        controleCliente.alterarCliente(cliente);
    }
}
```


Adicionar o método apagar em `ClienteDao`

```
public void apagar(int codigo) {  
  
    String sql = "delete from cliente where codigo=?";  
    try {  
        PreparedStatement stmt = connection.prepareStatement(sql);  
        stmt.setInt(1, codigo);  
        stmt.execute();  
        stmt.close();  
        connection.close();  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
}
```

Alterar o método `TestaCliente`

```
TestaCliente.java ✕  
package persistence;  
  
import control.ControleCliente;  
  
public class TestaCliente {  
    public static void main(String[] args) {  
        ControleCliente controleCliente = new ControleCliente();  
  
        Cliente cliente = new Cliente(1, "ABC LTDA", "2221-1090",  
            "abc@abc.com.br");  
  
        controleCliente.apagarCliente(cliente);  
    }  
}
```

Adicionar o método `Cliente` em `ControleCliente`

```
public void apagarCliente(Cliente cliente){  
    ClienteDao dao = new ClienteDao();  
    dao.apagar(cliente.getCodigo());  
    System.out.println("Cliente Removido com Sucesso");  
}
```


Listagem de Registro do Banco de Dados

O processo para executar um comando de consulta é bem parecido com o processo de inserir registros no banco. A diferença é que para executar um comando de consulta é necessário utilizar o método **executeQuery()** ao invés do **execute()**. Esse método devolve um objeto da interface **java.sql.ResultSet**, que é responsável por armazenar os resultados da consulta. Uma vez que você possui um **ResultSet**, você pode obter valores de qualquer campo na linha, ou mover para a próxima linha no conjunto. **ResultSets** são sempre posicionados antes da primeira linha se ela não for nula, portanto precisamos chamar **ResultSet.next()** para checar se foi retornado **true** para indicar que o **ResultSet** conseguiu avançar para o próximo registro ou **false** quando não existe mais linhas.

Adicionar o método lista em **ClienteDao**

```
public List<Cliente> listagem() {
    String sql = "select * from cliente";
    try {
        List<Cliente> lista = new ArrayList<Cliente>();
        PreparedStatement stmt = connection.prepareStatement(sql);
        ResultSet rs = stmt.executeQuery();

        while (rs.next()) {
            Cliente cliente = new Cliente();
            cliente.setCodigo(rs.getInt("codigo"));
            cliente.setNome(rs.getString("nome"));
            cliente.setTelefone(rs.getString("telefone"));
            cliente.setEmail(rs.getString("email"));
            lista.add(cliente);
        }
        rs.close();
        stmt.close();
        connection.close();
        return lista;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

- Varre o dados e cria o objeto **Cliente**
- Armazena os dados no objeto
- Adiciona objeto **Cliente** a lista
- Fecha conexão
- Retorna a lista

Os dados contidos no **ResultSet** podem ser acessados através de métodos, como o **getString**, **getInt**, **getDouble** e outros. Esses métodos recebem como parâmetro uma string referente ao nome da coluna correspondente. Os **ResultSets** representam as linhas retomadas como uma resposta a uma consulta.

Adicionar o método **listagemCliente** na classe **ControleCliente**

```
public List<Cliente> listagemCliente(){
    ClienteDao dao = new ClienteDao();
    return dao.listagem();
}
```

Alterar o método TestaCliente

```
*TestaCliente.java ✕
package persistence;

import control.ControlaCliente;

public class TestaCliente {
    public static void main(String[] args) {
        ControlaCliente controlaCliente = new ControlaCliente();

        for (Cliente cliente : controlaCliente.listagemCliente()) {
            System.out.println("Código:" + cliente.getCodigo());
            System.out.println("Nome:" + cliente.getNome());
            System.out.println("Telefone:" + cliente.getTelefone());
            System.out.println("Email:" + cliente.getEmail());
            System.out.println("-----");
        }
    }
}
```

Metadados são informações sobre os seus dados. Os metadados de uma tabela são: nome das colunas, tipo de dados das colunas (VARCHAR, NUMBER), tamanho da coluna, proprietário da tabela e outras informações.

Em algumas situações é necessário recuperar esses metadados para construirmos nossas consultas dinamicamente, pois em uma grande base de dados algumas mudanças estruturais podem ocorrer com certa frequência.

Metadados Jdbc

Vamos explicar abaixo as alterações necessárias para exibição dos metadados. Foi criado o método selectMetaData.

```
public void selectMetaData() {  
    try {  
        String sql = "select * from cliente";  
        PreparedStatement stmt = connection.prepareStatement(sql);  
        ResultSet rs = stmt.executeQuery();  
        ResultSetMetaData rsmd = rs.getMetaData();  
        rs.next();  
        int quantColunas = rsmd.getColumnCount();  
        System.out.println("Quant. de Colunas: " + quantColunas);  
        for (int i = 1; i <= quantColunas; i++) {  
            System.out.println("Tabela: " + rsmd.getTableName(i));  
            System.out.println("Coluna: " + rsmd.getColumnName(i));  
            System.out.println("Tipo: " + rsmd.getColumnTypeName(i));  
        }  
  
        rs.close();  
        stmt.close();  
        connection.close();  
    } catch (Exception e) {  
        throw new RuntimeException(e);  
    }  
}
```

Classe utilizada para recuperar mais metadados é a ResultSetMetaData

Retorna o nome da tabela, nome da coluna e o tipo.

```
}  
  
public static void main(String[] args) {  
    ClienteDao clienteDao = new ClienteDao();  
    clienteDao.selectMetaData();  
}
```

É uma expressão comumente utilizada definir as quatro operações básicas usadas em Banco de Dados Relacionais.

1) Exercício:

Criar o banco de dados no mysql e a tabela como exemplo

- Criar um novo projeto no eclipse
- Criar um pacote com o nome model e outro com o nome persistence
- Criar o script sql: File-New-Other-SQL Development-SQLFile
- Adicionar o conteúdo abaixo no arquivo de script

model
persistence

```
create database exercicios;  
use exercicios;  
  
create table conta (numero_conta int primary key, titular varchar(30), saldo double);  
  
insert into conta values(34349, 'Ana', 1000);  
insert into conta values(12434, 'Maria', 2000);  
insert into conta values(12434, 'Jorge', 2800);  
insert into conta values(12434, 'José', 4000);
```

- Abrir uma conexão com o mysql via terminal ou workbench e colar o script

Criar a conexão com o banco

importar do pacote java.sql

```
package persistence;

import java.sql.Connection;

public class ConnectionFactory {
    String urlConexao = "jdbc:mysql://localhost:3306/exercicios";
    String usuario = "root";
    String senha = "mysql";
    Connection connection;

    public Connection getConnection() {
        System.out.println("Conectando ao Banco");
        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection = DriverManager
                .getConnection(urlConexao, usuario, senha);
            if (connection != null) {
                System.out.println("Conectado ao Banco");
            } else {
                System.out.println("Não foi possível conexão com o Banco");
            }
        } catch (SQLException e) {
            System.out.println("Erro de Conexão: %s");
        } catch (ClassNotFoundException e) {
            System.out.println("Driver não encontrado");
        }
        return connection;
    }
}
```

```
package model;

public class Conta {
    private Integer numConta;
    private String titular;
    private Double saldo;

    public Conta() {
    }

    public Conta(Integer numConta, String titular, Double saldo) {
        super();
        this.numConta = numConta;
        this.titular = titular;
        this.saldo = saldo;
    }

    public Integer getNumConta() {
        return numConta;
    }

    public String getTitular() {
        return titular;
    }

    public Double getSaldo() {
        return saldo;
    }
}
```

Criar a classe **Conta** no pacote **model**
Inserir construtores
Inserir o Getters

Criar a classe **ContaDao** no pacote **persistence**

Criar o construtor e colocar a instância para abertura da conexão

```
public class ContaDao {  
    private Connection connection;  
  
    public ContaDao() {  
        connection = new ConnectionFactory().getConnection();  
    }  
  
    public void inserir (Conta conta) {  
        String sql = "insert into conta values(?,?,?)";  
        try {  
            PreparedStatement stmt = connection.prepareStatement(sql);  
            stmt.setInt(1, conta.getNumConta());  
            stmt.setString(2, conta.getTitular());  
            stmt.setDouble(3, 0.0);  
            stmt.execute();  
            stmt.close();  
        } catch (Exception e) {  
            throw new RuntimeException();  
        }  
    }  
}
```

Implementar o método **inserir**

Alteração de Registro do Banco de Dados

Adicionar o método alterar em [ContaDao](#)

```
public void saqueDeposito(Conta conta) {  
    String sql = "update conta set saldo=? where numero_conta=?";  
    try {  
        PreparedStatement stmt = connection.prepareStatement(sql);  
        stmt.setInt(1, conta.getNumConta());  
        stmt.setString(2, conta.getTitular());  
        stmt.setDouble(3, conta.getSaldo());  
        stmt.execute();  
        stmt.close();  
    } catch (Exception e) {  
        throw new RuntimeException();  
    }  
}
```

Adicionar o método apagar em [ContaDao](#)

```
public void apagar (Conta conta) {  
    String sql = "delete from conta where numero_conta=?";  
    try {  
        PreparedStatement stmt = connection.prepareStatement(sql);  
        stmt.setInt(1, conta.getNumConta());  
        stmt.execute();  
        stmt.close();  
    } catch (Exception e) {  
        throw new RuntimeException();  
    }  
}
```

Adicionar o método `listarContas` em `ContaDao`

```
public List<Conta> listarContas() {
    String sql = "select * from conta;";
    try {
        List<Conta> lista = new ArrayList<Conta>();
        PreparedStatement stmt = connection.prepareStatement(sql);
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            Conta conta = new Conta(rs.getInt("numero_conta"), rs.getString("titular"), rs.getDouble("saldo"));
            lista.add(conta);
            conta = new Conta();
        }
        stmt.close();
        rs.close();
        connection.close();
        return lista;
    } catch (Exception e) {
        throw new RuntimeException();
    }
}
```

Adicionar o método buscarConta em [ContaDao](#)

```
public Conta buscarConta(int numConta) {  
    String sql = "select * from conta where numero_conta=?";  
    try {  
        PreparedStatement stmt = connection.prepareStatement(sql);  
        stmt.setInt(1, numConta);  
        Conta conta;  
        ResultSet rs = stmt.executeQuery();  
        if (rs.next()) {  
            conta = new Conta(rs.getInt("numero_conta"),rs.getString("titular"),rs.getDouble("saldo"));  
        } else {  
            conta = null;  
        }  
        stmt.close();  
        rs.close();  
        connection.close();  
        return conta;  
    } catch (SQLException e) {  
        throw new RuntimeException();  
    }  
}
```