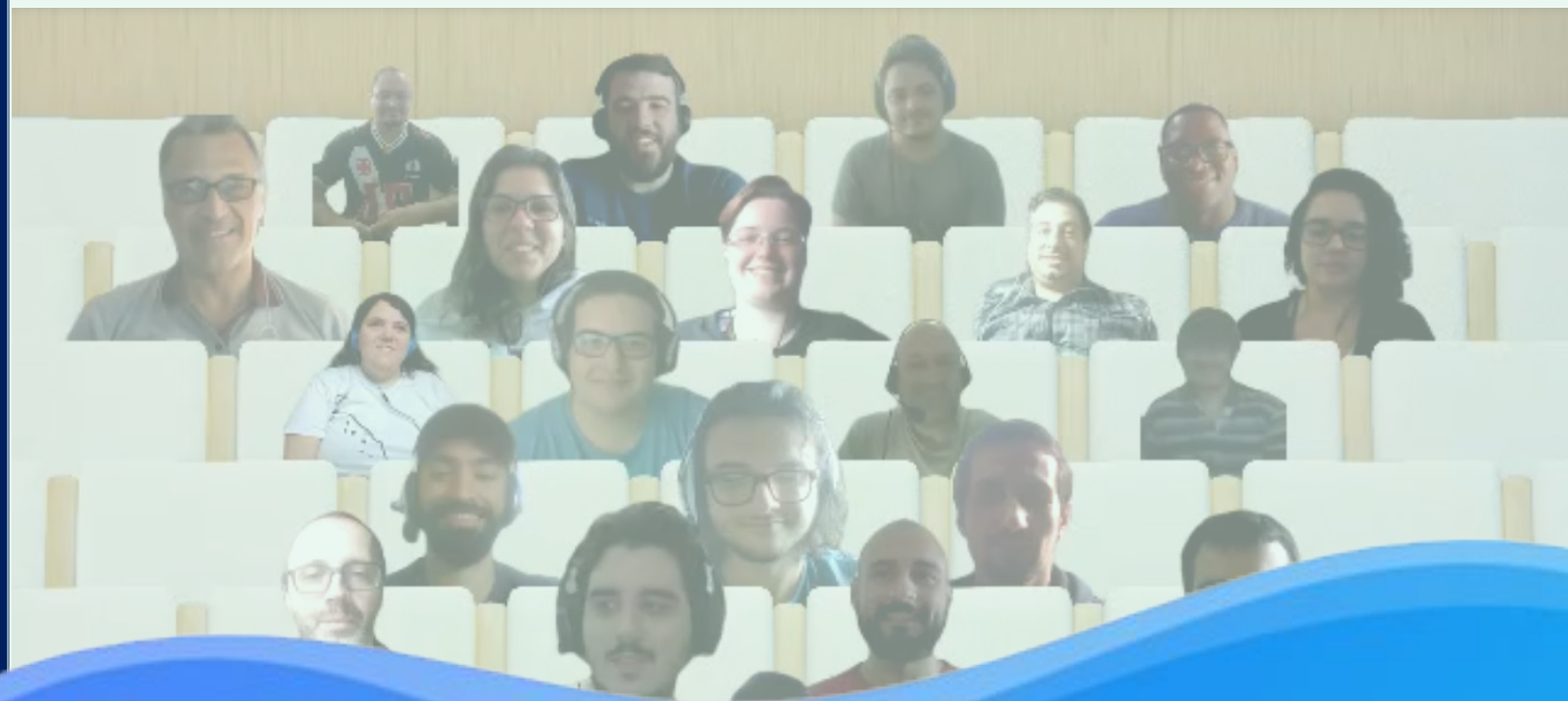


53 45 52 45 49 20 46 49 45 4c 20  
41 4f 53 20 50 52 45 43 45 49 54  
4f 53 20 44 41 20 48 4f 4e 52 41  
20 45 20 44 41 20 43 49 c3 8a 4e  
43 49 41 2c 20 50 52 4f 4d 4f 56  
45 4e 44 4f 20 4f 20 55 53 4f 20  
45 20 4f 20 44 45 53 45 4e 56 4f  
4c 56 49 4d 45 4e 54 4f 20 44 41  
20 49 4e 46 4f 52 4d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



Acessar um banco de dados a partir do backend  
Data: 27/04/2022

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 Postgres 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-File



```
CREATE TABLE cliente( codigo serial primary key, nome varchar(30),  
                        telefone varchar(11),  
                        email varchar(30) unique);
```

```
insert into cliente (nome,telefone,email) VALUES('Ana','2234-0909','ana@gmail.com');
```

# Instalar o driver para conexão do postgres

baixar do link <https://jdbc.postgresql.org/download.html>

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

## Criar a conexão com o banco

```
public class ConnectionFactory {  
    String url = "jdbc:postgresql://localhost:5432/aula";  
    String usuario = "postgres";  
    String senha = "postgres";  
  
    Connection connection;  
  
    public Connection getConnection() {  
        System.out.println("Conectando ao banco");  
        try {  
            connection = DriverManager.getConnection(url, usuario, senha);  
            if (connection != null) {  
                System.out.println("Conectado!");  
            } else {  
                System.out.println("Não foi possível");  
            }  
        } catch (SQLException e) {  
            System.out.println("Driver não encontrado");  
            return null;  
        }  
        return connection;  
    }  
}
```

]

- 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`.

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 um erro pois não colocamos o driver no path



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!



# DAO (Data Access Object)

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

```
public class ClienteDao {
    private Connection connection;

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

    public void inserir(Cliente cliente) {
        try {
            String sql = "insert into cliente (nome,telefone,email) values(?,?,?)";
            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 (Exception e) {
            System.out.println("Erro ao gravar !");
        }
    }
}
```

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 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 a classe `TestaCliente` no pacote `persistencia` e fazer a inserção do cliente no banco

```
public class TesteCliente {  
    public static void main(String[] args) {  
        Cliente cliente = new Cliente(null, "José", "23434", "jose@gmail.com");  
        ClienteDao dao = new ClienteDao();  
        dao.inserir(cliente);  
    }  
}
```

Adicionar o método alterar em `ClienteDao`

```
public void atualizar(Cliente cliente) {
    try {
        String sql = "update cliente set nome=?,telefone=?,email=? where codigo=?";
        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 (Exception e) {
        System.out.println("Erro ao gravar !");
    }
}
```

Alterar o método `TestaCliente`

```
public class TesteCliente {

    public static void main(String[] args) {
        Cliente cliente = new Cliente(2, "José Alves", "22451325", "josea@gmail.com");
        ClienteDao dao = new ClienteDao();
        dao.atualizar(cliente);
    }
}
```



Adicionar o método apagar em `ClienteDao`

```
public void remover(int codigo) {
    try {
        String sql = "delete from cliente where codigo=?";
        PreparedStatement stmt = connection.prepareStatement(sql);
        stmt.setInt(1, codigo);
        stmt.execute();
        stmt.close();
        connection.close();
    } catch (Exception e) {
        System.out.println("Erro ao gravar !");
    }
}
```

Alterar o método `TestaCliente`

```
public class TesteCliente {

    public static void main(String[] args) {
        ClienteDao dao = new ClienteDao();
        dao.remover(1);
    }
}
```

## 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 (Exception e) {  
        System.out.println("Erro ao listar os clientes !!");  
    }  
}
```

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.

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

### Listagem com like

```
public List<Cliente> listar(String nome) {  
    List<Cliente> clientes = new ArrayList<>();  
  
    try {  
        //String sql = "select * from cliente where nome like '" + nome + "%'";  
        //String sql = "select * from cliente where nome like '%" + nome + "%'";  
        String sql = "select * from cliente where nome like '%" + nome + "%'";  
        PreparedStatement stmt = connection.prepareStatement(sql);  
        ResultSet rs = stmt.executeQuery();  
        while (rs.next()) {  
            Cliente cliente = new Cliente(rs.getInt("codigo"), rs.getString("nome"), rs.getString("telefone"),  
                rs.getString("email"));  
            clientes.add(cliente);  
        }  
        stmt.close();  
        rs.close();  
    } catch (Exception e) {  
        System.out.println("Erro ao listar cliente !");  
    }  
    return clientes;  
}
```

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();  
}
```



Criar uma tabela no banco de dados com o nome conta com os seguintes campos:

- numero\_conta chave primária
- titular
- saldo

Criar o script para criação da tabela e inserir alguns registros na tabela.

Criar a classe Conta com os mesmos atributos da tabela.

Criar a classe ContaDao com os seguintes métodos:

- inserirConta
- saqueDeposito
- buscarConta
- listaContas
- removerConta