Análise e Desenvolvimento de Sistemas
Programação Java

Profª. Ana Luiza Scharf
ana.scharf@ifsc.edu.br

Laboratório: Exemplo 1 – Java com SQLite usando JDBC

Objetivo

Este laboratório apresenta os conceitos básicos para manipulação de banco de dados SQLite usando Java e JDBC 4. O aluno aprenderá a realizar consultas, inserções, alterações e remoções em um banco SQLite.

Pré-requisitos

- Java JDK instalado (versão 8 ou superior).
- IDE Java (Eclipse, IntelliJ, NetBeans ou outra).
- Driver JDBC para SQLite (SQLite JDBC Driver).
- Conhecimentos básicos em Java.

Estrutura Final de Arquivos

Antes de começarmos a implementar, confira a estrutura final do projeto que vamos construir passo a passo:

```
Projeto/
 src/
    main/
       java/
          bcd/
              Principal.java
       resources/
           application.properties
    test/
        java/bcd/teste/
 build.gradle
 settings.gradle
 README.md
```

Organizar assim facilita a manutenção e evita erros futuros.
Vamos implementar tudo junto, e você pode sempre comparar para garantir que está correto.

Passo 1: Criando o Projeto no IntelliJ IDEA

1. Abra o **IntelliJ IDEA**.
2. Clique em **New Project**.
3. No menu à esquerda, selecione a opção **Java**.
4. Preencha os seguintes campos:
   - **Name:** `Lab5`
   - **Location:** escolha o diretório desejado (ex: `arquivos`)
5. Em seguida, marque as seguintes opções:

- **Build system:** selecione **Gradle**
- **JDK:** escolha **21 Oracle OpenJDK 21.0.7**
- **Gradle DSL:** selecione **Groovy**
- **Add sample code:** deixe **marcado**

6. Clique no botão azul **Create**.

Passo 2 – configuração do `build.gradle`

Abra o arquivo `build.gradle` na raiz do projeto. Vamos analisar e entender cada parte do arquivo.

Passo 1: Plugins usados

Na primeira parte do arquivo, veja os plugins configurados:

```
plugins {
    id 'java'
    id 'application'
    id 'com.github.johnrengelman.shadow' version '8.1.1'
}
```

Explicação:

- `id 'java'` ativa o suporte para projetos Java.
- `id 'application'` facilita a execução do programa, definindo uma classe principal.
- `id 'com.github.johnrengelman.shadow'` é um plugin para gerar um "fat JAR"que inclui todas as dependências no JAR final. JAR é um arquivo compactado usado para agrupar vários arquivos Java em um único pacote executável ou reutilizável.

Passo 2: Definindo grupo e versão

```
    group 'com.aluno'
    version '1.0'
```

Explicação: Essas duas linhas são usadas no arquivo `build.gradle` para definir a **identidade do projeto** e sua versão. Essas informações são especialmente úteis na hora de gerar o `.jar` do projeto.

- `group`: identifica o grupo ou domínio do projeto. Por convenção, usa-se o formato `domínio.invertido` (ex: `com.aluno`, `br.ifsc`, `org.minhaprova`).
- `version`: indica a versão atual do projeto (ex: `1.0`, `1.1.2`, `2.0-SNAPSHOT`).

Passo 3: Configuração dos repositórios

```
repositories {
    mavenCentral()
}
```

Explicação: Define que as dependências serão baixadas do repositório Maven Central.

Passo 4: Definição da classe principal

```
application {
    mainClass = 'bcd.Principal'
}
```

Explicação: Diz ao Gradle qual é a classe que contém o método `main`, ponto de entrada da aplicação.

Passo 5: Configuração para entrada padrão no terminal

```
run {
  standardInput = System.in
}

tasks.withType(JavaExec) {
  standardInput = System.in
}
```

Explicação: Permite que a aplicação receba entrada do teclado quando executada pelo Gradle.

Passo 6: Configuração do JAR

```
jar {
    manifest {
        attributes "Main-Class": "bcd.Principal"
    }
}
```

Explicação: Define a classe principal no manifesto do JAR para que o JAR seja executável.

Passo 7: Dependências

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.10.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.10.0'

    implementation 'org.xerial:sqlite-jdbc:3.42.0.0'
    implementation 'mysql:mysql-connector-java:8.0.33'
}

test {
    useJUnitPlatform()
}
```

Explicação:

- Dependências para testes com JUnit.
- Driver JDBC para SQLite e MySQL, necessários para conectar aos bancos de dados.
- As dependências `testImplementation` e `testRuntimeOnly` adicionam o JUnit 5 ao projeto.
- O bloco `test { useJUnitPlatform() }` garante que o Gradle use o mecanismo do JUnit 5 para executar os testes.

Passo 8: Configuração do `shadowJar`

```
tasks.shadowJar {
    mergeServiceFiles()
}
```

Explicação: Garante que, ao criar o "fat JAR", arquivos de configuração do serviço (drivers JDBC) sejam combinados corretamente.

Passo 3– Criar o arquivo Java principal

Onde: Na pasta do seu projeto, crie o arquivo `ExemploMuitoSimples.java` dentro do pacote `exemplo01` (`src/main/java/exemplo01/ExemploMuitoSimples.java`).

Por que: O arquivo Java será o responsável por todas as operações com o banco SQLite.

Passo 4 – Definir o pacote e importar bibliotecas

```
package exemplo01;

import java.sql.*;
import org.sqlite.SQLiteConfig;
```

O que faz: Define o pacote do arquivo e importa as classes necessárias para manipulação do banco de dados SQLite e Java SQL.

Por que: O pacote organiza o código e as importações trazem funções que vamos usar para conectar, criar comandos SQL e configurar o SQLite.

Passo 5 – Começar a classe `ExemploMuitoSimples`

```
public class ExemploMuitoSimples {
```

O que faz: Inicia a definição da classe que vai conter todos os métodos para manipulação do banco.

Passo 6 – Definir variáveis importantes da classe

```
    private String DB_URI = "jdbc:sqlite:lab01.sqlite";
    private SQLiteConfig sqLiteConfig;
    private final String DIVISOR =
     "--------------------------------------------------------------------------------\n";
```

O que faz:

- `DB_URI`: Localização do banco de dados SQLite.
- `sqLiteConfig`: Configurações específicas do SQLite, como ativar restrição de chaves estrangeiras.
- `DIVISOR`: Uma linha para formatar visualmente a saída no console.

Dica: Você pode alterar o caminho do banco se quiser usar um arquivo externo, mas por padrão está dentro do recurso do projeto.

Passo 7 – Criar os construtores da classe

```
    public ExemploMuitoSimples(String dB_URI) {
        this();
        DB_URI = dB_URI;
    }

    public ExemploMuitoSimples() {
        this.sqLiteConfig = new SQLiteConfig();
        sqLiteConfig.enforceForeignKeys(true); // ativa restrição de chaves estrangeiras
    }
```

O que faz: Define duas formas de criar o objeto:

- Um construtor padrão que inicializa a configuração do SQLite.
- Outro que permite passar um URI diferente para o banco.

Passo 8 – Criar método para cadastrar uma pessoa no banco

```
public int cadastrarPessoa(String nome, double peso, int altura, String email) throws SQLException {
    int resultado = -1;
    try (Connection conexao = DriverManager.getConnection(DB_URI, this.sqLiteConfig.toProperties());
         Statement stmt = conexao.createStatement()) {

        String sql = "INSERT INTO Pessoa (nome, peso, altura, email) VALUES ('" + nome + "'," + peso + "," +
     altura + ",'" + email + "')";
        resultado = stmt.executeUpdate(sql);

    } catch (SQLException e) {
        throw new SQLException("Erro ao cadastrar pessoa", e);
    }
    return resultado;
}
```

O que faz: Insere uma nova pessoa na tabela `Pessoa`.

Importante: Aqui está concatenando os valores direto na string SQL, o que não é recomendado (você vai aprender a usar `PreparedStatement` depois).

Passo 9– Criar método para alterar dados de uma pessoa

```
public int alterarDadosPessoa(int idPessoa, String nome, double peso, int altura, String email) throws
     SQLException {
    int resultado = -1;
    try (Connection conexao = DriverManager.getConnection(DB_URI, this.sqLiteConfig.toProperties());
         Statement stmt = conexao.createStatement()) {

        String sql = "UPDATE Pessoa SET nome = '" + nome + "', peso=" + peso + ", altura=" + altura + ",
     email = '" + email + "' WHERE idPessoa=" + idPessoa;
        resultado = stmt.executeUpdate(sql);

    } catch (SQLException e) {
        throw new SQLException("Erro ao alterar dados de pessoas", e);
    }
    return resultado;
}
```

O que faz: Atualiza os dados da pessoa com o `idPessoa` informado.

Passo 10 – Criar método para excluir pessoa

```
public int excluirPessoa(int idPessoa) throws SQLException {
    int resultado = -1;
    try (Connection conexao = DriverManager.getConnection(DB_URI, this.sqLiteConfig.toProperties());
         Statement stmt = conexao.createStatement()) {

        String sql = "DELETE FROM Pessoa WHERE idPessoa = " + idPessoa;
        resultado = stmt.executeUpdate(sql);

    } catch (SQLException e) {
        throw new SQLException("Erro ao excluir pessoas", e);
    }
    return resultado;
}
```

O que faz: Exclui uma pessoa do banco pelo ID.

Passo 11 – Criar método para listar todas as pessoas

```
public String listarRegistros() throws SQLException {
    StringBuilder sb = new StringBuilder();
    String sql = "SELECT * FROM Pessoa";

    try (Connection conexao = DriverManager.getConnection(DB_URI, this.sqLiteConfig.toProperties());
         Statement stmt = conexao.createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {

        if (!rs.next()) {
            sb.append("\nNenhuma pessoa cadastrada no banco\n");
        } else {
            sb.append(DIVISOR);
            sb.append(String.format("|%-5s|%-25s|%-10s|%-10s|%-25s|\n", "ID", "Nome", "Peso", "Altura", "
     Email"));
            sb.append(DIVISOR);

            do {
                sb.append(String.format("|%-5d|%-25s|%-10.2f|%-10d|%-25s|\n",
                        rs.getInt("idPessoa"),
                        rs.getString("Nome"),
                        rs.getDouble("peso"),
                        rs.getInt("altura"),
                        rs.getString("email")));
            } while (rs.next());
            sb.append(DIVISOR);
        }
    } catch (SQLException e) {
        throw new SQLException("Erro ao listar todas pessoas", e);
    }
    return sb.toString();
}
```

O que faz: Retorna uma `String` com a listagem formatada de todas as pessoas no banco.

Passo 12 – Criar método para buscar pessoa pelo email

```
public String listarDadosPessoa(String emailPessoa) throws SQLException {
    StringBuilder sb = new StringBuilder();
    String sql = "SELECT * FROM Pessoa WHERE Email = '" + emailPessoa + "'";

    try (Connection conexao = DriverManager.getConnection(DB_URI, this.sqLiteConfig.toProperties());
         Statement stmt = conexao.createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {

        if (!rs.next()) {
            sb.append("\nNenhuma pessoa cadastrada possui o email informado\n");
        } else {
            do {
                sb.append(String.format("%d|%s|%.2f|%d|%s\n",
                        rs.getInt("idPessoa"),
                        rs.getString("Nome"),
                        rs.getDouble("peso"),
                        rs.getInt("altura"),
                        rs.getString("email")));
            } while (rs.next());
        }
    } catch (SQLException e) {
        throw new SQLException("Erro ao listar dados de pessoas", e);
    }
    return sb.toString();
}
```

O que faz: Busca pessoas que tenham o email fornecido.

Dica: Esse método também usa concatenação SQL e está vulnerável a SQL Injection. No futuro, você pode melhorar isso.

Passo 13 – Criar método para criar o banco e tabela do zero

```
public boolean criaBancoDeDados() throws Exception {
       try (Connection conexao = DriverManager.getConnection(DB_URI, this.sqLiteConfig.toProperties());
               Statement statement = conexao.createStatement();) {

           statement.executeUpdate("drop table if exists Pessoa");

           statement.executeUpdate("create table Pessoa ( idPessoa INTEGER not null\n" +
                   " primary key AUTOINCREMENT,\n" +
                   " Nome     TEXT    not null,\n" +
                   " Peso     REAL,\n" +
                   " Altura   INTEGER,\n" +
                   " Email    Text)");

           statement.executeUpdate("INSERT INTO Pessoa (Nome, Peso, Altura, Email) " +
                   "VALUES ('Aluno Teste', 85.2, 180, 'aluno@teste.com.br')");
       } catch (Exception e) {
           throw new Exception("Erro ao criar tabelas", e);
       }
       return true;
   }
```

O que faz: Apaga a tabela `Pessoa` se existir e cria uma nova com um registro inicial.

Passo 14 – Fechar a classe

```
}
```

Fecha a definição da classe.

Classe Principal

Crie um novo pacote chamado `bcd` dentro da pasta *test/java* e dentro dele uma classe chamada `Principal`. Copie o seguinte código e leia os comentários que explicam cada etapa:

```
package bcd;

import exemplo01.ExemploMuitoSimples;
import java.util.InputMismatchException;
import java.util.Scanner;

public class Principal {
    private final String[] EXEMPLOS = {
        "\n..:: Pequenos exemplos com Java, SQLite e MySQL ::..\n",
        "1 - Exemplo 01",
        "6 - Sair do programa"
    };

    private final String[] MENU_EX1 = {
        "\n...:: Exemplo com SQLite ::...\n",
        "1 - Cadastrar pessoa",
        "2 - Alterar dados de uma pessoa",
        "3 - Excluir uma pessoa",
        "4 - Listar dados de uma pessoa",
        "5 - Listar todas pessoas",
        "6 - Voltar ao menu anterior"
    };

    private Scanner teclado;

    public Principal() {
        this.teclado = new Scanner(System.in);
    }
```

Scanner teclado Permite a leitura de dados digitados pelo usuário no terminal.

EXEMPLOS/MENU Arrays de `String` que representam os menus interativos do sistema.

Passo 15 – Método `main`

```
public static void main(String[] args) throws Exception {
    Principal p = new Principal();
    int opcao = -1;
    do {
        opcao = p.menu(p.EXEMPLOS);
        switch (opcao) {
            case 1:
                p.exemplo01();
                break;
        }
    } while (opcao != 6);
}
```

Explicação: O programa exibe o menu principal e executa a ação correspondente. Por enquanto, apenas o Exemplo 01 está implementado. Os demais serão adicionados posteriormente.

Passo 14 – Método `menu`

```
private int menu(String[] menuComOpcoes) {
    int opcao = -1;
    if (menuComOpcoes != null) {
        for (String linha : menuComOpcoes) {
            System.out.println(linha);
        }

        try {
            System.out.print("Entre com uma opção: ");
            opcao = teclado.nextInt();
        } catch (InputMismatchException e) {
            System.err.println("Erro. Informe um número inteiro.");
            opcao = -1;
            teclado.nextLine(); // limpar entrada
        }
    }
    return opcao;
}
```

Explicação: Apresenta o menu recebido como parâmetro e retorna a opção digitada pelo usuário.

Passo 16 – Método `exemplo01`

```
private void exemplo01() throws Exception {
    int opcao;
    ExemploMuitoSimples app = new ExemploMuitoSimples();

    // Criar o banco de dados e tabela
    app.criaBancoDeDados();
    try {
        do {
            opcao = this.menu(this.MENU_EX1);
            switch (opcao) {
                case 1:
                    // Cadastrar
                case 2:
                    // Alterar
                case 3:
                    // Excluir
                case 4:
```

```
                    // Listar por email
                case 5:
                    // Listar todos
            }
        } while (opcao != 6);
    } catch (InputMismatchException e) {
        System.err.println("ERRO: Dados fornecidos estão em um formato diferente do esperado.");
    }
}
```

Explicação: Este método chama os métodos da classe `ExemploMuitoSimples`, responsável pela comunicação com o banco de dados SQLite. As opções são mostradas em um menu próprio e os dados são lidos com `Scanner`.

Passo 17 – Implementando o cadastro da pessoa (opção 1)

```
case 1:
    try {
        teclado.nextLine(); // Limpa buffer
        System.out.print("Entre com o nome: ");
        String nome = teclado.nextLine();

        System.out.print("Entre com o email: ");
        String email = teclado.nextLine();

        System.out.print("Entre com o peso: ");
        double peso = teclado.nextDouble();

        System.out.print("Entre com a altura: ");
        int altura = teclado.nextInt();

        int resultado = app.cadastrarPessoa(nome, peso, altura, email);

        if (resultado > 0) {
            System.out.println("\nPessoa cadastrada com sucesso.\n");
        } else {
            System.out.println("\nHouve algum problema e não foi possível cadastrar");
        }
    } catch (Exception e) {
        System.err.println("\nErro com os dados fornecidos. Tente novamente.\n");
        teclado.nextLine(); // Limpa buffer para evitar erro contínuo
    }
    break;
```

Passo 18 – Alterar dados de uma pessoa (opção 2)

```
case 2:
    System.out.println(app.listarRegistros());
    System.out.print("Informe o ID da pessoa que irá alterar os dados: ");
    int idPessoa = teclado.nextInt();
    teclado.nextLine(); // Limpa buffer

    System.out.print("Entre com o nome: ");
    String nome = teclado.nextLine();
    System.out.print("Entre com o email: ");
    String email = teclado.nextLine();
    System.out.print("Entre com o peso: ");
    double peso = teclado.nextDouble();
    System.out.print("Entre com a altura: ");
    int altura = teclado.nextInt();

    int resultado = app.alterarDadosPessoa(idPessoa, nome, peso, altura, email);

    if (resultado > 0) {
        System.out.println("\nDados alterados com sucesso.\n");
    } else {
```

```
        System.out.println("\nHouve algum problema e não foi possível alterar");
    }
    break;
```

Explicação:

- Mostra os registros existentes para o usuário escolher qual deseja alterar.
- Solicita os novos dados.
- Usa o método `alterarDadosPessoa()` da classe auxiliar para atualizar no banco.

Passo 19 – Excluir uma pessoa (opção 3)

```
case 3:
    System.out.println(app.listarRegistros());
    System.out.print("Informe o ID da pessoa que deseja excluir: ");
    idPessoa = teclado.nextInt();

    resultado = app.excluirPessoa(idPessoa);

    if (resultado > 0) {
        System.out.println("\nPessoa excluída com sucesso\n");
    } else {
        System.out.println("\nHouve algum problema e não foi possível excluir");
    }
    break;
```

Explicação:

- Mostra todos os registros atuais.
- Usuário informa o ID da pessoa a ser excluída.
- O método `excluirPessoa()` realiza a exclusão no banco de dados.

Passo 20 – Listar dados de uma pessoa pelo e-mail (opção 4)

```
case 4:
    System.out.print("Entre com o email da pessoa que deseja procurar: ");
    String e = teclado.next();
    System.out.println(app.listarDadosPessoa(e));
    break;
```

Explicação:

- Usuário fornece o e-mail.
- O método `listarDadosPessoa()` retorna as informações completas da pessoa.

Passo 21 – Listar todas as pessoas (opção 5)

```
case 5:
    System.out.println(app.listarRegistros());
    break;
```

Explicação:

- Exibe todos os registros presentes no banco de dados.
- Usa o método `listarRegistros()` para retornar a listagem geral.

Passo 22 – Testando a classe `ExemploMuitoSimples` com JUnit

Agora vamos criar uma classe de teste utilizando `JUnit 5` para garantir que os métodos da classe *ExemploMuitoSimples* estão funcionando corretamente.

Classe de teste: `TesteExemplo01SQLite`

```java
package bcd;

import exemplo01.ExemploMuitoSimples;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.junit.jupiter.api.*;

import static org.junit.jupiter.api.Assertions.*;

@TestMethodOrder(MethodOrderer.MethodName.class)
public class TesteExemplo01SQLite {

    private ExemploMuitoSimples app;

    public TesteExemplo01SQLite() throws Exception {
        this.app = new ExemploMuitoSimples();
        this.app.criaBancoDeDados(); // cria a tabela com um registro padrão
    }

    @Test
    public void testeAincluirRegistro() throws SQLException {
        int resultado = this.app.cadastrarPessoa("Juca", 71, 174, "juca@email.com");
        assertEquals(1, resultado);
    }

    @Test
    public void testeBlistarRegistros() throws SQLException {
        String registros = this.app.listarRegistros();
        assertFalse(registros.equals(""), "Banco sem registros iniciais");
        Logger.getLogger(TesteExemplo01SQLite.class.getName()).log(Level.INFO, "\n" + registros);
    }

    @Test
    public void testeDalterarRegistro() throws Exception {
        int resultado = this.app.alterarDadosPessoa(1, "Novo nome", 82, 180, "aluno@teste.com.br");
        assertEquals(1, resultado);
        this.app.criaBancoDeDados(); // reinicia o banco
    }

    @Test
    public void testeExcluirRegistro() throws Exception {
        this.app.criaBancoDeDados(); // reinicia o banco
        assertEquals(1, this.app.excluirPessoa(1));
        this.app.criaBancoDeDados(); // reinicia o banco novamente
    }
}
```

Explicação:

- A anotação `@TestMethodOrder` define a ordem dos testes pelo nome do método, para garantir a sequência esperada.
- O construtor *TesteExemplo01SQLite()* inicializa o banco com um registro padrão chamando *criaBancoDeDados()*.
- Cada método de teste realiza uma verificação:
  - `testeAincluirRegistro` testa o cadastro de uma nova pessoa.
  - `testeBlistarRegistros` verifica se os dados estão sendo listados corretamente.

  - `testeDalterarRegistro` altera os dados de uma pessoa com ID 1.
  - `testeEexcluirRegistro` exclui a pessoa com ID 1.
- O uso de `assertEquals` e `assertFalse` garante que os métodos estão retornando os valores esperados.
- O banco de dados é restaurado ao estado inicial após cada teste crítico.

Como executar os testes no IntelliJ IDEA

- Clique com o botão direito sobre o arquivo `TesteExemplo01SQLite.java`.
- Selecione `Run 'TesteExemplo01SQLite'`.
- Verifique se todos os testes passaram com sucesso.

Requisitos

- Certifique-se de que o JUnit 5 está incluído nas dependências do projeto (exemplo: `build.gradle` ou `pom.xml`).
- O banco `lab01.sqlite` deve estar criado e acessível.

Próximos Passos

Os demais exemplos (02 a 05) serão adicionados no mesmo estilo, bastando ampliar o `switch-case` no método `main` e implementar os respectivos métodos auxiliares.

Dicas importantes do laboratório

- Sempre limpe o buffer do `Scanner` ao alternar entre `nextInt`/`nextDouble` e `nextLine` para evitar erros.
- Trate as exceções para não deixar o programa fechar abruptamente.
- Mostre mensagens claras para o usuário sobre sucesso ou erro.
- Use o loop para que o programa rode até o usuário decidir sair.

Cuidados e Más Práticas no Código Java

O código apresentado para a classe `Principal.java` serve para ilustrar um exemplo simples de interação com banco de dados via menu no terminal, mas contém algumas práticas que devem ser evitadas em projetos reais. Confira alguns pontos importantes:

- **Uso excessivo de tratamento genérico de exceções:** Embora haja `try-catch` para evitar erros na entrada de dados, o uso muito genérico pode esconder problemas mais específicos e dificulta o diagnóstico correto.
- **Repetição de código para leitura dos dados:** O código pede repetidamente os mesmos dados (nome, email, peso, altura) em diferentes partes, sem criar métodos auxiliares para evitar repetição e melhorar a manutenção.
- **Falta de validação detalhada dos dados de entrada:** Apesar de capturar erros de tipo na entrada, não há validação dos valores em si (por exemplo, se o peso é positivo, se o email tem formato válido, etc.).
- **Uso do `Scanner` diretamente e consumo manual do buffer:** O gerenciamento manual do buffer do `Scanner` com `nextLine()` para limpar o fluxo pode ser propenso a erros e confuso para quem está iniciando.
- **Dependência direta de implementação concreta:** A classe `Principal` instancia diretamente a classe `ExemploMuitoSimples`, o que dificulta a testabilidade e a flexibilidade do código.

- **Tratamento de fluxo de programa com muitos comandos dentro do `switch`:** O método `exemplo01` é longo e contém muita lógica em um único método, o que compromete a legibilidade e manutenção.

Estas observações visam incentivar boas práticas de programação que tornam o código mais seguro, legível e fácil de manter. Com pequenas melhorias, este exemplo pode se transformar em um sistema mais robusto e confiável!

Este projeto foi integralmente elaborado pelo professor **Emerson Ribeiro de Mello**, docente do IFSC – Campus São José.