Table of Contents

Learn JavaFX

Introdução ao JavaFX

JavaFX é uma plataforma de desenvolvimento de aplicações desktop e mobile da Oracle, que permite criar interfaces gráficas de usuário (GUIs) ricas e interativas. Ele é projetado para substituir a biblioteca Swing e fornecer uma alternativa mais moderna e eficiente para o desenvolvimento de aplicações desktop.

Mind Map: JavaFX

Este mind map apresenta uma visão geral das principais características e recursos do JavaFX, incluindo sua capacidade de criar interfaces gráficas de usuário ricas e interativas, suporte a multimídia e conectividade, integração com bases de dados e serviços web, e suporte a dispositivos móveis e desktop. Além disso, o JavaFX tem uma comunidade ativa e recursos, documentação e suporte a múltiplos sistemas operacionais e linguagens de programação.

Criando Estrutura de Projeto

Estrutura de Projeto JavaFX

```
+-- src
|  +-- main
|  |  +-- java
|  |  |  +-- br
|  |  |  |  +-- com
|  |  |  |  |  +-- exemplo
|  |  |  |  |  |  +-- Main.java
|  |  |  |  |  |  +-- Controller.java
|  |  |  |  |  |  +-- Model.java
|  |  |  |  |  |  +-- View.java
|  |  |  |  |  |  +-- ...
|  |  +-- resources
|  |  |  +-- css
|  |  |  |  +-- estilo.css
|  |  |  +-- images
|  |  |  |  +-- imagem.png
|  |  |  +-- fxml
|  |  |  |  +-- tela.fxml
|  +-- test
|  |  +-- java
|  |  |  +-- br
|  |  |  |  +-- com
|  |  |  |  |  +-- exemplo
|  |  |  |  |  |  +-- Teste.java
|  +-- module-info.java
+-- pom.xml (ou build.gradle)
+-- README.md
```

Explicação:

- `src`: diretório fonte do projeto

- `main`: diretório principal do projeto
  - `java`: diretório de código Java
    - `br/com/exemplo`: pacote do projeto
      - `Main.java`: classe principal do projeto
      - `Controller.java`: classe de controle do projeto
      - `Model.java`: classe de modelo do projeto
      - `View.java`: classe de visualização do projeto
  - `resources`: diretório de recursos do projeto
    - `css`: diretório de folhas de estilo CSS
    - `images`: diretório de imagens
    - `fxml`: diretório de arquivos FXML
- `test`: diretório de testes do projeto
  - `java`: diretório de código Java de testes
    - `br/com/exemplo`: pacote de testes do projeto
      - `Teste.java`: classe de teste do projeto
- `module-info.java`: arquivo de informações do módulo do projeto
- `pom.xml` (ou `build.gradle`): arquivo de configuração do projeto Maven (ou Gradle)
- `README.md`: arquivo de documentação do projeto

Essa é a estrutura básica de um projeto JavaFX. O diretório `src` contém o código fonte do projeto, o diretório `main` contém o código principal do projeto, e o diretório `test` contém os testes do projeto. O arquivo `module-info.java` contém as informações do módulo do projeto, e o arquivo `pom.xml` (ou `build.gradle`) contém as configurações do projeto Maven (ou Gradle).

FXML

FXML é uma linguagem de marcação XML (Extensible Markup Language) utilizada para criar interfaces gráficas de usuário (GUIs) em aplicações JavaFX. É uma forma de descrever a estrutura e o layout de uma interface gráfica de usuário de maneira declarativa, sem a necessidade de escrever código Java.

Um arquivo FXML é um arquivo XML que contém elementos que representam componentes gráficos, como botões, labels, text fields, etc. Esses elementos são organizados em uma hierarquia, onde cada elemento pode conter outros elementos.

Aqui está um exemplo simples de um arquivo FXML:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.Button?>

<VBox xmlns="http://javafx.com/javafx"
xmlns:fx="http://javafx.com/fxml">
    <Label text="Olá, mundo!" />
    <Button text="Clique aqui" onAction="#handleButtonClicked" />
</VBox>
```

Nesse exemplo, temos:

- Uma declaração de namespace (`xmlns`) que especifica a linguagem FXML e o namespace de JavaFX.
- Uma importação de componentes (`<?import ...?>`) que permite usar componentes específicos de JavaFX, como `VBox`, `Label` e `Button`.
- Um elemento `VBox` que é o contêiner principal da interface gráfica.
- Um elemento `Label` que exibe o texto "Olá, mundo!".
- Um elemento `Button` que tem o texto "Clique aqui" e um evento `onAction` que é disparado quando o botão é clicado.

Quando o arquivo FXML é carregado em uma aplicação JavaFX, o framework JavaFX cria os componentes gráficos correspondentes e os organiza de acordo com a hierarquia

definida no arquivo FXML.

O FXML é uma forma poderosa de criar interfaces gráficas de usuário em JavaFX, pois permite:

- Descrever a estrutura e o layout da interface gráfica de maneira declarativa.
- Separar a lógica de apresentação da lógica de negócios.
- Reutilizar componentes gráficos em diferentes partes da aplicação.

Criando com Maven + Intelijj

Criar um projeto com Maven no IntelliJ IDEA é um processo relativamente simples. Aqui está um passo a passo para criar um projeto Maven no IntelliJ IDEA:

Passo 1: Criar um novo projeto

- Abra o IntelliJ IDEA e clique em "Create New Project" (ou pressione `Ctrl + Shift + Alt + S` no Windows/Linux ou `Cmd + Shift + Alt + S` no Mac).
- Selecione "Maven" como tipo de projeto e clique em "Next".

Passo 2: Selecionar o modelo de projeto

- Selecione o modelo de projeto Maven que deseja usar (por exemplo, "maven-archetype-quickstart" para um projeto Java simples).
- Clique em "Next".

Passo 3: Configurar o projeto

- Insira o grupo, artefato e versão do projeto.
- Selecione a linguagem de programação (por exemplo, Java).
- Clique em "Next".

Passo 4: Selecionar o local do projeto

- Selecione o local onde deseja salvar o projeto.
- Clique em "Finish".

Passo 5: Criar o projeto

- O IntelliJ IDEA criará o projeto Maven com base nas configurações que você forneceu.
- Você verá a estrutura de diretórios do projeto Maven criada no seu projeto.

Passo 6: Adicionar dependências

- Abra o arquivo `pom.xml` e adicione as dependências necessárias para o seu projeto.
- Você pode usar a interface gráfica do IntelliJ IDEA para adicionar dependências ou editar o arquivo `pom.xml` manualmente.

Passo 7: Compilar e executar o projeto

- Clique em "Build" (ou pressione `Ctrl + F9` no Windows/Linux ou `Cmd + F9` no Mac) para compilar o projeto.
- Clique em "Run" (ou pressione `Shift + F10` no Windows/Linux ou `Shift + Cmd + R` no Mac) para executar o projeto.

Pronto! Você criou um projeto Maven no IntelliJ IDEA.

Criando a primeira tela

Para criar telas, o comum é usarmos o Scene Builder, que é um programa para manipular FXML

Tutorial de download:

Windows

1. Abra um navegador web e vá para o site da GluonHQ: https://gluonhq.com/products/scene-builder/ (https://gluonhq.com/products/scene-builder/)
2. Clique no botão "Download" no topo da página.
3. Selecione a opção "Windows" no menu dropdown.
4. Selecione a versão do Scene Builder que você deseja baixar (por exemplo, "Scene Builder 11.0.0").
5. Clique no botão "Download" para baixar o arquivo `.exe`.
6. Execute o arquivo baixado e siga as instruções para instalar o Scene Builder.
7. Uma vez instalado, você pode encontrar o Scene Builder no menu de aplicativos do seu sistema.

Linux

1. Abra um navegador web e vá para o site da GluonHQ: https://gluonhq.com/products/scene-builder/ (https://gluonhq.com/products/scene-builder/)
2. Clique no botão "Download" no topo da página.
3. Selecione a opção "Linux" no menu dropdown.

4. Selecione a versão do Scene Builder que você deseja baixar (por exemplo, "Scene Builder 11.0.0").
5. Clique no botão "Download" para baixar o arquivo `.deb` (para sistemas baseados em Debian) ou `.rpm` (para sistemas baseados em RPM).
6. Abra o arquivo baixado com um gerenciador de pacotes (como o `dpkg` ou `rpm`) e siga as instruções para instalar o Scene Builder.
7. Uma vez instalado, você pode encontrar o Scene Builder no menu de aplicativos do seu sistema.

Observações

- Certifique-se de que você tenha o Java 8 ou superior instalado no seu sistema para executar o Scene Builder.
- Se você estiver usando um sistema de 64 bits, certifique-se de baixar a versão de 64 bits do Scene Builder.
- Se você estiver usando um sistema de 32 bits, certifique-se de baixar a versão de 32 bits do Scene Builder.

Instalação via linha de comando (Linux)

Se você preferir instalar o Scene Builder via linha de comando, você pode usar os seguintes comandos:

- Para sistemas baseados em Debian:

```
sudo dpkg -i scene-builder-11.0.0.deb
```

- Para sistemas baseados em RPM:

```
sudo rpm -i scene-builder-11.0.0.rpm
```

Substitua `scene-builder-11.0.0.deb` ou `scene-builder-11.0.0.rpm` pelo nome do arquivo que você baixou.

Instalação via linha de comando (Windows)

Se você preferir instalar o Scene Builder via linha de comando, você pode usar o seguinte comando:

```
msiexec /i scene-builder-11.0.0.msi
```

Substitua `scene-builder-11.0.0.msi` pelo nome do arquivo que você baixou.

Criando o FXML

Vamos usar o arquivo `student.fxml` como exemplo para entender como funciona o JavaFX.

Analisando o arquivo `student.fxml`

O arquivo `student.fxml` é um arquivo XML que define a interface gráfica da nossa aplicação. Aqui está o conteúdo do arquivo:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.RadioButton?>
<?import javafx.scene.control.Separator?>
<?import javafx.scene.control.TableColumn?>
<?import javafx.scene.control.TableView?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.RowConstraints?>
<?import javafx.scene.text.Font?>

<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity"
minHeight="-Infinity" minWidth="-Infinity" prefHeight="727.0"
```

```
prefWidth="1176.0" xmlns="http://javafx.com/javafx/22"
xmlns:fx="http://javafx.com/fxml/1">
    <!-- Conteúdo do arquivo -->
</AnchorPane>
```

Vamos analisar os elementos do arquivo:

- `AnchorPane`: é o contêiner principal da interface gráfica. Ele é um contêiner que pode conter outros elementos e é usado para definir a estrutura da interface.
- `Label`, `Button`, `RadioButton`, `Separator`, `TableColumn`, `TableView`, `TextField`, `GridPane`, `ColumnConstraints`, `RowConstraints`, `Font`: são elementos que são usados para criar a interface gráfica da nossa aplicação.

Entendendo os atributos

Agora que analisamos os elementos, vamos entender os atributos que são usados no arquivo:

- `maxHeight`, `maxWidth`, `minHeight`, `minWidth`, `prefHeight`, `prefWidth`: são atributos que são usados para definir o tamanho do contêiner `AnchorPane`.
- `xmlns` e `xmlns:fx`: são atributos que são usados para definir o namespace do arquivo FXML.

Conectando o arquivo FXML ao código Java

O arquivo `Main.java` é o ponto de entrada da nossa aplicação JavaFX. Ele é responsável por carregar a interface gráfica e conectar o arquivo FXML ao código Java.

Importando as bibliotecas necessárias

O código começa importando as bibliotecas necessárias para trabalhar com JavaFX:

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;
```

Essas bibliotecas são responsáveis por:

- `Application`: é a classe base para todas as aplicações JavaFX.
- `FXMLLoader`: é a classe responsável por carregar os arquivos FXML.
- `Parent`: é a classe base para todos os elementos da interface gráfica.
- `Scene`: é a classe responsável por definir a cena da interface gráfica.
- `Stage`: é a classe responsável por definir a janela da interface gráfica.

Definindo a classe Main

A classe `Main` é definida como uma subclasse de `Application`:

```
public class Main extends Application {
```

Isso significa que a classe `Main` herda todos os métodos e propriedades da classe `Application`.

Definindo o método start

O método `start` é chamado quando a aplicação é iniciada:

```
@Override
public void start(Stage stage) throws Exception {
```

Esse método é responsável por carregar a interface gráfica e conectar o arquivo FXML ao código Java.

Carregando o arquivo FXML

O arquivo FXML é carregado usando o `FXMLLoader`:

```
Parent screen =
FXMLLoader.load(getClass().getResource("screens/student.fxml"));
```

O `FXMLLoader` carrega o arquivo FXML e retorna um objeto `Parent` que representa a interface gráfica.

Definindo a cena

A cena é definida usando o objeto `Parent` carregado anteriormente:

```
Scene scene = new Scene(screen);
```

A cena é responsável por definir a estrutura da interface gráfica.

Definindo a janela

A janela é definida usando o objeto `Stage` passado como parâmetro para o método `start`:

```
window.setScene(scene);
window.

show();
window.
```

```
setTitle("Prime Bank");
```

A janela é responsável por exibir a interface gráfica.

Conectando o arquivo FXML ao código Java

O arquivo FXML é conectado ao código Java usando o `FXMLLoader`. O `FXMLLoader` carrega o arquivo FXML e retorna um objeto `Parent` que representa a interface gráfica. Esse objeto é então usado para definir a cena e a janela da interface gráfica.

Método initialize

O método `initialize` é chamado quando a interface gráfica é carregada. Ele é usado para inicializar os elementos da interface e definir os eventos que são disparados quando os elementos são clicados.

No nosso caso, o método `initialize` é definido no arquivo `StudentController.java`:

```
public class StudentController {
    // ...
    @FXML
    public void initialize() {
        // Inicializar os elementos da interface
        // Definir os eventos que são disparados quando os
elementos são clicados
    }
}
```

O método `initialize` é chamado quando a interface gráfica é carregada e é responsável por inicializar os elementos da interface e definir os eventos que são disparados quando os elementos são clicados.

Nomeando os elementos da interface gráfica

Para acessarmos os elementos precisamos nomear eles e para isso usamos `ID`:

Código

Nesta tab, você pode ver o código XML que define a tabela de estudantes. O código é o seguinte:

```
<TableView fx:id="tv_estudante" layoutX="431.0"
layoutY="353.0" prefHeight="359.0" prefWidth="1146.0"
AnchorPane.bottomAnchor="15.0" AnchorPane.leftAnchor="15.0"
AnchorPane.rightAnchor="15.0" AnchorPane.topAnchor="353.0">
        <columns>
          <TableColumn fx:id="tc_id" prefWidth="75.0"
text="ID"/>
            <TableColumn fx:id="tc_nome" prefWidth="75.0"
text="Nome"/>
            <TableColumn fx:id="tc_sexo" prefWidth="75.0"
text="Sexo"/>
            <TableColumn fx:id="tc_idade" prefWidth="75.0"
text="Idade"/>
        </columns>
    <columnResizePolicy>
        <TableView fx:constant="CONSTRAINED_RESIZE_POLICY"/>
    </columnResizePolicy>
</TableView>
```

No Scene

Nesta tab, você pode ver as etapas para definir o ID de um elemento no Scene Builder. As etapas são as seguintes:

1. **Clique no elemento**: clique no elemento que você deseja definir o ID.
2. **Selecione o item "Code"**: selecione o item "Code" no menu de contexto do elemento.
3. **Selecione o item "FX:ID"**: selecione o item "FX:ID" no menu de contexto do elemento.
4. **Defina o ID**: defina o ID do elemento no campo de texto.

Definindo ID no Scene

Criar primeiro Controller

Controllers são componentes fundamentais em aplicações JavaFX, responsáveis por gerenciar a lógica de interação entre a interface do usuário (FXML) e o modelo de dados.

Estrutura Básica de um Controller

Um controller JavaFX típico segue esta estrutura:

```
public class StudentController implements Initializable {
    // Injeção de componentes FXML
    @FXML
    private Button btnSalvar;

    @FXML
    private TextField tfNome;

    // Método de inicialização
    @Override
    public void initialize(URL url, ResourceBundle resourceBundle)
{
        // Código de inicialização
    }

    // Handlers de eventos
    @FXML
    public void handleSalvar(ActionEvent event) {
        // Lógica do botão salvar
    }
}
```

Componentes Principais

1. Anotações FXML

- `@FXML`: Usada para injetar elementos definidos no FXML no controller

- Deve ser aplicada em:
  - Campos que representam elementos da UI
  - Métodos que tratam eventos
  - Método initialize (opcional)

2. Implementação do Initializable

```
public class SeuController implements Initializable {
    @Override
    public void initialize(URL url, ResourceBundle resourceBundle)
{
        // Código executado quando a tela é carregada
    }
}
```

3. Exemplo Prático: StudentController

Nosso `StudentController` atual demonstra vários conceitos importantes:

```
public class StudentController implements Initializable {
    // Campos de UI
    @FXML
    private Button btn_deletar;
    @FXML
    private Button btn_editar;
    @FXML
    private Button btn_salvar;
    @FXML
    private RadioButton rb_f;
    @FXML
    private RadioButton rb_m;
    @FXML
    private TextField tf_nome;
    @FXML
    private TextField tf_idade;
    @FXML
```

```
    private TableView tv_estudante;

    // Método de inicialização
    @Override
    public void initialize(URL url, ResourceBundle resourceBundle)
{
        // Inicialização dos componentes
    }

    // Exemplo de handler de evento
    @FXML
    public void showName(ActionEvent actionEvent) {
        String name = tf_nome.getText();
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Hello World");
        alert.setHeaderText("Hello World");
        alert.setContentText(name);
        alert.showAndWait();
    }
}
```

Conectando Controller ao FXML

Para conectar um controller ao arquivo FXML, você precisa:

1. Definir o controller no FXML:

```
<AnchorPane xmlns="http://javafx.com/javafx"
            xmlns:fx="http://javafx.com/fxml"

fx:controller="prime.punkdomus.primebank.controller.StudentControl
ler">
```

2. Referenciar os elementos no controller usando `@FXML`

3. Definir os handlers de eventos no FXML:

```
<Button onAction="#showName" text="Mostrar Nome"/>
```

Boas Práticas

1. **Nomenclatura**
   - Use nomes descritivos para métodos handlers
   - Prefixe campos de UI com identificadores (ex: btn_, tf_, lbl_)
2. **Organização**
   - Agrupe campos relacionados
   - Mantenha handlers próximos aos campos que eles manipulam
   - Separe lógica de negócios da lógica de UI
3. **Inicialização**
   - Use o método `initialize()` para configuração inicial
   - Configure listeners e bindings neste método
   - Inicialize coleções e outros dados necessários
4. **Tratamento de Eventos**
   - Mantenha handlers concisos
   - Delegue lógica complexa para classes de serviço
   - Trate exceções adequadamente

Dicas de Desenvolvimento

1. **Debug**
   - Use `System.out.println()` ou logging para debug
   - Verifique se os elementos FXML estão sendo injetados corretamente

- Confirme se os handlers estão sendo chamados

2. **Testes**
   - Teste diferentes cenários de interação
   - Verifique estados inválidos
   - Teste navegação entre telas
3. **Manutenção**
   - Mantenha o controller focado em uma responsabilidade
   - Documente comportamentos complexos
   - Refatore quando necessário

Próximos Passos

- Implementar validação de campos
- Adicionar persistência de dados
- Criar telas adicionais
- Implementar navegação entre telas

Primeiro Model

Models são classes que representam os dados e a lógica de negócios da sua aplicação JavaFX. Eles são a camada de dados no padrão MVC (Model-View-Controller).

Estrutura Básica de um Model

Vamos analisar nosso primeiro model, a classe `Student`:

```
public class Student {
    private long id;
    private String name;
    private int age;
    private char sex;

    // Getters e Setters
}
```

Anatomia do Model

1. **Atributos Privados**
   - Use `private` para encapsulamento
   - Escolha tipos de dados apropriados
   - Nomeie de forma clara e descritiva
2. **Getters e Setters**
   - Fornecem acesso controlado aos atributos
   - Permitem validação de dados
   - Mantêm o encapsulamento

Boas Práticas

1. Validação de Dados

```
public void setAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("Idade não pode ser
negativa");
    }
    this.age = age;
}

public void setSex(char sex) {
    if (sex != 'M' && sex != 'F') {
        throw new IllegalArgumentException("Sexo deve ser 'M' ou
'F'");
    }
    this.sex = sex;
}
```

2. Construtores

```
public class Student {
    // Construtor padrão
    public Student() {
    }

    // Construtor com parâmetros
    public Student(String name, int age, char sex) {
        this.name = name;
        setAge(age);
        setSex(sex);
    }
}
```

3. Métodos de Utilidade

```
public boolean isAdult() {
    return age >= 18;
}
```

```
@Override
public String toString() {
    return "Student{" +
           "id=" + id +
           ", name='" + name + '\'' +
           ", age=" + age +
           ", sex=" + sex +
           '}';
}
```

Integração com JavaFX

1. Observable Properties

Para melhor integração com JavaFX, podemos usar Properties:

```
public class Student {
    private final StringProperty name = new
SimpleStringProperty();
    private final IntegerProperty age = new
SimpleIntegerProperty();

    // Getters e Setters para Properties
    public StringProperty nameProperty() {
        return name;
    }

    public String getName() {
        return name.get();
    }

    public void setName(String value) {
        name.set(value);
    }
}
```

2. Binding com a Interface

No controller:

```
@FXML
private TextField tfNome;

public void initialize() {
    Student student = new Student();

tfNome.textProperty().bindBidirectional(student.nameProperty());
}
```

Persistência de Dados

1. Exemplo de Lista em Memória

```
public class StudentRepository {
    private static final List<Student> students = new ArrayList<>
();

    public static void add(Student student) {
        students.add(student);
    }

    public static List<Student> getAll() {
        return new ArrayList<>(students);
    }
}
```

2. Observable Collections

```
public class StudentRepository {
    private static final ObservableList<Student> students =
        FXCollections.observableArrayList();

    public static ObservableList<Student> getStudents() {
```

```
        return students;
    }
}
```

Boas Práticas de Modelagem

1. **Imutabilidade Quando Possível**
   - Use `final` para atributos que não devem mudar
   - Considere criar versões imutáveis dos objetos
2. **Validação Robusta**
   - Valide dados em setters
   - Lance exceções apropriadas
   - Documente restrições
3. **Documentação**
   - Use JavaDoc para documentar a classe e métodos importantes
   - Explique regras de negócio complexas
   - Documente exceções
4. **Testes**
   - Crie testes unitários
   - Teste casos de borda
   - Verifique validações

Próximos Passos

1. Implementar mais validações no model `Student`

2. Adicionar suporte a persistência em banco de dados

3. Criar models relacionados (ex: `Course`, `Grade`)

4. Implementar padrões de projeto relevantes

Exercícios Práticos

1. Adicione mais campos ao model `Student` (ex: email, matrícula)

2. Implemente validações para os novos campos

3. Crie um método para calcular a idade baseado na data de nascimento

4. Implemente `equals()` e `hashCode()`

Criando Classe para Connection MySQL

A conexão com banco de dados é uma parte fundamental de aplicações que necessitam persistir dados. Vamos criar uma classe responsável por gerenciar conexões com MySQL.

Estrutura Básica

Nossa classe `ConnectionDB` utiliza o padrão Singleton para gerenciar conexões com o banco de dados:

```
public class ConnectionDB {
    private static final String url =
"jdbc:mysql://172.18.0.2:3306/crud_student_javafx";
    private static final String user = "root";
    private static final String password = "rootpassword";

    public static Connection getInstance() {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            return DriverManager.getConnection(url, user,
password);
        } catch (SQLException e) {
            throw new RuntimeException(e.getMessage());
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Dependências Necessárias

Para usar o MySQL com Java, adicione a dependência no `pom.xml`:

```
<dependency>
    <groupId>mysql</groupId>
```

```
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.28</version>
</dependency>
```

Configuração da Conexão

1. String de Conexão

A string de conexão contém informações importantes:

- `jdbc:mysql://` - Protocolo e tipo de banco
- `172.18.0.2` - Endereço do servidor
- `3306` - Porta (padrão MySQL)
- `crud_student_javafx` - Nome do banco de dados

```
private static final String url =
"jdbc:mysql://172.18.0.2:3306/crud_student_javafx";
```

2. Credenciais

Armazene as credenciais de forma segura:

```
private static final String user = "root";
private static final String password = "rootpassword";
```

> ⚠ **Nota de Segurança**: Em ambiente de produção, considere usar variáveis de ambiente ou arquivos de configuração externos para armazenar credenciais.

Métodos Principais

1. Obter Conexão

```
public static Connection getInstance() {
    try {
```

```
        Class.forName("com.mysql.cj.jdbc.Driver");
        return DriverManager.getConnection(url, user, password);
    } catch (SQLException e) {
        throw new RuntimeException(e.getMessage());
    } catch (ClassNotFoundException e) {
        throw new RuntimeException(e);
    }
}
```

2. Fechar Conexão

```
public static void close(Connection connection) {
    try {
        if (connection != null) {
            connection.close();
        }
    } catch (SQLException e) {
        throw new RuntimeException(e.getMessage());
    }
}
```

Boas Práticas

1. **Tratamento de Exceções**
   - Sempre feche conexões em blocos `finally`
   - Use try-with-resources quando possível
   - Trate exceções específicas do SQL
2. **Gerenciamento de Recursos**
   - Feche conexões após o uso
   - Utilize pool de conexões em aplicações maiores
   - Monitore o número de conexões ativas

3. **Segurança**

- Não exponha credenciais no código
- Use prepared statements para evitar SQL injection
- Implemente timeout de conexão

Exemplo de Uso

```
public class ExemploUso {
    public void executarOperacao() {
        try (Connection conn = ConnectionDB.getInstance()) {
            // Usar a conexão para operações no banco
            PreparedStatement stmt = conn.prepareStatement("SELECT
* FROM students");
            ResultSet rs = stmt.executeQuery();
            // Processar resultados
        } catch (SQLException e) {
            // Tratar exceção
        }
    }
}
```

Configuração do Banco de Dados

1. Script SQL para Criar o Banco

```sql
CREATE DATABASE crud_student_javafx;

USE crud_student_javafx;

CREATE TABLE students (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    age INT NOT NULL,
```

```
    sex CHAR(1) NOT NULL
);
```

2. Permissões Necessárias

```
GRANT ALL PRIVILEGES ON crud_student_javafx.* TO 'root'@'%';
FLUSH PRIVILEGES;
```

Troubleshooting

1. **Conexão Recusada**
   - Verifique se o servidor MySQL está rodando
   - Confirme a porta e endereço IP
   - Verifique as permissões do usuário
2. **Driver não Encontrado**
   - Confirme se a dependência está no `pom.xml`
   - Verifique se o driver está no classpath
   - Confirme a versão correta do driver
3. **Problemas de Autenticação**
   - Verifique as credenciais
   - Confirme as permissões do usuário
   - Verifique o método de autenticação do MySQL

Próximos Passos

1. Implementar pool de conexões
2. Adicionar logging de operações

3. Criar classes DAO para cada entidade

4. Implementar transações

5. Adicionar testes de integração

Exercícios Práticos

1. Implemente um método para testar a conexão

2. Crie um arquivo de configuração para as credenciais

3. Adicione suporte a múltiplos bancos de dados

4. Implemente métricas de conexão