Análise e Desenvolvimento de Sistemas
Programação Java
Profª. Ana Luiza Scharf
ana.scharf@ifsc.edu.br

Laboratório: Exemplo 2 - Acessando Banco com Padrões de Projeto

Objetivo

Demonstrar como utilizar o padrão de projeto **Factory** para acessar o banco de dados de forma segura e organizada, utilizando Java e SQLite.

Pré-requisitos

Este exemplo é uma **continuação do Exemplo 01**, onde já foi criado o banco de dados *lab01.sqlite* e implementada uma classe básica de acesso ao banco usando JDBC.

Para prosseguir com este exemplo, é necessário:

- Ter concluído o Exemplo 01 com sucesso.
- Manter o arquivo *lab01.sqlite* no diretório raiz do projeto.
- Ter a classe *ExemploMuitoSimples.java* funcionando corretamente.

Organização das pastas esperada:

```
src/
 java/
    exemplo01/
       ExemploMuitoSimples.java
    exemplo02/
       PadroesDeProjeto.java
       db/
           ConnectionFactory.java
    bcd/
        Principal.java
 lab01.sqlite
```

Crie a pasta *exemplo02* e dentro dela uma classe PadroesDeProjeto e crie uma subpasta *db* para armazenar a classe *ConnectionFactory*, responsável por fornecer conexões com o banco.

Objetivo da Classe PadroesDeProjeto

Esta classe tem como objetivo demonstrar o uso de boas práticas na manipulação de banco de dados utilizando o padrão de projeto **Factory** para criação de conexões.

Ela implementa o método *listarPessoas()*, que realiza uma consulta no banco SQLite e apresenta os resultados formatados no terminal. Os principais pontos trabalhados na classe são:

- Uso de *try-with-resources* para garantir o fechamento automático da conexão com o banco.
- Aplicação de *StringBuilder* para construção eficiente da saída.
- Formatação tabular da saída para melhor visualização no terminal.
- Tratamento de exceções com *SQLException*.

Explicação do Código

```
package exemplo02;

import exemplo02.db.ConnectionFactory;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

O que faz:

- Define o pacote da classe.
- Importa a ConnectionFactory e recursos do JDBC.

```
public class PadroesDeProjeto {
    private final String DIVISOR =
     "--------------------------------------------------------------------------------\n";
```

O que faz:

- Inicia a classe que irá conter a lógica para listar pessoas do banco.
- *DIVISOR*: linha visual para separar o conteúdo no console.

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

O que faz:

- Método que consulta o banco e retorna os dados formatados.
- *sb*: usado para montar o texto final.
- *sql*: comando SELECT para buscar todas as pessoas.

```
        try (Connection conexao = ConnectionFactory.getDBConnection();
             Statement stmt = conexao.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {
```

O que faz:

- Abre conexão com o banco e executa a consulta SQL.
- Os recursos são fechados automaticamente após o uso.

```
            if (!rs.next()) {
                sb.append("\nNenhuma pessoa cadastrada no banco\n");
```

O que faz:

- Verifica se o banco está vazio e informa o usuário.

```
            } else {
                sb.append(DIVISOR);
                sb.append(String.format("|%-5s|%-25s|%-10s|%-10s|%-25s|\n", "ID", "Nome", "Peso", "Altura", "
    Email"));
                sb.append(DIVISOR);
```

O que faz:

- Imprime o cabeçalho da tabela com formatação.

```
                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);
```

O que faz:

- Percorre todos os registros e formata a saída para cada pessoa.

```
            }
        } catch (SQLException e) {
            throw new SQLException(e);
        }
        return sb.toString();
    }
}
```

O que faz:

- Trata erros de SQL.
- Retorna os dados formatados em texto.

Objetivo da Classe ConnectionFactory

A classe *ConnectionFactory* foi criada para centralizar e padronizar a criação de conexões com o banco de dados, utilizando o padrão de projeto **Factory Method**.

Por que usar essa classe?

- **Organização:** Em vez de repetir código para abrir conexões em vários pontos do sistema, colocamos toda a lógica em um só lugar.
- **Segurança e controle:** Permite configurar o banco de forma centralizada. Neste exemplo, ativamos a verificação de chaves estrangeiras com *PRAGMA foreign_keys = ON*.
- **Reaproveitamento:** Sempre que for necessário obter uma conexão, basta chamar o método *getDBConnection()*, evitando duplicação de código.
- **Facilidade de manutenção:** Se precisar mudar a forma como o banco é acessado (por exemplo, trocar SQLite por outro banco), isso será feito apenas nesta classe.

Resumo: A *ConnectionFactory* melhora a qualidade e a manutenibilidade do código, facilitando a conexão com o banco e evitando erros comuns de conexão duplicada, esquecimento de configuração ou códigos repetidos.

Explicação do Código - Classe *ConnectionFactory*

```
package exemplo02.db;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import org.sqlite.SQLiteConfig;
```

O que faz:

- Define o pacote da classe como *exemplo02.db*, indicando que está relacionada ao banco de dados.

- Importa classes necessárias para a manipulação de conexões JDBC.
- Importa *SQLiteConfig*, que permite configurar detalhes específicos do banco SQLite (como ativar chaves estrangeiras).

```
/**
 * Classe responsável por criar conexões com o banco
 */
public abstract class ConnectionFactory {
```

O que faz:

- Declara uma classe abstrata com a responsabilidade de fornecer conexões com o banco de dados.
- Ser abstrata impede sua instanciação direta, já que só fornece métodos utilitários.

```
    private static final String DB\_URI = "jdbc:sqlite:lab01.sqlite";
    private static Connection cnx;
    private static SQLiteConfig sqLiteConfig = new SQLiteConfig();
```

O que faz:

- *DB_URI*: especifica o caminho do banco SQLite (armazenado como recurso).
- *cnx*: referência para a conexão com o banco.
- *sqLiteConfig*: configuração para o banco SQLite.

```
    public static synchronized Connection getDBConnection() throws SQLException {
        sqLiteConfig.enforceForeignKeys(true);
        try {
            cnx = DriverManager.getConnection(DB_URI, sqLiteConfig.toProperties());
        } catch (SQLException e) {
            throw new SQLException("Erro ao conectar no banco de dados", e);
        }
        return cnx;
    }
```

O que faz:

- Método estático e sincronizado: garante que apenas uma thread por vez execute esse método.
- Ativa o uso de chaves estrangeiras com *enforceForeignKeys(true)*.
- Tenta obter uma conexão com o banco de dados via *DriverManager*.
- Em caso de erro, lança uma exceção personalizada com uma mensagem mais descritiva.
- Retorna a conexão *cnx*.

Objetivo da classe *Principal*

A classe *Principal* foi criada no **Exemplo 01** para permitir que o usuário interaja com o sistema por meio de um menu simples no terminal. Neste exemplo, ela foi **modificada** para incluir uma nova opção no menu principal: executar o *Exemplo 02*, que demonstra o uso de padrões de projeto (neste caso, o padrão *Factory*) para acessar o banco de dados.

Essas modificações permitem reutilizar a mesma estrutura de menu e facilitar a navegação entre diferentes exemplos e estilos de implementação.

Modificações na classe *Principal*

```
import exemplo01.ExemploMuitoSimples;
import exemplo02.PadroesDeProjeto;
import java.io.IOException;
import java.sql.SQLException;
```

O que faz:

- Permite o uso das classes de ambos os exemplos dentro do mesmo programa.
- Importa exceções necessárias para tratar erros de entrada e banco de dados.

```
private final String[] EXEMPLOS = {
    "\n..:: Pequenos exemplos com Java, SQLite e MySQL ::..\n",
    "1 - Exemplo 01",
    "2 - Exemplo 02 - uso de padrões de projeto",
    "6 - Sair do programa"
};
```

O que faz:

- Adiciona uma nova opção de menu para executar o *Exemplo 02*.

```
private void exemplo02() throws SQLException {
    PadroesDeProjeto app = new PadroesDeProjeto();
    System.out.println(app.listarPessoas());
}
```

O que faz:

- Instancia a classe *PadroesDeProjeto*.
- Chama o método *listarPessoas()*, que acessa o banco de dados usando a *ConnectionFactory*.
- Exibe os dados das pessoas no terminal.

```
case 2:
    p.exemplo02();
    break;
```

O que faz:

- Chama o novo método *exemplo02()* quando o usuário seleciona a opção 2 no menu principal.

Como executar a classe *Principal* e o que esperar

Para executar a classe *Principal*, siga os passos abaixo:

1. Certifique-se de que todas as classes do *Exemplo 01* e do *Exemplo 02* estão compiladas e que a estrutura de pastas está correta, ou seja:
    - Pasta *exemplo01* com a classe *ExemploMuitoSimples*
    - Pasta *exemplo02* com a classe *PadroesDeProjeto* e *ConnectionFactory*
    - Pasta *bcd* com a classe *Principal*
2. Execute a classe *bcd.Principal*. No terminal, o programa apresentará o menu principal com as opções:
    - **1 - Exemplo 01**: permite gerenciar um cadastro simples de pessoas (inserir, alterar, excluir, listar).
    - **2 - Exemplo 02**: demonstra o uso de padrões de projeto para acessar o banco de dados e listar pessoas.
    - **6 - Sair do programa**: encerra a aplicação.

3. Ao escolher a opção 1, seus alunos poderão interagir com o menu do *Exemplo 01*, testando operações básicas de cadastro em banco SQLite.
4. Ao escolher a opção 2, será exibida a lista de pessoas obtida pelo *Exemplo 02*, mostrando como acessar os dados com o padrão *Factory*.
5. Escolher a opção 6 finaliza o programa.

Diferenças entre Exemplo 01 e Exemplo 02

O *Exemplo 01* implementa diretamente a lógica de acesso ao banco de dados, utilizando chamadas JDBC explícitas para criar, ler, atualizar e excluir registros. Essa abordagem é simples, mas pode levar a código repetitivo e difícil de manter em projetos maiores.

O *Exemplo 02* introduz o padrão de projeto *Factory* por meio da classe *ConnectionFactory*, que centraliza a criação das conexões com o banco SQLite. Isso promove melhor organização, encapsulamento e reaproveitamento do código, além de facilitar a aplicação de configurações comuns (exemplo: ativação de restrição de chaves estrangeiras).

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