labWeek8 - atv2

Aluno: Arthur Calciolari

Grupo: Masterclass A, Tutoria A

```
a)
public class ConexaoDB {

    private static ConexaoDB conexao;

    private ConexaoDB() {
    }

    public static synchronized ConexaoDB getInstance() {
        if (conexao == null)
            conexao = new ConexaoDB();

        return conexao;
    }
}
```

> Esse código representa a implementação de um Singleton, um dos padrões de criação estipulados pela *Gang of Four*. O Singleton serve para assegurar que apenas uma instância de determinado objeto seja acessada pelo projeto todo, sendo o ponto principal dessa instância.

> Seus principais problemas são:

1. **Falta de Tratamento de Exceções na Criação da Instância**: Se ocorrer algum erro durante a criação da instância da classe `ConexaoDB`, como problemas de conexão com o banco de dados, não haverá nenhum tratamento de exceção nesse trecho de código. Isso pode resultar em comportamento inesperado ou em falhas silenciosas.
2. **Sincronização Pode Causar Overhead**: O uso de `synchronized` no método `getInstance()` garante que a criação da instância seja thread-safe. No entanto, isso também pode introduzir um overhead de desempenho, especialmente em cenários de alto tráfego, onde vários threads podem competir pelo acesso à instância.
3. **Falta de Injeção de Dependência**: Esse Singleton cria uma forte ligação entre classes, o que pode dificultar a realização de testes unitários e a modificação do código no futuro. A classe `ConexaoDB` não pode ser facilmente substituída por uma implementação diferente, o que é uma desvantagem em termos de flexibilidade e manutenção.
4. **Não Lida com a Liberação de Recursos**: Se a classe `ConexaoDB` estiver gerenciando uma conexão de banco de dados ou outros recursos, é importante garantir que esses recursos sejam liberados corretamente quando não forem mais necessários. O padrão Singleton não aborda automaticamente essa preocupação.
5. **Potencial para Uso Excessivo**: Como Singleton, essa classe pode ser acessada globalmente de qualquer lugar do código, o

> que pode levar a um uso excessivo e não planejado da instância única.

```
b)
public static java.sql.Connection getConnection() {
    Connection connection = null;

    try {
        String driverName = "com.mysql.jdbc.Driver";

        Class.forName(driverName);
        /* Configura os parâmetros da conexão */
        String url = "jdbc:mysql://localhost/jala";
        String username = "root";
        String password = "root";

        /* Tenta se conectar */
        connection = DriverManager.getConnection(url, username, password);

        /* Caso a conexão ocorra com sucesso, a conexão é retornada */
        //System.out.println("OK");

    }  catch (SQLException e) {
        System.out.println("Nao foi possivel conectar ao banco de dados: "+ e.getMessage());

    } catch (ClassNotFoundException e) {
        // TODO Auto-generated catch block
        System.out.println(e.getMessage());
    }
    return connection;
}
```

> Por mais que este código não siga diretamente nenhuma implementação de um padrão de criação, ele é bem similar ao padrão *factory*, já que esta classe serve apenas para criar uma instância de conexão ao banco de dados

> Seus principais problemas são:
>
> 1. **Gestão de Recursos**: O código não trata a liberação da conexão de forma adequada. Não é uma prática recomendada retornar uma conexão aberta da função `getConnection()`, pois

isso pode levar a problemas de vazamento de recursos. A conexão deveria ser fechada adequadamente após o uso, idealmente usando uma estrutura como `try-with-resources` para garantir a liberação automática.

2. **Tratamento de Exceções**: O tratamento de exceções é principalmente voltado para imprimir mensagens de erro no console. Isso pode ser insuficiente em um ambiente de produção. Pode ser mais apropriado relançar ou tratar as exceções de uma maneira que permita ao aplicativo lidar com elas de maneira mais sofisticada.
3. **Dureza no Código**: A impressão direta de mensagens de erro no console pode dificultar a manutenção e depuração posterior. Usar logging (por exemplo, o framework Log4j ou a API de logging do Java) seria uma abordagem melhor para registrar informações relevantes.
4. **Segurança de Senha**: Colocar a senha diretamente no código é uma prática insegura. As credenciais de banco de dados devem ser armazenadas de maneira segura, como em variáveis de ambiente ou arquivos de configuração protegidos.
5. **Arquitetura Geral**: O código não parece fazer parte de uma arquitetura mais ampla. Em sistemas mais complexos, a separação de responsabilidades, como a gestão de conexões, pode ser abordada por meio de frameworks de persistência ou injeção de dependência.
6. **Escalabilidade**: A conexão direta ao banco de dados pode não ser escalável para sistemas maiores. A utilização de um pool

> de conexões, por exemplo, pode ser mais adequada para gerenciar eficientemente o acesso ao banco de dados.