

# JDBC

**PROFESSOR: LEANDRO MAIA**  
**DISCIPLINA: PROG. ORIENTADA A OBJETOS**



# INTRODUÇÃO

## JDBC

1/1

JDBC (Java Database Connectivity) é uma API Java padrão para acessar bancos de dados relacionais. Ela permite que aplicações Java se conectem, executem consultas e atualizem dados em diversos SGBDs (Sistemas Gerenciadores de Bancos de Dados) independente do fornecedor do banco de dados.





## JDBC

### ARQUITETURA

1/2

Uma aplicação que utiliza JDBC possui em sua arquitetura três componentes:

- **Drivers JDBC:** fornecidos pelos fabricantes de SGBDs, traduzem as chamadas JDBC para o formato específico do banco de dados.
- **API JDBC:** conjunto de classes e interfaces Java que permitem interagir com os drivers e o banco de dados.



## JDBC

### ARQUITETURA

2/2

Uma aplicação que utiliza JDBC possui em sua arquitetura três componentes:

- **Aplicação Java:** utiliza a API JDBC para se comunicar com o banco de dados.

# VANTAGENS E DESVANTAGENS

## VANTAGENS

1/8

O uso de JDBC apresenta algumas vantagens:

- **Portabilidade:** Uma das maiores vantagens do JDBC é sua portabilidade. Uma vez que você escreve o código utilizando a API JDBC, ele pode ser executado em qualquer banco de dados que possua um driver JDBC compatível, sem a necessidade de reescrever a lógica de acesso aos dados.



# VANTAGENS E DESVANTAGENS

## VANTAGENS

2/8

O uso de JDBC apresenta algumas vantagens:

- **Controle Total:** O JDBC oferece controle total sobre as operações de banco de dados, permitindo que você escreva consultas SQL complexas e otimizadas, execute stored procedures e gerencie transações de forma granular.





# VANTAGENS E DESVANTAGENS

## VANTAGENS

3/8

O uso de JDBC apresenta algumas vantagens:

- **Flexibilidade:** A API JDBC é flexível e extensível, permitindo que você trabalhe com diferentes tipos de dados, execute operações em lote, acesse metadados do banco de dados e personalize o comportamento do driver.



# VANTAGENS E DESVANTAGENS

## VANTAGENS

4/8

O uso de JDBC apresenta algumas vantagens:

- **Desempenho:** Em cenários onde o desempenho é crítico e você precisa de controle total sobre as consultas SQL, o JDBC pode oferecer melhor performance do que frameworks ORM, que podem adicionar overhead devido às suas abstrações.





# VANTAGENS E DESVANTAGENS

## VANTAGENS

5/8

O uso de JDBC apresenta algumas vantagens:

- **Acesso a Recursos Específicos do SGBD:** O JDBC permite que você acesse recursos específicos do seu Sistema Gerenciador de Banco de Dados (SGBD) que podem não ser suportados por frameworks ORM.



# VANTAGENS E DESVANTAGENS

## VANTAGENS

6/8

O uso de JDBC apresenta algumas vantagens:

- **Curva de Aprendizado Relativamente Baixa:** A API JDBC em si é relativamente simples de aprender, especialmente se você já possui familiaridade com SQL e conceitos de bancos de dados relacionais.



# VANTAGENS E DESVANTAGENS

## VANTAGENS

7/8

O uso de JDBC apresenta algumas vantagens:

- **Adequado para Projetos Pequenos e Médios:** Para projetos menores ou com requisitos de acesso a dados mais simples, o JDBC pode ser uma solução mais leve e direta do que frameworks ORM mais complexos.





# VANTAGENS E DESVANTAGENS

## VANTAGENS

8/8

O uso de JDBC apresenta algumas vantagens:

- **Comunidade Ativa e Documentação:** O JDBC possui uma comunidade ativa e extensa documentação disponível, o que facilita a resolução de problemas e o aprendizado.



# VANTAGENS E DESVANTAGENS

## DESVANTAGENS

1/6

JDBC também possui desvantagens importantes:

- **Código Boilerplate:** O JDBC exige a escrita de bastante código repetitivo para tarefas comuns, como abrir e fechar conexões, criar *statements*, iterar pelos resultados, etc. Isso pode tornar o código mais verboso e difícil de manter.



# VANTAGENS E DESVANTAGENS

## DESVANTAGENS

2/6

JDBC também possui desvantagens importantes:

- **Acoplamento com o Banco de Dados:** O código JDBC está diretamente ligado à estrutura do banco de dados e ao SQL específico do SGBD utilizado. Mudanças no esquema do banco podem exigir alterações significativas no código da aplicação.





# VANTAGENS E DESVANTAGENS

## DESVANTAGENS

3/6

JDBC também possui desvantagens importantes:

- **Vulnerabilidade a SQL Injection:** Se não utilizado corretamente, o JDBC pode abrir brechas para ataques de SQL injection, onde um usuário malicioso insere código SQL malicioso na aplicação. O uso de *prepared statements* é crucial para evitar esse problema.



# VANTAGENS E DESVANTAGENS

## DESVANTAGENS

4/6

JDBC também possui desvantagens importantes:

- **Complexidade em Consultas Complexas:**

Consultas SQL complexas podem se tornar difíceis de gerenciar e manter diretamente no código Java.

*Frameworks* ORM podem oferecer abstrações e ferramentas para lidar com essas situações de forma mais elegante.



# VANTAGENS E DESVANTAGENS

## DESVANTAGENS

5/6

JDBC também possui desvantagens importantes:

- **Curva de Aprendizado:** Embora a API JDBC seja relativamente simples, dominar o SQL e entender os conceitos de bancos de dados relacionais exige um certo conhecimento técnico.





# VANTAGENS E DESVANTAGENS

## DESVANTAGENS

6/6

JDBC também possui desvantagens importantes:

- **Menor Produtividade:** Comparado a *frameworks* ORM, o desenvolvimento com JDBC puro pode ser menos produtivo devido à necessidade de escrever mais código e lidar com detalhes de baixo nível.





## COMPONENTES PRINCIPAIS

1/2

- **DriverManager:** gerencia os drivers JDBC e estabelece conexões com o banco de dados.
- **Connection:** representa uma sessão com o banco de dados.
- **Statement:** usado para executar consultas SQL simples.



## COMPONENTES PRINCIPAIS

2/2

- **PreparedStatement:** usado para executar consultas SQL parametrizadas, mais seguro contra SQL injection.
- **CallableStatement:** usado para executar *stored procedures*.
- **ResultSet:** contém os resultados de uma consulta SQL.





## CONEXÃO VIA JDBC

1/2

Para estabelecer uma conexão com o SGBD via JDBC são necessários dois passos:

- **Carregar o driver JDBC:** Inicia o *driver* responsável por fazer a implementação dos métodos para o SGBD específico. **Isso é opcional hoje em dia!**

Exemplo:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```



## CONEXÃO VIA JDBC

2/2

Para estabelecer uma conexão com o SGBD via JDBC são necessários dois passos:

- **Obter uma conexão:** Estabelecer uma conexão com o SGBD para realizar as ações demandadas.

Exemplo:

```
Connection con = DriverManager.getConnection(url,  
usuario, senha);
```



## CONSULTAS

### STATEMENT

1/4

Uma vez estabelecida uma conexão com o BD, o próximo passo é executar uma consulta. A forma mais simples é criar um Statement e em seguida executar uma *query*.

Dependendo do tipo de *query*, o tipo de retorno também varia. É importante ficar atento a isso.



## CONSULTAS STATEMENT

2/4

Se a sua consulta for um *select*, você deve chamar o método `executeQuery` que retornará um `ResultSet`. O próximo passo é iterar nos elementos do `ResultSet` de forma a obter os dados desejados.

# CONSULTAS

## STATEMENT

3/4

```
public class JDBC {

    final private static String MYSQL_JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
    final private static int DB_PORT = 3306;
    final private static String DB_HOST = "localhost";
    final private static String DB_NAME = "coltec";

    public static void main(String[] args) {

        // Configurações de conexão
        String url = "jdbc:mysql:// " + DB_HOST + ":" + DB_PORT + "/" + DB_NAME;
        String usuario = "root";
        String senha = "senhaSuperSegura";

        // Carrega o Driver do JDBC MYSQL
        // Totalmente opcional, mas se o driver não estiver presente, já encerra o programa logo
        // não deixando pra depois a descoberta de que você não tem o driver
        try {
            Class.forName(MYSQL_JDBC_DRIVER);
        } catch (ClassNotFoundException ex) {
            System.err.println("Falha ao carregar o Driver do JDBC MySQL");
            return;
        }
    }
}
```



# CONSULTAS

## STATEMENT

4/4

```
try {
    // Estabelece uma conexão
    Connection conexao = DriverManager.getConnection(url, usuario, senha);

    // Cria um statement para execução da query e executa
    Statement stmt = conexao.createStatement();
    ResultSet tabelaAlunos = stmt.executeQuery("SELECT * FROM alunos");

    // Processar os resultados
    while (tabelaAlunos.next()) {
        System.out.println(tabelaAlunos.getInt("matricula") + " | " + tabelaAlunos.getString("nome") + " | " +
tabelaAlunos.getString("senha"));
    }

    // Encerra os recursos utilizados
    tabelaAlunos.close();
    stmt.close();
    conexao.close();

} catch (SQLException e) {
    System.out.println("Erro na conexão com o banco de dados: " + e.getMessage());
}

}
```



## CONSULTAS

### PREPAREDSTATEMENT

1/2

Algumas vezes é interessante já deixar seu *statement* já pronto, aguardando apenas os dados. Isso além de melhorar seu código, deixa ele menos suscetível a ataques como SQL Injection.

Para isso você cria um PreparedStatement e depois só atualiza os valores na hora de executar a *query*.

# CONSULTAS

## PREPAREDSTATEMENT

2/2

```
// Excerto do código de testes do JDB
```

```
// Cria um novo registro
```

```
PreparedStatement pstmt = conexao.prepareStatement("INSERT INTO alunos (matricula, nome, senha) VALUES (?, ?, ?)");
```

```
pstmt.setInt(1, 2024950203);
```

```
pstmt.setString(2, "Leandro Silva");
```

```
pstmt.setString(3, "6d67d51dc8c954735d961b377b4a2544"); // MD5("senhaQuaseSegura")
```

```
pstmt.executeUpdate();
```

```
// Encerra os recursos utilizado
```

```
pstmt.close();
```

```
conexao.close();
```

## EXTRAS

### FORA DO ESCOPO

O JDBC também pode ser utilizado para execução de StoredProcedures através da CallableStatement e também de transações através dos métodos setAutoCommit, commit e rollback da conexão.

Não é complicado, mas está fora do nosso escopo no momento.





## **BOAS PRÁTICAS**

### **USAR PREPAREDSTATEMENT**

1/4

Utilize PreparedStatement sempre que possível. Além de deixar o texto da declaração separado dos valores, ainda ajuda a mitigar os possíveis ataques de SQL Injection.





## **BOAS PRÁTICAS**

### **LIBERE OS RECURSOS**

**2/4**

Os exemplos não fizeram porque nada será feito depois e o programa será encerrado, mas as boas práticas pedem que seus ResultSets, Statements, PreparedStatement e Connections sempre em blocos Finally para evitar que elas fiquem abertas em caso de exceções levantadas.



## BOAS PRÁTICAS

### POOL DE CONEXÕES

3/4

O processo de estabelecer uma conexão é custoso. Por essa razão, para melhorar o desempenho onde há muitas requisições (geralmente servidores), cria um *pool* de conexões e vá utilizando a medida que são solicitadas/liberadas. Isso permite alocar o recurso sem necessidade de estabelecer uma nova conexão toda vez.



## BOAS PRÁTICAS

### FRAMEWORKS ORM

4/4

Em projetos grandes, pode ser bem trabalhoso criar todos os mapeamentos entre o banco e as classes. Para isso, existem diversos *frameworks* (Hibernate, EclipseLink, MyBatis e etc) que fazem o mapeamento ORM (*Object/Relational Mapping*) para você.