

JDBC e Manipulação de Bancos de Dados com Java

Conectando aplicações Java a bancos de dados relacionais

1. Introdução

JDBC (*Java Database Connectivity*) é uma API padrão do Java que permite a aplicações se conectarem e interagirem com bancos de dados relacionais, como MySQL, PostgreSQL, Oracle e SQL Server. Por meio do JDBC, é possível executar comandos SQL diretamente do código Java — consultas, inserções, atualizações e exclusões de dados — de forma padronizada, independentemente do banco de dados utilizado.

2. Como o JDBC Funciona

O JDBC atua como uma camada intermediária entre a aplicação Java e o banco de dados. A aplicação utiliza as interfaces padrão do pacote `java.sql`, enquanto um **driver JDBC** específico do banco de dados (geralmente distribuído como um arquivo `.jar`) traduz essas chamadas para o protocolo de comunicação do banco.

Etapas	Interface/Classe	Responsabilidade
1	<code>DriverManager</code>	Localiza e carrega o driver adequado para a URL informada.
2	<code>Connection</code>	Representa a conexão ativa com o banco de dados.
3	<code>Statement</code> / <code>PreparedStatement</code>	Envia comandos SQL para o banco de dados.
4	<code>ResultSet</code>	Armazena e percorre os resultados de uma consulta.

3. Estabelecendo uma Conexão

A conexão com o banco é obtida por meio de `DriverManager.getConnection()`, informando uma **URL de conexão** (que identifica o banco, o servidor e a porta), além de usuário e senha, quando exigidos.

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

public class ConexaoBanco {
    private static final String URL = "jdbc:mysql://localhost:3306/escola";
    private static final String USUARIO = "root";
    private static final String SENHA = "senha123";

    public static Connection conectar() throws SQLException {
        return DriverManager.getConnection(URL, USUARIO, SENHA);
    }
}
```

Todo método que interage com o banco de dados por meio do JDBC pode lançar uma SQLException — uma exceção verificada (checked exception) que deve ser tratada com try/catch ou declarada com throws.

4. Fechando Recursos com try-with-resources

Conexões, statements e result sets consomem recursos do sistema e **precisam ser fechados** após o uso. A forma recomendada em Java moderno é o bloco try-with-resources, que fecha automaticamente todos os recursos declarados entre parênteses, mesmo que ocorra uma exceção.

```
import java.sql.*;

public class ExemploConsulta {
    public static void main(String[] args) {
        String sql = "SELECT id, nome FROM aluno";

        try (Connection conn = ConexaoBanco.conectar();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql)) {

            while (rs.next()) {
                int id = rs.getInt("id");
                String nome = rs.getString("nome");
                System.out.println(id + " - " + nome);
            }

        } catch (SQLException e) {
            System.out.println("Erro ao consultar: " + e.getMessage());
        }
    }
}
```

5. PreparedStatement e Prevenção de SQL Injection

Embora Statement permita executar comandos SQL simples, ele é vulnerável a **SQL Injection** quando os valores vêm de entradas do usuário e são concatenados diretamente na string SQL. A classe PreparedStatement resolve esse problema usando **parâmetros posicionais** (representados por ?), que são tratados como valores, e não como parte do comando SQL.

```
// Forma vulnerável (NÃO recomendada):
// String sql = "SELECT * FROM aluno WHERE nome = '" + nomeDigitado + "'";
// Um valor malicioso poderia alterar o comportamento da consulta.

// Forma segura, com PreparedStatement:
String sql = "SELECT id, nome FROM aluno WHERE nome = ?";

try (Connection conn = ConexaoBanco.conectar();
    PreparedStatement stmt = conn.prepareStatement(sql)) {

    stmt.setString(1, nomeDigitado); // define o valor do primeiro "?"

    try (ResultSet rs = stmt.executeQuery()) {
        while (rs.next()) {
            System.out.println(rs.getString("nome"));
        }
    }

} catch (SQLException e) {
    System.out.println("Erro na consulta: " + e.getMessage());
}
```

6. Operações CRUD com PreparedStatement

As quatro operações básicas de manipulação de dados — **Create**, **Read**, **Update** e **Delete** (CRUD) — podem ser implementadas de forma direta com `PreparedStatement`. A seguir, uma classe `AlunoDAO` (*Data Access Object*) que encapsula essas operações para uma tabela `aluno`.

```

import java.sql.*;

public class AlunoDAO {

    // CREATE
    public void inserir(String nome, double media) throws SQLException {
        String sql = "INSERT INTO aluno (nome, media) VALUES (?, ?)";
        try (Connection conn = ConexaoBanco.conectar();
            PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setString(1, nome);
            stmt.setDouble(2, media);
            stmt.executeUpdate();
        }
    }

    // UPDATE
    public void atualizarMedia(int id, double novaMedia) throws SQLException {
        String sql = "UPDATE aluno SET media = ? WHERE id = ?";
        try (Connection conn = ConexaoBanco.conectar();
            PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setDouble(1, novaMedia);
            stmt.setInt(2, id);
            stmt.executeUpdate();
        }
    }

    // DELETE
    public void remover(int id) throws SQLException {
        String sql = "DELETE FROM aluno WHERE id = ?";
        try (Connection conn = ConexaoBanco.conectar();
            PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setInt(1, id);
            stmt.executeUpdate();
        }
    }
}

```

Note que executeUpdate() é usado para comandos que modificam dados (INSERT, UPDATE, DELETE), retornando o número de linhas afetadas, enquanto executeQuery() é usado para consultas (SELECT), retornando um ResultSet.

6.1. READ: Convertendo Linhas em Objetos

É comum mapear cada linha retornada por uma consulta para um objeto de uma classe do domínio, como Aluno, facilitando o uso dos dados no restante da aplicação.

```

import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class AlunoDAO {
    // ... métodos anteriores ...

    public List<Aluno> listarTodos() throws SQLException {
        String sql = "SELECT id, nome, media FROM aluno";
        List<Aluno> alunos = new ArrayList<>();

        try (Connection conn = ConexaoBanco.conectar();
            PreparedStatement stmt = conn.prepareStatement(sql);
            ResultSet rs = stmt.executeQuery()) {

            while (rs.next()) {
                Aluno aluno = new Aluno(
                    rs.getInt("id"),
                    rs.getString("nome"),
                    rs.getDouble("media")
                );
                alunos.add(aluno);
            }
        }
        return alunos;
    }
}

```

7. Transações

Quando várias operações precisam ser executadas como uma **unidade atômica** (todas aplicadas com sucesso ou nenhuma delas), utiliza-se o controle manual de transações, desativando o modo de *auto-commit* e chamando `commit()` ou `rollback()` conforme o resultado das operações.

```

try (Connection conn = ConexaoBanco.conectar()) {
    conn.setAutoCommit(false); // inicia controle manual da transação

    try (PreparedStatement stmt1 = conn.prepareStatement(
        "UPDATE conta SET saldo = saldo - ? WHERE id = ?");
        PreparedStatement stmt2 = conn.prepareStatement(
            "UPDATE conta SET saldo = saldo + ? WHERE id = ?")) {

        stmt1.setDouble(1, 100.0);
        stmt1.setInt(2, 1);
        stmt1.executeUpdate();

        stmt2.setDouble(1, 100.0);
        stmt2.setInt(2, 2);
        stmt2.executeUpdate();

        conn.commit(); // confirma as duas operações juntas
    } catch (SQLException e) {
        conn.rollback(); // desfaz tudo em caso de erro
    }
} catch (SQLException e) {
    System.out.println("Erro de conexão: " + e.getMessage());
}

```

8. Boas Práticas

- Prefira sempre `PreparedStatement` a `Statement` quando algum valor da consulta vier de uma entrada externa, para evitar SQL Injection.
- Utilize `try-with-resources` para garantir o fechamento correto de conexões, statements e result sets, mesmo em caso de exceção.
- Separe a lógica de acesso a dados em classes específicas (padrão **DAO**), mantendo o restante da aplicação independente dos detalhes de SQL e JDBC.
- Utilize um **pool de conexões** (como HikariCP) em aplicações reais, evitando abrir e fechar conexões a todo momento, o que é custoso em termos de desempenho.

9. Conclusão

O JDBC é a base sobre a qual praticamente todo acesso a bancos de dados relacionais em Java é construído, seja diretamente, seja por meio de frameworks de mais alto nível como o Hibernate ou o Spring Data, que utilizam o JDBC internamente. Dominar seus conceitos fundamentais — conexão, execução de comandos com `PreparedStatement`, leitura de resultados com `ResultSet` e controle de transações — é essencial para construir aplicações Java robustas e seguras que interagem com bancos de dados.