

# DESCRIÇÃO

Utilização de tecnologia Java, com base no middleware JDBC, para manipulação e consulta aos dados de bancos relacionais, com base em comandos diretos, ou por meio de mapeamento objeto-relacional.

# PROPÓSITO

Implementar os diversos processos de persistência necessários aos sistemas cadastrais, com base em modelos robustos de programação, permitindo satisfazer a uma demanda de mercado natural na construção de sistemas com esse perfil. Ao final dos estudos, você estará apto a construir sistemas com acesso a bancos de dados relacionais, utilizando tecnologia Java.

# PREPARAÇÃO

Antes de iniciar o conteúdo, é necessário configurar o ambiente, com a instalação do JDK e Apache NetBeans, definindo a plataforma de desenvolvimento que será utilizada na codificação e execução dos exemplos práticos.

# OBJETIVOS

## MÓDULO 1

Descrever os recursos para acesso a banco de dados no ambiente Java

## MÓDULO 2

Descrever o modelo de persistência baseado em mapeamento objeto-relacional

## MÓDULO 3

Aplicar tecnologia Java para a viabilização da persistência em banco de dados

# INTRODUÇÃO

Neste conteúdo, analisaremos as ferramentas para acesso a banco de dados com uso das tecnologias JDBC e JPA, pertencentes ao ambiente Java, incluindo exemplos de código para efetuar consulta e manipulação de registros.

Após compreender os princípios funcionais das tecnologias sob análise, construiremos um sistema cadastral simples e veremos como aproveitar os recursos de automatização do NetBeans para construção de componentes JPA.

## MÓDULO 1

---

🕒 Descrever os recursos para acesso a banco de dados no ambiente Java

## MIDDLEWARE

Para definirmos **middleware**, devemos entender os conceitos de **front-end** e **back-end**.

Veja suas definições:

## FRONT-END

Quando falamos de **front-end**, estamos nos referindo à camada de software responsável pelo interfaceamento do sistema, com o uso de uma linguagem de programação. Aqui, utilizaremos os aplicativos Java como opção de front-end.

## BACK-END

Já o **back-end** compreende o conjunto de tecnologias com a finalidade de fornecer recursos específicos, devendo ser acessadas a partir de nosso front-end, embora não façam parte do mesmo ambiente, como os bancos de dados e as mensagerias. Para nossos exemplos, adotaremos o **banco de dados** Derby como back-end.

## SAIBA MAIS

As **mensagerias** são outro bom exemplo de back-end, com uma arquitetura voltada para a comunicação assíncrona entre sistemas, efetuada por meio da troca de mensagens. Essa é uma tecnologia crucial para diversos sistemas corporativos, como os da rede bancária.

Um grande problema, enfrentado pelas linguagens de programação mais antigas, é que deveríamos ter versões específicas do programa para acesso a cada tipo de servidor de banco de dados, como Oracle, Informix, DB2 e SQL Server, entre diversos outros, o que também ocorria com relação aos sistemas de mensagerias, em que podemos citar, como exemplos, MQ Series, JBossMQ, ActiveMQ e Microsoft MQ.

Com diferentes componentes para acesso e modelos de programação heterogêneos, a probabilidade de ocorrência de erros é simplesmente enorme, levando à necessidade de uma camada de software intermediária, responsável por promover a comunicação entre o **front-end** e o **back-end**.

Foi definido o termo **middleware** para a classificação desse tipo de tecnologia, que permite integração de forma transparente e mudança de fornecedor com pouca ou nenhuma alteração de código.

No ambiente Java, temos o **JDBC** (Java Database Connectivity) como middleware para acesso aos diferentes tipos de bancos de dados. Ele permite que utilizemos produtos de diversos fornecedores, sem modificações no código do aplicativo, sendo a consulta e manipulação de dados efetuadas por meio de comandos **SQL** (Structured Query Language) em meio ao código Java.

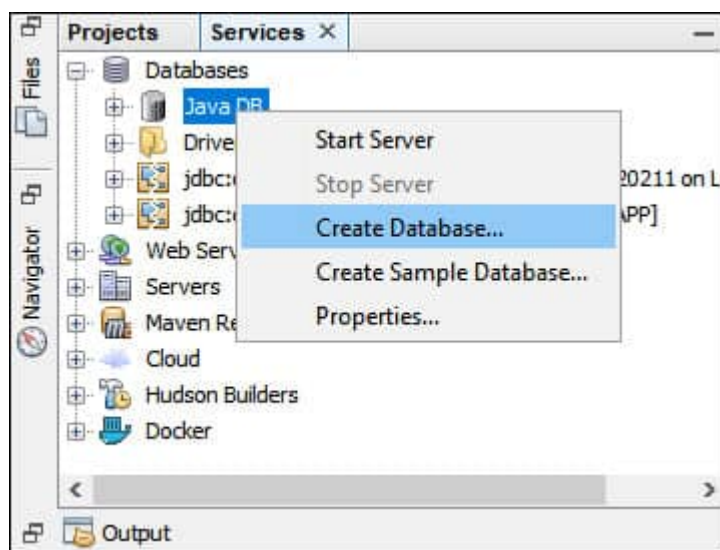
## ATENÇÃO

Devemos evitar a utilização de comandos para um tipo de banco de dados específico, mantendo sempre a sintaxe padronizada pelo **SQL ANSI**, pois, caso contrário, a mudança do fornecedor de back-end poderá exigir mudanças no front-end.

## BANCO DE DADOS DERBY

Entre as diversas opções de repositórios existentes, temos o **Derby**, ou **Java DB**, um banco de dados relacional construído totalmente com tecnologia Java, que não depende de um servidor e faz parte da distribuição padrão do **JDK**. Apache Derby é um subprojeto do Apache DB, disponível sob licença Apache, e que pode ser embutido em programas Java, bem como utilizado para transações on-line.

Podemos gerenciar nossos bancos de dados Derby de forma muito simples, por meio da aba **Services** do **NetBeans**, na divisão **Databases**. Para isso, devemos clicar com o botão direito sobre o driver **Java DB**, escolher a opção **Create Database**, como ilustrado na imagem, e preencher as informações necessárias para a configuração da nova instância do banco de dados:

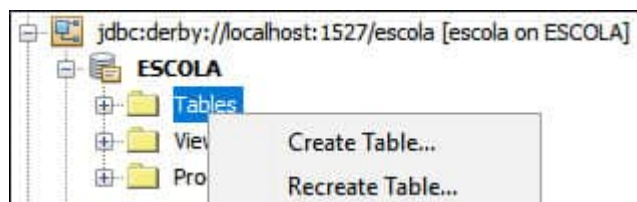


 Captura de tela da criação de um banco de dados Java DB utilizando o NetBeans.

Vamos agora criar um banco de dados Java DB chamado “escola”. Para isso, na janela que será aberta, devemos preencher o nome de nosso novo banco de dados com o valor "escola", bem como usuário e senha, onde seria interessante defini-los também como "escola", tornando

mais fácil lembrar os valores utilizados. Ao clicar no botão de confirmação, o banco de dados será criado e ficará disponível para conexão, sendo identificado por sua Connection String, a qual é formada a partir do endereço de rede (localhost), porta padrão (1527) e instância que será utilizada (escola).

A conexão é aberta com o duplo clique sobre o identificador, ou o clique com o botão direito e escolha da opção **Connect**. Com o banco de dados aberto, podemos criar uma tabela, navegando até a divisão **Tables**, no esquema **ESCOLA**, e utilizando o clique com o botão direito, para acessar a opção **Create Table** no menu de contexto.



📷 Captura de tela da criação de uma tabela no banco de dados.

Na janela que se abrirá, iremos configurar a tabela **Aluno**, com os campos definidos de acordo com o quadro seguinte.

Campo	Tipo	Complemento
<b>MATRICULA</b>	VARCHAR	Tamanho: 10 e Chave primária
<b>NOME</b>	VARCHAR	Tamanho: 40
<b>ENTRADA</b>	INTEGER	Sem atributos complementares

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Campos da tabela Aluno.

Elaborado por: Denis Gonçalves Cople.

Definindo o nome da tabela e adicionando os campos, teremos a configuração que pode ser observada a seguir, com o processo sendo finalizado por meio do clique em **OK**. Cada campo deve ser adicionado individualmente, com o clique em **Add Column**.

Table name:

Key	Index	Null	Unique	Column name	Data type	Size
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	MATRICULA	VARCHAR	10
<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NOME	VARCHAR	40
<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	ENTRADA	INTEGER	0

Buttons: Add column, Edit, Remove, Move Up, Move Down, OK, Cancel, Help

📷 Captura de tela da ferramenta de criação visual para tabelas.

A tabela criada será acessada por meio de um novo nó, na árvore de navegação, abaixo de **Tables**, com o nome **Aluno**. Utilizando o clique com o botão direito sobre o novo nó e escolhendo a opção **View Data**, teremos a abertura de uma área para execução de **SQL** e visualização de dados no editor de código, sendo possível acrescentar registros de forma visual com o uso de **ALT+I**, ou clique sobre o ícone referente.

## 💡 DICA

Experimente acrescentar alguns registros, na janela de inserção que será aberta com as teclas **ALT+I**, utilizando **Add Row** para abrir novas linhas e preenchendo os valores dos campos para cada linha.

Ao final do preenchimento dos dados, devemos clicar em **OK** para finalizar, e o NetBeans executará os comandos **INSERT** necessários.

Press CTRL+Tab to exit data entry mode from the table. Press CTRL+0 to set NULL value and CTRL+1 to set DEFAULT value for a given column.

#	MATRICULA	NOME	ENTRADA
1	E2020.0342	ANA MARIA	2020
2	M2020.4002	JOAO CALDEIRAS	2020
3	E2019.0987	LUIZ SAMPAIO	2019

Buttons: Show SQL, Add Row, Remove, OK, Cancel

📷 Captura de tela da ferramenta visual para inserção de registros.

# JAVA DATABASE CONNECTIVITY (JDBC)

Com o banco criado, podemos codificar o front-end em Java, utilizando os componentes do middleware **JDBC**, os quais são disponibilizados com a importação do pacote **java.sql**, tendo como elementos principais as classes DriverManager, Connection, Statement, PreparedStatement e ResultSet.

Existem quatro tipos de **drivers** JDBC, os quais são apresentados no quadro seguinte:

Tipo de Driver	Descrição
1 - JDBC-ODBC Bridge	Faz a conexão por meio do <b>ODBC</b> .
2 - JDBC-Native API	Utiliza o <b>cliente</b> do banco de dados para a conexão.
3 - JDBC-Net	Acessa servidores de middleware via <b>Sockets</b> , em uma arquitetura de três camadas.
4 - Pure Java	Com a implementação completa em Java, não precisa de cliente instalado. Também chamado de <b>Thin Driver</b> .

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Tipos de driver para JDBC.

Elaborado por: Denis Gonçalves Cople.

## ★ EXEMPLO

Se considerarmos o caso específico do **Oracle**, são disponibilizados um driver **Tipo 2**, com acesso via **OCI** (Oracle Call Interface), e um driver **Tipo 4**, com o nome **Oracle Thin Driver**. A

utilização da versão thin driver é mais indicada, pois não necessita da instalação do Oracle Cliente na máquina do usuário.

O processo para utilizar as funcionalidades básicas do JDBC segue quatro passos simples:

Instanciar a classe do driver de conexão;

Obter uma conexão (**Connection**) a partir da Connection String, usuário e senha;

Instanciar um executor de SQL (**Statement**);

Executar os comandos **DML** (linguagem de manipulação de dados).

Por exemplo, se quisermos efetuar a inclusão de um aluno na base de dados, podemos escrever o trecho de código apresentado a seguir, com o objetivo de demonstrar os passos descritos anteriormente:

// passo 1

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

// passo 2

```
Connection c1 = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/escola",
    "escola", "escola");
```

// passo 3

```
Statement st = c1.createStatement();
```

// passo 4

```
st.executeUpdate(
    "INSERT INTO ALUNO VALUES('E2018.5004','DENIS',2018)");
st.close();
c1.close();
```

No início do código, temos a instância do driver Derby sendo gerada a partir de uma chamada para o método **forName**. Com o driver na memória, podemos abrir conexões com o banco de dados por meio do middleware **JDBC**.

Em seguida, é instanciada a conexão **c1**, por meio da chamada ao método **getConnection**, da classe **DriverManager**, sendo fornecidos os valores para **Connection String**, **usuário** e



**senha**. Com relação à Connection String, ela pode variar muito, sendo iniciada pelo driver utilizado, seguido dos parâmetros específicos para aquele driver, os quais, no caso do Derby, são o endereço de rede, a porta e o nome da instância de banco de dados.

A partir da conexão **c1**, é gerado um executor de SQL de nome **st**, com a chamada para o método **createStatement**. Estando o executor instanciado, realizamos a inserção de um registro, invocando o método **executeUpdate**, com o comando SQL apropriado.

Na parte final, devemos fechar os componentes JDBC, na ordem inversa daquela em que foram criados, já que existe dependência sucessiva entre eles.

## ATENÇÃO

As consultas ao banco são efetuadas utilizando **executeQuery**, enquanto comandos para manipulação de dados são executados por meio de **executeUpdate**.

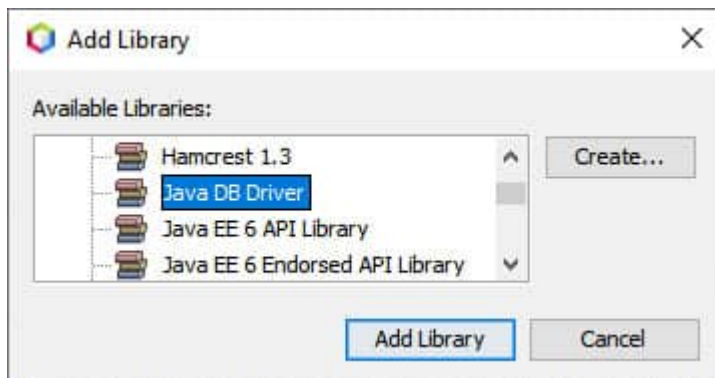
Para o comando de seleção existe mais um detalhe, que seria a recepção da consulta em um **ResultSet**, o que pode ser observado no trecho de código seguinte, no qual, com base na tabela criada anteriormente, efetuamos uma consulta aos dados inseridos:

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
Connection c1 = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/loja",
    "loja", "loja");
Statement st = c1.createStatement();
ResultSet r1 = st.executeQuery("SELECT * FROM ALUNO");
while(r1.next())
    System.out.println("Aluno: " + r1.getString("NOME")+
        " - " + r1.getInt("ENTRADA"));
r1.close();
st.close();
c1.close();
```

Após a recepção da consulta no objeto de nome **r1**, podemos nos movimentar pelos registros, com o uso de **next**, e acessar cada campo pelo nome, para a obtenção do valor, sempre lembrando de utilizar o método correto para o tipo do campo, como **getString**, para texto, e **getInt**, para valores numéricos inteiros.

Ao efetuar a consulta, o **ResultSet** fica posicionado antes do primeiro registro, na posição **BOF** (Beginning of File), e com o uso do comando **next** podemos mover para as posições seguintes, até atingir o final da consulta, na posição **EOF** (End of File). A cada registro visitado, efetuamos a impressão do nome do aluno e ano de entrada.

A biblioteca JDBC deve ser adicionada ao projeto, ou ocorrerá erro durante a execução. Para o nosso exemplo, devemos adicionar a biblioteca **Java DB Driver**, por meio do clique com o botão direito na divisão **Libraries** e uso da opção **Add Library**.



📷 Captura de tela da adição do driver Java DB ao projeto.

Um recurso muito interessante, oferecido por meio do componente **PreparedStatement**, é a definição de comandos SQL parametrizados, os quais são particularmente úteis no tratamento de datas, já que os bancos de dados apresentam interpretações diferentes para esse tipo de dado, e quem se torna responsável pela conversão para o formato correto é o próprio JDBC. Podemos observar um exemplo da utilização do componente no trecho de código seguinte.

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
Connection c1 = DriverManager.getConnection(
    "jdbc:derby://localhost:1527/loja",
    "loja", "loja");
PreparedStatement ps = c1.prepareStatement(
    "DELETE FROM ALUNO WHERE ENTRADA = ?");
ps.setInt(1,2018);
ps.executeUpdate();
ps.close();
c1.close();
```

O uso de parâmetros facilita a escrita do comando SQL, sem a preocupação com o uso de apóstrofe ou outro delimitador, além de representar uma proteção contra os ataques do tipo **SQL Injection**.

# SQL INJECTION

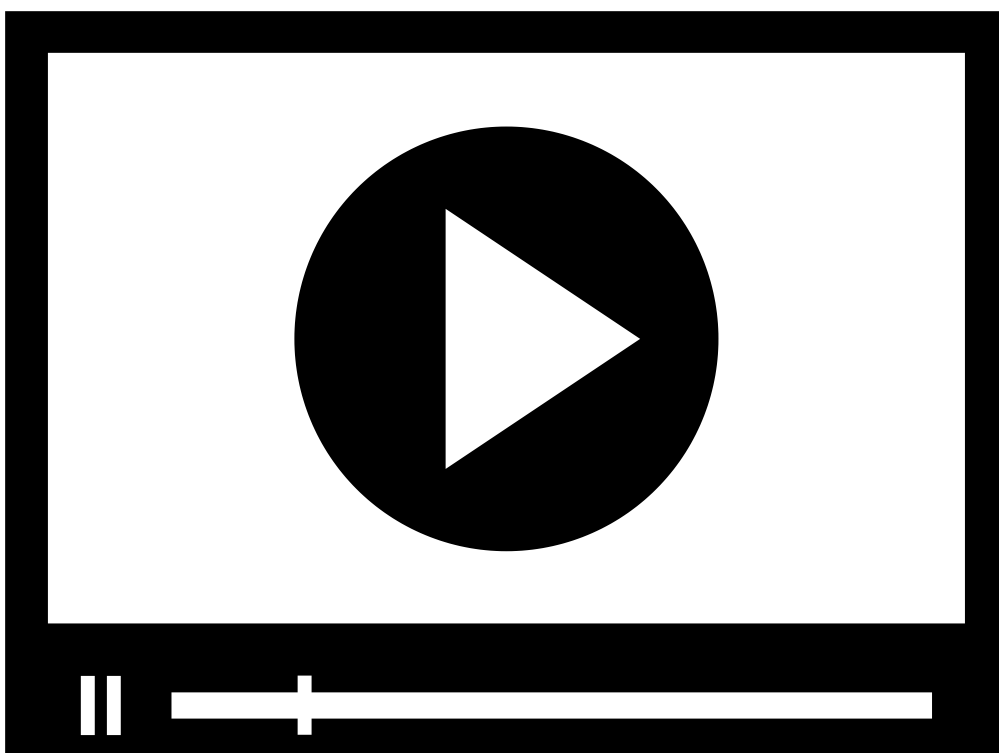
É um tipo de ataque a uma aplicação web, baseado na injeção e execução de instruções SQL mal-intencionadas, que é a linguagem utilizada para troca de informações entre aplicações e bancos de dados relacionais. O SQL Injection visa comprometer a base de dados associada à aplicação web atacada por ele.

Para definir os parâmetros, utilizamos pontos de **interrogação**, os quais assumem valores posicionais, a partir de **um**, o que é um pouco diferente da indexação dos vetores, que começa em zero.

## ATENÇÃO

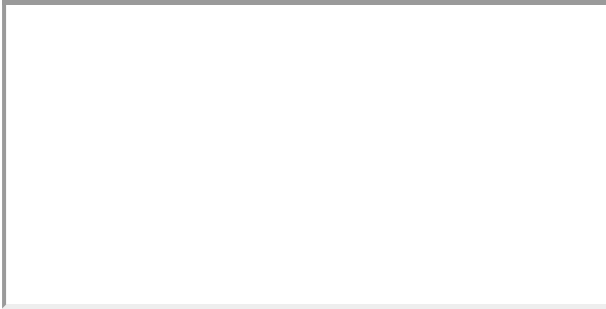
Cada parâmetro deve ser preenchido com a chamada ao método correto, de acordo com seu tipo, como **setInt**, para inteiro, e **setString**, para texto. Após o preenchimento, devemos executar o comando SQL, com a chamada para **executeUpdate**, no caso das instruções **INSERT**, **UPDATE** e **DELETE**, ou **executeQuery**, para a instrução **SELECT**.

No exemplo, o parâmetro foi preenchido com o valor 2018, e a execução do comando SQL resultante irá remover da base todos os alunos com entrada no referido ano.



# COMPONENTES DO JDBC

Chegamos ao fim do módulo 1! Antes de verificar o seu aprendizado, assista ao vídeo a seguir, em que são apresentados os componentes oferecidos pelo JDBC.



## VERIFICANDO O APRENDIZADO

## MÓDULO 2

---

- ⦿ Descrever o modelo de persistência baseado em mapeamento objeto-relacional

## ORIENTAÇÃO A OBJETOS E O MODELO RELACIONAL

Bases de dados cadastrais seguem um modelo estrutural baseado na álgebra relacional e no cálculo relacional, áreas da matemática voltadas para a manipulação de conjuntos, apresentando ótimos resultados na implementação de consultas. Mesmo apresentando grande eficiência, a representação matricial dos dados, com registros separados em campos e apresentados sequencialmente, não é a mais adequada para um ambiente de programação orientado a objetos.

## ATENÇÃO

Torna-se necessário efetuar uma conversão entre os dois modelos de representação, o que é feito com a criação de uma classe de entidade para expressar cada tabela do banco de dados, à exceção das tabelas de relacionamento. Segundo a conversão efetuada, os atributos da classe serão associados aos campos da tabela, e cada registro poderá ser representado por uma instância da classe.

Para a nossa tabela de exemplo, podemos utilizar a classe apresentada a seguir, na qual, para simplificar o código, não iremos utilizar métodos getters e setters.

```
public class Aluno {  
    public String matricula;  
    public String nome;  
    public int ano;  
  
    public Aluno() { }  
  
    public Aluno(String matricula, String nome, int ano) {  
        this.matricula = matricula;  
        this.nome = nome;  
        this.ano = ano;  
    }  
}
```

Tendo definido a entidade, podemos mudar a forma de lidar com os dados, efetuando a leitura dos dados da tabela e alimentando uma coleção que representará os dados para o restante do programa.

```
List<Aluno> lista = new ArrayList<>();  
Class.forName("org.apache.derby.jdbc.ClientDriver");  
Connection c1 = DriverManager.getConnection(  
    "jdbc:derby://localhost:1527/loja",  
    "loja", "loja");  
Statement st = c1.createStatement();  
ResultSet r1 = st.executeQuery("SELECT * FROM ALUNO");  
while(r1.next())  
    lista.add(new Aluno(r1.getString("MATRICULA"),
```

```
r1.getString("NOME"),  
r1.getInt("ENTRADA"))));  
r1.close();  
st.close();  
c1.close();
```

Como podemos observar, o código apresenta grande similaridade com o trecho para consulta apresentado anteriormente, mas aqui instanciamos uma coleção, e para cada registro obtido, adicionamos um objeto inicializado com seus dados. Tendo concluído o preenchimento da coleção, aqui com o nome **lista**, não precisamos acessar novamente a tabela e podemos lidar com os dados segundo uma perspectiva orientada a objetos.

A conversão de tabelas, e respectivos registros, em coleções de entidades é uma técnica conhecida como **Mapeamento Objeto-Relacional**.

Se quisermos listar o conteúdo da tabela, em outro ponto do código, após alimentar a coleção de entidades, podemos utilizar um código baseado na funcionalidade padrão das coleções do Java.

```
for (Aluno aluno : lista) {  
    System.out.println("Aluno: " + aluno.nome +  
        "(" + aluno.matricula + ") - " +  
        aluno.ano);  
}
```

## DATA ACCESS OBJECT (DAO)

Agora que sabemos lidar com as operações sobre o banco de dados e definimos uma entidade para representar um registro da tabela, seria interessante organizar a forma de programar, pois é fácil imaginar a dificuldade para efetuar manutenção em sistemas com dezenas de milhares de linhas de código Java, contendo diversos comandos SQL espalhados ao longo das linhas.

 **ATENÇÃO**

Baseado na observação anterior, foi desenvolvido um padrão de desenvolvimento com o nome **DAO** (Data Access Object), cujo objetivo é concentrar as instruções SQL em um único tipo de classe, permitindo o agrupamento e a reutilização dos diversos comandos relacionados ao banco de dados.

Normalmente, temos uma classe DAO para cada classe de entidade relevante para o sistema.

Como a codificação das operações sobre o banco apresenta muitos trechos comuns para as diversas entidades do sistema, podemos concentrar as similaridades em uma classe de base, e derivar as demais, segundo o princípio de herança. A utilização de elementos genéricos também será um facilitador na padronização das operações comuns sobre a tabela, como inclusão, exclusão, alteração e consultas.

Vamos observar, no trecho de código seguinte, a definição de uma classe de base, a partir da qual serão derivadas todas as classes DAO do sistema.

```
public abstract class GenericDAO<E,K> {  
    protected Connection getConnection() throws Exception{  
        Class.forName("org.apache.derby.jdbc.ClientDriver");  
        return DriverManager.getConnection(  
            "jdbc:derby://localhost:1527/escola",  
            "escola", "escola");  
    }  
    protected Statement getStatement() throws Exception{  
        return getConnection().createStatement();  
    }  
    protected void closeStatement(Statement st) throws Exception{  
        st.getConnection().close();  
    }  
    public abstract void incluir(E entidade);  
    public abstract void excluir(K chave);  
    public abstract void alterar(E entidade);  
    public abstract E obter(K chave);  
    public abstract List<E> obterTodos();  
}
```

Observe que iniciamos criando os métodos **getStatement** e **closeStatement**, com o objetivo de gerar executores de SQL e eliminá-los, efetuando também as conexões e desconexões nos momentos necessários. Outro método utilitário é o **getConnection**, utilizado apenas para encapsular o processo de conexão com o banco.

Nossa classe **GenericDAO** é abstrata, definindo de forma genérica as assinaturas para os métodos que acessam o banco, onde **E** representa a classe da **entidade** e **K** representa a classe da **chave primária**. Os descendentes de **GenericDAO** deverão implementar os métodos abstratos, preocupando-se apenas com os aspectos gerais do mapeamento objeto-relacional e fazendo ampla utilização dos métodos utilitários.

## COMENTÁRIO

Uma grande vantagem da estratégia adotada é a de que viabilizamos a mudança de fornecedor de banco de dados de forma simples, já que o processo de conexão pode ser encontrado em apenas um método, reutilizado por todo o restante do código.

Se quiser utilizar um banco de dados **Oracle**, com acesso local e instância padrão **XE**, mantendo o usuário e a senha definidos, modifique o corpo do método **getConnection**, conforme sugerido no trecho de código seguinte.

```
Class.forName("oracle.jdbc.OracleDriver");
return DriverManager.getConnection(
jdbc:oracle:thin:@localhost:1521:XE",
"escola","escola");
```

Com a classe de base definida, podemos implementar a classe **AlunoDAO**, concentrando as operações efetuadas sobre nossa tabela, a partir da entidade **Aluno** e chave primária do tipo **String**, sendo o início de sua codificação apresentado a seguir.

```
public class AlunoDAO extends GenericDAO<Aluno, String>{
    @Override
    public List<Aluno> obterTodos() {
        List<Aluno> lista = new ArrayList<>();
        try {
            ResultSet r1 = getStatement().executeQuery(
                "SELECT * FROM ALUNO");
            while(r1.next())
                lista.add(new Aluno(r1.getString("MATRICULA"),
                    r1.getString("NOME"),r1.getInt("ENTRADA")));
            closeStatement(r1.getStatement());
        }catch(Exception e){
        }
    }
}
```



```

        return lista;
    }

    @Override
    public Aluno obter(String chave) {
        Aluno aluno = null;
        try {
            PreparedStatement ps =
                getConnection().prepareStatement(
                    "SELECT * FROM ALUNO WHERE MATRICULA = ?");
            ps.setString(1, chave);
            ResultSet r1 = ps.executeQuery();
            if (r1.next())
                aluno = new Aluno(r1.getString("MATRICULA"),
                                    r1.getString("NOME"),
                                    r1.getInt("ENTRADA"));
            closeStatement(ps);
        } catch (Exception e) {
        }
        return aluno;
    }
}

```

## COMENTÁRIO

O código ainda não está completo, e certamente apresentará erro, devido ao fato de que não implementamos todos os métodos abstratos definidos, mas já temos o método **obterTodos** codificado neste ponto. Ele irá retornar todos os registros de nossa tabela, no formato de um **ArrayList** de entidades do tipo **Aluno**, sendo inicialmente executado o SQL necessário para a consulta e, em seguida, adicionada uma entidade à lista para cada registro obtido no cursor.

Também podemos observar o método **obter**, para consulta individual, retornando uma entidade do tipo **Aluno** para uma chave fornecida do tipo **String**. A implementação do método envolve a execução de uma consulta **parametrizada**, em que o campo matrícula deve coincidir com o valor da chave, sendo retornado o registro equivalente por meio de uma instância de **Aluno**.

Note que, como a consulta foi efetuada a partir da chave, sempre retornará um registro ou nenhum, sendo necessário apenas o comando **if** para mover do **BOF** para o primeiro e único registro. Caso a chave não seja encontrada, a rotina não entrará nessa estrutura condicional e retornará um produto nulo.

Agora que a consulta aos registros foi implementada, devemos acrescentar os métodos de manipulação de dados na classe AlunoDAO.

@Override

```
public void incluir(Aluno entidade) {  
    try {  
        PreparedStatement ps = getConnection().prepareStatement(  
            "INSERT INTO ALUNO VALUES (?, ?, ?)";  
        ps.setString(1, entidade.matricula);  
        ps.setString(2, entidade.nome);  
        ps.setInt(3, entidade.ano);  
        ps.executeUpdate();  
        closeStatement(ps);  
    } catch (Exception e) { }  
}
```

@Override

```
public void excluir(String chave) {  
    try {  
        PreparedStatement ps = getConnection().prepareStatement(  
            "DELETE FROM ALUNO WHERE MATRICULA = ?";  
        ps.setString(1, chave);  
        ps.executeUpdate();  
        closeStatement(ps);  
    } catch (Exception e) { }  
}
```

@Override

```
public void alterar(Aluno entidade) {  
    try {  
        PreparedStatement ps = getConnection().prepareStatement(  
            "UPDATE ALUNO SET NOME = ?, ENTRADA = ? "+  
            " WHERE MATRICULA = ?";  
        ps.setString(1, entidade.nome);  
        ps.setInt(2, entidade.ano);
```

```

        ps.setString(3, entidade.matricula);
        ps.executeUpdate();
        closeStatement(ps);
    } catch (Exception e) { }
}

```

Todos os métodos para manipulação de dados utilizam **PreparedStatement**, obtido a partir de **getConnection**, com o fornecimento da instrução SQL **parametrizada**. As linhas seguintes sempre envolvem o preenchimento de parâmetros e chamada para o método **executeUpdate**, em que o comando SQL resultante, após a substituição das interrogações pelos valores, é efetivamente executado no banco de dados.

O mais simples dos métodos implementados se refere ao **excluir**, por necessitar apenas da chave primária e uso da instrução **DELETE** condicionada à chave fornecida. Os demais métodos seguem a mesma forma de implementação, com a obtenção dos valores para preenchimento dos parâmetros a partir dos atributos da entidade fornecida, sendo o método **incluir** relacionado ao comando **INSERT**, e o método **alterar** representando as instruções SQL do tipo **UPDATE**.

## ATENÇÃO

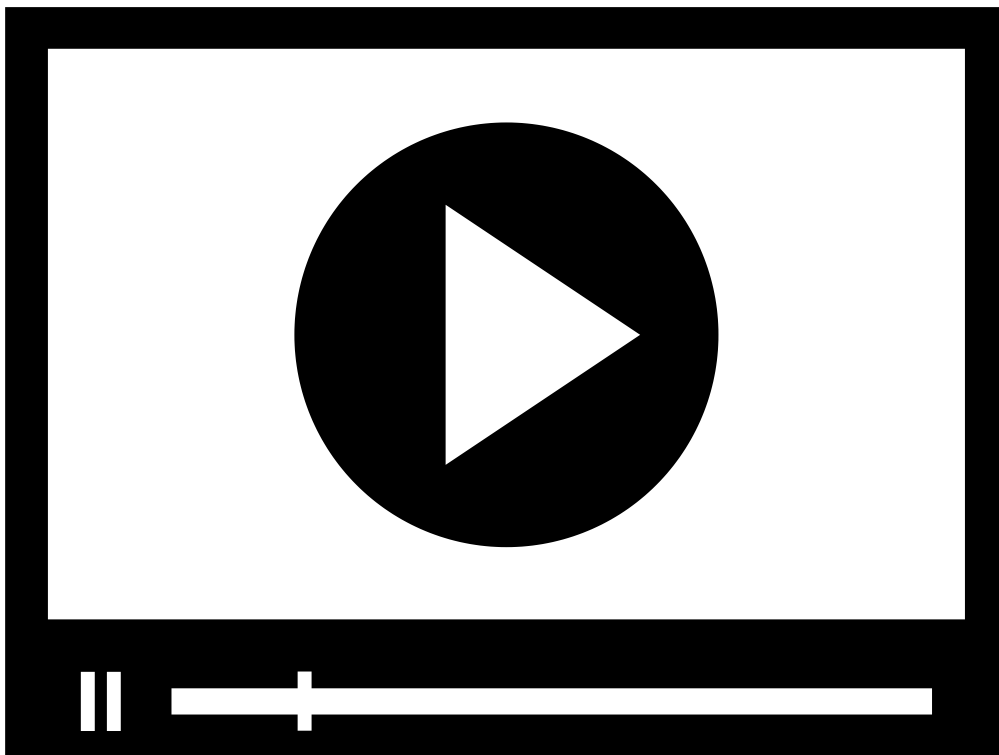
Uma regra para efetuar mapeamento objeto-relacional, e que é seguida por qualquer framework com esse objetivo, é a de que a chave primária da tabela não pode ser alterada. Isso permite manter o referencial dos registros ao longo do tempo.

Após construir a classe DAO, podemos utilizá-la ao longo de todo o sistema, consultando e manipulando os dados sem a necessidade de utilização direta de comandos SQL, como pode ser observado no trecho de exemplo apresentado a seguir, que permitiria imprimir o nome de todos os alunos da base de dados.

```

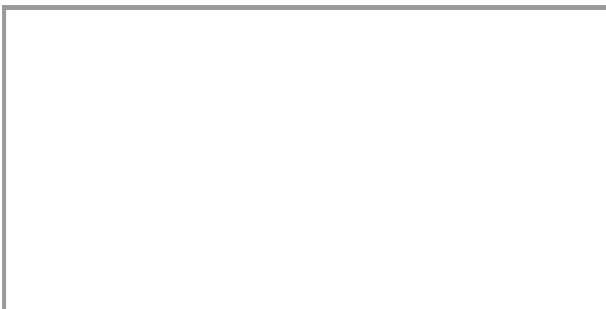
AlunoDAO dao = new AlunoDAO();
dao.obterTodos().forEach(aluno -> {
    System.out.println(aluno.nome);
});

```



## PADRÃO DAO

Assista ao vídeo a seguir e veja um resumo sobre o padrão de desenvolvimento DAO. Além disso, veja como ele pode ser implementado em ambiente Java.



## JAVA PERSISTENCE ARCHITECTURE (JPA)

Devido à padronização oferecida com a utilização de mapeamento objeto-relacional e classes DAO, e considerando a grande similaridade existente nos comandos SQL mais básicos, foi simples chegar à conclusão de que seria possível criar ferramentais para a automatização de diversas tarefas referentes à persistência. Surgiram **frameworks** de persistência, para as mais diversas linguagens, como Hibernate, Entity Framework, Pony e Speedo, entre diversos outros.

Atualmente, no ambiente Java, concentramos os frameworks de persistência sob uma arquitetura própria, conhecida como **Java Persistence Architecture**, ou **JPA**.

O modelo de programação **anotado** é adotado na arquitetura, simplificando muito o mapeamento entre objetos do Java e registros do banco de dados.

Tomando como exemplo nossa tabela de alunos, a entidade Java receberia as anotações observadas no fragmento de código seguinte, para que seja feito o mapeamento com base no JPA.

```
@Entity
@Table(name = "ALUNO")
@NamedQueries({
    @NamedQuery(name = "Aluno.findAll",
        query = "SELECT a FROM Aluno a")})
public class Aluno implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Basic(optional = false)
    @Column(name = "MATRICULA")
    private String matricula;
    @Column(name = "NOME")
    private String nome;
    @Column(name = "ENTRADA")
    private Integer ano;

    public Aluno() {
    }

    public Aluno(String matricula) {
        this.matricula = matricula;
    }

    // getters e setters para os atributos internos
}
```

As anotações utilizadas são bastante intuitivas, como **Entity** transformando a classe em uma entidade, **Table** selecionando a tabela na qual os dados serão escritos, e **Column** associando o atributo a um campo da tabela. As características específicas dos campos podem ser

mapeadas por meio de anotações como **Id**, que determina a chave primária, e **Basic**, na qual o parâmetro **optional** permite definir a obrigatoriedade ou não do campo.

Também é possível definir consultas em sintaxe **JPQL**, uma linguagem de consulta do JPA que retorna objetos, ao invés de registros. As consultas em JPQL podem ser criadas em meio ao código do aplicativo, ou associadas à classe com anotações **NamedQuery**.

Toda a configuração da conexão com banco é efetuada em um arquivo no formato **XML** com o nome **persistence**. No mesmo arquivo deve ser escolhido o driver de persistência.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="" xmlns:xsi=""
xsi:schemaLocation="">
    <persistence-unit name="ExemploJavaDB01PU"
        transaction-type="RESOURCE_LOCAL">
        <provider>
            org.eclipse.persistence.jpa.PersistenceProvider
        </provider>
        <class>modelJPA.Aluno</class>
        <properties>
            <property name="javax.persistence.jdbc.url"
                value="jdbc:derby://localhost:1527/escola"/>
            <property name="javax.persistence.jdbc.user"
                value="escola"/>
            <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="javax.persistence.jdbc.password"
                value="escola"/>
        </properties>
    </persistence-unit>
</persistence>
```

Analisando o arquivo **persistence** do exemplo, temos uma unidade de persistência com o nome **ExemploJavaDB01PU**, sendo a conexão com o banco de dados definida por meio das propriedades **url**, **user**, **driver** e **password**, com valores equivalentes aos que são adotados na utilização padrão do JDBC. Também temos a escolha das entidades que serão incluídas no esquema de persistência, no caso apenas **Aluno**, e do provedor de persistência, em que foi escolhido **Eclipse Link**, mas que poderia ser trocado por **Hibernate** ou **Oracle Top Link**, entre outras opções.

Com os elementos do projeto devidamente configurados, poderíamos utilizá-los para listar o nome dos alunos, por meio do fragmento de código apresentado a seguir.

```
EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("ExemploJavaDB01PU");  
EntityManager em = emf.createEntityManager();  
Query query = em.createNamedQuery("Aluno.findAll", Aluno.class);  
List<Aluno> lista = query.getResultList();  
lista.forEach(aluno->{  
    System.out.println(aluno.getNome());  
});
```

Observe como o uso de JPA diminui muito a necessidade de programação nas tarefas relacionadas ao mapeamento objeto-relacional. Tudo que precisamos fazer é instanciar um **EntityManager** a partir da unidade de persistência, recuperar o objeto **Query** e, a partir dele, efetuar a consulta por meio do método **getResultList**, o qual já retorna uma lista de entidades, sem a necessidade de programar o preenchimento dos atributos.

## VERIFICANDO O APRENDIZADO

### MÓDULO 3

---

- 🕒 **Aplicar tecnologia Java para a viabilização da persistência em banco de dados**

## SISTEMA CADASTRAL SIMPLES

Podemos utilizar a classe **AlunoDAO** na construção de um sistema cadastral simples, em modo texto, com pouquíssimo esforço. Como as operações sobre o banco já estão todas configuradas, nossa preocupação será apenas com a entrada e saída do sistema.

Vamos criar uma classe com o nome **SistemaEscola**, sendo apresentada, a seguir, sua codificação inicial, incluindo a definição da instância de **AlunoDAO** e os métodos para **exibição** de valores, tanto para um aluno quanto para o conjunto completo deles.

```
public class SistemaEscola {  
    private final AlunoDAO dao = new AlunoDAO();  
    private static final BufferedReader entrada =  
        new BufferedReader(  
            new InputStreamReader(System.in));  
  
    private void exibir(Aluno aluno){  
        System.out.println("Aluno: "+aluno.nome+  
            "\nMatricula: "+aluno.matricula+  
            "\tEntrada: "+aluno.ano+  
            "\n=====");  
    }  
  
    public void exibirTodos(){  
        dao.obterTodos().forEach(aluno->exibir(aluno));  
    }  
}
```

## COMENTÁRIO

Note como o método **exibirTodos** utiliza notação lambda para percorrer toda a coleção de alunos, com chamadas sucessivas para o método **exibir**, o qual recebe uma instância de **Aluno** e imprime seus dados no console.

Também temos uma instância estática de **BufferedReader** encapsulando o teclado, com o nome **entrada**, que será utilizada para receber as respostas do usuário nos demais métodos da classe. Vamos verificar sua utilização nos métodos apresentados a seguir, que deverão ser adicionados ao código de **SistemaEscola**.

```
public void inserirAluno() throws IOException{  
    Aluno aluno = new Aluno();  
    System.out.println("Nome:");  
    aluno.nome = entrada.readLine();  
}
```



```

        System.out.println("Matricula:");
        aluno.matricula = entrada.readLine();
        System.out.println("Entrada:");
        aluno.ano = Integer.parseInt(entrada.readLine());
        dao.incluir(aluno);
    }

```

```

public void excluirAluno() throws IOException{
    System.out.println("Matricula:");
    String matricula = entrada.readLine();
    dao.excluir(matricula);
}

```

No método **inserirAluno**, instanciamos um objeto **Aluno** e preenchemos seus atributos com os valores informados pelo usuário, sendo a inclusão na base efetivada ao final, enquanto **excluirAluno** solicita a matrícula e efetua a chamada ao método excluir, para que ocorra a remoção na base de dados.

Para finalizar nosso sistema cadastral, precisamos adicionar à classe um método **main**, conforme o trecho de código apresentado a seguir.

```

public static void main(String args[]) throws IOException{
    SistemaEscola sistema = new SistemaEscola();
    while(true){
        System.out.println(
            "1-Listar\t2-Inserir\t3-Excluir\t0-Sair");
        int opcao = Integer.parseInt(entrada.readLine());
        if(opcao==0)
            break;
        switch(opcao){
            case 1:
                sistema.exibirTodos();
                break;
            case 2:
                sistema.inserirAluno();
                break;
            case 3:
                sistema.excluirAluno();

```

```

        break;
    }
}
}

```

O método **main** permite executar o exemplo, que na prática é bem simples, oferecendo as opções de **listagem**, **inclusão**, **exclusão** e **término**, a partir da digitação do número correto pelo usuário. Feita a escolha da opção, foi necessário apenas ativar o método correto da classe, dentre aqueles que acabamos de codificar.

Com tudo pronto, podemos utilizar as opções **Build** e **Run File** do NetBeans, causando a execução pelo painel **Output**, como pode ser observado a seguir:

```

run:
1-Listar      2-Inserir      3-Excluir      0-Sair
1
Aluno: ANA MARIA
Matricula: E2020.0342   Entrada: 2020
=====
Aluno: JOAO CALDEIRAS
Matricula: M2020.4002   Entrada: 2020
=====

```

📷 Captura de tela do sistema de exemplo em execução.

## GERENCIAMENTO DE TRANSAÇÕES

O **controle transacional** é uma funcionalidade que permite o isolamento de um conjunto de operações, garantindo a consistência na execução do processo completo. De forma geral, uma transação é iniciada, as operações são executadas, e temos ao final uma confirmação do bloco, com o uso de **commit**, ou a reversão das operações, com **rollback**.

### 🔹 SAIBA MAIS

Com o uso de **transações**, temos a garantia de que o banco irá desfazer as operações anteriores à ocorrência de um erro, como na inclusão dos itens de uma nota fiscal. Sem o uso

de uma transação, caso ocorresse um erro na inclusão de algum item, seríamos obrigados a desfazer as inclusões que ocorreram antes do erro, de forma programática, mas com a transação será necessário apenas emitir um comando de rollback.

A inclusão de transações em nosso sistema seria algo bastante simples, pois bastaria efetuar modificações pontuais na classe **AlunoDAO**. Podemos observar um dos métodos modificados a seguir.

@Override

```
public void excluir(String chave) {
    Connection c1 = null;
    try {
        c1 = getConnection();
        c1.setAutoCommit(false);
        PreparedStatement ps = getConnection().prepareStatement(
            "DELETE FROM ALUNO WHERE MATRICULA = ?");
        ps.setString(1, chave);
        ps.executeUpdate();
        c1.commit();
        closeStatement(ps);
    } catch (Exception e) {

        if(c1!=null)
            try {
                c1.rollback();
                c1.close();
            } catch (SQLException e2){}
    }
}
```

Aqui temos a modificação do método **excluir** para utilizar transações, o que exige que a **conexão** seja modificada.

Por padrão, cada alteração é confirmada automaticamente, mas utilizando **setAutoCommit** com valor **false**, a sequência de operações efetuadas deve ser confirmada com o uso de **commit**.

Podemos observar, no código, que após desligar a confirmação automática, temos o mesmo processo utilizado antes para geração e execução do SQL parametrizado, mas com a diferença

do uso de **commit** antes de **closeStatement**. Caso ocorra um erro, será acionado o bloco **catch**, com a chamada para **rollback** e o fechamento da conexão.

A forma de lidar com transações no JPA segue um processo muito similar, como pode ser observado no trecho de código a seguir.

```
public void incluir(Aluno aluno) {  
    EntityManagerFactory emf = Persistence.  
        createEntityManagerFactory("ExemploJavaDB01PU");  
    EntityManager em = emf.createEntityManager();  
    em.getTransaction().begin();  
    try {  
        em.persist(aluno);  
        em.getTransaction().commit();  
    } catch (Exception e) {  
        em.getTransaction().rollback();  
    } finally {  
        em.close();  
    }  
}
```

Como já era esperado, o uso de JPA permite um código muito mais simples, embora os princípios relacionados ao controle transacional sejam similares. O controle deve ser obtido com **getTransaction**, a partir do qual uma transação é iniciada com **begin**, sendo confirmada com o uso de **commit**, ou cancelada com **rollback**.

No fluxo de execução temos a obtenção do **EntityManager**, início da transação, inclusão no banco com o uso de **persist** e confirmação com **commit**. Caso ocorra um erro, temos a reversão das operações da transação como **rollback**, e independentemente do resultado temos a chamada para **close**, dentro de um bloco **finally**.


## SISTEMA COM JPA NO NETBEANS

Diversas ferramentas de produtividade são oferecidas pelo NetBeans, e talvez uma das mais interessantes seja o gerador automático de entidades JPA. Para iniciar o processo, vamos criar projeto comum Java, adotando o nome **ExemploEntidadeJPA**, e seguir estes passos:


Acionar o menu **New File**, escolhendo a opção **Entity Classes from Database**.




Selecionar a conexão JDBC correta (**escola**).



Adicionar as tabelas de interesse, o que no caso seria apenas **Aluno**.

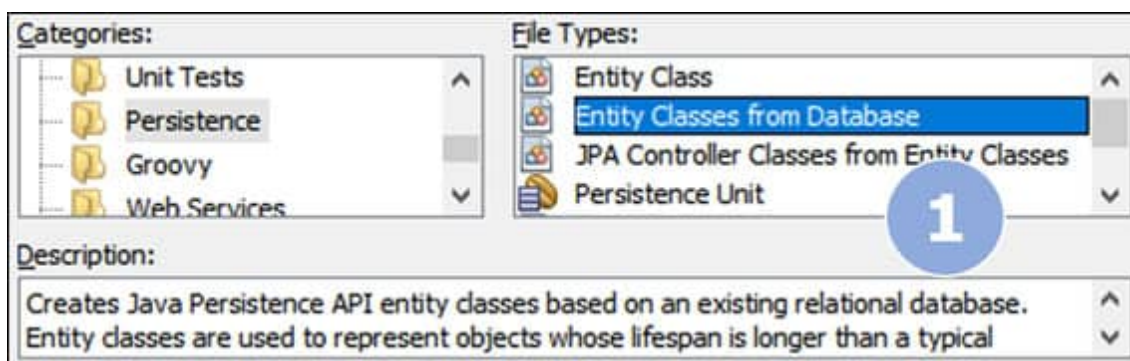


Escrever o nome do pacote (**model**) e deixar marcada apenas a opção de criação para a unidade de persistência.



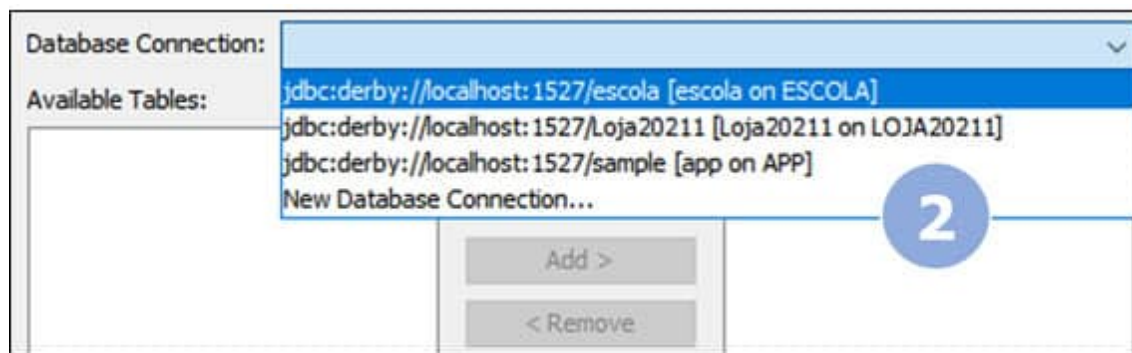
Definir o tipo de coleção como **List**.

Podemos observar os passos descritos na sequência de imagens, capturas de tela do gerador automático de entidades JPA, apresentada a seguir:



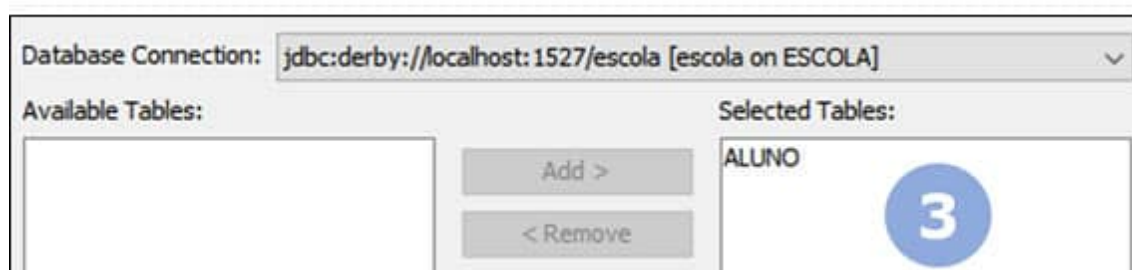
 Capturas de tela do gerador automático de entidades JPA

Acionar o menu **New File**, escolhendo a opção **Entity Classes from Database**.



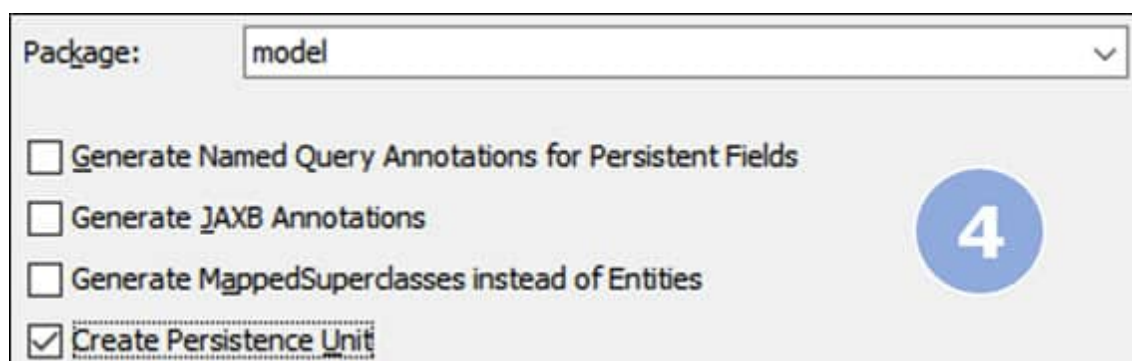
📷 Capturas de tela do gerador automático de entidades JPA

Selecionar a conexão JDBC correta (**escola**).



📷 Capturas de tela do gerador automático de entidades JPA

Adicionar as tabelas de interesse, o que no caso seria apenas **Aluno**.



📷 Capturas de tela do gerador automático de entidades JPA

Escrever o nome do pacote (**model**) e deixar marcada apenas a opção de criação para a unidade de persistência.

Specify the default mapping options.

Association Fetch:  5

Collection Type:

📷 Capturas de tela do gerador automático de entidades JPA

Definir o tipo de coleção como **List**.

Após a conclusão do procedimento, teremos a criação da entidade **Aluno**, no pacote **model**, com a codificação equivalente à que foi apresentada no módulo 2, abordando o JPA, bem como o arquivo **persistence**, em **META-INF**.

Com nossa entidade gerada, podemos gerar um DAO de forma automatizada, por meio dos seguintes passos:

Acionar o menu **New File**, escolhendo **JPA Controller Classes for Entity Classes**.



Selecionar as entidades, o que no caso seria apenas **Aluno**.



Escrever o nome do pacote (**manager**).

Novamente, é possível observar os passos descritos por meio de uma sequência de imagens capturadas da tela do gerador automático de entidades JPA:

Filter:

Categories:

- Unit Tests
- Persistence
- Groovy
- Web Services

File Types:

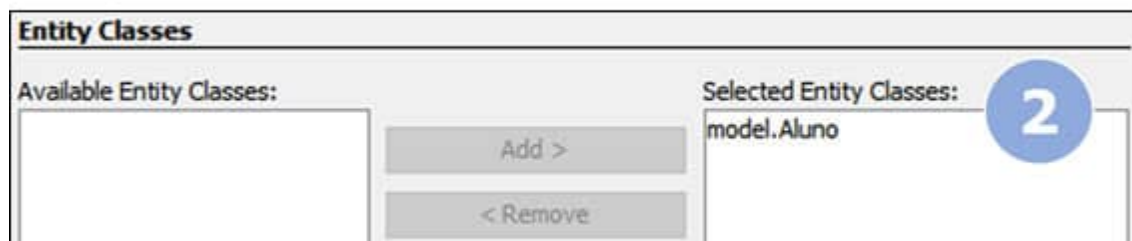
- Entity Class
- Entity Classes from Database
- JPA Controller Classes from Entity Classes** 1
- Persistence Unit

Description:

Creates a set of JPA controller classes and related classes from a set of entity classes.

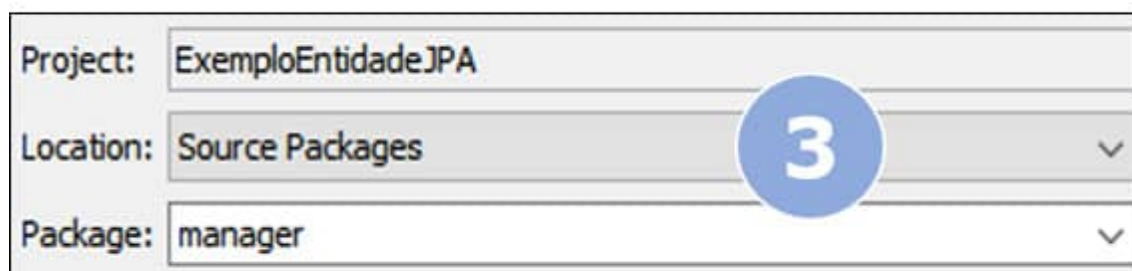
📷 Capturadas da tela do gerador automático de entidades JPA

Acionar o menu **New File**, escolhendo **JPA Controller Classes for Entity Classes**.



📷 Capturadas da tela do gerador automático de entidades JPA

Selecionar as entidades, o que no caso seria apenas **Aluno**.



📷 Capturadas da tela do gerador automático de entidades JPA

Escrever o nome do pacote (**manager**).

Ao término do processo, teremos um pacote com exceções customizadas, e outro com a classe DAO, com o nome **AlunoJpaController**, onde temos todas as operações básicas sobre a tabela Aluno, utilizando tecnologia JPA.

Todas as operações para manipulação de dados são feitas a partir do EntityManager, em que os métodos adequados podem ser observados no quadro apresentado a seguir:

SQL	DAO	JpaController	EntityManager
INSERT	Incluir	create	<b>persist</b>



<b>UPDATE</b>	Alterar	edit	<b>merge</b>
<b>DELETE</b>	Excluir	destroy	<b>remove</b>

**Atenção!** Para visualização completa da tabela utilize a rolagem horizontal

Quadro: Correlação de métodos para diferentes componentes.

Elaborado por: Denis Gonçalves Cople.

A análise do código completo de **AlunoJpaController** não é necessária, já que apresenta diversas partes que se repetem, mas o método de remoção é replicado aqui:

```
public void destroy(String id) throws NonexistentEntityException{
    EntityManager em = null;
    try {
        em = getEntityManager();
        em.getTransaction().begin();
        Aluno aluno;
        try {
            aluno = em.getReference(Aluno.class, id);
            aluno.getMatricula();
        } catch (EntityNotFoundException enfe) {
            throw new NonexistentEntityException("....", enfe);
        }
        em.remove(aluno);
        em.getTransaction().commit();
    } finally {
        if (em != null)
            em.close();
    }
}
```

Inicialmente, é obtida uma instância de **EntityManager**, permitindo efetuar as operações que se seguem. A transação é iniciada, uma referência para a entidade a ser removida é obtida com **getReference** e testada em seguida com **getMatricula**, ocorrendo erro para uma referência nula, ou sendo efetivada a exclusão com **remove**, no caso contrário.

O método **getEntityManager** retorna uma instância do gerenciador de entidades, com base no método **createEntityManager** da fábrica, do tipo **EntityManagerFactory**, que deve ser fornecida por meio do **construtor**.

Após utilizar os recursos de automatização do NetBeans, podemos criar uma classe Java com o nome **SistemaEscola**, muito similar à que foi gerada anteriormente, mas com as modificações necessárias para uso do JPA.

```
public class SistemaEscola {  
    private final AlunoJpaController dao = new AlunoJpaController(  
        Persistence.createEntityManagerFactory(  
            "ExemploEntidadeJPAPU"));  
    private static final BufferedReader entrada =  
        new BufferedReader(new InputStreamReader(System.in));  
  
    private void exibir(Aluno aluno){  
        System.out.println("Aluno: "+aluno.getNome()+  
            "\nMatricula: "+aluno.getMatricula()+  
            "\tEntrada: "+aluno.getEntrada()+  
            "\n=====");  
    }  
    public void exibirTodos(){  
        dao.findAlunoEntities().forEach(aluno->exibir(aluno));  
    }  
    public void inserirAluno() throws Exception{  
        Aluno aluno = new Aluno();  
        System.out.println("Nome:");  
        aluno.setNome(entrada.readLine());  
        System.out.println("Matricula:");  
        aluno.setMatricula(entrada.readLine());  
        System.out.println("Entrada:");  
        aluno.setEntrada(Integer.parseInt(entrada.readLine()));  
        dao.create(aluno);  
    }  
    public void excluirAluno() throws Exception{  
        System.out.println("Matricula:");  
        String matricula = entrada.readLine();  
        dao.destroy(matricula);  
    }  
}
```

```

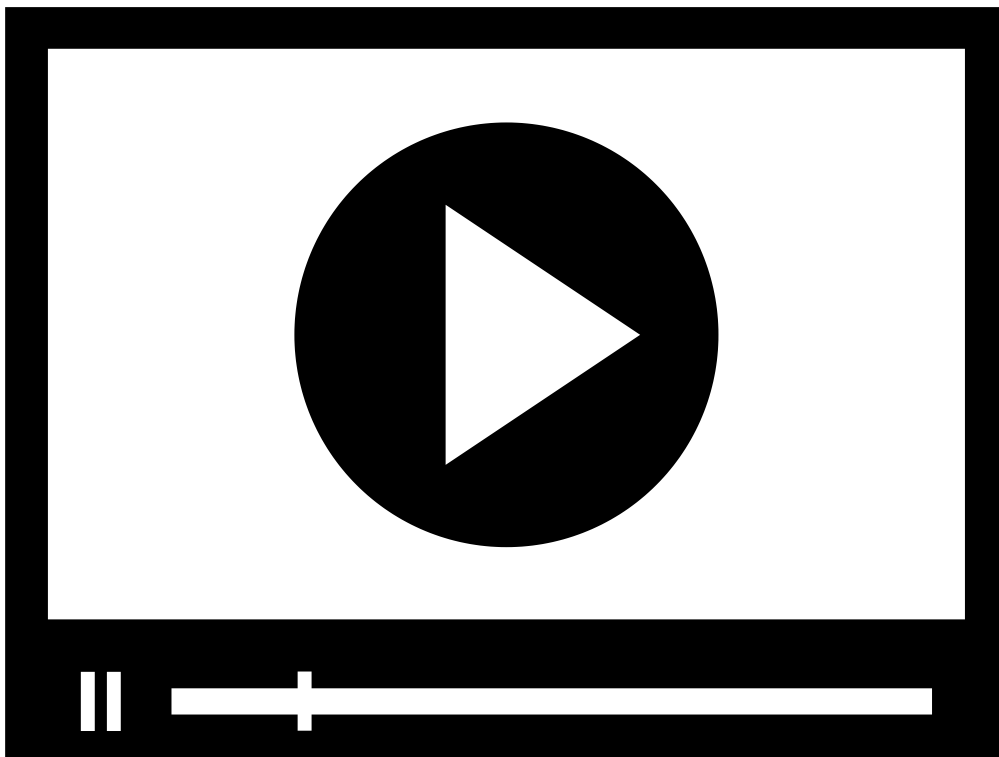
    }
    public static void main(String args[]) throws Exception{
        SistemaEscola sistema = new SistemaEscola();
        while(true){
            System.out.println(
                "1-Listar\t2-Inserir\t3-Excluir\t0-Sair");
            int opcao = Integer.parseInt(entrada.readLine());
            if(opcao==0)
                break;
            switch(opcao){
                case 1: sistema.exibirTodos(); break;
                case 2: sistema.inserirAluno(); break;
                case 3: sistema.excluirAluno(); break;
            }
        }
    }
}

```

As modificações que devem ser efetuadas incluem a utilização de **AlunoJpaController**, inicializado a partir da unidade de persistência **ExemploEntidadeJPAPU**, no lugar de **AlunoDAO**, além de **getters** e **setters** no acesso aos atributos de **Aluno**. Também temos modificações nos nomes dos métodos utilizados para efetuar consultas e modificações no banco, devido à nomenclatura própria do JPA, sendo adotados **findAlunoEntities** para obter o conjunto de registros, **create** para a inclusão na base de dados e **destroy** para executar a remoção.

## ATENÇÃO

Enquanto as bibliotecas JPA são adicionadas de forma automática no projeto, teremos de adicionar manualmente a biblioteca **Java DB Driver**, conforme processo já descrito. Após adicionar a biblioteca, podemos executar o projeto, obtendo o mesmo comportamento do exemplo criado anteriormente, com uso de DAO.



## AUTOMATIZAÇÕES DO NETBEANS

Estamos concluindo o módulo 3! Agora, assista ao vídeo a seguir para obter uma descrição das funcionalidades do NetBeans para geração de entidades a partir do banco de dados, bem como controladores JPA a partir das entidades, e demonstração do uso do ferramental na construção rápida de aplicativos cadastrais.



## VERIFICANDO O APRENDIZADO

# CONCLUSÃO

## CONSIDERAÇÕES FINAIS

Analizamos neste conteúdo duas tecnologias para acesso e manipulação de dados no ambiente Java, que são JDBC e JPA, em que o primeiro é o middleware de acesso a bancos de dados do Java, e o segundo uma arquitetura de persistência baseada em anotações.

Vimos que é possível trabalhar apenas com JDBC, utilizando os componentes ao longo do código, mas a propagação de comandos SQL ao longo dos códigos nos levou à adoção do padrão DAO, organizando nossos códigos e trazendo o nível de padronização exigido para o surgimento de ferramentas como o JPA.

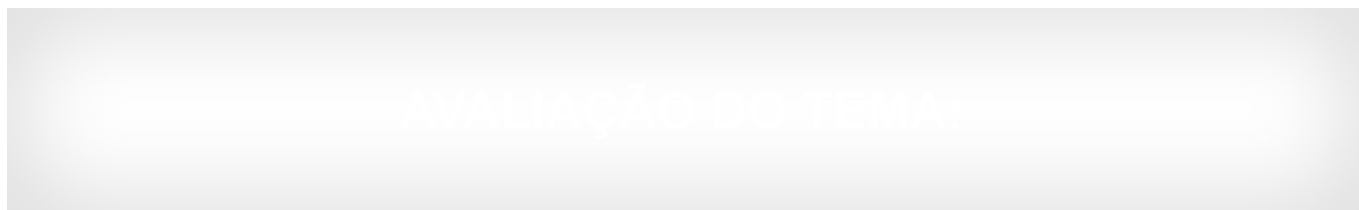
Finalmente, utilizamos os conhecimentos adquiridos na construção de um sistema cadastral simples, criado com programação pura inicialmente, mas que foi refeito com o ferramental de geração do NetBeans, demonstrando como obter maior produtividade.



PODCAST

## PODCAST

Para finalizar nosso conteúdo, ouça o podcast a seguir, que apresenta uma visão geral sobre integração com banco de dados em Java.



# REFERÊNCIAS

CORNELL, G.; HORSTMANN, C. **Core Java**. 8. ed. São Paulo: Pearson, 2010.

DEITEL, P.; DEITEL, H. **Ajax, Rich Internet Applications e Desenvolvimento Web para Programadores**. São Paulo: Pearson Education, 2009.

DEITEL, P.; DEITEL, H. **Java, Como Programar**. 8. ed. São Paulo: Pearson, 2010.

MONSON-HAEFEL, R.; BURKE, B. **Enterprise Java Beans 3.0.5**. USA: O'Reilly, 2006.

---

## EXPLORE+

Para saber mais sobre os assuntos tratados neste conteúdo, leia:

Guia da Oracle sobre uso de JDBC com SWING;

Guia da Oracle sobre transações no JDBC;

Transações e concorrência no Hibernate;

Tutorial de JPA com NetBeans.

---

## CONTEUDISTA

Denis Gonçalves Cople

 **CURRÍCULO LATTES**