



# Desenvolvimento de Sistemas

---

## Manipulação de dados: recursos de linguagem para manipulação de banco de dados, inserção, atualização, consulta e remoção de registros

### Introdução

Em conteúdos anteriores foi estudada a forma correta de se conectar a um banco de dados utilizando a linguagem de programação Java. Você utilizará esses conhecimentos aprendidos sempre que precisar fazer alguma modificação em sua base de dados.

Lembre-se de que, após a criação do banco e das tabelas, sempre que for necessária uma modificação/atualização em seu conteúdo, será preciso fazer antes a conexão com o banco. Além disso, por segurança, ao final de cada modificação concluída, você deve fazer o fechamento desse banco para diminuir as chances de ataques externos aos dados.

Depois de estabelecer essa conexão, há a possibilidade de manipular a base de dados por meio de comandos de inserção, atualização, consulta e inserção de registros, seja de modo textual (linhas de comando), seja de modo visual, utilizando as interfaces gráficas já aprendidas com o Java Swing.

O conhecimento da manipulação de dados permite ao programador uma robustez muito maior em seus sistemas, viabilizando a criação de sistemas dinâmicos, que mantenham o armazenamento, mesmo enquanto fora de operação, com maior segurança, centralização e agilidade na recuperação das operações.

# Recursos de linguagem para manipulação de banco de dados

Na área de desenvolvimento, o acesso à base de dados acontece na maioria dos sistemas, existindo poucas exceções em sistemas de menor complexidade. O Java, diferentemente de outras linguagens, não contém recursos nativos de acesso direto a um banco de dados, e, quando ele precisa desse recurso, utiliza uma API (conjunto de classes e interfaces) chamada JDBC (Java Database Connectivity). Essa API permite o envio dos comandos SQL para qualquer banco relacional, entre eles o MySQL, desde que haja um *driver* previamente instalado, como já estudado anteriormente na parte de conexão a banco de dados com Java.

Serão criados neste material exemplos reais, utilizando tanto a forma de programação textual (somente por linha de código) quanto a visual (com uso de interfaces gráficas) para demonstrar de forma completa os processos de inserção, atualização, consulta e remoção de dados. Além disso, com o auxílio do NetBeans 13, serão criados códigos e o resultado será visualizado por meio do MySQL Workbench.

## Inserção

A primeira etapa da manipulação de dados a ser trabalhada é a inserção de dados. Tenha sempre em mente que, para a realização dessa importante etapa, é preciso que você já tenha criado no MySQL o banco de dados e as tabelas nos quais pretende inserir os dados.

Comece criando um banco de dados chamado **escola** e utilize tabelas simplificadas, sem muitos campos, para facilitar. O mais importante neste momento é a inserção correta dos dados.



Figura 1 – Criação do banco de dados **escola** no Workbench

Fonte: MySQL Workbench (2022)

Abaixo, somente desta vez, coloque separada, para facilitar o entendimento, a mensagem que deve aparecer para indicar que o banco foi criado com sucesso.

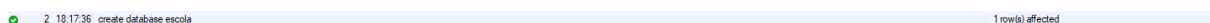


Figura 2 – Mensagem de sucesso após criação da base de dados

Fonte: Adaptado de MySQL Workbench (2022)

Depois de criar a base de dados, crie uma tabela chamada alunos, na qual haverá os campos “id”, “nome”, “idade” e “endereço”. Sabe-se que uma tabela pode ter muitos mais campos, mas com esses quatro campos já se obtém todo o entendimento do que é necessário para inserção de dados sem precisar construir um código tão extenso. Para criar essa tabela, utilize no MySQL Workbench o código a seguir:

```
use escola;
create table alunos(
    id int NOT NULL AUTO_INCREMENT,
    nome varchar(70) NOT NULL,
    idade int,
    endereco varchar(100) NOT NULL,
    PRIMARY KEY (id)
);
```

Após a utilização desse código, é possível notar que a tabela foi criada por meio da imagem a seguir:



Figura 3 – Tabela “alunos”

Fonte: MySQL Workbench (2022)

Pronto. Agora que a estrutura já está preparada, é possível criar o código que inserirá dados dentro da tabela “alunos”. Com variações desse código, pode-se criar qualquer tipo de inserção em tabelas futuras.

Não se esqueça de que, antes desse comando, é preciso carregar o *driver* JDBC e estabelecer a conexão com o banco de dados. Você verá um exemplo de código simplificado, mas poderá utilizar o código já aprendido nos conhecimentos anteriores.

Para iniciar seus testes, crie um projeto Java chamado “Conexao2” e inclua como dependência a biblioteca MySQL Connector J. Na classe principal, inclua o seguinte código em **main()**.

```
package conexao2;

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

public class Conexao2 {

    public static void main(String[] args) {
        try {

            Class.forName( "com.mysql.cj.jdbc.Driver" );
            System.out.println( "Driver JDBC carregado" );
        } catch ( ClassNotFoundException cnfe ) {
            System.out.println( "Driver JDBC nao encontrado : " +
                                cnfe.getMessage() );
        }

        Connection con = null;
        try {

            con = DriverManager.getConnection("jdbc:mysql://localhost:3306/escola", "root", "");
            System.out.println( "Conexao com o banco de dados estabelecida." );
        } catch ( SQLException sqle ) {
            System.out.println( "Erro na conexao ao Bando de Dados : " +
                                sqle.getMessage() );
        }
    }
}
```

Com esse código, comece carregando o *driver* MySQL "**com.mysql.cj.jdbc.Driver**". Essa etapa permite estabelecer a conexão com o banco de dados "alunos".

Para começar a preparação da inserção de dados, é preciso importar uma nova biblioteca, a qual permitirá a utilização dos códigos da classe **Statement**, que será explicada a seguir.

```
import java.sql.Statement;
```

## Inserção com **Statement**

**Statement** é uma interface utilizada para executar instruções SQL. Nela existem vários métodos que podem ser utilizados para inserir, atualizar, deletar e até consultar informações do banco de dados. Em resumo, o **Statement** é o responsável pela execução do SQL no seu banco.

## Criando o objeto para inserir os comandos SQL

Ao final do código de **main**, depois de definido e preenchido o objeto **con**, inclua o seguinte código (é necessário incluir `import java.sql.Statement;`).

```
Statement stmt = null;
try {
    stmt = con.createStatement();
    System.out.println( "Pronto para execucao de comandos sql." );
} catch ( SQLException sqle ) {
    System.out.println( "Erro no acesso ao Bando de Dados : " +
                        sqle.getMessage() );
}
```

Antes de qualquer etapa de inserção, é preciso criar o objeto do tipo **Statement**, que será o responsável pela execução dos códigos SQL. No código citado criou-se um objeto chamado **stmt**, que sempre deve ser iniciado como **null**.

O objeto gerado a partir de uma classe **Statement** sempre tem que estar ligado a uma conexão. Assim, utiliza-se a linha:

```
stmt = con.createStatement()
```

Nessa linha, vincula-se o objeto **stmt** ao objeto de conexão **con**, criado nessa conexão com o banco de dados, e utiliza-se o comando **createStatement()**, que cria um objeto que permite o envio de instruções SQL ao banco de dados.

Note que, em todas etapas de manipulação de banco de dados, o Java considera algum risco e é obrigatório o uso do comando *try-catch*, para o que sistema tenha já alguma resposta no caso de ocorrer algum erro.

Com o objeto do tipo **Statement** iniciado, é possível começar a etapa de inserção ou qualquer outro tipo de manipulação de dados.

Note que, neste momento, no MySQL Workbench, a tabela “alunos” do banco de dados “escola” está vazia. Observe a imagem a seguir:

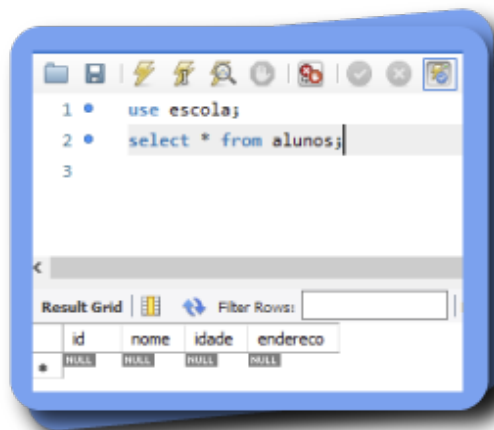


Figura 4 – Tabela “alunos” vazia antes da inserção de dados  
Fonte: MySQL Workbench (2022)

## Inserindo os valores na tabela com Statement

O código seguinte deve ser incluído em **main()** após a definição do **Statement stmt** implementada no trecho anterior.

```
try{
    String sql = null;

    sql = "insert into alunos (nome, idade, endereco) values ('Ana','23','Rua 7 de setembro 826')";
    stmt.executeUpdate(sql);
    System.out.println("Dados inseridos.");
}
catch (SQLException sqle ){
    System.out.println( "Erro inserindo : " + sqle.getMessage() );
}
```

Primeiramente, deve-se criar uma variável *string* que será a responsável por armazenar o comando de inserção dos dados. Somente após esse comando (insert into alunos values ('Ana','23','Rua 7 de setembro 826')) ser colocado dentro da variável é que será possível, com a variável **stmt**, executar o comando **executeUpdate(sql)**, que enviará ao banco e executará o *script* SQL definido.

Isso significa que o objeto **stmt** permite o uso do comando **executeUpdate()**, que recebe por parâmetro uma *string* e a envia ao banco de dados. Essa *string* é o *script* SQL que seria executado no banco diretamente se você estivesse usando o MySQL Workbench, por exemplo.

Se o código estiver correto, você receberá a mensagem “dados inseridos”, ainda dentro do comando **try**. Caso qualquer erro aconteça, como medida de segurança obrigatória do Java, ele apresentará o erro “Erro inserindo” + mensagem de erro padrão do **SQLException**, que é o tratamento de exceção do SQL.

Depois da inserção de dados, nota-se a tabela já com uma linha de dados inserida, como na imagem a seguir:



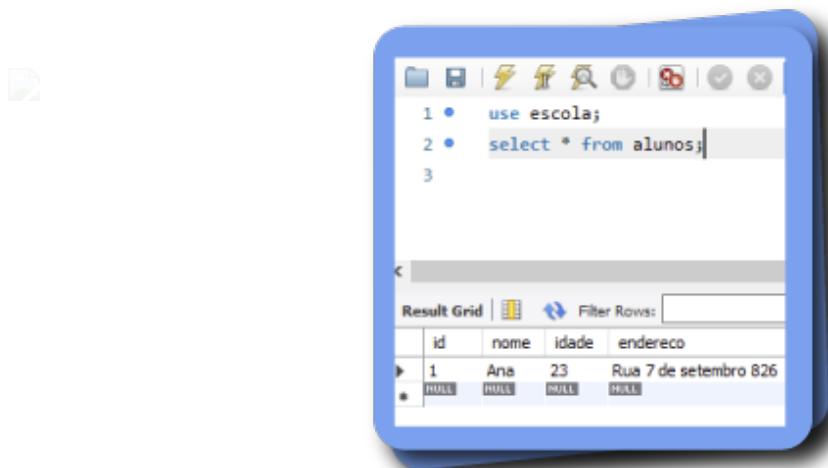


Figura 5 – Tabela “alunos” com dados já inseridos

Fonte: MySQL Workbench

## Deletando uma linha da tabela com Statement

O código seguinte deve ser incluído em **main()**, após a definição de **Statement stmt** implementada anteriormente, e substitui o código anterior em que foi realizada a inclusão.

```
try{  
    String sql = null;  
    sql = "delete from alunos where id =1";  
    stmt.executeUpdate(sql);  
    System.out.println("Dados removidos.");  
}  
catch (SQLException sqle ){  
    System.out.println( "Erro remocao : " + sqle.getMessage() );  
}
```

Note que a exclusão de dados com o **Statement** segue a mesma linha de raciocínio e execução que a inserção de dados. Dessa forma, primeiro você cria uma variável **sql** como nula e logo armazena dentro dela a *string* com o código que era

para excluir a linha, nesse caso utilizando como parâmetro o **id** do aluno por ser a chave primária e não ter a possibilidade de existirem **ids** repetidos.

Quando o código estiver dentro da variável **sql**, basta executar o comando **executeUpdate(sql)**, utilizando o objeto **stmt**, responsável por permitir a utilização dos comandos **sql**.

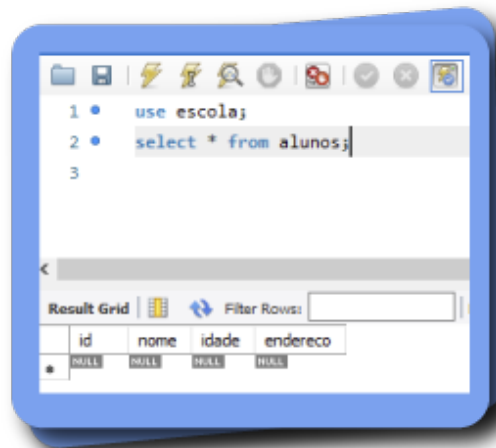


Figura 6 – Tabela “alunos” vazia após a deleção da linha 1

Fonte: MySQL Workbench (2022)

## Atualizando dados da tabela com Statement

O código seguinte deve ser incluído em **main()** após a definição de **Statement stmt** implementada anteriormente e substitui a operação de exclusão também implementada anteriormente.

```
try{
    String sql = null;
    sql = "update alunos set idade = '58' where id =1";
    stmt.executeUpdate(sql);
    System.out.println("Dados atualizados.");
}
catch (SQLException sqle ){
    System.out.println( "Erro atualizacao : " +
        sqle.getMessage() );
}
```

Para atualizar os dados de uma tabela específica, a lógica é a mesma já apresentada na inserção e na exclusão de dados.

Pode-se observar a mesma lógica apresentada anteriormente e também o uso da variável **sql** e do objeto **stmt** com o comando **executeUpdate(sql)**. A linha de código **“update alunos set idade=’28’ Where id=1”** mudará a idade do aluno com “id = 1” para “28”, independentemente de qual idade estiver armazenada anteriormente.

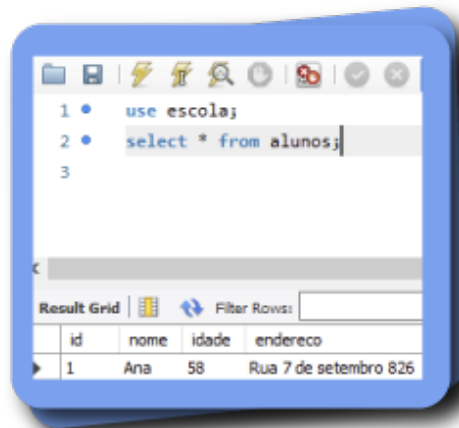


Figura 7 – Tabela “alunos” com a idade alterada para 58

Fonte: MySQL Workbench (2022)

## Consultando dados

Para começar as consultas, é preciso antes de tudo importar a biblioteca apresentada a seguir:

```
import java.sql.ResultSet;
```

O seguinte código deve ser incluído em **main()** após a definição de **Statement stmt** implementada anteriormente e substitui a operação de atualização também implementada anteriormente. É necessário incluir `import java.sql.ResultSet;`.

```
try {
    String sql = "select id,nome,idade,endereco from alunos" ;

    /* Executando o comando select */
    ResultSet rs = stmt.executeQuery(sql);

    /* Exibindo os resultados */
    while (rs.next()) {
        int id = rs.getInt("id");
        String nome = rs.getString("nome");
        int idade = rs.getInt("idade");
        String endereco = rs.getString("endereco");

        System.out.println("dados da tabela autor do banco de dados");
        System.out.println("-----");
        System.out.println(id + " - " + nome + " - " + idade + " - " + endereco);
    }
} catch (SQLException sqle) {
    System.out.println( "Erro efetuando consulta : " + sqle.getMessage() );
}

try {
    con.close();
    System.out.println( "Conexão com o banco de dados fechada" );
} catch (SQLException sqle) {
    System.out.println( "Erro no fechamento da conexão : " + sqle.getMessage() );
}
```

Para consultar os dados de uma tabela específica, a lógica é a mesma já apresentada na inserção, na atualização e na exclusão de dados. Porém, agora será criado um objeto chamado **rs** do tipo **ResultSet**, que é responsável pelos códigos de consulta no Java. Além disso, agora se utiliza o comando **executeQuery(sql)** para executar códigos de consulta no banco de dados por meio do Java.

O diferencial maior do código de consulta está no *looping* que é necessário fazer para percorrer todos os dados dentro de uma tabela. Para isso, utiliza-se um comando **while(rs.next())**, que percorre todos os campos de uma tabela do banco de dados. Para mostrar os dados, é preciso salvar os resultados dentro de variáveis, conforme o tipo de cada campo da tabela, como no código a seguir:

```
int id = rs.getInt("id");
```

Nesse código, uma variável **id** foi criada e receberá o valor armazenado dentro do campo “id” da tabela “alunos”. Esse código é obtido pelo comando **rs.getInt**, que utiliza a variável **rs** do **ResultSet**, a qual, juntamente ao comando **getInt**, obtém um valor do tipo inteiro.

Note que, no segundo campo, já se utilizou um **getString** que busca um campo do tipo texto. Depois, basta apresentar os dados obtidos nas variáveis com o comando **system.out.println**.

É sempre importante, ao final de cada etapa de utilização, o fechamento da conexão por motivos de segurança. Para isso, use o comando **com.close()**;

Depois da consulta, o resultado apresentado, de forma completa no NetBeans, será o seguinte:

```
run:
Driver JDBC carregado
Conexão com o banco de dados estabelecida.
Pronto para execução de comandos sql.
Dados atualizados.
dados da tabela autor do banco de dados
-----
1 - Ana - 58 - Rua 7 de setembro 826
Conexão com o banco de dados fechada
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Usando dados variáveis na construção de consultas SQL

Antes de começar o **Prepared Statement**, aprenda como funcionaria uma manipulação de dados que fossem inseridos pelo usuário analisando dois códigos: um de atualização e outro de deleção de dados.

Para permitir a inserção de valores, você utilizará a classe **Scanner** e precisará importar a biblioteca a seguir:

```
import java.util.Scanner;
```

Depois de importar a biblioteca, você terá este código para parte de atualização de dados:

```
int idBusca,idadeNova;  
Scanner IdFunc = new Scanner(System.in);  
System.out.println("Informe o id que será atualizado");  
idBusca = IdFunc.nextInt();  
  
System.out.println("Informe a nova idade");  
idadeNova = IdFunc.nextInt();  
  
try{  
    String sql = null;  
    sql = "update alunos set idade = "+ idadeNova +" where id =" + idBusca;  
    stmt.executeUpdate(sql);  
    System.out.println("Dados atualizados.");  
}  
catch (SQLException sqle ){  
    System.out.println( "Erro atualizacao : " + sqle.getMessage() );  
}
```

Note que agora os dados não são definidos antes, mas serão inseridos pelo usuário. Para isso, foram utilizadas duas variáveis inteiras, **idBusca** e **idadeNova**, que receberão os dados vindos da classe **Scanner**.

Para receber esses dados, utiliza-se o comando **nextInt()** e, com os dados armazenados dentro da classe, é possível concatená-los à variável **sql** para fazer a atualização funcionar por meio da seguinte linha de código:

```
sql = "update alunos set idade = "+ idadeNova +" where id =" + idBusca;
```

Essa é a forma de permitir ao usuário saber qual será a nova idade do funcionário e em qual **id** ocorrerá essa modificação. Pode-se utilizar a mesma lógica em um processo de exclusão de dados, como você pode ver no código a seguir:

```
int idBusca;
Scanner IdFunc = new Scanner(System.in);

System.out.println("Informe o id da linha a ser excluída");
idBusca = IdFunc.nextInt();

try{
    String sql = null;
    sql = "delete from alunos where id =" + idBusca;
    stmt.executeUpdate(sql);
    System.out.println("Dados removidos.");
}
catch (SQLException sqle ){
    System.out.println( "Erro remocao : " + sqle.getMessage() );
}
```

Note aqui que a classe **Scanner** pode ser utilizada em qualquer um dos processos de manipulação de dados, e isso aumenta muito a interação com o usuário final do sistema.

## Inserindo dados com Prepared Statement

Para começar o uso do **Prepared Statement**, é preciso antes de tudo importar a seguinte biblioteca:

```
import java.sql.PreparedStatement;
```

A manipulação de dados com **Prepared Statement** é mais rápida e segura, por isso é sempre indicado utilizá-la. Ainda assim, você deve aprender das duas formas, pois, quando for necessário modificar/atualizar sistemas criados por outras pessoas, você poderá encontrar qualquer uma delas dentro do Java. Além disso, o aprendizado das duas formas se complementa.



Uma das grandes vantagens desse meio de manipulação é que ele trata separadamente os dados inseridos no banco. Isso impede um dos ataques mais conhecidos a banco de dados, que é o SQL Injection, no qual códigos **sql** são colocados nos campos de inserção de valores e podem manipular seu banco de dados. Esses códigos podem tanto excluir dados importantes quanto modificar consultas confidenciais ou alterar dados, o que fere o princípio da confiabilidade e da integridade dos dados.

Para saber mais informações sobre **Prepared Statement** e questões de segurança, confira o conteúdo **Segurança da Informação** desta unidade curricular.

O código a seguir pode ser aplicado em **main()** após a definição de **Statement**.

```
/* Objeto para executar comandos sql */
PreparedStatement ps = null;
String sql = "insert into alunos values (?, ?, ?, ?)";

try {
    ps = con.prepareStatement(sql);
    ps.setInt(1, 2);
    ps.setString(2, "Érico");
    ps.setInt(3, 30);
    ps.setString(4, "Rua das araucárias 20");

    ps.executeUpdate();

    System.out.println( " Dados inseridos com sucesso." );
} catch ( SQLException sqle ) {
    System.out.println( "Erro no acesso ao Bando de Dados : "+ sqle.getMessage());
}

/* Fechando a conexão */
try {
    con.close();
    System.out.println( "Conexão com o banco de dados fechada" );
} catch ( SQLException sqle ) {
    System.out.println( "Erro no fechamento da conexão : " + sqle.getMessage());
}
```

Analisando esse código, note que há algumas semelhanças entre ele e o código aprendido anteriormente, pois também foi criada uma primeira variável iniciada como nula, mas agora ela é do tipo **Prepared Statement**.

Também foi criada uma variável **sql** do tipo *string*, que recebe o código SQL que será utilizado na inserção de dados. Porém, agora, no lugar do valor que irá ao banco será inserido o sinal “?”, que indica que ali haverá um valor. Mas, antes de inserido, esse sinal precisa ser tratado. É ele que impede o uso do SQL Injection, explicado anteriormente. O total de símbolos “?” indica o número de dados que será inserido: como nesse exemplo há quatro símbolos “?”, isso significa que quatro dados serão inseridos na tabela.

Sabendo que são quatro valores, deve-se mostrar ao programa quais são esses valores. É preciso ligar a variável **ps** a uma conexão já existente, que, neste programa, é a variável **con**, por meio deste comando:

```
ps = con.prepareStatement(sql);
```

Nesse momento, serão tratados os dados a serem recebidos com o comando **set()**, no qual é preciso indicar qual é o tipo de dado e, dentro dos atributos, nos parênteses, quais são a ordem e o valor inseridos, assim como mostram os exemplos a seguir:

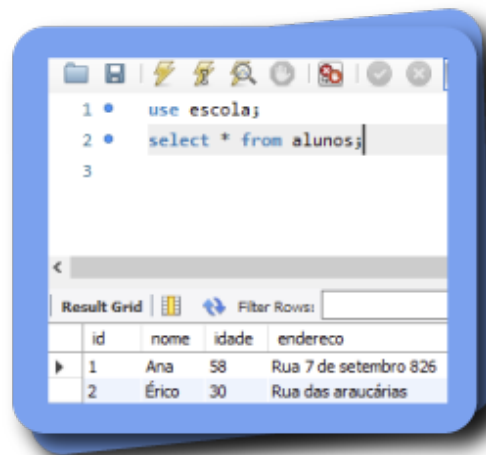
```
ps.setInt(1,2);
```

Nesse exemplo é recebido um dado do tipo inteiro, por isso foi usado o **setInt**. Ele é o primeiro valor inserido, então tem o número 1, e o conteúdo dele é o valor inteiro 2.

```
ps.setString(2, "Érico");
```

Nesse exemplo, o valor é do tipo texto, *string*. Ele é o segundo valor inserido, por isso o número 2, e o conteúdo dele é “Érico”.

Depois, basta utilizar o comando **ps.executeUpdate()** para finalizar a inserção dos dados. Depois de o código ser executado, a tabela ficará desta maneira:



The screenshot shows the MySQL Workbench interface. The SQL editor contains the following code:

```
1 • use escola;  
2 • select * from alunos;  
3
```

Below the editor, the 'Result Grid' tab is active, displaying the following data:

	id	nome	idade	endereço
▶	1	Ana	58	Rua 7 de setembro 826
	2	Érico	30	Rua das araucárias

Figura 8 – Tabela com os dados inseridos com **Prepared Statement**

Fonte: MySQL Workbench (2022)

## Deletando dados com Prepared Statement

A parte de exclusão de dados segue lógica semelhante à da inserção. Para deletar uma linha da tabela, utiliza-se este código:

```
PreparedStatement ps = null;  
String sql = "delete from alunos where id=?";  
  
try {  
    ps = con.prepareStatement(sql);  
    ps.setInt(1,2);  
    ps.executeUpdate();  
  
    System.out.println( "Dados excluídos com sucesso." );  
} catch ( SQLException sqle ) {  
    System.out.println( "Erro no acesso ao Bando de Dados : "+ sqle.getMessage());  
}
```

Note que, nesse caso, é preciso “dizer” à variável **sql** somente qual **id** da tabela será deletado. Utiliza-se o **id** por ser a chave primária e não existir a possibilidade de haver dois **ids** iguais. Como há apenas um valor a ser buscado, utiliza-se somente um símbolo “?”. Depois, utiliza-se o comando **ps.setInt(1,2)**, que indica que é um valor inteiro. Como é somente 1, começa-se por esse valor e o conteúdo procurado é o **id=2**. É basicamente essa a diferença entre a deleção e a inserção de dados. O restante dos comandos se repete, como foi visto no código anterior. A tabela agora ficará novamente com uma linha, como você pode ver no MySQL Workbench.

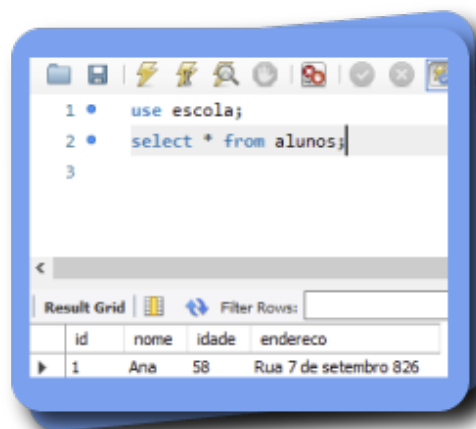


Figura 9 – Tabela com os dados inseridos com **Prepared Statement**

Fonte: MySQL Workbench (2022)

## Atualizando dados com Prepared Statement



A atualização de dados também segue a lógica da inserção e da deleção, o que facilita muito o aprendizado e a implementação dessa forma de manipulação.

```
PreparedStatement ps = null;
String sql = "update alunos set nome = ? where id =1";

try {
    ps = con.prepareStatement(sql);
    ps.setString(1,"Frederico");
    ps.executeUpdate();

    System.out.println( "Dados atualizados com sucesso." );
} catch ( SQLException sqle ) {
    System.out.println( "Erro no acesso ao Bando de Dados : "+ sqle.getMessage());
}
```

Note que, na atualização de dados, também foi utilizado somente um símbolo “?”, pois há somente um campo da tabela que será atualizado, nesse caso, o campo “nome”. Todo o restante do código segue a mesma lógica aprendida anteriormente.

## Padrão DAO (*data access object*)

Agora que você aprendeu os principais processos de manipulação de dados, pode evoluir para um padrão em que consegue separar as classes de acessos aos dados. Até esse momento, todo código está sendo feito no mesmo local, tanto a parte de CRUD (*create, read, update, delete*) quanto os códigos que acessam essas manipulações, ou seja, sempre que você alterar algo referente a banco de dados, terá que efetuar alteração em todas as etapas do processo. Além disso, sempre que precisar manipular qualquer informação com o banco, será preciso alterar os códigos nos quais existem as conexões, o que aumenta o risco de erros e diminui a segurança do projeto, além de aumentar o retrabalho e diminuir a agilidade do código.

Com o padrão DAO, todo código de acesso ao banco de dados fica separado das *views* (telas de visualização). Logo, sempre que algo de acesso e manipulação de dados for alterado, você precisará ajustar apenas a camada de dados DAO, e vice-versa. Será mais fácil entender isso no projeto que se iniciará a seguir. Mas, para possibilitar melhor visualização, veja a diferença entre as pastas do projeto anterior e este projeto utilizando o seguinte padrão:

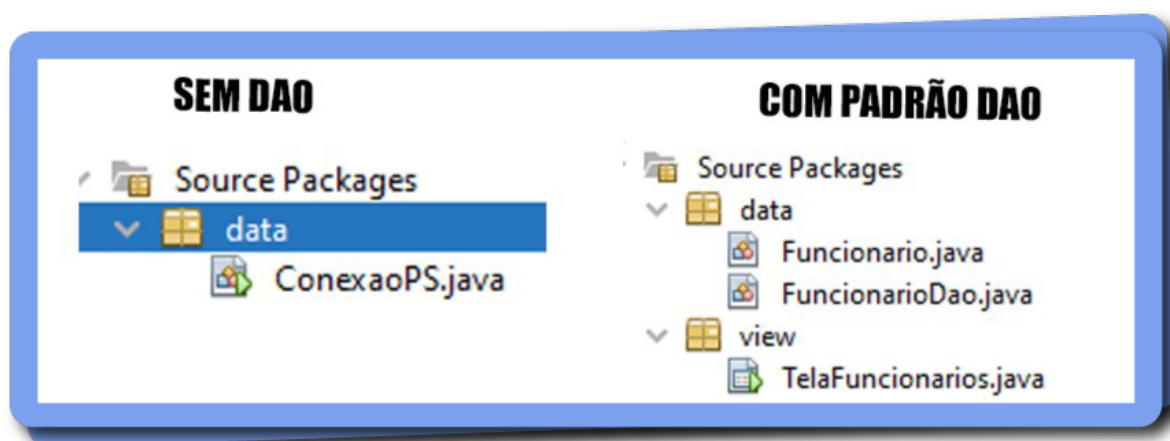


Figura 10 – Diferença das pastas com e sem o padrão DAO

Fonte: NetBeans 13 (2022)

Nessa imagem, note que, com o uso do DAO, há uma separação bem clara entre a parte de acesso aos dados, dentro da pasta **data**, e a parte de visualização e interação do usuário, que acontece na tela **view**, ou seja, isso fortalece o encapsulamento dos dados, pois o usuário que utilizará o sistema usará um arquivo diferente do arquivo que permite as conexões com o banco. Outra vantagem é que, em qualquer modificação na conexão – ou na estrutura da classe “funcionários” ou até nas *views* do projeto –, somente uma classe é alterada, não sendo necessário alterar o projeto como um todo.



Para exemplificar bem as etapas da criação de um projeto no padrão DAO, será criado um sistema com interface visual que englobe as etapas de inserção, atualização, deleção e consulta de dados. Para isso, foi escolhido um sistema de funcionários de uma empresa.

No primeiro passo, cria-se a base de dados que será chamada de “exemplo\_senac”, mas você pode colocar o nome que achar melhor. Dentro dessa base haverá apenas uma tabela chamada “funcionários”.

No Workbench, utiliza-se o código a seguir para a criação do banco:

```
create database exemplo_senac;
```

Depois, utilize este código para a criação da tabela:

```
use escola;
create table funcionarios(
    matricula varchar(70) NOT NULL,
    nome varchar(70) NOT NULL,
    cargo varchar(70) NOT NULL,
    salario double NOT NULL,
    PRIMARY KEY (matricula)
);
```

É possível utilizar a matrícula como inteiro também. Aqui, decidiu-se por **varchar**, aceitando tanto números quanto letras. Com o banco e a tabela criados, comece seu sistema no NetBeans 13 iniciando um projeto novo chamado “BDVisual”, utilizando a categoria **Java With Ant** e o **Projects** como **Java Application**.

Renomeie o pacote (*package*) para “data” utilizando o botão direito do *mouse* e escolhendo o item **Refactor/Rename** e então basta colocar o nome desejado. Sugere-se que você exclua a classe que vem por padrão e comece a criação do zero.

No pacote **data**, crie duas classes Java: **funcionários.java** e **funcionáriosDao.java**.

Depois, crie outro pacote chamado **View** e dentro dele, apenas uma classe chamada **TelaFuncionários.java**. A estrutura deve ficar como a apresentada a seguir:

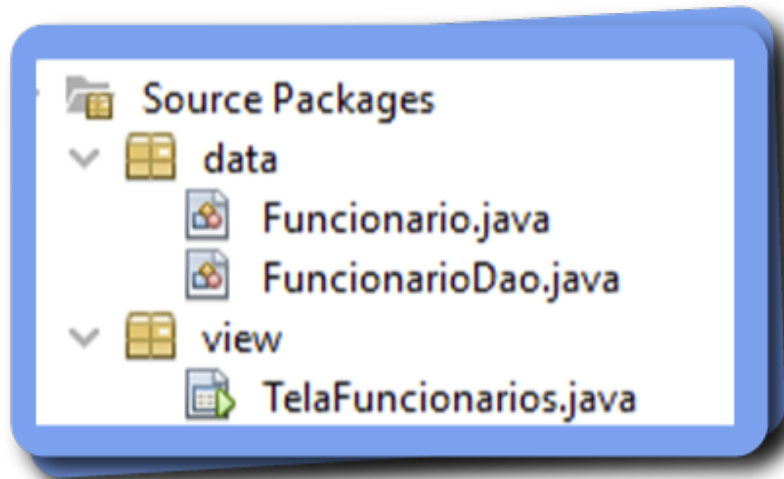


Figura 11 – Estrutura de pastas do projeto “Visual”

Fonte: NetBeans 13 (2022)

## Classe Funcionário

Primeiro, crie a classe **Funcionário.java**, que conterá os atributos e métodos que poderão ser utilizados em cada funcionário salvo na tabela. Para isso, utilize este código:



```
package data;

public class Funcionario {
    // atributos da classe
    private String matricula;
    private String nome;
    private String cargo;
    private double salario;

    //construtor da classe

    public Funcionario() {
    }

    //metodos gets e sets

    public String getMatricula() {
        return matricula;    }

    public void setMatricula(String matricula) {
        this.matricula = matricula;    }

    public String getNome() {
        return nome;    }

    public void setNome(String nome) {
        this.nome = nome;    }

    public String getCargo() {
        return cargo;    }

    public void setCargo(String cargo) {
        this.cargo = cargo;    }

    public double getSalario() {
        return salario;    }

    public void setSalario(double salario) {
        this.salario = salario;    }
}
```

Analisando o código, percebe-se que é uma classe simples, chamada **Funcionário**, com os atributos “matrícula”, “nome”, “cargo” e “salário”, todos do tipo *private* para que possam ser alterados somente com o uso de métodos **GET** e **SET**, o que é chamado de encapsulamento de dados.

Cria-se um método construtor, que não obriga a utilização de todos os atributos, caso não sejam necessários, e depois métodos **GET** e **SET** para cada um dos atributos.

Essa é a estrutura da classe **Funcionário**, e sempre que você criar um objeto com base nela, poderá utilizar todos os seus atributos e métodos.

## Classe Funcionário DAO

Essa será a classe responsável pela conexão e finalização da conexão com o banco e com todas as suas etapas de CRUD para inserção, atualização, deleção e consulta de dados. Utilize o **PreparedStatement** como padrão de acesso, pelos motivos já explicados. Além disso, recomenda-se utilizá-lo sempre que iniciar um projeto do zero, pois ele é mais rápido e seguro.

```
package data;

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

public class FuncionarioDao {

    Connection conn;
    PreparedStatement st;
    ResultSet rs;

    public boolean conectar(){
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/exemp
lo_senac","root", "");
            return true;
        } catch (ClassNotFoundException | SQLException ex) {
            System.out.println("Erro ao conectar: " + ex.getMessage());
            return false;
        }
    }

    public int salvar(Funcionario func){
        int status;
        try {
            st = conn.prepareStatement("INSERT INTO funcionarios VALUES
(?,?,?,?)");
            st.setString(1,func.getMatricula());
            st.setString(2,func.getNome());
            st.setString(3,func.getCargo());
            st.setDouble(4, func.getSalario());
            status = st.executeUpdate();
            return status; //retornar 1
        } catch (SQLException ex) {
            System.out.println("Erro ao conectar: " + ex.getMessage());
            return ex.getErrorCode();
        }
    }

    public Funcionario consultar (String matricula){

        try {
            Funcionario funcionario = new Funcionario();
            st = conn.prepareStatement("SELECT * from funcionarios WHERE matricul
a = ?");
```

```

        st.setString(1, matricula);
        rs = st.executeQuery();
        //verificar se a consulta encontrou o funcionário com a matrícula informada
        if(rs.next()){ // se encontrou o funcionário, vamos carregar os dados
            funcionario.setMatricula(rs.getString("matricula"));
            funcionario.setNome(rs.getString("nome"));
            funcionario.setCargo(rs.getString("cargo"));
            funcionario.setSalario(rs.getDouble("salario"));
            return funcionario;
        }else{
            return null;
        }
    } catch (SQLException ex) {
        System.out.println("Erro ao conectar: " + ex.getMessage());
        return null;
    }
}

public boolean excluir(String matricula){
    try {
        st = conn.prepareStatement("DELETE FROM funcionarios WHERE matricula
= ?");

        st.setString(1,matricula);
        st.executeUpdate();
        return true;
    } catch (SQLException ex) {
        return false;
    }
}

public int atualizar(Funcionario func){
    int status;
    try {
        st = conn.prepareStatement("UPDATE funcionarios SET nome = ?, cargo =
?, salario = ? where matricula = ?");
        st.setString(1,func.getNome());
        st.setString(2,func.getCargo());
        st.setDouble(3, func.getSalario());
        st.setString(4,func.getMatricula());
        status = st.executeUpdate();
        return status; //retornar 1
    } catch (SQLException ex) {
        System.out.println(ex.getErrorCode());
        return ex.getErrorCode();
    }
}

public void desconectar(){
    try {
        conn.close();
    }
}

```

```
        } catch (SQLException ex) {  
            //pode-se deixar vazio para evitar uma mensagem de erro desnecessária  
            ao usuário  
        }  
    }  
}
```

Note que, aqui, a conexão foi criada um pouco diferente da conexão do tipo booleana, para que haja um *return* na classe, *true* quando a conexão for estabelecida e *false* quando ocorrer algum erro. O princípio utilizado em todas as classes desse código é o mesmo aprendido nos processos de manipulação de dados utilizando **PreparedStatement**.

Primeiramente, criam-se as variáveis, pois elas são necessárias para conexão e manipulação de dados, e o **ResultSet** para as consultas:

```
Connection conn;  
PreparedStatement st;  
ResultSet rs;
```

Assim como foi feito anteriormente, carrega-se o *driver* JDBC e estabelece-se a conexão com o banco de dados “exemplo\_senac”.

O método de **inserção de dados**, chamado de *salvar()*, recebe uma variável “func” do tipo *Funcionario* e retorna um valor do tipo *int* 00 1 quando os dados forem inseridos corretamente e 0 quando houver algum erro na inserção. Note que o modo de inserção dos dados é muito semelhante ao já aprendido. Utiliza-se essa segunda forma para que você conheça os dois meios de utilização das variáveis. Confira no código:

```
st = conn.prepareStatement("INSERT INTO funcionarios VALUES(?,?,?,?)");
```

Aqui, não foi criada uma variável **sql** com o código, mas o código foi colocado diretamente dentro da variável **st**. É uma forma resumida de fazer a mesma declaração, mas ambas funcionam perfeitamente.

Depois, os dados são recebidos no local do símbolo “?” e tratadas conforme seu tipo *string* ou *double*. Esse tratamento separado de cada dado é o que aumenta a segurança dessa etapa.

Ao término da classe **salvar()** tem-se a parte de **consulta de dados**, com a classe **consultar()**, que recebe uma *string* **matricula**, a qual será a base para a consulta dos dados. Comece criando um objeto do tipo “funcionário”, que será criado com base na classe **Funcionario.java**:

```
Funcionario funcionario = new Funcionario();
```

Estando o objeto criado, o código de busca **sql** será o seguinte:

```
st = conn.prepareStatement("SELECT * from funcionarios WHERE matricula = ?");
```

Depois do código, ainda há o tratamento das variáveis conforme seu tipo:

```
if(rs.next()){ // se encontrou o funcionário, vamos carregar os dados
    funcionario.setMatricula(rs.getString("matricula"));
    funcionario.setNome(rs.getString("nome"));
    funcionario.setCargo(rs.getString("cargo"));
    funcionario.setSalario(rs.getDouble("salario"));
    return funcionario;
}else{
    return null;
}
```

Veja que os valores “matricula”, “nome”, “cargo” e “salario” são provenientes da classe **funcionário.java** e é devido a isso a criação do objeto no início do código. Da mesma forma aprendida, utiliza-se o **rs.next()** para percorrer todos os dados da tabela.

Ao término da consulta, começa-se a parte de **exclusão dos dados**, com a classe **excluir()**, que também recebe como atributo a *string* **matricula**. Esta parte é a mais simples entre todos os processos e simplesmente usa o código SQL a seguir para excluir uma linha da tabela com base em sua matrícula recebida:

```
st = conn.prepareStatement("DELETE FROM funcionarios WHERE matricula = ?");
```

Logo após a exclusão, tem-se a parte de **atualização dos dados** com a classe **atualizar()**, que recebe uma variável **func** do tipo “funcionário” e contém um comando *update* que utiliza quatro símbolos “?”, tratando cada um conforme sua posição e seu tipo na tabela:

```
st = conn.prepareStatement("UPDATE funcionarios SET nome = ?, cargo = ?, salario = ? where matricula = ?");
st.setString(1,func.getNome());
st.setString(2,func.getCargo());
st.setDouble(3, func.getSalario());
st.setString(4,func.getMatricula());
```

Para finalizar, ainda há a classe **desconectar()**, que deve ser utilizada sempre que finalizado o uso do banco.

## Classe Tela Funcionário

Neste momento será iniciada a criação do pacote **view**, visto na figura 11. Nele, utilize a classe **TelaFuncionários.Java**, na qual será criado o formulário que viabiliza a interação do usuário com o sistema.

Essa será a parte em que o usuário visualizará o projeto e poderá interagir, inserir, consultar, atualizar ou excluir dados. Para isso, antes de tudo, deve-se criar uma tela utilizando os conhecimentos de Java Swing. A tela ficará da seguinte forma:



Figura 12 – Tela do sistema

Fonte: NetBeans 13 (2022)

Para começar, ficaram definidos os nomes das caixas de texto: **txtMatricula**, **txtNome**, **txtCargo** e **txtSalario**. Os botões têm os seguintes nomes, respectivamente: **btnSalvar**, **btnConsultar**, **btnExcluir** e **btnAtualizar**. É preciso que os nomes sejam exatamente iguais a esses para que o código que virá a seguir funcione. Se você quiser utilizar outros nomes, eles devem ser alterados também no código. Por isso, sugere-se que você, ao menos nesta primeira vez, utilize esses mesmos nomes para melhor entendimento do conteúdo.



Todos os códigos ficam dentro de cada um dos quatro botões, portanto haverá um código para o botão **Salvar**, um para o botão **Consultar** e assim sucessivamente.

Confira a explicação dos códigos pelo botão **Inserir**.

## Botão Inserir

Quando se clica duas vezes no botão **Salvar**, abre-se o código a seguir, começando pelas bibliotecas que devem ser importadas:

```
import data.Funcionario;  
import data.FuncionarioDao;  
import javax.swing.JOptionPane;
```

Aqui, são importadas as duas classes criadas dentro do pacote **data** e a classe **JOptionPane**, que é necessária quando se precisa usar caixas de diálogos no Java Swing.

Dentro do botão **Salvar** está o seguinte código:

Dois cliques acima do botão **Salvar** já abrem o local correto no qual se deve inserir o código, que é a classe mostrada a seguir:

```

private void btnSalvarActionPerformed(java.awt.event.ActionEvent evt) {

    Funcionario funcionario = new Funcionario();
    FuncionarioDao dao =new FuncionarioDao();
    boolean status;
    int resposta;

    funcionario.setMatricula(txtMatricula.getText());
    funcionario.setNome(txtNome.getText());
    funcionario.setCargo(txtCargo.getText());
    funcionario.setSalario(Double.parseDouble(txtSalario.getText()));

    dao = new FuncionarioDao();

    status = dao.conectar();
    if(status == false){
        JOptionPane.showMessageDialog(null,"Erro de conexão");
    }else{
        resposta = dao.salvar(funcionario);
        if(resposta == 1){
            JOptionPane.showMessageDialog(null,"Dados incluídos com suc
esso");

            //limpar os campos
            txtMatricula.setText("");
            txtNome.setText("");
            txtCargo.setText("");
            txtSalario.setText("");
            //posicionar o cursor para um próximo
            txtMatricula.requestFocus();
        }else if (resposta ==1062){
            JOptionPane.showMessageDialog(null,"Matricula já foi cadast
rada");
        }else{
            JOptionPane.showMessageDialog(null,"Erro ao tentar inserir
dados");
        }
        dao.desconectar();
    }
}
}

```

Primeiro, comece criando o objeto “funcionário” do tipo “Funcionário” e o objeto DAO do tipo “Funcionario Dao”, pois seus atributos e métodos serão necessários no código, assim como as variáveis *status* (booleana) e *resposta* (inteira), cuja função já foi mostrada anteriormente.

Para você relembrar da forma de receber um valor vindo de um formulário em Java, veja dois exemplos do código:

```
funcionario.setMatricula(txtMatricula.getText());  
funcionario.setSalario(Double.parseDouble(txtSalario.getText()));
```

No caso de salário, basta usar o **txtMatricula.getText()** para receber o valor da matrícula do campo de texto e colocá-lo como atributo do método **setMatricula** do objeto “funcionário”. No caso de matrícula, basta usar o **txtMatricula.getText()** para receber o valor da matrícula do campo de texto e colocá-lo como atributo do método **setMatricula()** do objeto "funcionario". No caso de **setSalario()**, usamos o método **Double.parseDouble()** para converter o valor textual do salário recebido do campo de texto txtSalario.

Com os dados recebidos do formulário, utiliza-se o código **resposta = dao.salvar(funcionario);** para obter os valores e salvá-los no banco utilizando a classe **salvar()**. Se tudo estiver correto, retornará um valor “1”, que indica que os dados foram salvos, e então deve-se retornar os valores do formulário para vazio com estes comandos:

```
txtMatricula.setText("");  
txtNome.setText("");  
txtCargo.setText("");  
txtSalario.setText("");
```

O comando a seguir testa se o campo já existe na base de dados com um valor pré-cadastrado do Java.

```
(resposta ==1062){
```

No final no código, não se pode esquecer de desconectar o banco, por ser a política correta de uso de banco de dados.

## Botão Consultar

Dê dois cliques acima do botão consultar e insira este código:

```
private void btnConsultarActionPerformed(java.awt.event.ActionEvent evt) {  
  
    String matricula;  
    matricula = txtMatricula.getText();  
    FuncionarioDao dao = new FuncionarioDao();  
    boolean status = dao.conectar();  
    if(status == true){  
        Funcionario funcionario = dao.consultar(matricula);  
        System.out.println(dao.consultar(matricula));  
        if(funcionario == null){  
            JOptionPane.showMessageDialog(null,"Funcionário não localizado");  
        }else{  
            txtNome.setText(funcionario.getNome());  
            txtCargo.setText(funcionario.getCargo());  
            txtSalario.setText(String.valueOf(funcionario.getSalario()));  
            //habilitar botão de exclusão  
            btnExcluir.setEnabled(true);  
        }  
        dao.desconectar();  
    }else{  
        JOptionPane.showMessageDialog(null,"Erro de conexão");  
    }  
}  
}
```

A base para consulta, como visto anteriormente, é o campo “matrícula”. Por isso, o primeiro passo é criar uma variável que receberá a matrícula e armazenará o valor por meio deste comando:

```
matricula = txtMatricula.getText();
```

Em todos os projetos, é preciso criar o objeto “dao” para conexão. Para realizar a consulta, utilize este comando:

```
Funcionario funcionario = dao.consultar(matricula);
```

Se o resultado for nulo, o comando indicará que nada foi encontrado, do contrário, ele seta o valores nas caixas de texto com os comandos a seguir e habilita o botão **Excluir**.

```
txtNome.setText(funcionario.getNome());  
txtCargo.setText(funcionario.getCargo());  
txtSalario.setText(String.valueOf(funcionario.getSalario()));  
//habilitar botão de exclusão  
btnExcluir.setEnabled(true);
```

## Botão Excluir

Dê dois cliques acima do botão excluir e insira este código:

```

private void btnConsultarActionPerformed(java.awt.event.ActionEvent evt) {

FuncionarioDao dao = new FuncionarioDao();
    boolean status = dao.conectar();
    if(status == false){
        JOptionPane.showMessageDialog(null,"Erro de conexão");
    }else{
        status = dao.excluir(txtMatricula.getText());
        if(status ==true){
            //int confirma = JOptionPane.showConfirmDialog(null,"Tem certeza
a?");
            JOptionPane.showMessageDialog(null,"Funcionário excluído com suc
esso!");

            txtMatricula.setText("");
            txtMatricula.requestFocus();
            txtNome.setText("");
            txtCargo.setText("");
            txtSalario.setText("");
            //desabilitar botão de exclusão
            btnExcluir.setEnabled(false);
        }else{
            JOptionPane.showMessageDialog(null,"Erro na exclusão do funcioná
rio!");
        }
        dao.desconectar();
    }
}

```

Assim como no **Salvar()**, o comando inicia os objetos do tipo “Funcionário” e “FuncionarioDao” e depois, com a matrícula recebida, ele utiliza o comando a seguir para excluir a linha de código:

```
status = dao.excluir(txtMatricula.getText());
```

Depois da exclusão, os dados são mostrados como vazio na caixa de texto, como já abordado anteriormente.

## Botão Atualizar



Dê dois cliques acima do botão atualizar e insira este código:

```
private void btnAtualizarActionPerformed(java.awt.event.ActionEvent evt) {
    Funcionario funcionario = new Funcionario();
    FuncionarioDao dao;
    boolean status;
    int resposta;

    funcionario.setMatricula(txtMatricula.getText());
    funcionario.setNome(txtNome.getText());
    funcionario.setCargo(txtCargo.getText());
    funcionario.setSalario(Double.parseDouble(txtSalario.getText()));
    dao = new FuncionarioDao();
    status = dao.conectar();
    if(status == false){
        JOptionPane.showMessageDialog(null,"Erro de conexão");
    }else{
        resposta = dao.atualizar(funcionario);
        System.out.println(funcionario.getMatricula());
        if(resposta == 1){
            JOptionPane.showMessageDialog(null,"Dados atualizados com sucesso");

            //limpar os campos
            txtMatricula.setText("");
            txtNome.setText("");
            txtCargo.setText("");
            txtSalario.setText("");
            //posicionar o cursor para um próximo
            txtMatricula.requestFocus();
        }else if (resposta == 1062){
            JOptionPane.showMessageDialog(null,"Matricula já foi cadastrada");
        }else{
            JOptionPane.showMessageDialog(null,"Erro ao tentar inserir dados");
        }
        dao.desconectar();
    }
}
```

É interessante notar que, depois de compreender a utilização das classes em um dos modos de manipulação, torna-se muito mais fácil entender os outros módulos, pois os meios utilizados são muito semelhantes entre si. Neste caso também precisamos criar um objeto **Funcionario** para ser informado aos métodos do objeto da classe **FuncionarioDao**. Este objeto de **Funcionario** é preenchido com os dados dos quatro atributos, que são obtidos dos campos de texto da tela. Dessa maneira podemos realizar a atualização do registro a usando o seguinte comando:

```
resposta = dao.atualizar(funcionario);
```

O restante do código todo já foi analisado nos exemplos anteriores.

Aqui, há um sistema completo com todas etapas de manipulação do banco de dados em Java, com uso do padrão DAO, e todo funcionando com uma interação visual com o usuário final.

## Encerramento

Com certeza, este conhecimento sobre manipulação de dados é um grande divisor de águas para todo programador, pois permite não apenas mostrar o resultado para o usuário, como se fazia anteriormente, mas também armazenar e modificar esses dados para uso posterior.

Com esse conhecimento, você poderá incrementar seu sistema com poder, garantindo mais segurança e agilidade para seus processos, possibilitando também infinitos novos processos de interação do usuário por meio de inserção, atualização, deleção e consulta de dados. Com esse incremento, certamente o seu sistema atinge um outro nível de interesse.

Portanto, é importante que você domine os conhecimentos abordados neste conteúdo, pois eles o acompanharão durante toda sua vida como programador.