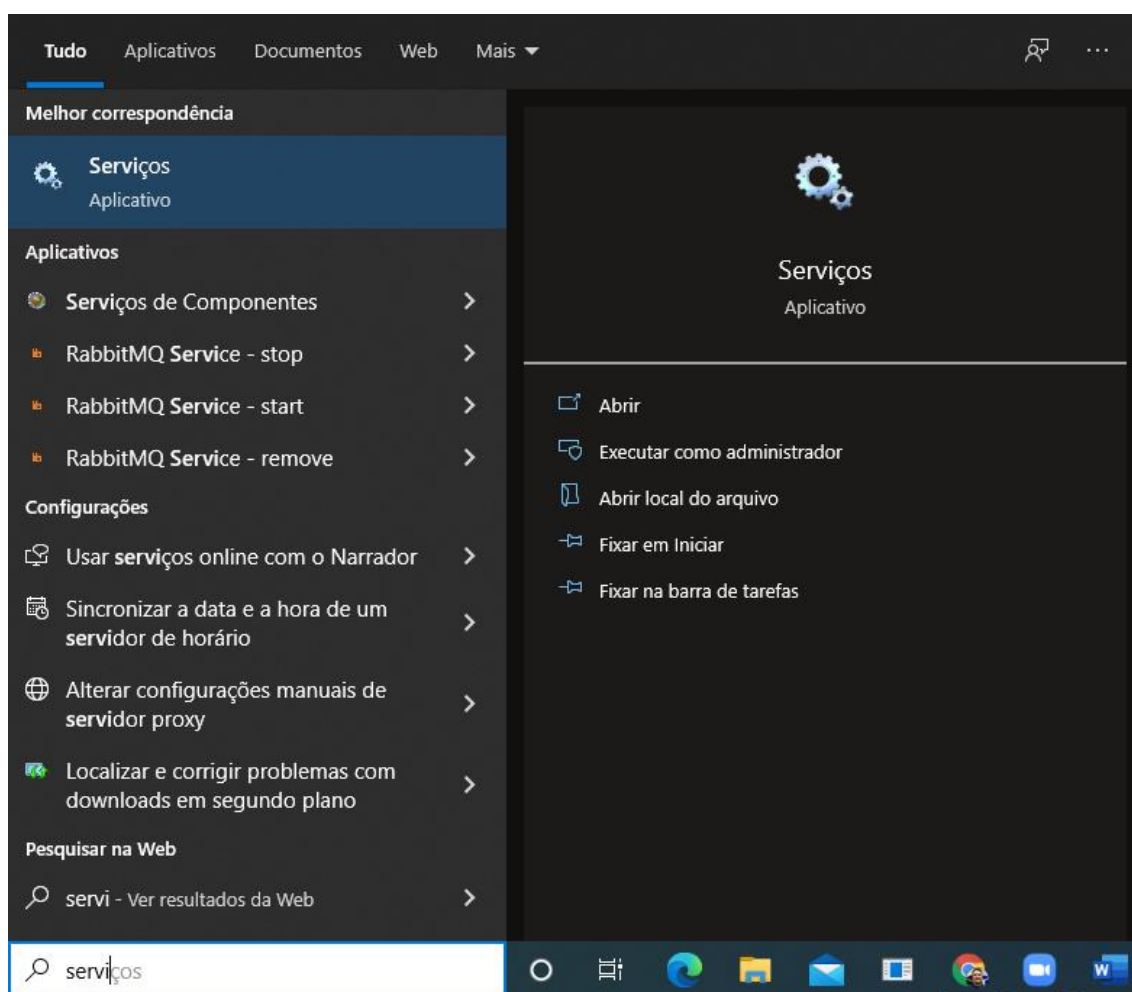


Instalando o MySQL

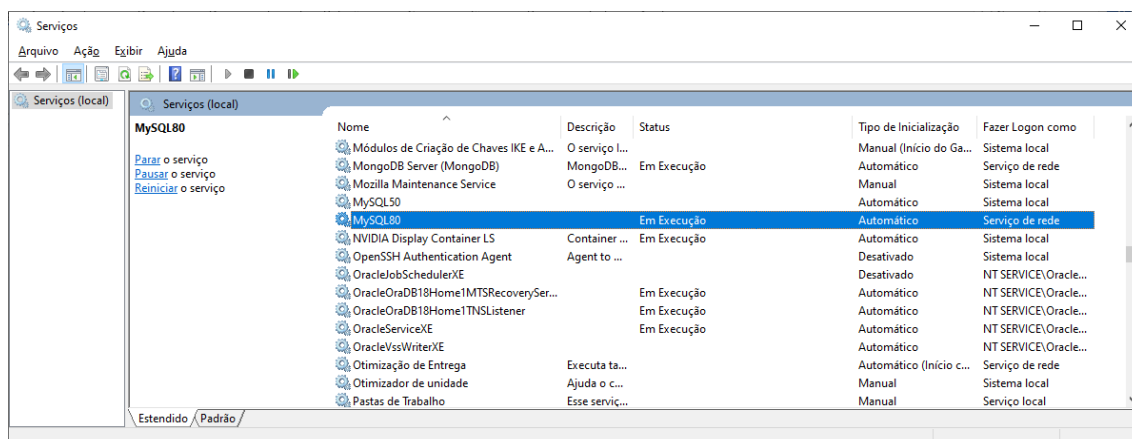
Servidor de banco de dados.

<https://dev.mysql.com/downloads/windows/installer/8.0.html>

- Inicializando o serviço:



Serviço em execução: (MYSQL80)



Abrindo a ferramenta de administração da base de dados: MYSQL WORKBENCH

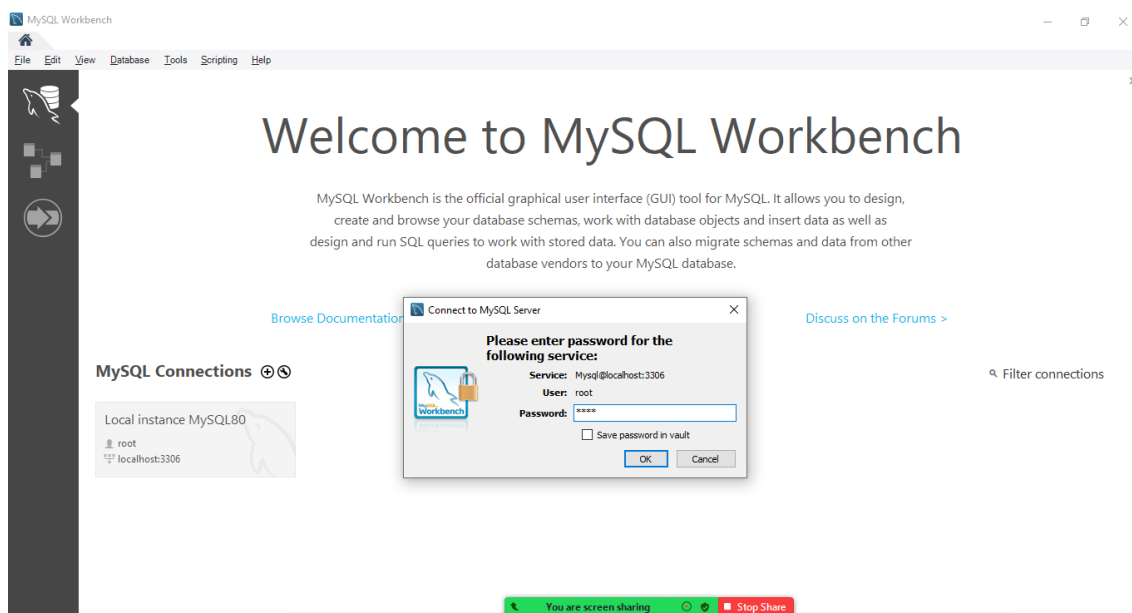
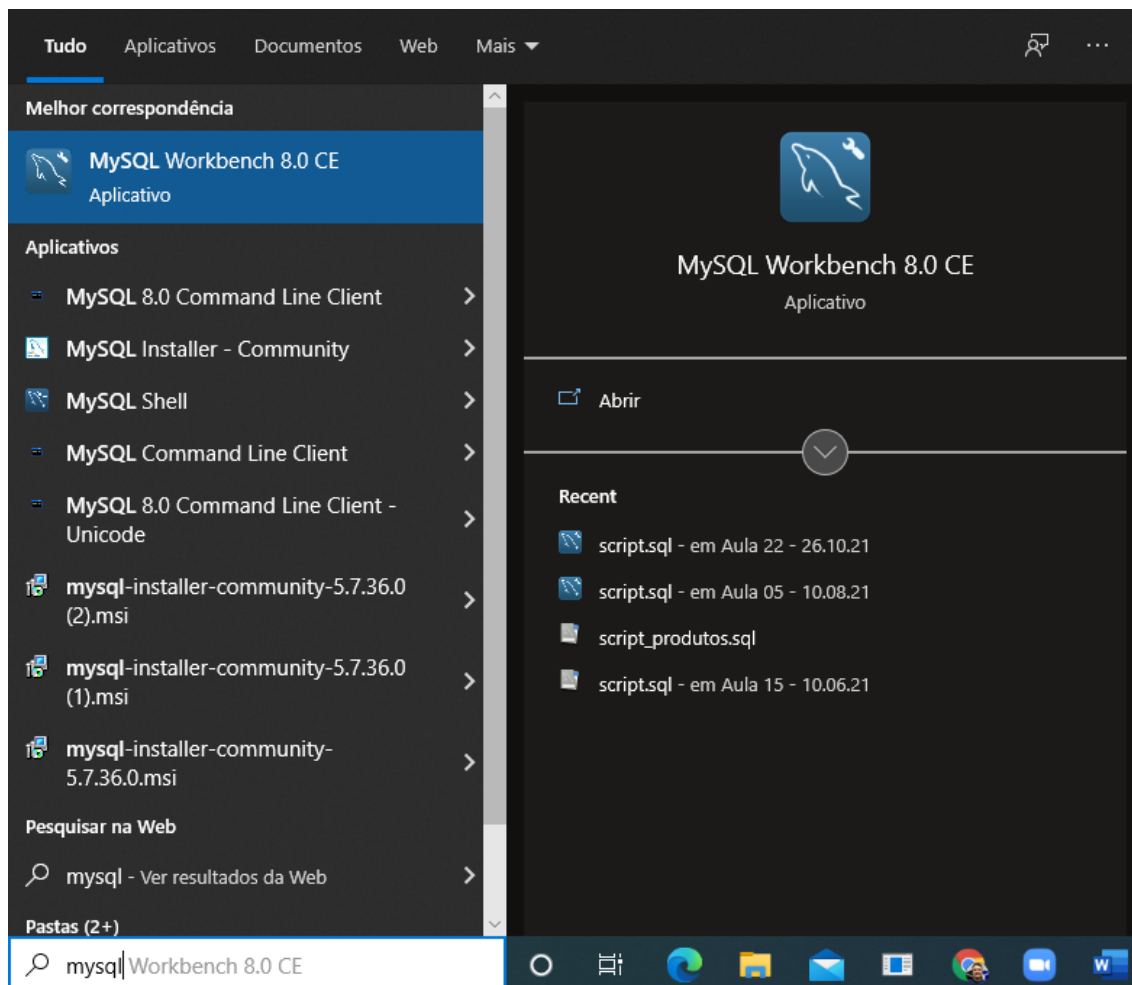
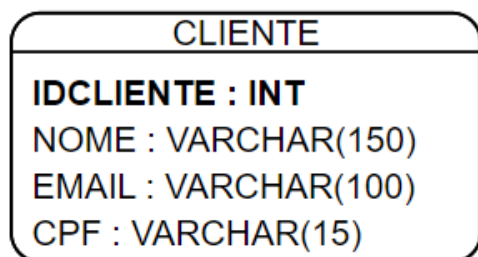


Diagrama de Entidade / Relacionamento (DER)

Modelo do banco de dados

<<TABELA SQL>>



#criando a base de dados

CREATE DATABASE AULA05JAVA;

#acessando a base de dados

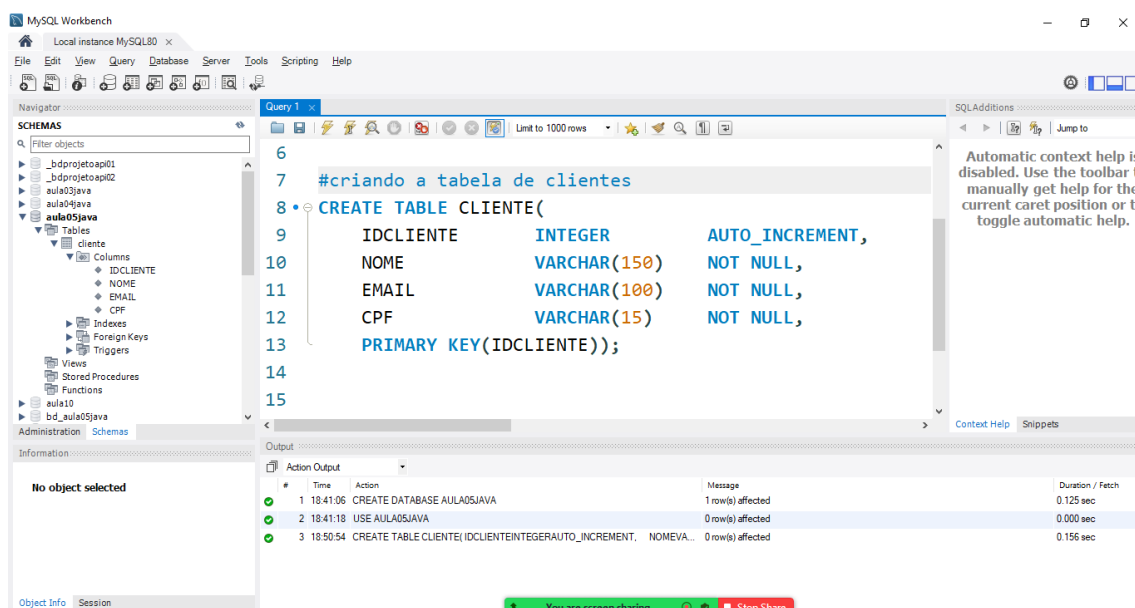
USE AULA05JAVA;

#criando a tabela de clientes

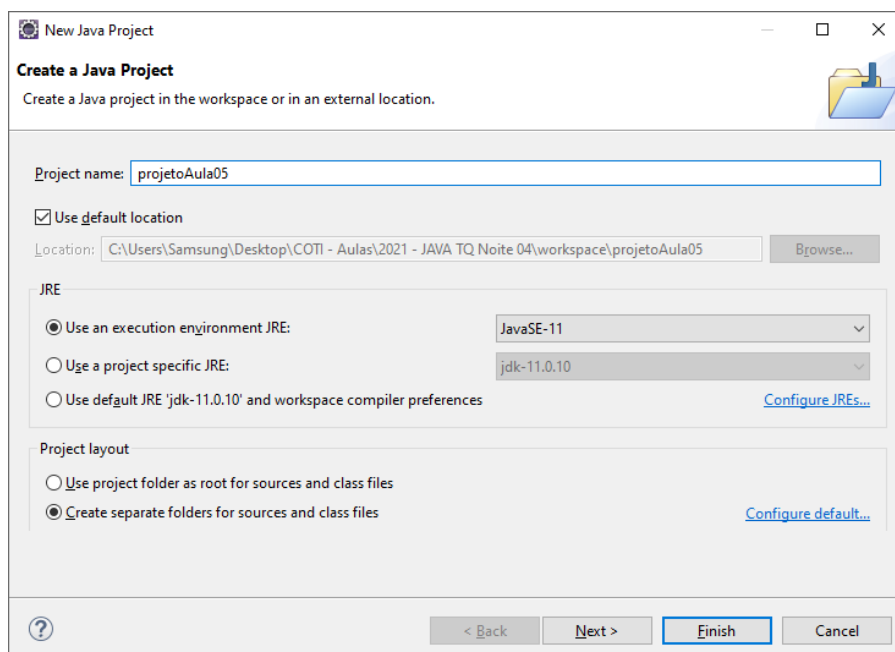
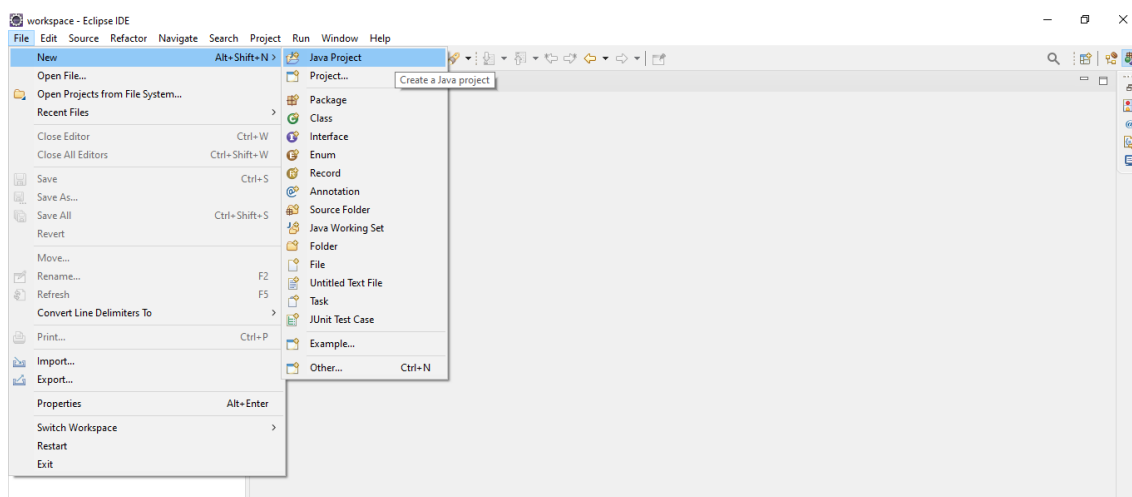
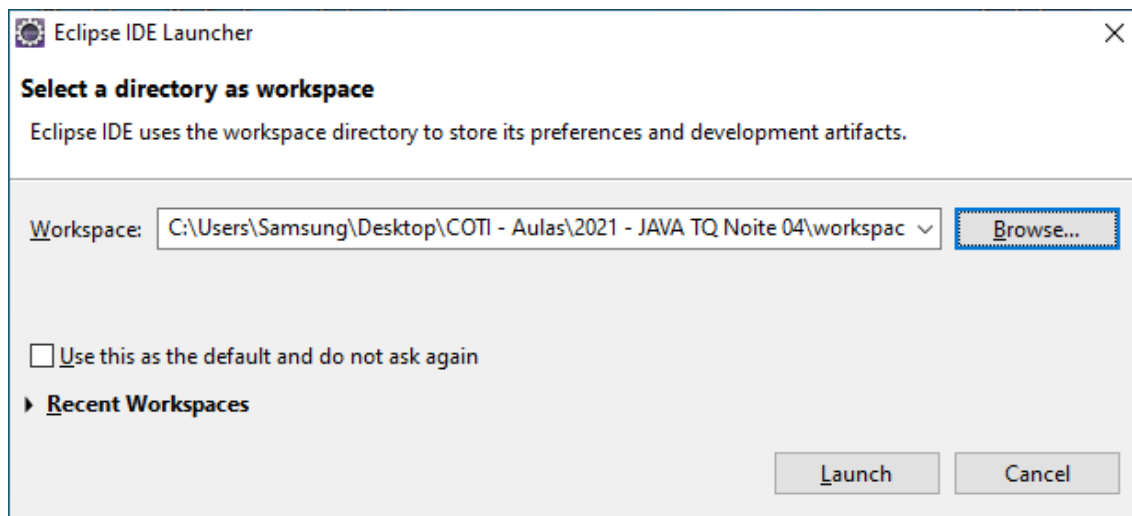
```

CREATE TABLE CLIENTE(
    IDCLIENTE          INTEGER          AUTO_INCREMENT,
    NOME               VARCHAR(150)      NOT NULL,
    EMAIL              VARCHAR(100)      NOT NULL,
    CPF                VARCHAR(15)       NOT NULL,
    PRIMARY KEY(IDCLIENTE));
  
```

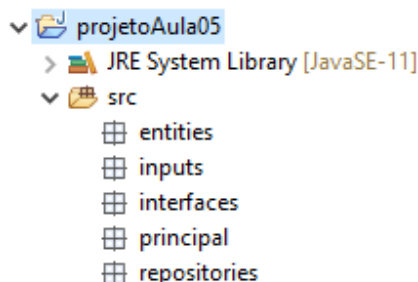
Tabela criada:



Criando um projeto Java:



Criando os pacotes do projeto:



/entities/**Cliente.java**

Classe de entidade similar a tabela criada no banco de dados. Para criarmos este modelo de dados iremos seguir o padrão **JAVABEAN**, ou seja, a classe irá conter:

- Atributos privados
- Construtor sem argumentos
- Construtor com argumentos
- Métodos de encapsulamento
 - Setters
 - Getters
- Sobrescrita dos métodos da classe Object
 - toString
 - equals
 - hashCode

```
package entidades;
```

```
public class Cliente {
```

```
    // atributos privados
```

```
    private Integer idCliente;
```

```
    private String nome;
```

```
    private String email;
```

```
    private String cpf;
```

```
    // construtor sem argumentos
```

```
    public Cliente() {
```

```
        // TODO Auto-generated constructor stub
```

```
    }
```

```
    // construtor com entrada de argumentos (sobrecarga de método)
```

```
    public Cliente(Integer idCliente, String nome,  
                    String email, String cpf) {
```

```
        super();
```

```
        this.idCliente = idCliente;
```

```
        this.nome = nome;
```

```
        this.email = email;
```

```
        this.cpf = cpf;
```

```
    }
```

```
// métodos setters (atribuição) e getters (saída)
public Integer getIdCliente() {
    return idCliente;
}

public void setIdCliente(Integer idCliente) {
    this.idCliente = idCliente;
}

public String getNome() {
    return nome;
}

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

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getCpf() {
    return cpf;
}

public void setCpf(String cpf) {
    this.cpf = cpf;
}

// sobrescrita do método toString da classe Object
@Override
public String toString() {
    return "Cliente [idCliente=" + idCliente
        + ", nome=" + nome + ", email="
        + email + ", cpf=" + cpf + "]";
}
}
```

Próximo passo:

Criar uma classe no pacote **/inputs** composta de métodos **estáticos** para fazer a leitura de cada campo da classe Cliente.

/inputs/ClienteInput.java

```
package inputs;

import java.util.Scanner;
```

```
public class ClienteInput {

    // método para ler o id do cliente
    public static Integer lerIdCliente() {

        @SuppressWarnings("resource")
        Scanner scanner = new Scanner(System.in);

        System.out.print("Informe o id do cliente.....: ");
        return Integer.parseInt(scanner.nextLine());
    }

    // método para ler o nome do cliente
    public static String lerNome() {

        @SuppressWarnings("resource")
        Scanner scanner = new Scanner(System.in);

        System.out.print("Informe o nome do cliente....: ");
        return scanner.nextLine();
    }

    // método para ler o email do cliente
    public static String lerEmail() {

        @SuppressWarnings("resource")
        Scanner scanner = new Scanner(System.in);

        System.out.print("Informe o email do cliente...: ");
        return scanner.nextLine();
    }

    // método para ler o cpf do cliente
    public static String lerCpf() {

        @SuppressWarnings("resource")
        Scanner scanner = new Scanner(System.in);

        System.out.print("Informe o cpf do cliente.....: ");
        return scanner.nextLine();
    }
}
```

Criando a classe para executarmos o projeto:

/principal/**Program.java**

```
package principal;

import entities.Cliente;
import inputs.ClienteInput;
```



```
public class Program {

    public static void main(String[] args) {

        try {

            System.out.println("\nCADASTRO DE CLIENTE:\n");

            Cliente cliente = new Cliente();

            cliente.setNome(ClienteInput.LerNome());
            cliente.setEmail(ClienteInput.LerEmail());
            cliente.setCpf(ClienteInput.LerCpf());

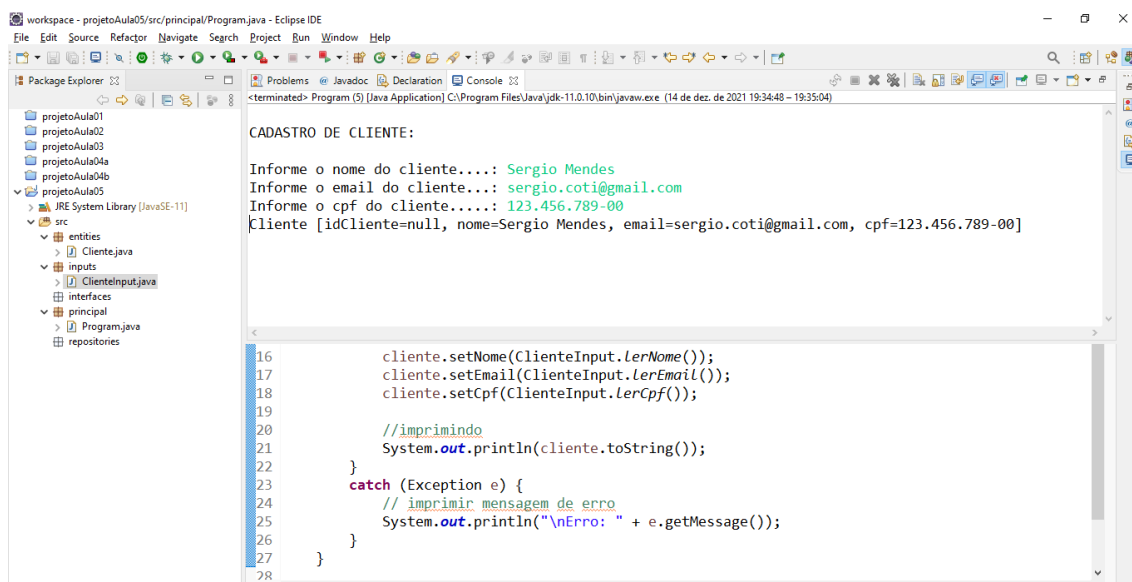
            //imprimindo
            System.out.println(cliente.toString());

        } catch (Exception e) {
            // imprimir mensagem de erro
            System.out.println("\nErro: " + e.getMessage());
        }

    }

}
```

Testando:



CADASTRO DE CLIENTE:

Informe o nome do cliente....: Sergio Mendes
 Informe o email do cliente....: sergio.coti@gmail.com
 Informe o cpf do cliente.....: 123.456.789-00

Cliente [idCliente=null, nome=Sergio Mendes,
 email=sergio.coti@gmail.com, cpf=123.456.789-00]

JDBC – JAVA DATABASE CONNECTIVITY

Biblioteca Java responsável por conectar as aplicações desenvolvidas em Java em uma base de dados.

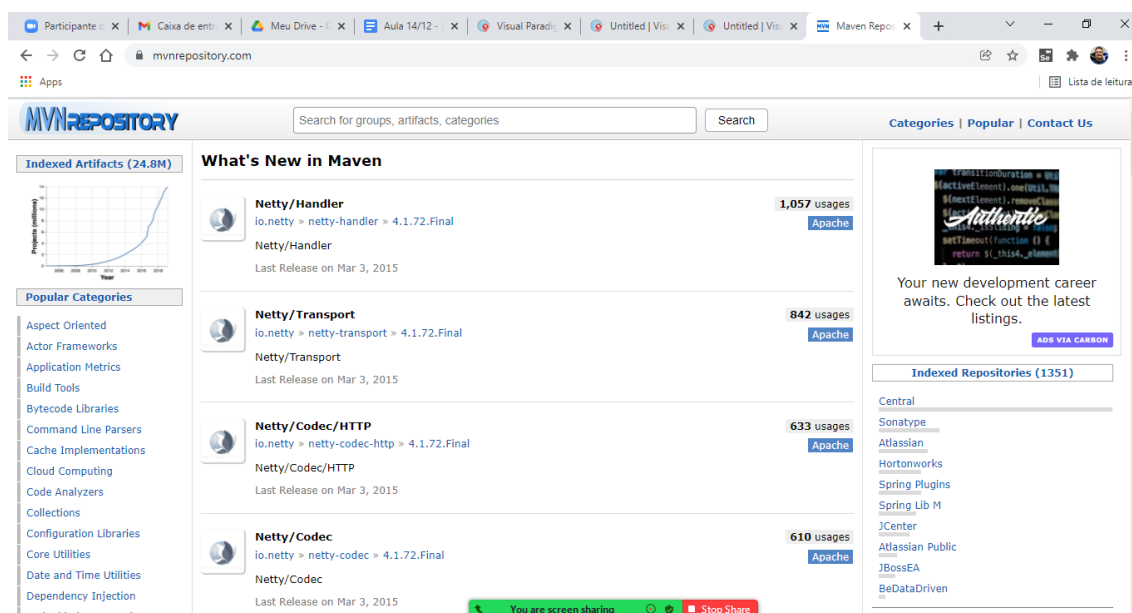
Para tipo de banco de dados que pretendemos utilizar nós precisamos incluir no projeto o **DRIVER JDBC** correspondente do banco de dados que estamos trabalhando. Exemplo: MYSQL, ORACLE, POSTGRE etc.

Precisamos, no nosso caso, baixar o **DRIVER JDBC para o MYSQL**.

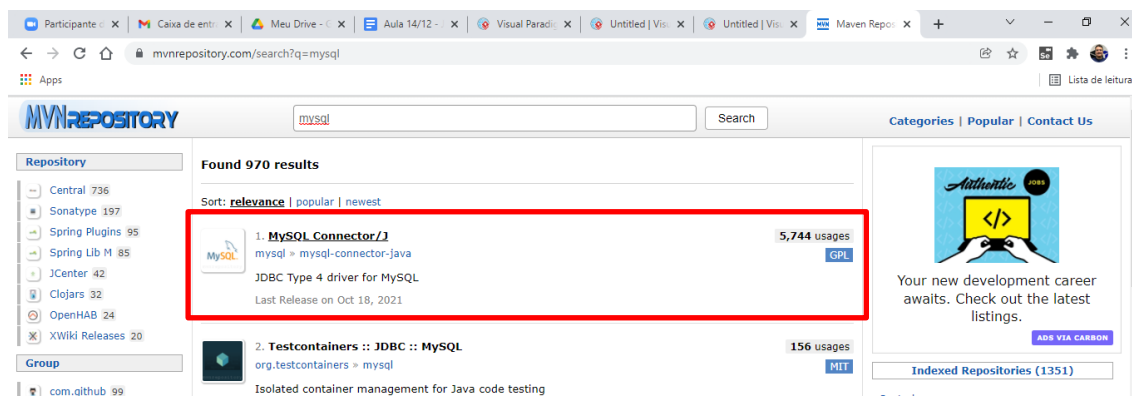
Como estamos ainda trabalhando em POO (programação orientada a objetos), iremos baixar o DRIVER manualmente e depois adicioná-lo também de forma manual no projeto.

Quando estivermos desenvolvendo aplicações web nas próximas aulas iremos utilizar um framework chamado **MAVEN** que irá instalar essas bibliotecas de forma automatizada.

<https://mvnrepository.com/>

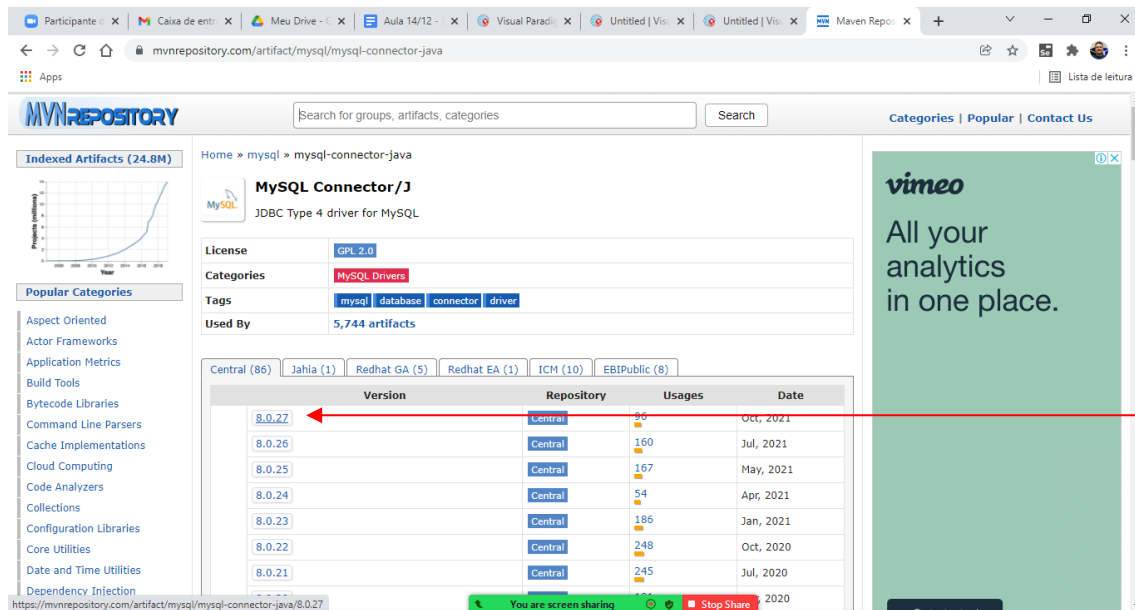


The screenshot shows the Maven Repository website. The 'What's New in Maven' section is visible, listing several updates. The first update, 'Netty/Handler', is highlighted with a red box. It shows the version 'io.netty > netty-handler > 4.1.72.Final' and the release date 'Last Release on Mar 3, 2015'. The update is also marked as '1,057 usages' and 'Apache' licensed.



The screenshot shows the Maven Repository website with search results for 'mysql'. The first result, 'MySQL Connector/J', is highlighted with a red box. It shows the version 'mysql > mysql-connector-java' and the release date 'Last Release on Oct 18, 2021'. The update is also marked as '5,744 usages' and 'GPL' licensed.

Baixando a versão 8.0.27



MySQL Connector/J
JDBC Type 4 driver for MySQL

License: GPL 2.0

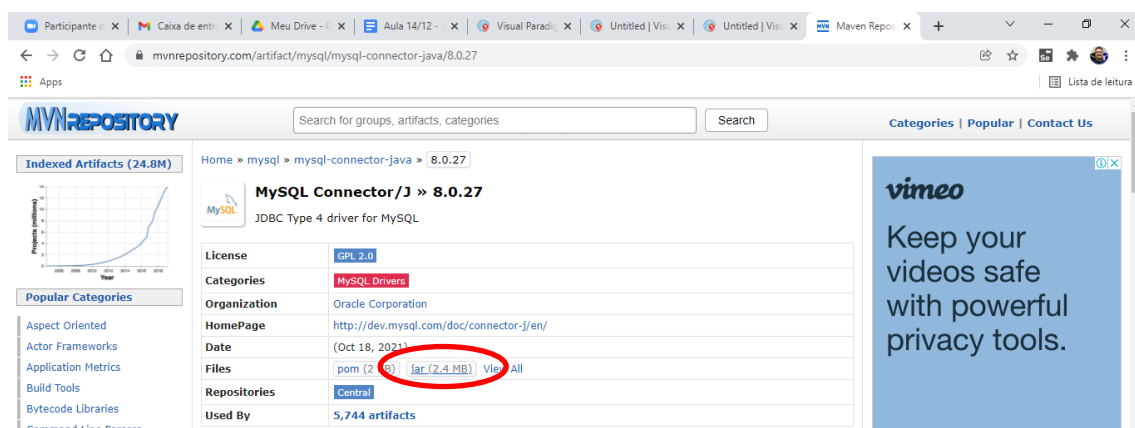
Categories: MySQL Drivers

Tags: mysql, database, connector, driver

Used By: 5,744 artifacts

Version	Repository	Usages	Date
8.0.27	Central	36	Oct, 2021
8.0.26	Central	160	Jul, 2021
8.0.25	Central	167	May, 2021
8.0.24	Central	54	Apr, 2021
8.0.23	Central	186	Jan, 2021
8.0.22	Central	248	Oct, 2020
8.0.21	Central	245	Jul, 2020

<https://mvnrepository.com/artifact/mysql/mysql-connector-java/8.0.27>



MySQL Connector/J » 8.0.27
JDBC Type 4 driver for MySQL

License: GPL 2.0

Categories: MySQL Drivers

Organization: Oracle Corporation

HomePage: <http://dev.mysql.com/doc/connector-j/en/>

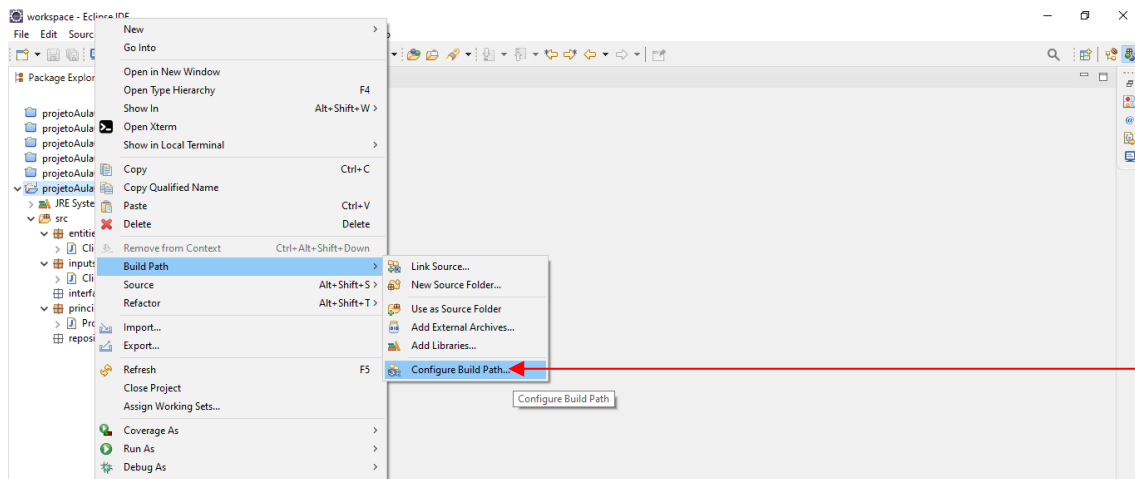
Date: (Oct 18, 2021)

Files: pom (2.6 KB) **jar (2.4 MB)** [View All](#)

Repositories: Central

Used By: 5,744 artifacts

Adicionando referência no projeto para esta biblioteca:



workspace - Eclipse IDE

File Edit Source Window Help

Package Explorer

- projetoAula
- projetoAula
- projetoAula
- projetoAula
- projetoAula
- src
- entidade
- inputs
- interf
- princ
- reposit

Build Path

Source

Refactor

Import...

Export...

Refresh

Close Project

Assign Working Sets...

Coverage As

Run As

Debug As

Profile As

Link Source...

New Source Folder...

Use as Source Folder

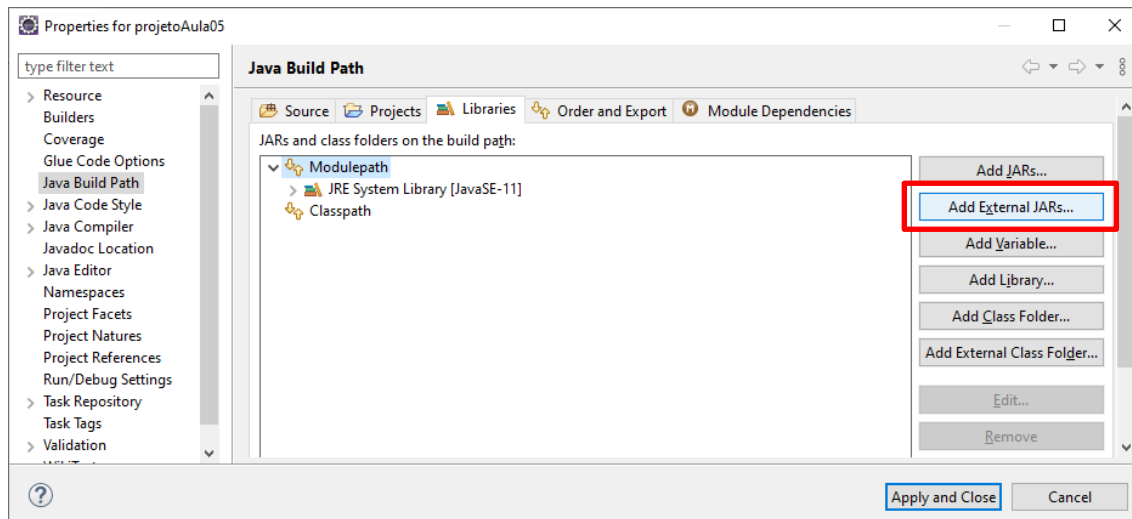
Add External Archives...

Add Libraries...

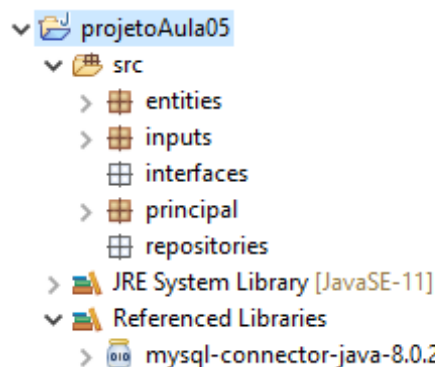
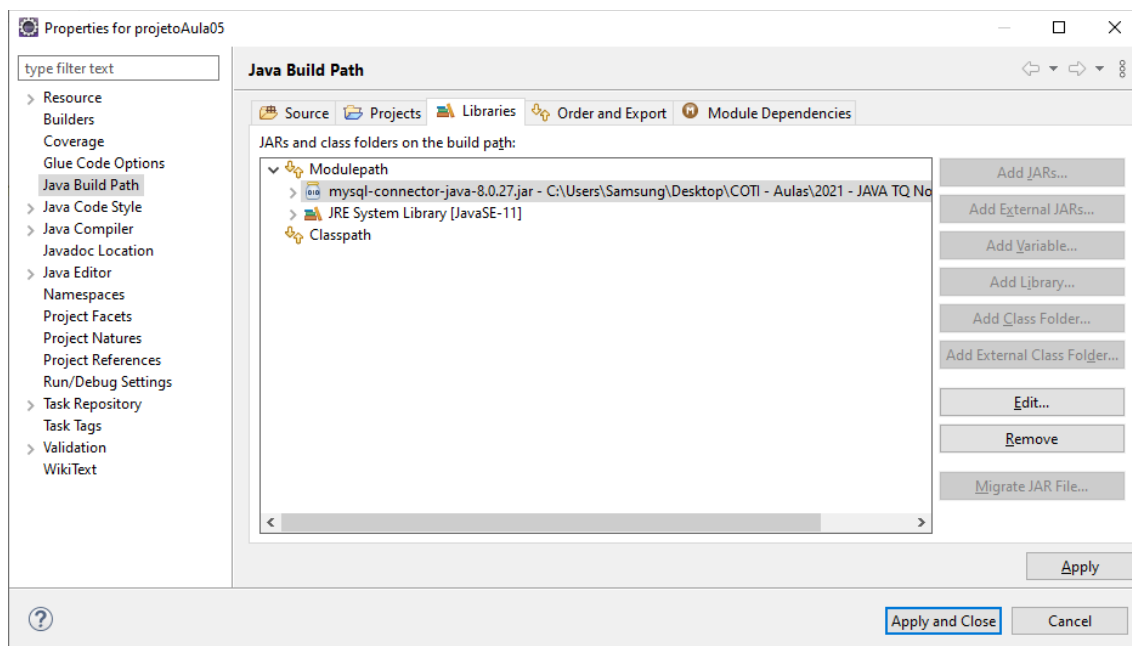
Configure Build Path...

ADD EXTERNAL JARS

Adicionando bibliotecas externas.



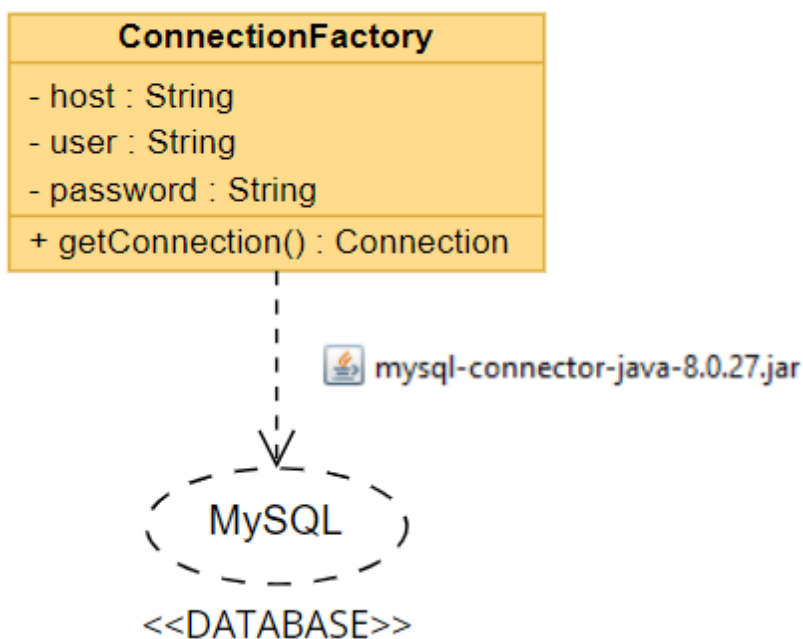
Selecione o conector do MySQL:



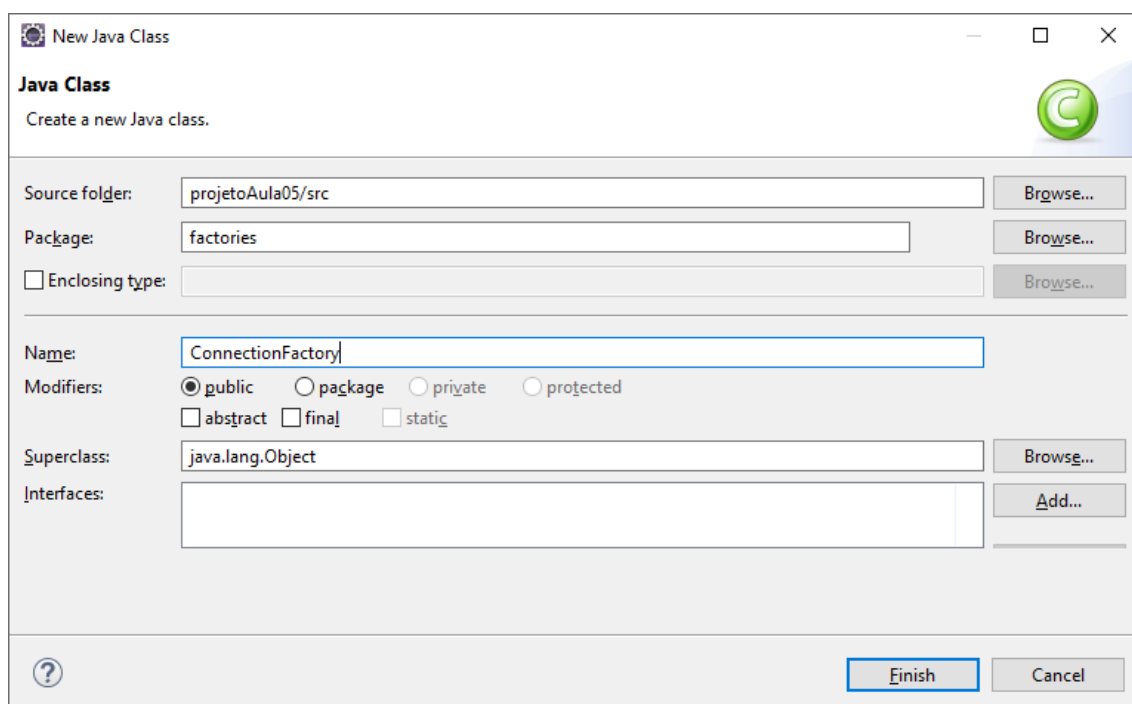
Para que possamos conectar na base de dados, iremos criar uma classe Java somente para esta finalidade (Abrir uma conexão com o MySQL)

Esta classe será uma "Fábrica de conexões" pois ela terá um método que irá retornar conexões com uma base de dados do MySQL, neste caso o banco criado com o nome: **AULA05JAVA**

/factories



Criando a classe:



The screenshot shows the 'New Java Class' dialog box. The 'Java Class' section is active. The 'Source folder' is set to 'projetoAula05/src'. The 'Package' is set to 'factories'. The 'Name' field contains 'ConnectionFactory'. The 'Modifiers' section has 'public' selected. The 'Superclass' is set to 'java.lang.Object'. The 'Interfaces' field is empty. The 'Finish' button is highlighted.

```
package factories;

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

public class ConnectionFactory {

    // atributos de valores já definidos (constantes)
    private static final String HOST = "jdbc:mysql://localhost:3306/
        AULA05JAVA?useTimezone=true
        &serverTimezone=UTC&useSSL=false";
    private static final String USER = "root";
    private static final String PASSWORD = "coti";

    // método para criar e retornar uma conexão com o banco de dados
    public static Connection getConnection() throws Exception {

        // criando e retornando uma conexão com o banco de dados
        return DriverManager.getConnection(HOST, USER, PASSWORD);
    }
}
```

Padrão Repository

Iremos criar classes para persistir as informações de clientes em uma base de dados (GRAVAR, ALTERAR, EXCLUIR e CONSULTAR). Este tipo de rotina é chamada de CRUD (CREATE, READ, UPDATE e DELETE).



Primeiro, iremos criar uma interface para definir quais métodos deverão ser implementados para construirmos o repositório de clientes.

Uma interface é composta de métodos abstratos, que depois alguma classe deverá implementar (fornecer corpo para os métodos).

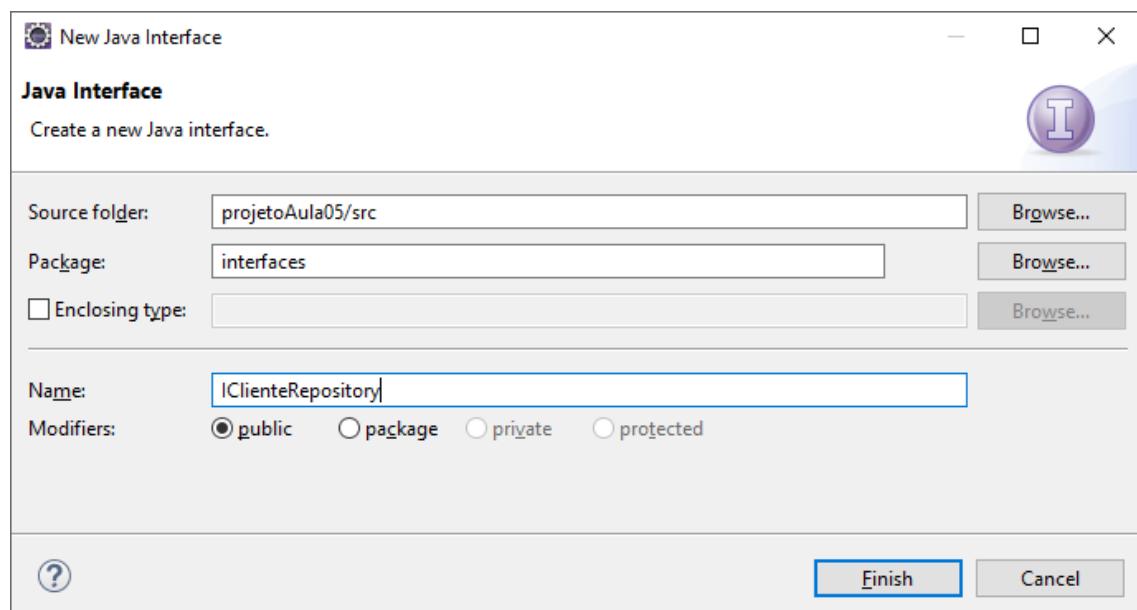
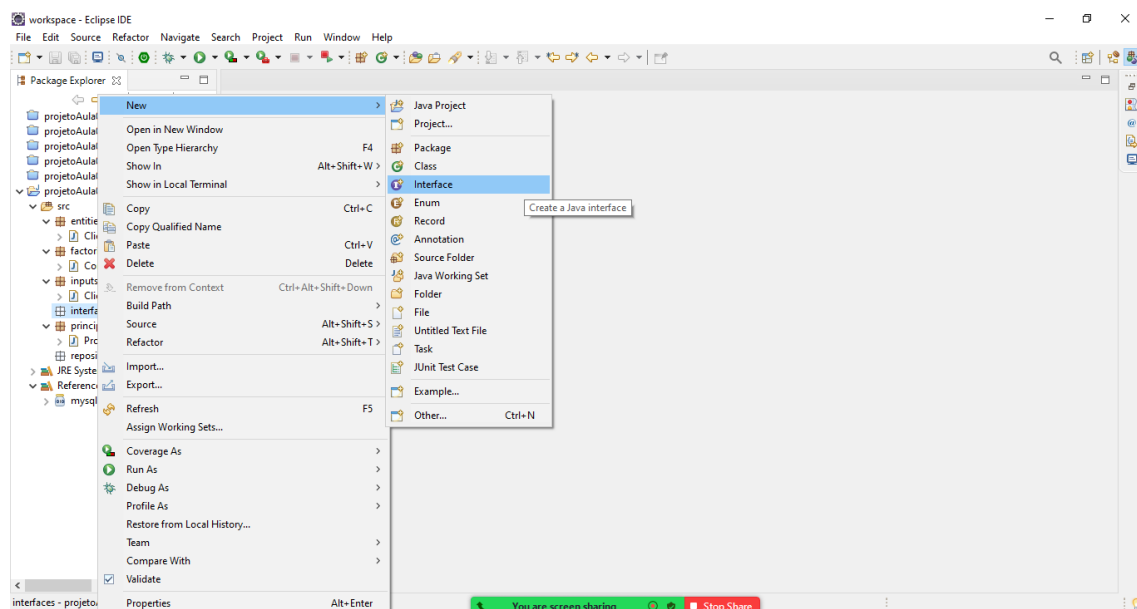
Primeiro, iremos criar a interface e a abstração dos métodos
Depois iremos criar a classe para implementar os métodos.

<<INTERFACE>>

IClienteRepository

```
+ create(cliente : Cliente) : void
+ update(cliente : Cliente) : void
+ delete(idCliente : Integer) : void
+ findAll() : List<Cliente>
```

/interfaces/**IClienteRepository.java**



```
package interfaces;

import java.util.List;

import entities.Cliente;

public interface IClienteRepository {

    // métodos abstratos (somente assinatura)
    void create(Cliente cliente) throws Exception;

    void update(Cliente cliente) throws Exception;

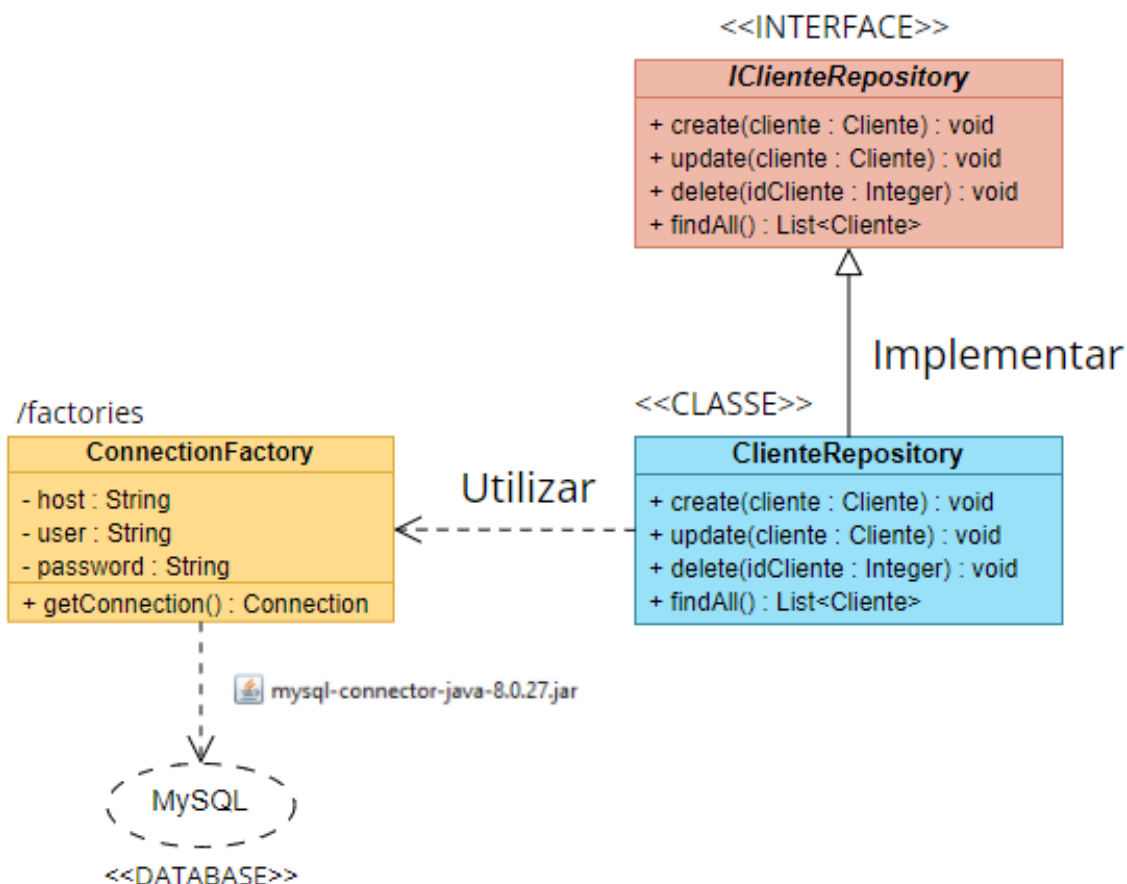
    void delete(Cliente cliente) throws Exception;

    List<Cliente> findAll() throws Exception;
}
```

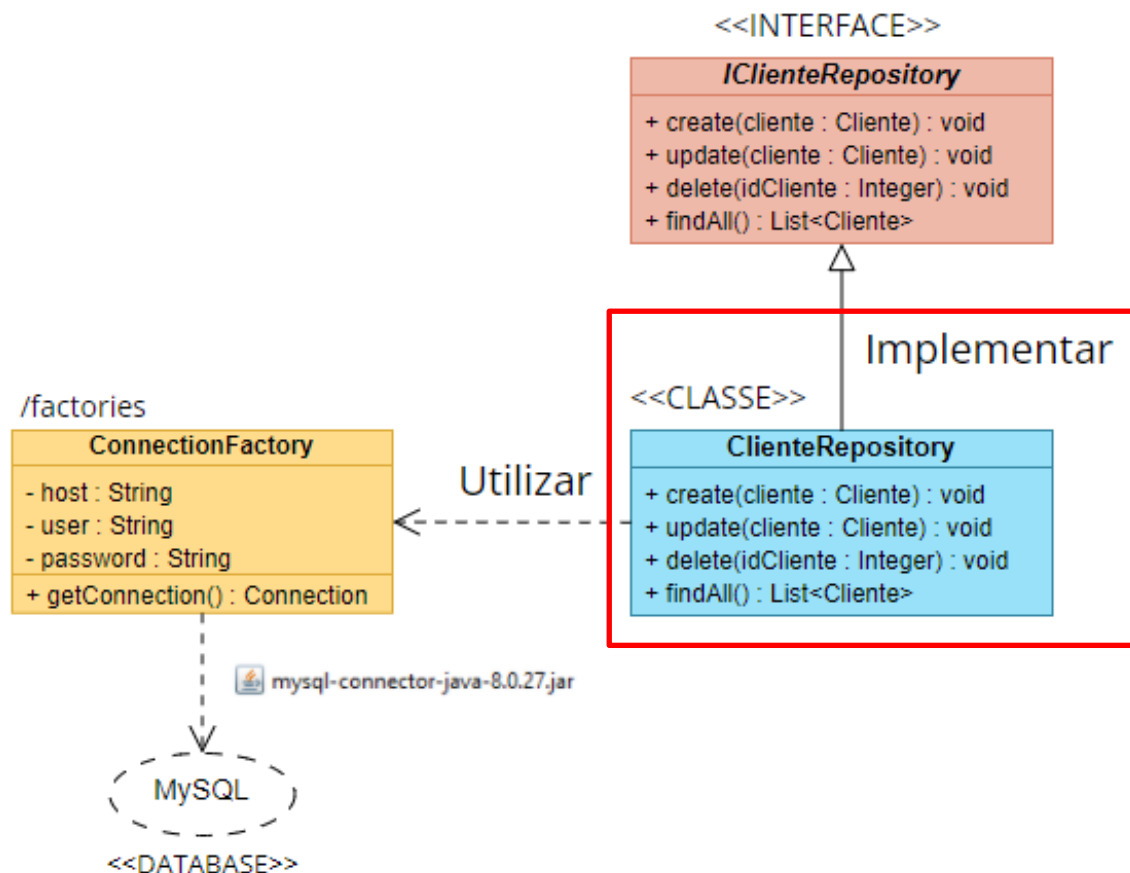
Próximo passo:

Implementar a interface

Iremos criar uma classe que forneça implementação para todos os métodos abstratos da interface.



/repositories/**ClienteRepository.java**



```

package repositories;

import java.util.List;

import entities.Cliente;
import interfaces.IClienteRepository;

public class ClienteRepository implements IClienteRepository {

    @Override
    public void create(Cliente cliente) throws Exception {
        // TODO Auto-generated method stub
    }

    @Override
    public void update(Cliente cliente) throws Exception {
        // TODO Auto-generated method stub
    }
}

```

```
@Override
public void delete(Cliente cliente) throws Exception {
    // TODO Auto-generated method stub

}

@Override
public List<Cliente> findAll() throws Exception {
    // TODO Auto-generated method stub
    return null;
}
}
```

java.sql

Pacote JAVA onde estão a maioria das classes para manipulação de banco de dados, as principais são:

Connection

Interface para armazenar a conexão aberta com o banco de dados.

PreparedStatement

Utilizado para que possamos executar comandos SQL no banco de dados, tais como INSERT, UPDATE, DELETE e SELECT.

CallableStatement

Utilizado para executar STORED PROCEDURES no banco de dados.

ResultSet

Utilizado para que possamos executar e ler resultados obtidos de consultas (SELECT) feitas no banco de dados.

Implementando o método **create** para gravar um cliente na base de dados:

```
package repositories;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.util.List;

import entities.Cliente;
import factories.ConnectionFactory;
import interfaces.IClienteRepository;

public class ClienteRepository implements IClienteRepository {
```

```
@Override
public void create(Cliente cliente) throws Exception {

    //abrindo uma conexão com o banco de dados
    Connection connection = ConnectionFactory.getConnection();

    //gravar um cliente na base de dados
    PreparedStatement statement = connection.prepareStatement
    ("INSERT INTO CLIENTE(NOME, CPF, EMAIL) VALUES(?, ?, ?)");

    statement.setString(1, cliente.getNome());
    statement.setString(2, cliente.getCpf());
    statement.setString(3, cliente.getEmail());
    statement.execute(); //executando o comando
    statement.close();

    //fechando a conexão
    connection.close();
}

@Override
public void update(Cliente cliente) throws Exception {
    // TODO Auto-generated method stub
}

@Override
public void delete(Cliente cliente) throws Exception {
    // TODO Auto-generated method stub
}

@Override
public List<Cliente> findAll() throws Exception {
    // TODO Auto-generated method stub
    return null;
}
}
```

Testando o cadastro do cliente na classe **Program.java**

package principal;

```
import entities.Cliente;
import inputs.ClienteInput;
import repositories.ClienteRepository;
```

```
public class Program {
```

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

```
        try {
```

```
            System.out.println("\nCADASTRO DE CLIENTE:\n");
```

```
            Cliente cliente = new Cliente();
```

```
            cliente.setNome(ClienteInput.lerNome());
            cliente.setEmail(ClienteInput.lerEmail());
            cliente.setCpf(ClienteInput.lerCpf());
```

```
            //cadastrando no banco de dados
            ClienteRepository clienteRepository
            = new ClienteRepository();
```

```
            clienteRepository.create(cliente);
```

```
            System.out.println
            (" \nCLIENTE CADASTRADO COM SUCESSO!");
```

```
        }
```

```
        catch (Exception e) {
```

```
            // imprimir mensagem de erro
```

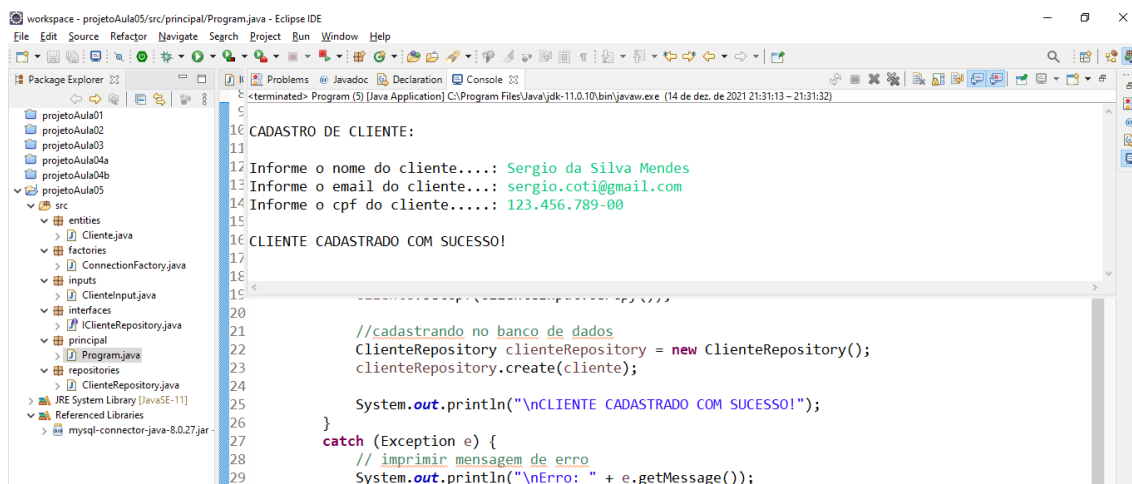
```
            System.out.println("\nErro: " + e.getMessage());
```

```
        }
```

```
    }
```

```
}
```

Testando:



```
workspace - projetoAula05/src/principal/Program.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
terminated> Program (5) [Java Application] C:\Program Files\Java\jdk-11.0.10\bin\javaw.exe (14 de dez. de 2021 21:31:13 - 21:31:32)
5
10 CADASTRO DE CLIENTE:
11
12 Informe o nome do cliente....: Sergio da Silva Mendes
13 Informe o email do cliente....: sergio.coti@gmail.com
14 Informe o cpf do cliente.....: 123.456.789-00
15
16 CLIENTE CADASTRADO COM SUCESSO!
17
18
19
20
21 //cadastrando no banco de dados
22 ClienteRepository clienteRepository = new ClienteRepository();
23 clienteRepository.create(cliente);
24
25 System.out.println("\nCLIENTE CADASTRADO COM SUCESSO!");
26
27 }
28 catch (Exception e) {
29     // imprimir mensagem de erro
30     System.out.println("\nErro: " + e.getMessage());
31 }
```

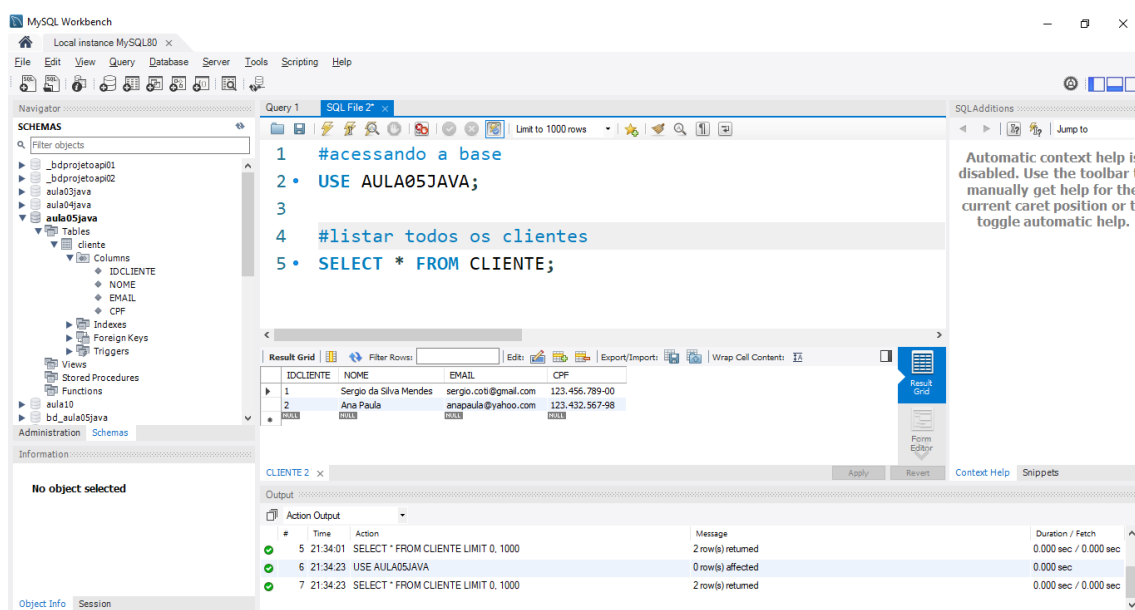
CADASTRO DE CLIENTE:

Informe o nome do cliente.....: Sergio da Silva Mendes

Informe o email do cliente....: sergio.coti@gmail.com

Informe o cpf do cliente.....: 123.456.789-00

CLIENTE CADASTRADO COM SUCESSO!



Desenvolvendo os demais métodos de UPDATE e DELETE: /repositories/ClienteRepository.java

```
package repositories;
```

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.util.List;
```

```
import entities.Cliente;
import factories.ConnectionFactory;
import interfaces.IClienteRepository;
```

```
public class ClienteRepository implements IClienteRepository {

    @Override
    public void create(Clientes cliente) throws Exception {

        //abrindo uma conexão com o banco de dados
        Connection connection = ConnectionFactory.getConnection();
```

```
//gravar um cliente na base de dados
PreparedStatement statement = connection.prepareStatement
("INSERT INTO CLIENTE(NOME, CPF, EMAIL) VALUES(?, ?, ?)");

statement.setString(1, cliente.getNome());
statement.setString(2, cliente.getCpf());
statement.setString(3, cliente.getEmail());
statement.execute(); //executando o comando
statement.close();
//fechando a conexão
connection.close();
}

@Override
public void update(Cliente cliente) throws Exception {

    Connection connection = ConnectionFactory.getConnection();

    PreparedStatement statement = connection.prepareStatement
    ("UPDATE CLIENTE SET NOME = ?, EMAIL = ?,
        CPF = ? WHERE IDCLIENTE = ?");

    statement.setString(1, cliente.getNome());
    statement.setString(2, cliente.getEmail());
    statement.setString(3, cliente.getCpf());
    statement.setInt(4, cliente.getIdCliente());
    statement.execute();
    statement.close();
    connection.close();
}

@Override
public void delete(Cliente cliente) throws Exception {

    Connection connection = ConnectionFactory.getConnection();

    PreparedStatement statement = connection.prepareStatement
    ("DELETE FROM CLIENTE WHERE IDCLIENTE = ?");

    statement.setInt(1, cliente.getIdCliente());
    statement.execute();
    statement.close();
    connection.close();
}

@Override
public List<Cliente> findAll() throws Exception {
    // TODO Auto-generated method stub
    return null;
}
}
```

Continua...