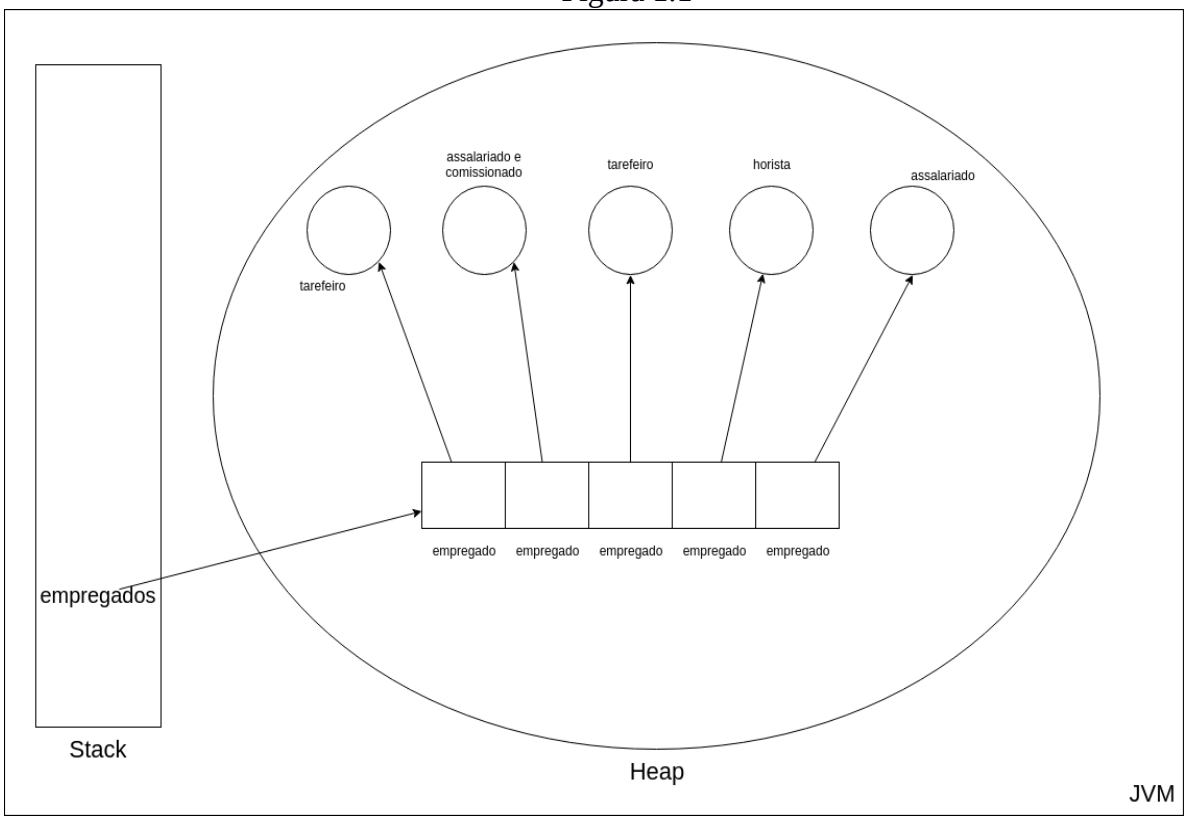
1 Introdução - Acesso a bases de dados

As aplicações que desenvolvemos até então representam os dados de interesse como objetos. Eles ficam armazenados na **memória principal** do computador, a memória RAM. Veja, por exemplo, a Figura 1.1. Ela ilustra alguns detalhes de interesse sobre a manipulação de objetos, como a existência de objetos e variáveis de referência, as memórias stack e heap entre outras coisas. Trata-se do nosso **minimundo** representado na memória gerenciada pela JVM.

Figura 1.1



Uma característica inerente à memória principal do computador é o fato de ela ser **volátil**. Isso quer dizer que os dados nela armazenados são perdidos uma vez que o computador seja desligado, ou mesmo quando a aplicação for fechada.

Ocorre que isso pode não ser suficiente para resolver muitos problemas do mundo real. Por exemplo, considere o sistema acadêmico de uma universidade. Quando um aluno se matricula, seus dados precisam ser armazenados ao longo do tempo, não podem ser perdidos quando algum computador for desligado, por exemplo. Quando o professor (cujos dados também precisam ser armazenados) cadastrar suas notas e faltas, esses dados também não podem ser perdidos. Eles precisam ser armazenados em **meio persistente**. Uma possibilidade é armazenar os dados no HD do computador, que faz parte do que conhecemos como **memória secundária.**

O armazenamento de dados em arquivos comuns no sistema de arquivos é muito utilizado por diversas aplicações. Porém, seu uso é trabalhoso e as principais operações desejadas já possuem implementações eficientes e convenientes, que podem ser simplesmente reutilizadas.

As principais operações que desejamos realizar envolvendo memória persistente são:

- Inserção de dados
- Obtenção de dados
- Atualização de dados
- Remoção de dados

Essas operações são muito comumente identificadas como operações CRUD, do inglês **C**reate, **R**ead, **U**pdate e **D**elete.

Um **Sistema Gerenciador de Banco de Dados** (SGBD) é um software que manipula a memória persistente oferecendo uma abstração que simplifica o código a ser escrito. Os arquivos são apresentados de acordo com um modelo predeterminado, que visa facilitar seu uso. Uma de suas principais características é simplificar a implementação das operações CRUD.

Neste material, iremos aprender a escrever uma aplicação Java que se comunica com um Sistema Gerenciador de Banco de Dados para fazer o armazenamento de dados em meio persistente.

2 Passo a passo

O SGBD que utilizaremos neste material é o **MySQL**, um SGBD gratuito e muito utilizado comercialmente. Ele pode ser obtido no Link 2.1.

Link 2.1
https://dev.mysql.com/downloads/installer/

Faça a instalação para **Developers**. O assistente de instalação deve oferecer essa opção. Não deixe de instalar também o MySQL Workbench, um cliente de interface gráfica para o MySQL.

2.1 (O modelo de dados) O modelo de dados utilizado pelo MySQL é o **modelo relacional**. A palavra **relacional** vem de **relação**, que é sinônimo de **tabela**. Ou seja, todos os dados que manipularemos no MySQL serão representados como tabelas. Tabelas possuem **colunas**, cada qual com o seu nome. Veja um exemplo na Figura 2.1.1. Trata-se de uma tabela capaz de armazenar dados de pessoas. Neste mini mundo, pessoas têm nome, telefone e e-mail.

Figura 2.1.1

Nome	Telefone	Email
José	12345678	jose@email.com
João	11223344	joao@email.com
Maria	44332211	maria@email.com

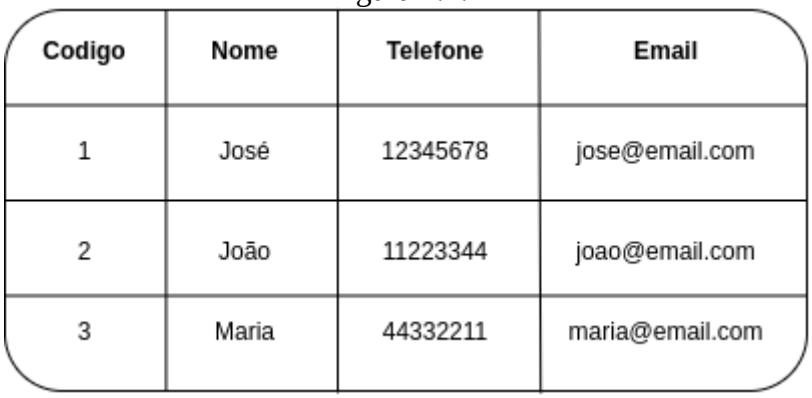
- Note que cada coluna tem um nome único. A ordem delas não importa. Cada linha contém os dados de uma pessoa específica.

- Quando utilizamos o modelo relacional, precisamos tomar o cuidado de garantir que cada linha de uma tabela possa ser diferenciada das demais. Isso é necessário pois pessoas podem ter muitos dados iguais, como nome, idade etc.

- O mecanismo utilizado para viabilizar a distinção entre linhas de uma tabela chama-se **chave primária**. Informalmente, trata-se de uma coluna cujos valores permitem diferenciar uma linha das demais. Ou seja, um determinado valor nunca ocorre em mais de uma linha.

- É comum (mas não obrigatório) adicionar uma coluna chamada **código** ou **id** às tabelas e, para cada linha, armazenar um número inteiro que tenha as características descritas. Veja o exemplo da Figura 2.1.2.

Figura 2.1.2



Codigo	Nome	Telefone	Email
1	José	12345678	jose@email.com
2	João	11223344	joao@email.com
3	Maria	44332211	maria@email.com

- O modelo relacional possui muito mais características do que as apresentadas nesta seção. O que apresentamos é suficiente para essa primeira aula de acesso a bases de dados.

2.2 (Criando uma base dados) O MySQL é um Sistema Gerenciador de Bancos de Dados, ou seja, um software capaz de gerenciar diversos bancos de dados. O primeiro passo para utilizá-lo, é criar um novo banco de dados. Faremos isso usando o MySQL Workbench.

- Abra o WorkBench e digite o comando da Listagem 2.2.1.

Listagem 2.2.1

```
CREATE DATABASE db_pessoas;
```

- A seguir, precisamos indicar que os comandos que virão, por exemplo, para a criação de tabelas, terão impacto sobre esse banco de dados recém criado. Isso é necessário porque o MySQL pode gerenciar diversos bancos de dados. Por isso, ele precisa saber em qual banco de dados executar os comandos que iremos utilizar. Veja a Listagem 2.2.2.

Listagem 2.2.2

```
USE db_pessoas;
```

2.3 (Criando uma tabela) Agora podemos criar nossa primeira tabela. Precisamos especificar um nome para a tabela, o nome de cada coluna de interesse, o tipo de cada coluna, eventuais restrições e qual coluna será a sua chave primária. Veja a Listagem 2.3.1.

Listagem 2.3.1

```
CREATE TABLE tb_pessoa(
        codigo INT PRIMARY KEY AUTO_INCREMENT,
        nome VARCHAR(200),
        fone VARCHAR(200),
        email VARCHAR(200)
);
```

2.4 (Inserção de dados) A seguir, podemos fazer a inserção de alguns dados na tabela recém criada. Veja a Listagem 2.3.2.

Listagem 2.3.2

```
INSERT INTO tb_pessoa (nome, fone, email) VALUES ('Jose', '11223344',
'jose@email.com');
INSERT INTO tb_pessoa (nome, fone, email) VALUES ('Joao', '22334455',
'joao@email.com');
INSERT INTO tb_pessoa (nome, fone, email) VALUES ('Maria', '00117788',
'maria@email.com');
```

2.5 (Obtenção de dados) Podemos verificar os dados armazenados na tabela com o comando SELECT, com na Listagem 2.5.1.

Listagem 2.5.1

```
SELECT * FROM tb_pessoa;
```

2.6 (Atualização de dados) Podemos alterar os dados existentes em uma tabela, eventualmente especificando um critério para que a atualização tenha impacto somente sobre uma linha. Veja o exemplo da Listagem 2.6.1.

Listagem 2.6.1

```
UPDATE tb_pessoa SET nome='José da Silva' WHERE codigo = 1;
```

2.7 (Apagando dados) Para apagar dados de uma tabela, usamos o comando DELETE. Veja a Listagem 2.7.1.

Listagem 2.7.1

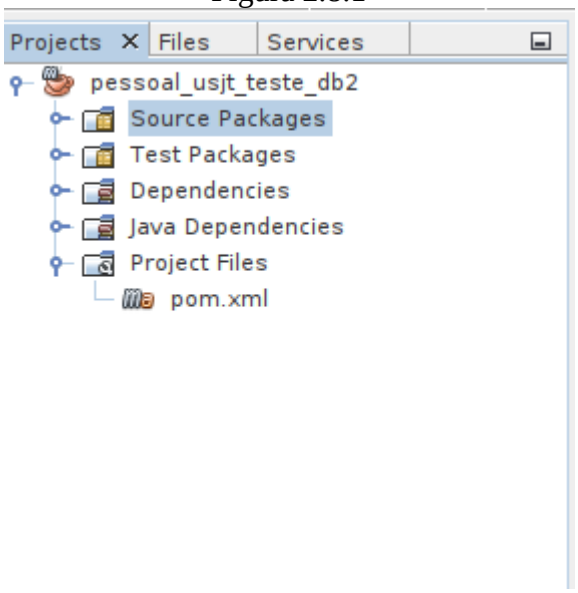
```
DELETE FROM tb_pessoa WHERE codigo=3;
```

2.8 (Driver do MySQL para JDBC) A API que utilizaremos para acessar o MySQL usando a linguagem Java chama-se JDBC. A comunicação entre a aplicação Java e o SGBD MySQL é implementada por uma coleção de classes que é disponibilizada pelo fabricante do MySQL. Essa coleção de classes chama-se **Driver**. Precisamos fazer o download do driver do MySQL e adicioná-lo ao classpath do projeto, ou seja, a variável utilizada pelo compilador e pela JVM para decidir quais classes considerar no momento de compilação e interpretação, respectivamente. O Driver do MySQL para JDBC pode ser obtido manualmente no Link 2.8.1.

Link 2.8.1
https://dev.mysql.com/downloads/connector/j/

- No Netbeans, temos criado projetos gerenciados pelo Maven que, entre outras coisas, é capaz de fazer o download de dependências e ajustá-las adequadamente. Para fazer o download do driver do MySQL usando o Maven, expanda o seu projeto e encontre a opção "**Project Files**", como na Figura 2.8.1.

Figura 2.8.1



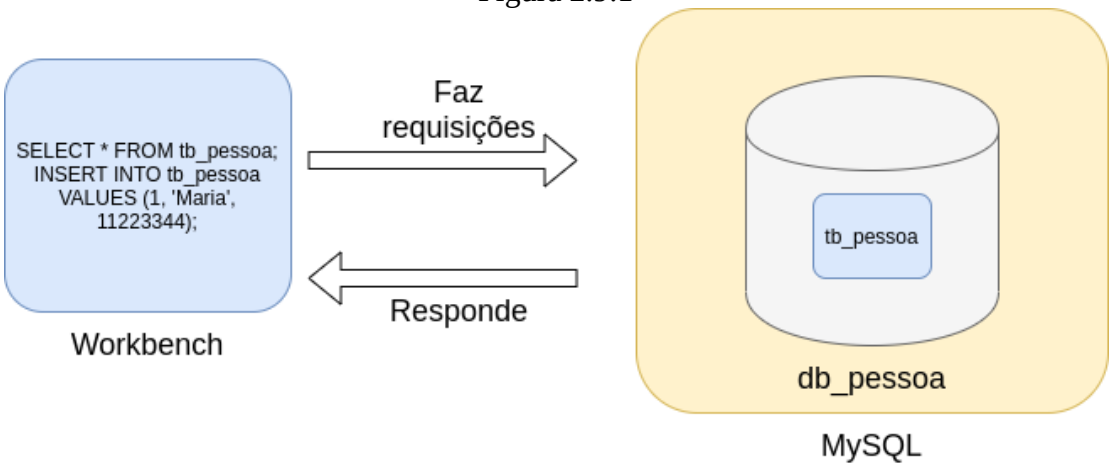
- Note que há um arquivo chamado **pom.xml.** É ali que especificaremos nossas dependências. Abra o arquivo e adicione o conteúdo da Listagem 2.8.2.

Listagem 2.8.2

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>br.com.bossini</groupId>
   <artifactId>pessoal_teste_db2</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <maven.compiler.source>11</maven.compiler.source>
      <maven.compiler.target>11</maven.compiler.target>
   </properties>
   <dependencies>
      <dependency>
         <groupId>mysql</groupId>
         <artifactId>mysql-connector-java</artifactId>
         <version>8.0.19</version>
      </dependency>
   </dependencies>
</project>
```

2.9 (Arquitetura cliente/servidor) Quando fizemos uso do MySQL por meio do Workbench, fizemos uso da arquitetura conhecida como cliente servidor. A ideia é muito simples. De um lado, temos um software em execução esperamos por requisições a serem feitas (o MySQL), de outro lado, temos um software em execução que, de tempos em tempos, pode realizar requisições a um servidor (o Workbench). Veja a Figura 2.9.1.

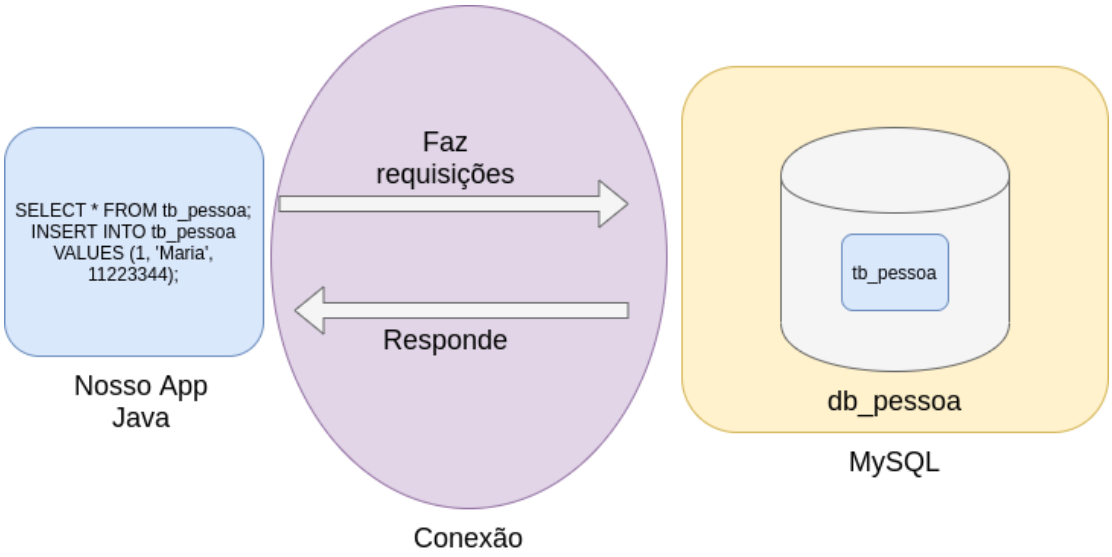
Figura 2.9.1



2.10 (Conexão entre a aplicação Java e o servidor MySQL) Entender a arquitetura cliente/servidor é fundamental pois a aplicação Java que implementaremos desempenhará o papel de cliente. Ela enviará comandos ao MySQL que, por sua vez, os executará e devolverá respostas de acordo.

- O primeiro passo é obter uma **conexão** com o banco. Ou seja, um canal de comunicação que permita o envio de comandos e o recebimento de respostas. Veja a Figura 2.10.1.

Figura 2.10.1



- Segundo o princípio conhecido como **alta coesão**, é interessante criarmos uma classe cuja única razão de ser seja a manipulação de conexões com o MySQL. Sua implementação é dada na Listagem 2.10.1.

Listagem 2.10.1

```
import java.sql.Connection;
import java.sql.DriverManager;
public class ConnectionFactory {

    private String usuario = "root";
    private String senha = "1234";
    private String host = "localhost";
    private String porta = "3306";
    private String bd = "db_pessoas";

    public Connection obtemConexao (){
        try{
            Connection c = DriverManager.getConnection(
                "jdbc:mysql://" + host + ":" + porta + "/" + bd,
                usuario,
                senha
            );
            return c;
        }
        catch (Exception e){
            e.printStackTrace();
            return null;
```

```
        }
    }
}
```

2.11 (A classe Pessoa) Agora iremos definir uma classe para representar pessoas. Ela terá um campo apropriado para cada coluna existente na base, pois nosso objetivo é armazenar objetos do tipo pessoa na base. Além disso, caberá à classe Pessoa definir as operações de acesso à base. Embora estejamos violando o princípio da alta coesão, isso simplifica a solução e permite que tenhamos foco na parte técnica. Futuramente aplicaremos um procedimento chamado **refatoração** ao código, respeitando as boas práticas e obtendo código mais facilmente reutilizável e de mais fácil manutenção. Veja a Listagem 2.11.1.

Listagem 2.11.1

```
public class Pessoa {
    private int codigo;
    private String nome;
    private String fone;
    private String email;

    //getters/setters
}
```

2.12 (Menu para operações na base) Vamos implementar uma classe que oferece o seguinte menu ao usuário:

1- Cadastrar pessoa
2- Atualizar pessoa
3- Apagar pessoa
4- Listar pessoas
0- Sair

- A classe com o menu principal é exibida na Listagem 2.12.1.

Listagem 2.12.1

```
public class Principal {
    public static void main(String[] args) {
        String menu = "1-Cadastrar\n2-Atualizar\n3-Apagar\n4-Listar\n0-Sair";
        int op;
        do{
            op = Integer.parseInt(JOptionPane.showInputDialog(menu));
            switch (op){
                case 1:
```

```
                break;
            case 2:
                break;
            case 3:
                break;
            case 4:
                break;
            case 0:
                break;
            default:
                JOptionPane.showMessageDialog(null, "Opção inválida");
        }

    }while (op != 0);
  }
}
```

- Na Listagem 2.12.2 fazemos o cadastro de uma Pessoa. O método faz parte da classe Pessoa.

Listagem 2.12.2

```
public void inserir (){
    //1: Definir o comando SQL
    String sql = "INSERT INTO tb_pessoa(nome, fone, email) VALUES (?, ?, ?)";
    //2: Abrir uma conexão
    ConnectionFactory factory = new ConnectionFactory();
    try (Connection c = factory.obtemConexao()){
        //3: Pré compila o comando
        PreparedStatement ps = c.prepareStatement(sql);
        //4: Preenche os dados faltantes
        ps.setString(1, nome);
        ps.setString(2, fone);
        ps.setString(3, email);
        //5: Executa o comando
        ps.execute();
    }
    catch (Exception e){
        e.printStackTrace();
    }
  }
```

- O comando já pode ser usado no menu, como mostra a Listagem 2.12.3.

Listagem 2.12.3

```
case 1:{
                String nome = JOptionPane.showInputDialog("Nome?");
                String fone = JOptionPane.showInputDialog("Fone?");
                String email = JOptionPane.showInputDialog("Email?");
```

```
                Pessoa p = new Pessoa ();
                p.setNome(nome);
                p.setFone(fone);
                p.setEmail(email);
                p.inserir();
                break;
}
```

- As estruturas dos métodos de acesso à base são sempre muito semelhantes. A Listagem 2.12.4 mostra como atualizar os dados de uma pessoa cujo código é informado pelo usuário. O método pertence à classe Pessoa.

Listagem 2.12.4

```
public void atualizar (){
      //1: Definir o comando SQL
      String sql = "UPDATE tb_pessoa SET nome = ?, fone = ?, email = ? WHERE
codigo = ?";
      //2: Abrir uma conexão
      ConnectionFactory factory = new ConnectionFactory();
      try (Connection c = factory.obtemConexao()){
         //3: Pré compila o comando
         PreparedStatement ps = c.prepareStatement(sql);
         //4: Preenche os dados faltantes
         ps.setString(1, nome);
         ps.setString(2, fone);
         ps.setString(3, email);
         ps.setInt(4, codigo);
         //5: Executa o comando
         ps.execute();

      }
      catch (Exception e){
         e.printStackTrace();
      }
   }
```

- A seguir, adicione o método ao menu, como na Listagem 2.12.5.

Listagem 2.12.5

```
case 2:{
                String nome = JOptionPane.showInputDialog("Nome?");
                String fone = JOptionPane.showInputDialog("Fone?");
                String email = JOptionPane.showInputDialog("Email?");
                int codigo =
Integer.parseInt(JOptionPane.showInputDialog("Codigo?"));
                Pessoa p = new Pessoa ();
                p.setNome(nome);
                p.setFone(fone);
```

```
                p.setEmail(email);
                p.setCodigo(codigo);
                p.atualizar();
                break;
}
```

- O método que permite a remoção de pessoas é exibido na Listagem 2.12.6. Ele pertence à classe Pessoa.

Listagem 2.12.6

```
public void apagar (){
      //1: Definir o comando SQL
      String sql = "DELETE FROM tb_pessoa WHERE codigo = ?";
      //2: Abrir uma conexão
      ConnectionFactory factory = new ConnectionFactory();
      try (Connection c = factory.obtemConexao()){
         //3: Pré compila o comando
         PreparedStatement ps = c.prepareStatement(sql);
         //4: Preenche os dados faltantes
         ps.setInt(1, codigo);
         //5: Executa o comando
         ps.execute();
      }
         catch (Exception e){
         e.printStackTrace();
      }
}
```

- Ajuste o menu principal para utilizá-lo, como na Listagem 2.12.7.

Listagem 2.12.7

```
case 3:{
                int codigo =
                      Integer.parseInt(JOptionPane.showInputDialog("Codigo?"));
                Pessoa p = new Pessoa ();
                p.setCodigo(codigo);
                p.apagar();
                break;
             }
```

- O método para listar todos os clientes é semelhante. Porém, ele faz uso de um objeto do tipo **ResultSet** para representar os dados trazidos da base. Veja a Listagem 2.12.8. Note que ela faz uso da classe JoptionPane, o que certamente não é ideal. Como mencionado, o código deverá passar por um processo de **refatoração**.

Listagem 2.12.8

```
public void listar (){
      //1: Definir o comando SQL
      String sql = "SELECT * FROM tb_pessoa";
      //2: Abrir uma conexão
      ConnectionFactory factory = new ConnectionFactory();
      try (Connection c = factory.obtemConexao()){
         //3: Pré compila o comando
         PreparedStatement ps = c.prepareStatement(sql);
         //4: Executa o comando e guarda
         //o resultado em um ResultSet
         ResultSet rs = ps.executeQuery();
         //5: itera sobre o resultado
         while (rs.next()){
            int codigo = rs.getInt("codigo");
            String nome = rs.getString("nome");
            String fone = rs.getString("fone");
            String email = rs.getString("email");
            String aux = String.format(
               "Código: %d, Nome: %s, Fone: %s, Email: %s",
               codigo,
               nome,
               fone,
               email
            );
            JOptionPane.showMessageDialog (null, aux);

         }
      }
         catch (Exception e){
         e.printStackTrace();
      }
   }
```

- Encaixe o método no menu principal, como exibe a Listagem 2.12.9.

Listagem 2.12.9

```
case 4:{
        Pessoa p = new Pessoa();
        p.listar();
        break;
}
```

Referências

DEITEL, P. e DEITEL, H. **Java Como Programar**. 8ª Edição. São Paulo, SP: Pearson, 2010.

LOPES, A. e GARCIA, G. **Introdução à Programação – 500 Algoritmos Resolvidos**. 1ª Edição. São Paulo, SP: Elsevier, 2002.