

Laboratório de desenvolvimento de software

Banco de dados + Java + Sockets

Sockets

- Sockets são uma abstração que permite a comunicação entre processos em diferentes dispositivos através de uma rede.
- Eles fornecem uma maneira de estabelecer conexões e enviar dados entre computadores, permitindo a construção de aplicações distribuídas.
- Em Java, a API de Sockets faz parte do pacote `java.net` e é usada para criar aplicações cliente-servidor robustas e escaláveis.

Sockets

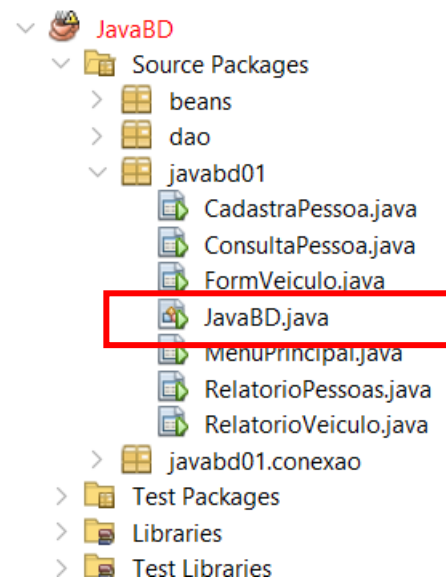
- Existem dois tipos principais de sockets em Java:
- Socket Cliente (Socket):
 - Um socket cliente é usado pelo lado que inicia a comunicação.
 - Ele se conecta a um servidor esperando por conexões em um endereço específico e porta.
- Socket Servidor (ServerSocket):
 - Um socket servidor espera por conexões de clientes.
 - Uma vez que um cliente se conecta, o servidor cria um novo socket dedicado para lidar com essa conexão.

Sockets

- Vamos seguir no nosso projeto de aula, com as tabelas Pessoas e Veículos
 - Abra, configure o projeto e seguiremos e iniciaremos configurando o servidor do sockets

Servidor

- O nosso servidor, nós iremos implementar sem interface gráfica, portanto, podemos usar a nossa classe principal criada com o projeto.
- No caso do meu projeto, é chamado de JavaBD



Servidor

```
public static void main(String[] args) {
    int porta = 12345; // Use uma constante para a porta

    try (ServerSocket servidorSocket = new ServerSocket(porta)) {
        System.out.println("Servidor aguardando conexões na porta " + porta);

        while (true) {
            try {
                Socket clienteSocket = servidorSocket.accept();
                System.out.println("Conexão aceita de " + clienteSocket.getInetAddress());
                ObjectOutputStream out = new ObjectOutputStream(out: clienteSocket.getOutputStream());
                ObjectInputStream in = new ObjectInputStream(in: clienteSocket.getInputStream());

                int id = in.readInt();
                System.out.println("ID recebido: " + id);

                // Simule a obtenção de uma Pessoa a partir do ID
                PessoaDAO pdao = new PessoaDAO();
                Pessoa p = pdao.getPessoa(id);

                out.writeObject(obj: p);

            } catch (IOException ex) {
                System.out.println(x: "Erro ao aceitar conexão do cliente");
            }
        }
    } catch (IOException ex) {
        System.out.println(x: "Erro ao criar o ServerSocket");
    }
}
```

```
public static void main(String[] args) {  
    int porta = 12345; // Use uma constante para a porta  
  
    try (ServerSocket servidorSocket = new ServerSocket(porta)) {  
        System.out.println("Servidor aguardando conexões na porta " + porta);  
  
        while (true) {  
            try {  
                Socket clienteSocket = servidorSocket.accept();  
                System.out.println("Conexão aceita de " + clienteSocket.getInetAddress());  
                ObjectOutputStream out = new ObjectOutputStream(out: clienteSocket.getOutputStream());  
                ObjectInputStream in = new ObjectInputStream(in: clienteSocket.getInputStream());  
  
                int id = in.readInt();  
                System.out.println("ID recebido: " + id);  
  
                // Simule a obtenção de uma Pessoa a partir do ID  
                PessoaDAO pdao = new PessoaDAO();  
                Pessoa p = pdao.getPessoa(id);  
  
                out.writeObject(obj:p);  
  
            } catch (IOException ex) {  
                System.out.println(x: "Erro ao aceitar conexão do cliente");  
            }  
        }  
    } catch (IOException ex) {  
        System.out.println(x: "Erro ao criar o ServerSocket");  
    }  
}
```

Cliente

- Crie um projeto para o cliente:

New Java Application

Steps

1. Choose Project
2. **Name and Location**

Name and Location

Project Name:

Project Location:

Project Folder:

☐ Use Dedicated Folder for Storing Libraries

Libraries Folder:

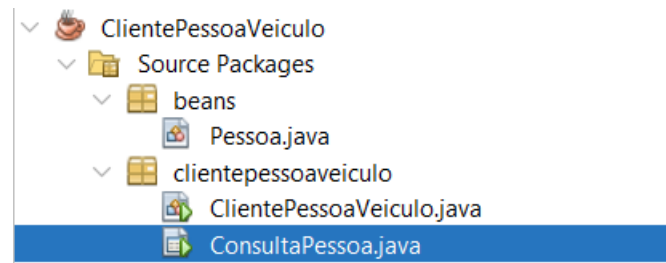
Different users and projects can share the same compilation libraries (see Help for details).

☒ Create Main Class

< Back Next > **Finish** Cancel Help

Cliente

- No cliente, precisaremos criar uma classe Pessoa, para ele ter conhecimento da estrutura.
- Seguiremos o mesmo padrão, crie um pacote chamado beans e copie e cole a classe Pessoa para este projeto, dentro desta classe:



Cliente

- Na classe Pessoa, agora, precisamos informar que ela será serializável:

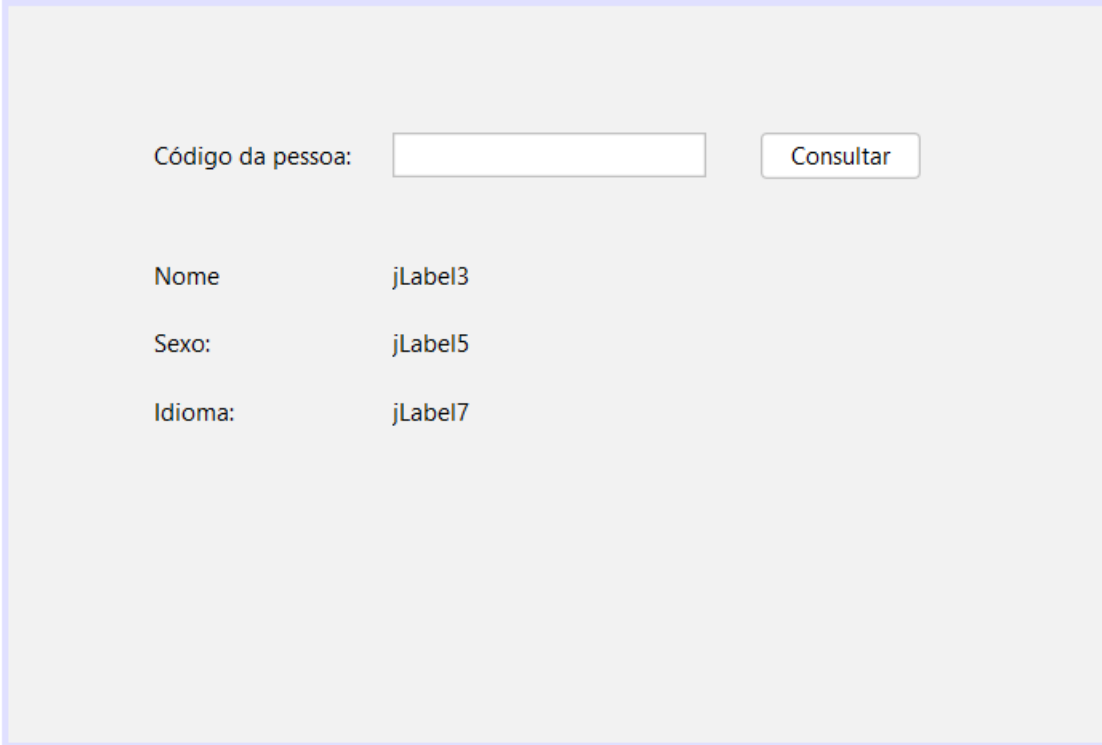
```
import java.io.Serializable;

/**
 *
 * @author ricar
 */
public class Pessoa implements Serializable {
    private int id;
    private String nome;
```

- Faça isso, no outro projeto também...

Cliente

- Agora, vamos montar a interface para a consulta:



Código da pessoa:

Nome jLabel3

Sexo: jLabel5

Idioma: jLabel7

Cliente

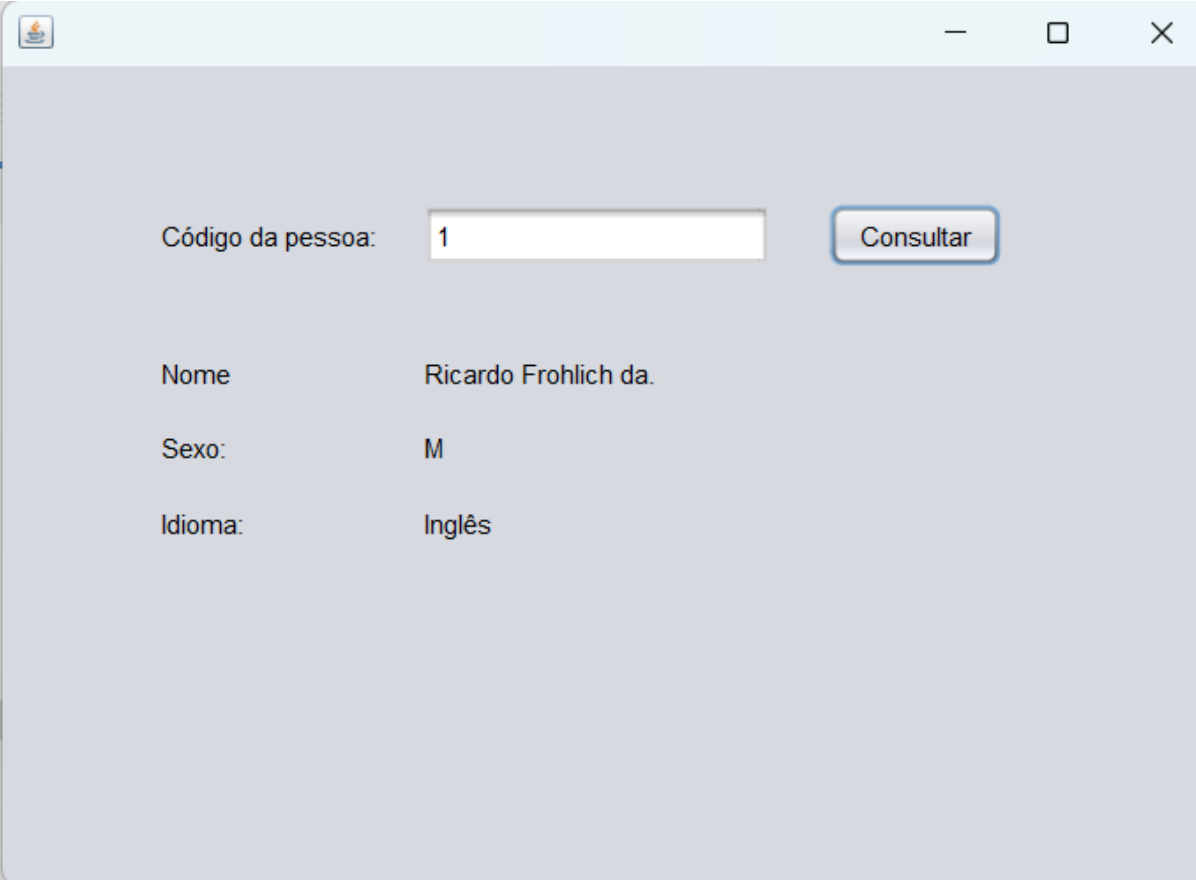
- E no botão consultar, adicione o seguinte código:

```
private void btnConsultarActionPerformed(java.awt.event.ActionEvent evt) {  
    try {  
        String host = "localhost";  
        int porta = 12345;  
  
        Socket clienteSocket = new Socket(host, porta);  
        System.out.println(x: "Conexão efetuada com sucesso!");  
  
        ObjectOutputStream out = new ObjectOutputStream(out: clienteSocket.getOutputStream());  
        ObjectInputStream in = new ObjectInputStream(in: clienteSocket.getInputStream());  
        int id = Integer.parseInt(s: txtIDPessoa.getText());  
        out.writeInt(val: id);  
        out.flush();  
        Pessoa p = (Pessoa) in.readObject();  
        System.out.println("Resposta: " + p.getNome());  
        lblNome.setText(text: p.getNome());  
        lblSexo.setText(text: p.getSexo());  
        lblIdioma.setText(text: p.getIdioma());  
        // Fechamento adequado dos fluxos e do socket  
        out.close();  
        in.close();  
        clienteSocket.close();  
    } catch (IOException ex) {  
        System.out.println(x: "Erro na comunicação com o servidor");  
    } catch (ClassNotFoundException ex) {  
        System.out.println(x: "Classe Pessoa não encontrada");  
    }  
}
```

```
private void btnConsultarActionPerformed(java.awt.event.ActionEvent evt) {  
    try {  
        String host = "localhost";  
        int porta = 12345;  
  
        Socket clienteSocket = new Socket(host, port: porta);  
        System.out.println(x: "Conexão efetuada com sucesso!");  
  
        ObjectOutputStream out = new ObjectOutputStream(out: clienteSocket.getOutputStream());  
        ObjectInputStream in = new ObjectInputStream(in: clienteSocket.getInputStream());  
        int id = Integer.parseInt(s: txtIDPessoa.getText());  
        out.writeInt(val: id);  
        out.flush();  
        Pessoa p = (Pessoa) in.readObject();  
        System.out.println("Resposta: " + p.getNome());  
        lblNome.setText(text: p.getNome());  
        lblSexo.setText(text: p.getSexo());  
        lblIdioma.setText(text: p.getIdioma());  
        // Fechamento adequado dos fluxos e do socket  
        out.close();  
        in.close();  
        clienteSocket.close();  
    } catch (IOException ex) {  
        System.out.println(x: "Erro na comunicação com o servidor");  
    } catch (ClassNotFoundException ex) {  
        System.out.println(x: "Classe Pessoa não encontrada");  
    }  
}
```

Cliente

- Resultado:



A screenshot of a web application window with a light blue title bar and standard window controls (minimize, maximize, close). The main content area is light gray and displays search results. At the top, there is a label 'Código da pessoa:' followed by a text input field containing the number '1' and a blue 'Consultar' button. Below this, the search results are displayed in a list format:

Nome	Ricardo Frohlich da.
Sexo:	M
Idioma:	Inglês

Servidor

- E se tentarmos conectar mais de um cliente?
 - Acontece que ele fica preso no laço infinito, para isso, precisamos usar threads, que vocês verão melhor em Sistemas Distribuídos

Threads

- Uma thread é uma unidade de execução leve que permite que um programa execute tarefas concorrentemente.
- Ela é uma forma de tornar uma aplicação capaz de realizar múltiplas operações simultaneamente, aumentando assim a eficiência e melhorando a capacidade de resposta.

Threads

- Existem dois tipos principais de threads em Java:
- Thread Principal (ou "main"):
 - A thread principal é a thread principal que é iniciada quando um programa Java é executado.
 - Ela é responsável por iniciar outras threads e pode executar operações simultaneamente com essas threads secundárias.
- Threads Secundárias:
 - Threads secundárias são criadas para executar tarefas específicas em paralelo com a thread principal.
 - Elas são úteis para realizar operações que não bloqueiam a execução do programa principal.

Threads

- É possível criar threads de duas maneiras principais: implementando a interface Runnable ou estendendo a classe Thread.
- Ambas as abordagens permitem que você execute código de forma concorrente, mas há diferenças na flexibilidade e no design entre elas.

Threads

```
class MinhaThread extends Thread {  
    private String mensagem;  
    private int intervalo;  
  
    public MinhaThread(String mensagem, int intervalo) {  
        this.mensagem = mensagem;  
        this.intervalo = intervalo;  
    }  
  
    @Override  
    public void run() {  
        try {  
            while (true) {  
                System.out.println(x: mensagem);  
                Thread.sleep(millis:intervalo);  
            }  
        } catch (InterruptedException e) {  
            // Lidar com a interrupção, se necessário  
        }  
    }  
}
```

Threads

```
public class ExemploThreads {  
    public static void main(String[] args) {  
        // Criar e iniciar threads secundárias  
        MinhaThread thread1 = new MinhaThread(mensagem: "Thread 1 - Mensagem a cada 1 segundo", intervalo:1000);  
        MinhaThread thread2 = new MinhaThread(mensagem: "Thread 2 - Mensagem a cada 2 segundos", intervalo:2000);  
  
        thread1.start();  
        thread2.start();  
  
        // Executar operações na thread principal  
        for (int i = 0; i < 5; i++) {  
            System.out.println("Thread Principal - Iteração " + i);  
            try {  
                Thread.sleep(millis:1500);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
  
        // Interromper as threads secundárias (opcional)  
        thread1.interrupt();  
        thread2.interrupt();  
    }  
}
```

Servidor

- Agora, vamos implementar no servidor uma thread para que cada cliente que seja conectado no servidor, seja executado em paralelo.
- Ou seja, o servidor poderá tratar conexões simultâneas.

Servidor

```
class MinhaThread extends Thread {
    private String mensagem;
    private int intervalo;

    public MinhaThread(String mensagem, int intervalo) {
        this.mensagem = mensagem;
        this.intervalo = intervalo;
    }

    @Override
    public void run() {
        try {
            while (true) {
                System.out.println(x: mensagem);
                Thread.sleep(millis: intervalo);
            }
        } catch (InterruptedException e) {
            System.out.println(x: "Thread interrompida!");
        }
    }
}
```

```
public class ThreadServer extends Thread {
    private Socket clienteSocket;

    public ThreadServer(Socket clienteSocket) {
        this.clienteSocket = clienteSocket;
    }

    @Override
    public void run() {
        try (ObjectOutputStream out = new ObjectOutputStream(out: clienteSocket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(in: clienteSocket.getInputStream())) {

            int id = in.readInt();
            System.out.println("ID recebido: " + id);

            // Simule a obtenção de uma Pessoa a partir do ID
            PessoaDAO pdao = new PessoaDAO();
            Pessoa p = pdao.getPessoa(id);

            out.writeObject(obj:p);

        } catch (IOException ex) {
            System.out.println(x: "Erro ao lidar com o cliente");
        }
    }
}
```

```
public class ThreadServer extends Thread {  
    private Socket clienteSocket;  
  
    public ThreadServer(Socket clienteSocket) {  
        this.clienteSocket = clienteSocket;  
    }  
  
    @Override  
    public void run() {  
        try {  
            ObjectOutputStream out = new ObjectOutputStream(out: clienteSocket.getOutputStream());  
            ObjectInputStream in = new ObjectInputStream(in: clienteSocket.getInputStream());  
  
            int id = in.readInt();  
            System.out.println("ID recebido: " + id);  
  
            // Simule a obtenção de uma Pessoa a partir do ID  
            PessoaDAO pdao = new PessoaDAO();  
            Pessoa p = pdao.getPessoa(id);  
  
            out.writeObject(obj:p);  
  
        } catch (IOException ex) {  
            System.out.println(x: "Erro ao lidar com o cliente");  
        }  
    }  
}
```


Servidor

```
public static void main(String[] args) {
    int porta = 12345; // Use uma constante para a porta

    try (ServerSocket servidorSocket = new ServerSocket(porta)) {
        System.out.println("Servidor aguardando conexões na porta " + porta);

        while (true) {
            try {
                Socket clienteSocket = servidorSocket.accept();
                System.out.println("Conexão aceita de " + clienteSocket.getInetAddress());

                // Criar uma nova thread para lidar com o cliente
                Thread threadCliente = new ThreadServer(clienteSocket);
                threadCliente.start();

            } catch (IOException ex) {
                System.out.println(x: "Erro ao aceitar conexão do cliente");
            }
        }
    } catch (IOException ex) {
        System.out.println(x: "Erro ao criar o ServerSocket");
    }
}
```

Atividade

- Desenvolver uma aplicação cliente x servidor para efetuar o cadastro de uma Pessoa
- Na aplicação cliente deve ser obtido todos os dados em uma interface gráfica e enviada para o servidor.
- No servidor, a classe pessoa deve ser inserida no banco de dados e deve retornar ao cliente uma mensagem de sucesso se foi inserido com sucesso ou de erro se ocorreu alguma exceção.