



UNIVERSIDADE
Estácio de Sá

Universidade	Estácio de Sá
Campus	Polo de Cobilândia / Vila – Velha/ES
Nome do Curso	Desenvolvimento Full Stack
Nome da Disciplina	RPG0018 - Por que não paralelizar
Turma	9001
Semestre	Primeiro Semestre de 2024
Integrantes do Grupo	Tiago de Jesus Pereira Furtado
Matrícula	202306189045

**VILA VELHA
2024**

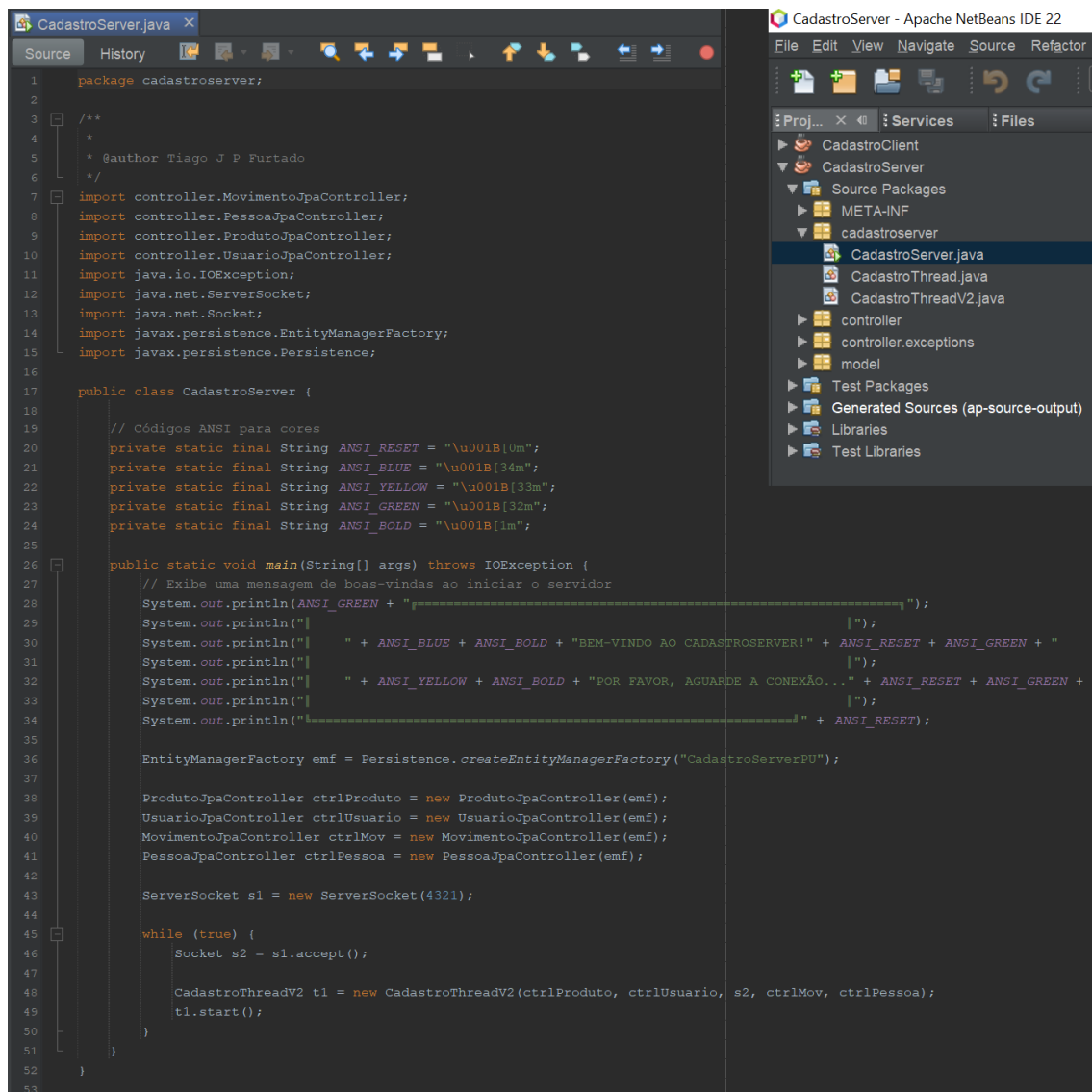
Servidores e clientes baseados em Socket, com uso de Threads tanto no lado cliente quanto no lado servidor, acessando o banco de dados via JPA.

👉 1º Procedimento | Criando o Servidor e Cliente de Teste

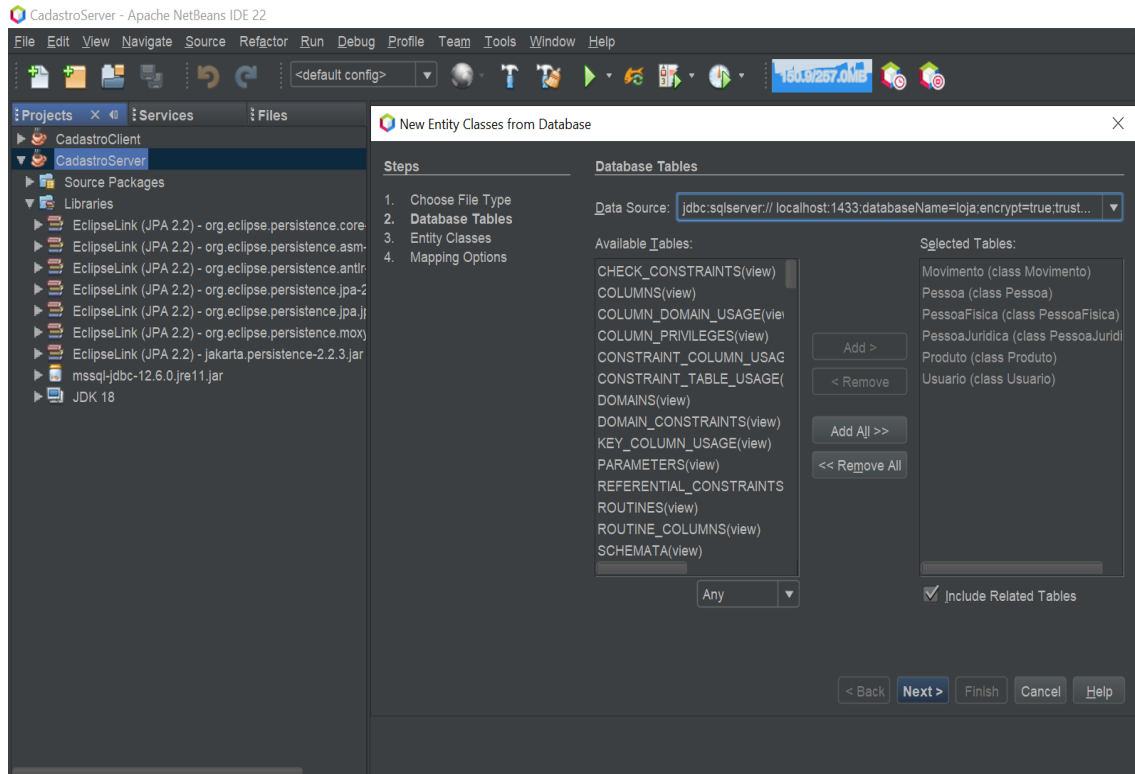
1- Objetivo da Prática:

- **Criar servidores Java com base em Sockets.**
- **Criar clientes síncronos para servidores com base em Sockets.**
- **Criar clientes assíncronos para servidores com base em Sockets.**
- **Utilizar Threads para implementação de processos paralelos.**
- **No final do exercício, o aluno terá criado um servidor Java baseado em Socket, com acesso ao banco de dados via JPA, além de utilizar os recursos nativos do Java para implementação de clientes síncronos e assíncronos. As Threads serão usadas tanto no servidor, para viabilizar múltiplos clientes paralelos, quanto no cliente, para implementar a resposta assíncrona.**





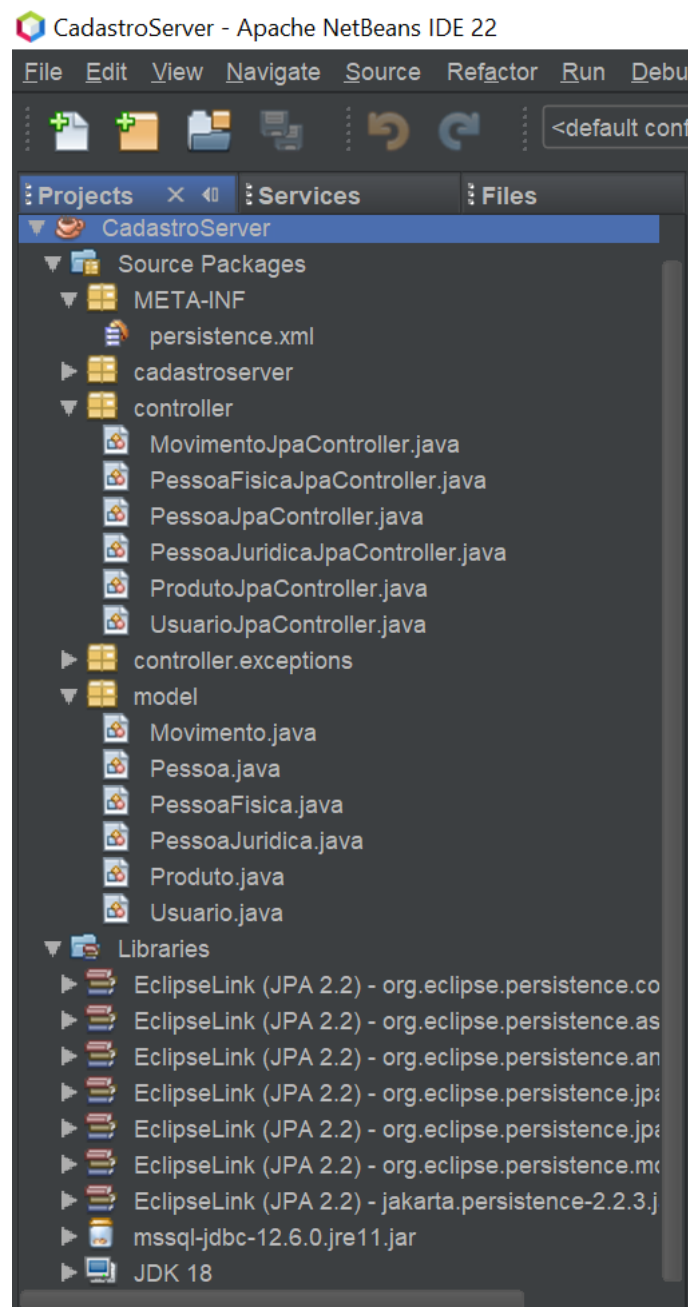
2. Criar a camada de persistência em CadastroServer.
- ✓ Criar o pacote model para implementação das entidades.
 - ✓ Utilizar a opção New...Entity Classes from Database.



- ✓ Selecionar a conexão com o SQL Server, previamente configurada na aba **Services**, divisão **Databases**, do NetBeans e adicionar todas as tabelas.
- ✓ Acrescentar ao projeto a biblioteca **Eclipse Link (JPA 2.2)**.
- ✓ Acrescentar o arquivo **jar** do conector JDBC para o **SQL Server**.

3. Criar a camada de controle em CadastroServer:

- ✓ Criar o pacote controller para implementação dos controladores.
- ✓ Utilizar a opção New...JPA Controller Classes from Entity Classes.
- ✓ Na classe UsuarioJpaController, adicionar o método findUsuario, tendo como parâmetros o login e a senha, retornando o usuário a partir de uma consulta JPA, ou nulo, caso não haja um usuário com as credenciais.
- ✓ Ao final o projeto ficará como o que é apresentado a seguir.



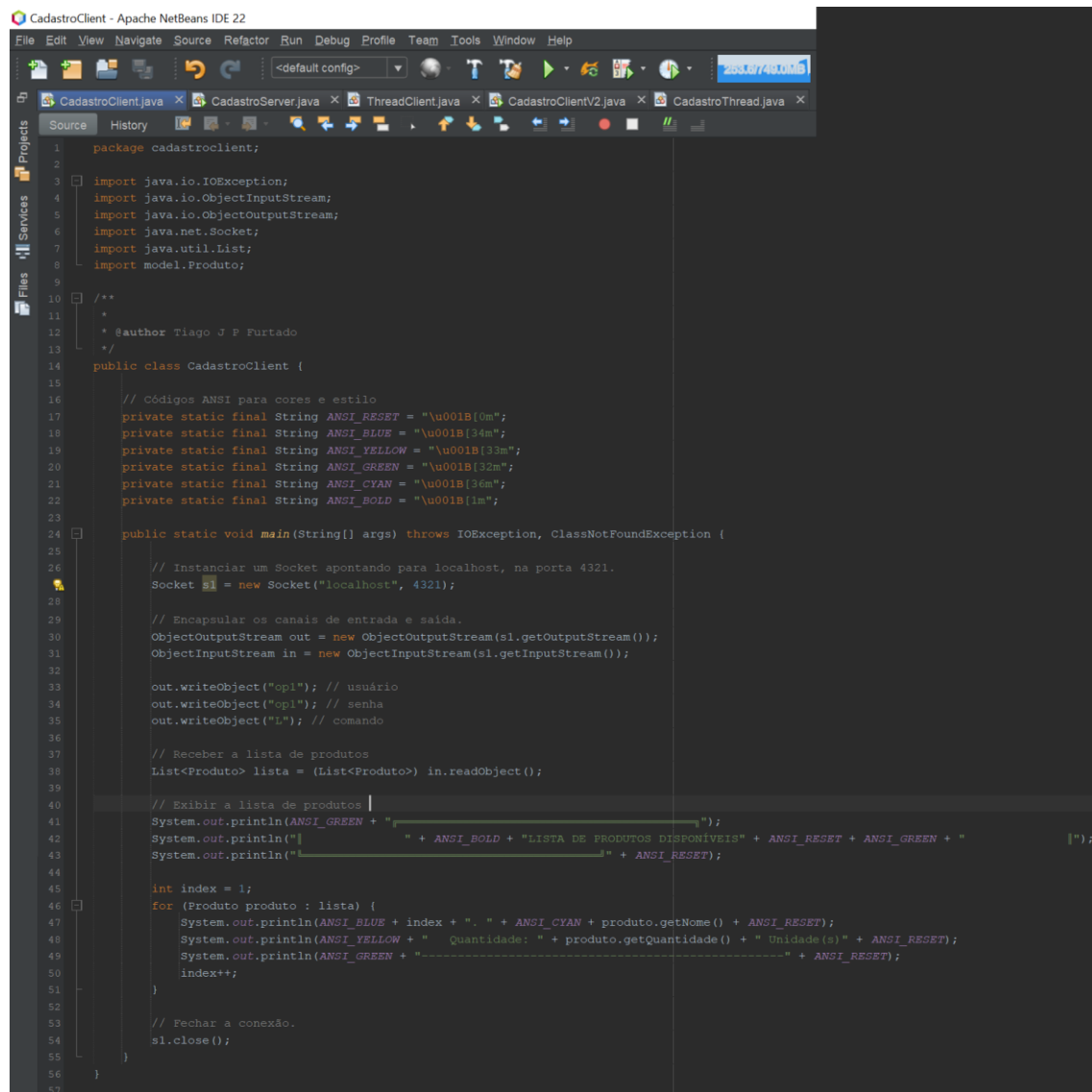
4. No pacote principal, cadastroserver, adicionar a Thread de comunicação, com o nome CadastroThread.
- ✓ Acrescentar os atributos ctrl e ctrlUsu, dos tipos ProdutoJpaController e UsuarioJpaController, respectivamente.
 - ✓ Acrescentar o atributo s1 para receber o Socket.
 - ✓ Definir um construtor recebendo os controladores e o Socket, com a passagem dos valores para os atributos internos.
 - ✓ Implementar o método run para a Thread, cujo funcionamento será o descrito a seguir.
 - Encapsular os canais de entrada e saída do Socket em objetos dos tipos ObjectOutputStream (saída) e ObjectInputStream (entrada).
 - Obter o login e a senha a partir da entrada.
 - Utilizar ctrlUsu para verificar o login, terminando a conexão caso o retorno seja nulo.
 - Com o usuário válido, iniciar o loop de resposta, que deve obter o comando a partir da entrada.
 - Caso o comando seja L, utilizar ctrl para retornar o conjunto de produtos através da saída.

```
CadastroServer.java x CadastroThread.java x
Source History
1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this
4   */
5   package cadastroserver;
6
7   /**
8    *
9    * @author Tiago J F Furtado
10   */
11  import java.io.IOException;
12  import java.io.ObjectInputStream;
13  import java.io.ObjectOutputStream;
14  import java.net.Socket;
15  import java.time.LocalDateTime;
16  import java.time.format.DateTimeFormatter;
17  import java.util.List;
18
19  import model.Produto;
20  import model.Usuario;
21  import controller.ProdutoJpaController;
22  import controller.UsuarioJpaController;
23
24  public class CadastroThread extends Thread {
25
26      private ProdutoJpaController ctrlProduto;
27      private UsuarioJpaController ctrlUsuario;
28      private Socket s1;
29
30      public CadastroThread(ProdutoJpaController ctrlProduto, UsuarioJpaController ctrlUsuario, Socket s1) {
31          this.ctrlProduto = ctrlProduto;
32          this.ctrlUsuario = ctrlUsuario;
33          this.s1 = s1;
34      }
35
36      @Override
37      public void run() {
38          try {
39              ObjectInputStream in = new ObjectInputStream(s1.getInputStream());
40              ObjectOutputStream out = new ObjectOutputStream(s1.getOutputStream());
41
42              String login = (String) in.readObject();
43              String senha = (String) in.readObject();
44
45              Usuario usuario = ctrlUsuario.findUsuario(login, senha);
46              if (usuario == null) {
47                  System.out.println("Usuário inválido");
48                  s1.close();
49                  return;
50              }
51
52              // Obter a data e hora atuais
53              LocalDateTime now = LocalDateTime.now();
54              DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
55
56              // Exibir a mensagem de sucesso com a data e hora
57              System.out.println("Usuário conectado com sucesso : " + now.format(formatter));
58
59              while (true) {
60                  String comando = (String) in.readObject();
61
62                  if (comando.equals("L")) {
63                      List<Produto> produtos = ctrlProduto.findProdutoEntities();
64                      out.writeObject(produtos);
65                  } else {
66                      System.out.println("Comando inválido");
67                  }
68              }
69          } catch (IOException | ClassNotFoundException e) {
70              e.printStackTrace(); // Para ajudar na depuração, imprimir o stack trace em caso de exceção
71          }
72      }
73  }
```

5. Implementar a classe de execução (main), utilizando as características que são apresentadas a seguir.
- ✓ Instanciar um objeto do tipo EntityManagerFactory a partir da unidade de persistência.
 - ✓ Instanciar o objeto ctrl, do tipo ProdutoJpaController.
 - ✓ Instanciar o objeto ctrlUsu do tipo UsuarioJpaController.
 - ✓ Instanciar um objeto do tipo ServerSocket, escutando a porta 4321.
 - ✓ Dentro de um loop infinito, obter a requisição de conexão do cliente, instanciar uma Thread, com a passagem de ctrl, ctrlUsu e do Socket da conexão, iniciando-a em seguida.
 - ✓ Com a Thread respondendo ao novo cliente, o servidor ficará livre para escutar a próxima solicitação de conexão.

```
1 package cadastroserver;
2
3 /**
4  *
5  * @author Tiago J P Furtado
6  */
7 import controller.MovimentoJpaController;
8 import controller.PessoaJpaController;
9 import controller.ProdutoJpaController;
10 import controller.UsuarioJpaController;
11 import java.io.IOException;
12 import java.net.ServerSocket;
13 import java.net.Socket;
14 import javax.persistence.EntityManagerFactory;
15 import javax.persistence.Persistence;
16
17 public class CadastroServer {
18
19     // Códigos ANSI para cores
20     private static final String ANSI_RESET = "\u001B[0m";
21     private static final String ANSI_BLUE = "\u001B[34m";
22     private static final String ANSI_YELLOW = "\u001B[33m";
23     private static final String ANSI_GREEN = "\u001B[32m";
24     private static final String ANSI_BOLD = "\u001B[1m";
25
26     public static void main(String[] args) throws IOException {
27         // Exibe uma mensagem de boas-vindas ao iniciar o servidor
28         System.out.println(ANSI_GREEN + "=====");
29         System.out.println(" ");
30         System.out.println(" " + ANSI_BLUE + ANSI_BOLD + "BEM-VINDO AO CADASTROSERVER!" + ANSI_RESET + ANSI_GREEN + " ");
31         System.out.println(" ");
32         System.out.println(" " + ANSI_YELLOW + ANSI_BOLD + "POR FAVOR, AGUARDE A CONEXÃO..." + ANSI_RESET + ANSI_GREEN + " ");
33         System.out.println(" ");
34         System.out.println("=====");
35
36         EntityManagerFactory emf = Persistence.createEntityManagerFactory("CadastroServerPU");
37
38         ProdutoJpaController ctrlProduto = new ProdutoJpaController(emf);
39         UsuarioJpaController ctrlUsuario = new UsuarioJpaController(emf);
40         MovimentoJpaController ctrlMov = new MovimentoJpaController(emf);
41         PessoaJpaController ctrlPessoa = new PessoaJpaController(emf);
42
43         ServerSocket s1 = new ServerSocket(4321);
44
45         while (true) {
46             Socket s2 = s1.accept();
47
48             CadastroThreadV2 t1 = new CadastroThreadV2(ctrlProduto, ctrlUsuario, s2, ctrlMov, ctrlPessoa);
49             t1.start();
50         }
51     }
52 }
53
```

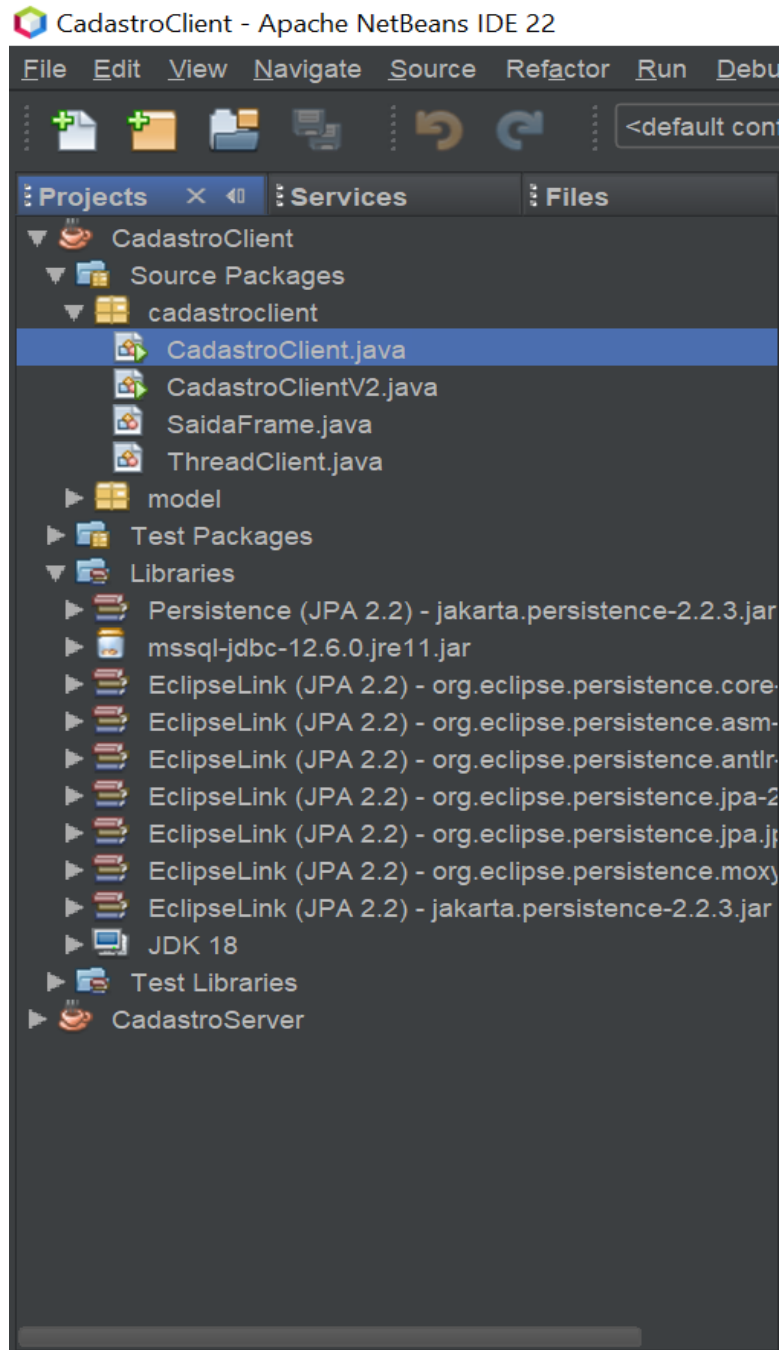
6. Criar o cliente de teste, utilizando o nome CadastroClient, do tipo console, no modelo Ant padrão, para implementar a funcionalidade apresentada a seguir:
- ✓ Instanciar um Socket apontando para localhost, na porta 4321.
 - ✓ Encapsular os canais de entrada e saída do Socket em objetos dos tipos ObjectOutputStream (saída) e ObjectInputStream (entrada).
 - ✓ Escrever o login e a senha na saída, utilizando os dados de algum dos registros da tabela de usuários (op1/op1).
 - ✓ Enviar o comando L no canal de saída.
 - ✓ Receber a coleção de entidades no canal de entrada.
 - ✓ Apresentar o nome de cada entidade recebida.
 - ✓ Fechar a conexão.



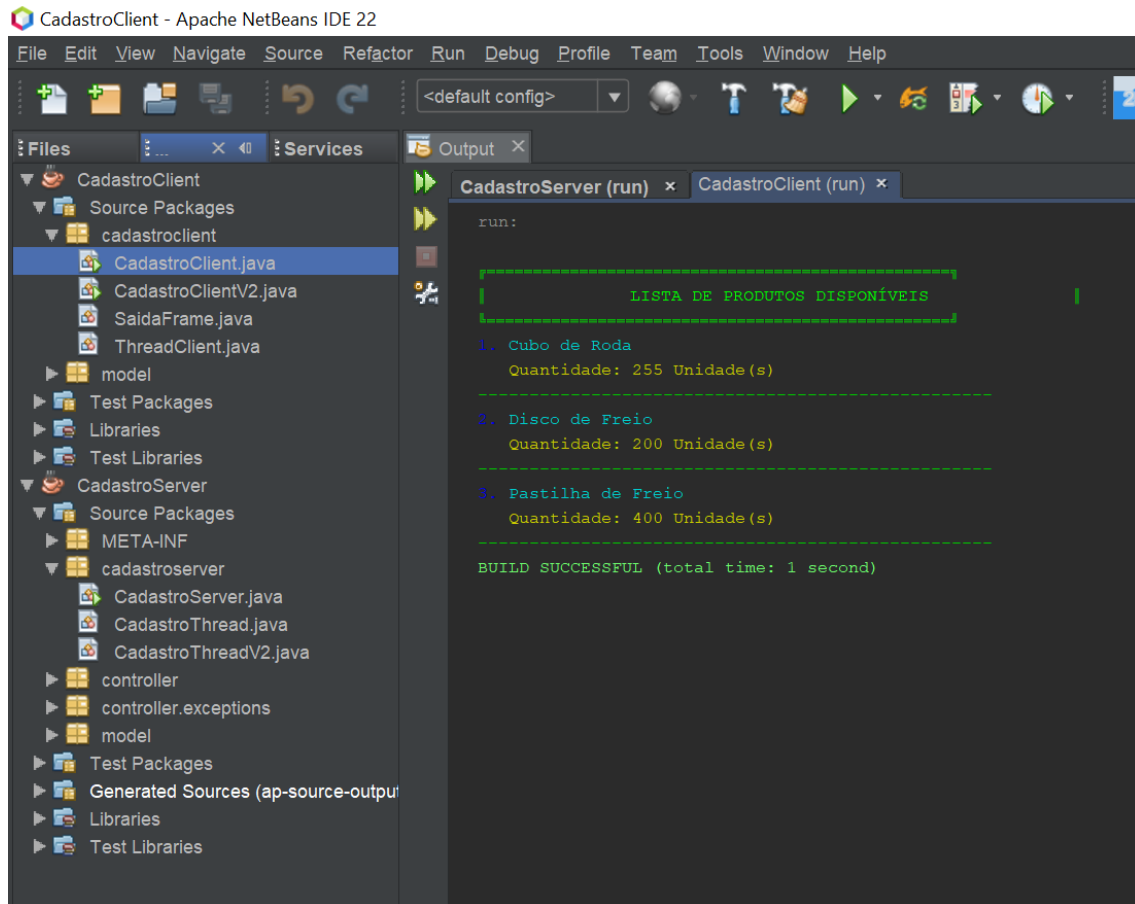
```
1 package cadastroclient;
2
3 import java.io.IOException;
4 import java.io.ObjectInputStream;
5 import java.io.ObjectOutputStream;
6 import java.net.Socket;
7 import java.util.List;
8 import model.Produto;
9
10 /**
11  *
12  * @author Tiago J F Furtado
13  */
14 public class CadastroClient {
15
16     // Códigos ANSI para cores e estilo
17     private static final String ANSI_RESET = "\u001B[0m";
18     private static final String ANSI_BLUE = "\u001B[34m";
19     private static final String ANSI_YELLOW = "\u001B[33m";
20     private static final String ANSI_GREEN = "\u001B[32m";
21     private static final String ANSI_CYAN = "\u001B[36m";
22     private static final String ANSI_BOLD = "\u001B[1m";
23
24     public static void main(String[] args) throws IOException, ClassNotFoundException {
25
26         // Instanciar um Socket apontando para localhost, na porta 4321.
27         Socket s1 = new Socket("localhost", 4321);
28
29         // Encapsular os canais de entrada e saída.
30         ObjectOutputStream out = new ObjectOutputStream(s1.getOutputStream());
31         ObjectInputStream in = new ObjectInputStream(s1.getInputStream());
32
33         out.writeObject("op1"); // usuário
34         out.writeObject("op1"); // senha
35         out.writeObject("L"); // comando
36
37         // Receber a lista de produtos
38         List<Produto> lista = (List<Produto>) in.readObject();
39
40         // Exibir a lista de produtos
41         System.out.println(ANSI_GREEN + "=====");
42         System.out.println(" " + ANSI_BOLD + "LISTA DE PRODUTOS DISPONÍVEIS" + ANSI_RESET + ANSI_GREEN + " ");
43         System.out.println("=====");
44
45         int index = 1;
46         for (Produto produto : lista) {
47             System.out.println(ANSI_BLUE + index + ". " + ANSI_CYAN + produto.getNome() + ANSI_RESET);
48             System.out.println(ANSI_YELLOW + "    Quantidade: " + produto.getQuantidade() + " Unidade(s)" + ANSI_RESET);
49             System.out.println(ANSI_GREEN + "-----" + ANSI_RESET);
50             index++;
51         }
52
53         // Fechar a conexão.
54         s1.close();
55     }
56 }
57
```


7. Configurar o projeto do cliente para uso das entidades:

- ✓ Copiar o pacote model do projeto servidor para o cliente.
- ✓ Adicionar a biblioteca Eclipse Link (JPA 2.1).
- ✓ A configuração final pode ser observada a seguir.

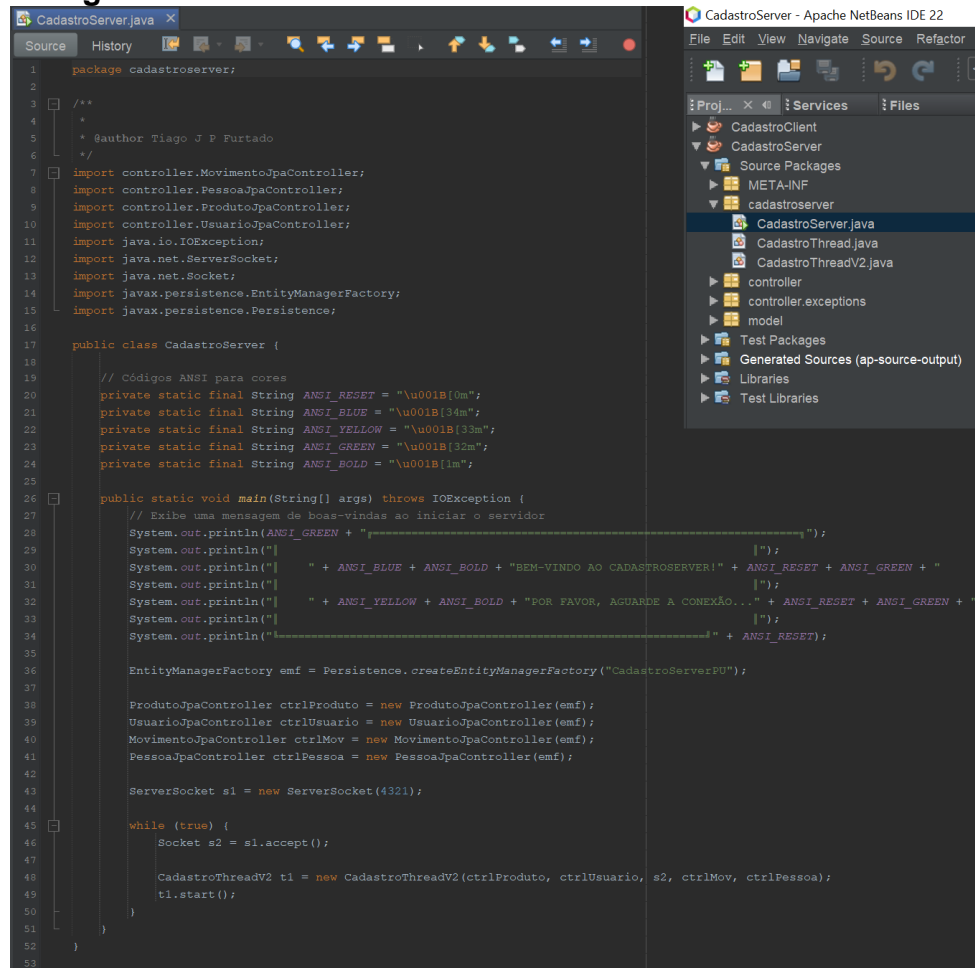


8. Testar o sistema criado, com a execução dos dois projetos:
- ✓ Executar o projeto servidor.
 - ✓ Executar, em seguida, o cliente.
 - ✓ A saída do cliente deverá ser como a que é apresentada a seguir.



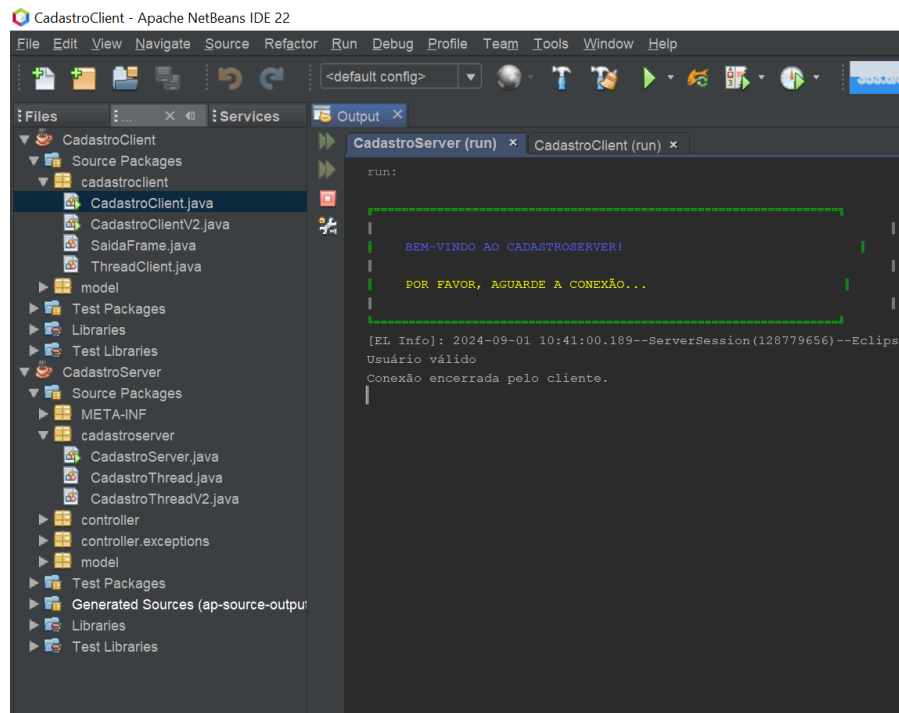
4 - Os resultados da execução dos códigos também devem ser apresentados:

Codigo do CadastroServer:



```
1 package cadastroserver;
2
3 /**
4  *
5  * @author Tiago J P Furtado
6  */
7 import controller.MovimentoJpaController;
8 import controller.PessoaJpaController;
9 import controller.ProdutoJpaController;
10 import controller.UsuarioJpaController;
11 import java.io.IOException;
12 import java.net.ServerSocket;
13 import java.net.Socket;
14 import javax.persistence.EntityManagerFactory;
15 import javax.persistence.Persistence;
16
17 public class CadastroServer {
18
19     // Códigos ANSI para cores
20     private static final String ANSI_RESET = "\u001B[0m";
21     private static final String ANSI_BLUE = "\u001B[34m";
22     private static final String ANSI_YELLOW = "\u001B[33m";
23     private static final String ANSI_GREEN = "\u001B[32m";
24     private static final String ANSI_BOLD = "\u001B[1m";
25
26     public static void main(String[] args) throws IOException {
27         // Exibe uma mensagem de boas-vindas ao iniciar o servidor
28         System.out.println(ANSI_GREEN + "-----");
29         System.out.println("                                ");
30         System.out.println(" " + ANSI_BLUE + ANSI_BOLD + "BEM-VINDO AO CADASTROSERVER!" + ANSI_RESET + ANSI_GREEN + " ");
31         System.out.println("                                ");
32         System.out.println(" " + ANSI_YELLOW + ANSI_BOLD + "POR FAVOR, AGUARDE A CONEXÃO..." + ANSI_RESET + ANSI_GREEN + " ");
33         System.out.println("                                ");
34         System.out.println("-----" + ANSI_RESET);
35
36         EntityManagerFactory emf = Persistence.createEntityManagerFactory("CadastroServerPU");
37
38         ProdutoJpaController ctrlProduto = new ProdutoJpaController(emf);
39         UsuarioJpaController ctrlUsuario = new UsuarioJpaController(emf);
40         MovimentoJpaController ctrlMov = new MovimentoJpaController(emf);
41         PessoaJpaController ctrlPessoa = new PessoaJpaController(emf);
42
43         ServerSocket s1 = new ServerSocket(4321);
44
45         while (true) {
46             Socket s2 = s1.accept();
47
48             CadastroThreadV2 t1 = new CadastroThreadV2(ctrlProduto, ctrlUsuario, s2, ctrlMov, ctrlPessoa);
49             t1.start();
50         }
51     }
52 }
```

Resultado:



```
run:
[EL Info]: 2024-09-01 10:41:00.189--ServerSession(128779656)--Eclipse
Usuário válido
Conexão encerrada pelo cliente.
```

Codigo CadastroClient:

```
CadastroClient - Apache NetBeans IDE 22
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

CadastroClient.java CadastroServer.java ThreadClient.java CadastroClientV2.java CadastroThread.java

Source History

1 package cadastroclient;
2
3 import java.io.IOException;
4 import java.io.ObjectInputStream;
5 import java.io.ObjectOutputStream;
6 import java.net.Socket;
7 import java.util.List;
8 import model.Produto;
9
10 /**
11  *
12  * @author Tiago J P Furtado
13  */
14 public class CadastroClient {
15
16     // Códigos ANSI para cores e estilo
17     private static final String ANSI_RESET = "\u001B[0m";
18     private static final String ANSI_BLUE = "\u001B[34m";
19     private static final String ANSI_YELLOW = "\u001B[33m";
20     private static final String ANSI_GREEN = "\u001B[32m";
21     private static final String ANSI_CYAN = "\u001B[36m";
22     private static final String ANSI_BOLD = "\u001B[1m";
23
24     public static void main(String[] args) throws IOException, ClassNotFoundException {
25
26         // Instanciar um Socket apontando para localhost, na porta 4321.
27         Socket s1 = new Socket("localhost", 4321);
28
29         // Encapsular os canais de entrada e saída.
30         ObjectOutputStream out = new ObjectOutputStream(s1.getOutputStream());
31         ObjectInputStream in = new ObjectInputStream(s1.getInputStream());
32
33         out.writeObject("op1"); // usuário
34         out.writeObject("op1"); // senha
35         out.writeObject("L"); // comando
36
37         // Receber a lista de produtos
38         List<Produto> lista = (List<Produto>) in.readObject();
39
40         // Exibir a lista de produtos
41         System.out.println(ANSI_GREEN + "-----");
42         System.out.println(" " + ANSI_BOLD + "LISTA DE PRODUTOS DISPONÍVEIS" + ANSI_RESET + ANSI_GREEN + " " + ANSI_BOLD + "-----");
43         System.out.println("-----");
44
45         int index = 1;
46         for (Produto produto : lista) {
47             System.out.println(ANSI_BLUE + index + ". " + ANSI_CYAN + produto.getNome() + ANSI_RESET);
48             System.out.println(ANSI_YELLOW + "    Quantidade: " + produto.getQuantidade() + " Unidade(s)" + ANSI_RESET);
49             System.out.println(ANSI_GREEN + "-----");
50             index++;
51         }
52
53         // Fechar a conexão.
54         s1.close();
55     }
56 }
57
```

Resultado:

```
CadastroClient - Apache NetBeans IDE 22
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

Files Services Output

CadastroClient
└─ Source Packages
   └─ cadastroclient
      ├── CadastroClient.java
      ├── CadastroClientV2.java
      ├── SaldaFrame.java
      └── ThreadClient.java
  model
  Test Packages
  Libraries
  Test Libraries
  CadastroServer
  └─ Source Packages
     ├── META-INF
     └─ cadastroserver
        ├── CadastroServer.java
        ├── CadastroThread.java
        └── CadastroThreadV2.java
  controller
  controller.exceptions
  model
  Test Packages
  Generated Sources (ap-source-output)
  Libraries
  Test Libraries

CadastroServer (run) CadastroClient (run)

run:
-----
LISTA DE PRODUTOS DISPONÍVEIS
-----
1. Cubo de Roda
   Quantidade: 255 Unidade(s)
-----
2. Disco de Freio
   Quantidade: 200 Unidade(s)
-----
3. Pastilha de Freio
   Quantidade: 400 Unidade(s)
-----
BUILD SUCCESSFUL (total time: 1 second)
```

5 – Análise e Conclusão:

❖ Como funcionam as classes Socket e ServerSocket?

- ✓ As classes Socket e ServerSocket são fundamentais na programação de redes em Java. Elas permitem que você estabeleça comunicação entre diferentes máquinas em uma rede, utilizando o protocolo TCP (Transmission Control Protocol). Aqui está um resumo de como elas funcionam:


❖ Classe Socket

- ✓ A classe Socket representa um endpoint (ou ponto final) de uma conexão entre dois dispositivos na rede. É utilizada pelo lado do cliente para se conectar a um servidor.

❖ Principais métodos:

- ✓ **Socket (String host, int port):** Cria um socket e conecta-o ao host e porta especificados.
- ✓ **getInputStream ():** Retorna um InputStream que pode ser usado para ler dados do servidor.
- ✓ **getOutputStream ():** Retorna um OutputStream que pode ser usado para enviar dados ao servidor.
- ✓ **Close ():** Fecha o socket, encerrando a conexão.

Exemplo de uso:

```
1
2  Socket socket = new Socket("localhost", 12345);
3  OutputStream out = socket.getOutputStream();
4  InputStream in = socket.getInputStream();
5
6  // Comunicação (envio e recebimento de dados)
7  
8  socket.close(); // Fecha a conexão
9
10
```

❖ Classe **ServerSocket**

- ✓ A classe **ServerSocket** é utilizada no lado do servidor para esperar e aceitar conexões de clientes. Quando um cliente tenta se conectar ao servidor, um novo **Socket** é criado para essa conexão específica.

❖ Principais métodos:

- ✓ **ServerSocket (int port):** Cria um socket de servidor que espera conexões na porta especificada.
- ✓ **Accept ():** Aguarda até que um cliente se conecte, e então retorna um **Socket** que pode ser usado para comunicação com o cliente.
- ✓ **Close ():** Fecha o **ServerSocket**, impedindo novas conexões.

Exemplo de uso:

```
1
2  ServerSocket serverSocket = new ServerSocket(12345);
3  Socket clientSocket = serverSocket.accept(); // Aguarda uma conexão
4
5  OutputStream out = clientSocket.getOutputStream();
6  InputStream in = clientSocket.getInputStream();
7
8  // Comunicação (envio e recebimento de dados)
9
10 clientSocket.close(); // Fecha a conexão com o cliente
11 serverSocket.close(); // Fecha o servidor
12 |
```

❖ Como elas funcionam juntas?

- ✓ **No lado do servidor:** Cria-se um objeto **ServerSocket** para escutar as conexões dos clientes em uma porta específica. Quando um cliente tenta se conectar, o método **accept ()** do **ServerSocket** retorna um **Socket** que representa a conexão entre o cliente e o servidor.
- ✓ **No lado do cliente:** Cria-se um objeto **Socket** que tenta se conectar ao servidor em um endereço IP e porta específicos. Uma vez conectado, tanto o cliente quanto o servidor podem enviar e receber dados através dos streams de entrada e saída do socket.

❖ Qual a importância das portas para a conexão com servidores?

- ✓ As portas são fundamentais na comunicação de redes, pois permitem que os servidores possam gerenciar múltiplas conexões e serviços simultaneamente. Aqui está um resumo da importância das portas na conexão com servidores:

Identificação de Serviços

- ✓ Cada porta em um servidor representa um ponto de entrada para um serviço específico. Quando um cliente deseja se comunicar com um servidor, ele precisa saber tanto o endereço IP do servidor quanto a porta correta que o serviço desejado está escutando.

Exemplo:

- ✓ HTTP geralmente usa a porta 80.
- ✓ HTTPS geralmente usa a porta 443.
- ✓ FTP geralmente usa a porta 21.
- ✓ Servidores de e-mail (SMTP) usam a porta 25.

Multiplexação de Conexões

- ✓ Um único servidor pode executar vários serviços diferentes ao mesmo tempo, cada um em uma porta diferente. Isso significa que o servidor pode atender várias solicitações simultâneas, com diferentes serviços ouvindo em diferentes portas.

Exemplo:

- ✓ Um servidor pode estar executando um site (porta 80), um serviço de e-mail (porta 25), e um serviço de banco de dados (porta 3306) ao mesmo tempo, sem conflitos.

Segurança

- ✓ As portas também são um elemento importante na segurança da rede. Firewalls e outras medidas de segurança de rede frequentemente monitoram ou bloqueiam o tráfego para certas portas, permitindo apenas conexões em portas específicas.

Exemplo:

- ✓ Um firewall pode bloquear todas as portas exceto a 80 e 443 para permitir apenas o tráfego HTTP e HTTPS, protegendo o servidor de acessos não autorizados a outros serviços.

Controle e Manutenção

- ✓ Administradores de sistemas podem configurar servidores para escutar em portas específicas como uma forma de controle ou para evitar conflitos. Por exemplo, um servidor de desenvolvimento pode usar portas diferentes das de um servidor de produção para evitar colisões.

Protocolo de Comunicação

- ✓ O número da porta, junto com o endereço IP, forma um "socket" que identifica de maneira única uma conexão de rede. Isso permite que o servidor diferencie entre várias conexões simultâneas de diferentes clientes ou até mesmo do mesmo cliente para diferentes serviços.

Conclusão

- ✓ As portas são essenciais para a estrutura e funcionamento da comunicação em redes. Elas permitem que um servidor ofereça múltiplos serviços simultâneos, ajudam na segurança, e são fundamentais para a administração e operação de servidores em um ambiente de rede.

❖ **Para que servem as classes de entrada e saída `ObjectInputStream` e `ObjectOutputStream`, e por que os objetos transmitidos devem ser serializáveis?**

- ✓ As classes `ObjectInputStream` e `ObjectOutputStream` são utilizadas em Java para facilitar a leitura e gravação de objetos em streams, permitindo a transmissão de objetos entre diferentes partes de um programa, como entre clientes e servidores em uma rede ou para armazenamento em arquivos.

`ObjectInputStream` e `ObjectOutputStream`

- ✓ Essas classes fazem parte da biblioteca de I/O (entrada/saída) de Java e são usadas para ler e escrever objetos em streams. Elas são especialmente úteis quando você precisa persistir o estado de um objeto ou transmiti-lo pela rede.

`ObjectOutputStream`:

- ✓ Esta classe é usada para gravar objetos em um stream de saída. Quando um objeto é gravado usando `ObjectOutputStream`, ele é convertido em uma sequência de bytes, que pode ser transmitida ou armazenada.

Principais métodos:

- ✓ **`writeObject (Object obj)`:** Escreve o objeto especificado no stream de saída.
- ✓ **`Flush ()`:** Esvazia o stream de saída, garantindo que todos os dados sejam gravados.

`ObjectInputStream`:

- ✓ Esta classe é usada para ler objetos de um stream de entrada. Ela reconstrói o objeto a partir da sequência de bytes lida do stream.

Principais métodos:

`readObject ()`:

- ✓ Lê um objeto do stream de entrada e o reconstrói.

Exemplo de uso:

```
1 // Gravar um objeto em um arquivo
2 ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("objeto.ser"));
3 out.writeObject(meuObjeto);
4 out.close();
5
6 // Ler um objeto de um arquivo
7 ObjectInputStream in = new ObjectInputStream(new FileInputStream("objeto.ser"));
8 MeuObjeto meuObjetoRecuperado = (MeuObjeto) in.readObject();
9 in.close();
10 |
```


Serialização e Por Que É Necessária

- ✓ Para que um objeto possa ser gravado em um stream (e consequentemente transmitido ou armazenado), ele deve ser **serializável**.
- ✓ Serialização é o processo de converter o estado de um objeto em uma sequência de bytes. Deserialização é o processo inverso, onde a sequência de bytes é convertida de volta em um objeto.

Implementação da Interface Serializable:

- ✓ Em Java, um objeto é considerado serializável se a sua classe implementar a interface Serializable. Essa interface é um marcador (ou seja, não possui métodos) que indica que os objetos dessa classe podem ser serializados.

Exemplo:

```
1
2  public class MeuObjeto implements Serializable {
3      private static final long serialVersionUID = 1L;
4      private String nome;
5      private int idade;
6      // Construtores, getters e setters...
7  }
8  |
```

Por que a Serialização é Necessária:

Persistência de Dados:

- ✓ Ao serializar um objeto, você pode salvá-lo em um arquivo ou banco de dados, e recuperá-lo posteriormente, preservando seu estado.

Transmissão de Dados:

- ✓ Quando você precisa enviar um objeto pela rede, ele deve ser convertido em bytes para que possa ser transmitido via streams de entrada e saída.

Clonagem Profunda:

- ✓ A serialização pode ser usada para criar uma cópia profunda de um objeto.

Importância do serialVersionUID:

- ✓ A classe serializável geralmente inclui um campo serialVersionUID, que é uma versão única para a classe. Isso ajuda a garantir a compatibilidade entre diferentes versões da classe durante a serialização e deserialização.

Conclusão

- ✓ As classes ObjectOutputStream e ObjectOutputStream são essenciais para a transmissão e armazenamento de objetos em Java. A serialização é necessária para que os objetos possam ser convertidos em uma forma que possa ser facilmente transmitida ou armazenada. Ao marcar uma classe como Serializable, você está indicando que os objetos dessa classe podem ser convertidos em uma sequência de bytes e reconstruídos a partir deles.

❖ **Por que, mesmo utilizando as classes de entidades JPA no cliente, foi possível garantir o isolamento do acesso ao banco de dados?**

- ✓ Mesmo utilizando as classes de entidades JPA (Java Persistence API) no lado do cliente, o isolamento do acesso ao banco de dados pode ser garantido devido à arquitetura e às práticas de design que controlam como e onde a lógica de acesso ao banco de dados é executada. Aqui estão os principais motivos:

Controle Centralizado de Acesso ao Banco de Dados no Servidor
Camada de Serviço:

- ✓ Em uma arquitetura típica de aplicação corporativa (como a arquitetura de três camadas), o acesso ao banco de dados é centralizado na camada de serviço, que reside no servidor. As operações de banco de dados, como consultas, inserções, atualizações e exclusões, são realizadas exclusivamente nesta camada, e não no lado do cliente.

Cliente como Consumidor de Serviços:

- ✓ O cliente, mesmo possuindo classes de entidade JPA, não acessa diretamente o banco de dados. Em vez disso, o cliente consome serviços expostos pelo servidor (como via REST, SOAP ou RMI), que encapsulam as operações de banco de dados. O cliente envia pedidos ao servidor, que por sua vez interage com o banco de dados, garantindo que o banco de dados não seja exposto diretamente.

Desacoplamento Entre Entidades JPA e Persistência

Transferência de Dados (DTOs):

- ✓ Muitas arquiteturas utilizam objetos de transferência de dados (DTOs) para enviar dados entre o cliente e o servidor. As entidades JPA são usadas principalmente no servidor para mapeamento objeto-relacional e não são necessariamente expostas diretamente ao cliente. Mesmo que as entidades JPA sejam compartilhadas entre o cliente e o servidor, o cliente manipula essas entidades apenas como objetos simples (POJOs), sem qualquer capacidade de persistência direta.

Uso de Facades ou Repositórios:

- ✓ Padrões de design como Facades ou Repositórios são utilizados para encapsular o acesso a dados no servidor. Assim, o cliente não interage diretamente com o EntityManager ou as transações JPA, mas sim com interfaces de serviço que protegem o banco de dados de acessos não autorizados ou diretos.

Segurança e Controle de Transações

Controle de Transações no Servidor:

- ✓ O controle de transações (commit, rollback) é gerido pelo servidor, onde reside o EntityManager do JPA. Isso garante que todas as transações sejam tratadas de forma centralizada e segura, evitando inconsistências que poderiam ocorrer se o controle fosse distribuído para os clientes.

Segurança de Conexão e Autenticação:

- ✓ Somente o servidor possui as credenciais e a lógica necessária para autenticar e conectar-se ao banco de dados. O cliente não tem acesso a essas credenciais, garantindo que qualquer tentativa de acesso ao banco de dados deve passar pela camada de serviço do servidor.

Arquitetura Cliente-Servidor

Isolamento Natural:

- ✓ A arquitetura cliente-servidor naturalmente isola o banco de dados do cliente. O cliente não sabe como as operações são realizadas no banco de dados; ele apenas faz requisições ao servidor e recebe respostas. Essa separação de responsabilidades garante que o banco de dados esteja protegido de acessos diretos e de manipulações indevidas.

Conclusão

Mesmo que as classes de entidades JPA sejam utilizadas no cliente, o isolamento do acesso ao banco de dados é garantido por uma combinação de controle centralizado de operações de banco de dados no servidor, desacoplamento entre entidades e persistência, controle rigoroso de transações e segurança, além da própria arquitetura cliente-servidor. O cliente nunca interage diretamente com o banco de dados, garantindo a integridade, segurança e consistência dos dados.

2º Procedimento | Servidor Completo e Cliente Assíncrono

Objetivo da Prática:

- Criar servidores Java com base em Sockets.
- Criar clientes síncronos para servidores com base em Sockets.
- Criar clientes assíncronos para servidores com base em Sockets.
- Utilizar Threads para implementação de processos paralelos.
- No final do exercício, o aluno terá criado um servidor Java baseado em Socket, com acesso ao banco de dados via JPA, além de utilizar os recursos nativos do Java para implementação de clientes síncronos e assíncronos. As Threads serão usadas tanto no servidor, para viabilizar múltiplos clientes paralelos, quanto no cliente, para implementar a resposta assíncrona.



Todos os códigos solicitados neste roteiro de aula:

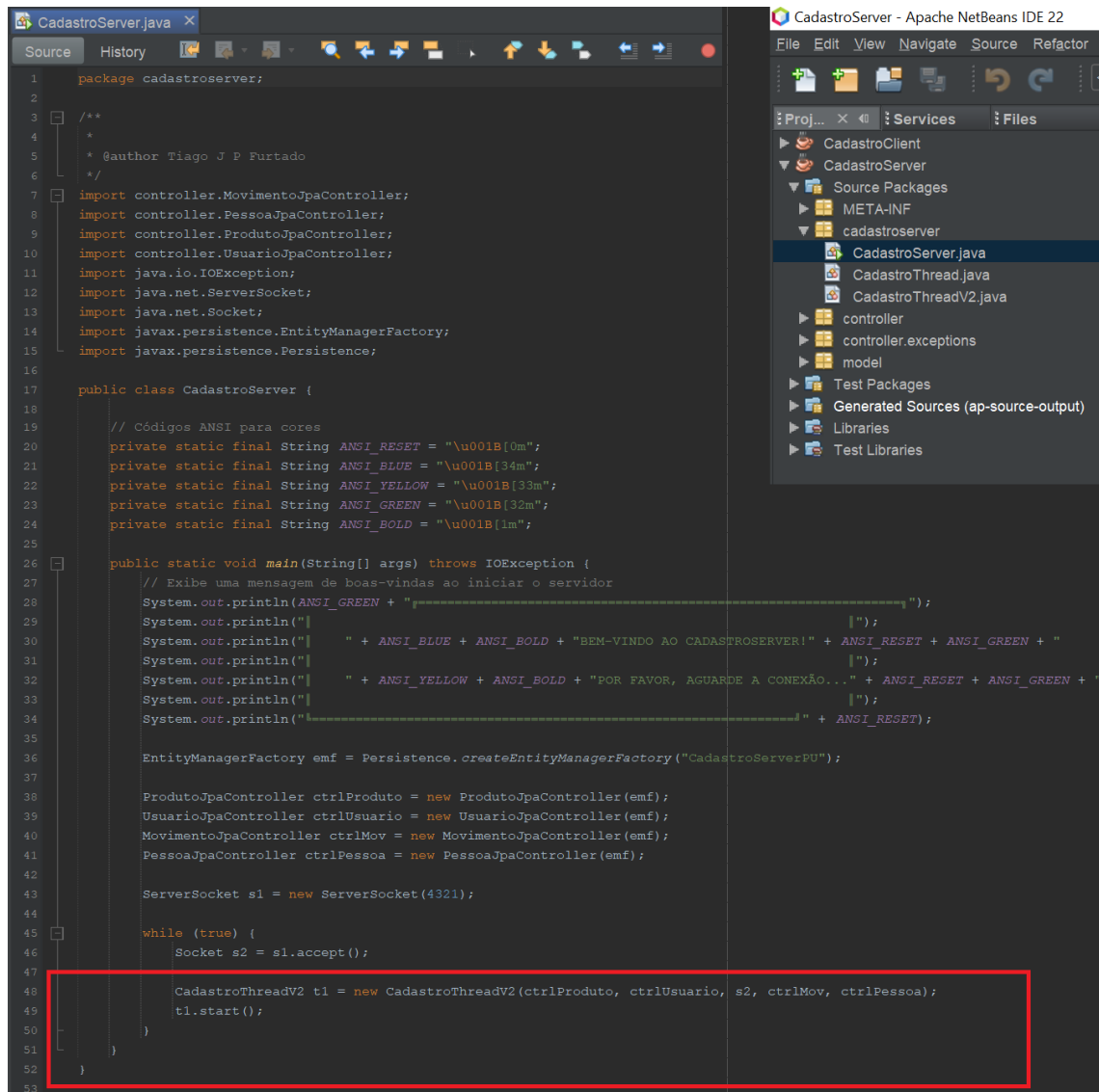
1. Criar uma segunda versão da Thread de comunicação, no projeto do servidor, com o acréscimo da funcionalidade apresentada a seguir:
 - a- Servidor recebe comando E, para entrada de produtos, ou S, para saída.
 - b- Gerar um objeto Movimento, configurado com o usuário logado e o tipo, que pode ser E ou S.
 - c- Receber o Id da pessoa e configurar no objeto Movimento.
 - d- Receber o Id do produto e configurar no objeto Movimento.
 - e- Receber a quantidade e configurar no objeto Movimento.
 - f- Receber o valor unitário e configurar no objeto Movimento.
 - g- Persistir o movimento através de um MovimentoJpaController com o nome ctrlMov.
 - h- Atualizar a quantidade de produtos, de acordo com o tipo de movimento, através de ctrlProd.
- ✓ Observação! Devem ser acrescentados os atributos ctrlMov e ctrlPessoa, dos tipos MovimentoJpaController e PessoaJpaController, alimentados por meio de parâmetros no construtor.

Segue abaixo o Código CadastroThreadV2 Referente as Configurações Solicitadas acima.

Codigo CadastroThreadV2:

```
1 package cadastrothreadv2;
2
3 import controller.MovimentoJpaController;
4 import controller.PessoaJpaController;
5 import controller.ProdutoJpaController;
6 import controller.UsuarioJpaController;
7 import controller.UsuarioLogado;
8 import java.io.IOException;
9 import java.io.InputStream;
10 import java.io.ObjectInputStream;
11 import java.io.ObjectOutputStream;
12 import java.net.Socket;
13 import java.util.List;
14 import java.util.logging.Level;
15 import java.util.logging.Logger;
16 import model.Movimento;
17 import model.Pessoa;
18 import model.Produto;
19 import model.Usuario;
20
21 public class CadastroThreadV2 extends Thread {
22
23     private ProdutoJpaController ctrlProduto;
24     private UsuarioJpaController ctrlUsuario;
25     private Socket si;
26     private MovimentoJpaController ctrlMov;
27     private PessoaJpaController ctrlPessoa;
28
29     public CadastroThreadV2(ProdutoJpaController ctrlProduto, UsuarioJpaController ctrlUsuario, Socket si, MovimentoJpaController ctrlMov, PessoaJpaController ctrlPessoa) {
30         this.ctrlProduto = ctrlProduto;
31         this.ctrlUsuario = ctrlUsuario;
32         this.si = si;
33         this.ctrlMov = ctrlMov;
34         this.ctrlPessoa = ctrlPessoa;
35     }
36
37     @Override
38     public void run() {
39         try (ObjectInputStream in = new ObjectInputStream(si.getInputStream()); ObjectOutputStream out = new ObjectOutputStream(si.getOutputStream())) {
40             // Já o login e senha do usuário
41             String login = (String) in.readObject();
42             String senha = (String) in.readObject();
43
44             Usuario usuario = ctrlUsuario.findUsuario(login, senha);
45             if (usuario == null) {
46                 System.out.println("Usuário inválido");
47                 si.close();
48                 return;
49             }
50             System.out.println("Usuário válido");
51
52             boolean isRunning = true;
53             while (isRunning) {
54                 try {
55                     // Verifica se o cliente fechou a conexão
56                     if (si.isClosed() || in.available() == -1) {
57                         System.out.println("Conexão fechada pelo cliente.");
58                         isRunning = false;
59                         break;
60                     }
61
62                     // Lê o comando do cliente e verifica o tipo
63                     Object comandoObj = in.readObject();
64
65                     if (comandoObj instanceof String) {
66                         System.out.println("Tipo de comando inesperado: " + comandoObj.getClass().getName());
67                         continue;
68                     }
69
70                     String comando = ((String) comandoObj).toUpperCase(); // Converte o comando para maiúsculas
71
72                     switch (comando) {
73                         case "C":
74                             // Entrada de produtos
75                             int idPessoa = (int) in.readObject();
76                             int idProduto = (int) in.readObject();
77                             int quantidade = (int) in.readObject();
78                             float valorUnitario = (float) in.readObject();
79
80                             Pessoa pessoa = ctrlPessoa.findPessoa(idPessoa);
81                             Produto produto = ctrlProduto.findProduto(idProduto);
82
83                             produto.setQuantidade(produto.getQuantidade() + quantidade);
84                             ctrlProduto.edit(produto);
85
86                             Movimento movimento = new Movimento();
87                             movimento.setUsuario(usuario);
88                             movimento.setTipo("C");
89                             movimento.setPessoa(pessoa);
90                             movimento.setProduto(produto);
91                             movimento.setQuantidade(quantidade);
92                             movimento.setValorUnitario(valorUnitario);
93                             ctrlMov.create(movimento);
94                             break;
95
96                         case "R":
97                             // Saída de produtos
98                             int idPessoa = (int) in.readObject();
99                             int idProduto = (int) in.readObject();
100                             int quantidade = (int) in.readObject();
101                             float valorUnitario = (float) in.readObject();
102
103                             Pessoa pessoa = ctrlPessoa.findPessoa(idPessoa);
104                             Produto produto = ctrlProduto.findProduto(idProduto);
105
106                             produto.setQuantidade(produto.getQuantidade() - quantidade);
107                             ctrlProduto.edit(produto);
108
109                             Movimento movimento = new Movimento();
110                             movimento.setUsuario(usuario);
111                             movimento.setTipo("R");
112                             movimento.setPessoa(pessoa);
113                             movimento.setProduto(produto);
114                             movimento.setQuantidade(quantidade);
115                             movimento.setValorUnitario(valorUnitario);
116                             ctrlMov.create(movimento);
117                             break;
118
119                         case "L":
120                             // Listar produtos
121                             List<Produto> produtos = ctrlProduto.findProdutoEntities();
122                             out.writeObject(produtos);
123                             break;
124
125                         default:
126                             System.out.println("Comando inválido: " + comando);
127                             break;
128                     }
129                 } catch (IOException ex) {
130                     System.out.println("Comando encerrado pelo cliente.");
131                     isRunning = false;
132                 } catch (Exception e) {
133                     e.printStackTrace();
134                 }
135             }
136         } catch (IOException | ClassNotFoundException e) {
137             e.printStackTrace(); // Log de erro para depuração
138         } catch (Exception ex) {
139             Logger.getLogger(CadastroThreadV2.class.getName()).log(Level.SEVERE, null, ex);
140         } finally {
141             try {
142                 if (si.isClosed()) {
143                     si.close();
144                 }
145             } catch (IOException e) {
146                 e.printStackTrace();
147             }
148         }
149     }
150 }
```

2. Acrescentar os controladores necessários na classe principal, método **main**, e trocar a instância da Thread anterior pela nova Thread no loop de conexão.



```
1 package cadastroserver;
2
3 /**
4  *
5  * @author Tiago J P Furtado
6  */
7 import controller.MovimentoJpaController;
8 import controller.PessoaJpaController;
9 import controller.ProdutoJpaController;
10 import controller.UsuarioJpaController;
11 import java.io.IOException;
12 import java.net.ServerSocket;
13 import java.net.Socket;
14 import javax.persistence.EntityManagerFactory;
15 import javax.persistence.Persistence;
16
17 public class CadastroServer {
18
19     // Códigos ANSI para cores
20     private static final String ANSI_RESET = "\u001B[0m";
21     private static final String ANSI_BLUE = "\u001B[34m";
22     private static final String ANSI_YELLOW = "\u001B[33m";
23     private static final String ANSI_GREEN = "\u001B[32m";
24     private static final String ANSI_BOLD = "\u001B[1m";
25
26     public static void main(String[] args) throws IOException {
27         // Exibe uma mensagem de boas-vindas ao iniciar o servidor
28         System.out.println(ANSI_GREEN + "=====");
29         System.out.println("|");
30         System.out.println("|" + ANSI_BLUE + ANSI_BOLD + "BEM-VINDO AO CADASTROSERVER!" + ANSI_RESET + ANSI_GREEN + "|");
31         System.out.println("|");
32         System.out.println("|" + ANSI_YELLOW + ANSI_BOLD + "POR FAVOR, AGUARDE A CONEXÃO..." + ANSI_RESET + ANSI_GREEN + "|");
33         System.out.println("|");
34         System.out.println("=====|" + ANSI_RESET);
35
36         EntityManagerFactory emf = Persistence.createEntityManagerFactory("CadastroServerPU");
37
38         ProdutoJpaController ctrlProduto = new ProdutoJpaController(emf);
39         UsuarioJpaController ctrlUsuario = new UsuarioJpaController(emf);
40         MovimentoJpaController ctrlMov = new MovimentoJpaController(emf);
41         PessoaJpaController ctrlPessoa = new PessoaJpaController(emf);
42
43         ServerSocket s1 = new ServerSocket(4321);
44
45         while (true) {
46             Socket s2 = s1.accept();
47
48             CadastroThreadV2 t1 = new CadastroThreadV2(ctrlProduto, ctrlUsuario, s2, ctrlMov, ctrlPessoa);
49             t1.start();
50         }
51     }
52 }
53
```

- ✓ Criar o cliente assíncrono, utilizando o nome **CadastroClientV2**, do tipo console, no modelo Ant padrão, para implementar a funcionalidade apresentada a seguir:
- ✓ Instanciar um Socket apontando para **localhost**, na porta **4321**.
- ✓ Encapsular os canais de entrada e saída do Socket em objetos dos tipos `ObjectOutputStream` (**saída**) e `ObjectInputStream` (**entrada**).
- ✓ Escrever o **login** e a **senha** na **saída**, utilizando os dados de algum dos registros da tabela de usuários (op1/op1).
- ✓ Encapsular a leitura do teclado em um `BufferedReader`.
- ✓ Instanciar a janela para apresentação de mensagens (**Passo 4**) e a `Thread` para preenchimento assíncrono (**Passo 5**), com a passagem do canal de entrada do Socket.
- ✓ Apresentar um menu com as opções: L – Listar, X – Finalizar, E – Entrada, S – Saída.
- ✓ Receber o comando a partir do teclado.
- ✓ Para o comando **L**, apenas enviá-lo para o servidor.
- ✓ Para os comandos **E** ou **S**, enviar para o servidor e executar os seguintes passos:
 - Obter o Id da **pessoa** via teclado e enviar para o servidor.
 - Obter o Id do **produto** via teclado e enviar para o servidor.
 - Obter a **quantidade** via teclado e enviar para o servidor.
 - Obter o **valor unitário** via teclado e enviar para o servidor.
- ✓ Voltar ao passo **f** até que o comando **X** seja escolhido.

Segue Abaixo o Codigo Conforme Solicitado:

Codigo CadastroClientV2:

```
1 package cadastroclient;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6 import java.io.ObjectInputStream;
7 import java.io.ObjectOutputStream;
8 import java.net.Socket;
9 import java.swing.SwingUtilities;
10
11 public class CadastroClientV2 {
12
13     private static volatile boolean isRunning = true;
14
15     // Códigos ANSI para cores
16     private static final String ANSI_RESET = "\u001B[0m";
17     private static final String ANSI_BLUE = "\u001B[34m";
18     private static final String ANSI_CYAN = "\u001B[36m";
19     private static final String ANSI_YELLOW = "\u001B[33m";
20     private static final String ANSI_PURPLE = "\u001B[35m";
21     private static final String ANSI_BOLD = "\u001B[1m";
22
23     public static void main(String[] args) {
24         try (Socket socket = new Socket("localhost", 4321); ObjectOutputStream outputStream
25             = new ObjectOutputStream(socket.getOutputStream()); ObjectInputStream inputStream
26             = new ObjectInputStream(socket.getInputStream()); BufferedReader reader
27             = new BufferedReader(new InputStreamReader(System.in))) {
28
29             // Leitura do login e senha
30             System.out.print("Digite o login: ");
31             String login = reader.readLine();
32             System.out.print("Digite a senha: ");
33             String senha = reader.readLine();
34
35             // Enviar o login e a senha para o servidor.
36             outputStream.writeObject(login);
37             outputStream.writeObject(senha);
38
39             // Instancia a janela de mensagens
40             SaidaFrame saidaFrame = new SaidaFrame(login);
41             SwingUtilities.invokeLater(() -> saidaFrame.setVisible(true));
42
43             // Instanciar e Thread para comunicação com o servidor.
44             ThreadClient threadClient = new ThreadClient(inputStream, saidaFrame, login);
45             threadClient.start();
46
47             while (isRunning) {
48                 // Menu utilizado e mais chamativo
49                 System.out.println(ANSI_CYAN + "
50                 |" + ANSI_RESET + ANSI_PURPLE + ANSI_BOLD + "          Menu de Opções          " + ANSI_RESET + ANSI_CYAN + "|" + ANSI_RESET);
51                 System.out.println(ANSI_CYAN + "
52                 |" + ANSI_RESET + ANSI_BLUE + ANSI_BOLD + "          " + ANSI_RESET);
53                 System.out.println(ANSI_CYAN + "
54                 |" + ANSI_RESET + ANSI_BLUE + ANSI_BOLD + " [E] Entrada          " + ANSI_RESET + ANSI_CYAN + "|" + ANSI_RESET);
55                 System.out.println(ANSI_CYAN + "
56                 |" + ANSI_RESET + ANSI_BLUE + ANSI_BOLD + " [S] Salda          " + ANSI_RESET + ANSI_CYAN + "|" + ANSI_RESET);
57                 System.out.println(ANSI_CYAN + "
58                 |" + ANSI_RESET + ANSI_YELLOW + ANSI_BOLD + " [X] Finalizar          " + ANSI_RESET + ANSI_CYAN + "|" + ANSI_RESET);
59                 System.out.println(ANSI_CYAN + "
60                 |" + ANSI_RESET);
61                 System.out.print(ANSI_PURPLE + ANSI_BOLD + "Escolha uma opção: " + ANSI_RESET);
62                 String comando = reader.readLine();
63
64                 // Processamento do comando
65                 if (comando.equalsIgnoreCase("L")) {
66                     outputStream.writeObject("L");
67                 } else if (comando.equalsIgnoreCase("X")) {
68                     isRunning = false; // Define a condição de saída da thread
69                     break; // Sai do loop principal
70                 } else if (comando.equalsIgnoreCase("E") || comando.equalsIgnoreCase("S")) {
71                     // Enviar o comando para o servidor.
72                     outputStream.writeObject(comando);
73
74                     // ID Pessoa
75                     System.out.print("Digite o ID da pessoa: ");
76                     int pessoaId = Integer.parseInt(reader.readLine());
77                     outputStream.writeObject(pessoaId);
78
79                     // ID Produto
80                     System.out.print("Digite o ID do produto: ");
81                     int produtoId = Integer.parseInt(reader.readLine());
82                     outputStream.writeObject(produtoId);
83
84                     // Quantidade
85                     System.out.print("Digite a quantidade: ");
86                     int quantidade = Integer.parseInt(reader.readLine());
87                     outputStream.writeObject(quantidade);
88
89                     // Valor Unitário
90                     System.out.print("Digite o valor unitário: ");
91                     float valorUnitario = Float.parseFloat(reader.readLine());
92                     outputStream.writeObject(valorUnitario);
93                 } else {
94                     System.out.println(ANSI_YELLOW + "Comando inválido." + ANSI_RESET);
95                 }
96             }
97
98             catch (IOException e) {
99                 e.printStackTrace();
100             } finally {
101                 // Encerrar o programa corretamente
102                 System.exit(0);
103             }
104         }
105     }
106 }
```

- ✓ Criar a janela para apresentação das mensagens:
- ✓ Definir a classe **SaidaFrame** como descendente de **JDialog**
- ✓ Acrescentar um atributo público do tipo **JTextArea**, com o nome **texto**
- ✓ Ao nível do construtor, efetuar os passos apresentados a seguir:
 - Definir as dimensões da janela via **setBounds**
 - Definir o status **modal** como **false**
 - Acrescentar o componente JTextArea na janela

Segue abaixo o Código Conforme foi solicitado acima.

Codigo do SaidaFrame:

```
1 package cadastroclient;
2
3 import javax.swing.*;
4 import java.awt.*;
5 import java.io.FileWriter;
6 import java.io.IOException;
7 import java.time.LocalDateTime;
8 import java.time.format.DateTimeFormatter;
9
10 /**
11  *
12  * @author Tiago J P Furtado
13  */
14
15 /**
16  * SaidaFrame é uma classe personalizada que exibe uma janela de diálogo com uma
17  * área de texto onde as mensagens são exibidas.
18  */
19 public class SaidaFrame extends JFrame { // Mudança de JDialog para JFrame
20
21     private JTextArea texto;
22     private String mensagemInicial;
23
24     public SaidaFrame(String usuario) {
25         // Configurações básicas da janela
26         setTitle("Controle de Movimento");
27         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Fecha a janela ao clicar no "X"
28         setResizable(true); // Permite redimensionar a janela
29
30         // Configura a área de texto
31         texto = new JTextArea();
32         texto.setEditable(false); // Impede a edição do texto
33         texto.setFont(new Font("Arial", Font.PLAIN, 14)); // Define a fonte do texto
34         texto.setLineWrap(true); // Ativa a quebra de linha automática
35         texto.setWrapStyleWord(true); // Ativa a quebra de linha por palavra
36         texto.setBackground(new Color(240, 240, 240)); // Define uma cor de fundo suave
37         texto.setForeground(new Color(50, 50, 50)); // Define a cor do texto
38         texto.setBorder(BorderFactory.createLineBorder(new Color(100, 100, 100), 1)); // Adiciona uma borda ao redor da área de texto
39
40         // Adiciona um painel de título
41         JPanel titlePanel = new JPanel();
42         titlePanel.setBackground(new Color(70, 130, 180)); // Cor de fundo do painel de título
43         titlePanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10)); // Margem ao redor do painel
44         titlePanel.setLayout(new FlowLayout(FlowLayout.CENTER)); // Layout do painel de título
45
46         // Adiciona um rótulo ao painel de título
47         JLabel titleLabel = new JLabel("Mensagens do Sistema");
48         titleLabel.setFont(new Font("Arial", Font.BOLD, 16)); // Fonte do rótulo
49         titleLabel.setForeground(Color.WHITE); // Cor do texto do rótulo
50         titlePanel.add(titleLabel); // Adiciona o rótulo ao painel de título
51
52         // Adiciona o painel de título à parte superior da janela
53         getContentPane().add(titlePanel, BorderLayout.NORTH);
54
55         // Envolva a área de texto em um painel de rolagem
56         JScrollPane scrollPane = new JScrollPane(texto);
57         scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
58         scrollPane.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10)); // Adiciona margem ao redor do texto
59
60         // Adiciona o painel de rolagem à janela
61         getContentPane().add(scrollPane, BorderLayout.CENTER);
62
63         // Adiciona a mensagem inicial com o usuário e a data/hora atuais
64         LocalDateTime now = LocalDateTime.now();
65         DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
66         mensagemInicial = "Usuário: " + usuario + " | Conectado com sucesso em: " + now.format(formatter) + " | \n";
67         texto.append(mensagemInicial);
68
69         // Cria o painel de botões
70         JPanel buttonPanel = new JPanel(new FlowLayout(FlowLayout.CENTER, 10, 10));
71         buttonPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10)); // Margem ao redor do painel
72
73         // Personalização dos botões
74         JButton clearButton = createCustomButton("Limpar", new Color(255, 99, 71));
75         clearButton.addActionListener(e -> texto.setText(mensagemInicial));
76         buttonPanel.add(clearButton);
77
78         JButton printButton = createCustomButton("Imprimir", new Color(50, 175, 113));
79         printButton.addActionListener(e -> {
80             try {
81                 texto.print();
82             } catch (Exception ex) {
83                 JOptionPane.showMessageDialog(null, "Erro ao imprimir: " + ex.getMessage());
84             }
85         });
86         buttonPanel.add(printButton);
87
88         JButton saveButton = createCustomButton("Salvar", new Color(70, 130, 180));
89         saveButton.addActionListener(e -> salvarTexto());
90         buttonPanel.add(saveButton);
91
92         getContentPane().add(buttonPanel, BorderLayout.SOUTH);
93
94         // Torna a janela visível
95         setVisible(true);
96     }
97
98     /**
99      * Retorna a área de texto onde as mensagens são exibidas.
100      */
101     public JTextArea getTextArea() {
102         return this.texto;
103     }
104
105     public void adicionarMensagem(String mensagem) {
106         texto.append(mensagem + "\n");
107     }
108
109     private void salvarTexto() {
110         try {
111             JFileChooser fileChooser = new JFileChooser();
112             int option = fileChooser.showSaveDialog(this);
113             if (option == JFileChooser.APPROVE_OPTION) {
114                 FileWriter writer = new FileWriter(fileChooser.getSelectedFile().getAbsolutePath());
115                 writer.write(texto.getText());
116                 writer.close();
117                 JOptionPane.showMessageDialog(this, "Arquivo salvo com sucesso.");
118             }
119         } catch (IOException ex) {
120             JOptionPane.showMessageDialog(this, "Erro ao salvar arquivo: " + ex.getMessage());
121         }
122     }
123
124     /**
125      * Cria um botão personalizado com cores e bordas arredondadas.
126      */
127     private JButton createCustomButton(String text, Color backgroundColor) {
128         JButton button = new JButton(text);
129         button.setFont(new Font("Arial", Font.PLAIN, 14));
130         button.setForeground(Color.WHITE);
131         button.setBackground(backgroundColor);
132         button.setFocusPainted(false);
133         button.setBorder(BorderFactory.createCompoundBorder(
134             BorderFactory.createLineBorder(backgroundColor.darker(), 1),
135             BorderFactory.createEmptyBorder(5, 15, 5, 15)
136         ));
137         button.setCursor(new Cursor(Cursor.HAND_CURSOR));
138         return button;
139     }
140 }
```

- ✓ Definir a Thread de preenchimento assíncrono, com o nome ThreadClient, de acordo com as características apresentadas a seguir:
- ✓ Adicionar o atributo entrada, do tipo ObjectInputStream, e textArea, do tipo JTextArea, que devem ser preenchidos via construtor da Thread.
- ✓ Alterar o método run, implementando um loop de leitura contínua.
- ✓ Receber os dados enviados pelo servidor via método readObject.
- ✓ Para objetos do tipo String, apenas adicionar ao JTextArea.
- ✓ Para objetos do tipo List, acrescentar o nome e a quantidade de cada produto ao JTextArea.
- ✓ Observação! É necessário utilizar invokeLater nos acessos aos componentes do tipo Swing.
Segue Abaixo o Código ThreadClient Conforme Solicitado Acima.

Codigo ThreadClient:

```
1 package cadastroclient;
2
3 import java.awt.EventQueue;
4 import java.io.IOException;
5 import java.io.ObjectInputStream;
6 import java.math.BigDecimal;
7 import java.net.SocketException;
8 import java.text.NumberFormat;
9 import java.time.LocalDate;
10 import java.time.format.DateTimeFormatter;
11 import java.util.HashMap;
12 import java.util.List;
13 import java.util.Map;
14 import model.Produto;
15
16 /**
17  *
18  * @author Tiago J.P. Furtado
19  */
20 public class ThreadClient extends Thread {
21
22     private ObjectInputStream entrada;
23     private SaidasFrame saidasFrame;
24     private Map<String, BigDecimal> valoresTotaisAnteriores;
25     private String usuario;
26
27     public ThreadClient(ObjectInputStream entrada, SaidasFrame saidasFrame, String usuario) {
28         this.entrada = entrada;
29         this.saidasFrame = saidasFrame;
30         this.valoresTotaisAnteriores = new HashMap<>();
31         this.usuario = usuario;
32     }
33
34     @Override
35     public void run() {
36         try {
37             while (true) {
38                 try {
39                     // Recebe os dados enviados pelo servidor
40                     Object obj = entrada.readObject();
41
42                     // Se o objeto for do tipo String, apenas adiciona ao JTextArea
43                     if (obj instanceof String) {
44                         String mensagem = (String) obj;
45                         // Atualiza a interface gráfica na thread de eventos do Swing
46                         EventQueue.invokeLater(() -> {
47                             saidasFrame.adicionarMensagem(mensagem);
48                         });
49                     } // Se o objeto for do tipo List, presume-se que é uma lista de Produtos
50                     else if (obj instanceof List) {
51                         List<Produto> produtos = (List<Produto>) obj;
52                         // Atualiza a interface gráfica na thread de eventos do Swing
53                         EventQueue.invokeLater(() -> {
54                             // Exibe a mensagem de sucesso
55                             saidasFrame.adicionarMensagem("\n");
56                             saidasFrame.adicionarMensagem("Movimentação realizada com sucesso!");
57                             saidasFrame.adicionarMensagem("-----");
58                             saidasFrame.adicionarMensagem(String.format("%40s %15s %15s %20s", "Nome do Produto", "Quantidade", "Valor Unitário", "Valor Total"));
59                             saidasFrame.adicionarMensagem("-----");
60
61                             // Para cada produto na lista recebida
62                             for (Produto produto : produtos) {
63                                 // Formata o valor unitário
64                                 String valorUnitarioFormatado = formatarValor(produto.getPrecoVenda());
65                                 // Calcula o valor total (quantidade * valor unitário)
66                                 BigDecimal valorTotal = produto.getPrecoVenda().multiply(new BigDecimal(produto.getQuantidade()));
67                                 String valorTotalFormatado = formatarValor(valorTotal);
68
69                                 // Captura a data e hora atual
70                                 LocalDate dataHora = LocalDate.now();
71                                 DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
72                                 String dataHoraFormatada = dataHora.format(formatter);
73
74                                 // Verifica se houve alteração no valor total
75                                 BigDecimal valorTotalAnterior = valoresTotaisAnteriores.get(produto.getNome());
76                                 if (valorTotalAnterior == null || valorTotal.compareTo(valorTotalAnterior) != 0) {
77                                     // Exibe as informações no JTextArea com formatação para alinhamento
78                                     saidasFrame.adicionarMensagem(String.format("%40s %15s %15s %20s",
79                                         produto.getNome(),
80                                         produto.getQuantidade(),
81                                         valorUnitarioFormatado,
82                                         valorTotalFormatado));
83
84                                     saidasFrame.adicionarMensagem("-----");
85                                     saidasFrame.adicionarMensagem(String.format("Quantidade do produto '%s' alterada para %d.",
86                                         produto.getNome(),
87                                         produto.getQuantidade()));
88                                     saidasFrame.adicionarMensagem(String.format("Valor total do produto '%s' alterado para %s.",
89                                         produto.getNome(), valorTotalFormatado));
90                                     saidasFrame.adicionarMensagem(String.format("Alteração feita em: %s por %s",
91                                         dataHoraFormatada, usuario));
92                                     valoresTotaisAnteriores.put(produto.getNome(), valorTotal);
93                                     saidasFrame.adicionarMensagem("\n");
94                                     saidasFrame.adicionarMensagem("-----");
95                                 }
96                             }
97                         });
98                     }
99                 } catch (SocketException se) {
100                     // Socket foi fechado, termina o loop
101                     System.out.println("Conexão encerrada: " + se.getMessage());
102                     break;
103                 } catch (IOException | ClassNotFoundException e) {
104                     e.printStackTrace();
105                     break;
106                 }
107             }
108         } finally {
109             try {
110                 if (entrada != null) {
111                     entrada.close(); // Fecha o ObjectInputStream
112                 }
113             } catch (IOException e) {
114                 e.printStackTrace();
115             }
116         }
117     }
118
119     // Método auxiliar para formatar o valor como R$
120     private String formatarValor(BigDecimal valor) {
121         NumberFormat formatador = NumberFormat.getCurrencyInstance();
122         return formatador.format(valor);
123     }
124 }
125
```

Com o projeto CadastroServer **em execução**, iniciar o sistema do cliente, e testar todas as funcionalidades oferecidas.

Menu: Com as Opções Listar – Entrada – Saída - Finalizar

```
=====
|                               |
|           Menu de Opções     |
|                               |
| [L] Listar                   |
| [E] Entrada                  |
| [S] Saída                    |
| [X] Finalizar                |
|                               |
|=====|
Escolha uma opção: E
Digite o ID da pessoa: 5
Digite o ID do produto: 1
Digite a quantidade: 100
Digite o valor unitário: 100

=====
|                               |
|           Menu de Opções     |
|                               |
| [L] Listar                   |
| [E] Entrada                  |
| [S] Saída                    |
| [X] Finalizar                |
|                               |
|=====|
Escolha uma opção: L
```

Janela do SaidaFrame:

📄 Controle de Movimentação

=====

Movimentação realizada com sucesso!

=====

Nome do Produto	Quantidade	Valor Unitário	Valor Total
Cubo de Roda	255	R\$ 60,00	R\$ 15.300,00

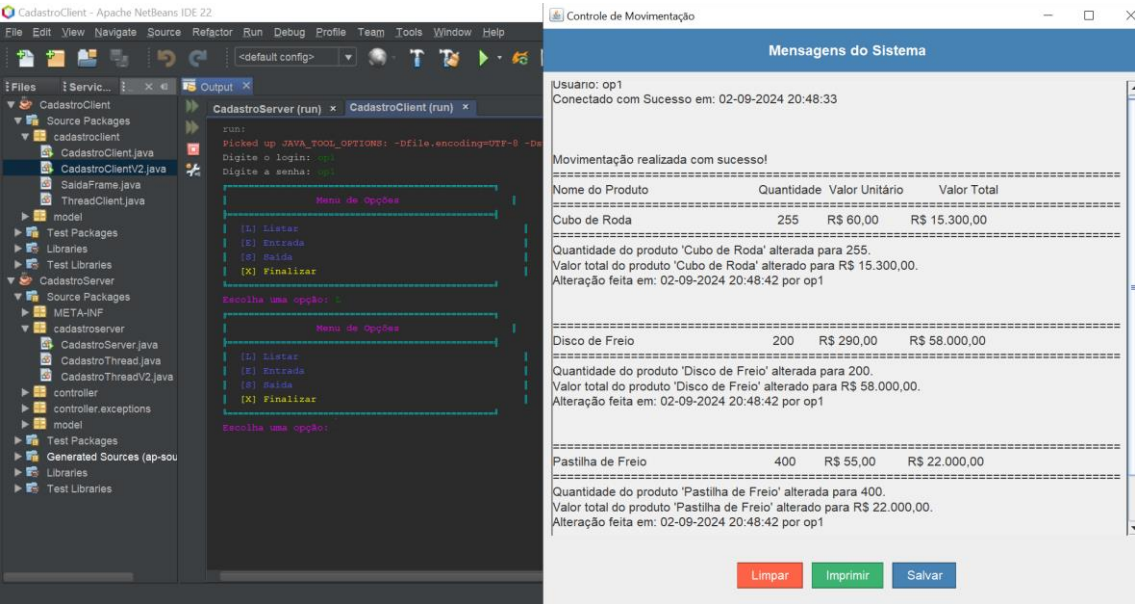
=====

Quantidade do produto 'Cubo de Roda' alterada para 255.
Valor total do produto 'Cubo de Roda' alterado para R\$ 15.300,00.
Alteração feita em: 01-09-2024 08:53:24 por op1

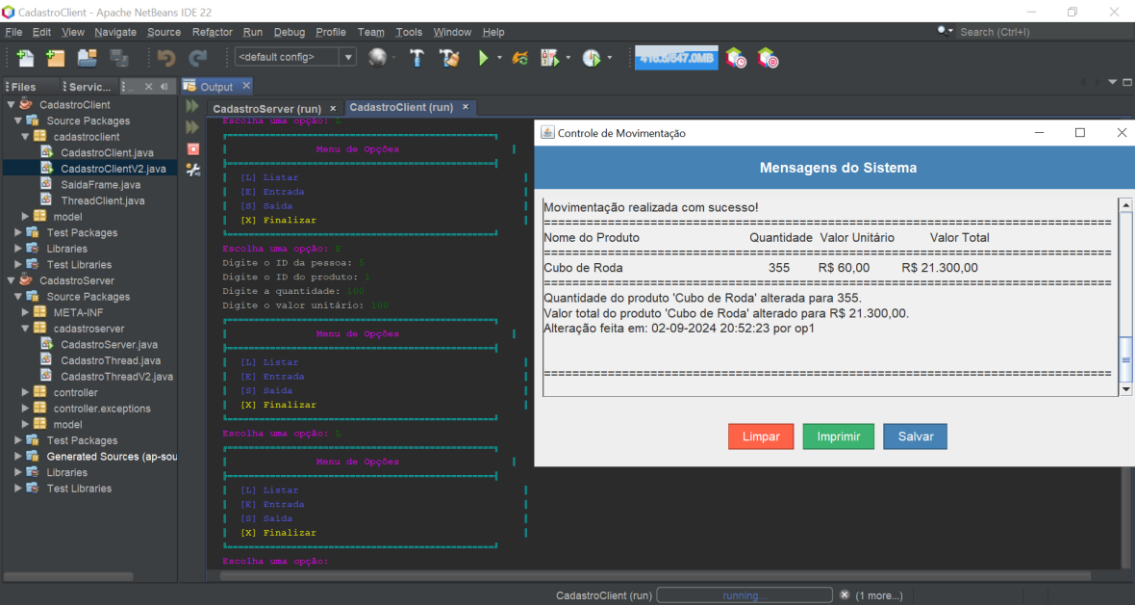
=====

Os resultados da execução dos códigos também devem ser apresentados:

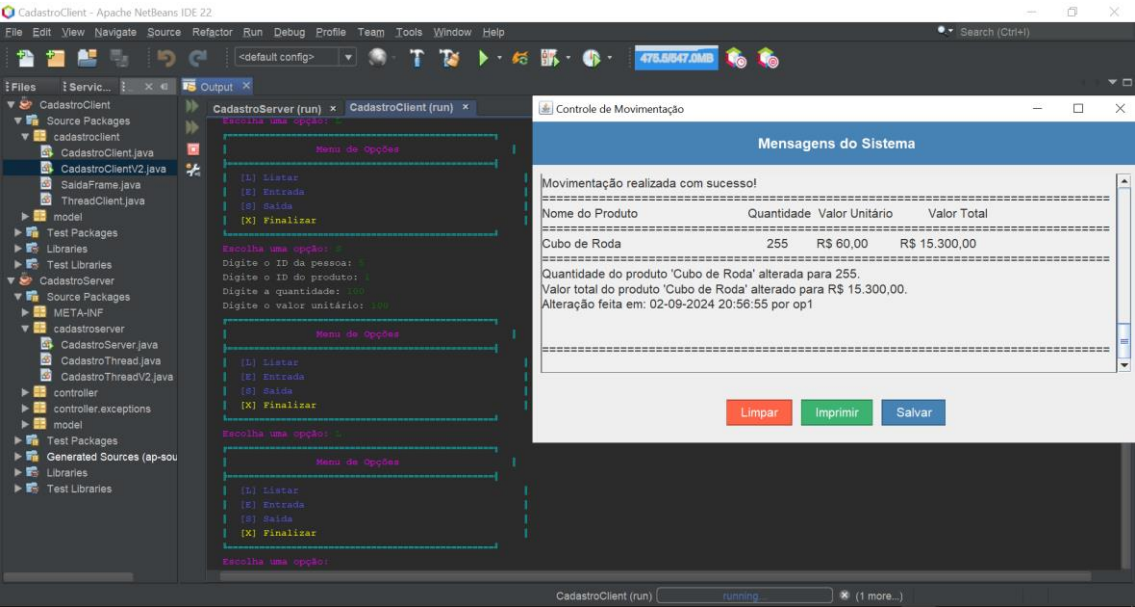
Opção Lista:



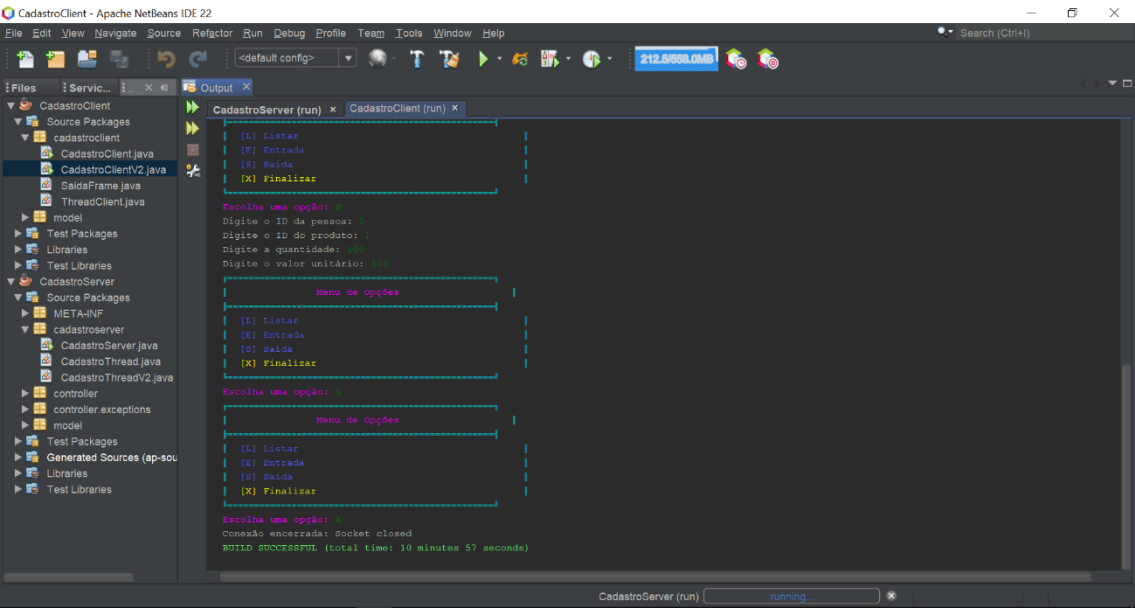
Opção Entrada:



Opção Saída:



Opção Finalizar:



5- Análise e Conclusão:

❖ Como as Threads podem ser utilizadas para o tratamento assíncrono das respostas enviadas pelo servidor?

Em Java, as **Threads** são frequentemente usadas para realizar operações de forma assíncrona, permitindo que uma aplicação continue executando outras tarefas enquanto espera por uma resposta do servidor. Aqui está uma visão geral de como isso pode ser feito:

Criação de Threads para Processamento Assíncrono:

- ✓ Em Java, você pode criar uma nova thread para processar a resposta de uma solicitação ao servidor, enquanto a thread principal da aplicação continua a executar outras tarefas.
- ✓ Isso é útil em aplicações que precisam realizar múltiplas operações de I/O (como comunicação com o servidor), onde o bloqueio da thread principal enquanto espera por uma resposta poderia afetar o desempenho e a responsividade.

Implementação de Threads em Java:

- ✓ Existem duas maneiras principais de criar threads em Java: estendendo a classe Thread ou implementando a interface Runnable. A segunda abordagem é mais flexível, pois permite que a classe implemente outras interfaces ou estenda outra classe.

Exemplo de Implementação:

- ✓ Abaixo está um exemplo de como usar a interface Runnable para processar respostas de um servidor de forma assíncrona.

```
1 // Classe que implementa a interface Runnable
2 class ResponseProcessor implements Runnable {
3     private String response;
4
5     public ResponseProcessor(String response) {
6         this.response = response;
7     }
8
9     @Override
10    public void run() {
11        // Simulação do processamento da resposta
12        System.out.println("Processando a resposta: " + response);
13        // Aqui você pode adicionar o código para realmente processar a resposta do servidor
14    }
15 }
16
17 public class Main {
18     public static void main(String[] args) {
19         // Simulação de respostas recebidas do servidor
20         String[] responses = {"resposta1", "resposta2", "resposta3"};
21
22         // Para cada resposta, criamos e iniciamos uma nova thread
23         for (String response : responses) {
24             ResponseProcessor processor = new ResponseProcessor(response);
25             Thread thread = new Thread(processor);
26             thread.start();
27         }
28
29         System.out.println("Todas as threads foram iniciadas. A aplicação continua rodando...");
30         // A aplicação continua rodando enquanto as respostas são processadas em threads separadas
31     }
32 }
33
```

Utilizando Thread Pools:

- ✓ Criar uma nova thread para cada resposta pode ser ineficiente, especialmente em sistemas que recebem muitas respostas. Em vez disso, você pode usar um **thread pool** com a classe `ExecutorService` para gerenciar um número fixo de threads reutilizáveis.

```
1 import java.util.concurrent.ExecutorService;
2 import java.util.concurrent.Executors;
3
4 public class Main {
5     Run | Debug
6     public static void main(String[] args) {
7         // Criando um pool de threads com 3 threads
8         ExecutorService executor = Executors.newFixedThreadPool(3);
9
10        // Simulação de respostas recebidas do servidor
11        String[] responses = {"resposta1", "resposta2", "resposta3"};
12
13        // Para cada resposta, submetemos uma tarefa ao pool de threads
14        for (String response : responses) {
15            executor.submit(new ResponseProcessor(response));
16        }
17
18        // Finalizando o executor (não aceitará mais novas tarefas)
19        executor.shutdown();
20
21        System.out.println("Todas as tarefas foram submetidas ao pool. A aplicação continua rodando...");
22    }
23 }
```

Tratamento de Exceções e Sincronização:

- ✓ Ao trabalhar com threads, é importante gerenciar exceções adequadamente e garantir que os recursos compartilhados sejam sincronizados para evitar problemas como condições de corrida.

Benefícios:**Eficiência:**

- ✓ O uso de threads permite que múltiplas respostas sejam processadas simultaneamente, melhorando a eficiência.

Responsividade:

- ✓ A aplicação continua responsiva, mesmo enquanto espera pelas respostas do servidor.
- ✓ Estas práticas são comuns em aplicações de rede em Java, onde a comunicação com o servidor pode levar um tempo variável e onde manter a interface do usuário ou outras operações não bloqueadas é crucial.

❖ Para que serve o método `invokeLater`, da classe `SwingUtilities`?

- ✓ O método `invokeLater` da classe `SwingUtilities` em Java é usado para garantir que o código relacionado à interface gráfica (GUI) seja executado na **Event Dispatch Thread (EDT)**.

Event Dispatch Thread (EDT):

- ✓ Em aplicações Java que utilizam Swing para criar interfaces gráficas, a **Event Dispatch Thread (EDT)** é uma thread especial dedicada ao gerenciamento de eventos e à atualização da interface gráfica.
- ✓ Todas as operações que modificam a GUI (como atualizar componentes visuais) devem ser feitas na EDT para evitar condições de corrida e inconsistências na interface.

Por que utilizar `invokeLater`?

- ✓ Se uma operação que altera a GUI for executada fora da EDT (por exemplo, em uma thread de background ou na thread principal da aplicação), isso pode levar a problemas como congelamento da interface ou comportamentos imprevisíveis.
- ✓ O método `invokeLater` garante que o código fornecido será executado na EDT assim que possível, permitindo que a interface gráfica seja atualizada de forma segura e consistente.

Como o `invokeLater` funciona:

- ✓ Quando você chama `SwingUtilities.invokeLater(Runnable doRun)`, você está dizendo ao sistema para enfileirar o `Runnable` fornecido para execução na EDT.
- ✓ Isso significa que, mesmo que você esteja em outra thread, a ação que você deseja realizar na GUI será executada na thread correta.

Exemplo de Uso:

```
1  import javax.swing.SwingUtilities;
2
3  public class Example {
4      public static void main(String[] args) {
5          // Simulação de uma operação em uma thread separada
6          new Thread(new Runnable() {
7              @Override
8              public void run() {
9                  // Alguma lógica que não está na EDT
10                 System.out.println(x:"Executando lógica em thread separada.");
11
12                 // Atualizando a GUI de forma segura usando invokeLater
13                 SwingUtilities.invokeLater(new Runnable() {
14                     @Override
15                     public void run() {
16                         System.out.println(x:"Atualizando GUI na Event Dispatch Thread.");
17                         // Aqui você colocaria o código para atualizar a interface gráfica
18                     }
19                 });
20             }
21         }).start();
22     }
23 }
24
```

Quando Usar:

Atualização de componentes Swing:

- ✓ Sempre que você precisar alterar a interface gráfica de dentro de um código que não está sendo executado na EDT, como em uma operação de background.

Evitar erros e travamentos:

- ✓ Garantir que todas as modificações na GUI sejam feitas de forma segura.

6. Resumo:

- ✓ **invokeLater** é essencial para garantir que a interface gráfica seja manipulada corretamente, evitando problemas de concorrência e garantindo que a GUI responda conforme o esperado.

❖ Como os objetos são enviados e recebidos pelo Socket Java?

Em Java, os objetos podem ser enviados e recebidos através de sockets utilizando a serialização. A serialização é o processo de converter um objeto em um formato que pode ser facilmente transmitido através de uma rede e, posteriormente, reconstruído no outro lado. A seguir, explico como isso funciona:

Preparação para a Serialização:

- ✓ Para que um objeto seja enviado através de um socket, a classe do objeto deve implementar a interface `Serializable`. Isso indica que a classe pode ser convertida em um fluxo de bytes.
- ✓ Todas as propriedades do objeto (ou seja, os atributos) também devem ser serializáveis. Caso alguma não deva ser serializada, pode-se usar a palavra-chave `transient` para excluí-la do processo de serialização.

Envio de Objetos via Sockets:

- ✓ O envio de objetos através de sockets é feito usando o `ObjectOutputStream`, que converte o objeto em um fluxo de bytes e o envia pelo `OutputStream` do socket.

```
1  import java.io.ObjectOutputStream;
2  import java.io.OutputStream;
3  import java.net.Socket;
4
5  public class Cliente {
6      Run | Debug
7      public static void main(String[] args) {
8          try {
9              // Conectando ao servidor na porta 1234
10             Socket socket = new Socket("localhost", port:1234);
11
12             // Criando um objeto para enviar
13             MeuObjeto obj = new MeuObjeto(nome:"Exemplo", valor:42);
14
15             // Enviando o objeto
16             OutputStream outputStream = socket.getOutputStream();
17             ObjectOutputStream objectOutputStream = new ObjectOutputStream(outputStream);
18             objectOutputStream.writeObject(obj);
19             objectOutputStream.flush();
20
21             System.out.println("Objeto enviado ao servidor.");
22
23             // Fechando o socket
24             objectOutputStream.close();
25             socket.close();
26         } catch (Exception e) {
27             e.printStackTrace();
28         }
29     }
30
31     class MeuObjeto implements java.io.Serializable {
32         private String nome;
33         private int valor;
34
35         public MeuObjeto(String nome, int valor) {
36             this.nome = nome;
37             this.valor = valor;
38         }
39     }
40 }
```

Recebimento de Objetos via Sockets:

- ✓ Para receber objetos através de um socket, utilizamos o `ObjectInputStream`, que converte o fluxo de bytes de volta no objeto original.

```
1  import java.io.ObjectInputStream;
2  import java.net.ServerSocket;
3  import java.net.Socket;
4
5  public class Servidor {
6      public static void main(String[] args) {
7          try {
8              // Criando um servidor socket na porta 1234
9              ServerSocket serverSocket = new ServerSocket(port:1234);
10             System.out.println("Servidor aguardando conexão...");
11
12             // Aceitando a conexão do cliente
13             Socket socket = serverSocket.accept();
14
15             // Recebendo o objeto
16             ObjectInputStream objectInputStream = new ObjectInputStream(socket.getInputStream());
17             MeuObjeto objRecebido = (MeuObjeto) objectInputStream.readObject();
18
19             System.out.println("Objeto recebido: " + objRecebido);
20
21             // Fechando os streams e o socket
22             objectInputStream.close();
23             socket.close();
24             serverSocket.close();
25         } catch (Exception e) {
26             e.printStackTrace();
27         }
28     }
29 }
30
```

Considerações Importantes:

- ✓ **Compatibilidade de versão:** As classes do objeto devem ser compatíveis entre o cliente e o servidor. Caso haja uma mudança na classe, como adicionar ou remover campos, pode ser necessário ajustar a serialização usando o campo `serialVersionUID`.
- ✓ **Segurança:** Evite desserializar objetos de fontes não confiáveis, pois isso pode permitir ataques de desserialização maliciosa.

Resumo:

- ✓ **Enviar objetos:** Use `ObjectOutputStream` para serializar e enviar o objeto pelo socket.
- ✓ **Receber objetos:** Use `ObjectInputStream` para desserializar e reconstruir o objeto a partir do fluxo de bytes recebido.
- ✓ Essa abordagem é amplamente utilizada em aplicações Java para comunicação entre clientes e servidores, especialmente em sistemas distribuídos.

❖ Compare a utilização de comportamento assíncrono ou síncrono nos clientes com Socket Java, ressaltando as características relacionadas ao bloqueio do processamento.

A escolha entre o uso de comportamento assíncrono ou síncrono ao lidar com sockets em Java tem implicações significativas em termos de desempenho, responsividade e complexidade de implementação. Abaixo, comparo essas duas abordagens, destacando as características relacionadas ao bloqueio do processamento.

Comportamento Síncrono:

- ✓ **Descrição:** No comportamento síncrono, o código é executado sequencialmente, ou seja, cada operação de I/O (entrada/saída) no socket deve ser concluída antes que a próxima operação seja iniciada. Isso significa que a thread que realiza a operação fica bloqueada até que a operação seja concluída.
- ✓ **Exemplo:** Um cliente que envia uma solicitação ao servidor e aguarda a resposta. Durante esse tempo de espera, a thread está bloqueada.

Características:

- ✓ **Bloqueio de Processamento:** Quando uma thread está bloqueada esperando uma resposta do servidor, ela não pode realizar outras tarefas. Isso pode levar ao congelamento da interface do usuário em aplicações com interface gráfica ou ao uso ineficiente dos recursos em servidores que tratam múltiplos clientes.
- ✓ **Simplicidade:** A programação síncrona é geralmente mais simples de entender e implementar, uma vez que o fluxo de controle é linear.

Exemplo de Uso:

```
1
2 Socket socket = new Socket("localhost", 1234);
3 InputStream inputStream = socket.getInputStream();
4 // A thread fica bloqueada aqui até que os dados sejam recebidos
5 int data = inputStream.read();
6
7
```


Comportamento Assíncrono:

- ✓ **Descrição:** No comportamento assíncrono, as operações de I/O não bloqueiam a thread que as inicia. Em vez disso, o código pode continuar a execução enquanto a operação de I/O é realizada em segundo plano. Em Java, isso pode ser implementado usando múltiplas threads ou utilizando recursos da API de NIO (Non-blocking I/O).
- ✓ **Exemplo:** Um cliente que envia uma solicitação ao servidor e, enquanto aguarda a resposta, continua processando outras tarefas.

Características:

- ✓ **Não Bloqueio de Processamento:** Como as operações de I/O são executadas de forma não bloqueante, a aplicação pode continuar processando outras tarefas enquanto aguarda a conclusão das operações de rede. Isso é especialmente útil para aplicações que precisam manter a responsividade ou para servidores que tratam múltiplos clientes simultaneamente.
- ✓ **Complexidade:** A programação assíncrona é mais complexa, pois exige o gerenciamento de múltiplas threads ou a utilização de callbacks e futures para lidar com a conclusão de operações de I/O.
- ✓ **Uso de Múltiplas Threads:** Uma abordagem comum é criar uma thread separada para lidar com a comunicação via socket, permitindo que a thread principal continue a executar outras tarefas.

Exemplo de Uso com ExecutorService:

```
1  ExecutorService executor = Executors.newCachedThreadPool();
2  executor.submit(() -> {
3      try (Socket socket = new Socket("localhost", 1234)) {
4          InputStream inputStream = socket.getInputStream();
5          int data = inputStream.read(); // Não bloqueia a thread principal
6          System.out.println("Dados recebidos: " + data);
7      } catch (IOException e) {
8          e.printStackTrace();
9      }
10 });
11 // A thread principal pode continuar a fazer outras tarefas
12 System.out.println("Thread principal continua...");
```

Comparação em Termos de Bloqueio de Processamento:

Síncrono:

- ✓ **Bloqueio:** A thread que realiza a operação de I/O fica bloqueada até a conclusão, o que pode levar a um uso ineficiente dos recursos e uma menor responsividade.
- ✓ **Adequado para:** Aplicações simples ou aquelas em que o bloqueio não prejudica a performance ou a experiência do usuário.

Assíncrono:

- ✓ Não Bloqueio: A thread pode continuar a execução enquanto espera pela conclusão da operação de I/O, resultando em melhor uso dos recursos e maior responsividade.
- ✓ **Adequado para:** Aplicações que precisam ser altamente responsivas, como interfaces gráficas e servidores que tratam múltiplas conexões simultâneas.

Considerações Finais:

- ✓ A escolha entre síncrono e assíncrono deve ser baseada nos requisitos específicos da aplicação. Para tarefas que exigem alta responsividade ou para servidores que precisam gerenciar múltiplas conexões ao mesmo tempo, o comportamento assíncrono geralmente é preferível, apesar da maior complexidade. Para aplicações mais simples, o comportamento síncrono pode ser mais fácil de implementar e manter.