 <b>Estácio</b>	Universidade Estácio Campus Castelo Curso de Desenvolvimento Full Stack Relatório da Missão Prática 5 - Mundo 3
Disciplina:	RPG0018 - Por Que Não Paralelizar?
Nome:	Breno Ambrosim Louzada
Turma:	2023.1

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

#### 1. Título da Prática: “1º Procedimento | Criando o Servidor e Cliente de Teste”

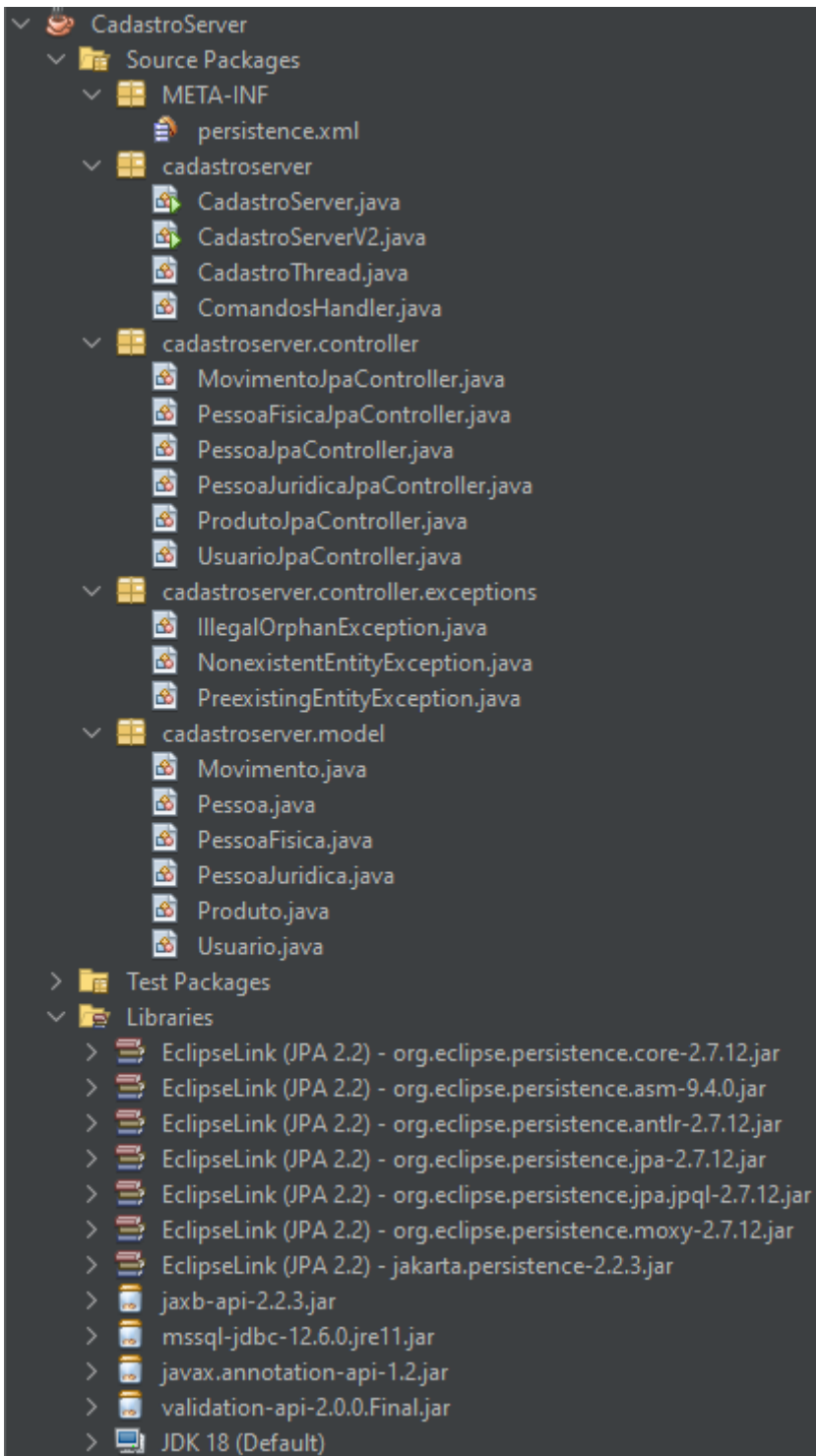
#### 2. Objetivo da Prática

- 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.

#### 3. Códigos solicitados: estão disponíveis no repositório

<https://github.com/BrenoAmbrosim/Mundo-3-N-vel-5>

#### 4. Resultados da execução dos códigos



```

public class CadastroServer {

    private final int PORT = 4321;

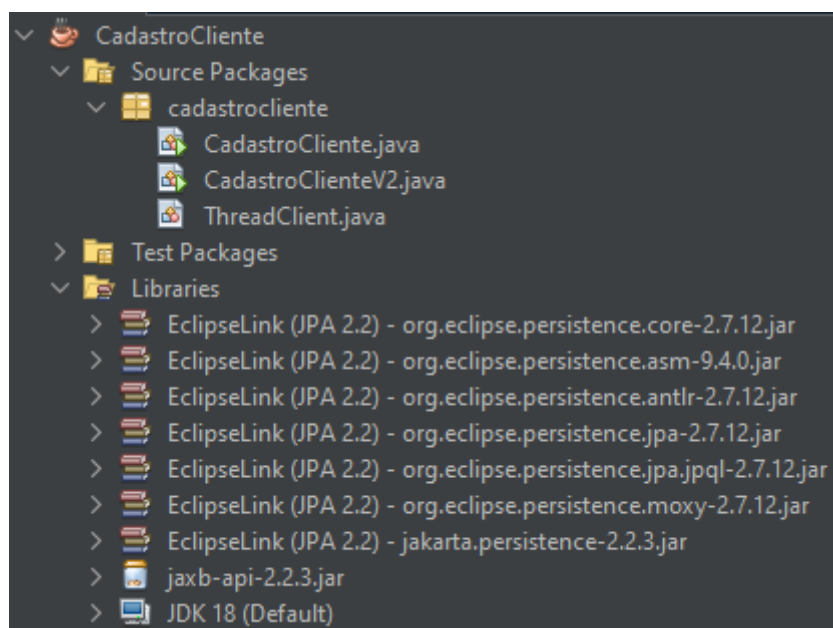
    public CadastroServer() {

    }

    private void run() {
        try (ServerSocket serverSocket = new ServerSocket(port: PORT)) {
            System.out.println("===== SERVIDOR CONECTADO - PORTA " + PORT + " =====");
            // Inicializa controladores
            EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnitName: "CadastroServerPU");
            ProdutoJpaController produtoController = new ProdutoJpaController(emf);
            MovimentoJpaController movimentoController = new MovimentoJpaController(emf);
            PessoaJpaController pessoaController = new PessoaJpaController(emf);
            UsuarioJpaController usuarioController = new UsuarioJpaController(emf);
            while (true) {
                System.out.println(x: "Aguardando conexao de cliente...");
                Socket socket = serverSocket.accept();
                System.out.println(x: "Cliente conectado.");
                ClientHandler clientHandler = new ClientHandler(socket, produtoController,
                    movimentoController, pessoaController, usuarioController);
                Thread thread = new Thread(target: clientHandler);
                thread.start();
            }
        } catch (IOException e) {
            Logger.getLogger(name: CadastroServer.class.getName()).log(level: Level.SEVERE, msg: null, thrown: e);
        }
    }

    public static void main(String[] args) {
        new CadastroServer().run();
    }
}

```



```

public class CadastroCliente {

    private final String SERVER_ADDRESS = "localhost";
    private final int SERVER_PORT = 4321;

    public CadastroCliente() {

    }

    private void run() {
        try (Socket socket = new Socket(host: SERVER_ADDRESS, port: SERVER_PORT);
            BufferedReader in = new BufferedReader(new InputStreamReader(in: socket.getInputStream()));
            PrintWriter out = new PrintWriter(out: socket.getOutputStream(), autoFlush: true);
            BufferedReader consoleIn = new BufferedReader(new InputStreamReader(in: System.in))) {
            System.out.println(x: "Conectado ao servidor CadastroServer.");
            System.out.print(s: "Digite seu nome de usuario: ");
            String username = consoleIn.readLine();
            System.out.print(s: "Digite sua senha: ");
            String password = consoleIn.readLine();
            out.println(x: username);
            out.println(x: password);
            String response = in.readLine();
            System.out.println(x: response);
            if (response.equals(anObject: "Autenticacao bem-sucedida. Aguardando comandos...")) {
                boolean exitChoice = false;
                while (!exitChoice) {
                    System.out.print(s: "Digite 'L' para listar produtos ou 'S' para sair: ");
                    String command = consoleIn.readLine().toUpperCase();
                    out.println(x: command);
                    switch (command) {
                        case "S" → exitChoice = true;
                        case "L" → receiveAndDisplayProductList(in);
                        default → System.out.println(x: "Opcao invalida!");
                    }
                }
            }
        } catch (IOException e) {
            Logger.getLogger(name: CadastroCliente.class.getName()).log(level: Level.SEVERE, msg: null, thrown: e);
        }
    }
}

```

```

Output x
GlassFish Server x  CadastroServer (run) x  CadastroCliente (run) x
run:
Conectado ao servidor CadastroServer.
Digite seu nome de usuario: op1
Digite sua senha: op1
Autenticacao bem-sucedida. Aguardando comandos...
Digite 'L' para listar produtos ou 'S' para sair: L
Conjunto de produtos disponiveis:
Banana
Laranja
Manga
Digite 'L' para listar produtos ou 'S' para sair: S
BUILD SUCCESSFUL (total time: 12 seconds)

```

## 5. Análise e Conclusão

1. Como funcionam as classes Socket e ServerSocket? As classes Socket e ServerSocket são fundamentais para a programação de rede em Java. A classe Socket é usada para criar um ponto de conexão (socket) para comunicação entre dois dispositivos em uma rede. Quando um cliente quer estabelecer uma conexão com um servidor, ele cria um Socket e tenta se conectar a um ServerSocket que está escutando em um endereço IP e porta específicos. Por outro lado, a classe ServerSocket é usada no lado do servidor para escutar e aceitar conexões de clientes. Quando um ServerSocket aceita uma conexão, ele retorna um novo Socket que é usado para a comunicação com o cliente específico. Assim, essa combinação de Socket e ServerSocket permite a troca de dados entre clientes e servidores em uma rede.

2. Qual a importância das portas para a conexão com servidores? As portas são essenciais na comunicação de rede, funcionando como pontos de extremidade em um sistema operacional para diferenciar o tráfego de rede destinado a diferentes serviços ou aplicações. Quando um cliente estabelece uma conexão com um servidor, ele precisa saber não apenas o endereço IP do servidor, mas também a porta específica na qual o serviço desejado está escutando. Isso permite que múltiplos serviços, como servidores web, de email, ou de banco de dados, operem simultaneamente no mesmo servidor físico sem interferência, pois cada serviço escuta em uma porta diferente. As portas, portanto, são fundamentais para organizar o tráfego de rede e garantir que as comunicações sejam encaminhadas corretamente para os serviços apropriados.

3. Para que servem as classes de entrada e saída ObjectOutputStream e ObjectInputStream, e por que os objetos transmitidos devem ser serializáveis? As classes ObjectOutputStream e ObjectInputStream em Java são usadas para realizar a serialização e deserialização de objetos, respectivamente, permitindo que eles sejam transmitidos através de streams, como sockets ou arquivos. ObjectOutputStream converte um objeto em uma sequência de bytes (serialização) que pode ser enviada por um stream, enquanto ObjectInputStream reconstrói o objeto a partir dessa sequência de bytes (deserialização). Para que um objeto seja elegível para essa transmissão, ele deve ser serializável, o que é indicado pela implementação da interface Serializable. Isso é necessário porque apenas objetos serializáveis podem ser convertidos de maneira confiável em um formato que pode ser enviado e reconstruído em outro local ou momento, mantendo a integridade e o estado dos dados do objeto.

4. Com Por que, mesmo utilizando as classes de entidades JPA no cliente, foi possível garantir o isolamento do acesso ao banco de dados? O isolamento do acesso ao banco de dados, mesmo utilizando classes de entidades JPA no cliente, é garantido porque a JPA, como parte da especificação Java EE, é projetada para separar claramente as operações de lógica de negócios da camada de persistência de dados. As classes de entidade JPA representam a estrutura dos dados e são usadas tanto no cliente quanto no servidor, mas a gestão real do acesso ao banco de dados é realizada exclusivamente no lado do servidor. Isso significa que as operações de banco de dados - como consultas, inserções, atualizações e exclusões - são executadas por meio de sessões e transações gerenciadas pelo servidor, geralmente através de um provedor de persistência como o Hibernate. Portanto, mesmo que o cliente use as mesmas classes de entidade, ele não tem acesso direto ao banco de dados, garantindo o isolamento e a segurança dos dados.

## 1. Título da Prática: “2º Procedimento | Servidor Completo e Cliente Assíncrono”

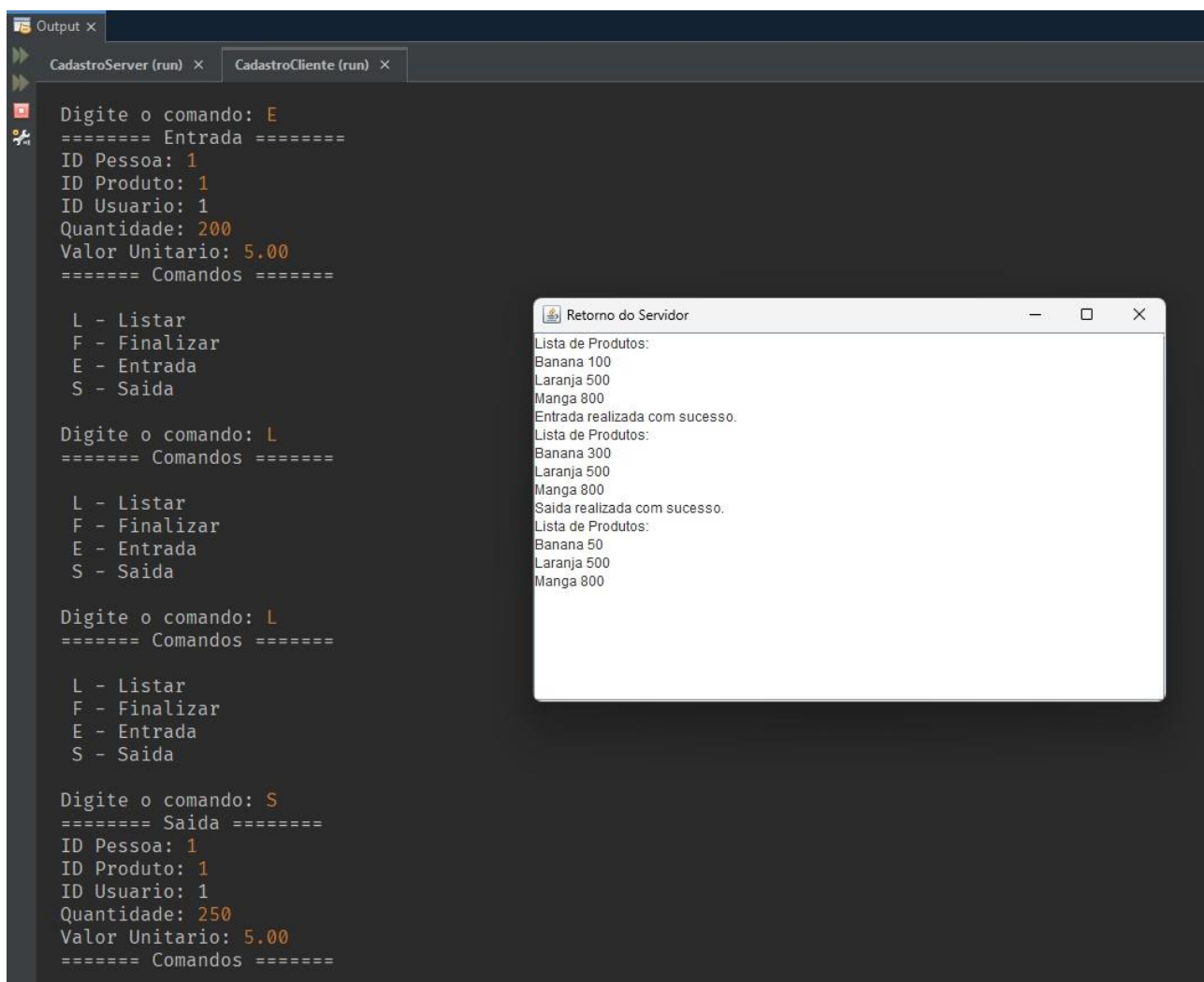
## 2. Objetivo da Prática

- 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.

## 3. Códigos solicitados: estão disponíveis no repositório

<https://github.com/BrenoAmbrosim/Mundo-3-N-vel-5>

## 4. Resultados da execução dos códigos



```
Output x
CadastroServer (run) x CadastroCliente (run) x

Digite o comando: E
===== Entrada =====
ID Pessoa: 1
ID Produto: 1
ID Usuario: 1
Quantidade: 200
Valor Unitario: 5.00
===== Comandos =====

L - Listar
F - Finalizar
E - Entrada
S - Saida

Digite o comando: L
===== Comandos =====

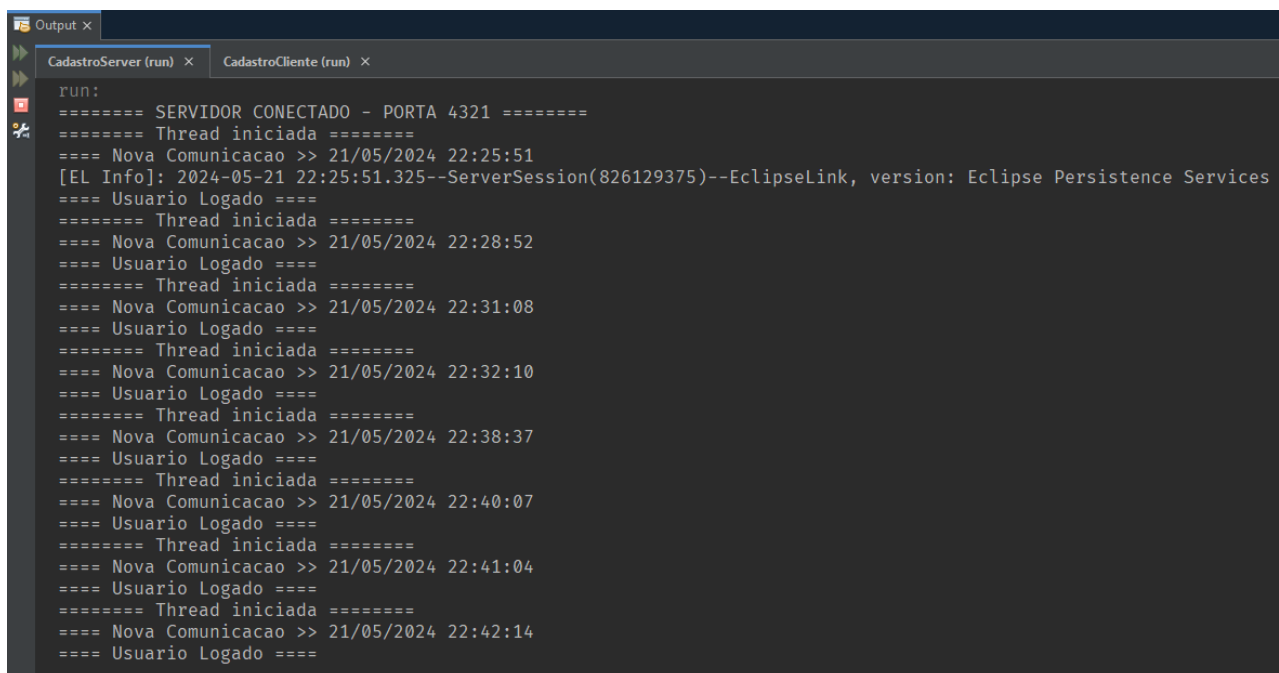
L - Listar
F - Finalizar
E - Entrada
S - Saida

Digite o comando: L
===== Comandos =====

L - Listar
F - Finalizar
E - Entrada
S - Saida

Digite o comando: S
===== Saida =====
ID Pessoa: 1
ID Produto: 1
ID Usuario: 1
Quantidade: 250
Valor Unitario: 5.00
===== Comandos =====

Retorno do Servidor
Lista de Produtos:
Banana 100
Laranja 500
Manga 800
Entrada realizada com sucesso.
Lista de Produtos:
Banana 300
Laranja 500
Manga 800
Saida realizada com sucesso.
Lista de Produtos:
Banana 50
Laranja 500
Manga 800
```



```
run:
===== SERVIDOR CONECTADO - PORTA 4321 =====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:25:51
[EL Info]: 2024-05-21 22:25:51.325--ServerSession(826129375)--EclipseLink, version: Eclipse Persistence Services
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:28:52
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:31:08
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:32:10
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:38:37
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:40:07
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:41:04
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 21/05/2024 22:42:14
==== Usuario Logado =====
```

## 5. Análise e Conclusão

1. Como as Threads podem ser utilizadas para o tratamento assíncrono das respostas enviadas pelo servidor? As Threads são essenciais para o tratamento assíncrono de respostas em aplicações cliente-servidor. Quando um servidor recebe múltiplas solicitações de clientes, ele pode utilizar Threads para processar cada solicitação simultaneamente, evitando assim o bloqueio enquanto espera por uma operação de longa duração em uma única solicitação. Cada solicitação é processada em uma Thread separada, permitindo que o servidor continue recebendo e respondendo a outras solicitações. Essa abordagem melhora a eficiência e a capacidade de resposta do servidor, pois as Threads operam de forma independente e paralela, garantindo que o processamento de uma solicitação não seja interrompido ou atrasado pelas outras. Em resumo, o uso de Threads no servidor possibilita o tratamento assíncrono das respostas, melhorando o desempenho geral do sistema em ambientes de rede.

2. Para que serve o método `invokeLater`, da classe `SwingUtilities`? O método `invokeLater` da classe `SwingUtilities` é usado na programação de interfaces gráficas em Java, especificamente no Swing framework. Ele é essencial para garantir que as alterações na interface do usuário sejam feitas de maneira segura no contexto da Event Dispatch Thread (EDT), que é a thread responsável por gerenciar eventos e atualizações de interface gráfica em Swing. Quando você quer executar um bloco de código que altera a interface do usuário, como atualizar um componente gráfico ou responder a eventos de usuário, `invokeLater` é utilizado para enfileirar esse bloco de código na EDT. Isso assegura que as mudanças na interface sejam realizadas de maneira sequencial e segura, evitando problemas de concorrência e inconsistências na interface gráfica.

3. Como os objetos são enviados e recebidos pelo Socket Java? No Java, objetos são enviados e recebidos através de sockets utilizando fluxos de objetos, especificamente as classes `ObjectOutputStream` e `ObjectInputStream`. Para enviar um objeto, o programa primeiro serializa o objeto (converte-o em uma sequência de bytes) usando `ObjectOutputStream` e depois o envia através do Socket. No lado receptor, `ObjectInputStream` é usado para ler a sequência de bytes do Socket e desserializá-la (reconstruir o objeto original). É crucial que o objeto a ser enviado implemente a interface `Serializable`, que permite essa serialização e desserialização. Essa abordagem permite a transmissão de objetos complexos e suas propriedades através da rede de forma eficiente, mantendo a integridade e o estado dos objetos.

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

com Sockets em Java, o comportamento síncrono e assíncrono tem implicações significativas no bloqueio do processamento. Operações síncronas bloqueiam a execução do programa até que a operação de rede seja concluída; por exemplo, um método `read()` em um Socket síncrono pausará a execução até que os dados sejam recebidos. Isso pode simplificar a lógica de programação, mas pode levar à ineficiência, pois o thread que executa a operação fica inativo enquanto espera. Por outro lado, um comportamento assíncrono, frequentemente implementado através de callbacks ou futures, permite que o programa continue executando outras tarefas enquanto aguarda a conclusão da operação de rede. Isso aumenta a eficiência e a capacidade de resposta do aplicativo, pois os recursos de processamento podem ser utilizados para outras tarefas durante as operações de espera, mas pode tornar a lógica do programa mais complexa devido ao gerenciamento das operações não lineares.