



Curso:

**Desenvolvimento Full Stack**

Campus:

**POLO JARDIM BRASÍLIA - ÁGUAS LINDAS DE GOIÁS - GO**

Disciplina:

**Por que não paralelizar**

Turma:

**23.2**

Aluno:

**BRUNO SANTIAGO DE OLIVEIRA**

## Missão Prática | Nível 5 | Mundo 3

### Cadastro tread

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.List;

public class CadastroThread extends Thread {
    private final Socket socket;
    private final ProdutoJpaController ctrl;
    private final UsuarioJpaController ctrlUsu;

    public CadastroThread(Socket socket, ProdutoJpaController ctrl,
        UsuarioJpaController ctrlUsu) {
        this.socket = socket;
        this.ctrl = ctrl;
        this.ctrlUsu = ctrlUsu;
    }

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

            // Obter login e senha do cliente
            String login = (String) in.readObject();
            String senha = (String) in.readObject();

            // Verificar as credenciais com o controlador de usuários
            Usuario usuario = ctrlUsu.findUsuario(login, senha);
```

```

if (usuario == null) {
    // Se as credenciais forem inválidas, encerrar a conexão
    System.out.println("Credenciais inválidas. Desconectando cliente.");
    return;
}

// Loop de resposta
while (true) {
    String comando = (String) in.readObject();

    if (comando.equals("L")) {
        // Se o comando for 'L', retornar o conjunto de produtos
        List<Produto> produtos = ctrl.listarProdutos();
        out.writeObject(produtos);
    } else if (comando.equals("C")) {
        // Se o comando for 'C', criar um novo produto
        String nomeProduto = (String) in.readObject();
        double precoProduto = (Double) in.readObject();

        // Criar o novo produto com os dados recebidos
        Produto novoProduto = new Produto();
        novoProduto.setNome(nomeProduto);
        novoProduto.setPreco(precoProduto);

        // Salvar o novo produto no banco de dados
        ctrl.create(novoProduto);
    } else if (comando.equals("D")) {
        // Se o comando for 'D', excluir um produto por ID
        int idProduto = (Integer) in.readObject();

        // Excluir o produto com o ID especificado
        try {
            ctrl.destroy(idProduto);
        } catch (NonexistentEntityException e) {
            // Produto não encontrado, você pode enviar uma mensagem de erro de
            volta para o cliente
            out.writeObject("Produto não encontrado.");
        }
    } else {
        // Comando desconhecido, você pode enviar uma mensagem de erro de
        volta para o cliente
        out.writeObject("Comando desconhecido.");
    }
}
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}

```

```

    } finally {
        try {
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}
}

```

## CADASTRO CLIENTE

```

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.List;

public class CadastroClient {
    public static void main(String[] args) {
        try {
            // Instanciar um Socket apontando para localhost, na porta 4321
            Socket socket = new Socket("localhost", 4321);

            // Encapsular os canais de entrada e saída do Socket
            ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(socket.getInputStream());

            // Escrever o login e a senha na saída
            String login = "op1";
            String senha = "op1";
            out.writeObject(login);
            out.writeObject(senha);

            // Enviar o comando "L" no canal de saída
            out.writeObject("L");

            // Receber a coleção de entidades no canal de entrada
            List<Produto> produtos = (List<Produto>) in.readObject();

            // Apresentar o nome de cada entidade recebida
            for (Produto produto : produtos) {
                System.out.println("Nome do Produto: " + produto.getNome());
            }
        }
    }
}

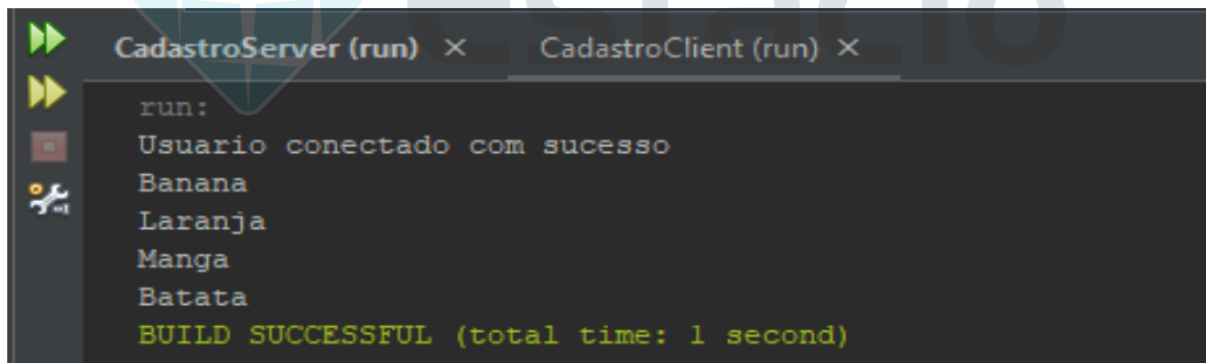
```

```

        // Fechar a conexão
        socket.close();
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
}
}

```

## RESULTADO



```

run:
Usuario conectado com sucesso
Banana
Laranja
Manga
Batata
BUILD SUCCESSFUL (total time: 1 second)

```

## CADASTRO THREAD 2

Aqui está a implementação da segunda versão da classe `CadastroThread` com a funcionalidade adicional para entrada (E) e saída (S) de produtos, bem como a gestão de movimentos e pessoas:

```

```java
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.List;

public class CadastroThread extends Thread {
    private final Socket socket;
    private final ProdutoJpaController ctrl;
    private final UsuarioJpaController ctrlUsu;
    private final MovimentoJpaController ctrlMov;

```

```

private final PessoaJpaController ctrlPessoa;

public CadastroThread(Socket socket, ProdutoJpaController ctrl,
UsuarioJpaController ctrlUsu, MovimentoJpaController ctrlMov, PessoaJpaController
ctrlPessoa) {
    this.socket = socket;
    this.ctrl = ctrl;
    this.ctrlUsu = ctrlUsu;
    this.ctrlMov = ctrlMov;
    this.ctrlPessoa = ctrlPessoa;
}

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

        // Obter login e senha do cliente
        String login = (String) in.readObject();
        String senha = (String) in.readObject();

        // Verificar as credenciais com o controlador de usuários
        Usuario usuario = ctrlUsu.findUsuario(login, senha);

        if (usuario == null) {
            // Se as credenciais forem inválidas, encerrar a conexão
            System.out.println("Credenciais inválidas. Desconectando cliente.");
            return;
        }

        // Loop de resposta
        while (true) {
            String comando = (String) in.readObject();

            if (comando.equals("L")) {
                // Se o comando for 'L', retornar o conjunto de produtos
                List<Produto> produtos = ctrl.listarProdutos();
                out.writeObject(produtos);
            } else if (comando.equals("E") || comando.equals("S")) {
                // Se o comando for 'E' (entrada) ou 'S' (saída)
                int idPessoa = (Integer) in.readObject();
                int idProduto = (Integer) in.readObject();
                int quantidade = (Integer) in.readObject();
                double valorUnitario = (Double) in.readObject();
            }
        }
    }
}

```

```

// Verificar se a pessoa e o produto existem
Pessoa pessoa = ctrlPessoa.findPessoa(idPessoa);
Produto produto = ctrl.findProduto(idProduto);

if (pessoa == null || produto == null) {
    // Se a pessoa ou o produto não forem encontrados, enviar mensagem
    out.writeObject("Pessoa ou produto não encontrados.");
} else {
    // Gerar um objeto Movimento
    Movimento movimento = new Movimento();
    movimento.setUsuario(usuario);
    movimento.setTipo(comando);
    movimento.setPessoa(pessoa);
    movimento.setProduto(produto);
    movimento.setQuantidade(quantidade);
    movimento.setValorUnitario(valorUnitario);

    // Persistir o movimento
    ctrlMov.create(movimento);

    // Atualizar a quantidade de produtos
    if (comando.equals("E")) {
        // Entrada de produtos
        produto.setQuantidade(produto.getQuantidade() + quantidade);
    } else {
        // Saída de produtos
        produto.setQuantidade(produto.getQuantidade() - quantidade);
    }
    ctrl.edit(produto);
}
} else {
    // Comando desconhecido, você pode enviar uma mensagem de erro de
    volta para o cliente
    out.writeObject("Comando desconhecido.");
}
}
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
} finally {
    try {
        socket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

```
}  
}
```

## CADASTRO CLIENTE V2

```
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.net.Socket;  
  
public class CadastroClientV2 {  
    public static void main(String[] args) {  
        try (Socket socket = new Socket("localhost", 4321);  
            ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());  
            ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {  
  
            BufferedReader reader = new BufferedReader(new  
InputStreamReader(System.in));  
  
            // Enviar login e senha para o servidor  
            out.writeObject("op1"); // Substitua com seu login  
            out.writeObject("op1"); // Substitua com sua senha  
  
            // Inicializar a janela para apresentação de mensagens (Passo 4)  
            // Iniciar a Thread para preenchimento assíncrono (Passo 5) com 'in'  
  
            while (true) {  
                System.out.println("Menu:");  
                System.out.println("L - Listar");  
                System.out.println("X - Finalizar");  
                System.out.println("E - Entrada");  
                System.out.println("S - Saída");  
                System.out.print("Escolha uma opção: ");  
  
                String comando = reader.readLine().trim();
```



```

if (comando.equalsIgnoreCase("L")) {
    // Enviar comando 'L' para o servidor
    out.writeObject("L");
} else if (comando.equalsIgnoreCase("E") || comando.equalsIgnoreCase("S"))
{
    // Enviar comando 'E' ou 'S' para o servidor
    out.writeObject(comando);

    // Obter o Id da pessoa via teclado e enviar para o servidor
    System.out.print("Digite o Id da pessoa: ");
    int idPessoa = Integer.parseInt(reader.readLine().trim());
    out.writeObject(idPessoa);

    // Obter o Id do produto via teclado e enviar para o servidor
    System.out.print("Digite o Id do produto: ");
    int idProduto = Integer.parseInt(reader.readLine().trim());
    out.writeObject(idProduto);

    // Obter a quantidade via teclado e enviar para o servidor
    System.out.print("Digite a quantidade: ");
    int quantidade = Integer.parseInt(reader.readLine().trim());
    out.writeObject(quantidade);

    // Obter o valor unitário via teclado e enviar para o servidor
    System.out.print("Digite o valor unitário: ");
    double valorUnitario = Double.parseDouble(reader.readLine().trim());
    out.writeObject(valorUnitario);
} else if (comando.equalsIgnoreCase("X")) {
    // Enviar comando 'X' para o servidor e encerrar o cliente
    out.writeObject("X");
    break;
} else {
    System.out.println("Comando inválido.");
}
}
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

## **Análise e Conclusão:**

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

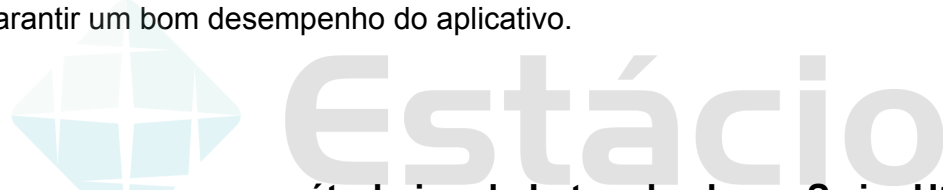
As Threads podem ser usadas para o tratamento assíncrono das respostas enviadas pelo servidor em um aplicativo Java. Quando você precisa realizar tarefas em paralelo, como receber e processar respostas do servidor enquanto mantém a interface do usuário responsiva, as Threads são uma abordagem fundamental. Aqui estão algumas maneiras de usar Threads para tratamento assíncrono:

1. **\*\*Thread de Comunicação\*\***: Como visto em sua solicitação original, você pode criar uma Thread dedicada para lidar com a comunicação do servidor. Isso permite que a Thread principal do aplicativo continue executando outras tarefas, como atualizações da interface do usuário, enquanto a Thread de comunicação aguarda e processa respostas do servidor. Isso mantém a interface do usuário responsiva.
2. **\*\*Threads Executoras\*\***: A API `java.util.concurrent` oferece uma variedade de classes, como `ExecutorService`, que facilitam a criação e o gerenciamento de Threads executoras. Você pode usar um `ExecutorService` para enviar tarefas (por exemplo, processamento de respostas do servidor) para serem executadas em Threads separadas. Isso ajuda a gerenciar e reutilizar Threads de forma eficiente.
3. **\*\*SwingWorker (para aplicativos Swing)\*\***: Se você estiver desenvolvendo um aplicativo Swing, pode usar a classe `SwingWorker` para realizar tarefas em segundo plano e atualizar a interface do usuário de maneira segura. Ele fornece métodos para executar código em uma Thread separada e, em seguida, atualizar a interface do usuário na Thread de despacho de eventos do Swing.
4. **\*\*CompletableFuture (Java 8+)\*\***: Se estiver usando Java 8 ou posterior, a classe `CompletableFuture` pode ser útil para tratar assincronicamente os resultados de operações assíncronas. Você pode criar `CompletableFuture`s para executar operações em Threads separadas e combinar os resultados quando estiverem prontos.
5. **\*\*Tarefas de Fundo\*\***: Em aplicativos Android, você pode usar Threads ou `AsyncTask` para realizar tarefas em segundo plano e atualizar a interface do usuário conforme necessário.

6. **\*\*Java Concurrency Framework\*\***: O Java oferece muitas outras classes e recursos para lidar com programação concorrente, como semáforos, bloqueios, filas concorrentes, entre outros. Esses recursos podem ser usados para implementar tratamento assíncrono em diferentes cenários.

Em todos esses casos, é importante garantir que a comunicação entre Threads seja feita de maneira segura para evitar problemas como condições de corrida e bloqueios. O Java fornece mecanismos, como sincronização e objetos `Lock`, para ajudar a garantir a segurança da Thread.

Além disso, você deve considerar a gerência adequada de Threads, incluindo a criação, inicialização e término adequados, para evitar vazamento de recursos e garantir um bom desempenho do aplicativo.



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

O método `invokeLater` da classe `SwingUtilities` é usado em aplicativos Java Swing para executar tarefas na Thread de despacho de eventos do Swing (Event Dispatch Thread - EDT). A principal finalidade desse método é garantir que determinado código que afeta a interface do usuário seja executado na EDT, que é a Thread responsável pela atualização e manipulação dos componentes gráficos do Swing. Isso é importante porque o Swing não é thread-safe, ou seja, a maioria das operações de manipulação da interface do usuário deve ocorrer na EDT para evitar problemas de concorrência e bloqueios.

Aqui está para que serve o método `invokeLater`:

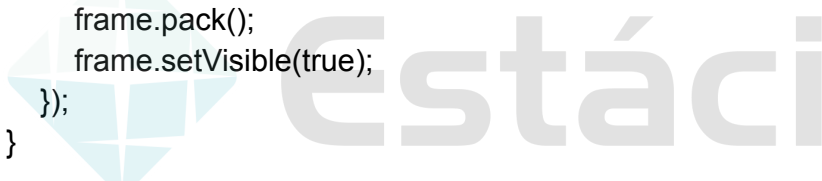
1. **\*\*Execução de Código na EDT\*\***: O método `invokeLater` permite que você agende a execução de uma tarefa (geralmente uma função ou um pedaço de código) na EDT. Isso é útil quando você precisa atualizar a interface do usuário, como modificar propriedades de componentes gráficos, adicionar ou remover elementos em uma GUI, entre outras operações que afetam a interface do usuário.
2. **\*\*Mantém a Interface do Usuário Responsiva\*\***: Ao executar tarefas na EDT, você garante que a interface do usuário permaneça responsiva e não congele enquanto o código é executado. Isso é essencial para fornecer uma experiência de usuário suave, especialmente em aplicativos GUI.
3. **\*\*Evita Bloqueios e Condições de Corrida\*\***: Como o Swing não é thread-safe, executar operações de GUI em Threads não-EDT pode levar a problemas como bloqueios e condições de corrida. Usando o `invokeLater`, você evita esses problemas, pois as operações são executadas sequencialmente na EDT.

Aqui está um exemplo de como usar o `invokeLater`:

```
```java
import javax.swing.SwingUtilities;

public class ExemploSwingInvokeLater {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            // Código que afeta a interface do usuário
            // Exemplo: atualizar um rótulo de texto em um JFrame
            JFrame frame = new JFrame("Exemplo");
            JLabel label = new JLabel("Olá, mundo!");
            frame.add(label);
            frame.pack();
            frame.setVisible(true);
        });
    }
}
...

```



Neste exemplo, o código que cria e exibe uma janela com um rótulo de texto é colocado dentro do `invokeLater`. Isso garante que a criação e exibição da janela ocorram na EDT, mantendo a interface do usuário responsiva.

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

No Java, os objetos são enviados e recebidos por meio de Sockets utilizando as classes `ObjectInputStream` e `ObjectOutputStream`, que são baseadas em fluxos de bytes (InputStream e OutputStream) e permitem a serialização e desserialização de objetos.

**\*\*Envio de Objetos:\*\***

Para enviar um objeto por um Socket, siga estes passos:

1. Crie um `ObjectOutputStream` vinculado ao `OutputStream` do Socket.
2. Use o método `writeObject()` para escrever o objeto no `ObjectOutputStream`.
3. O `ObjectOutputStream` serializa o objeto em bytes e envia esses bytes pelo Socket para o destino.

Exemplo de envio de objeto:

```
```java
Socket socket = new Socket("localhost", 12345);
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

```

```
MyObject myObject = new MyObject(); // MyObject é uma classe que implementa
Serializable
out.writeObject(myObject);

out.close();
socket.close();
...

```

#### **\*\*Recebimento de Objetos:\*\***

Para receber um objeto por um Socket, siga estes passos:

1. Crie um `ObjectInputStream` vinculado ao `InputStream` do Socket.
2. Use o método `readObject()` para ler o objeto do `ObjectInputStream`. Esse método desserializa os bytes recebidos em um objeto Java.
3. Você precisa fazer o casting do objeto recebido para o tipo apropriado.

Exemplo de recebimento de objeto:

```
``java
ServerSocket serverSocket = new ServerSocket(12345);
Socket socket = serverSocket.accept(); // Aguarda a conexão do cliente
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());

MyObject receivedObject = (MyObject) in.readObject();

in.close();
socket.close();
serverSocket.close();
...

```

É importante notar que as classes utilizadas para a comunicação via Socket devem implementar a interface `Serializable` para que possam ser serializadas e desserializadas de forma adequada. Além disso, a ordem de leitura e escrita dos objetos deve ser a mesma no cliente e no servidor para evitar problemas de sincronização.

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

A utilização de comportamento assíncrono e síncrono em clientes Socket Java tem impacto significativo no bloqueio do processamento e na experiência do usuário. Vamos comparar esses dois modos de operação:

### **\*\*Comportamento Síncrono:\*\***

1. **\*\*Bloqueio do Processamento:\*\*** Em operações síncronas, o cliente aguarda até que uma operação no socket seja concluída antes de continuar. Isso significa que o processamento do cliente é bloqueado até que a ação no socket (enviar ou receber dados) seja finalizada.
2. **\*\*Simplicidade:\*\*** O código síncrono tende a ser mais simples de entender e depurar, uma vez que as operações ocorrem sequencialmente.
3. **\*\*Facilidade de Implementação:\*\*** É mais fácil garantir a consistência dos dados e sincronização quando as operações ocorrem sequencialmente.
4. **\*\*Bloqueio da Interface do Usuário:\*\*** Se a comunicação via socket ocorre na thread principal de uma aplicação de GUI, as operações síncronas podem bloquear a interface do usuário, tornando-a não responsiva.

### **\*\*Comportamento Assíncrono:\*\***

1. **\*\*Não Bloqueio do Processamento:\*\*** Em operações assíncronas, o cliente pode iniciar uma operação no socket e continuar executando outras tarefas sem esperar pela conclusão da operação no socket. Isso evita o bloqueio do processamento e permite que o cliente seja mais responsivo.
2. **\*\*Complexidade:\*\*** O código assíncrono pode ser mais complexo de implementar e depurar, pois envolve lidar com tarefas concorrentes e potencialmente sincronizar o acesso a recursos compartilhados.
3. **\*\*Responsividade da Interface do Usuário:\*\*** O comportamento assíncrono é útil em aplicativos com interface do usuário, pois permite que a interface permaneça responsiva enquanto as operações de socket estão em andamento em segundo plano.
4. **\*\*Uso de Threads:\*\*** A implementação de operações assíncronas frequentemente envolve o uso de threads ou estruturas de programação assíncronas, como `CompletableFuture` em Java, para gerenciar tarefas concorrentes de maneira organizada.

A escolha entre comportamento síncrono e assíncrono depende dos requisitos e das características da aplicação:

- Em cenários onde a responsividade da interface do usuário é crucial, o comportamento assíncrono é preferível para evitar bloqueios.
- Em operações que podem ser realizadas em segundo plano sem impacto significativo no desempenho da aplicação, o comportamento assíncrono é uma escolha lógica.

- Em operações que dependem estritamente de uma sequência específica de ações, o comportamento síncrono pode ser mais apropriado.

- Em situações complexas de concorrência e gerenciamento de threads, o comportamento assíncrono pode ser mais desafiador de implementar e requer atenção especial à sincronização.

Portanto, a escolha entre comportamento síncrono e assíncrono depende dos requisitos específicos do aplicativo e das considerações de desempenho e experiência do usuário.

