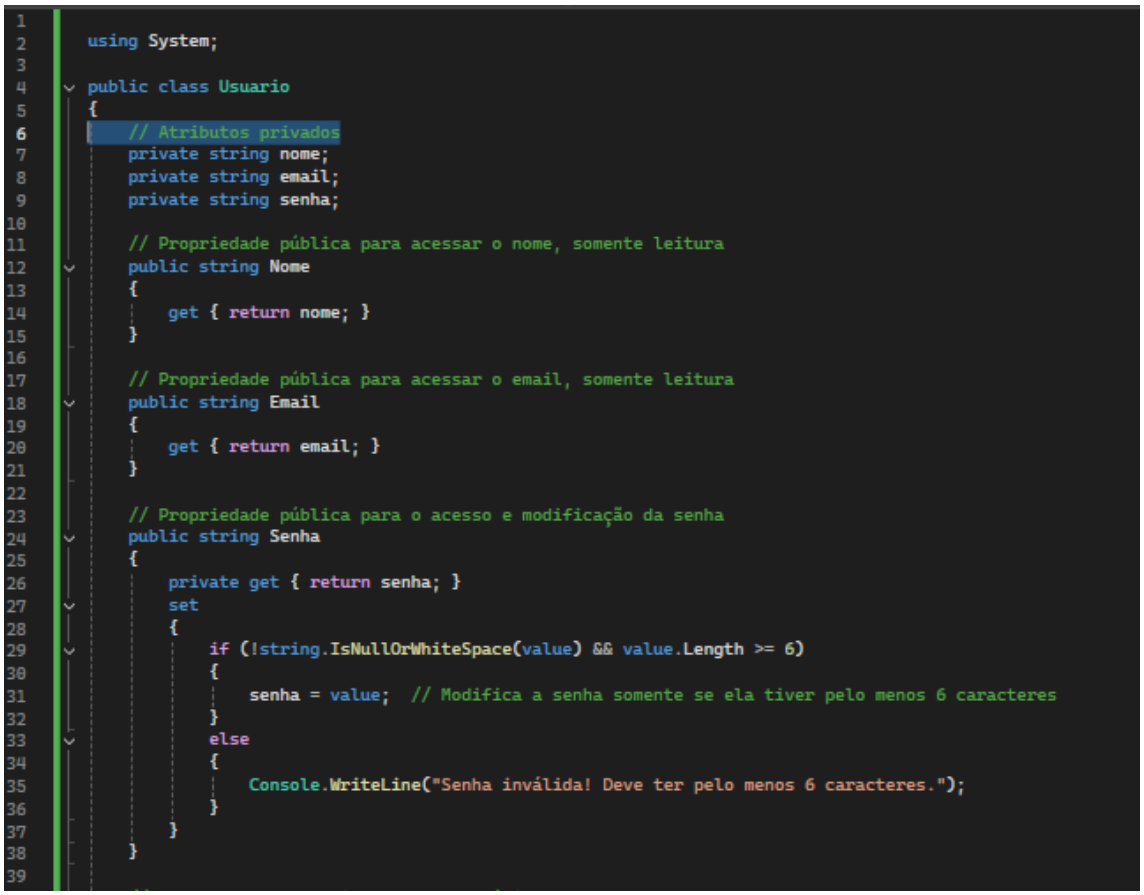
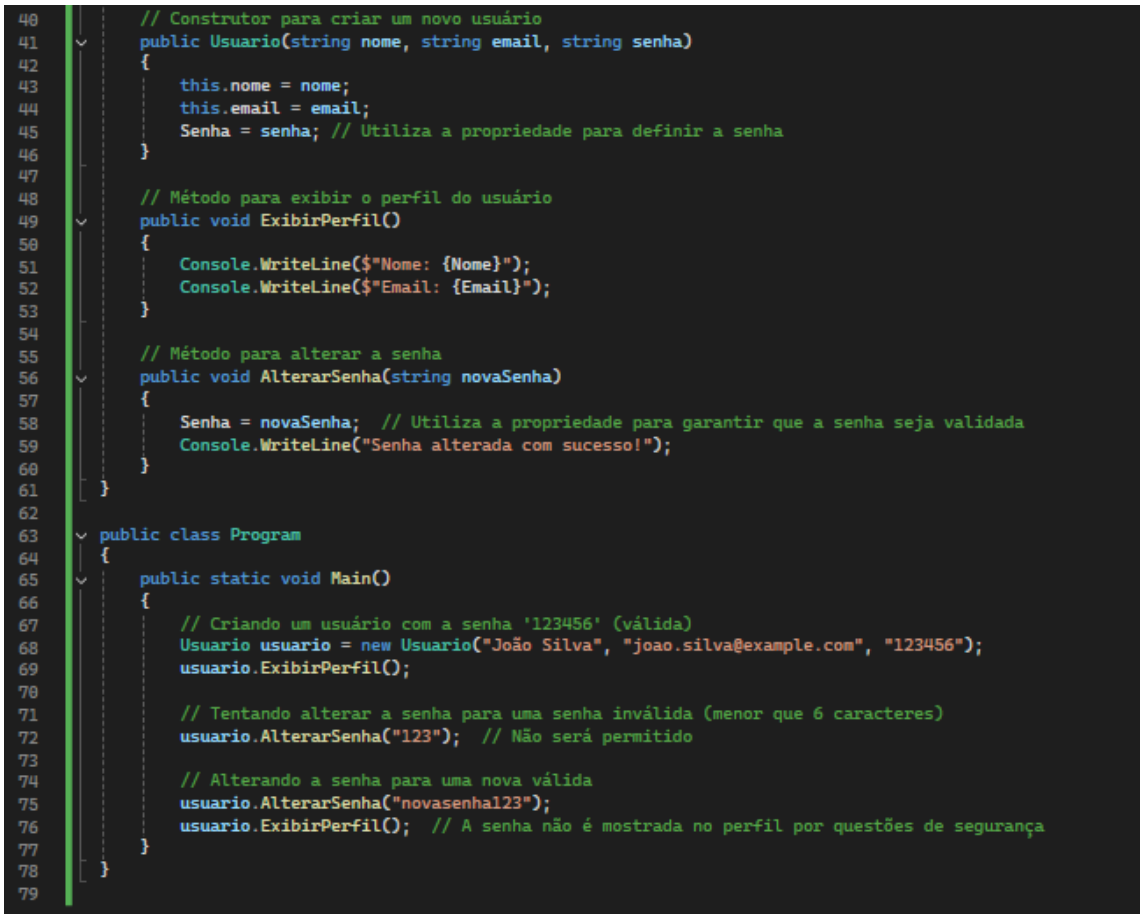
Teste Técnico – Desenvolvedor | Raphael Gundim

1 – Encapsulamento em c#.

O **encapsulamento** é um dos princípios fundamentais da programação orientada a objetos (OOP). O funcionamento desse princípio consiste em ocultar os detalhes da implementação de uma classe e fornecer acesso aos dados por meio de métodos ou propriedades controladas. O objetivo do encapsulamento é proteger os dados de alterações indevidas e garantir que a interação com o objeto ocorra de com as definições de acesso na classe.



```
using System;

public class Usuario
{
    // Atributos privados
    private string nome;
    private string email;
    private string senha;

    // Propriedade pública para acessar o nome, somente leitura
    public string Nome
    {
        get { return nome; }
    }

    // Propriedade pública para acessar o email, somente leitura
    public string Email
    {
        get { return email; }
    }

    // Propriedade pública para o acesso e modificação da senha
    public string Senha
    {
        private get { return senha; }
        set
        {
            if (!string.IsNullOrWhiteSpace(value) && value.Length >= 6)
            {
                senha = value;  // Modifica a senha somente se ela tiver pelo menos 6 caracteres
            }
            else
            {
                Console.WriteLine("Senha inválida! Deve ter pelo menos 6 caracteres.");
            }
        }
    }

```



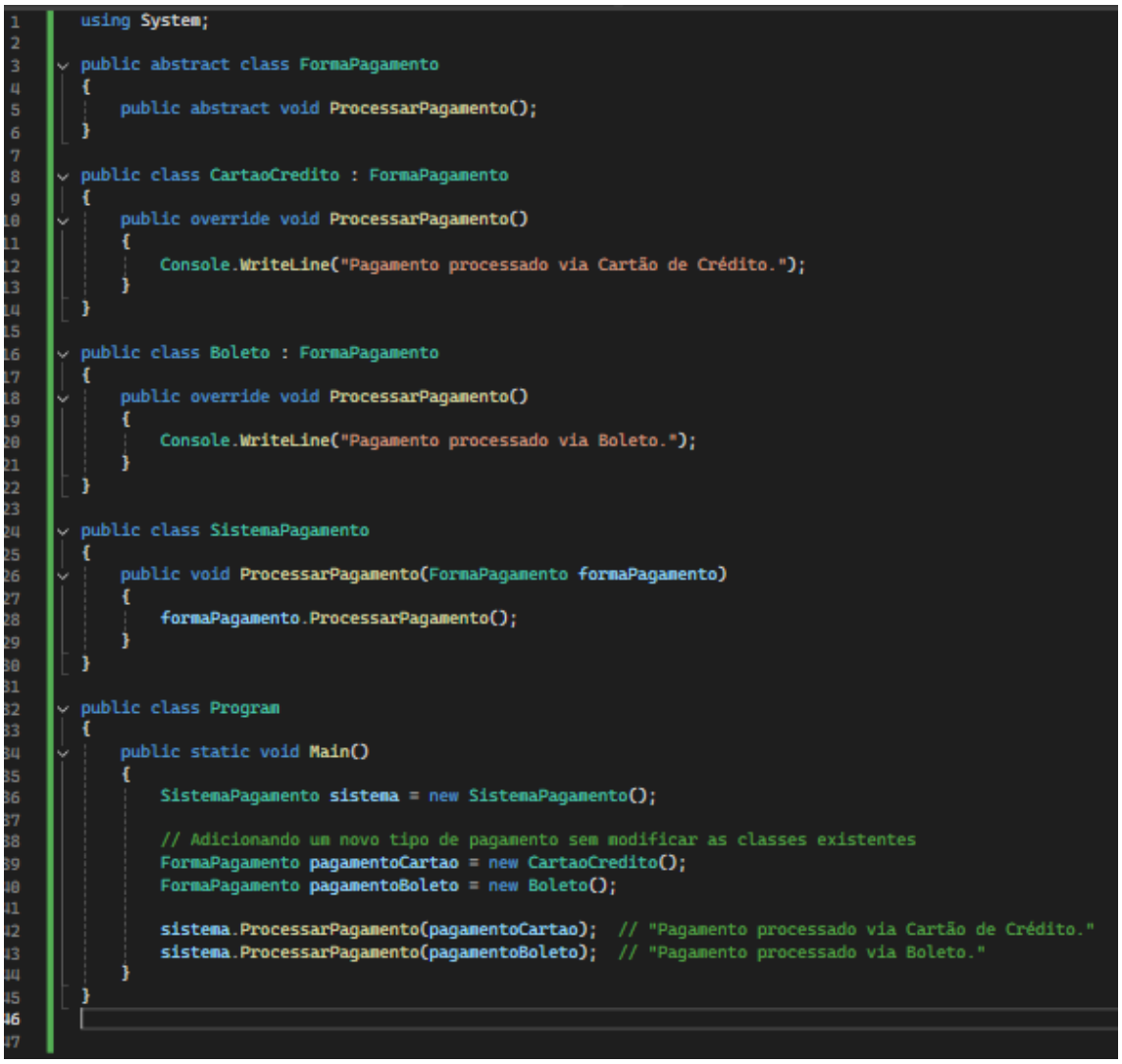
```
    // Construtor para criar um novo usuário
    public Usuario(string nome, string email, string senha)
    {
        this.nome = nome;
        this.email = email;
        Senha = senha; // Utiliza a propriedade para definir a senha
    }

    // Método para exibir o perfil do usuário
    public void ExibirPerfil()
    {
        Console.WriteLine($"Nome: {Nome}");
        Console.WriteLine($"Email: {Email}");
    }

    // Método para alterar a senha
    public void AlterarSenha(string novaSenha)
    {
        Senha = novaSenha;  // Utiliza a propriedade para garantir que a senha seja validada
        Console.WriteLine("Senha alterada com sucesso!");
    }
}

public class Program
{
    public static void Main()
    {
        // Criando um usuário com a senha '123456' (válida)
        Usuario usuario = new Usuario("João Silva", "joao.silva@example.com", "123456");
        usuario.ExibirPerfil();

        // Tentando alterar a senha para uma senha inválida (menor que 6 caracteres)
        usuario.AlterarSenha("123");  // Não será permitido

        // Alterando a senha para uma nova válida
        usuario.AlterarSenha("novasenha123");
        usuario.ExibirPerfil();  // A senha não é mostrada no perfil por questões de segurança
    }
}
```

2. Princípio Open/Closed (Aberto/Fechado)

O **princípio Open/Closed** faz parte dos **princípios SOLID** e consiste na ideia de que uma classe deve ser **aberta para extensão, mas fechada para modificação**. Com isso, deve ser liberado para criar novos comportamentos à classe sem precisar alterar o código já implementado.

Exemplo:

Vamos pensar em um sistema que calcula o preço de diferentes formas de pagamento (cartão de crédito, boleto etc.). Aqui entra o princípio Open/Closed para adicionar novos tipos de pagamento sem modificar as classes já existentes.



```
using System;

public abstract class FormaPagamento
{
    public abstract void ProcessarPagamento();
}

public class CartaoCredito : FormaPagamento
{
    public override void ProcessarPagamento()
    {
        Console.WriteLine("Pagamento processado via Cartão de Crédito.");
    }
}

public class Boleto : FormaPagamento
{
    public override void ProcessarPagamento()
    {
        Console.WriteLine("Pagamento processado via Boleto.");
    }
}

public class SistemaPagamento
{
    public void ProcessarPagamento(FormaPagamento formaPagamento)
    {
        formaPagamento.ProcessarPagamento();
    }
}

public class Program
{
    public static void Main()
    {
        SistemaPagamento sistema = new SistemaPagamento();

        // Adicionando um novo tipo de pagamento sem modificar as classes existentes
        FormaPagamento pagamentoCartao = new CartaoCredito();
        FormaPagamento pagamentoBoleto = new Boleto();

        sistema.ProcessarPagamento(pagamentoCartao);  // "Pagamento processado via Cartão de Crédito."
        sistema.ProcessarPagamento(pagamentoBoleto);  // "Pagamento processado via Boleto."
    }
}
```

Neste exemplo, você pode adicionar novos tipos de pagamento criando novas subclasses que herdam de FormaPagamento, sem precisar modificar a classe SistemaPagamento, respeitando o princípio Open/Closed.

3. Qual será a saída do código abaixo? Justifique sua resposta.

Chamada do método:

Animal a1 = new Cachorro();

Animal a2 = new Gato();

a1.Falar();

a2.Falar();

Saída:

Latido

Miau

Nessa questão, o que diferencia as duas classes e como estão implementando os métodos é:

override: Substitui o comportamento do método na classe base, independentemente de como o objeto é referenciado (por classe base ou classe derivada).

new: Oculta o método da classe base, e o comportamento do método depende do tipo da referência do objeto. Se a referência for do tipo da classe base, o método da classe base será chamado; caso contrário, o método da classe derivada será chamado.

4. Modificadores private, protected e public no TypeScript

Em TypeScript, os modificadores de acesso controlam a visibilidade dos membros de uma classe (propriedades e métodos).

- **private**: O membro é acessível apenas dentro da própria classe.
- **protected**: O membro é acessível dentro da classe e nas classes que herdam dela.
- **public**: O membro é acessível de qualquer lugar.

Exemplo:

```typescript
class Pessoa
{
    private nome: string;        // Acessível apenas dentro da classe Pessoa
    protected idade: number;     // Acessível dentro da classe Pessoa e das suas subclasses
    public endereco: string;     // Acessível de qualquer lugar

    constructor(nome: string, idade: number, endereco: string)
    {
        this.nome = nome;
        this.idade = idade;
        this.endereco = endereco;
    }

    public mostrarEndereco()
    {
        console.log(this.endereco);  // Acessando o endereço (público)
    }
}

class Cliente extends Pessoa
{
    public mostrarIdade()
{
    console.log(this.idade);  // Acessando a idade (protegido, pois é herdado)
}
}

const pessoa = new Pessoa("João", 30, "Rua 123");
console.log(pessoa.endereco);  // Acessível (público)
pessoa.mostrarEndereco();      // Acessível (público)

//const cliente = new Cliente("Maria", 25, "Rua 456");
//console.log(cliente.nome);    // Erro: nome é privado, não pode ser acessado
```

5. Pipes no Angular

Pipes no Angular são usados para transformar dados na visualização de forma declarativa. Eles podem ser usados para transformar texto, formatos de data, moedas, etc., diretamente no template.

Como criar um pipe personalizado para formatar um CPF?

```typescript
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
name: 'cpf'
})
export class CpfPipe implements PipeTransform
{
    transform(value: string): string {
        if (!value) return value;

        // Verifica se o CPF tem 11 caracteres
        return value.replace(/ (\d{ 3})(\d{ 3})(\d{ 3})(\d{ 2})/, '$1.$2.$3-$4');
    }
}
```

Uso no template:

Html

```
<p>{{ '12345678901' | cpf }}</p>  <!-- Resultado: 123.456.789-01 -->
```

Métodos de formatação de dados para visualização como esse, é conhecido popularmente de máscara. Os pipes em Angular resolvem um problema que é a formatação de CEP's, CNPJ's, contas contábeis e principalmente como no exemplo acima, os CPF's.

6. **Query SQL**

```
SELECT *
FROM Pessoas
WHERE Estado = 'SP'
AND Idade BETWEEN 25 AND 40
ORDER BY Nome;
```

7. **Ciclo de Vida de um Componente Angular**

O ciclo de vida de um componente no Angular consiste em uma série de eventos que ocorrem durante a vida útil de um componente.

Os 3 hooks principais são:

ngOnInit: Chamado uma vez, logo após a construção do componente, ideal para inicializar dados.

ngOnChanges: Chamado quando uma propriedade de entrada do componente é alterada.

ngOnDestroy: Chamado quando o componente é destruído.

Para a chamada HTTP quando o componente for exibido, o correto é usar o ngOnInit

8. **Código Incompleto ou Errado**

O código está incorreto por conta da chamada HTTP assíncrona usando o Subscribe. No momento que o console.log é chamado, a variável pessoas ainda não foi populada. Para corrigir basta incluir o console.log para dentro da chamada HTTP.