MODIFICADORES DE ACESSO, PALAVRAS CHAVES, CONSTRUTORES, INTERFACES E NAMESPACES

Autor(es)

Douglas Gomes Ferreira de Moraes

Thiago Keller Torquato Vicco

@2TDS-2024

Sumário

INTRODUÇÃO

Lembrando que essa apostila é somente um apoio, nas aulas teremos explicações mais profundas e outros conteúdos, **por isso é de grande importância a sua presença nas aulas.**

1. MODIFICADORES DE ACESSO

1.1. Introdução

Quando estamos desenvolvendo um software é importante tomarmos cuidado com a visibilidade que os nossos métodos, funções e variáveis possuem dentro da nossa solução.

Mas por que isso é relevante?

Imagine que você tem uma aplicação que controla todo o sistema de pagamentos da sua empresa. Por algum motivo, você precisa alterar uma função de cálculo de juros para um determinado cenário. Ao navegar até a classe desejada, você faz a alteração no seu método que deveria ser acessível somente para aquele contexto, porém, ao subir a sua aplicação em produção, você nota que algo está errado em outras partes de cobrança da sua empresa o que ocasional uma perda operacional de **milhares de reais**. O que aconteceu?

Nesse exemplo, é muito provável que o método alterado estava como "**public"** e estava sendo utilizado em outras partes da sua solução.

Para evitarmos esse cenário, é importante entendermos como funcionam os controladores de acesso no C# e como podemos limitar os acessos aos tipos

e membros de uma classe somente para quem precisa de fato acessar essas informações.

1.2. Tipos de Modificadores

A hierarquia de acesso em C# pode ser resumida da seguinte forma (do mais restrito para o menos restrito): private, protected, internal e public.

1.2.1. private

O modificador de acesso "**private**" em C# é um dos modificadores utilizados para controlar a visibilidade de membros de uma classe.

O modificador "**private**" é essencial para criar classes bem encapsuladas, fornecendo controle preciso sobre quais partes do código podem interagir diretamente com os detalhes internos da implementação da classe. Isso promove a modularidade, reutilização de código e facilita a manutenção do sistema.

Aqui estão alguns detalhes importantes sobre o modificador "**private**":

1. **Acesso Restrito à Classe:** O modificador "**private**" restringe o acesso ao membro somente à própria classe em que o membro está declarado. Isso significa que o membro não pode ser acessado por outras classes, nem mesmo por subclasses.

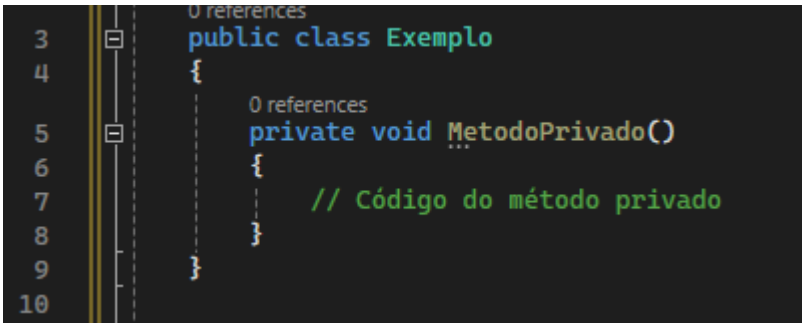
```
0 references
public class MinhaClasse
{
    //Essa propriedade não é acessível por outras classes
    private int numeroPrivado;

    0 references
    public void SetNumero(int novoNumero)
    {
        numeroPrivado = novoNumero;
    }

    0 references
    public int GetNumero()
    {
        return numeroPrivado;
    }
}
```

2. **Encapsulamento:** O uso de "**private**" promove o encapsulamento, que é um dos princípios fundamentais da programação orientada a objetos. O encapsulamento ajuda a ocultar detalhes internos da implementação da classe, permitindo que ela controle o acesso aos seus membros.
3. **Proteção de Dados:** Ao declarar membros como "**private**", você protege esses membros de manipulação direta por outras partes do programa. Isso ajuda a garantir que o estado interno da classe seja modificado de maneira controlada, geralmente por meio de métodos públicos (métodos de acesso).
4. **Exemplo com Propriedade:** Além de campos, você pode usar "**private**" em propriedades para controlar o acesso aos dados internos da classe.

```
0 references
public class Pessoa
{
    private string nome;

    0 references
    public string Nome
    {
        get { return nome; }
        set { nome = value; }
    }
}
```

Neste exemplo, a propriedade **Nome** tem um campo privado associado (**nome**) e é acessível apenas dentro da própria classe.

5. **Membros Privados em Métodos:** Além de campos e propriedades, o modificador "**private**" também pode ser aplicado a métodos, limitando seu acesso à própria classe.



1.2.2. protected

O modificador de acesso "**protected**" em C# é utilizado para controlar a visibilidade de membros de uma classe, permitindo que esses membros sejam acessíveis na própria classe e também nas classes derivadas (subclasses) da classe onde esses membros são declarados.

O "**protected**" desempenha um papel importante em cenários de herança, fornecendo uma forma de compartilhar funcionalidades entre uma classe base e suas classes derivadas, ao mesmo tempo que controla o acesso ao código externo. É uma ferramenta útil para criar hierarquias de classes flexíveis e reutilizáveis em aplicações orientadas a objetos.

Aqui estão alguns pontos importantes sobre o modificador "**protected**":

1. **Acesso à Própria Classe e Classes Derivadas:** Membros marcados como "**protected**" são acessíveis dentro da própria classe e também em classes derivadas.

```
1 reference
public class ClasseBaseProtected
{
    protected int numeroProtegido;

    1 reference
    protected void MetodoProtegido()
    {
        // Código do método protegido
    }
}

0 references
public class ClasseDerivadaProtected : ClasseBaseProtected
{
    0 references
    public void Exemplo()
    {
        // Acesso ao membro protegido da classe base
        numeroProtegido = 42;
        MetodoProtegido();
    }
}
```

2. **Hierarquia de Herança:** O “**protected**” é frequentemente utilizado em membros de uma classe base que se deseja que sejam acessíveis nas classes derivadas. Isso é especialmente útil em cenários de herança, onde a classe derivada herda e estende a funcionalidade da classe base.

3. **Proteção de Dados:** O “**protected**” é útil para proteger dados que devem ser acessíveis para classes derivadas, mas não para o código externo.

```
/// <summary>
/// Propriedade protegida
/// </summary>
2 references
public class PessoaProtected
{
    protected string nome;

    0 references
    public void SetNome(string novoNome)
    {
        nome = novoNome;
    }
}

/// <summary>
/// Propriedade protegida sendo acessada pela classe que herda. Ver exemplo na program sobre acesso sem herdar
/// </summary>
0 references
public class ClienteProtected : PessoaProtected
{
    0 references
    public void Exemplo()
    {
        // Acesso ao membro protegido da classe base
        nome = "João";
    }
}
```

```
var exemploProtected1 = new PessoaProtected();
exemploProtected1.nome = "teste"; //Note a propriedade não sendo encontrada
```

4. **Métodos Virtuais e override:** Membros "**protected**" frequentemente são usados em conjunto com métodos virtuais para permitir que classes derivadas forneçam implementações específicas.

```
54      /// <summary>
55      /// Método virtual para substituição
56      /// </summary>
        1 reference
57      public class AnimalProtected
58      {
            1 reference
59          protected virtual void EmitirSom()
60          {
61              Console.WriteLine("Som genérico de animal");
62          }
63      }
64
65      /// <summary>
66      /// Substituição do método virtual
67      /// </summary>
        0 references
68      public class CachorroProtected : AnimalProtected
69      {
            1 reference
70          protected override void EmitirSom()
71          {
72              Console.WriteLine("Au Au");
73          }
74      }
```

Neste exemplo, a classe derivada **Cachorro** substitui o método virtual **EmitirSom** da classe base **Animal**.

5. **Acesso Limitado a Outras Classes:** Ao contrário do "**public**" ou "**internal**", o "**protected**" limita o acesso a membros apenas às classes derivadas, o que ajuda a manter um nível de encapsulamento.

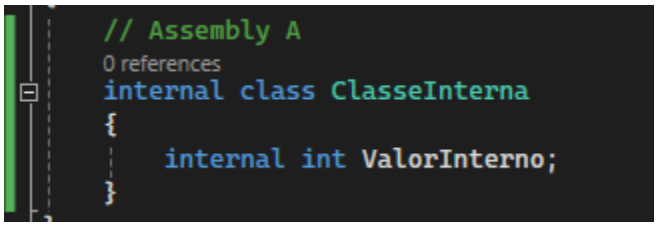
1.2.3. internal

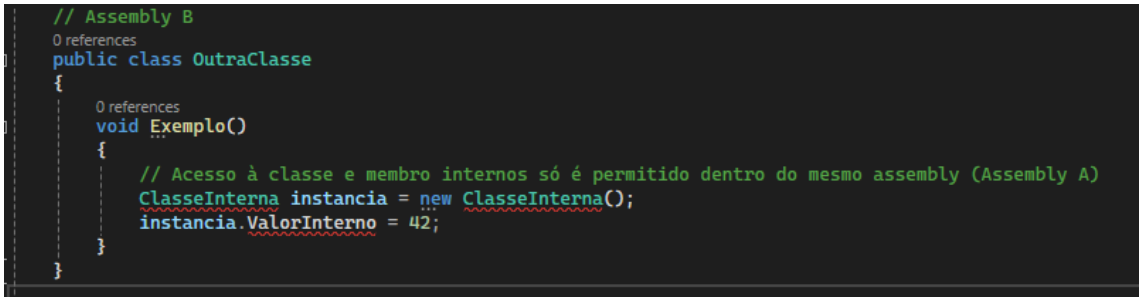
O modificador de acesso “**internal**” em C# é utilizado para controlar a visibilidade de membros de uma classe no âmbito do assembly em que estão declarados. Isso significa que os membros marcados como internal são acessíveis somente por classes dentro do mesmo assembly, mas não são visíveis fora desse assembly.

O modificador “**internal**” é parte integrante da estratégia de encapsulamento em C#, permitindo que você restrinja o acesso a membros apenas ao âmbito do assembly em que estão definidos. Essa prática ajuda a criar sistemas modulares, coesos e facilita a manutenção do código em escala.

Aqui estão alguns pontos importantes sobre o modificador “**internal**”:

1. **Escopo Limitado ao Assembly:** Membros marcados como “**internal**” são acessíveis apenas dentro do mesmo assembly. Isso ajuda a controlar o acesso a membros específicos, limitando-os a um escopo definido.





2. **Proteção contra Acesso Indevido:** O “**internal**” fornece uma camada adicional de proteção contra acesso não autorizado aos membros. Isso é útil para ocultar implementações internas de um assembly e evitar que outros assemblies manipulem detalhes internos.

3. **Separando Componentes em Assemblies:** O “**internal**” é útil ao criar assemblies separados para diferentes partes de um sistema. Dessa forma, cada assembly pode ter implementações internas que não precisam ser acessíveis fora do seu escopo.
4. **Amplamente Utilizado em Bibliotecas (Libraries):** Em bibliotecas (libraries) e frameworks, o internal é frequentemente utilizado para membros que são parte da implementação interna, mas não devem ser expostos publicamente para os consumidores da biblioteca.

```
// Biblioteca
0 references
internal class ImplementacaoInterna
{
    0 references
    internal void MetodoInterno()
    {
        // Código interno
    }
}
```

```
// Consumidor da Biblioteca
0 references
public class Consumidor
{
    0 references
    void Exemplo()
    {
        // Acesso a membros internos não é permitido fora do assembly da biblioteca
        ImplementacaoInterna implementacao = new ImplementacaoInterna();
        implementacao.MetodoInterno(); // Erro de compilação
    }
}
```

5. **Amplamente Utilizado em Aplicações Grandes:** Em grandes aplicações, onde há vários componentes e módulos separados, o “**internal**” é uma ferramenta útil para organizar o código e controlar o acesso a partes específicas do sistema.

1.2.4. public

Em C#, o modificador de acesso “**public**” é um dos modificadores usados para especificar a visibilidade de membros (métodos, propriedades, campos, classes etc.) em um programa. Quando um membro é marcado como “**public**”,

ele pode ser acessado de qualquer lugar dentro do mesmo assembly (unidade de compilação) ou de outros assemblies.

Aqui estão alguns pontos-chave sobre o modificador de acesso “**public**”:

1. **Acessibilidade Global:** O modificador public torna o membro ao qual está associado acessível de forma global. Isso significa que o membro pode ser acessado por outras classes ou componentes fora da classe que o contém, desde que essas classes estejam no mesmo assembly (projeto ou biblioteca) ou em assemblies diferentes, dependendo das configurações de visibilidade.

2. **Exemplo de Uso em uma Classe:**

```
0 references
public class MinhaClasse
{
    // Membro público (campo)
    public int MeuCampoPublico;

    // Membro público (método)
    0 references
    public void MeuMetodoPublico()
    {
        // código aqui
    }
}
```

3. **Acesso de Outras Classes:** Outras classes podem criar instâncias de “**MinhaClasse”** e acessar seus membros públicos.

```
MinhaClasse instancia = new MinhaClasse();
instancia.MeuCampoPublico = 42;
instancia.MeuMetodoPublico();
```

4. **Restrições em Ambientes de Produção:** É importante considerar a segurança e a encapsulação ao usar o modificador “**public**”. Normalmente, em ambientes de produção, é uma boa prática manter apenas o que é necessário como “**public**” e limitar a exposição de detalhes de implementação.

5. **Unidade de Compilação (Assembly):** O escopo de visibilidade do “**public**” está relacionado à unidade de compilação, que geralmente é um assembly no contexto do C#. A visibilidade “**public**” é mais ampla que “**internal**” (visível apenas dentro do

mesmo assembly), mas mais restrita do que “**protected**” (visível dentro da classe e suas subclasses).

```
0 references
15  public class TesteAcessoPublic()
16  {
        0 references
17      public void TesteInstancia()
18      {
19          MinhaClasse instancia = new MinhaClasse();
20          instancia.MeuCampoPublico = 42;
21          instancia.MeuMetodoPublico();
22      }
23  }
```

Ao usar o modificador “**public**", é fundamental equilibrar a acessibilidade com a encapsulação, garantindo que apenas o necessário seja exposto para promover uma boa prática de design de software.

1.2.5. Combinações de Controladores

Além dos controladores explorados anteriormente, temos a possibilidade de combinar **protected** com **private** e **internal**. Essa abordagem nada mais é do que combinar as características dos dois controladores para algumas abordagens específicas.

O modificador **protected internal** é uma ferramenta poderosa para proporcionar uma forma de acesso mais ampla a membros em cenários específicos. Ele é útil quando você deseja permitir o acesso a membros tanto em herança quanto no contexto de um mesmo assembly.

O modificador **private protected** é uma adição específica para casos em que você precisa restringir o acesso a membros apenas a classes derivadas dentro do mesmo assembly. Sua utilização é útil quando é necessário um controle rigoroso sobre o acesso em cenários específicos de design de classes.

2. PALAVRAS CHAVES

Palavras chaves podem ser chamadas de palavras reservadas. **Mas o que isso quer dizer?** – Quer dizer que não podemos utilizar elas para definirmos por exemplo nomes de variáveis e/ou métodos.

Quando falamos de palavras chaves em C#, nós temos duas distribuições diferentes, sendo elas:

1. Identificadores Reservados:

abstract	event	namespace	static	char	implicit	foreach	readonly	private	typeof
as	explicit	new	string	checked	in	goto	ref	protected	uint
base	extern	null	struct	class	int	if	return	public	ulong
bool	false	object	switch	const	interface	while	sbyte	ushort	unchecked
break	finally	operator	this	continue	internal	do	sealed	using	unsafe
byte	fixed	out	throw	decimal	is	double	short	virtual	
case	float	override	true	default	lock	else	sizeof	void	
catch	for	params	try	delegate	long	enum	stackalloc	volatile	

2. Identificadores Contextuais:

add	descending	init	notnull	remove	var
and	dynamic	into	nuint	required	when (filter condition)
alias	equals	join	on	scoped	where (generic type constraint)
ascending	file	let	or	select	where (query clause)
args	from	managed (function pointer calling convention)	orderby	set	with
async	get	nameof	partial (type)	unmanaged (function pointer calling convention)	yield
await	global	nint	partial (method)	unmanaged (generic type constraint)	
by	group	not	record	value	

Uma curiosidade sobre as palavras reservadas é que podemos utilizar elas se estiverem acompanhadas do “@” (exemplo abaixo).

```
0 references
public class MinhaClasse
{
    0 references
    public void PalavraReservada()
    {
        bool @bool = true;

        if (@bool)
        {
            Console.WriteLine("Palavra reservada");
        }

    }
}
```

3. CONSTRUTORES

Os construtores em C# são métodos especiais dentro de uma classe que são chamados automaticamente quando uma instância da classe é criada. Eles são responsáveis por inicializar o estado do objeto e executar qualquer lógica necessária durante a criação da instância. Aqui estão alguns pontos importantes sobre os construtores em C#:

1. Construtores possuem o mesmo nome da classe;

```
//Construtor precisa ter o nome da classe
2 references
public class Construtores
{
    1 reference
    public int Id { get; set; }
    0 references
    public string Nome { get; set; }
    0 references
    public string Endereco { get; set; }

    /// <summary>
    /// Construtor padrão, também considerado nessa estrutura caso não seja declarado
    /// </summary>
    0 references
    public Construtores()
    {
        //Não possui retorno
    }
```

2. Construtores não possuem um tipo de retorno;

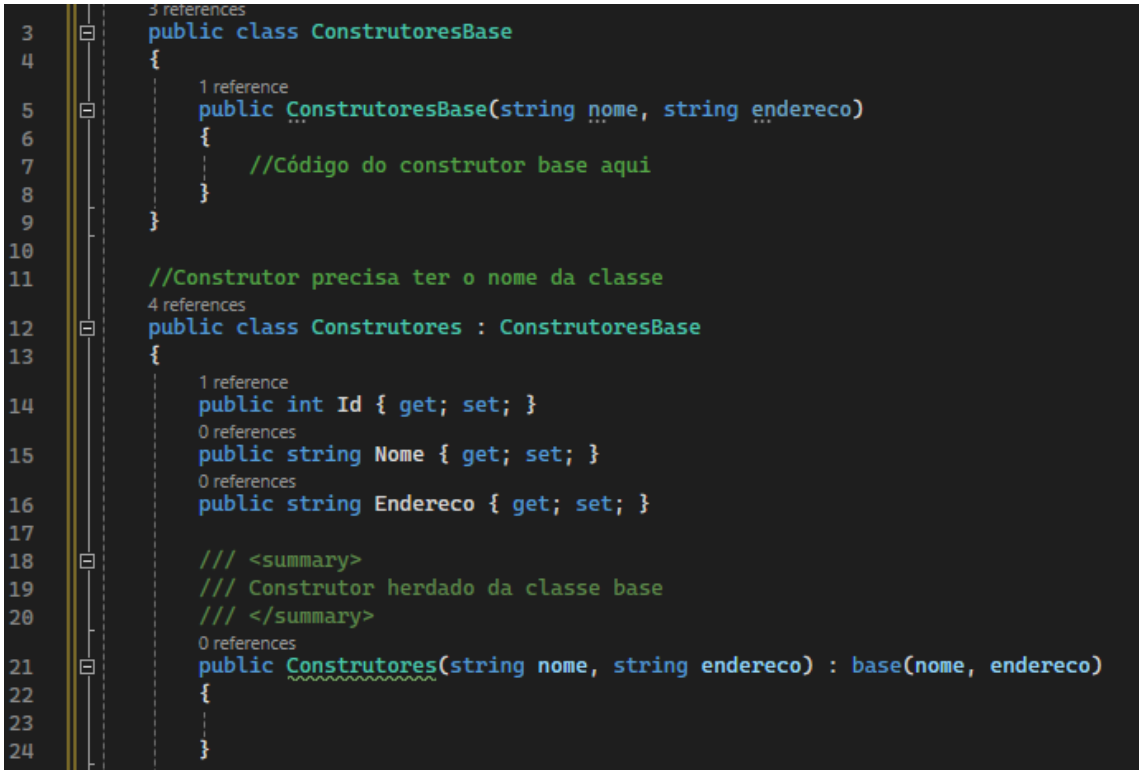
3. É possível ter mais de um construtor na mesma classe com entrada de parâmetros diferentes;

```
/// <summary>
/// Construtor padrão, também considerado nessa estrutura caso não seja declarado
/// </summary>
0 references
public Construtores()
{
    //Não possui retorno
}

/// <summary>
/// Construtor que recebe um Id como parâmetro
/// </summary>
0 references
public Construtores(int id)
{
    Id = id;
    //Não possui retorno
}
```

4. É possível chamar um construtor através de outro construtor;

```
/// <summary>
/// Construtor que recebe um Id como parâmetro
/// </summary>
1 reference
public Construtores(int id)
{
    Id = id;
    //Não possui retorno
}

/// <summary>
/// Construtor chamando um outro construtor
/// </summary>
0 references
public Construtores() : this(42) { }
```

5. Se eu não definir um construtor, um construtor padrão é considerado automaticamente para a classe;
6. É possível herdar um construtor de uma classe base.



```
3 references
public class ConstrutoresBase
{
    1 reference
    public ConstrutoresBase(string nome, string endereco)
    {
        //Código do construtor base aqui
    }
}

//Construtor precisa ter o nome da classe
4 references
public class Construtores : ConstrutoresBase
{
    1 reference
    public int Id { get; set; }
    0 references
    public string Nome { get; set; }
    0 references
    public string Endereco { get; set; }

    /// <summary>
    /// Construtor herdado da classe base
    /// </summary>
    0 references
    public Construtores(string nome, string endereco) : base(nome, endereco)
    {

    }
```

Os construtores desempenham um papel crucial na inicialização de objetos em C# e são fundamentais para garantir que os objetos sejam criados de maneira consistente e pronta para uso. Eles também ajudam a garantir que a lógica de inicialização seja encapsulada dentro da própria classe.

4. NAMESPACES

Namespaces são utilizados exclusivamente para organização e estruturação de código em C#. A ideia por trás dos Namespaces é garantir que não teremos conflitos de nomenclatura, além de promover uma estrutura de código mais limpa e modularizada.

Pensando nessas características, conseguimos inferir que: Se temos um código organizado e modularizado com os namespaces, podemos localizar os tipos de classes de maneira mais fácil.

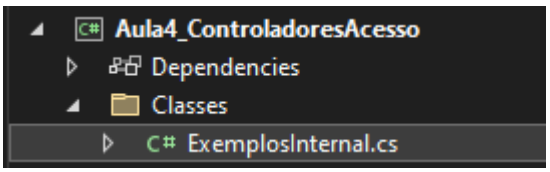
Quando falamos de Namespaces, a Microsoft já cria os mesmo de acordo com a nossa estruturação de arquivos, cabendo ao desenvolvedor mudar ou não o seu nome. Contudo, é importante nos atentarmos as boas práticas de

nomenclatura dos namespaces. Os nomes de namespaces devem seguir as boas práticas de nomenclatura, usando PascalCase (cada palavra começando com maiúscula) e escolhendo nomes significativos e descritivos.

Veja um exemplo de uma classe que criamos nos tópicos anteriores deste documento:

Os namespaces por padrão possuem o nome do projeto e o nome da pasta em que estão. No exemplo acima podemos entender então que a estrutura é:

[NOME_PROJETO].[NOME_PASTA]....



Por padrão, não é comum mudarmos por conta o nome dos Namespaces que são criados quando criamos um arquivo no nosso projeto, já que a IDE do VS já se encarrega de seguir as boas práticas.