C#: Modificadores de Acesso e Escopo

Fundamentos, Problemas e Implementações

Objetivos de Aprendizagem

1 **Diferenciar Escopo e Acesso**

Compreender onde um símbolo é visível versus quem está autorizado a usá-lo

2 **Dominar Modificadores de Acesso**

Aplicar corretamente public, private, protected, internal e suas variações

3 **Entender Restrições por Contexto**

Conhecer limitações entre tipos top-level, aninhados, membros e interfaces

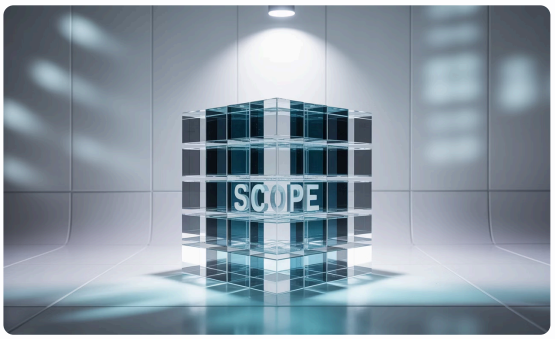
4 **Projetar APIs Eficientes**

Criar superfícies mínimas mantendo encapsulamento e testabilidade

Conceitos Fundamentais

Escopo

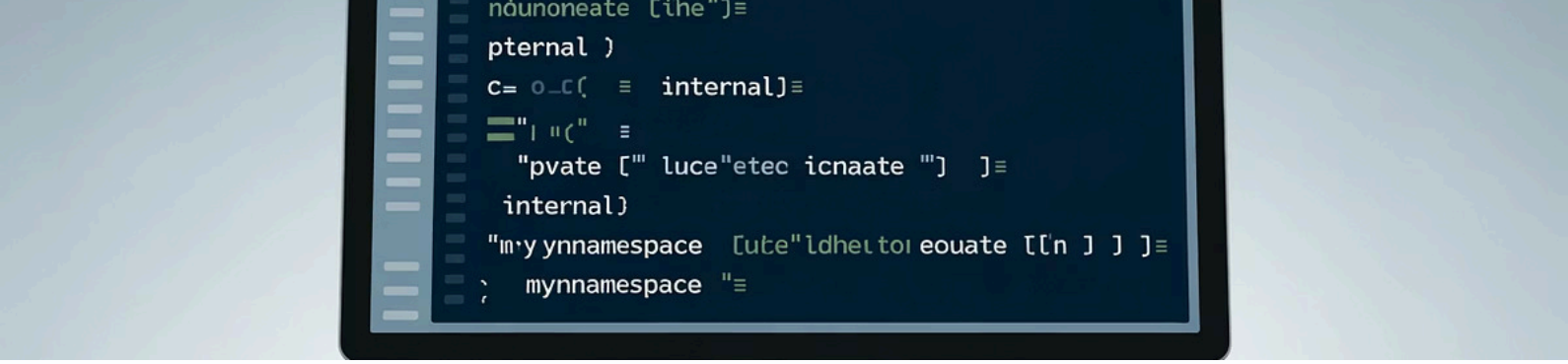
A região do código onde um identificador é **visível**. Por exemplo, uma variável local dentro de um bloco { ... }.



Acesso

O **grau de exposição** de um tipo ou membro a partir de outros tipos e assemblies, controlado pelos modificadores de acesso.

Regra Prática: Primeiro decida onde você precisa ver o símbolo (escopo), depois quem poderá usá-lo (acesso). Sempre prefira o menor acesso necessário.



Modificadores de Acesso - Visão Geral

public

Acessível por **qualquer** código em qualquer lugar

private

Acessível **somente dentro do tipo** que o declara

protected

Acessível dentro do tipo **e das subclasses**

internal

Acessível apenas dentro do mesmo assembly

Modificadores Avançados

protected internal

Acessível **ou** por estar no mesmo assembly **ou** por herança (união dos dois conjuntos)

private protected

Acessível apenas por **subclasses que estejam no mesmo assembly** (interseção: herança e mesmo assembly)

file

Tipo top-level visível **somente dentro do mesmo arquivo fonte** (C# 11+)

Padrões e Restrições por Contexto

Contexto	Modificadores Permitidos
Tipo top-level (namespace)	public, internal, file
Tipo aninhado	Todos os modificadores
Membros de classe/struct/record	Todos (padrão: private)
Membros de interface	Sempre public (implícito)
Enumerações	Membros são públicos, enum segue regras de tipo

Por Que Isso Importa?

Encapsulamento e Invariantes

Restringir o acesso evita que regras internas sejam quebradas por consumidores externos

Evolução Segura

Uma API com superfície reduzida sofre menos breaking changes durante atualizações

Coesão e Clareza

Quem lê o código sabe o que é público (contrato) versus interno (implementação)

Testabilidade Madura

Com internal + InternalsVisibleTo, expõe-se apenas o necessário aos testes

Problemas ao Não Usar Adequadamente

Vazamento de Detalhes Internos

Membros expostos à toa viram dependências externas e **engessam** refatorações futuras

Quebra de Invariantes

Estados que deveriam ser protegidos tornam-se alteráveis de fora, causando bugs

Superfície Pública Inchada

Confunde usuários, dificulta documentação e revisão de código

Conflitos Entre Projetos

Ausência de internal estimula "acessos cruzados" frágeis entre assemblies

Benefícios da Aplicação Correta

API Limpa e Estável

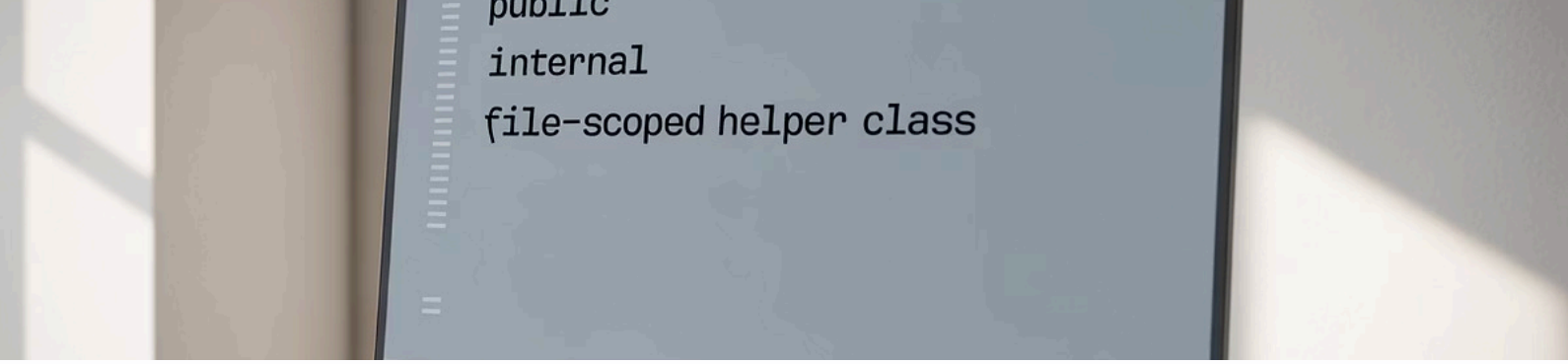
O que é público comunica a intenção; o resto pode evoluir livremente

Menos Bugs

Reduz significativamente a chance de uso indevido do código

Organização por Camadas

Domínio exposto no que precisa; infraestrutura e utilitários permanecem internos



Analogia do Cotidiano

Casa Residencial

- **Porta da rua** é public - qualquer um pode ver
- **Escritório trancado** é private - só o dono acessa
- **Quarto da família** é protected - acesso por parentes/herdeiros
- **Condomínio** é internal - moradores do mesmo prédio/assembly
- **Sala reservada à família moradora** seria private protected
- **Visitas com passe livre ou moradores ou parentes** têm protected internal

Empresa: website (public), intranet (internal), recursos de um departamento (private dentro do tipo), recursos compartilhados a times herdeiros (protected).

Boas Práticas Essenciais

01

Menor Acesso Necessário

Comece sempre com private/internal, promovendo a public só quando houver caso de uso real

02

Evite protected internal em Bibliotecas

É ambíguo. Prefira composições claras ou API pública explícita

03

Use InternalsVisibleTo para Testes

Mantenha o código como internal e autorize apenas o projeto de testes

04

Use file para Helpers Locais

Evita "vazar" tipos auxiliares para o assembly inteiro

05

Prefira Composição

Só exponha protected quando realmente for extensão por herança

06

Documente o Contrato Público

Trate o resto como implementação privada

Implementação: Tipos Top-Level

Demonstração prática dos diferentes níveis de acesso para tipos no nível de namespace:

```
// Arquivo: Helpers.cs
file sealed class FileOnlyHelper // visível apenas neste arquivo
{
 public static string Normalize(string s) => s.Trim();
}

// Arquivo: Services.cs
namespace MyApp.Services;

public sealed class PublicService { /* contrato público */ }
internal sealed class InternalUtility { /* uso interno ao assembly */ }
```

Implementação: Membros com Diferentes Acessos

```
public sealed class BankAccount
{
 private decimal _balance; // apenas a classe manipula diretamente
 public string Number { get; } // parte do contrato público
 protected virtual decimal Fee => 1.75m; // extensível por herança

 internal BankAccount(string number, decimal opening)
 {
   Number = number;
   _balance = opening;
 }

 public void Deposit(decimal amount) => _balance += amount;

 public bool Withdraw(decimal amount)
 {
   var total = amount + Fee;
   if (total > _balance) return false;
   _balance -= total;
   return true;
 }
}
```

Implementação: protected internal vs private protected

```
public class Base
{
    protected internal int A;
    // disponível por herança OU mesmo assembly

    private protected int B;
    // disponível por herança E mesmo assembly
}
```

```
public class DerivadaMesmoAssembly : Base
{
    void Teste()
    {
        A = 1; // ok (herança)
        B = 2; // ok (herança e mesmo assembly)
    }
}
```

```
// Em outro assembly (biblioteca diferente)
public class DerivadaOutroAssembly : Base
{
 void Teste()
 {
    A = 3; // ok (herança em outro assembly)
    // B = 4; // ERRO: não está no mesmo assembly
 }
}
```

Implementação: Tipos Aninhados Privados

```
public class JwtToken
{
    public string Value { get; }

    public JwtToken(string value) => Value = value;
```

```
    // helper só para implementação interna do tipo
    private static class Clock
    {
        public static DateTime UtcNow => DateTime.UtcNow;
    }
}
```

Tipos aninhados privados são úteis para organizar código auxiliar que só faz sentido dentro do contexto da classe principal.

Implementação: Padrão para Testes

```
// AssemblyInfo.cs (ou no .csproj via ItemGroup)
using System.Runtime.CompilerServices;

[assembly: InternalsVisibleTo("MyApp.Tests")]
// autoriza testes a ver 'internal'

namespace MyApp.Core;

internal sealed class CpfValidator
{
    public static bool IsValid(string value)
    {
        /* ... */
        return true;
    }
}
```

Esta abordagem mantém a API pública limpa enquanto permite testes abrangentes dos componentes internos.

Implementação: Escopo de Variáveis

```
void Process()
{
 int x = 10; // escopo é o bloco do método
 {
    int y = 20; // escopo é o bloco interno
    Console.WriteLine(x + y);
 }
```

```
using var stream = File.OpenRead("data.txt");
// escopo até o fim do bloco

if (obj is string s)
// 's' existe só no escopo do 'if' e ramos onde foi atribuído
{
   Console.WriteLine(s.Length);
}
}
```

Checklist de Revisão Rápida

O **acesso mínimo** foi aplicado (comece por private/internal)?	O que é **público** é realmente parte do **contrato**?	Preciso mesmo de herança? Se sim, protected está justificado?
Evitei protected internal sem necessidade?	Usei file para helpers locais ao arquivo?	Para testes, preferi internal + InternalsVisibleTo em vez de tornar tudo público?

Exercícios Práticos

Reestruturação de Classe

Reestruture uma classe "inchada" expondo apenas métodos públicos essenciais; torne campos/métodos auxiliares private

Separação em Camadas

Separe um projeto em camadas (Domain, App, Infra) e aplique public/internal de forma criteriosa

Tipo file Local

Crie um tipo top-level file para um helper usado por um único arquivo e demonstre que ele não é visível em outro arquivo

Herança com protected

Transforme um membro public em protected justificando com herança real; discuta prós/cons em equipe

Configuração de Testes

Habilite InternalsVisibleTo apenas para o projeto de testes e mantenha o restante internal

Conclusão

Projetar o **acesso** é tão importante quanto escrever o **código**. Ao aplicar o **menor acesso necessário**, você protege invariantes, simplifica a API pública e permite que o sistema **evolua** com menos fricção.

Combine isso com uma boa estrutura de pastas/assemblies e você terá bases sólidas para um projeto sustentável.

> "O código que você não expõe é o código que você pode mudar livremente."

7	1
Modificadores	Princípio
Diferentes níveis de acesso disponíveis	Menor acesso necessário

∞

Benefícios

Para manutenibilidade do código