Métodos estáticos em C# — Fundamentos, Boas Práticas e Labs

Continuidade do nosso material: abordagem didática, exemplos do cotidiano, mapeamento para C# e roteiro de prática.

Objetivos de aprendizagem

1 Explicar métodos estáticos

Compreender o que são **métodos estáticos** e quando preferi-los a membros de instância.

2 Diferenciar conceitos estáticos

Entender as diferenças entre **classe estática**, **campo/propriedade estáticos**, **construtor estático** e **método estático**.

3 Aplicar estáticos corretamente

Implementar estáticos sem criar **estado global acoplado** (pitfalls e como evitar).

4 Criar métodos de extensão

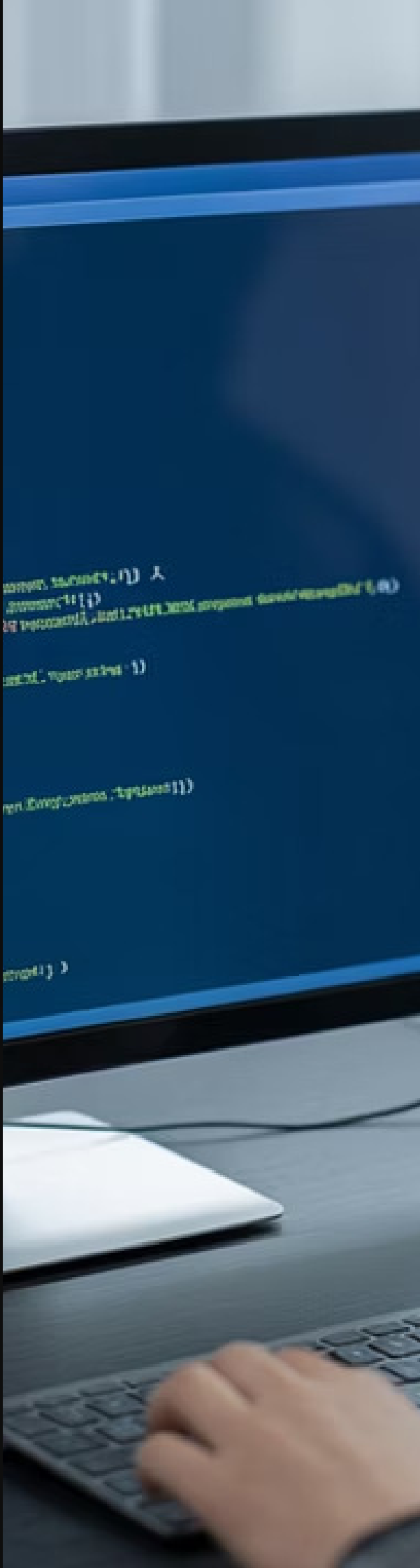
Desenvolver e utilizar **métodos de extensão** (que são estáticos) para enriquecer tipos existentes.

5 Escolher constantes apropriadas

Decidir entre const e static readonly e entender suas implicações.

6 Escrever código thread-safe

Implementar código **thread-safe** quando houver estado estático.



Motivação didática

Calculadora de bolso

Não guarda estado entre operações; você passa os números e obtém o resultado → excelente analogia para **método estático puro**.

Placar compartilhado

Um placar na parede é único para todos (estado global). Se qualquer um altera, todos veem a mudança → cuidado com **campo estático mutável**.

Constantes do sistema

CEP do Brasil ter 8 dígitos, taxa fixa de um cálculo didático, fuso de referência para log → **const** ou static readonly.

Resumo pedagógico

- Use **métodos estáticos puros** para **cálculo/transformação sem estado** (previsíveis, fáceis de testar).
- Evite **estado estático mutável** salvo quando estritamente necessário (ex.: caches devidamente sincronizados).

Conceitos essenciais

Método estático

Pertence ao **tipo**, não ao objeto. Invocado como Tipo.Metodo().

Campo/Propriedade estáticos

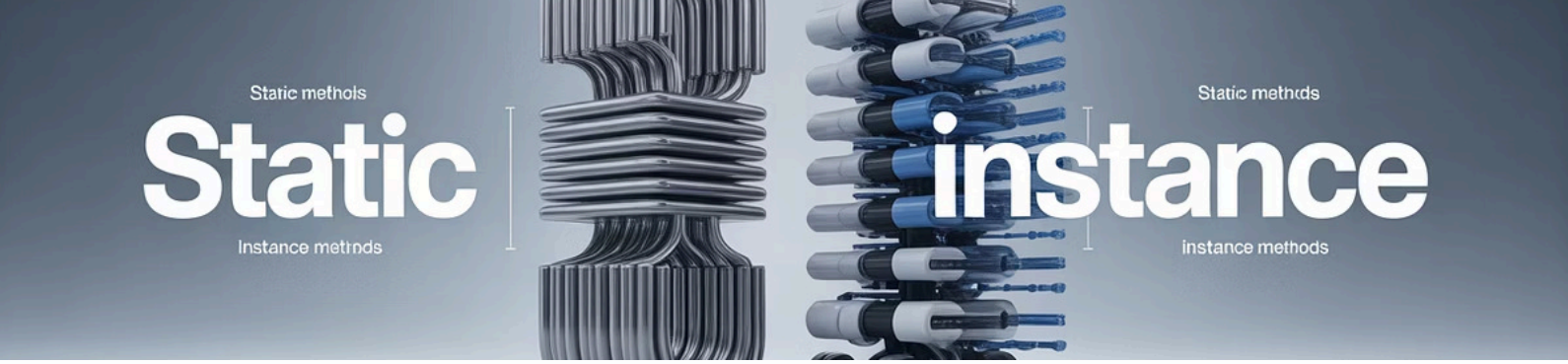
Um único valor **compartilhado** por todo o app.

Classe estática

static class. Não pode ser instanciada nem herdada; tudo dentro é estático.

Construtor estático

static Tipo(); executa **uma vez** antes do primeiro uso, ideal para inicialização de dados estáticos.



Métodos estáticos vs. de instância

Quando preferir estático

- Regra **pura**: não depende do estado do objeto (ex.: conversões, validações).
- Utilitários **determinísticos**: mesma entrada → mesma saída.

Quando preferir instância

- A operação depende do **estado do objeto** ou altera esse estado (ex.: pedido.Total(), conta.Sacar()).

Exemplo comparativo

```
public class ConversorTemperatura
{
   public double Celsius { get; }
   public ConversorTemperatura(double celsius) => Celsius = celsius;
   public double EmFahrenheit() => (
     Celsius * 9.0 / 5.0) + 32.0; // depende do estado
}

public static class Temp
{
   public static double CelsiusParaFahrenheit(double c) =>
     (c * 9.0 / 5.0) + 32.0; // puro
}
```

Instância

Guarda estado (Celsius)

Método usa o estado interno

Estático

Sem estado

Recebe todos os dados como parâmetros

Classe estática e inicialização

```
public static class AppInfo
{
   public static readonly string Version;
   public static readonly DateTime BuildDate;

   static AppInfo() // construtor estático: roda 1x
   {
      Version = Environment.GetEnvironmentVariable("APP_VERSION")
       ?? "dev";
      BuildDate = DateTime.UtcNow; // exemplo didático
   }
}
```

> ⓘ **Observação:** readonly permite definir em tempo de execução e impede reatribuição depois da construção.

Características do construtor estático

- Executado automaticamente antes do primeiro acesso à classe
- Executado apenas uma vez durante a vida do aplicativo
- Não pode ser chamado diretamente pelo código
- Não pode ter parâmetros
- Ideal para inicializar campos estáticos que precisam de lógica

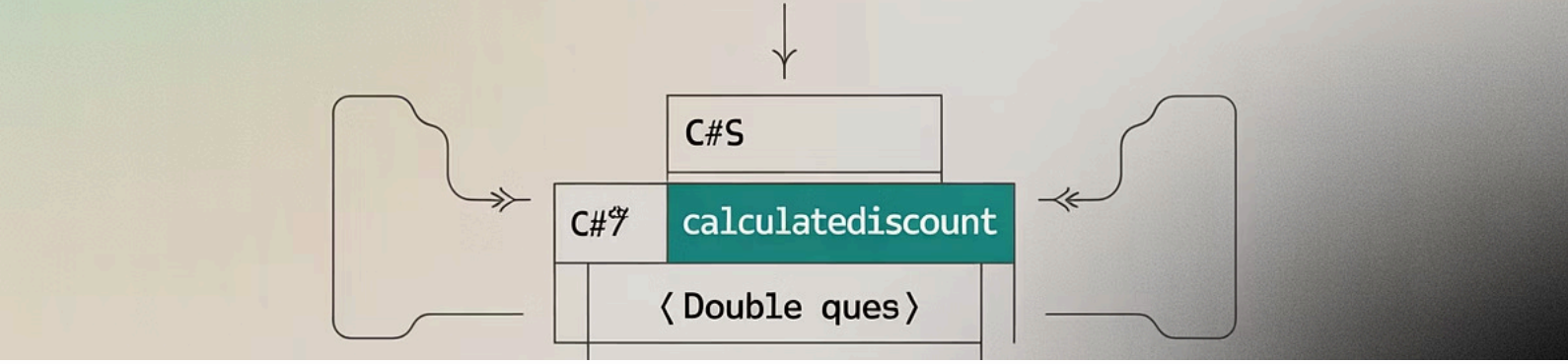
const vs static readonly

const

- Valor **literal** em tempo de compilação (primitivos/strings)
- Mudança exige **recompilar** consumidores
- Mais eficiente (substituído em tempo de compilação)
- Limitado a tipos primitivos e string

static readonly

- Definido em tempo de execução (construtor estático)
- Útil para Guid, DateTime, resultados de cálculo
- Pode ser inicializado com valores não literais
- Suporta qualquer tipo de referência ou valor




```
public static class Finance
{
 public const decimal IOFPercent = 0.0638m; // literal, estável
 public static readonly DateTime InicioAnoFiscal =
    new DateTime(DateTime.UtcNow.Year, 1, 1); // calculado em
      // runtime
}
```

Cuidado com estado estático (pitfalls)

Anti-padrões comuns

"God Class"/Utility bag

Uma classe estática com funções aleatórias e responsabilidades misturadas.

Estado global mutável

Campos estáticos modificáveis usados como atalho para comunicação entre partes do sistema.

Riscos

- Dificulta testes (ordem de execução, interferência entre casos).
- Introduz bugs sutis em **concorrência**.
- Aumenta acoplamento e reduz clareza arquitetural.

Se precisar de estado estático

- Garanta **thread-safety**. Ex.: locks ou APIs atômicas (Interlocked).
- Documente **quem pode escrever** e **por quê**.

```
public static class Placar
{
 private static int _pontos;
 public static int Pontos => _pontos;
 public static void MarcarPonto() => Interlocked.Increment(ref _pontos);
}
```

Métodos de extensão (são estáticos!)

Permitem **adicionar comportamentos** a tipos existentes **sem herdar** nem alterar o código original.

```
public static class DecimalExtensions
{
    private static readonly System.Globalization.CultureInfo PtBr =
        new("pt-BR");
    public static string ToBrl(this decimal valor)
        => valor.ToString("C", PtBr);
}

// Uso
decimal total = 123.45m;
Console.WriteLine(total.ToBrl()); // R$ 123,45
```

Outro exemplo simples:

```
public static class IntExtensions
{
    public static bool IsBetween(this int value, int min, int max)
        => value >= min && value <= max;
}

int idade = 17;
Console.WriteLine(idade.IsBetween(0, 18)); // True
```

Regras: a classe deve ser static, o método também, e o **primeiro parâmetro** leva this Tipo alvo.

Boas práticas

Faça

- Prefira estáticos **puros** para funções matemáticas, conversores, validadores, formatações.
- Agrupe por **responsabilidade única** (ex.: MoneyFormat, CpfValidator).
- Em libs públicas, prefira **métodos de extensão** com nomes claros.

Evite

- Guardar **estado mutável** em estáticos (especialmente coleções).
- Injetar dependências ocultas via estático (ex.: Logger.Global, Db.Global). Prefira **injeção de dependência**.
- Usar estáticos para **contornar design** (atalhos que depois viram dívidas).

Funções puras

Métodos estáticos são ideais para operações matemáticas e transformações sem estado.

Organização por domínio

Agrupe métodos estáticos por área funcional, não em classes utilitárias genéricas.

Preferir DI a estáticos

Para serviços com estado ou dependências, use injeção de dependência em vez de acessos estáticos.

Labs guiados (incrementais)

Copie os trechos para o seu Program.cs ou use arquivos separados.

Lab A — Biblioteca utilitária pura

Crie static class MathUtil com: - Percent(decimal parte, decimal total) → (parte / total) * 100 (trate divisão por zero retornando 0 ou lançando exceção conforme sua diretriz). - Clamp(int valor, int min, int max) → limita o valor ao intervalo.

Tarefas

1. Escreva 5 casos de teste no console.
2. Explique por que são **puros** e fáceis de testar.

Lab B — const vs static readonly

Crie uma classe CalendarioFiscal com const int MesInicio = 1 e static readonly DateTime InicioAtual calculado no construtor estático. Imprima ambos e discuta quando cada um deve mudar.

Lab C — Extensões úteis

Implemente:

- ToBrl(this decimal) (do exemplo)
- Slugify(this string) (troque espaços por -, minúsculas, remova acentos simples)
- IsBetween(this DateTime, DateTime, DateTime)

Mostre 3 usos no console.

Lab D — Estado estático thread-safe

Implemente ContadorGlobal com _valor estático e métodos Incrementar() usando Interlocked.Increment. Dispare 50 Task.Run(Incrementar) e comprove que o valor final é 50.

Lab E — Refatoração para instância/DI

Comece com um Logger.Global estático que escreve em arquivo (anti-padrão). Refatore para ILogger + FileLogger de instância, e passe por **injeção** a quem precisa. Descreva vantagens para **testes** e **acoplamento**.

Exercícios com saída esperada

1

Clamping

MathUtil.Clamp(130, 0, 100)

Saída esperada: 100

2

Percentual

MathUtil.Percent(25, 200)

Saída esperada: 12,5

3

Extensão de intervalo

17.IsBetween(18, 60)

Saída esperada: False

4

Thread-safe

Após 1.000 incrementos paralelos, o ContadorGlobal.Valor deve ser 1000.

Checklist de consolidação

- [ ] **Métodos estáticos puros**

  Sei identificar quando um método pode ser estático **puro**.

- [ ] **Estado global seguro**

  Sei evitar estado global e, se necessário, torná-lo **thread-safe**.

- [ ] **Métodos de extensão**

  Sei criar **métodos de extensão** e quando usá-los.

- [ ] **Constantes**

  Diferencio const de static readonly.

Refatoração

Sei refatorar de estático para **instância + DI** quando o domínio pede.

Próximos passos

Padrões complementares

Singleton (com parcimônia) vs. **DI**.

Extensões avançadas

Extensões avançadas com **LINQ** e coleções.

Benchmarks

Benchmarks rápidos (BenchmarkDotNet) comparando abordagens puras e com alocação.

Implementação do Lab A

```
public static class MathUtil
{
    public static decimal Percent(decimal parte, decimal total)
    {
        if (total == 0)
            return 0; // ou throw new DivideByZeroException();

        return (parte / total) * 100;
    }

    public static int Clamp(int valor, int min, int max)
    {
        if (valor < min) return min;
        if (valor > max) return max;
        return valor;
    }
}

// Testes
Console.WriteLine($"Percent(25, 200) = {MathUtil.Percent(25, 200)}");
Console.WriteLine($"Percent(200, 100) = {MathUtil.Percent(200, 100)}");
Console.WriteLine($"Percent(0, 100) = {MathUtil.Percent(0, 100)}");
Console.WriteLine($"Percent(50, 0) = {MathUtil.Percent(50, 0)}");
Console.WriteLine($"Clamp(130, 0, 100) = {MathUtil.Clamp(130, 0, 100)}");
```

Estes métodos são **puros** porque:

- Não dependem de estado externo
- Para as mesmas entradas, sempre produzem as mesmas saídas
- Não têm efeitos colaterais
- São facilmente testáveis com valores de entrada e saída esperada

Implementação do Lab C

```
public static class StringExtensions
{
  public static string Slugify(this string texto)
  {
     if (string.IsNullOrEmpty(texto))
        return string.Empty;

  // Simplificação didática - em produção use uma biblioteca
  // para lidar com todos os casos de normalização Unicode
     string semAcentos = new string(
       texto.Normalize(System.Text.NormalizationForm.FormD)
       .Where(c => System.Globalization.CharUnicodeInfo
       .GetUnicodeCategory(c) != System.Globalization.UnicodeCategory
       .NonSpacingMark).ToArray());

     return semAcentos.ToLower().Replace(" ", "-").Replace(".", "")
        .Replace(",", "");
  }
}

public static class DateTimeExtensions
{
  public static bool IsBetween(this DateTime date, DateTime start,
       DateTime end)
  {
     return date >= start && date <= end;
  }
}

// Testes
string titulo = "Métodos Estáticos em C#";
Console.WriteLine($"Slug: {titulo.Slugify()}");
```

```
DateTime hoje = DateTime.Now;
DateTime inicio = hoje.AddDays(-5);
DateTime fim = hoje.AddDays(5);
Console.WriteLine($"Hoje está entre {inicio:d} e {fim:d}?
  {hoje.IsBetween(inicio, fim)}");

decimal valor = 1234.56m;
Console.WriteLine($"Valor em BRL: {valor.ToBrl()}");
```

Implementação do Lab D

```
public static class ContadorGlobal
{
    private static int _valor;
    public static int Valor => _valor;

    public static void Incrementar()
    {
        Interlocked.Increment(ref _valor);
    }
    public static void Reset()
    {
        Interlocked.Exchange(ref _valor, 0);
    }
}

// Teste de concorrência
async Task TestarContadorAsync()
{
    ContadorGlobal.Reset();

    var tarefas = new List();
    for (int i = 0; i < 1000; i++)
    {
        tarefas.Add(Task.Run(() => ContadorGlobal.Incrementar()));
    }

    await Task.WhenAll(tarefas);

    Console.WriteLine($"Valor final: {ContadorGlobal.Valor}");
    // Deve imprimir "Valor final: 1000"
}

await TestarContadorAsync();
```

⚠ Sem o uso de Interlocked, o resultado seria imprevisível devido a condições de corrida entre threads!

Fechamento

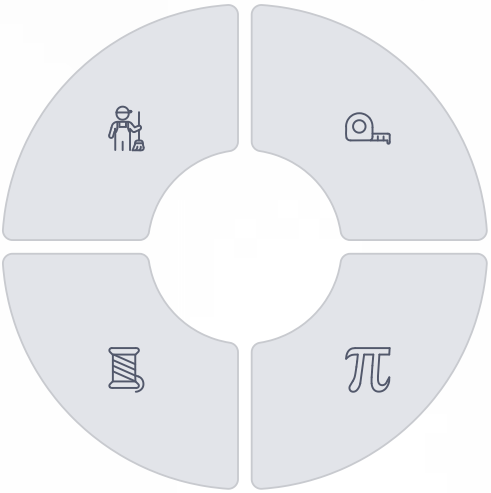
> Métodos estáticos são ideais para **operações puras** e utilitários de alto reuso. Usados com critério, simplificam o design e a testabilidade. O segredo está em separar o que é **cálculo** (estático) do que é **comportamento de objeto** (instância), evitando estado global e preservando a clareza arquitetural.

Métodos Puros

Operações determinísticas sem estado

Extensões

Enriquecem tipos existentes



Thread-Safety

Proteção para estado compartilhado

Constantes

Valores imutáveis compartilhados

Lembre-se: o uso adequado de métodos estáticos melhora a legibilidade, manutenção e testabilidade do seu código C#, desde que aplicado nos contextos corretos e seguindo as boas práticas apresentadas.