

Programação Orientada a Objetos em C#: Fundamentos para Desenvolvedores C

Este documento apresenta uma introdução completa à Programação Orientada a Objetos (POO) usando C#, especialmente pensada para desenvolvedores que vêm da programação estruturada em C. Abordamos desde os conceitos fundamentais até exemplos práticos, auxiliando você na transição do paradigma estruturado para o orientado a objetos, com foco na consolidação de boas práticas e técnicas essenciais.

Por que Orientação a Objetos hoje?

A **Programação Orientada a Objetos (POO)** revolucionou o desenvolvimento de software ao aproximar o código do domínio real dos problemas. Em vez de espalhar variáveis e funções soltas pelo código, como no paradigma estruturado, a POO permite modelar **tipos** que reúnem **estado** (dados) e **comportamento** (regras), com limites claros de responsabilidade.

Legibilidade e Manutenibilidade

Classes com nomes significativos mapeiam conceitos do problema diretamente para o código, tornando-o mais intuitivo e fácil de entender.

Encapsulamento

Proteção do estado interno e das regras, evitando estados inválidos e escondendo a complexidade da implementação.

Reuso

Objetos colaboram entre si; comportamentos são organizados em métodos e tipos especializados, facilitando o reaproveitamento.

Evolução Segura

Invariante e contratos tornam mudanças menos arriscadas, pois as regras de negócio estão protegidas dentro dos objetos.

Ponte mental: Em C você juntava dados (structs) + funções. Em POO, você cria um **tipo** que embala dados **e** funções (agora chamadas de **métodos**) sob **regras** claras de manipulação.

Este novo paradigma é especialmente valioso em sistemas complexos, onde a modelagem adequada reduz significativamente a complexidade acidental e torna o software mais resiliente a mudanças. À medida que os sistemas crescem, os benefícios da POO se tornam ainda mais evidentes, permitindo abstrações mais poderosas e facilitando o trabalho em equipe.

.NET e C# em 15 minutos

Antes de mergulharmos nos conceitos de POO, é importante entender o básico sobre a plataforma .NET e a linguagem C# — focando apenas no que realmente importa para nossa jornada de aprendizado.

.NET SDK

O **SDK (Software Development Kit)** do .NET fornece tudo o que você precisa para desenvolver aplicações C#:

- Compilador (transforma código C# em IL)
- Runtime (executa as aplicações compiladas)
- Ferramentas de linha de comando (CLI)
- Bibliotecas base do framework

Comandos Essenciais

```
# Criar um novo projeto de console  
dotnet new console -n AulaPOO
```

```
# Navegar para a pasta do projeto  
cd AulaPOO
```

```
# Compilar e executar o projeto  
dotnet run
```

Estrutura Básica de um Projeto C#

O ponto de partida mais simples para praticar POO é um **projeto de Console**. A estrutura mínima consiste em um arquivo Program.cs com código principal, que pode usar **Top-Level Statements** (C# 9+) ou o tradicional static void Main.

Namespaces

Agrupam tipos relacionados, evitando colisões de nomes e organizando logicamente o projeto.

```
namespace MeuProjeto.Dominio  
{  
    public class Produto { /* ... */ }  
}
```

Tipos Fundamentais

class (tipo referência): objetos vivem no heap, gerenciados pelo Garbage Collector

struct (tipo valor): normalmente residem na stack quando locais

Para modelar entidades de domínio,
comece sempre com class

 **Nota sobre memória:** Em C# objetos de **classe** vivem no *heap* (gerenciados pelo GC), enquanto tipos **valor** (como int, double ou struct) costumam residir na *stack* quando locais. O compilador e o runtime otimizam isso; foque na **semântica** (referência vs valor) ao modelar.

Tipos, Classes e Objetos: o núcleo do paradigma

A Programação Orientada a Objetos se baseia em três conceitos fundamentais que formam seu núcleo: tipos, classes e objetos. Entender a relação entre eles é essencial para dominar o paradigma.



Classe

É a "fórmula" ou "planta" que define propriedades (estado) e métodos (comportamento). Uma classe é uma definição abstrata de um tipo de objeto.



Objeto (instância)

É a coisa concreta criada a partir da classe usando `new MinhaClasse()`. Cada objeto tem seu próprio estado, independente de outros objetos da mesma classe.



Identidade

Dois objetos podem ter exatamente o mesmo estado e ainda assim serem instâncias diferentes. Cada objeto tem sua própria identidade única em memória.

Exemplo Mínimo de uma Classe

```
public class Produto
{
    public string Nome { get; private set; }
    public double PrecoUnitario { get; private set; }
    public int QuantidadeEmEstoque { get; private set; }

    public Produto(string nome, double precoUnitario, int quantidade)
    {
        if (string.IsNullOrWhiteSpace(nome))
            throw new ArgumentException("Nome inválido");
        if (precoUnitario < 0)
            throw new ArgumentOutOfRangeException(nameof(precoUnitario));
        if (quantidade < 0)
            throw new ArgumentOutOfRangeException(nameof(quantidade));

        Nome = nome;
        PrecoUnitario = precoUnitario;
        QuantidadeEmEstoque = quantidade;
    }

    public double ValorTotal() => PrecoUnitario * QuantidadeEmEstoque;
}
```

Valor vs Referência

Uma diferença crucial entre C e C# está na semântica de tipos:

Tipos de Referência (class)

Variáveis guardam referências para objetos no heap. Ao atribuir uma variável a outra, ambas apontam para o mesmo objeto.

```
var p1 = new Produto("Café", 18.90, 10);
var p2 = p1; // p2 e p1 apontam para o mesmo
            // objeto
p2.AdicionarAoEstoque(5); // Modifica o objeto
                          // que p1 também referencia
```

Tipos de Valor (struct)

Variáveis guardam o valor diretamente. Ao atribuir uma variável a outra, o valor é copiado.

```
int a = 10;
int b = a; // b recebe uma cópia do valor de a
b = 20; // Altera apenas b, a continua sendo 10
```

Ideia-chave: Regras que antes ficavam soltas (funções) agora moram **dentro** de métodos da classe, junto com os dados que elas manipulam.

Campos, Propriedades, Métodos e Construtores

As classes em C# são compostas por quatro elementos principais que trabalham juntos para definir o estado e o comportamento dos objetos: campos, propriedades, métodos e construtores.

Campo (Field)

É o armazenamento "cru" dos dados dentro da classe. Geralmente declarado como **private** para proteger o estado interno.

```
private double _precoUnitario;
```

Propriedade

É a interface de acesso ao estado, com acessadores **get** e **set**. Podem ser automáticas ou implementadas com lógica customizada.

```
public double PrecoUnitario { get; private
                             set; }
```

CLASS

Método

Implementa uma ação ou regra de negócio, definindo o comportamento da classe. Geralmente usa verbos no nome.

```
public double ValorTotal() =>  
    PrecoUnitario * QuantidadeEmEstoque;
```

Construtor

Método especial executado na criação do objeto, garantindo que ele nasça em um estado válido.

```
public Produto(string nome, double preco)  
{ /* validações e inicialização */ }
```

Auto-Properties vs Propriedades Completas

Propriedade Automática

Ideal para casos simples, sem validação adicional. O compilador gera um campo privado automaticamente.

```
// Simples (automática)  
public string Nome { get; private set; }
```

Propriedade Completa

Permite validação e lógica personalizada nos acessadores get/set.

```
// Completa (com campo e validação)  
private double _precoUnitario;  
public double PrecoUnitario  
{  
    get => _precoUnitario;  
    private set  
    {  
        if (value < 0)  
            throw new ArgumentOutOfRangeException(  
                nameof(PrecoUnitario));  
        _precoUnitario = value;  
    }  
}
```



“Overloading”

Sobrecarga de Construtores

É comum fornecer múltiplos construtores para facilitar a criação de objetos em diferentes contextos:

```
// Construtor principal com todas as validações
public Produto(string nome, double precoUnitario, int quantidade)
{
    // Validações e inicialização
}

// Construtor secundário que chama o principal
public Produto(string nome, double precoUnitario)
    : this(nome, precoUnitario, quantidade: 0)
{ }
```

Dica: Mantenha seus métodos **curtos**, com nomes verbais que expressem claramente a ação: `AdicionarAoEstoque`, `RemoverDoEstoque`, `AplicarDesconto`.

Encapsulamento, Invariante e Exceções

O **encapsulamento** é um dos pilares da POO, permitindo que as classes controlem como seu estado interno é acessado e modificado. Esse conceito trabalha em conjunto com **invariante**s (condições que sempre devem ser verdadeiras) e **exceções** (para sinalizar violações das regras).





SOFTWARE CLASS

Encapsulamento em Ação

Note como a classe Produto controla o acesso aos seus dados internos:

```
public class Produto
{
    // Propriedade encapsulada - só pode ser modificada
    // dentro da classe
    public int QuantidadeEmEstoque { get; private set; }

    // Método público que aplica regras de negócio
    // antes de modificar o estado
    public void RemoverDoEstoque(int quantidade)
    {
        if (quantidade <= 0 || quantidade > QuantidadeEmEstoque)
            throw new ArgumentOutOfRangeException(
                nameof(quantidade),
                "Quantidade inválida para remoção");

        QuantidadeEmEstoque -= quantidade;
    }
}
```

Invariante de Classe

Invariante é uma condição que deve ser sempre verdadeira para que um objeto seja considerado válido. Exemplos comuns incluem:

Quantidade em estoque nunca pode ser negativa

Garantido pela validação no construtor e no método
RemoverDoEstoque

Preço unitário deve ser maior ou igual a zero

Garantido pela validação no construtor e potencialmente na propriedade PrecoUnitario

Nome do produto não pode ser vazio

Garantido pela validação no construtor

Usando Exceções Adequadamente

Exceções devem ser lançadas quando as regras de negócio ou invariantes são violadas:

ArgumentException

Para argumentos inválidos passados a métodos ou construtores (ex: nome vazio)

ArgumentOutOfRangeException

Para argumentos numericamente inválidos (ex: quantidade negativa)

ArgumentNullException

Especificamente para argumentos null quando não são permitidos

InvalidOperationException

Quando a operação não pode ser realizada no estado atual do objeto

Regra de ouro: Valide **na fronteira** onde o estado pode mudar — no **construtor** e nos **métodos** que alteram invariantes. Nunca permita que um objeto entre em um estado inválido.

Guia de Refatoração: do estilo estruturado para OO

Transformar código estruturado em orientado a objetos é uma habilidade essencial para quem está fazendo a transição entre paradigmas. Este guia passo a passo ajudará você a refatorar seu código de forma sistemática.

01

Liste os dados do problema

Identifique todas as variáveis e estruturas de dados relacionadas. Estas se tornarão **propriedades** da sua classe.

```
// Antes (código estruturado)
string nomeProduto = "Café";
double precoProduto = 18.90;
int quantidadeEmEstoque = 10;

// Depois (orientado a objetos)
public class Produto
{
    public string Nome { get; private set; }
    public double PrecoUnitario { get; private set; }
    public int QuantidadeEmEstoque { get; private set; }
    // ...
}
```

2

Liste as regras e cálculos

Funções e procedimentos que manipulam os dados se tornarão **métodos** da classe.

```
// Antes (código estruturado)
double CalcularValorTotal(double preco, int quantidade)
{
    return preco * quantidade;
}

// Depois (orientado a objetos)
public class Produto
{
    // ...
    public double ValorTotal()
    {
        return PrecoUnitario * QuantidadeEmEstoque;
    }
}
```

3

Defina o construtor com validações

Estabeleça as invariantes que garantem que o objeto sempre estará em um estado válido.

```
public Produto(string nome, double precoUnitario, int quantidade)
{
    if (string.IsNullOrWhiteSpace(nome))
        throw new ArgumentException("Nome inválido");
    if (precoUnitario < 0)
        throw new ArgumentOutOfRangeException(nameof(precoUnitario));
    if (quantidade < 0)
        throw new ArgumentOutOfRangeException(nameof(quantidade));

    Nome = nome;
    PrecoUnitario = precoUnitario;
    QuantidadeEmEstoque = quantidade;
}
```

Encapsule o acesso aos dados

Use `private set` nas propriedades para controlar como o estado interno pode ser modificado.

```
public class Produto
{
    public string Nome { get; private set; }
    public double PrecoUnitario { get; private set; }
    public int QuantidadeEmEstoque { get; private set; }
    // ...
}
```

Crie `ToString()` para depuração

Implemente uma representação textual do objeto para facilitar depuração e logs.

```
public override string ToString()
=> $"{Nome} | {QuantidadeEmEstoque} un. | R$ {PrecoUnitario:F2} | Total: R$ {ValorTotal():F2}";
```

 **Dica importante:** Avance em **passos pequenos**. Compile e rode após cada mudança significativa para identificar problemas rapidamente. Não tente refatorar tudo de uma vez.