

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; }
```



AUTO-PROPS

CLASS

FULL PROPERTIES

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.

Estudo de Caso 1: Produto e Operações de Estoque

Vamos examinar nosso primeiro estudo de caso completo: uma classe `Produto` com operações de estoque, que demonstra encapsulamento, validações e métodos bem definidos.

```
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;

public void AdicionarAoEstoque(int quantidade)
{
    if (quantidade <= 0)
        throw new ArgumentOutOfRangeException(nameof(quantidade));
    QuantidadeEmEstoque += quantidade;
}

public void RemoverDoEstoque(int quantidade)
{
    if (quantidade <= 0 || quantidade > QuantidadeEmEstoque)
        throw new ArgumentOutOfRangeException(nameof(quantidade));
    QuantidadeEmEstoque -= quantidade;
}

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

```

Como Utilizar a Classe Produto

```

public static class DemoProduto
{
    public static void Executar()
    {
        var p1 = new Produto("Café", 18.90, 10);
        var p2 = new Produto("Filtro", 7.50, 40);

        Console.WriteLine(p1);
        Console.WriteLine(p2);

        p1.AdicionarAoEstoque(5);
        p2.RemoverDoEstoque(10);
    }
}

```

```
Console.WriteLine("\nDepois das operações:");
Console.WriteLine(p1);
Console.WriteLine(p2);
}
}
```

Pontos Fortes do Design

- Encapsulamento dos dados com `private set`
- Validações claras no construtor e métodos
- Métodos específicos para cada operação de negócio
- `ToString()` útil para depuração e apresentação
- Invariantes mantidos em todas as operações

Possíveis Melhorias

- Adicionar método para alterar o preço (com validação)
- Implementar `IEquatable<Produto>` para comparações
- Adicionar categorização do produto
- Implementar cálculo de lucro baseado no custo

Estudo de Caso 2: Aluno e Cálculo de Média

Neste estudo de caso, veremos como uma classe `Aluno` pode encapsular o cálculo de média e a determinação de aprovação, usando um array de notas.

```
public class Aluno
{
    public string Nome { get; }
    public double[] Notas { get; }

    public Aluno(string nome, double n1, double n2)
    {
        if (string.IsNullOrWhiteSpace(nome))
            throw new ArgumentException("Nome inválido");

        Notas = new[] { n1, n2 };
        Nome = nome;
    }

    public double Media() => (Notas[0] + Notas[1]) / 2.0;

    public bool Aprovado => Media() >= 7.0;
}
```



```
public override string ToString()
=> $"{Nome} | Notas: {Notas[0]:F1}, {Notas[1]:F1} | " +
    $"Média: {Media():F1} | Aprovado: {Aprovado}";
}
```

Como Utilizar a Classe Aluno

```
public static class DemoAluno
{
    public static void Executar()
    {
        var s1 = new Aluno("Lina", 8.2, 6.9);
        var s2 = new Aluno("Klaus", 9.1, 8.7);

        Console.WriteLine(s1);
        Console.WriteLine(s2);
    }
}
```

Análise do Exemplo

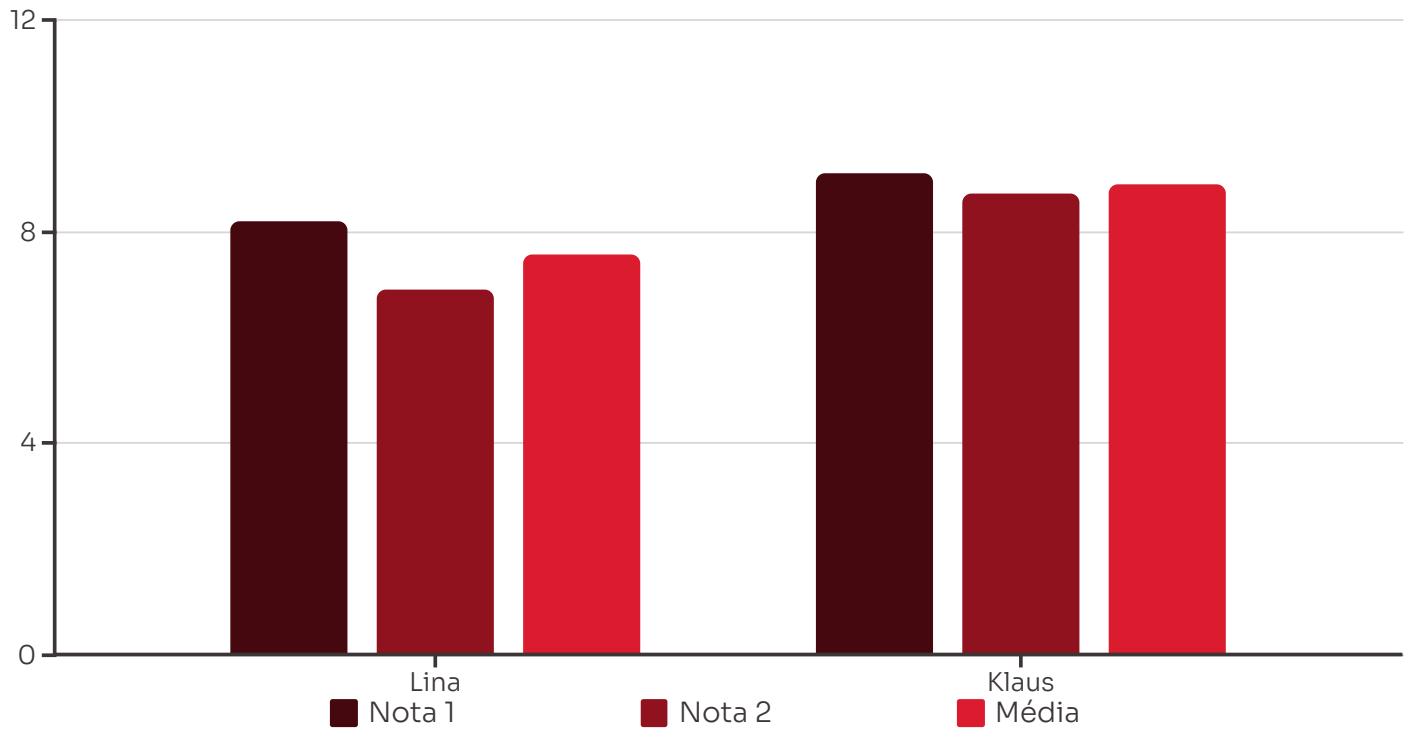
Pontos Interessantes

- O array `Notas` é somente leitura (não possui setter), garantindo que os valores não sejam alterados após a criação
- O método `Media()` encapsula a lógica de cálculo
- A propriedade `Aprovado` implementa uma regra de negócio como uma expressão booleana

Considerações

Este exemplo usa um array de tamanho fixo (`double[]`) para armazenar as notas. Em situações mais complexas, poderíamos usar coleções dinâmicas como `List<double>`, mas isso será abordado em módulos futuros.

Note como a propriedade `Aprovado` usa uma **expressão lambda** para calcular dinamicamente o resultado baseado na média, sem armazenar esse valor.



Estudo de Caso 3: Catálogo Simples com Arrays

Neste exemplo, veremos como implementar um catálogo de produtos usando arrays com capacidade fixa, sem recorrer a coleções dinâmicas como `List<T>`. Este modelo é útil para entender os fundamentos de gerenciamento de coleções.

```
public class CatalogoProdutos
{
    private readonly Produto[] _itens;
    private int _tamanho; // quantos itens já foram inseridos

    public CatalogoProdutos(int capacidade)
    {
        if (capacidade <= 0)
            throw new ArgumentOutOfRangeException(nameof(capacidade));

        _itens = new Produto[capacidade];
        _tamanho = 0;
    }

    public void Adicionar(Produto item)
    {
        if (_tamanho == _itens.Length)
            throw new InvalidOperationException("O catálogo está cheio");

        _itens[_tamanho] = item;
        _tamanho++;
    }

    public Produto Remover(int index)
    {
        if (index < 0 || index >= _tamanho)
            throw new ArgumentOutOfRangeException(nameof(index));

        var resultado = _itens[index];
        for (int i = index; i < _tamanho - 1; i++)
            _itens[i] = _itens[i + 1];
        _tamanho--;
        return resultado;
    }

    public void Atualizar(int index, Produto novoItem)
    {
        if (index < 0 || index >= _tamanho)
            throw new ArgumentOutOfRangeException(nameof(index));

        _itens[index] = novoItem;
    }

    public Produto Get(int index)
    {
        if (index < 0 || index >= _tamanho)
            throw new ArgumentOutOfRangeException(nameof(index));
        return _itens[index];
    }

    public void Desfazer()
    {
        _tamanho = 0;
    }

    public void Refazer()
    {
        throw new NotImplementedException();
    }
}
```

```

public void Adicionar(Produto p)
{
    if (p is null)
        throw new ArgumentNullException(nameof(p));
    if (_tamanho >= _itens.Length)
        throw new InvalidOperationException("Capacidade esgotada");

    _itens[_tamanho++] = p;
}

public Produto? BuscarPorNome(string nome)
{
    for (int i = 0; i < _tamanho; i++)
    {
        if (string.Equals(_itens[i].Nome, nome,
            StringComparison.OrdinalIgnoreCase))
            return _itens[i];
    }
    return null; // não encontrado
}

public double ValorTotalDoEstoque()
{
    double total = 0;
    for (int i = 0; i < _tamanho; i++)
    {
        total += _itens[i].ValorTotal();
    }
    return total;
}

```

Como Utilizar o Catálogo

```

public static class DemoCatalogo
{
    public static void Executar()
    {
        var cat = new CatalogoProdutos(capacidade: 5);

        cat.Adicionar(new Produto("Café", 18.90, 10));
        cat.Adicionar(new Produto("Filtro", 7.50, 40));

        Console.WriteLine($"Total: R$ {cat.ValorTotalDoEstoque():F2}");
    }
}

```

```
        var produtoBuscado = cat.BuscarPorNome("café");
        Console.WriteLine(produtoBuscado?.ToString() ?? "Produto não encontrado");
    }
}
```

Criação do Catálogo

Inicializamos o catálogo com uma capacidade fixa, criando um array interno para armazenar os produtos.

Busca de Produtos

O método `BuscarPorNome` percorre os produtos inseridos (até `_tamanho`) e retorna o primeiro com nome correspondente.

Adição de Produtos

O método `Adicionar` verifica se há espaço disponível e armazena o produto na próxima posição livre, incrementando o contador `_tamanho`.

Cálculo de Valor Total

O método `ValorTotalDoEstoque` soma o valor total de cada produto no catálogo, usando o método `ValorTotal()` da classe `Produto`.

 **Observação:** Este exemplo mostra como implementar um contêiner básico sem usar coleções dinâmicas. Em aplicações reais, você normalmente usaria `List<Produto>` para maior flexibilidade e funcionalidade.

Estudo de Caso 4: Pedido com Array de Itens

Neste estudo de caso, implementaremos um sistema de pedido que contém múltiplos itens, demonstrando como objetos podem conter e gerenciar outros objetos.

Classe ItemPedido

```
public class ItemPedido
{
    public string NomeDoProduto { get; }
    public double PrecoUnitario { get; }
    public int Quantidade { get; private set; }
```

```

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

    NomeDoProduto = nome;
    PrecoUnitario = precoUnitario;
    Quantidade = quantidade;
}

public double Subtotal() => PrecoUnitario * Quantidade;
}

```

Classe Pedido

```

public class Pedido
{
    private readonly ItemPedido[] _itens;
    private int _tamanho;

    public int Numero { get; }
    public DateTime CriadoEm { get; } = DateTime.Now;

    public Pedido(int numero, int capacidadeDeltens)
    {
        if (capacidadeDeltens <= 0)
            throw new ArgumentOutOfRangeException(nameof(capacidadeDeltens));

        Numero = numero;
        _itens = new ItemPedido[capacidadeDeltens];
        _tamanho = 0;
    }

    public void AdicionarItem(ItemPedido item)
    {
        if (item is null)
            throw new ArgumentNullException(nameof(item));
        if (_tamanho >= _itens.Length)
            throw new InvalidOperationException("Capacidade de itens
esgotada");

        _itens[_tamanho++] = item;
    }
}

```

System Flow



```
public double Total()
{
    double total = 0;
    for (int i = 0; i < _tamanho; i++)
        total += _itens[i].Subtotal();
    return total;
}
```

Como Utilizar o Sistema de Pedidos

```
public static class DemoPedido
{
    public static void Executar()
    {
        var ped = new Pedido(numero: 123, capacidadeDeltens: 3);

        ped.AdicionarItem(new ItemPedido("Café", 18.90, 2));
        ped.AdicionarItem(new ItemPedido("Filtro", 7.50, 1));

        Console.WriteLine($"Total do pedido: R$ {ped.Total():F2}");
    }
}
```

Análise da Relação entre Objetos



Pedido

Gerencia uma coleção de itens e coordena operações como cálculo de total

ItemPedido

Encapsula informações específicas de cada produto no pedido

Cálculos

Cada objeto calcula seus próprios valores, que são agregados pelo contêiner

Este exemplo demonstra um padrão comum em POO: objetos contendo e gerenciando outros objetos. A classe `Pedido` é responsável pela coleção de itens, enquanto cada `ItemPedido` mantém seu próprio estado e calcula seu subtotal. O pedido então agrupa esses valores para calcular o total geral.

Qualidade de Código: Nomenclatura e Organização

Uma parte fundamental da programação orientada a objetos é adotar boas práticas de nomenclatura e organização do código. Código bem estruturado é mais fácil de entender, manter e evoluir.



f(x)



Nomenclatura de Classes

Use substantivos no singular que representem claramente a entidade modelada. Exemplos: `Produto`, `Cliente`, `Pedido`, `ItemPedido`.

Nomenclatura de Métodos

Use verbos ou frases verbais que descrevam a ação. Exemplos: `CalcularTotal()`, `AdicionarAoEstoque()`, `ProcessarPagamento()`.

Organização de Arquivos

Um arquivo por classe; namespace coerente; organize pastas por área funcional (ex.: `Dominio`, `Servicos`, `App`).

Padrões de Nomenclatura em C#



PascalCase

Use para tipos (classes, interfaces, structs, enums) e membros públicos (propriedades, métodos).

```
public class Produto {}  
public double ValorTotalDoEstoque() {}
```



camelCase

Use para variáveis locais, parâmetros e campos privados (com ou sem prefixo `_`).

```
public void AdicionarItem(ItemPedido item)  
{  
    double valorItem = item.Subtotal();  
    _itens[_tamanho++] = item;  
}
```



UPPER_CASE

Reservado para constantes, quando aplicável.

```
private const int TAMANHO_MAXIMO = 100;
```



Implementação de `ToString()`

A sobrescrita do método `ToString()` é essencial para facilitar a depuração e apresentação de objetos:

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

Cuidado: Evite vazamento de detalhes de implementação no `ToString()`. Concentre-se em propriedades significativas para o usuário ou desenvolvedor.

Estrutura Sugerida de Pastas

```
AulaPOO/  
├── Program.cs  
├── Dominio/  
│   ├── Produto.cs  
│   ├── Aluno.cs  
│   ├── ItemPedido.cs  
│   └── Pedido.cs  
└── Servicos/  
    └── CatalogoProdutos.cs
```

Uma organização clara de pastas ajuda a manter o código coeso e facilita a navegação pelo projeto, especialmente à medida que ele cresce.

Implementando Igualdade em Classes de Domínio

Em muitos casos, precisamos comparar objetos de nossas classes de domínio de maneira que reflita sua identidade de negócio, não apenas sua referência em memória. Para isso, podemos implementar a interface `IEquatable<T>` e sobrescrever os métodos `Equals` e `GetHashCode`.

```

public class Produto : IEquatable<Produto>
{
    // ... outros membros da classe ...

    // Implementação de IEquatable
    public bool Equals(Produto? outro)
        => outro is not null &&
            string.Equals(Nome, outro.Nome,
                StringComparison.OrdinalIgnoreCase);

    // Sobrescrita de Object.Equals
    public override bool Equals(object? obj)
        => Equals(obj as Produto);

    // Sobrescrita de GetHashCode
    public override int GetHashCode()
        => StringComparer.OrdinalIgnoreCase.GetHashCode(Nome);
}

```

Quando e Como Implementar Igualdade

1

Identifique o critério de igualdade

Determine o que torna dois objetos "iguais" no contexto do seu domínio. Pode ser um único campo (como um código ou nome) ou uma combinação de campos.

2

Implemente IEquatable<T>

Esta interface define o método `Equals(T? other)` que permite comparações fortemente tipadas e mais eficientes.

3

Sobrescreva Object.Equals

Delegue para o método específico de tipo para manter a consistência quando o objeto for comparado com `object`.

4

Sobrescreva GetHashCode

Garanta que objetos considerados iguais produzam o mesmo código hash, essencial para uso em coleções baseadas em hash como `Dictionary` e `HashSet`.

- ✖ **Cuidado:** Escolha critérios de igualdade **com parcimônia**. Se o nome de um produto pode mudar, talvez não seja o melhor identificador para igualdade. Considere campos que sejam realmente imutáveis ou que representem a identidade do objeto no domínio.



Benefícios da Implementação Correta

- Comparações mais intuitivas com operadores == e != (quando sobrescritos)
- Funcionamento correto em coleções como Dictionary, HashSet e List.Contains()
- Código mais claro e semântico, expressando o conceito de igualdade no domínio
- Testes mais expressivos e menos propensos a erros

Erros Comuns e Como Evitá-los

Ao transicionar do paradigma estruturado para o orientado a objetos, é comum cometer alguns erros. Identificá-los e entender como evitá-los acelerará seu aprendizado.



Tudo Public

Expor todos os campos e propriedades como públicas quebra o encapsulamento e permite que o estado interno seja modificado sem controle, potencialmente violando invariantes.

Solução: Use private set nas propriedades e crie métodos específicos para manipular o estado com validações.



Métodos Gigantes

Criar métodos muito longos que fazem várias coisas diferentes dificulta a manutenção e o reuso do código.

Solução: Dívida em métodos menores com responsabilidades bem definidas. Cada método deve fazer apenas uma coisa.



Falta de ToString()

Não implementar `ToString()` dificulta a depuração e a exibição de informações úteis sobre o objeto.

Solução: Sempre sobrescreva `ToString()` para fornecer uma representação textual útil do objeto.



Buscas sem Tratar "Não Encontrado"

Ao procurar por um item em uma coleção, não tratar o caso de "não encontrado" pode levar a `NullReferenceException`.

Solução: Sempre verifique se o resultado é `null` antes de usá-lo, ou use o operador condicional nulo (`?.`).



Misturar UI com Regras de Negócio

Colocar código de interação com o usuário (`Console.ReadLine/WriteLine`) dentro das classes de domínio dificulta o reuso e os testes.

Solução: Mantenha regras na classe de domínio e a interação com o usuário no programa principal.



Ignorar Invariantes

Não validar entradas ou garantir condições que sempre devem ser verdadeiras leva a estados inválidos e bugs difíceis de encontrar.

Solução: Identifique as invariantes e valide-as em construtores e métodos que modificam o estado.

Exemplos de Código Problemático vs. Correto

✗ Problemático

```
public class ContaBancaria
{
    // Tudo público, sem validações
    public string Titular;
    public double Saldo;

    // Método gigante que faz muitas coisas
    public void ProcessarOperacao(string tipo, double valor)
    {
        if (tipo == "deposito")
        {
            Saldo += valor;
            Console.WriteLine($"Depósito de R$ {valor}");
            Console.WriteLine($"Novo saldo: R$ {Saldo}");
        }
    }
}
```

```

else if (tipo == "saque")
{
    // Sem validação de saldo suficiente
    Saldo -= valor;
    Console.WriteLine($"Saque de R$ {valor}");
    Console.WriteLine($"Novo saldo: R$ {Saldo}");
}
}
}

```

Correto

```

public class ContaBancaria
{
    // Propriedades encapsuladas
    public string Titular { get; }
    public double Saldo { get; private set; }

    public ContaBancaria(string titular, double saldoInicial = 0)
    {
        if (string.IsNullOrWhiteSpace(titular))
            throw new ArgumentException("Titular inválido");
        if (saldoInicial < 0)
            throw new ArgumentException("Saldo inicial não pode ser negativo");

        Titular = titular;
        Saldo = saldoInicial;
    }

    // Métodos específicos com validações
    public void Depositar(double valor)
    {
        if (valor <= 0)
            throw new ArgumentException("Valor deve ser positivo");

        Saldo += valor;
    }

    public void Sacar(double valor)
    {
        if (valor <= 0)
            throw new ArgumentException("Valor deve ser positivo");
        if (valor > Saldo)
            throw new InvalidOperationException("Saldo insuficiente");

        Saldo -= valor;
    }
}

```

```
public override string ToString() =>
    $"Conta de {Titular} | Saldo: R$ {Saldo:F2}";
}
```

Nullability e Tratamento de Valores Nulos

O C# tem recursos avançados para lidar com valores nulos e evitar o temido `NullReferenceException`. Entender esses recursos é essencial para escrever código mais seguro e expressivo.

Nullable Reference Types

A partir do C# 8.0, podemos habilitar "nullable reference types", que ajudam o compilador a detectar possíveis referências nulas não tratadas:

```
// No arquivo .csproj
<PropertyGroup>
    <Nullable>enable</Nullable>
</PropertyGroup>
```

Com essa configuração ativada, o compilador ajuda a distinguir entre:

Tipos não-nulos (padrão)

```
string nome; // Não pode ser null
Produto produto; // Não pode ser null
```

O compilador emitirá avisos se você tentar atribuir `null` a esses tipos ou usá-los sem verificar se foram inicializados.

Tipos anuláveis (com ?)

```
string? nome; // Pode ser null
Produto? produto; // Pode ser null
```

Indica explicitamente que a variável pode conter `null`, e o compilador lembrará você de verificar isso antes de usá-la.

Operadores para Tratamento de Null

Operador ?. (Null-conditional)

```
// Em vez de:  
if (produto != null)  
{  
    Console.WriteLine(produto.Nome);  
}  
  
// Use:  
Console.WriteLine(produto?.Nome);
```

Acessa a propriedade apenas se o objeto não for null; caso contrário, retorna null.

Operador ?? (Null-coalescing)

```
// Em vez de:  
string nome;  
if (produto != null)  
    nome = produto.Nome;  
else  
    nome = "Desconhecido";  
  
// Use:  
string nome = produto?.Nome ?? "Desconhecido";
```

Fornece um valor padrão para usar quando a expressão é null.

Operador ??= (Null-coalescing assignment)

```
// Em vez de:  
if (lista == null)  
    lista = new List<string>();  
  
// Use:  
lista ??= new List<string>();
```

Atribui um valor à variável apenas se ela for null.

Exemplo Prático de Busca com Tratamento de Null

```
public class CatalogoProdutos  
{  
    // ... outros membros ...
```



```
public Produto? BuscarPorNome(string nome)
{
    for (int i = 0; i < _tamanho; i++)
    {
        if (string.Equals(_itens[i].Nome, nome,
            StringComparison.OrdinalIgnoreCase))
            return _itens[i];
    }
    return null; // não encontrado
}

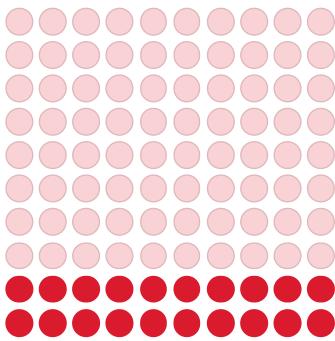
// Uso seguro:
var produto = catalogo.BuscarPorNome("café");
Console.WriteLine(produto?.ToString() ?? "Produto não encontrado");

// OU com pattern matching:
if (catalogo.BuscarPorNome("café") is Produto p)
{
    Console.WriteLine($"Encontrado: {p}");
}
else
{
    Console.WriteLine("Produto não encontrado");
}
```

Dica: Habitue-se a usar o modificador `?` sempre que uma referência puder ser `null`. Isso torna seu código mais expressivo e ajuda a evitar bugs comuns.

Checklist de Aprendizado

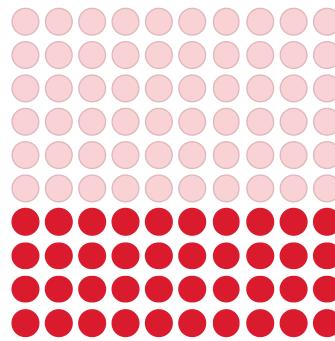
Use esta lista para verificar seu progresso na compreensão dos conceitos fundamentais de Programação Orientada a Objetos em C#.



20%

Conceitos Fundamentais

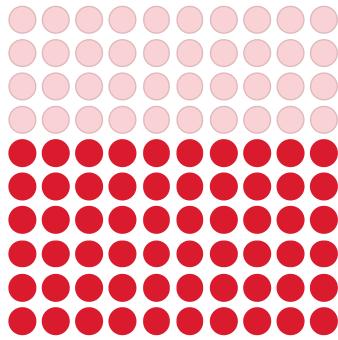
- Entendo o que são classes e objetos
- Sei explicar a diferença entre valor e referência
- Compreendo o conceito de estado e comportamento



40%

Elementos de Classe

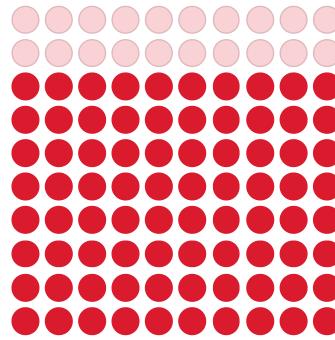
- Sei criar propriedades com encapsulamento adequado
- Entendo a função dos construtores
- Consigo implementar métodos com responsabilidade única
- Sei implementar o método `ToString()` de forma útil



60%

Boas Práticas

- Aplico o encapsulamento para proteger invariantes
- Utilizo exceções apropriadas para validações
- Sigo convenções de nomenclatura do C#
- Sei organizar o código em arquivos e namespaces



80%

Manipulação de Objetos

- Consigo usar arrays para armazenar múltiplos objetos
- Implemento busca e manipulação em coleções de objetos
- Entendo como objetos podem conter outros objetos
- Sei tratar adequadamente valores nulos

Próximos Passos

Após dominar os conceitos desta primeira etapa, você estará preparado para avançar para tópicos mais avançados em POO:

Herança e Polimorfismo

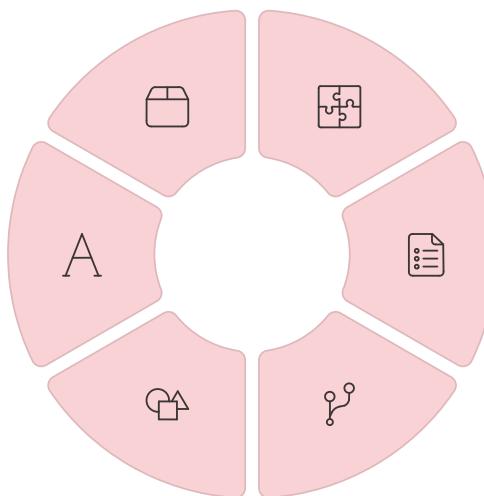
Mecanismos para reutilização e extensão de código através de hierarquias de classes

Nomenclatura em Inglês

Transição para padrões internacionais de codificação

Associação e Composição

Padrões de relacionamento entre classes



Interfaces

Contratos que definem comportamentos que classes podem implementar

Coleções Genéricas

List<T>, Dictionary<K,V>, e outras estruturas de dados dinâmicas

LINQ

Language Integrated Query para consultas e manipulação de dados

Lembre-se: a fluência em POO vem com a prática. Aplique os conceitos aprendidos em projetos reais e exercícios, sempre com foco na clareza e na manutenibilidade do código.

Exercícios Práticos

Para consolidar seu aprendizado, recomendamos realizar os seguintes exercícios práticos. Comece do mais simples e avance gradualmente.

01

Refatore um Cálculo Simples

Transforme um algoritmo estruturado (como cálculo de salário ou média ponderada) em uma classe com propriedades, métodos e invariantes bem definidos.

Exemplo: Criar uma classe CalculadoraSalario que calcule salário líquido a partir de horas trabalhadas, valor/hora e descontos.

02

Implemente um Sistema de Pedidos

Crie uma classe Pedido que contenha um array de ItemPedido. Valide quantidades, calcule subtotais e totais.

Desafio adicional: Adicione desconto por quantidade e limite de itens por pedido.

03

Desenvolva um Mini-Catálogo

Crie um Catalogo com capacidade fixa e operações básicas: adicionar, remover, listar, buscar por nome ou código.

Desafio adicional: Implemente ordenação e filtragem básica dos itens.

04

Sistema de Notas Escolares

Crie classes para Aluno, Disciplina e Avaliacao, permitindo registrar e calcular médias por disciplina.

Desafio adicional: Implementar critérios diferentes de aprovação por disciplina.

Dicas para Desenvolvimento dos Exercícios

Planeje Antes de Codificar

Identifique os dados (propriedades) e comportamentos (métodos) necessários. Faça um esboço da classe antes de começar a implementação.

Avance em Pequenos Passos

Implemente uma funcionalidade por vez, compile e teste antes de avançar. Isso facilita a identificação de problemas.

Foque nas Invariante

Identifique condições que sempre devem ser verdadeiras e garanta que o código as mantenha, com validações nos construtores e métodos.

Teste com Casos Limite

Experimente valores extremos ou inválidos para verificar se suas validações estão funcionando corretamente.

Exemplo de Solução para o Exercício 1

```
public class CalculadoraSalario
{
    public string NomeFuncionario { get; }
    public double ValorHora { get; private set; }
    public int HorasTrabalhadas { get; private set; }
    public double PercentualDesconto { get; private set; }

    public CalculadoraSalario(string nome, double valorHora, int horasTrabalhadas,
        double percentualDesconto)
    {
        if (string.IsNullOrWhiteSpace(nome))
            throw new ArgumentException("Nome inválido");
        if (valorHora <= 0)
            throw new ArgumentOutOfRangeException(nameof(valorHora), "Valor hora deve
                ser positivo");
        if (horasTrabalhadas < 0)
            throw new ArgumentOutOfRangeException(nameof(horasTrabalhadas), "Horas não
                pode ser negativo");
        if (percentualDesconto < 0 || percentualDesconto > 100)
            throw new ArgumentOutOfRangeException(nameof(percentualDesconto),
                "Percentual deve estar entre 0 e 100");

        NomeFuncionario = nome;
        ValorHora = valorHora;
        HorasTrabalhadas = horasTrabalhadas;
        PercentualDesconto = percentualDesconto;
    }
```

```

public double SalarioBruto() => ValorHora * HorasTrabalhadas;

public double ValorDesconto() => SalarioBruto() * (PercentualDesconto /
100.0);

public double SalarioLiquido() => SalarioBruto() - ValorDesconto();

public void AtualizarHorasTrabalhadas(int novasHoras)
{
    if (novasHoras < 0)
        throw new ArgumentOutOfRangeException(nameof(novasHoras), "Horas não pode
ser negativo");
    HorasTrabalhadas = novasHoras;
}

public override string ToString()
=> $"{NomeFuncionario} | {HorasTrabalhadas}h x R$ {ValorHora:F2} | " +
    $"Bruto: R$ {SalarioBruto():F2} | Desconto: {PercentualDesconto}% | " +
    $"Líquido: R$ {SalarioLiquido():F2}";
}

```

Program.cs para Testar Tudo

Após implementar suas classes de domínio, é útil ter um ponto de entrada centralizado para testar todas as funcionalidades. O arquivo `Program.cs` serve a esse propósito, permitindo executar diferentes cenários de teste.

```

using System;

namespace AulaPOO
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("== DEMONSTRAÇÃO DE PRODUTO ==");
            DemoProduto.Executar();

            Console.WriteLine("\n== DEMONSTRAÇÃO DE ALUNO ==");
            DemoAluno.Executar();

            Console.WriteLine("\n== DEMONSTRAÇÃO DE CATÁLOGO ==");
            DemoCatalogo.Executar();

            Console.WriteLine("\n== DEMONSTRAÇÃO DE PEDIDO ==");
            DemoPedido.Executar();
        }
    }
}

```

```
// Descomente para executar as demonstrações dos exercícios
// Console.WriteLine("\n==== DEMONSTRAÇÃO DE CALCULADORA DE SALÁRIO ===");
// DemoCalculadoraSalario.Executar();
}
}
}
```

Exemplo de Classe de Demonstração

Cada classe de domínio pode ter uma classe de demonstração correspondente, como mostrado abaixo:

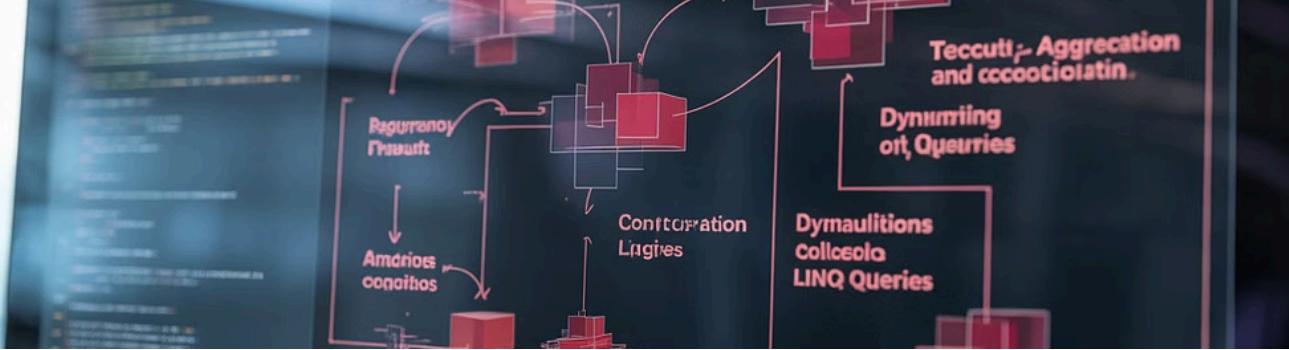
```
public static class DemoProduto
{
    public static void Executar()
    {
        try
        {
            Console.WriteLine("Criando produtos...");
            var p1 = new Produto("Café", 18.90, 10);
            var p2 = new Produto("Filtro", 7.50, 40);

            Console.WriteLine("Produtos criados:");
            Console.WriteLine(p1);
            Console.WriteLine(p2);

            Console.WriteLine("\nRealizando operações de estoque...");
            p1.AdicionarAoEstoque(5);
            p2.RemoverDoEstoque(10);

            Console.WriteLine("\nEstado final dos produtos:");
            Console.WriteLine(p1);
            Console.WriteLine(p2);

            // Demonstrando tratamento de erro
            Console.WriteLine("\nTentando remover quantidade inválida...");
            p1.RemoverDoEstoque(100); // Vai lançar exceção
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Erro: {ex.Message}");
        }
    }
}
```



Vantagens dessa Abordagem

- Separa claramente o código de teste do código de domínio
- Permite executar demonstrações específicas conforme necessário
- Facilita a adição de novos cenários de teste
- Centraliza a interação com o console em um único lugar
- Demonstra o tratamento de exceções em cenários reais

Alternativas para Testes

Em projetos reais, você usaria frameworks de teste como xUnit, NUnit ou MSTest para validar suas classes de forma automatizada, sem depender de saídas no console.

A abordagem mostrada aqui é útil para fins didáticos e para verificar rapidamente o comportamento das classes durante o desenvolvimento.

□ **Observação:** Este padrão de classes de demonstração estáticas é adequado para fins educacionais. Em aplicações reais, você provavelmente usaria um design mais sofisticado com injeção de dependência e separação clara entre camadas.

Próximos Passos na Jornada POO

Agora que você domina os fundamentos da Programação Orientada a Objetos em C#, vamos dar uma olhada no que vem a seguir em sua jornada de aprendizado.

Tópicos Avançados

Associação, Agregação e Composição

Diferentes formas de relacionamento entre classes, fundamentais para modelar sistemas complexos. Você aprenderá a representar relações como "tem um" e "usa um".

Coleções Dinâmicas com `List<T>`

Evolução dos arrays fixos para coleções que crescem conforme necessário, com operações poderosas de manipulação e busca.



LINQ (Language Integrated Query)

Consultas poderosas em coleções, com sintaxe expressiva para filtrar, ordenar, agrupar e transformar dados.



Herança e Polimorfismo

Reutilização de código através de hierarquias de classes e substituição de comportamentos.
Aprofundaremos em classes abstratas e virtuais.

Interfaces

Definição de contratos que classes podem implementar, promovendo baixo acoplamento e facilitando testes unitários.

Nomenclatura em Inglês

Transição para padrões internacionais de codificação, preparando você para projetos profissionais e colaboração global.

Aplicação Prática

À medida que avança nos conceitos, você trabalhará em projetos cada vez mais complexos:

1

Mini Sistema de Vendas

Implementando relacionamentos entre Cliente, Produto, Pedido e ItemPedido com coleções dinâmicas.

2

Biblioteca Digital

Gerenciamento de livros, autores e empréstimos usando herança e interfaces.

3

Aplicação Completa

Sistema com múltiplas camadas, integrando todos os conceitos aprendidos e seguindo padrões de design.

Lembre-se: A transição completa para o paradigma orientado a objetos é uma jornada. Não se preocupe em dominar tudo de uma vez. Continue praticando, refatore código antigo e esteja aberto a novas formas de resolver problemas.

Recursos Adicionais

Para aprofundar seus conhecimentos, recomendamos:

- Documentação oficial da Microsoft sobre C# e .NET
- Livros como "C# in Depth" de Jon Skeet e "Clean Code" de Robert C. Martin
- Plataformas de exercícios como Exercism.io e LeetCode
- Cursos online sobre design patterns e SOLID principles
- Participação em comunidades como Stack Overflow e fóruns em português

Conclusão

Ao longo deste material, exploramos os fundamentos da Programação Orientada a Objetos em C#, com foco especial na transição do paradigma estruturado (típico de C) para o orientado a objetos.

Você aprendeu conceitos essenciais como:

- Classes como moldes para objetos, combinando dados e comportamentos
- Encapsulamento para proteger o estado interno e garantir invariantes
- Propriedades, métodos e construtores como elementos de uma classe
- Uso adequado de exceções para validação e tratamento de erros
- Implementação de igualdade customizada para objetos de domínio
- Manipulação de coleções simples com arrays
- Boas práticas de nomenclatura e organização de código

Este conhecimento forma a base sólida que você precisa para avançar para tópicos mais complexos como herança, interfaces, coleções dinâmicas e LINQ, que serão abordados nas próximas etapas do seu aprendizado.

Pontos-Chave para Lembrar

	Encapsule Sempre Proteja o estado interno com private set e métodos de acesso controlado
	Valide na Fronteira Garanta invariantes no construtor e em métodos que modificam o estado
	Responsabilidade Única Cada classe deve ter um propósito claro e coeso no domínio do problema
	Código Expressivo Use nomes significativos, métodos pequenos e organize logicamente seu código
	Aprenda Praticando A verdadeira compreensão vem da aplicação prática dos conceitos em projetos reais

"A Programação Orientada a Objetos não é apenas uma técnica de codificação, mas uma forma de pensar sobre problemas e suas soluções."

Bons estudos e sucesso em sua jornada de programação!

Continue praticando, refatorando código existente e explorando novos conceitos. A fluência em POO vem com o tempo e a experiência, mas os fundamentos que você aprendeu aqui servirão como base sólida para toda a sua carreira de desenvolvimento.