Tipos, Classes e Objetos em C#: do cotidiano ao código

Material didático para estudo individual, orientação de aula prática e transformação em slides.

Objetivos de aprendizagem

1. **Diferenciar tipo, classe, objeto (instância) e identidade**

   Compreender os conceitos fundamentais da programação orientada a objetos e suas diferenças.

2. **Explicar estado e comportamento**

   Mapear estes conceitos para propriedades e métodos em C#.

3. **Distinguir tipos por valor de tipos por referência**

   Entender a diferença entre struct e class e prever os efeitos de cópia/atribuição.

4. **Aplicar encapsulamento e invariantes de domínio**

   Implementar regras que nunca podem ser quebradas em seu código.

5. **Implementar igualdade de domínio**

   Utilizar IEquatable e sobrescrever Equals/GetHashCode.

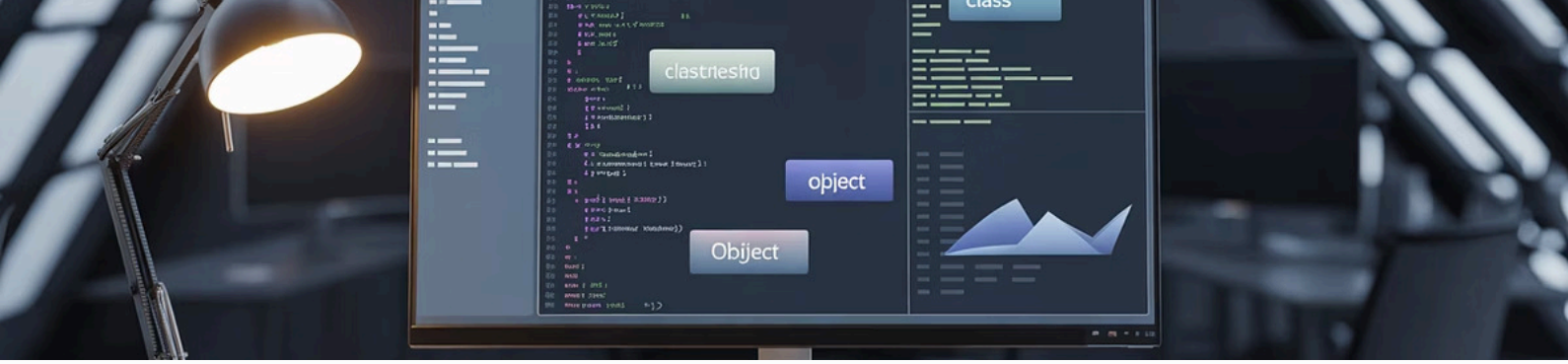
6. **Criar pequenos modelos de domínio**

   Desenvolver e evoluir incrementalmente modelos como Pedido/Item, Aluno/Notas, etc.

Antes de começar (ambiente)

Verifique o .NET SDK

```
dotnet --version
```



Crie um projeto console para seus experimentos

```
dotnet new console -n AulaPOO
cd AulaPOO
dotnet run
```

Configure seu ambiente de desenvolvimento

Abra a pasta no VS Code e habilite o C# Dev Kit (se disponível). Use Program.cs como "playground" ou crie arquivos adicionais para os exemplos deste material.

Dica didática: mantenha uma pasta experimentos/ com um arquivo por tópico (ex.: Produto.cs, Pedido.cs). Isso facilita comparar versões.

Conceitos fundamentais

Tipo

Um tipo descreve a forma e as operações possíveis sobre um conjunto de valores. Exemplos: int, double, string, DateTime, e tipos definidos por você (classes e structs).

Classe

Classe é a **definição abstrata** que reúne **estado** (dados) e **comportamento** (operações). Pense como uma "receita" ou "molde".

Objeto (instância)

Objeto é um **exemplar concreto** de uma classe, criado em C# normalmente com new. Cada objeto possui seu **próprio estado**.

Identidade

Dois objetos podem ter o **mesmo estado** e ainda assim serem **instâncias diferentes** em memória. Identidade é "quem é" o objeto (seu endereço/vida própria) e não apenas "como ele está".

No cotidiano:

- **Estado** de uma conta bancária: saldo atual, titular, número.
- **Comportamento**: depositar, sacar, encerrar.

Em C# mapeamos assim:

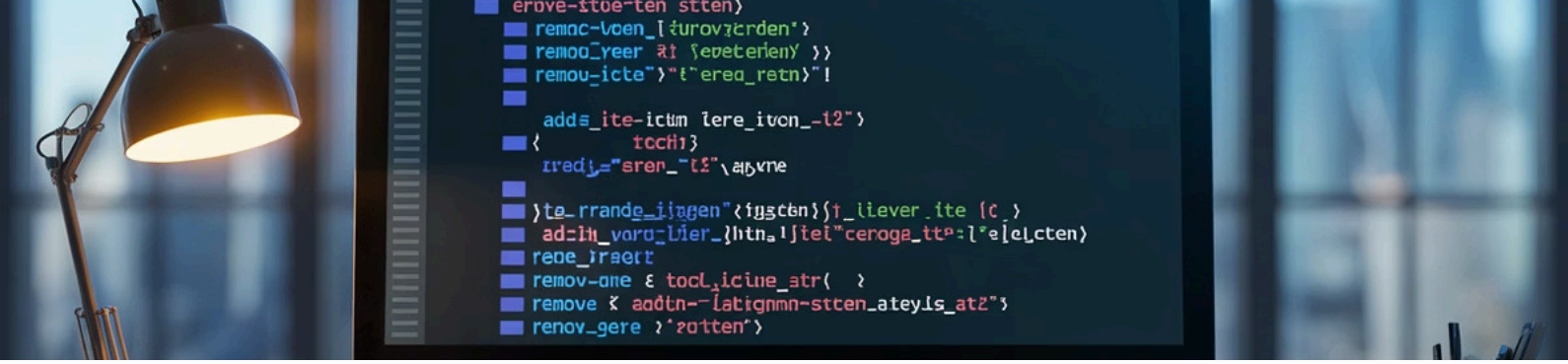
- **Propriedades** representam estado (com get/set).
- **Métodos** representam comportamento.

A classe representa o conceito abstrato, enquanto os objetos são instâncias concretas com estado próprio.

Exemplo base: Produto

```
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 void AdicionarAoEstoque(int qtd)
    {
        if (qtd <= 0)
            throw new ArgumentOutOfRangeException(nameof(qtd));
        QuantidadeEmEstoque += qtd;
    }
    public void RemoverDoEstoque(int qtd)
    {
        if (qtd <= 0 || qtd > QuantidadeEmEstoque)
            throw new ArgumentOutOfRangeException(nameof(qtd),
                "Quantidade inválida");
        QuantidadeEmEstoque -= qtd;
    }

    public double ValorTotal() => PrecoUnitario * QuantidadeEmEstoque;
    public override string ToString() => $"{Nome} |
        R$ {PrecoUnitario:F2} | Qtd: {QuantidadeEmEstoque}";
}
```

> ⓘ **Ideia chave:** o **construtor** garante que objetos só nascem **válidos**; os **métodos** preservam as **regras** (invariantes) ao longo da vida do objeto.

Tipos por Valor × Tipos por Referência

class → referência

Variáveis guardam ponteiros para o objeto no heap. Ao fazer p2 = p1, ambas **apontam para o mesmo objeto**.

struct → valor

O **valor é copiado**. c2 = c1 cria **cópia independente**.

Experimento guiado (cole no Program.cs)

```
var p1 = new Produto("Café", 18.90, 10);
var p2 = p1; // mesma referência
Console.WriteLine(object.ReferenceEquals(p1, p2)); // true

// Struct de exemplo
public struct Caneca
{
    public int CapacidadeEmMl { get; set; }
}

var c1 = new Caneca { CapacidadeEmMl = 300 };
var c2 = c1; // cópia por valor

c2.CapacidadeEmMl = 500;
Console.WriteLine(c1.CapacidadeEmMl); // 300 (não mudou)
Console.WriteLine(c2.CapacidadeEmMl); // 500
```

> **Quando usar struct?** Tipos pequenos, imutáveis, sem identidade própria marcante (ex.: Point, DateOnly). Para quase todo o "domínio" de negócio, use **classes**.

Encapsulamento e Invariantes

Encapsular é expor apenas o necessário e proteger regras. Técnicas práticas:

- Propriedades com private set.
- Validações no **construtor** e nos **métodos**.
- Preferir métodos que **expressem intenção** do domínio (ex.: RemoverDoEstoque).

Checklist rápido de invariantes:

Algum número não pode ser negativo?	Algum texto precisa de formato específico?
Existe limite superior para certa quantidade?	Quem pode mudar este estado (regra de autorização/fluxo)?

Igualdade de domínio

Às vezes "igualdade" não é ser o **mesmo objeto**, mas representar a **mesma entidade de negócio** (ex.: mesmo código de produto). Use IEquatable:

```
public class Produto : IEquatable
{
   // ... (demais membros)
   public string Codigo { get; }
   public Produto(string codigo, string nome, double preco, int qtd)
   {
      if (string.IsNullOrWhiteSpace(codigo))
        throw new ArgumentException("Código inválido");
      Codigo = codigo;
      // validações restantes...
      Nome = nome; PrecoUnitario = preco; QuantidadeEmEstoque = qtd;
   }
   public bool Equals(Produto? outro) =>
      outro is not null && string.Equals(Codigo, outro.Codigo,
         StringComparison.OrdinalIgnoreCase);

   public override bool Equals(object? obj) => Equals(obj as Produto);

   public override int GetHashCode() =>
     StringComparer.OrdinalIgnoreCase.GetHashCode(Codigo);
}
```

ⓘ Com esta implementação, dois produtos com o mesmo código serão considerados iguais, mesmo sendo objetos diferentes em memória.

Estudos de caso do cotidiano → mapeados para OO

Carrinho de mercado

Tipos

Pedido (agrega) e ItemPedido (compõe).

Estado

Itens, quantidade, preço unitário.

Comportamento

AdicionarItem, RemoverItem, Total().

```
public class ItemPedido
{
 public Produto Produto { get; }
 public int Quantidade { get; private set; }

 public ItemPedido(Produto produto, int quantidade)
 {
   if (quantidade <= 0)
     throw new ArgumentOutOfRangeException(nameof(quantidade));
   Produto = produto ??
     throw new ArgumentNullException(nameof(produto));
   Quantidade = quantidade;
 }

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

Classe Pedido

```
public class Pedido
{
   private readonly List _itens = new();
   public IReadOnlyCollection Itens => _itens.AsReadOnly();
   public void AdicionarItem(Produto p, int qtd)
   {
      if (p is null)
        throw new ArgumentNullException(nameof(p));
      if (qtd <= 0)
        throw new ArgumentOutOfRangeException(nameof(qtd));
      _itens.Add(new ItemPedido(p, qtd));
   }
   public void RemoverItemPorNome(string nome)
   {
      var idx = _itens.FindIndex(i => i.Produto.Nome.Equals(nome,
        StringComparison.OrdinalIgnoreCase));
      if (idx >= 0)
       _itens.RemoveAt(idx);
   }

   public double Total() => _itens.Sum(i => i.Subtotal());
}
```

Observe como o encapsulamento protege a lista de itens, expondo apenas uma coleção somente leitura e métodos específicos para manipulação.

Vida acadêmica

Tipos

Aluno, Avaliacao.

Estado

Notas, pesos.

Comportamento

Media(), Aprovado(limite).

```
public class Avaliacao
{
 public string Descricao { get; }
 public double Nota { get; }
 public double Peso { get; }

 public Avaliacao(string desc, double nota, double peso)
 {
   if (nota < 0 || nota > 10)
     throw new ArgumentOutOfRangeException(nameof(nota));
   if (peso <= 0)
     throw new ArgumentOutOfRangeException(nameof(peso));
   Descricao = desc; Nota = nota; Peso = peso;
 }
}
```

Classe Aluno

```
public class Aluno
{
   private readonly List _avaliacoes = new();
   public string Nome { get; }

   public Aluno(string nome) => Nome = string.IsNullOrWhiteSpace(nome) ?
     throw new ArgumentException("Nome inválido") : nome;
   public void Registrar(double nota, double peso, string desc = "") =>
     _avaliacoes.Add(new Avaliacao(desc, nota, peso));
   public double Media() => _avaliacoes.Sum(a => a.Nota * a.Peso) /
     _avaliacoes.Sum(a => a.Peso);

   public bool Aprovado(double limite = 6.0) => Media() >= limite;
```


```
    public override string ToString() => $"{Nome} | Média: {Media():F2} | Aprovado: {Aprovado()}";
}
```

Note como o cálculo da média ponderada é encapsulado no método Media(), e o status de aprovação é determinado pelo método Aprovado().

Cozinha (receitas/bolos)

Tipos

Receita e Bolo.

Estado

Ingredientes, cobertura.

Comportamento

Assar(), Cobrir().

Conceito reforçado: cada **Bolo** é um **objeto** distinto, mesmo que venha da mesma **Receita**.

Este exemplo ilustra perfeitamente a relação entre classe e objeto: a receita é o "molde" (classe) e cada bolo é uma "instância" única com seu próprio estado.

Laboratórios guiados (incrementais)

Como usar: leia o objetivo, copie o código base, rode, depois implemente as tarefas. Documente em comentários o que mudou e por quê.

Lab A – "Receita → Bolo"

1. Modele Receita (nome, lista de ingredientes) e um método Assar() que retorne um Bolo.
2. Crie **dois** bolos a partir da **mesma** receita e altere coberturas diferentes.
3. Mostre no console que são objetos distintos (compare com ReferenceEquals).

Lab B – Estoque com invariantes

1. Partindo de Produto, proíba preço negativo e remoções acima da quantidade.
2. Escreva um teste simples no Program.cs que tente quebrar as regras e capture exceções.

Mais Laboratórios

Lab C – Pedido/ItemPedido

1. Adicione itens, remova por nome e calcule o total.
2. Faça **testes de fronteira** (qtd = 0, preços com centavos).

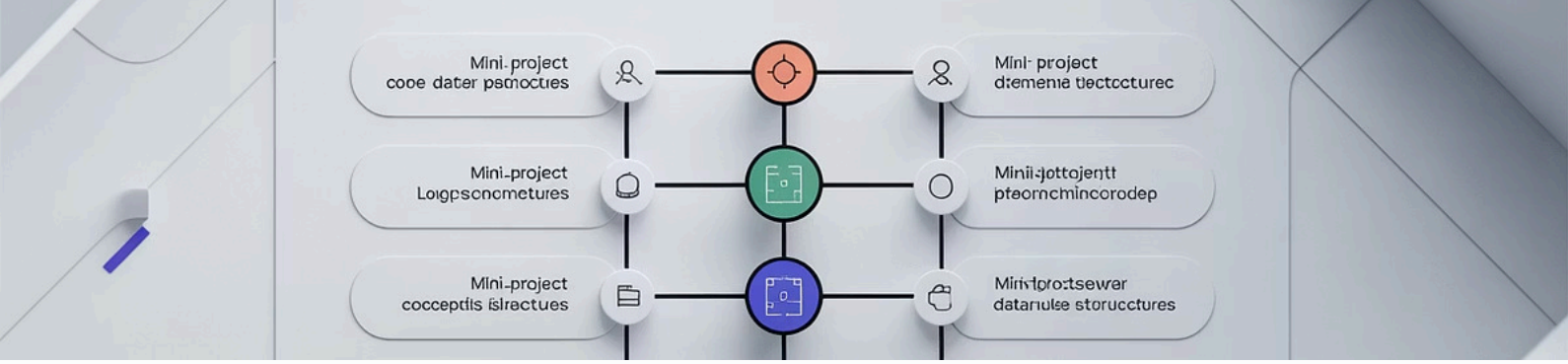
Lab D – Aluno/Notas

1. Registre avaliações com pesos.
2. Exiba a média e o status de aprovação.

Lab E – Igualdade de domínio

1. Adicione Codigo a Produto e implemente IEquatable.
2. Crie dois objetos com mesmo código e compare com Equals (true) e ReferenceEquals (false).

Dica: ao terminar cada Lab, escreva em uma linha: *"O que eu aprendi?"* e *"Qual regra eu garanto com meu código?"*.



Ponte para quem vem de C (struct + funções → classe)

Liste uma struct sua e as funções que a manipulam

Por exemplo: AlunoC com funções Cadastrar, CalcularMedia.

Refatore para uma classe com construtor e métodos

Transforme em uma classe com métodos Registrar, Media, Aprovado.

Garanta as regras via encapsulamento

Valide parâmetros e proteja o estado interno.

Meta cognitiva: observe como a **responsabilidade** migra das funções dispersas para **dentro** do tipo.

Estratégias de estudo

Leitura ativa

Sublinhe palavras-âncora (estado, comportamento, identidade, invariante). Escreva uma frase sua definindo cada termo.

Técnica Feynman

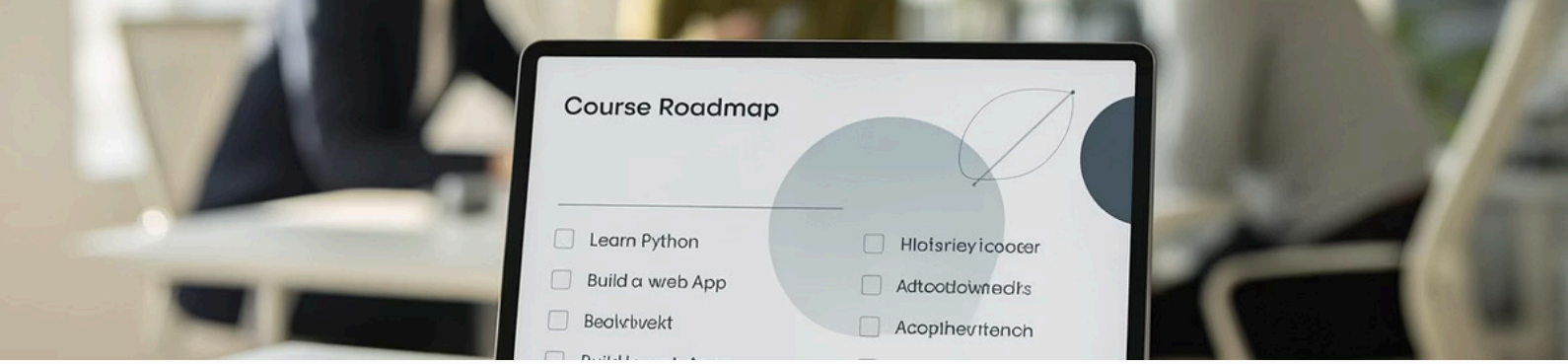
Explique a alguém (ou a si mesmo) o que é **classe** sem usar jargão depois revise o que ficou confuso.

Mapa mental rápido

Faça um diagrama com "Tipo → Classe → Objeto → Identidade" e anote exemplos reais ao lado.

Flashcards pessoais

Frente: *"Diferença entre struct e class?"*; verso: **valor vs referência**, cópia vs compartilhamento, quando escolher cada um.



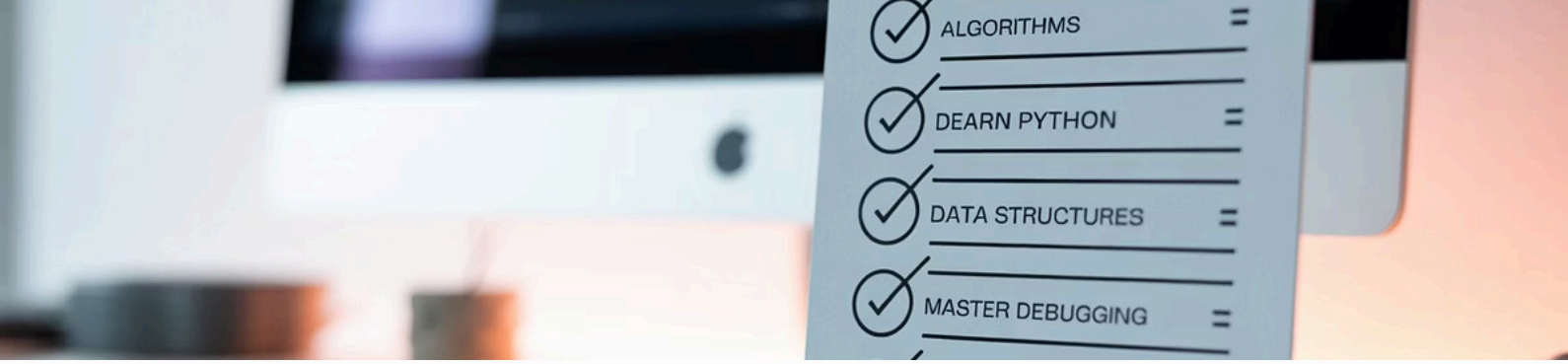
Roteiro de "missão de estudo"

Objetivo da missão: consolidar os fundamentos e entregar mini-projetos executáveis que demonstrem domínio do paradigma.

1. **Setup do projeto AulaPOO**
   Criar e configurar o ambiente de desenvolvimento.
2. **Implementar Produto com invariantes e ToString()**
   Completar o Lab B com validações e testes.
3. **Implementar Pedido/ItemPedido com Total()**
   Completar o Lab C com testes de fronteira.
4. **Implementar Aluno com Media() e Aprovado()**
   Completar o Lab D com testes de aprovação.
5. **Implementar igualdade de domínio em Produto**
   Completar o Lab E com testes de igualdade.
6. **Escrever um Program.cs demonstrativo**
   Criar um programa que demonstre todos os cenários e imprima resultados.

Critérios de conclusão (autoavaliação):

- Sem exceções inesperadas em tempo de execução.
- Regras de negócio cobertas.
- Saída do console explica o que está sendo testado.
- Código comentado apenas onde agrega entendimento (sem ruído).



Checklist final e próximos passos

Checklist final de consolidação

- Consigo definir **tipo**, **classe**, **objeto** e **identidade** com minhas próprias palavras.
- Sei apontar o **estado** e o **comportamento** em qualquer exemplo do cotidiano.
- Prevejo corretamente os efeitos de **atribuição** em class e struct.
- Aplico **encapsulamento** e **invariantes** ao modelar um domínio simples.
- Diferencio **igualdade por identidade** de **igualdade por domínio**.

Próximos passos

- **Coleções genéricas** (List, Dictionary): organizar múltiplos objetos.
- **LINQ**: consultas e projeções sobre coleções (já com base em tipos).
- **Interfaces** e **injeção de dependência**: programar para **contratos**.
- **Herança e polimorfismo**: especialização de comportamentos (usar com critério; prefira composição sempre que possível).

Exercícios propostos

Ex. 1: Crie dois Produto com mesmo Codigo e preços diferentes. Compare com Equals e com ReferenceEquals.

Saída esperada: Equals: True | ReferenceEquals: False.

Ex. 2: Em Pedido, adicione 3 itens e imprima Total() com casas decimais.

Saída esperada: Total do pedido: 123,45.

Ex. 3: Em Aluno, registre notas (7, 8 e 6) com pesos (2, 3, 1). Imprima média e status.

Saída esperada: Média: 7,50 | Aprovado: True.

Ex. 4: No exemplo de **struct** Caneca, mostre que alterar c2 não altera c1.

Saída esperada: c1=300 | c2=500.