DO CONSOLE AO JOGO ONLINE: C# EM 1 ANO

Caderno de Respostas

Gabarito comentado: 73 aulas de prática, do #01 ao #73

C# 14 · .NET 10 · 10 módulos · 88 questões · 2 desafios integradores

Autor Desconhecido · theknowledgeisle.com

Consulte este gabarito só depois de tentar.

As respostas são comentadas para ensinar, não só para conferir. Nos exercícios de código existem outras soluções igualmente válidas: o que importa é entender por que a sua funciona e o que ela troca em relação à mostrada aqui.

Os programas com um quadro verde de **Saída** foram executados com o .NET 10 (dotnet run arquivo.cs) no Linux, e a saída mostrada é a real.

Sumário

Módulo	Tema	Aulas	Questões
MÓDULO 1	Fundamentos da linguagem	#01–#08	Q1–Q13
MÓDULO 2	Orientação a objetos	#09–#15	Q14–Q23
MÓDULO 3	Genéricos, delegates e LINQ	#16–#21	Q24–Q32
MÓDULO 4	A plataforma por dentro	#22–#30	Q33–Q44
MÓDULO 5	Testes e bancos de dados	#31–#42	Q45–Q53
MÓDULO 6	ASP.NET Core e serviços	#43–#52	Q54–Q65
MÓDULO 7	A vitrine web e o deploy	#53–#59	Q66–Q72
MÓDULO 8	Aplicativos de desktop e celular	#60–#62	Q73–Q77
MÓDULO 9	Jogos	#63–#69	Q78–Q82
MÓDULO 10	Arquitetura, IA e desempenho	#70–#73	Q83–Q88
DESAFIOS	Desafios integradores	todas	D1–D2

MÓDULO 1

Fundamentos da linguagem

A história do C#, o .NET por dentro, tipos e conversões, texto e console, decisões e laços, métodos, coleções e exceções.

Artigos 01 a 08

#01 Da Rivalidade com o Java ao Código Aberto: a História do C#
#02 Do Código-Fonte ao Processo: Instalando o .NET e Entendendo o que Acontece ao Rodar
#03 Cada Valor Tem um Tipo: Variáveis, Números e Conversões em C#
#04 Operadores, Texto e Conversa com o Usuário
#05 O Programa Que Escolhe e Repete: if, switch e Laços
#06 Métodos: Dividindo o Problema em Partes Que Conversam
#07 Guardando Muitos Valores: Arrays, Listas e Dicionários
#08 Quando Algo Dá Errado: Exceções em C#

Perguntas conceituais — respostas

Q1 Do código ao processo

Descreva o caminho de um `Program.cs` até a execução: o que o compilador gera, o que é IL e o que o JIT faz. Por que o mesmo `.dll` roda em Windows, Linux e macOS?

O compilador do C# (Roslyn) não gera código de máquina: gera **IL** (*Intermediate Language*), um código para uma máquina virtual, guardado num *assembly* (`.dll`) junto com os metadados dos tipos. Ao executar, o *runtime* (CLR) carrega o *assembly* e o **JIT** (*just-in-time*) traduz cada método para o código de máquina do processador **na primeira vez em que ele é chamado**. Como o IL não depende do processador nem do sistema, o mesmo `.dll` roda em qualquer plataforma que tenha o *runtime* do .NET; só o executável de partida (o *apphost*) é específico. O Native AOT, visto no fim do curso, faz o caminho inverso: compila tudo para código de máquina antes, sem JIT.

Q2 Tipos de valor e de referência

Qual a diferença entre `int` (tipo de valor) e `string` ou uma `class` (tipos de referência) numa atribuição `b = a`? Por que a `string`, sendo de referência, se comporta "como valor"?

Num **tipo de valor**, a variável contém o próprio dado: `b = a` copia o valor, e as duas variáveis seguem independentes. Num **tipo de referência**, a variável contém uma referência para um objeto no *heap*: `b = a` copia a referência, e as duas passam a apontar para **o mesmo** objeto; alterar o objeto por `b` aparece em `a`. A `string` é de referência, mas é **imutável**: nenhum método a altera; `ToUpper`, `Replace` e `+` criam strings novas. Como ninguém consegue mudar o objeto compartilhado, compartilhá-lo é indistinguível de copiá-lo.

Q3 decimal, double e dinheiro

Por que `0.1 + 0.2 == 0.3` é falso com `double` e verdadeiro com `decimal`? Qual usar para preços na livraria do curso, e qual o custo?

O `double` guarda números em **binário** (IEEE 754), e 0,1 não tem representação binária finita, assim como 1/3 não tem em decimal: o valor guardado é o mais próximo possível, e os erros se somam. O `decimal` guarda um inteiro de 96 bits e uma **escala decimal**, então 0,1 é exatamente 1×10^{-1}. Para dinheiro, use `decimal`: centavos não podem sumir num arredondamento. O custo é desempenho (as contas são feitas em software, muitas vezes mais lentas) e alcance menor (cerca de $7{,}9 \times 10^{28}$, contra $1{,}8 \times 10^{308}$ do `double`). Para física, gráficos e estatística, `double`.

Q4 Parse ou TryParse

> *Qual a diferença entre `int.Parse(texto)` e `int.TryParse(texto, out int n)`? Qual usar para ler o que o usuário digitou, e por quê?*

O `Parse` **lança** `FormatException` (ou `OverflowException`) quando o texto não é um número; o `TryParse` devolve `false` e deixa `n` com 0, sem exceção. Para entrada do usuário, use `TryParse`: digitar errado é **esperado**, não excepcional, e um `if` deixa o fluxo mais claro e mais barato que um `try/catch`. O `Parse` cabe onde um texto inválido é de fato um erro de programa, como um valor de configuração que deveria estar sempre correto. Os dois aceitam um `IFormatProvider`: `double.Parse("1,5")` depende da cultura da máquina; para formato fixo, use `CultureInfo.InvariantCulture`.

Q5 ref, out e in

> *Explique a diferença entre os modificadores de parâmetro `ref`, `out` e `in`. Por que o C# exige escrever `ref` também na chamada?*

- ** `ref` **: passa a própria variável (por referência). Ela precisa estar inicializada antes, e o método pode ler e alterar.
- ** `out` **: também por referência, mas para** saída**: não precisa estar inicializada, e o método é** obrigado** a atribuir antes de retornar (como no `TryParse`).
- ** `in` **: por referência e** somente leitura**; serve para evitar a cópia de *structs* grandes.

Exigir o modificador na chamada (`Trocar(ref a, ref b)`) deixa visível, no ponto de uso, que a variável pode mudar. É a resposta do C# ao problema das referências escondidas que o C++ tem.

Q6 List, array ou Dictionary

> *Para cada caso, escolha entre `T[]`, `List<T>` e `Dictionary<TKey, TValue>`: (a) as notas de um aluno numa prova com 10 questões; (b) os itens de um carrinho de compras; (c) buscar um livro pelo ISBN.*

- (a) **Array**: o tamanho é fixo e conhecido (10).
- (b) ** `List<T>` **: itens entram e saem; a lista cresce sozinha.
- (c) ** `Dictionary<string, Livro>` **: busca pela chave em O(1) em média. Procurar o ISBN numa lista seria percorrê-la inteira a cada busca, em O(n).

Q7 Quando capturar uma exceção

> *"Capture apenas as exceções que você sabe tratar." Explique a regra e por que `catch (Exception) { }` vazio é um dos piores trechos que um programa pode ter.*

Capturar uma exceção é afirmar que, naquele ponto, existe uma **ação útil**: tentar de novo, usar um valor padrão, avisar o usuário. Se não há, a exceção deve subir até um nível que saiba o que fazer, que no pior caso registra o erro e encerra. O `catch (Exception) { }` vazio engole **qualquer** erro, inclusive bugs (`NullReferenceException`, `IndexOutOfRangeException`): o programa segue num estado inconsistente, e o problema reaparece depois, longe da causa e sem nenhum rastro. Se for preciso registrar e repassar, use `throw;` (sem argumento), que preserva o *stack trace* original; `throw ex;` o reinicia.

Exercícios de código — respostas

Q8 Lendo um número com segurança

Escreva um programa que leia linhas da entrada até receber um inteiro entre 1 e 100, avisando quando o texto não for número e quando estiver fora do intervalo. Use `int.TryParse`. Em seguida, mostre o número, o dobro e se é par.

```
int? lido = null;
while (lido is null)
{
    Console.Write("Número entre 1 e 100: ");
    string? linha = Console.ReadLine();
    if (linha is null)
    {
        Console.WriteLine("fim da entrada");
        return;
    }
    if (!int.TryParse(linha, out int n))
        Console.WriteLine($"\"{linha}\" não é um número inteiro");
    else if (n is < 1 or > 100)
        Console.WriteLine($"{n} está fora do intervalo");
    else
        lido = n;
}
Console.WriteLine($"lido {lido}, dobro {lido * 2}, {(lido % 2 == 0 ? "par" : "ímpar")}");
```

Entrada digitada:

```
dez
150
42
```

Saída

```
Número entre 1 e 100: "dez" não é um número inteiro
Número entre 1 e 100: 150 está fora do intervalo
Número entre 1 e 100: lido 42, dobro 84, par
```

O `ReadLine` devolve `null` quando a entrada acaba (Ctrl+D no Linux, ou o fim de um arquivo redirecionado), e o tipo `string?` obriga a tratar esse caso. O padrão `n is < 1 or > 100` (C# 9) lê como a frase que descreve a regra.

Q9 Formatando valores em reais

Mostre o preço `1234.5m` formatado: (a) com a cultura `pt-BR` como moeda; (b) com duas casas e separador de milhar em `pt-BR`; (c) na cultura invariante. Depois leia o texto `"1.234,50"` de volta para `decimal` usando `pt-BR`.

```
using System.Globalization;

var br = CultureInfo.GetCultureInfo("pt-BR");
decimal preco = 1234.5m;

Console.WriteLine(preco.ToString("C", br));
Console.WriteLine(preco.ToString("N2", br));
Console.WriteLine(preco.ToString("N2", CultureInfo.InvariantCulture));

decimal lido = decimal.Parse("1.234,50", NumberStyles.Number, br);
Console.WriteLine($"lido de volta: {lido.ToString(CultureInfo.InvariantCulture)} (igual? {lido ==
preco})");
```

Saída

```
R$ 1.234,50
1.234,50
1,234.50
lido de volta: 1234.50 (igual? True)
```

Sem a cultura explícita, o resultado depende da configuração da máquina: o mesmo programa mostra "1,234.50" num servidor em inglês. Para gravar em arquivo ou trocar dados entre sistemas, use sempre `InvariantCulture`; para mostrar ao usuário, a cultura dele. O espaço depois do `R$` é um espaço inseparável (U+00A0), detalhe que derruba comparações de texto feitas sem cuidado.

Q10 switch moderno

Escreva `static string Classificar(int idade)` usando uma expressão `switch` com padrões relacionais (`< 0`, `< 13`, `< 18`, `< 60`, `_`), devolvendo "inválida", "criança", "adolescente", "adulto" ou "idoso". Depois escreva `static decimal Frete(string uf, decimal total)` com uma expressão `switch` sobre uma tupla: frete grátis acima de R$ 200, R$ 10 para SP e RJ, R$ 25 para as demais UFs.

```
static string Classificar(int idade) => idade switch
{
    < 0 => "inválida",
    < 13 => "criança",
    < 18 => "adolescente",
    < 60 => "adulto",
    _ => "idoso",
};

static decimal Frete(string uf, decimal total) => (uf, total) switch
{
    (_, > 200m) => 0m,
    ("SP" or "RJ", _) => 10m,
    _ => 25m,
};

foreach (var i in new[] { -1, 8, 15, 30, 70 })
    Console.WriteLine($"{i,3}: {Classificar(i)}");
Console.WriteLine($"SP R$ 50: {Frete("SP", 50m)} | BA R$ 50: {Frete("BA", 50m)} | BA R$ 300:
{Frete("BA", 300m)}");
```

Saída

```
 -1: inválida
  8: criança
 15: adolescente
 30: adulto
 70: idoso
SP R$ 50: 10 | BA R$ 50: 25 | BA R$ 300: 0
```

Os casos são testados **na ordem**: `< 13` só é avaliado se `< 0` falhou, por isso não é preciso escrever `>= 0 and < 13`. Se um caso nunca puder ser alcançado (por exemplo, `< 10` depois de `< 13`), o compilador acusa erro. Sem o `_` final, ele avisa que o `switch` não cobre todos os valores.

Q11 Métodos com parâmetros opcionais e out

Escreva `static bool TentarDividir(int a, int b, out int quociente, out int resto)` e `static string Repetir(string texto, int vezes = 2, string separador = " ")`. Chame o segundo com argumentos nomeados, pulando um dos opcionais.

```
static bool TentarDividir(int a, int b, out int quociente, out int resto)
{
    if (b == 0)
    {
        quociente = resto = 0;  // out obriga a atribuir mesmo na falha
        return false;
    }
    quociente = a / b;
    resto = a % b;
    return true;
}

static string Repetir(string texto, int vezes = 2, string separador = " ")
    => string.Join(separador, Enumerable.Repeat(texto, vezes));

if (TentarDividir(17, 5, out int q, out int r))
    Console.WriteLine($"17 / 5 = {q}, resto {r}");
if (!TentarDividir(1, 0, out _, out _))
    Console.WriteLine("divisão por zero recusada");

Console.WriteLine(Repetir("oi"));
Console.WriteLine(Repetir("ha", 3, ""));
Console.WriteLine(Repetir("lá", separador: "-"));
```

Saída

```
17 / 5 = 3, resto 2
divisão por zero recusada
oi oi
hahaha
lá-lá
```

O `out _` descarta um resultado que não interessa. Os valores padrão dos parâmetros opcionais são **copiados para o código de quem chama** na compilação: mudar o padrão numa biblioteca não afeta quem já foi compilado contra a versão antiga, o que é motivo para usá-los com parcimônia em APIs públicas.

Q12 Estoque com Dictionary

> *Escreva um programa que mantenha o estoque de uma livraria num `Dictionary<string, int>` (título → quantidade), processe uma lista de movimentações (`"+Dom Casmurro:3"`, `"-Vidas Secas:1"`) e recuse saídas maiores que o estoque. Mostre o estoque final em ordem alfabética.*

```
var estoque = new Dictionary<string, int> { ["Dom Casmurro"] = 2, ["Vidas Secas"] = 1 };
string[] movimentos = ["+Dom Casmurro:3", "-Vidas Secas:1", "-Vidas Secas:1", "+Macunaíma:4", "-
Iracema:1"];

foreach (var m in movimentos)
{
    char sinal = m[0];
    var partes = m[1..].Split(':');
    string titulo = partes[0];
    int qtd = int.Parse(partes[1]);
    int atual = estoque.GetValueOrDefault(titulo);  // 0 se o título não existe

    if (sinal == '-' && qtd > atual)
    {
        Console.WriteLine($"recusado: {m} (há {atual})");
        continue;
    }
    estoque[titulo] = atual + (sinal == '+' ? qtd : -qtd);
}

foreach (var (titulo, qtd) in estoque.OrderBy(p => p.Key))
    Console.WriteLine($"{titulo,-14} {qtd}");
```

Saída

```
recusado: -Vidas Secas:1 (há 0)
recusado: -Iracema:1 (há 0)
Dom Casmurro   5
Macunaíma      4
Vidas Secas    0
```

O indexador `estoque[titulo] = ...` cria a chave se ela não existe e substitui o valor se existe; ler `estoque["Iracema"]` de uma chave inexistente, ao contrário, lança `KeyNotFoundException`, por isso o `GetValueOrDefault`. A ordem de iteração de um `Dictionary` não é garantida; o `OrderBy` deixa a saída previsível. Note que "Vidas Secas" ficou com 0 e continua no dicionário.

Q13 Exceções próprias

Crie `class EstoqueInsuficienteException : Exception` com as propriedades `Titulo` e `Disponivel`. Escreva `static void Vender(Dictionary<string, int> estoque, string titulo, int qtd)` que a lance quando faltar estoque e `ArgumentOutOfRangeException` para quantidade menor que 1. Trate as duas de forma diferente e use `finally` para registrar o fim de cada tentativa.

```
var estoque = new Dictionary<string, int> { ["Dom Casmurro"] = 2 };

foreach (var (titulo, qtd) in new[] { ("Dom Casmurro", 1), ("Dom Casmurro", 5), ("Dom Casmurro",
0) })
{
    try
    {
        Vender(estoque, titulo, qtd);
        Console.WriteLine($"vendidos {qtd}; restam {estoque[titulo]}");
    }
    catch (EstoqueInsuficienteException e)
    {
        Console.WriteLine($"{e.Message} Disponível: {e.Disponivel}.");
    }
    catch (ArgumentOutOfRangeException e)
    {
        Console.WriteLine($"erro de programação: {e.ParamName} inválido");
    }
    finally
    {
        Console.WriteLine("  (tentativa registrada)");
    }
}

static void Vender(Dictionary<string, int> estoque, string titulo, int qtd)
{
    ArgumentOutOfRangeException.ThrowIfLessThan(qtd, 1);
    int disponivel = estoque.GetValueOrDefault(titulo);
    if (qtd > disponivel)
        throw new EstoqueInsuficienteException(titulo, disponivel);
    estoque[titulo] = disponivel - qtd;
}

class EstoqueInsuficienteException(string titulo, int disponivel)
    : Exception($"Estoque insuficiente de \"{titulo}\".")
{
    public string Titulo { get; } = titulo;
    public int Disponivel { get; } = disponivel;
}
```

Saída

```
vendidos 1; restam 1
  (tentativa registrada)
Estoque insuficiente de "Dom Casmurro". Disponível: 1.
  (tentativa registrada)
erro de programação: qtd inválido
  (tentativa registrada)
```

A ordem dos `catch` vai do mais específico ao mais geral; o compilador recusa um `catch (Exception)` antes de um mais específico. O `ThrowIfLessThan` (.NET 8) preenche o `ParamName` sozinho, a partir do nome do argumento. Num arquivo com instruções de nível superior, os tipos (`class`) precisam vir **depois** delas.

MÓDULO 2

Orientação a objetos

Classes e objetos da livraria, propriedades, herança, interfaces, class/struct/record, enums e pattern matching, namespaces, projetos e pacotes NuGet.

Artigos 09 a 15

#09 Classes e Objetos: Dando Forma aos Dados da Livraria
#10 Propriedades e Encapsulamento: Deixar Ler, Controlar a Escrita
#11 Herança: Quando um Tipo É uma Versão Especializada de Outro
#12 Interfaces: Contratos Que Qualquer Tipo Pode Assinar
#13 Classe, Struct ou Record: Identidade, Valor e Igualdade
#14 Enums e Pattern Matching: Nomes para Estados e Perguntas sobre Formas
#15 Do Arquivo Único à Solução: Namespaces, Projetos e Pacotes NuGet

Perguntas conceituais — respostas

Q14 Campo ou propriedade

> *Por que, em C#, os dados públicos de uma classe são expostos como propriedades (`public decimal Preco { get; private set; }`) e não como campos públicos? Cite duas vantagens concretas.*

- **Validação e regras**: uma propriedade pode ter lógica no `set` (recusar preço negativo) ou no `get` (calcular um valor), e essa lógica pode ser acrescentada **depois**, sem mudar quem usa a classe. Trocar um campo público por propriedade quebra a compatibilidade binária: quem foi compilado contra o campo precisa ser recompilado.
- **Acesso diferenciado**: `get` público com `private set` (ou `init`) deixa todos lerem e só a própria classe alterar.

Além disso, *data binding*, serialização e muitos *frameworks* (WPF, EF Core) trabalham com propriedades, não com campos.

Q15 Herança ou interface

> *A livraria vende livros físicos e e-books. Quando modelar `LivroFisico` e `Ebook` herdando de uma classe `Produto`, e quando fazer os dois implementarem uma interface `IVendavel`? Por que o C# permite várias interfaces, mas uma só classe base?*

Herança quando há uma relação "é um" com **estado e comportamento comuns** a reaproveitar: título, preço e as regras de desconto vivem em `Produto`. **Interface** quando o que se compartilha é só um **contrato**, entre tipos que podem não ter nada mais em comum: um `IVendavel` pode ser implementado por um livro, um vale-presente e uma assinatura. A herança simples evita o "problema do diamante" (duas bases com o mesmo membro, com implementações diferentes); as interfaces, que tradicionalmente não têm estado, podem ser combinadas à vontade. Desde o C# 8 elas podem ter implementações padrão, mas continuam sem campos de instância.

Q16 class, struct ou record

Para cada tipo da livraria, escolha `class`, `struct`, `record` ou `record struct`, justificando: (a) `Cliente`, com identidade e dados que mudam; (b) `Isbn`, um código de 13 dígitos; (c) `ItemDoPedido` (título, quantidade e preço unitário), comparado por valor; (d) `Coordenada` com dois `double`, criada aos milhões num mapa.

- (a) ** `class` **: dois clientes com o mesmo nome continuam pessoas diferentes (identidade), e o objeto muda ao longo do tempo.
- (b) ** `readonly record struct` *: é um valor pequeno e imutável, comparado pelo conteúdo, e vale a pena não alocar no *heap*.
- (c) ** `record` ** (classe): igualdade por valor, `ToString` legível e `with` para criar cópias alteradas, sem se preocupar com tamanho.
- (d) ** `record struct` ** (ou `struct`): 16 bytes, criado em grande quantidade; como tipo de valor, fica dentro do array que o contém, sem uma alocação por coordenada para o coletor de lixo acompanhar.

Regra geral: `struct` só para tipos pequenos (até uns 16 bytes), imutáveis e com semântica de valor; o resto, `class` ou `record`.

Q17 Pattern matching

O que este método devolve para um `Ebook` de 30 reais, para um `LivroFisico` de 0,5 kg e para `null`? Por que a ordem dos casos importa?

```
static decimal Frete(Produto? p) => p switch
{
    null => throw new ArgumentNullException(nameof(p)),
    Ebook => 0m,
    LivroFisico { PesoKg: <= 1 } => 10m,
    LivroFisico l => 10m + (decimal)(l.PesoKg - 1) * 5m,
    _ => 15m,
};
```

- `Ebook` de 30 reais: **0**. O padrão de tipo `Ebook` casa, e o preço nem é consultado.
- `LivroFisico` de 0,5 kg: **10**. O padrão de propriedade `{ PesoKg: <= 1 }` casa antes do caso geral.
- `null`: lança `ArgumentNullException`.

A ordem importa porque o primeiro caso que casa vence: se `LivroFisico l` viesse antes de `LivroFisico { PesoKg: <= 1 }`, o segundo nunca seria alcançado, e o compilador acusaria **erro** por isso. O `_` final cobre os outros produtos (um vale-presente, por exemplo).

Q18 Pacotes e projetos

Qual a diferença entre uma solution (`.sln` ou `.slnx`), um projeto (`.csproj`) e um pacote NuGet? Como a livraria seria dividida em projetos, e o que significa um `<PackageReference Include="Dapper" Version="2.1.66" />`?

O **projeto** é a unidade de compilação: gera um *assembly* (`.dll` ou executável) e declara as dependências. A **solution** só agrupa projetos para abri-los e compilá-los juntos. Um **pacote NuGet** é um *assembly* publicado e versionado, baixado de um repositório (nuget.org). A livraria poderia ter `Livraria.Dominio` (biblioteca de classes, sem dependências), `Livraria.Dados` (que referencia o domínio e os pacotes de banco), `Livraria.Api` e `Livraria.Testes`. O `PackageReference` pede o pacote Dapper na versão 2.1.66; o `dotnet restore` o baixa, e as dependências dele vêm junto. Fixar a versão exata dá *builds* reprodutíveis.

Exercícios de código — respostas

Q19 Uma classe com invariante

Escreva `class Livro` com `Titulo` (obrigatório, não vazio), `Preco` (não negativo, com `private set`) e o método `AplicarDesconto(decimal percentual)`, que aceita de 0 a 50% e arredonda para centavos. Use `ArgumentException` e `ArgumentOutOfRangeException`. Mostre uma criação válida, um desconto e duas tentativas inválidas.

```
var l = new Livro("Vidas Secas", 59.90m);
l.AplicarDesconto(15);
Console.WriteLine($"{l.Titulo}: R$ {l.Preco}");

try { l.AplicarDesconto(80); }
catch (ArgumentOutOfRangeException e) { Console.WriteLine($"recusado: {e.ParamName} =
{e.ActualValue}"); }

try { _ = new Livro("  ", 10m); }
catch (ArgumentException e) { Console.WriteLine($"recusado: {e.Message}"); }

class Livro
{
    public string Titulo { get; }
    public decimal Preco { get; private set; }

    public Livro(string titulo, decimal preco)
    {
        ArgumentException.ThrowIfNullOrWhiteSpace(titulo);
        ArgumentOutOfRangeException.ThrowIfNegative(preco);
        Titulo = titulo;
        Preco = preco;
    }

    public void AplicarDesconto(decimal percentual)
    {
        if (percentual is < 0 or > 50)
            throw new ArgumentOutOfRangeException(nameof(percentual), percentual, "entre 0 e
50");
        Preco = Math.Round(Preco * (1 - percentual / 100), 2, MidpointRounding.AwayFromZero);
    }
}
```

Saída

```
Vidas Secas: R$ 50.92
recusado: percentual = 80
recusado: The value cannot be an empty string or composed entirely of whitespace. (Parameter
'titulo')
```

O `Math.Round` sem o terceiro argumento usa o **arredondamento bancário** (`ToEven`): 2,345 vira 2,34, e não 2,35. Para preços, a regra comercial costuma ser `AwayFromZero`. Como todo caminho de alteração passa pelo construtor ou pelo `AplicarDesconto`, nenhum `Livro` pode ter preço negativo. As mensagens das exceções da biblioteca saem em inglês: o .NET moderno não traz as traduções delas; para o usuário final, a mensagem deve vir do seu próprio código.

Q20 Herança com métodos virtuais

> *Escreva `abstract class Produto` com `Titulo`, `PrecoBase` e `public virtual decimal PrecoFinal() => PrecoBase;`, e duas derivadas: `LivroFisico` (acrescenta R$ 5 de embalagem) e `Ebook` (10% mais barato). Some o total de um carrinho com os dois tipos numa `List<Produto>` e mostre por que `ToString()` sobrescrito ajuda a depurar.*

```
List<Produto> carrinho = [new LivroFisico("Dom Casmurro", 40m), new Ebook("Dom Casmurro", 40m),
new LivroFisico("Iracema", 25m)];

foreach (var p in carrinho)
    Console.WriteLine(p);
Console.WriteLine($"Total: {carrinho.Sum(p => p.PrecoFinal()):C}");

abstract class Produto(string titulo, decimal precoBase)
{
    public string Titulo { get; } = titulo;
    public decimal PrecoBase { get; } = precoBase;
    public virtual decimal PrecoFinal() => PrecoBase;
    public override string ToString() => $"{GetType().Name,-12} {Titulo,-13}
{PrecoFinal(),7:F2}";
}

class LivroFisico(string titulo, decimal preco) : Produto(titulo, preco)
{
    public override decimal PrecoFinal() => base.PrecoFinal() + 5m;
}

class Ebook(string titulo, decimal preco) : Produto(titulo, preco)
{
    public override decimal PrecoFinal() => PrecoBase * 0.9m;
}
```

Saída

```
LivroFisico  Dom Casmurro    45.00
Ebook        Dom Casmurro    36.00
LivroFisico  Iracema         30.00
Total: $111.00
```

O `ToString` da base chama `PrecoFinal()`, que é virtual: cada objeto mostra o próprio preço, mesmo com o código escrito uma vez só. O formato `:C` usa a cultura atual do processo, que vem do sistema: aqui saiu `$111.00` porque a máquina em que o programa rodou está em inglês dos EUA (`LANG=en_US.UTF-8`). Num programa real, defina `CultureInfo.CurrentCulture` ou passe a cultura explicitamente, como no módulo 1.

Q21 Interfaces e polimorfismo sem herança

> *Defina `interface IDescontavel { decimal Desconto(decimal total); }` e três políticas: `SemDesconto`, `Percentual(decimal pct)` e `ProgressivoPorQuantidade` (5% a partir de 3 itens, 10% a partir de 5). Escreva um método que escolha a política mais vantajosa para o cliente entre várias.*

```
IDescontavel[] politicas = [new SemDesconto(), new Percentual(7), new
ProgressivoPorQuantidade(itens: 4)];
decimal total = 200m;

foreach (var p in politicas)
    Console.WriteLine($"{p.GetType().Name,-25} desconto {p.Desconto(total),6:F2}");

var melhor = politicas.MaxBy(p => p.Desconto(total))!;
Console.WriteLine($"Melhor: {melhor.GetType().Name}, total a pagar {total -
melhor.Desconto(total):F2}");

interface IDescontavel
{
    decimal Desconto(decimal total);
}

class SemDesconto : IDescontavel
{
    public decimal Desconto(decimal total) => 0m;
}

class Percentual(decimal pct) : IDescontavel
{
    public decimal Desconto(decimal total) => total * pct / 100;
}

class ProgressivoPorQuantidade(int itens) : IDescontavel
{
    public decimal Desconto(decimal total) => itens switch
    {
        >= 5 => total * 0.10m,
        >= 3 => total * 0.05m,
        _ => 0m,
    };
}
```

Saída

```
SemDesconto               desconto   0.00
Percentual                desconto  14.00
ProgressivoPorQuantidade  desconto  10.00
Melhor: Percentual, total a pagar 186.00
```

Cada política é uma **estratégia** (o padrão Strategy) intercambiável: acrescentar uma "Black Friday" é escrever uma classe nova, sem tocar no código que escolhe a melhor. O `MaxBy` (.NET 6) devolve o elemento com a maior chave, e não a chave em si.

Q22 Records e igualdade

Escreva `record ItemDoPedido(string Titulo, int Quantidade, decimal PrecoUnitario)` com uma propriedade calculada `Subtotal`. Mostre: a igualdade por valor entre dois itens iguais; o `ToString` gerado; uma cópia com `with` mudando a quantidade; e um `HashSet` que elimina itens repetidos. Compare com uma `class` comum de mesmos campos.

```
var a = new ItemDoPedido("Iracema", 2, 25m);
var b = new ItemDoPedido("Iracema", 2, 25m);
var c = a with { Quantidade = 3 };

Console.WriteLine(a);
Console.WriteLine($"a == b? {a == b} | mesma referência? {ReferenceEquals(a, b)}");
Console.WriteLine($"c: {c.Quantidade} un., subtotal {c.Subtotal}");
Console.WriteLine($"HashSet com a, b e c: {new HashSet<ItemDoPedido> { a, b, c }.Count} itens");

var x = new ItemClasse("Iracema", 2, 25m);
var y = new ItemClasse("Iracema", 2, 25m);
Console.WriteLine($"class: x == y? {x == y}");

record ItemDoPedido(string Titulo, int Quantidade, decimal PrecoUnitario)
{
    public decimal Subtotal => Quantidade * PrecoUnitario;
}

class ItemClasse(string titulo, int quantidade, decimal precoUnitario)
{
    public string Titulo { get; } = titulo;
    public int Quantidade { get; } = quantidade;
    public decimal PrecoUnitario { get; } = precoUnitario;
}
```

Saída

```
ItemDoPedido { Titulo = Iracema, Quantidade = 2, PrecoUnitario = 25, Subtotal = 50 }
a == b? True | mesma referência? False
c: 3 un., subtotal 75
HashSet com a, b e c: 2 itens
class: x == y? False
```

O `record` gera `Equals`, `GetHashCode`, `==`, `!=`, `ToString` e o suporte a `with`, todos baseados nas propriedades posicionais. A `class` compara **referências**: dois objetos com os mesmos dados são diferentes. O `ToString` gerado mostra todas as propriedades públicas, inclusive a calculada `Subtotal`; para mudar isso, sobrescreva o `PrintMembers`. A igualdade, por outro lado, usa só os campos: o `Subtotal` não guarda nada.

Q23 Enum e estados do pedido

> *Modele `enum StatusPedido { Novo, Pago, Enviado, Entregue, Cancelado }` e escreva `static bool PodeIr(StatusPedido de, StatusPedido para)` com uma expressão `switch` sobre a tupla: só é permitido Novo → Pago, Pago → Enviado, Enviado → Entregue e cancelar a partir de Novo ou Pago. Teste todas as transições a partir de `Pago`. Mostre também o que `Enum.IsDefined` evita.*

```
static bool PodeIr(StatusPedido de, StatusPedido para) => (de, para) switch
{
    (StatusPedido.Novo, StatusPedido.Pago) => true,
    (StatusPedido.Pago, StatusPedido.Enviado) => true,
    (StatusPedido.Enviado, StatusPedido.Entregue) => true,
    (StatusPedido.Novo or StatusPedido.Pago, StatusPedido.Cancelado) => true,
    _ => false,
};

foreach (var destino in Enum.GetValues<StatusPedido>())
    Console.WriteLine($"Pago -> {destino,-9}: {(PodeIr(StatusPedido.Pago, destino) ? "permitido"
: "negado")}");

var vindoDoBanco = (StatusPedido)42;  // compila: um enum é só um int com nomes
Console.WriteLine($"{vindoDoBanco} é válido? {Enum.IsDefined(vindoDoBanco)}");

enum StatusPedido { Novo, Pago, Enviado, Entregue, Cancelado }
```

Saída

```
Pago -> Novo     : negado
Pago -> Pago     : negado
Pago -> Enviado  : permitido
Pago -> Entregue : negado
Pago -> Cancelado: permitido
42 é válido? False
```

O *cast* `(StatusPedido)42` mostra a fragilidade dos enums: qualquer inteiro cabe neles, e o `ToString` devolve "42". Valores vindos de banco, JSON ou formulário precisam ser validados com `Enum.IsDefined` (ou com `Enum.TryParse` seguido dele). As regras de transição concentradas numa função pura podem ser testadas sem banco nenhum.

MÓDULO 3

Genéricos, delegates e LINQ

Genéricos, delegates, lambdas e eventos, LINQ (filtrar, agrupar, cruzar e a execução adiada), nullable reference types e sequências com yield.

Artigos 16 a 21

#16 Genéricos: Um Código, Muitos Tipos, Nenhuma Conversão
#17 Delegates, Lambdas e Eventos: Passando Comportamento Como Se Fosse Dado
#18 LINQ, Parte I: Perguntas ao Catálogo em Vez de Laços
#19 LINQ, Parte II: Agrupar, Cruzar e Entender Quando a Consulta Executa
#20 Nullable Reference Types: o Fim Anunciado do NullReferenceException
#21 Sequências Sob Demanda: IEnumerable, yield e Coleções Imutáveis

Perguntas conceituais — respostas

Q24 Por que genéricos

> *Antes dos genéricos (C# 2), a coleção padrão era o `ArrayList`, que guarda `object`. Cite dois problemas que a `List<int>` resolve em relação a ele.*

- **Segurança de tipo**: no `ArrayList` cabe qualquer coisa; um `string` inserido por engano numa lista de números só é descoberto em execução, no *cast* `(int)lista[i]`, que lança `InvalidCastException`. Na `List<int>`, é erro de compilação.
- **Desempenho**: guardar um `int` como `object` exige *boxing* (uma alocação no *heap* por número) e *unboxing* na leitura. A `List<int>` guarda os `int` direto num array interno, sem alocação por elemento.

Ao contrário dos *templates* do C++, os genéricos do .NET existem em tempo de execução: o JIT gera código especializado para tipos de valor e compartilha uma única versão entre os tipos de referência.

Q25 Func, Action e Predicate

> *O que representam `Func<int, int, int>`, `Action<string>` e `Func<Livro, bool>`? Escreva uma lambda compatível com cada um.*

- `Func<int, int, int>`: recebe dois `int` e devolve um `int` (o último parâmetro de tipo é o retorno). Ex.: `(a, b) => a + b`.
- `Action<string>`: recebe uma `string` e não devolve nada. Ex.: `msg => Console.WriteLine(msg)`.
- `Func<Livro, bool>`: um predicado sobre livros, usado no `Where`. Ex.: `l => l.Preco < 50m`. O `Predicate<Livro>` tem a mesma assinatura, mas é outro tipo e não converte implicitamente; por isso o LINQ usa `Func`.

Q26 Execução adiada

Por que este código imprime 3, e não 2? O que muda com `.ToList()` no fim da primeira linha?

```
var lista = new List<int> { 1, 2, 3, 4 };
var grandes = lista.Where(x => x > 2);
lista.Add(5);
Console.WriteLine(grandes.Count());
```

O `Where` não filtra nada na hora: devolve um objeto que **descreve** a consulta. Ela só executa quando alguém percorre o resultado, aqui no `Count()`, e nesse momento a lista já tem o 5: os maiores que 2 são 3, 4 e 5. Com `.ToList()` no fim da primeira linha, a consulta executa imediatamente e o resultado é **materializado** numa lista nova; o `Add` posterior não a afeta, e o programa imprime 2. A execução adiada também significa que percorrer `grandes` duas vezes executa o filtro duas vezes, o que importa quando a fonte é um banco de dados.

Q27 Nullable reference types

Com `<Nullable>enable</Nullable>`, qual a diferença entre `string` e `string?`? O que o operador `!` (null-forgiving) faz, e por que usá-lo com moderação?

Com o recurso ativo, `string` declara que a variável **nunca** deve ser nula e `string?` que pode ser. O compilador faz uma análise de fluxo e avisa quando um `string?` é usado sem verificação, ou quando `null` é atribuído a um `string`. Os avisos não mudam nada em execução: os dois tipos são a mesma `string`. O `!` diz ao compilador "confie em mim, aqui não é nulo" e silencia o aviso, **sem nenhuma verificação real**. Se a afirmação estiver errada, a `NullReferenceException` volta. Use-o só quando você sabe algo que o compilador não consegue deduzir, e prefira reescrever o código para que ele deduza.

Exercícios de código — respostas

Q28 Um repositório genérico

Escreva `interface IEntidade { int Id { get; } }` e `class RepositorioEmMemoria<T> where T : IEntidade` com `Adicionar`, `ObterPorId` (devolve `T?`) e `Todos()` (devolve `IReadOnlyList<T>`). Recuse IDs repetidos. Use-o com `Livro` e com `Cliente`.

```
var livros = new RepositorioEmMemoria<Livro>();
livros.Adicionar(new Livro(1, "Vidas Secas"));
livros.Adicionar(new Livro(2, "Iracema"));
try { livros.Adicionar(new Livro(1, "Repetido")); }
catch (InvalidOperationException e) { Console.WriteLine(e.Message); }

Console.WriteLine(livros.ObterPorId(2)?.Titulo ?? "não achou");
Console.WriteLine(livros.ObterPorId(9)?.Titulo ?? "não achou");

var clientes = new RepositorioEmMemoria<Cliente>();
clientes.Adicionar(new Cliente(7, "Ana"));
Console.WriteLine($"{livros.Todos().Count} livros, {clientes.Todos().Count} cliente(s)");

interface IEntidade
{
    int Id { get; }
}

record Livro(int Id, string Titulo) : IEntidade;
record Cliente(int Id, string Nome) : IEntidade;

class RepositorioEmMemoria<T> where T : class, IEntidade
{
    private readonly Dictionary<int, T> _itens = [];

    public void Adicionar(T item)
    {
        if (!_itens.TryAdd(item.Id, item))
            throw new InvalidOperationException($"{typeof(T).Name} {item.Id} já existe");
    }

    public T? ObterPorId(int id) => _itens.GetValueOrDefault(id);
    public IReadOnlyList<T> Todos() => [.. _itens.Values];
}
```

Saída

```
Livro 1 já existe
Iracema
não achou
2 livros, 1 cliente(s)
```

A restrição `where T : class, IEntidade` permite usar `item.Id` dentro da classe e faz o `T?` significar "referência que pode ser nula". Sem o `class`, para um `T` que fosse `struct`, o `T?` não viraria `Nullable<T>`, e o `GetValueOrDefault` devolveria um *struct* zerado em vez de `null`. O `Todos()` devolve uma **cópia** (a expressão de coleção `[.. ]`): quem recebe não consegue alterar o dicionário interno.

Q29 Eventos de estoque

Escreva `class Estoque` com o evento `public event EventHandler<EstoqueBaixoEventArgs>? EstoqueBaixo`, disparado quando uma retirada deixa um título abaixo de 2 unidades. Inscreva dois ouvintes (um que "envia e-mail" e um que registra em log), retire do estoque e mostre a ordem das notificações. Depois cancele uma das inscrições.

```
var estoque = new Estoque();
EventHandler<EstoqueBaixoEventArgs> email = (_, e) => Console.WriteLine($"  [e-mail] repor
{e.Titulo} (restam {e.Restante})");
estoque.EstoqueBaixo += email;
estoque.EstoqueBaixo += (_, e) => Console.WriteLine($"  [log] {e.Titulo}={e.Restante}");

estoque.Entrar("Iracema", 3);
Console.WriteLine("retira 1");
estoque.Retirar("Iracema", 1);
Console.WriteLine("retira 1");
estoque.Retirar("Iracema", 1);
estoque.EstoqueBaixo -= email;
Console.WriteLine("retira 1 (sem o e-mail)");
estoque.Retirar("Iracema", 1);

class EstoqueBaixoEventArgs(string titulo, int restante) : EventArgs
{
    public string Titulo { get; } = titulo;
    public int Restante { get; } = restante;
}

class Estoque
{
    private readonly Dictionary<string, int> _qtd = [];
    public event EventHandler<EstoqueBaixoEventArgs>? EstoqueBaixo;

    public void Entrar(string titulo, int n) => _qtd[titulo] = _qtd.GetValueOrDefault(titulo) +
n;

    public void Retirar(string titulo, int n)
    {
        _qtd[titulo] -= n;
        if (_qtd[titulo] < 2)
            EstoqueBaixo?.Invoke(this, new EstoqueBaixoEventArgs(titulo, _qtd[titulo]));
    }
}
```

Saída

```
retira 1
retira 1
  [e-mail] repor Iracema (restam 1)
  [log] Iracema=1
retira 1 (sem o e-mail)
  [log] Iracema=0
```

Os ouvintes são chamados **na ordem de inscrição**, na mesma thread de quem disparou. O `?.Invoke` cuida do caso sem nenhum ouvinte. A palavra `event` impede que código de fora faça `estoque.EstoqueBaixo = null` ou dispare o evento: de fora só são permitidos `+=` e `-=`. Um ouvinte esquecido mantém o objeto inscrito vivo, uma causa clássica de vazamento de memória em .NET.

Q30 Consultas LINQ no catálogo

> *Dado um catálogo de livros (título, autor, ano, preço, gênero), escreva consultas LINQ para: (a) os títulos de romance anteriores a 1950, em ordem de ano; (b) o preço médio por gênero; (c) o autor com mais livros; (d) se existe algum livro acima de R$ 100; (e) as páginas de 2 livros, ordenadas por título, mostrando a página 2.*

```
Livro[] catalogo =
[
    new("Dom Casmurro", "Machado de Assis", 1899, 35m, "romance"),
    new("Memórias Póstumas", "Machado de Assis", 1881, 32m, "romance"),
    new("Vidas Secas", "Graciliano Ramos", 1938, 49.9m, "romance"),
    new("São Bernardo", "Graciliano Ramos", 1934, 44m, "romance"),
    new("Quincas Borba", "Machado de Assis", 1891, 38m, "romance"),
    new("A Rosa do Povo", "Carlos Drummond", 1945, 59m, "poesia"),
    new("Claro Enigma", "Carlos Drummond", 1951, 52m, "poesia"),
];

var antigos = catalogo.Where(l => l.Genero == "romance" && l.Ano < 1950).OrderBy(l =>
l.Ano).Select(l => l.Titulo);
Console.WriteLine("(a) " + string.Join(", ", antigos));

foreach (var g in catalogo.GroupBy(l => l.Genero))
    Console.WriteLine($"(b) {g.Key}: {g.Average(l => l.Preco):F2}");

var autor = catalogo.CountBy(l => l.Autor).MaxBy(p => p.Value);
Console.WriteLine($"(c) {autor.Key} ({autor.Value} livros)");

Console.WriteLine($"(d) {catalogo.Any(l => l.Preco > 100m)}");

int tamanhoPagina = 2, pagina = 2;
var pag = catalogo.OrderBy(l => l.Titulo).Skip((pagina - 1) * tamanhoPagina).Take(tamanhoPagina);
Console.WriteLine("(e) " + string.Join(", ", pag.Select(l => l.Titulo)));

record Livro(string Titulo, string Autor, int Ano, decimal Preco, string Genero);
```

Saída

```
(a) Memórias Póstumas, Quincas Borba, Dom Casmurro, São Bernardo, Vidas Secas
(b) romance: 39.78
(b) poesia: 55.50
(c) Machado de Assis (3 livros)
(d) False
(e) Dom Casmurro, Memórias Póstumas
```

O `CountBy` (.NET 9) conta por chave sem montar os grupos; antes dele, o caminho era `GroupBy(l => l.Autor).MaxBy(g => g.Count())`. A paginação com `Skip` e `Take` é exatamente o que o EF Core traduz para

`OFFSET` e `LIMIT` no SQL, desde que haja um `OrderBy` antes: sem ordem, "a página 2" não tem significado.

Q31 Juntando pedidos e clientes

Com uma lista de clientes (id, nome, cidade) e uma de pedidos (id, idCliente, valor), use `Join` para listar pedido, cliente e valor; use `GroupJoin` para mostrar o total gasto por cliente, incluindo quem não comprou nada (total 0).

```
Cliente[] clientes = [new(1, "Ana", "Recife"), new(2, "Bruno", "Natal"), new(3, "Carla",
"Recife")];
Pedido[] pedidos = [new(10, 1, 120m), new(11, 2, 35m), new(12, 1, 80m)];

var linhas = pedidos.Join(clientes, p => p.ClienteId, c => c.Id, (p, c) => $"pedido {p.Id}:
{c.Nome,-5} {p.Valor,7:F2}");
foreach (var l in linhas)
    Console.WriteLine(l);

var totais = clientes.GroupJoin(pedidos, c => c.Id, p => p.ClienteId,
    (c, ps) => new { c.Nome, Total = ps.Sum(p => p.Valor), Qtd = ps.Count() });
foreach (var t in totais)
    Console.WriteLine($"{t.Nome,-5} {t.Qtd} pedido(s), total {t.Total,7:F2}");

record Cliente(int Id, string Nome, string Cidade);
record Pedido(int Id, int ClienteId, decimal Valor);
```

Saída

```
pedido 10: Ana    120.00
pedido 11: Bruno   35.00
pedido 12: Ana     80.00
Ana   2 pedido(s), total  200.00
Bruno 1 pedido(s), total   35.00
Carla 0 pedido(s), total    0.00
```

O `Join` é um *inner join*: Carla, sem pedidos, não aparece na primeira lista. O `GroupJoin` entrega, para cada cliente, a sequência (possivelmente vazia) de pedidos, o equivalente a um `LEFT JOIN` seguido de `GROUP BY`. Os dois montam uma tabela hash com a sequência interna: são O(n + m), e não O(n × m) como dois laços aninhados. O .NET 10 também traz `LeftJoin`, que dispensa o truque.

Q32 Sequências com yield

Escreva `static IEnumerable<long> Fibonacci()`, infinita, com `yield return`, e use LINQ para pegar os 10 primeiros, o primeiro acima de 1000 e a soma dos pares menores que 4 milhões. Depois escreva `static IEnumerable<string> LerLinhas(string caminho)` que leia um arquivo de forma preguiçosa e explique por que a sequência infinita não trava o programa.

```
static IEnumerable<long> Fibonacci()
{
    long a = 0, b = 1;
    while (true)
    {
        yield return a;
        (a, b) = (b, a + b);
    }
}

static IEnumerable<string> LerLinhas(string caminho)
{
    using var leitor = new StreamReader(caminho);  // fechado quando a iteração termina
    while (leitor.ReadLine() is { } linha)
        yield return linha;
}

Console.WriteLine(string.Join(" ", Fibonacci().Take(10)));
Console.WriteLine(Fibonacci().First(x => x > 1000));
Console.WriteLine(Fibonacci().TakeWhile(x => x < 4_000_000).Where(x => x % 2 == 0).Sum());

File.WriteAllLines("dados.txt", ["primeira", "", "terceira"]);
foreach (var l in LerLinhas("dados.txt").Where(l => l.Length > 0))
    Console.WriteLine($"> {l}");
```

Saída

```
0 1 1 2 3 5 8 13 21 34
1597
4613732
> primeira
> terceira
```

O compilador transforma o método com `yield` numa **máquina de estados**: cada `MoveNext()` executa até o próximo `yield return` e pausa ali. Nada é calculado além do que é pedido, então `Take(10)` para depois de dez elementos. O `using` dentro do iterador só fecha o arquivo quando o `foreach` termina (ou é interrompido por um `break`), porque o `foreach` chama `Dispose` no enumerador. O mesmo vale para `File.ReadLines`, que já faz isso pronto.

MÓDULO 4

A plataforma por dentro

Reflexão e atributos, source generators, memória e coletor de lixo, Span<T>, arquivos e JSON, async/await, concorrência e interop com C.

Artigos 22 a 30

#22 Reflexão e Atributos: Quando o Programa Examina a Si Mesmo
#23 Source Generators: Código Que Escreve Código Durante a Compilação
#24 Memória no .NET: Pilha, Heap, Coletor de Lixo e IDisposable
#25 Span<T> e Memory<T>: Olhar Sem Copiar, Processar Sem Alocar
#26 Arquivos, Streams e JSON: Dados Que Sobrevivem ao Programa
#27 async e await: Esperar Sem Bloquear
#28 Assincronia Avançada: Muitas Tarefas, Cancelamento e Fluxos Assíncronos
#29 Concorrência: Threads, Travas e Dados Compartilhados Sem Corrupção
#30 Interop com C: Chamando Código Nativo com P/Invoke e unsafe

Perguntas conceituais — respostas

Q33 Reflexão ou source generator

> *Um serializador pode descobrir as propriedades de um tipo por **reflexão**, em execução, ou gerá-las com um **source generator**, na compilação. Compare as duas abordagens em desempenho, em compatibilidade com Native AOT e em facilidade de depuração.*

- **Reflexão**: lê os metadados em execução (`typeof(T).GetProperties()`). É flexível e simples de escrever, mas é lenta (cada acesso passa por camadas de verificação), aumenta o tempo de partida e conflita com o *trimming* e o Native AOT, que removem o que parece não ser usado e não conseguem prever o que a reflexão vai pedir.
- **Source generator**: roda dentro do compilador e acrescenta código C# comum ao projeto. Em execução, é uma chamada direta, sem reflexão; funciona com AOT; e o código gerado pode ser lido e depurado como qualquer outro. O custo fica na compilação e na complexidade de escrever o gerador.

O `System.Text.Json` oferece os dois caminhos: reflexão por padrão e geração com `[JsonSerializable]` num `JsonSerializerContext`.

Q34 Gerações do coletor de lixo

> *Explique as gerações 0, 1 e 2 do coletor de lixo do .NET e a hipótese em que elas se apoiam. O que é o LOH?*

A hipótese geracional diz que **a maioria dos objetos morre jovem**. Objetos novos nascem na geração 0, que é pequena e coletada com frequência e rapidez. Quem sobrevive a uma coleta sobe para a geração 1 (um "amortecedor") e, sobrevivendo de novo, para a 2, coletada raramente, porque objetos antigos tendem a continuar vivos. Assim, a maioria das coletas examina só uma pequena parte do *heap*. O **LOH** (*Large Object Heap*) guarda objetos a partir de 85.000 bytes, como arrays grandes: eles já nascem lógicamente na geração

2 e, por padrão, não são compactados, porque mover blocos grandes é caro. Alocar e descartar muitos arrays grandes fragmenta o LOH, e é aí que entram o `ArrayPool<T>` e o `Span<T>`.

Q35 IDisposable e using

> *Se o .NET tem coletor de lixo, por que existe `IDisposable`? O que `using var arquivo = File.OpenRead(...)` garante, e o que é um finalizer?*

O coletor cuida de **memória gerenciada**, e só quando quiser. Arquivos, *sockets*, conexões de banco e travas são **recursos não gerenciados** ou escassos: precisam ser liberados assim que deixam de ser usados, e não "algum dia". O `IDisposable.Dispose()` é a liberação explícita, e o `using` garante a chamada no fim do escopo, mesmo com exceção (é o RAII do C#, mas opcional). O *finalizer* (`~Classe()`) é uma rede de segurança chamada pelo coletor antes de recuperar um objeto que não foi descartado; é lento, não tem hora certa e atrasa a coleta. O padrão é: `Dispose` libera e chama `GC.SuppressFinalize(this)`; o *finalizer*, quando existe, só cobre o esquecimento.

Q36 async não é thread

> *"Um método `async` roda em outra thread." Por que a frase está errada? O que acontece de fato num `await File.ReadAllTextAsync(...)` quando o arquivo ainda não foi lido?*

O `async` não cria thread nenhuma: ele transforma o método numa **máquina de estados** que pode ser pausada. O método roda na thread de quem chamou até o primeiro `await` de uma operação que ainda não terminou. Nesse ponto, ele registra a **continuação** (o resto do método) e devolve uma `Task` incompleta para quem chamou: a thread fica livre para outras tarefas. A leitura acontece no sistema operacional, sem thread nenhuma esperando bloqueada. Quando o SO avisa que terminou, a continuação é agendada (no *thread pool*, num servidor; na thread da interface, num app gráfico) e o método segue de onde parou. É assim que um servidor atende milhares de requisições com poucas dezenas de threads.

Q37 Task.Run e async void

> *Por que `Task.Run` não deve envolver chamadas de E/S num servidor ASP.NET Core, e por que `async void` só é aceitável em handlers de evento?*

O `Task.Run` empurra um trabalho para outra thread do *thread pool*. Para trabalho **de CPU** num app de interface, isso libera a thread da tela. Mas num servidor, onde a requisição já roda no *thread pool*, envolver E/S em `Task.Run` só troca uma thread por outra e acrescenta custo; o certo é chamar a versão assíncrona de verdade (`ReadAsync`, `QueryAsync`). Já o `async void` não devolve `Task`: quem chama não pode aguardar o fim, e uma exceção lançada dentro dele não tem para onde ir e derruba o processo. Existe porque *handlers* de evento precisam devolver `void`; em qualquer outro lugar, `async Task`.

Q38 lock e o que ele protege

> *O que `lock (_trava) { ... }` garante, e por que a trava deve ser um objeto privado, e não `this` ou um `string`? O que o .NET 9 trouxe para isso?*

O `lock` garante **exclusão mútua**: só uma thread por vez executa os blocos protegidos pela mesma trava, e as escritas feitas dentro dele ficam visíveis para a próxima thread que entrar. Travar `this` ou um `typeof(...)` expõe a trava a qualquer código de fora, que pode travar o mesmo objeto e causar *deadlocks* difíceis de achar; uma `string` literal é compartilhada pelo processo inteiro (*interning*). Um campo `private readonly` garante que só a própria classe usa aquela trava. O .NET 9 trouxe o tipo `System.Threading.Lock`, feito só para isso: `private readonly Lock _trava = new();` funciona com o mesmo `lock` e é mais eficiente que travar um `object`.

Exercícios de código — respostas

Q39 Atributos e reflexão

> *Crie o atributo `[Obrigatorio]` e um validador genérico que percorra, por reflexão, as propriedades `string` de qualquer objeto e liste as marcadas com o atributo que estejam vazias. Teste com uma classe `CadastroCliente`.*

```
#:property PublishAot=false
using System.Reflection;

var c = new CadastroCliente { Nome = "Ana", Email = "", Apelido = "" };
foreach (var erro in Validador.Validar(c))
    Console.WriteLine(erro);

static class Validador
{
    public static IEnumerable<string> Validar(object obj)
    {
        foreach (var p in obj.GetType().GetProperties(BindingFlags.Public |
BindingFlags.Instance))
        {
            if (p.PropertyType != typeof(string) || p.GetCustomAttribute<ObrigatorioAttribute>()
is null)
                continue;
            if (string.IsNullOrWhiteSpace((string?)p.GetValue(obj)))
                yield return $"{p.Name} é obrigatório";
        }
    }
}

[AttributeUsage(AttributeTargets.Property)]
class ObrigatorioAttribute : Attribute;

class CadastroCliente
{
    [Obrigatorio] public string Nome { get; set; } = "";
    [Obrigatorio] public string Email { get; set; } = "";
    [Obrigatorio] public string Cpf { get; set; } = "";
    public string Apelido { get; set; } = "";  // opcional
}
```

Saída

```
Email é obrigatório
Cpf é obrigatório
```

A primeira linha é uma diretiva dos programas de arquivo único do .NET 10 (`dotnet run arquivo.cs`): eles são compilados, por padrão, prontos para Native AOT, o que liga os avisos de *trimming* e desliga recursos que dependem de reflexão em execução. O `PublishAot=false` volta ao comportamento de um projeto `.csproj` comum, onde a linha não é necessária.

O atributo, sozinho, não faz nada: é só um metadado gravado no *assembly*. Quem lhe dá sentido é o código que o lê. É o mesmo mecanismo do `[Required]` do ASP.NET Core e do `[Key]` do EF Core. A ordem das propriedades devolvidas por `GetProperties` não é garantida pela especificação, embora na prática siga a declaração.

Q40 Span sem alocar

Escreva `static int SomarCsv(ReadOnlySpan<char> linha)` que some os inteiros de uma linha como `"10;20;x;30"`, ignorando os campos inválidos, sem criar nenhuma `string` (nada de `Split`). Compare a quantidade de bytes alocados com a versão que usa `Split` e `int.Parse`, medindo com `GC.GetAllocatedBytesForCurrentThread()`.

```
static int SomarCsv(ReadOnlySpan<char> linha)
{
    int soma = 0;
    foreach (var faixa in linha.Split(';'))  // .NET 9: Split de span devolve faixas, não strings
        if (int.TryParse(linha[faixa], out int n))
            soma += n;
    return soma;
}

static int SomarComSplit(string linha)
{
    int soma = 0;
    foreach (var campo in linha.Split(';'))
        if (int.TryParse(campo, out int n))
            soma += n;
    return soma;
}

string linha = "10;20;x;30;40;50";
SomarCsv(linha);  // aquecimento: a primeira chamada compila o método (JIT)
SomarComSplit(linha);

long antes = GC.GetAllocatedBytesForCurrentThread();
int a = 0;
for (int i = 0; i < 1000; i++) a = SomarCsv(linha);
long comSpan = GC.GetAllocatedBytesForCurrentThread() - antes;

antes = GC.GetAllocatedBytesForCurrentThread();
int b = 0;
for (int i = 0; i < 1000; i++) b = SomarComSplit(linha);
long comSplit = GC.GetAllocatedBytesForCurrentThread() - antes;

Console.WriteLine($"span:  soma {a}, {comSpan} bytes em 1000 chamadas");
Console.WriteLine($"split: soma {b}, {(comSplit > 100_000 ? "mais de 100 KB" : $"{comSplit}
bytes")} em 1000 chamadas");
```

Saída

```
span:  soma 150, 0 bytes em 1000 chamadas
split: soma 150, mais de 100 KB em 1000 chamadas
```

O `Split` cria um array e uma `string` para cada campo, em toda chamada; a versão com `Span` só olha para pedaços da string original e não aloca nada. O `ReadOnlySpan<char>` é uma `ref struct`: vive só na pilha, não pode ser campo de classe nem atravessar um `await`. Quando for preciso guardar ou atravessar `await`, o equivalente é o `ReadOnlyMemory<char>`.

Q41 Gravando e lendo JSON

> *Serialize uma lista de `record Livro(string Titulo, decimal Preco, DateOnly Lancamento)` para um arquivo JSON com nomes em camelCase e indentação, leia o arquivo de volta e confira que as listas são iguais. Mostre o JSON gravado.*

```
#:property PublishAot=false
using System.Text.Json;

var opcoes = new JsonSerializerOptions(JsonSerializerDefaults.Web) { WriteIndented = true };
List<Livro> livros = [new("Vidas Secas", 49.9m, new DateOnly(1938, 3, 1)), new("Memórias
Póstumas", 32m, new DateOnly(1881, 1, 1))];

await using (var saida = File.Create("livros.json"))
    await JsonSerializer.SerializeAsync(saida, livros, opcoes);

Console.WriteLine(await File.ReadAllTextAsync("livros.json"));

await using var entrada = File.OpenRead("livros.json");
var lidos = await JsonSerializer.DeserializeAsync<List<Livro>>(entrada, opcoes) ?? [];
Console.WriteLine($"iguais? {lidos.SequenceEqual(livros)}");

record Livro(string Titulo, decimal Preco, DateOnly Lancamento);
```

Saída

```
[
  {
    "titulo": "Vidas Secas",
    "preco": 49.9,
    "lancamento": "1938-03-01"
  },
  {
    "titulo": "Mem\u00F3rias P\u00F3stumas",
    "preco": 32,
    "lancamento": "1881-01-01"
  }
]
iguais? True
```

O `JsonSerializerDefaults.Web` liga o *camelCase* e a leitura sem diferenciar maiúsculas, o padrão das APIs web. O `decimal` sai como número JSON sem perder casas, e o `DateOnly` como texto ISO 8601. O `SequenceEqual` funciona porque `record` compara por valor. O `\u00F3` aparece no lugar do "ó" porque, por segurança, o codificador padrão escapa tudo que não é ASCII; `Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping` grava os acentos como estão.

Q42 Várias tarefas com cancelamento

> *Simule a consulta de preço de um livro em três fornecedores com tempos de resposta diferentes (`Task.Delay` de 300, 100 e 2000 ms). Use `Task.WhenAll` para esperar os que respondem em até 500 ms, cancelando o lento com um `CancellationTokenSource` com tempo limite, e mostre o menor preço entre os que responderam.*

```
using System.Diagnostics;

async Task<decimal?> ConsultarAsync(string fornecedor, int ms, decimal preco, CancellationToken
ct)
{
    try
    {
        await Task.Delay(ms, ct);
        Console.WriteLine($"  {fornecedor} respondeu");
        return preco;
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine($"  {fornecedor} cancelado");
        return null;
    }
}

var relogio = Stopwatch.StartNew();
using var cts = new CancellationTokenSource(TimeSpan.FromMilliseconds(500));
var precos = await Task.WhenAll(
    ConsultarAsync("Fornecedor A", 300, 47.5m, cts.Token),
    ConsultarAsync("Fornecedor B", 100, 49.9m, cts.Token),
    ConsultarAsync("Fornecedor C", 2000, 39.0m, cts.Token));

var validos = precos.OfType<decimal>().ToList();
Console.WriteLine($"menor preço: {validos.Min()} entre {validos.Count} respostas");
Console.WriteLine($"tempo total abaixo de 1 s? {relogio.ElapsedMilliseconds < 1000}");
```

Saída

```
  Fornecedor B respondeu
  Fornecedor A respondeu
  Fornecedor C cancelado
menor preço: 47.5 entre 2 respostas
tempo total abaixo de 1 s? True
```

As três consultas rodam **ao mesmo tempo**: o tempo total é o da mais lenta permitida (cerca de 500 ms), e não a soma (2,4 s). O cancelamento é **cooperativo**: o `CancellationToken` só tem efeito porque o `Task.Delay` o observa; um laço de CPU precisaria chamar `ct.ThrowIfCancellationRequested()` de tempos em tempos. Fornecedor C, mais barato, ficou de fora: o tempo limite é uma decisão de negócio, não só técnica.

Q43 Contando em paralelo sem corrupção

Conte as palavras de 4 textos em paralelo com `Parallel.ForEach`, acumulando num `Dictionary<string, int>` comum protegido por `lock` e, em outra versão, num `ConcurrentDictionary<string, int>` com `AddOrUpdate`. Confira que os dois dão o mesmo total.

```
using System.Collections.Concurrent;

string[] textos =
[
    "o rato roeu a roupa do rei",
    "o rei riu do rato",
    "a roupa do rei rasgou",
    "o rato fugiu",
];

var comLock = new Dictionary<string, int>();
var trava = new Lock();
Parallel.ForEach(textos, t =>
{
    foreach (var p in t.Split(' '))
        lock (trava)
            comLock[p] = comLock.GetValueOrDefault(p) + 1;
});

var concorrente = new ConcurrentDictionary<string, int>();
Parallel.ForEach(textos, t =>
{
    foreach (var p in t.Split(' '))
        concorrente.AddOrUpdate(p, 1, (_, atual) => atual + 1);
});

Console.WriteLine($"total com lock: {comLock.Values.Sum()} | concorrente:
{concorrente.Values.Sum()}");
foreach (var (p, n) in concorrente.OrderByDescending(x => x.Value).ThenBy(x => x.Key).Take(3))
    Console.WriteLine($"{p}: {n}");
```

Saída

```
total com lock: 20 | concorrente: 20
do: 3
o: 3
rato: 3
```

Sem o `lock`, o `Dictionary` comum pode perder contagens ou até corromper a estrutura interna e entrar em laço infinito: ele não é seguro para escritas concorrentes. No `ConcurrentDictionary`, a função de atualização pode ser chamada **mais de uma vez** sob disputa, por isso precisa ser pura (sem efeitos colaterais). Para quatro frases, o paralelismo custa mais do que economiza; ele só compensa com volumes grandes.

Q44 Chamando C com P/Invoke

Use `[LibraryImport]` para chamar a função `strlen` da biblioteca C do Linux (`libc`) e a função `abs`. Compare com `string.Length` para um texto com acentos e explique a diferença.

```
#:property AllowUnsafeBlocks=true
using System.Runtime.InteropServices;

string texto = "ação";
Console.WriteLine($"Length (UTF-16):  {texto.Length}");
Console.WriteLine($"strlen (UTF-8):   {Libc.strlen(texto)}");
Console.WriteLine($"abs(-42) pela libc: {Libc.abs(-42)}");

static partial class Libc
{
    [LibraryImport("libc", StringMarshalling = StringMarshalling.Utf8)]
    internal static partial nuint strlen(string s);

    [LibraryImport("libc")]
    internal static partial int abs(int x);
}
```

Saída

```
Length (UTF-16):  4
strlen (UTF-8):   6
abs(-42) pela libc: 42
```

O `Length` conta **unidades UTF-16** (4: a, ç, ã, o). Para chamar a função C, o *marshaller* converte a string para UTF-8 terminada em `'\0'`, e o `strlen` conta **bytes**: "ç" e "ã" ocupam dois bytes cada, então 6. O `[LibraryImport]` (.NET 7) gera o código de conversão na compilação, com um *source generator*, em vez de montá-lo em execução como o antigo `[DllImport]`; por isso a classe precisa ser `partial`; e o código gerado usa ponteiros, o que exige `AllowUnsafeBlocks` (num `.csproj`, `<AllowUnsafeBlocks>true</AllowUnsafeBlocks>`). O `nuint` corresponde ao `size_t` do C. O nome `libc` resolve no Linux; no Windows, o equivalente seria `msvcrt` ou `ucrtbase`.

MÓDULO 5

Testes e bancos de dados

xUnit, SQL com SQLite, ADO.NET, Dapper, Entity Framework Core, PostgreSQL, MySQL, MongoDB e um repositório independente do banco.

Artigos 31 a 42

#31 Testes Automatizados com xUnit: Mudar Sem Medo
#32 SQL Essencial com SQLite: Perguntas a um Banco de Dados
#33 ADO.NET: Conexões, Comandos Parametrizados e o Perigo do SQL Injection
#34 Dapper: SQL Seu, Sem a Cerimônia
#35 Entity Framework Core: Objetos no Banco Sem Escrever SQL
#36 EF Core: Relacionamentos e Migrations
#37 EF Core com Desempenho: N+1, AsNoTracking e o SQL por Trás de Cada Consulta
#38 PostgreSQL: o Banco de Dados Servidor, com Npgsql, JSONB e Arrays
#39 MySQL: MySqlConnector, o Provider do EF e as Diferenças Que Importam
#40 MongoDB: o Modelo de Documentos e o Driver Oficial
#41 MongoDB Avançado: Agregações, Índices e a Escolha Entre Documento e Relacional
#42 Um Repositório, Três Bancos: Injeção de Dependência e a Livraria Independente de Infraestrutura

Perguntas conceituais — respostas

Q45 Fact, Theory e o padrão AAA

No xUnit, qual a diferença entre `[Fact]` e `[Theory]` com `[InlineData]`? O que é o padrão Arrange, Act, Assert?

O `[Fact]` é um teste sem parâmetros, que verifica uma afirmação sempre verdadeira. A `[Theory]` é um teste **parametrizado**: o mesmo método roda uma vez para cada `[InlineData(...)]`, e cada execução aparece como um teste separado no relatório. É ideal para uma regra com vários casos (o frete de cada UF, os limites de um desconto). O padrão AAA organiza o corpo em três blocos: **Arrange** (montar os objetos e os dados), **Act** (executar a ação testada, de preferência numa linha só) e **Assert** (verificar o resultado). Um teste que tem vários "Act" geralmente são vários testes.

Q46 SQL injection

Mostre por que o código abaixo é vulnerável, o que um atacante digitaria e como o ADO.NET resolve o problema.

```
var cmd = conexao.CreateCommand();
cmd.CommandText = $"SELECT * FROM clientes WHERE email = '{email}'";
```

O valor digitado vira **parte do comando SQL**. Com o e-mail `' OR '1'='1`, a consulta fica `... WHERE email = '' OR '1'='1'`, que devolve todos os clientes. Com `'; DROP TABLE clientes; --`, conforme o banco, o atacante executa um comando novo. A correção é o **parâmetro**: `cmd.CommandText = "SELECT * FROM clientes WHERE email = $email"; cmd.Parameters.AddWithValue("$email", email);`. O texto do comando e o

valor viajam separados; o banco nunca interpreta o valor como SQL, qualquer que seja o conteúdo. A interpolação de string é o erro mais comum, justamente porque o `$"..."` parece inofensivo.

Q47 ADO.NET, Dapper ou EF Core

> *Compare ADO.NET puro, Dapper e EF Core quanto ao que você escreve, ao controle sobre o SQL e ao que cada um faz por você. Quando escolher cada um?*

- **ADO.NET**: você escreve o SQL e também o código de leitura linha a linha (`reader.GetString(0)`). Controle total e nenhuma dependência, mas muito código repetitivo.
- **Dapper**: você escreve o SQL; ele mapeia parâmetros e resultados para objetos (`conexao.Query<Livro>(sql, new { id })`). Quase a velocidade do ADO.NET, sem a cerimônia.
- **EF Core**: você escreve LINQ, e ele gera o SQL; acompanha as mudanças dos objetos (*change tracking*), gera *migrations* do esquema e cuida dos relacionamentos. Mais produtivo para CRUD, e o SQL fica menos visível.

Escolha EF Core para aplicações com muitas entidades e regras de negócio, Dapper para consultas de leitura sob medida ou de desempenho crítico, e ADO.NET quando não puder ter dependências. Elas convivem: é comum usar EF Core para escrever e Dapper para relatórios.

Q48 O problema N+1

> *O que é o problema N+1 no EF Core? Mostre o código que o causa e duas formas de corrigir.*

Ao listar N pedidos e acessar `pedido.Cliente.Nome` de cada um com *lazy loading*, o EF executa **uma consulta para os pedidos e mais uma para cada cliente**: N+1 idas ao banco, cada uma com a sua latência de rede. Com 500 pedidos, 501 consultas. Correções:

- **`Include`**: `db.Pedidos.Include(p => p.Cliente).ToList()` traz tudo num `JOIN`;
- **projeção**: `db.Pedidos.Select(p => new { p.Id, ClienteNome = p.Cliente.Nome }).ToList()` traz só as colunas usadas, também numa consulta.

Para leitura sem alteração posterior, `AsNoTracking()` ainda evita o custo do *change tracking*. O `ToQueryString()` ou o *log* do EF mostram o SQL realmente gerado.

Q49 Documento ou relacional

> *A livraria guarda pedidos com itens. Quando modelar no MongoDB, com os itens **embutidos** no documento do pedido, é melhor que no PostgreSQL, com tabelas `pedidos` e `itens`? E quando é pior?*

Melhor quando os dados são lidos e gravados juntos: o pedido é sempre exibido com os itens, e os itens não existem fora dele. Um documento traz tudo numa leitura só, sem *join*, e a estrutura pode variar entre pedidos (um item com dedicatória, outro com embrulho). **Pior** quando há relações entre entidades independentes e consultas transversais: "quantos exemplares de Vidas Secas vendemos este mês, por região?" exige agregações sobre todos os documentos, e o título de um livro repetido em milhares de pedidos precisa ser atualizado em todos. O relacional garante integridade (chaves estrangeiras, transações entre tabelas) e consultas *ad hoc*; o documento ganha em leitura do agregado e em flexibilidade de esquema.

Exercícios de código — respostas

Q50 Testes com xUnit

Escreva, como projeto de testes xUnit, os testes da regra de frete do módulo 1 (`Frete(uf, total)`): uma `[Theory]` cobrindo SP, RJ, outra UF e o limite de R$ 200 (exatamente 200 e 200,01), e um `[Fact]` que verifique que uma UF em minúsculas é tratada como as outras UFs (documentando o comportamento atual).

```
using Xunit;

public static class Regras
{
    public static decimal Frete(string uf, decimal total) => (uf, total) switch
    {
        (_, > 200m) => 0m,
        ("SP" or "RJ", _) => 10m,
        _ => 25m,
    };
}

public class FreteTests
{
    [Theory]
    [InlineData("SP", 50, 10)]
    [InlineData("RJ", 50, 10)]
    [InlineData("BA", 50, 25)]
    [InlineData("BA", 200, 25)]      // exatamente 200 ainda paga
    [InlineData("BA", 200.01, 0)]    // acima de 200 é grátis
    public void Frete_segue_a_tabela(string uf, decimal total, decimal esperado)
    {
        var frete = Regras.Frete(uf, total);

        Assert.Equal(esperado, frete);
    }

    [Fact]
    public void Uf_em_minusculas_paga_como_outras_ufs()
    {
        Assert.Equal(25m, Regras.Frete("sp", 50m));
    }
}
```

Crie o projeto com `dotnet new xunit -n Livraria.Testes` e rode com `dotnet test`: com o xUnit 2.9.3, os seis casos passam. O `[InlineData]` só aceita constantes, e `decimal` não é uma delas: o xUnit converte o `50` e o `200.01` (que chegam como `int` e `double`) para o tipo do parâmetro. O teste com "sp" documenta um comportamento provavelmente indesejado; quando a regra mudar para aceitar minúsculas, é ele que vai falhar e lembrar de atualizar a documentação. Os casos de limite (200 e 200,01) são os que mais acham erros: é ali que um `>` vira `>=` sem ninguém perceber.

Q51 ADO.NET com SQLite

Com o pacote `Microsoft.Data.Sqlite`, crie um banco em memória, a tabela `livros(id, titulo, preco)`, insira três livros com um comando parametrizado dentro de uma transação e liste os que custam menos de R$ 50, lendo com `SqliteDataReader`.

```
#:package Microsoft.Data.Sqlite@10.0.13
using Microsoft.Data.Sqlite;

using var conexao = new SqliteConnection("Data Source=:memory:");
conexao.Open();

using (var criar = conexao.CreateCommand())
{
    criar.CommandText = "CREATE TABLE livros (id INTEGER PRIMARY KEY, titulo TEXT NOT NULL, preco
REAL NOT NULL)";
    criar.ExecuteNonQuery();
}

using (var tx = conexao.BeginTransaction())
{
    var ins = conexao.CreateCommand();
    ins.Transaction = tx;
    ins.CommandText = "INSERT INTO livros (titulo, preco) VALUES ($titulo, $preco)";
    var pTitulo = ins.Parameters.Add("$titulo", SqliteType.Text);
    var pPreco = ins.Parameters.Add("$preco", SqliteType.Real);
    foreach (var (t, p) in new[] { ("Vidas Secas", 49.9), ("A Rosa do Povo", 59.0), ("Iracema",
25.0) })
    {
        pTitulo.Value = t;
        pPreco.Value = p;
        ins.ExecuteNonQuery();
    }
    tx.Commit();
}

using var sel = conexao.CreateCommand();
sel.CommandText = "SELECT id, titulo, preco FROM livros WHERE preco < $max ORDER BY preco";
sel.Parameters.AddWithValue("$max", 50);
using var r = sel.ExecuteReader();
while (r.Read())
    Console.WriteLine($"{r.GetInt32(0)} {r.GetString(1),-12} {r.GetDouble(2):F2}");
```

Saída

```
3 Iracema      25.00
1 Vidas Secas  49.90
```

A transação não é só por segurança: no SQLite, cada `INSERT` fora de uma transação explícita vira uma transação própria, com uma gravação em disco cada; com milhares de linhas, a diferença é de segundos para milissegundos. O mesmo comando é reaproveitado, trocando só os valores dos parâmetros. O SQLite

não tem tipo decimal (`REAL` é `double`): para dinheiro, guarde centavos num `INTEGER`. A linha `#:package` é a forma de um programa de arquivo único (.NET 10) declarar um pacote NuGet.

Q52 **EF Core em memória com relacionamentos**

> *Com o EF Core e o provedor SQLite em memória, modele `Autor` (um) e `Livro` (muitos), crie o banco com `EnsureCreated`, grave dois autores com livros e liste cada autor com a quantidade de livros e o mais caro, usando uma projeção. Diga como ver o SQL que o EF gerou.*

```
#:package Microsoft.EntityFrameworkCore.Sqlite@10.0.13
#:property PublishAot=false
using Microsoft.Data.Sqlite;
using Microsoft.EntityFrameworkCore;

using var conexao = new SqliteConnection("Data Source=:memory:");
conexao.Open();  // o banco em memória vive enquanto a conexão estiver aberta
var opcoes = new DbContextOptionsBuilder<LivrariaDb>().UseSqlite(conexao).Options;

using (var db = new LivrariaDb(opcoes))
{
    db.Database.EnsureCreated();
    db.Autores.AddRange(
        new Autor { Nome = "Machado de Assis", Livros = [new() { Titulo = "Dom Casmurro", Preco =
35 }, new() { Titulo = "Quincas Borba", Preco = 38 }] },
        new Autor { Nome = "Graciliano Ramos", Livros = [new() { Titulo = "Vidas Secas", Preco =
49.9m }] });
    db.SaveChanges();
}

using (var db = new LivrariaDb(opcoes))
{
    var consulta = db.Autores
        .OrderBy(a => a.Nome)
        .Select(a => new
        {
            a.Nome,
            Quantidade = a.Livros.Count,
            MaisCaro = a.Livros.OrderByDescending(l => l.Preco).Select(l => l.Titulo).First(),
        });
    foreach (var a in consulta)
        Console.WriteLine($"{a.Nome}: {a.Quantidade} livro(s), mais caro: {a.MaisCaro}");
}

class Autor
{
    public int Id { get; set; }
    public required string Nome { get; set; }
    public List<Livro> Livros { get; set; } = [];
}

class Livro
{
    public int Id { get; set; }
    public required string Titulo { get; set; }
    public decimal Preco { get; set; }
    public int AutorId { get; set; }
}

class LivrariaDb(DbContextOptions<LivrariaDb> opcoes) : DbContext(opcoes)
{
    public DbSet<Autor> Autores => Set<Autor>();
    public DbSet<Livro> Livros => Set<Livro>();
```

```
    protected override void ConfigureConventions(ModelConfigurationBuilder b)
        => b.Properties<decimal>().HaveConversion<double>();  // SQLite não ordena decimal
guardado como texto
}
```

Saída

```
Graciliano Ramos: 1 livro(s), mais caro: Vidas Secas
Machado de Assis: 2 livro(s), mais caro: Quincas Borba
```

As convenções do EF deduzem quase tudo: `Id` é a chave, `AutorId` é a chave estrangeira, e a lista `Livros` define o relacionamento um-para-muitos. A projeção traz **só** o nome, a contagem e um título, numa consulta só (com subconsultas: `Console.WriteLine(consulta.ToQueryString())` mostra o SQL exato, e `LogTo(Console.WriteLine)` nas opções registra cada comando executado), sem carregar os livros na memória; é a correção do N+1 vista acima. O `ConfigureConventions` existe por uma limitação do SQLite: o EF grava `decimal` como texto nele e não consegue ordenar por esse campo; com PostgreSQL ou SQL Server, a linha não é necessária. Num projeto real, o esquema evolui com *migrations* (`dotnet ef migrations add`), e não com `EnsureCreated`.

Q53 Um repositório, vários bancos

> *Defina `interface ILivroRepositorio { Task<Livro?> ObterAsync(int id); Task AdicionarAsync(Livro l); }`, uma implementação em memória e um `CatalogoService` que dependa só da interface. Registre tudo num `ServiceCollection` (injeção de dependência) e mostre que trocar a implementação é uma linha.*

```
#:package Microsoft.Extensions.DependencyInjection@10.0.13
using Microsoft.Extensions.DependencyInjection;

var servicos = new ServiceCollection();
servicos.AddSingleton<ILivroRepositorio, RepositorioEmMemoria>();  // trocar o banco = trocar
esta linha
servicos.AddTransient<CatalogoService>();

using var provedor = servicos.BuildServiceProvider();
var catalogo = provedor.GetRequiredService<CatalogoService>();
await catalogo.CadastrarAsync(new Livro(1, "Vidas Secas", 49.9m));
Console.WriteLine(await catalogo.DescreverAsync(1));
Console.WriteLine(await catalogo.DescreverAsync(2));

record Livro(int Id, string Titulo, decimal Preco);

interface ILivroRepositorio
{
    Task<Livro?> ObterAsync(int id);
    Task AdicionarAsync(Livro l);
}

class RepositorioEmMemoria : ILivroRepositorio
{
    private readonly Dictionary<int, Livro> _dados = [];
    public Task<Livro?> ObterAsync(int id) => Task.FromResult(_dados.GetValueOrDefault(id));
    public Task AdicionarAsync(Livro l)
    {
        _dados[l.Id] = l;
        return Task.CompletedTask;
    }
}

class CatalogoService(ILivroRepositorio repo)
{
    public Task CadastrarAsync(Livro l) => repo.AdicionarAsync(l);

    public async Task<string> DescreverAsync(int id)
        => await repo.ObterAsync(id) is { } l ? $"{l.Titulo} por R$ {l.Preco}" : $"livro {id} não
encontrado";
}
```

Saída

```
Vidas Secas por R$ 49.9
livro 2 não encontrado
```

O `CatalogoService` não sabe se os livros estão num dicionário, no PostgreSQL ou no MongoDB: recebe a interface pelo construtor, e o contêiner decide qual implementação entregar. Nos testes, a versão em memória substitui o banco sem nenhum *mock*. O tempo de vida importa: `Singleton` (um para o app inteiro) serve para o repositório em memória, mas um `DbContext` do EF deve ser `Scoped` (um por requisição), porque não é seguro para uso concorrente.

MÓDULO 6

ASP.NET Core e serviços

Hosting e pipeline de middleware, a API da livraria, validação e ProblemDetails, JWT, configuração e logging, cache, SignalR, workers e mensageria, gRPC e produção.

Artigos 43 a 52

#43 ASP.NET Core por dentro: hosting, pipeline de middleware e Minimal APIs
#44 A API da Livraria: recursos, verbos, DTOs e concorrência sobre HTTP
#45 Validação, ProblemDetails e OpenAPI: uma API que explica a si mesma
#46 Autenticação e autorização com JWT: quem é você e o que pode fazer
#47 Configuração, logging e testes de integração: a API pronta para ser operada
#48 Cache com Redis, HybridCache e output caching: responder sem ir ao banco
#49 Tempo real com SignalR: quando o servidor toma a iniciativa
#50 Worker Services e mensageria com RabbitMQ: trabalho fora da requisição
#51 gRPC: contratos binários e streaming entre serviços
#52 Produção: CORS, rate limiting, versionamento e OpenTelemetry

Perguntas conceituais — respostas

Q54 A ordem dos middlewares

O pipeline do ASP.NET Core é uma sequência de middlewares. Por que `app.UseAuthentication()` precisa vir antes de `app.UseAuthorization()`, e o que acontece se um middleware não chamar `next()`?

Cada *middleware* recebe a requisição, pode fazer algo **antes**, chama o próximo (`await next(context)`) e pode fazer algo **depois**, na volta: o pipeline é uma pilha de chamadas aninhadas. A autenticação descobre **quem** é o usuário (lê o *cookie* ou o *token* e preenche `HttpContext.User`); a autorização decide **se** ele pode acessar o *endpoint*. Com a ordem invertida, a autorização olharia para um usuário ainda anônimo e recusaria tudo. Um *middleware* que não chama `next()` **encerra** o pipeline (*short-circuit*): é o que fazem os arquivos estáticos ao achar o arquivo, o *rate limiter* ao devolver 429 e a autorização ao devolver 401 ou 403.

Q55 Entidade ou DTO

Por que a API da livraria não deve receber nem devolver diretamente a entidade `Livro` do EF Core? O que é um DTO, e que ataque o overposting permite?

A entidade é o modelo **interno**: tem chaves estrangeiras, propriedades de navegação, campos de auditoria e regras que mudam com o banco. Expô-la acopla o contrato público ao esquema (mudar uma coluna quebra os clientes) e pode vazar dados (o custo de aquisição, por exemplo). Um **DTO** (*Data Transfer Object*) é um tipo feito só para o contrato: `CriarLivroRequest(string Titulo, decimal Preco)` na entrada e `LivroResponse(...)` na saída. O *overposting* acontece quando a API faz *binding* da entidade inteira: o cliente envia um campo que não deveria poder definir, como `"Destaque": true` ou `"Preco": 0`, e ele é gravado. Com um DTO de entrada, só os campos previstos existem.

Q56 Concorrência otimista com ETag

> *Dois atendentes abrem o mesmo livro, e cada um altera o preço e salva. Como `ETag` e `If-Match` evitam que a segunda gravação apague a primeira sem ninguém perceber? Que código HTTP a API devolve?*

No `GET`, a API devolve o recurso com um `ETag`, uma versão (um contador ou um *hash* do conteúdo, como o `RowVersion` do EF). No `PUT`, o cliente envia `If-Match: <etag que ele leu>`. A API só grava se a versão atual ainda for aquela; senão, devolve **412 Precondition Failed**. O primeiro atendente grava e muda a versão; o segundo recebe 412, recarrega e decide. É **concorrência otimista**: nada é travado durante a edição, e o conflito é detectado na gravação. Sem isso, vale a "última gravação", que perde dados em silêncio.

Q57 O que um JWT protege

> *Um JWT tem três partes separadas por ponto. O que cada uma contém, e por que é um erro guardar a senha ou o CPF do usuário dentro dele?*

O cabeçalho (algoritmo), o *payload* (as *claims*: `sub`, `role`, `exp`...) e a **assinatura**. As duas primeiras são só JSON em **Base64Url**: qualquer um que tenha o *token* consegue lê-las, sem chave nenhuma. A assinatura garante a **integridade** (ninguém altera o *payload* sem invalidá-la), não o **sigilo**. Por isso dado sensível nunca vai no JWT. A API valida assinatura, emissor, audiência e validade (`exp`) a cada requisição; como o *token* não pode ser revogado antes de expirar sem um controle extra, o *token* de acesso deve durar pouco (minutos) e ser renovado com um *refresh token*.

Q58 De onde vem a configuração

> *No ASP.NET Core, a mesma chave pode vir do `appsettings.json`, do `appsettings.Production.json`, de variáveis de ambiente e da linha de comando. Qual vence, e onde deve ficar a connection string de produção?*

As fontes são lidas em ordem, e **a última vence**: `appsettings.json`, depois `appsettings.{Ambiente}.json`, depois *user secrets* (só em desenvolvimento), variáveis de ambiente e, por fim, argumentos de linha de comando. A chave aninhada `ConnectionStrings:Livraria` vira a variável de ambiente `ConnectionStrings__Livraria` (dois sublinhados). A *connection string* de produção, com senha, **não** vai para nenhum arquivo versionado: vem de variável de ambiente, de um cofre de segredos (Azure Key Vault, AWS Secrets Manager) ou do orquestrador. O `appsettings.json` do repositório guarda só valores de desenvolvimento e padrões sem segredo.

Q59 Cache: o problema difícil

> *"Há duas coisas difíceis em computação: invalidar cache e dar nomes às coisas." Ao guardar o catálogo da livraria no Redis, que estratégias existem para que um preço alterado não continue aparecendo velho, e o que é uma cache stampede?*

- **Expiração** (TTL): o item some depois de N minutos. Simples, mas aceita um atraso de até N minutos.
- **Invalidação explícita**: quem altera o preço apaga (ou atualiza) a chave do livro. Exige que **todo** caminho de escrita lembre de invalidar.

- **Etiquetas**: o `HybridCache` (.NET 9) permite invalidar todas as entradas de uma etiqueta (`"catalogo"`) de uma vez.

A *cache stampede* acontece quando uma chave muito usada expira e centenas de requisições simultâneas encontram o cache vazio e vão **todas** ao banco ao mesmo tempo. O `HybridCache` evita isso: só uma requisição calcula o valor, e as outras aguardam o resultado dela.

Q60 **Entrega pelo menos uma vez**

> *O RabbitMQ garante entrega "pelo menos uma vez" quando o consumidor só confirma (`ack`) depois de processar. Por que isso obriga o consumidor que envia o e-mail de confirmação de pedido a ser **idempotente**, e como fazê-lo?*

Se o consumidor processa a mensagem e cai **antes** de enviar o `ack`, o RabbitMQ entrega a mesma mensagem de novo, para ele ou para outro consumidor: o cliente receberia dois e-mails. Não há como garantir "exatamente uma vez" só com a fila. Ser idempotente significa que processar a mesma mensagem duas vezes tem o mesmo efeito que uma: cada mensagem carrega um identificador único (o id do pedido mais o tipo do evento), e o consumidor registra os ids já processados, de preferência na mesma transação do efeito, e ignora repetidos.

Q61 **gRPC ou REST**

> *Para a comunicação entre o serviço de pedidos e o de estoque, por que gRPC pode ser melhor que uma API REST com JSON? Por que ele não costuma ser usado diretamente pelo navegador?*

O gRPC usa **Protocol Buffers**, um formato binário compacto e rápido de serializar, sobre HTTP/2, com várias chamadas multiplexadas numa conexão e *streaming* nos dois sentidos. O contrato (`.proto`) é a fonte da verdade: cliente e servidor são **gerados** a partir dele, com tipos verificados na compilação. Entre serviços internos, isso significa menos banda, menos CPU e menos erros de integração. O navegador não dá ao JavaScript o controle sobre os quadros HTTP/2 de que o gRPC precisa; é preciso o gRPC-Web, com um *proxy* ou suporte no servidor. Para APIs públicas, REST com JSON continua mais simples de consumir e depurar.

Exercícios de código — respostas

Q62 **O pipeline em ação**

> *Escreva uma aplicação ASP.NET Core mínima com dois middlewares próprios: um que mede o tempo e acrescenta o cabeçalho `X-Tempo-Ms`, e outro que recusa com 403 qualquer requisição para `/admin` sem o cabeçalho `X-Chave: segredo`. Suba o servidor numa porta local, faça três requisições com `HttpClient` dentro do próprio programa e mostre a ordem em que os middlewares rodam.*

```
#:sdk Microsoft.NET.Sdk.Web
#:property PublishAot=false
using Microsoft.AspNetCore.Hosting.Server;
using Microsoft.AspNetCore.Hosting.Server.Features;

var builder = WebApplication.CreateBuilder();
builder.Logging.ClearProviders();  // sem os logs do host, para a saída ficar limpa
builder.WebHost.UseUrls("http://127.0.0.1:0");  // porta 0: o sistema escolhe uma livre
var app = builder.Build();

app.Use(async (ctx, next) =>
{
    var inicio = TimeProvider.System.GetTimestamp();
    Console.WriteLine($"  [tempo] entra {ctx.Request.Path}");
    ctx.Response.OnStarting(() =>
    {
        ctx.Response.Headers["X-Tempo-Ms"] =
TimeProvider.System.GetElapsedTime(inicio).TotalMilliseconds.ToString("F0");
        return Task.CompletedTask;
    });
    await next(ctx);
    Console.WriteLine($"  [tempo] sai {ctx.Request.Path} com {ctx.Response.StatusCode}");
});

app.Use(async (ctx, next) =>
{
    if (ctx.Request.Path.StartsWithSegments("/admin") && ctx.Request.Headers["X-Chave"] !=
"segredo")
    {
        Console.WriteLine("  [guarda] recusou");
        ctx.Response.StatusCode = StatusCodes.Status403Forbidden;
        return;  // não chama next: o pipeline termina aqui
    }
    await next(ctx);
});

app.MapGet("/livros", () => "catálogo");
app.MapGet("/admin", () => "painel");

await app.StartAsync();
var endereco = app.Services.GetRequiredService<IServer>().Features.Get<IServerAddressesFeature>
()!.Addresses.First();
using var http = new HttpClient { BaseAddress = new Uri(endereco) };

foreach (var (rota, chave) in new[] { ("/livros", ""), ("/admin", ""), ("/admin", "segredo") })
{
    Console.WriteLine($"GET {rota} {(chave == "" ? "" : "com chave")}");
    using var req = new HttpRequestMessage(HttpMethod.Get, rota);
    if (chave != "")
        req.Headers.Add("X-Chave", chave);
    using var resp = await http.SendAsync(req);
    Console.WriteLine($"-> {(int)resp.StatusCode} \"{await resp.Content.ReadAsStringAsync()}\"
(tem X-Tempo-Ms? {resp.Headers.Contains("X-Tempo-Ms")})");
```

```
}
await app.StopAsync();
```

Saída

```
GET /livros
  [tempo] entra /livros
  [tempo] sai /livros com 200
-> 200 "catálogo" (tem X-Tempo-Ms? True)
GET /admin
  [tempo] entra /admin
  [guarda] recusou
  [tempo] sai /admin com 403
-> 403 "" (tem X-Tempo-Ms? True)
GET /admin com chave
  [tempo] entra /admin
  [tempo] sai /admin com 200
-> 200 "painel" (tem X-Tempo-Ms? True)
```

O "sai" do primeiro *middleware* aparece **depois** de tudo o que vem abaixo dele, inclusive quando o segundo encerra o pipeline: é a pilha de chamadas. O cabeçalho é gravado no `OnStarting` porque, depois que a resposta começa a ser enviada, os cabeçalhos já partiram; tentar alterá-los depois do `next` lançaria exceção. Subir o servidor dentro do próprio programa é também a ideia do `WebApplicationFactory`, usado nos testes de integração.

Q63 Uma API de livros com DTOs e ProblemDetails

Escreva os endpoints `GET /livros/{id}` (200 ou 404), `POST /livros` (201 com o cabeçalho `Location`, ou 400 com `ValidationProblem` quando o título está vazio ou o preço não é positivo) e `PUT /livros/{id}` com `If-Match` (412 se a versão não bater). Use DTOs de entrada e saída e um dicionário como armazenamento. Teste com `HttpClient` no próprio programa.

```
#:sdk Microsoft.NET.Sdk.Web
#:property PublishAot=false
using System.Collections.Concurrent;
using System.Net.Http.Json;
using Microsoft.AspNetCore.Hosting.Server;
using Microsoft.AspNetCore.Hosting.Server.Features;

var builder = WebApplication.CreateBuilder();
builder.Logging.ClearProviders();
builder.WebHost.UseUrls("http://127.0.0.1:0");
builder.Services.AddProblemDetails();
var app = builder.Build();

var livros = new ConcurrentDictionary<int, Livro>();
var proximoId = 0;

app.MapGet("/livros/{id:int}", (int id, HttpContext ctx) =>
{
    if (!livros.TryGetValue(id, out var l))
        return Results.NotFound();
    ctx.Response.Headers.ETag = $"\"{l.Versao}\"";
    return Results.Ok(new LivroResponse(l.Id, l.Titulo, l.Preco));
});

app.MapPost("/livros", (CriarLivroRequest req) =>
{
    var erros = Validar(req);
    if (erros.Count > 0)
        return Results.ValidationProblem(erros);
    var l = new Livro(Interlocked.Increment(ref proximoId), req.Titulo, req.Preco, Versao: 1);
    livros[l.Id] = l;
    return Results.Created($"/livros/{l.Id}", new LivroResponse(l.Id, l.Titulo, l.Preco));
});

app.MapPut("/livros/{id:int}", (int id, CriarLivroRequest req, HttpContext ctx) =>
{
    if (!livros.TryGetValue(id, out var atual))
        return Results.NotFound();
    if (ctx.Request.Headers.IfMatch.ToString() != $"\"{atual.Versao}\"")
        return Results.Problem(statusCode: 412, title: "O livro foi alterado por outra pessoa.
Recarregue.");
    var erros = Validar(req);
    if (erros.Count > 0)
        return Results.ValidationProblem(erros);
    var novo = atual with { Titulo = req.Titulo, Preco = req.Preco, Versao = atual.Versao + 1 };
    if (!livros.TryUpdate(id, novo, atual))  // outra requisição gravou entre a leitura e aqui
        return Results.Problem(statusCode: 412, title: "Conflito de versão.");
    return Results.NoContent();
});

await app.StartAsync();
var endereco = app.Services.GetRequiredService<IServer>().Features.Get<IServerAddressesFeature>
()!.Addresses.First();
```

```
using var http = new HttpClient { BaseAddress = new Uri(endereco) };

var criado = await http.PostAsJsonAsync("/livros", new CriarLivroRequest("Vidas Secas", 49.9m));
Console.WriteLine($"POST válido: {(int)criado.StatusCode}, Location {criado.Headers.Location}");

var invalido = await http.PostAsJsonAsync("/livros", new CriarLivroRequest("", -1m));
var problema = await invalido.Content.ReadFromJsonAsync<HttpValidationProblemDetails>();
Console.WriteLine($"POST inválido: {(int)invalido.StatusCode}, campos: {string.Join(", ",
problema!.Errors.Keys.Order())}");

Console.WriteLine($"GET 99: {(int)(await http.GetAsync("/livros/99")).StatusCode}");

var lido = await http.GetAsync("/livros/1");
var etag = lido.Headers.ETag!.Tag;
Console.WriteLine($"GET 1: {(int)lido.StatusCode}, ETag {etag}");

foreach (var atendente in new[] { "Ana", "Bruno" })
{
    using var put = new HttpRequestMessage(HttpMethod.Put, "/livros/1") { Content =
JsonContent.Create(new CriarLivroRequest("Vidas Secas", 45m)) };
    put.Headers.TryAddWithoutValidation("If-Match", etag);  // os dois leram a mesma versão
    Console.WriteLine($"PUT de {atendente}: {(int)(await http.SendAsync(put)).StatusCode}");
}
await app.StopAsync();

static Dictionary<string, string[]> Validar(CriarLivroRequest r)
{
    var erros = new Dictionary<string, string[]>();
    if (string.IsNullOrWhiteSpace(r.Titulo))
        erros["titulo"] = ["O título é obrigatório."];
    if (r.Preco <= 0)
        erros["preco"] = ["O preço deve ser positivo."];
    return erros;
}

record Livro(int Id, string Titulo, decimal Preco, int Versao);
record CriarLivroRequest(string Titulo, decimal Preco);
record LivroResponse(int Id, string Titulo, decimal Preco);
```

Saída

```
POST válido: 201, Location /livros/1
POST inválido: 400, campos: preco, titulo
GET 99: 404
GET 1: 200, ETag "1"
PUT de Ana: 204
PUT de Bruno: 412
```

O segundo `PUT` recebe 412 porque a versão mudou com o primeiro: Bruno é avisado, em vez de apagar a alteração de Ana. O `TryUpdate` fecha a pequena janela entre comparar a versão e gravar, quando duas requisições chegam ao mesmo tempo; com EF Core, o mesmo papel é do `[Timestamp]` (ou `IsRowVersion`). O corpo do 400 segue a RFC 9457 (*Problem Details*): um formato padrão de erro, com `title`, `status` e o

dicionário `errors`, que qualquer cliente sabe ler. No .NET 10, um `AddValidation()` com *data annotations* nos DTOs substitui o `Validar` manual.

Q64 Limitando requisições

> *Use o rate limiter embutido do ASP.NET Core para permitir no máximo 3 requisições a cada 10 segundos em `/busca`, respondendo 429 às excedentes. Faça 5 requisições seguidas e mostre os códigos.*

```
#:sdk Microsoft.NET.Sdk.Web
#:property PublishAot=false
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.Hosting.Server;
using Microsoft.AspNetCore.RateLimiting;
using Microsoft.AspNetCore.Hosting.Server.Features;

var builder = WebApplication.CreateBuilder();
builder.Logging.ClearProviders();
builder.WebHost.UseUrls("http://127.0.0.1:0");
builder.Services.AddRateLimiter(o =>
{
    o.RejectionStatusCode = StatusCodes.Status429TooManyRequests;
    o.AddFixedWindowLimiter("busca", j =>
    {
        j.PermitLimit = 3;
        j.Window = TimeSpan.FromSeconds(10);
        j.QueueLimit = 0;  // sem fila: excedeu, recusa na hora
    });
});
var app = builder.Build();
app.UseRateLimiter();
app.MapGet("/busca", (string q) => $"resultados para {q}").RequireRateLimiting("busca");

await app.StartAsync();
var endereco = app.Services.GetRequiredService<IServer>().Features.Get<IServerAddressesFeature>
()!.Addresses.First();
using var http = new HttpClient { BaseAddress = new Uri(endereco) };
for (int i = 1; i <= 5; i++)
    Console.WriteLine($"requisição {i}: {(int)(await http.GetAsync($"/busca?
q=machado{i}")).StatusCode}");
await app.StopAsync();
```

Saída

```
requisição 1: 200
requisição 2: 200
requisição 3: 200
requisição 4: 429
requisição 5: 429
```

Aqui o limite é **global** para o *endpoint*: todos os clientes juntos dividem as 3 requisições. Em produção, a partição costuma ser por usuário ou por IP (`AddPolicy` com `RateLimitPartition.GetFixedWindowLimiter(ip,`

`...`)), e atrás de um *proxy* o IP real vem do `X-Forwarded-For`, que só deve ser aceito de *proxies* confiáveis (`UseForwardedHeaders` com `KnownProxies`). A janela fixa permite uma rajada de 6 requisições em pouco tempo, 3 no fim de uma janela e 3 no começo da seguinte; a janela deslizante (`AddSlidingWindowLimiter`) suaviza isso.

Q65 Um worker com fila em memória

> *Sem RabbitMQ, simule o padrão de mensageria com `System.Threading.Channels`: um `Channel<T>` limitado a 2 itens recebe eventos `PedidoCriado`. Mostre que a fila recusa um terceiro evento enquanto ninguém consome (`TryWrite`), depois inicie um consumidor que processa cada evento de forma idempotente, ignorando um evento entregue duas vezes, e publique o restante com `WriteAsync`, que espera quando a fila enche.*

```
using System.Threading.Channels;

var canal = Channel.CreateBounded<PedidoCriado>(new BoundedChannelOptions(2) { FullMode =
BoundedChannelFullMode.Wait });
var repetido = new PedidoCriado(Guid.NewGuid(), 3);
PedidoCriado[] eventos = [new(Guid.NewGuid(), 1), new(Guid.NewGuid(), 2), repetido, repetido,
new(Guid.NewGuid(), 4)];

// Sem consumidor, a fila enche no segundo evento.
foreach (var ev in eventos[..3])
    Console.WriteLine($"TryWrite pedido {ev.PedidoId}: {(canal.Writer.TryWrite(ev) ? "na fila" :
"fila cheia, recusado")}");

var processados = new HashSet<Guid>();
var consumidor = Task.Run(async () =>
{
    await foreach (var ev in canal.Reader.ReadAllAsync())
    {
        if (!processados.Add(ev.EventoId))
        {
            Console.WriteLine($"consumidor: evento repetido do pedido {ev.PedidoId}, ignorado");
            continue;
        }
        await Task.Delay(50);  // "envia o e-mail"
        Console.WriteLine($"consumidor: e-mail do pedido {ev.PedidoId}");
    }
});

foreach (var ev in eventos[2..])
    await canal.Writer.WriteAsync(ev);  // espera enquanto a fila tiver 2 itens
canal.Writer.Complete();  // "não vem mais nada": o ReadAllAsync termina depois do último
await consumidor;

record PedidoCriado(Guid EventoId, int PedidoId);
```

Saída

```
TryWrite pedido 1: na fila
TryWrite pedido 2: na fila
TryWrite pedido 3: fila cheia, recusado
consumidor: e-mail do pedido 1
consumidor: e-mail do pedido 2
consumidor: e-mail do pedido 3
consumidor: evento repetido do pedido 3, ignorado
consumidor: e-mail do pedido 4
```

O `TryWrite` devolve `false` na hora em que a fila está cheia; o `WriteAsync` **espera** até haver espaço. Essa contrapressão (*backpressure*) impede que um produtor rápido consuma toda a memória. Num `BackgroundService` do ASP.NET Core, o consumidor seria o `ExecuteAsync`, e o *endpoint* que cria o pedido só escreveria no canal e responderia na hora. A fila em memória some se o processo cair; é aí que entra o RabbitMQ, com mensagens persistentes e o `ack` depois do processamento. O `HashSet` de ids processados, em produção, fica no banco, gravado na mesma transação do efeito.

MÓDULO 7

A vitrine web e o deploy

Razor Pages, formulários e validação, Blazor e os modos de renderização, o site consumindo a API, login e upload, e o deploy com contêineres, nginx e Aspire.

Artigos 53 a 59

#53 Razor Pages: HTML montado no servidor para a vitrine da Livraria
#54 Formulários e validação: o site recebe dados
#55 Blazor: componentes, parâmetros, eventos e ciclo de vida
#56 Blazor: modos de renderização, WebAssembly e estado
#57 O site consumindo a API: clientes resilientes, cache condicional e um BFF com YARP
#58 Login, área administrativa, upload de capas e notificações em tempo real
#59 Deploy: contêineres, nginx, Linux e a orquestração com Aspire

Perguntas conceituais — respostas

Q66 Post, Redirect, Get

Depois de processar o formulário de "finalizar pedido", por que a página deve responder com um redirecionamento (303 ou 302) para uma página de confirmação, e não renderizar a confirmação direto?

Se a resposta ao `POST` for a própria página, o navegador guarda o `POST` como última ação: um F5 (ou o botão voltar e avançar) **reenvia o formulário**, e o cliente pode criar o pedido duas vezes (o navegador até avisa "deseja reenviar os dados?"). Com o padrão PRG, o `POST` processa e responde `303 See Other` com `Location: /pedidos/123/confirmacao`; o navegador faz um `GET` nessa URL, e recarregar a página só repete o `GET`, que é inofensivo. Nas Razor Pages, isso é `return RedirectToPage("Confirmacao", new { id })`. O PRG não substitui a idempotência no servidor: um duplo clique rápido ainda pode enviar dois `POST`.

Q67 O token antiforgery

O que o token antiforgery das Razor Pages protege, e por que ele não é necessário numa API que só aceita `Authorization: Bearer`?

Ele protege contra **CSRF**: um site malicioso faz o navegador da vítima enviar um formulário para a livraria, e o navegador anexa sozinho o *cookie* de sessão. O *token* é um valor secreto embutido no formulário (um campo oculto) e conferido no servidor; o site atacante não consegue lê-lo, então não consegue forjar uma requisição válida. As Razor Pages o incluem e validam automaticamente nos formulários. Uma API autenticada por `Bearer` não sofre CSRF porque o navegador **não** anexa esse cabeçalho sozinho: o JavaScript da própria aplicação precisa colocá-lo. O risco volta se a API também aceitar *cookies*.

Q68 Modos de renderização do Blazor

Compare os modos Static SSR, Interactive Server e Interactive WebAssembly do Blazor quanto a onde o código roda, latência das interações e custo para o servidor. Qual usar na página de um livro, no carrinho e no painel administrativo?

- **Static SSR**: o componente é renderizado em HTML no servidor, sem interatividade (como Razor Pages). Rápido e barato. Bom para a **página do livro**, que precisa ser indexada e carregar rápido.
- **Interactive Server**: o componente roda no servidor e mantém uma conexão SignalR (um *circuito*) com o navegador; cada clique viaja até o servidor e volta. Sem *download* inicial, mas cada usuário conectado consome memória no servidor, e a latência de rede aparece em cada interação. Bom para o **painel administrativo**: poucos usuários, acesso direto a dados e regras no servidor.
- **Interactive WebAssembly**: o .NET roda no navegador. As interações são locais, sem custo de servidor, mas há um *download* inicial grande, e o código fica exposto ao usuário. Bom para o **carrinho**, se ele precisar ser muito responsivo (e o modo **Auto** começa no servidor e passa para WebAssembly depois do *download*).

Q69 Cache condicional

> *O site consulta `GET /livros/1` na API a cada visita. Como `ETag` e `If-None-Match` economizam banda, e que código a API devolve quando nada mudou?*

Na primeira resposta, a API envia o `ETag`. O site (ou o `HttpClient` com um cache próprio) guarda o corpo e o `ETag`; na próxima vez, envia `If-None-Match: "<etag>"`. Se a versão não mudou, a API responde **304 Not Modified** sem corpo, e o site reaproveita o que tinha. Ainda há uma ida à rede, mas sem transferir o conteúdo, e a API pode responder sem montar o JSON. É o mesmo `ETag` da concorrência otimista, usado para outro fim: no `If-Match`, ele protege a escrita; no `If-None-Match`, economiza na leitura.

Exercícios de código — respostas

Q70 Validando um upload de capa

> *Escreva `static string? ValidarCapa(string nomeArquivo, long tamanho, ReadOnlySpan<byte> inicio)` que devolva uma mensagem de erro (ou `null` se estiver tudo certo) para o upload da capa de um livro: aceite só JPEG e PNG, conferindo a extensão **e** os primeiros bytes do arquivo (a "assinatura"), com até 2 MB. Teste com um PNG válido, um "PNG" que na verdade é um executável, uma capa grande demais e um nome sem extensão.*

```
static string? ValidarCapa(string nomeArquivo, long tamanho, ReadOnlySpan<byte> inicio)
{
    const long Max = 2 * 1024 * 1024;
    ReadOnlySpan<byte> png = [0x89, 0x50, 0x4E, 0x47, 0x0D, 0x0A, 0x1A, 0x0A];
    ReadOnlySpan<byte> jpeg = [0xFF, 0xD8, 0xFF];

    if (tamanho is <= 0 or > Max)
        return $"tamanho inválido ({tamanho / 1024} KB; máximo 2048 KB)";
    var ext = Path.GetExtension(nomeArquivo).ToLowerInvariant();
    bool assinaturaOk = ext switch
    {
        ".png" => inicio.StartsWith(png),
        ".jpg" or ".jpeg" => inicio.StartsWith(jpeg),
        _ => false,
    };
    if (ext is not (".png" or ".jpg" or ".jpeg"))
        return $"extensão não permitida: \"{ext}\"";
    return assinaturaOk ? null : "o conteúdo não corresponde à extensão";
}

byte[] pngReal = [0x89, 0x50, 0x4E, 0x47, 0x0D, 0x0A, 0x1A, 0x0A, 0x00];
byte[] executavel = [0x4D, 0x5A, 0x90, 0x00];  // "MZ": cabeçalho de um .exe do Windows

Console.WriteLine(ValidarCapa("capa.png", 150_000, pngReal) ?? "aceita");
Console.WriteLine(ValidarCapa("capa.PNG", 150_000, executavel) ?? "aceita");
Console.WriteLine(ValidarCapa("capa.jpg", 5_000_000, [0xFF, 0xD8, 0xFF, 0xE0]) ?? "aceita");
Console.WriteLine(ValidarCapa("capa", 1_000, pngReal) ?? "aceita");
```

Saída

```
aceita
o conteúdo não corresponde à extensão
tamanho inválido (4882 KB; máximo 2048 KB)
extensão não permitida: ""
```

A extensão e o `Content-Type` são escolhidos por quem envia: só a assinatura diz algo sobre o conteúdo. Mesmo assim, ela não é prova de segurança (um arquivo pode ser um PNG válido e carregar outra coisa no fim); em produção, a capa é **regravada** por uma biblioteca de imagens (reduzindo e convertendo), guardada fora da pasta pública ou num *storage* de objetos, e com um nome **gerado pelo servidor**, nunca o enviado. Limitar o tamanho também no servidor web (`MaxRequestBodySize`) evita que o arquivo inteiro seja lido antes da recusa.

Q71 Repetição com espera exponencial

O site chama a API, que às vezes falha por instantes. Escreva `static async Task<T> ComRepeticao<T>(Func<Task<T>> acao, int tentativas, TimeSpan esperaInicial)` que repita a ação em caso de `HttpRequestException`, dobrando a espera a cada falha, e desista depois do número de tentativas, relançando o último erro. Simule uma API que falha duas vezes e depois responde.

```
static async Task<T> ComRepeticao<T>(Func<Task<T>> acao, int tentativas, TimeSpan esperaInicial)
{
    var espera = esperaInicial;
    for (int i = 1; ; i++)
    {
        try
        {
            return await acao();
        }
        catch (HttpRequestException e) when (i < tentativas)  // na última, o filtro falha e a
exceção sobe
        {
            Console.WriteLine($"  tentativa {i} falhou ({e.Message}); nova tentativa em
{espera.TotalMilliseconds} ms");
            await Task.Delay(espera);
            espera *= 2;
        }
    }
}

int chamadas = 0;
Task<string> ApiInstavel() => ++chamadas <= 2
    ? Task.FromException<string>(new HttpRequestException("503 Service Unavailable"))
    : Task.FromResult("{\"titulo\":\"Vidas Secas\"}");

Console.WriteLine(await ComRepeticao(ApiInstavel, tentativas: 4, TimeSpan.FromMilliseconds(50)));

chamadas = -10;  // agora falha sempre
try
{
    await ComRepeticao(ApiInstavel, tentativas: 3, TimeSpan.FromMilliseconds(10));
}
catch (HttpRequestException e)
{
    Console.WriteLine($"desistiu: {e.Message}");
}
```

Saída

```
  tentativa 1 falhou (503 Service Unavailable); nova tentativa em 50 ms
  tentativa 2 falhou (503 Service Unavailable); nova tentativa em 100 ms
{"titulo":"Vidas Secas"}
  tentativa 1 falhou (503 Service Unavailable); nova tentativa em 10 ms
  tentativa 2 falhou (503 Service Unavailable); nova tentativa em 20 ms
desistiu: 503 Service Unavailable
```

O filtro `when (i < tentativas)` é o que faz a última falha subir com o *stack trace* original, sem `throw` explícito. Só se repete o que é **transitório** (503, *timeout*, conexão recusada); um 400 ou um 404 vão falhar igual na próxima vez. Em produção, acrescenta-se um valor aleatório à espera (*jitter*), para que mil clientes não tentem de novo no mesmo instante, e usa-se o `Microsoft.Extensions.Http.Resilience` (sobre o Polly), que junta repetição, *timeout* e *circuit breaker* com uma linha: `AddStandardResilienceHandler()`.

Q72 Um Dockerfile para a API

> *Escreva um `Dockerfile` em múltiplos estágios para a API da livraria: um estágio com o SDK que restaura e publica, e um estágio final só com o runtime do ASP.NET, rodando com um usuário sem privilégios. Explique por que copiar o `.csproj` antes do restante do código acelera os builds.*

```
# Estágio 1: compilar (imagem grande, com o SDK)
FROM mcr.microsoft.com/dotnet/sdk:10.0 AS build
WORKDIR /src
COPY Livraria.Api/Livraria.Api.csproj Livraria.Api/
RUN dotnet restore Livraria.Api/Livraria.Api.csproj
COPY . .
RUN dotnet publish Livraria.Api/Livraria.Api.csproj -c Release -o /app --no-restore

# Estágio 2: executar (só o runtime, sem compilador nem código-fonte)
FROM mcr.microsoft.com/dotnet/aspnet:10.0 AS final
WORKDIR /app
COPY --from=build /app .
USER $APP_UID
ENV ASPNETCORE_HTTP_PORTS=8080
EXPOSE 8080
ENTRYPOINT ["dotnet", "Livraria.Api.dll"]
```

O Docker guarda cada instrução numa camada em cache e só refaz a partir da primeira cuja entrada mudou. Copiando primeiro só o `.csproj` e restaurando, a camada do `restore` (que baixa os pacotes, a parte lenta) só é refeita quando as **dependências** mudam; alterar um `.cs` invalida só as camadas a partir do `COPY . .`. A imagem final não tem o SDK nem o código-fonte: é menor e oferece menos superfície de ataque. As imagens oficiais do .NET 8 em diante já trazem um usuário sem privilégios, `app`, cujo id está em `$APP_UID`, e escutam na porta 8080, que não exige *root*. Um `.dockerignore` com `bin/` e `obj/` impede que artefatos locais entrem no `COPY . .`.

Se o Docker não estiver disponível, o caminho equivalente sem contêiner é publicar com `dotnet publish -c Release -o /srv/livraria`, criar um usuário de sistema (`sudo useradd --system livraria`) e registrar um serviço do systemd com `ExecStart=/usr/bin/dotnet /srv/livraria/Livraria.Api.dll`, `User=livraria` e `Environment=ASPNETCORE_HTTP_PORTS=8080`, atrás do nginx como *proxy* reverso.

MÓDULO 8

Aplicativos de desktop e celular

WinForms, WPF, MAUI e Avalonia; o caixa da livraria com MVVM e vendas offline; o aplicativo do cliente em .NET MAUI.

Artigos 60 a 62

#60 Interfaces no .NET: WinForms, WPF, MAUI, Avalonia e o que escolher
#61 O caixa da Livraria em Avalonia: MVVM, navegação, leitor de código de barras e vendas offline
#62 O aplicativo do cliente em .NET MAUI: Shell, listas, login no celular e acompanhamento de pedidos

Perguntas conceituais — respostas

Q73 Qual framework de interface

O caixa da livraria precisa rodar em Windows e em Linux; o aplicativo do cliente, em Android e iOS. Por que o curso escolheu Avalonia para o primeiro e .NET MAUI para o segundo, e onde WinForms e WPF ainda fazem sentido?

O **Avalonia** desenha os próprios controles e roda em Windows, Linux e macOS com a mesma aparência, usando XAML e MVVM como o WPF: serve ao caixa, que precisa de Linux. O **.NET MAUI** é o caminho oficial da Microsoft para Android e iOS (e também Windows e macOS), usando os controles nativos de cada plataforma: serve ao aplicativo do cliente. **WinForms** e **WPF** são só para Windows: o WinForms é o mais rápido para montar telas simples de cadastro arrastando controles, e o WPF tem o ecossistema mais maduro para aplicações de desktop ricas. Fazem sentido em sistemas internos que só rodam em Windows, e são a realidade de muitos sistemas legados.

Q74 A regra da thread de interface

Por que um clique que faz `var dados = http.GetStringAsync(url).Result;` congela a janela, e por que alterar um controle a partir de uma thread de segundo plano lança exceção?

Os *frameworks* de interface têm uma única **thread de interface**, que processa os eventos (cliques, redesenho) num laço. O `.Result` **bloqueia** essa thread até a resposta chegar: nesse intervalo nenhum evento é processado, e a janela "não responde". Pior, se a continuação precisar voltar para a mesma thread, o resultado é um *deadlock*. O certo é `var dados = await http.GetStringAsync(url);`: a thread fica livre durante a espera, e o código depois do `await` volta a rodar nela. Os controles não são seguros para acesso concorrente; por isso só a thread de interface pode alterá-los, e quem está em outra thread precisa despachar a alteração (`Dispatcher.UIThread.Post` no Avalonia, `MainThread.BeginInvokeOnMainThread` no MAUI).

Q75 Por que MVVM

O que o padrão MVVM separa, e qual a vantagem concreta de ter a lógica do caixa num ViewModel em vez de nos eventos de clique da janela?

O MVVM separa a **View** (o XAML, só aparência), o **ViewModel** (o estado da tela e os comandos, em C# puro, sem referência a controles) e o **Model** (domínio e dados). A View se liga ao ViewModel por *data binding*: quando uma propriedade muda e o ViewModel avisa por `INotifyPropertyChanged`, a tela se atualiza sozinha; quando o usuário aciona um botão, um `ICommand` do ViewModel é executado. A vantagem concreta é que o ViewModel pode ser **testado sem abrir janela nenhuma**: "adicionar um item soma o total", "o botão Finalizar fica desabilitado com o carrinho vazio". Com a lógica nos eventos de clique, o teste exigiria automatizar a interface, e a mesma lógica não poderia ser reaproveitada na versão MAUI.

Exercícios de código — respostas

Q76 Um ViewModel testável

Escreva `class CaixaViewModel : INotifyPropertyChanged` com uma lista de itens, a propriedade calculada `Total`, um `CommandoSimples` (implementação mínima de `ICommand`) para finalizar a venda, habilitado só com itens no carrinho, e o método `Adicionar(string titulo, decimal preco)`. Sem interface gráfica, mostre as notificações disparadas e o estado do comando.

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;

var vm = new CaixaViewModel();
vm.PropertyChanged += (_, e) => Console.WriteLine($"  notificou: {e.PropertyName}");
vm.Finalizar.CanExecuteChanged += (_, _) => Console.WriteLine($"  Finalizar habilitado?
{vm.Finalizar.CanExecute(null)}");

Console.WriteLine($"início: total {vm.Total}, pode finalizar? {vm.Finalizar.CanExecute(null)}");
Console.WriteLine("adiciona Vidas Secas");
vm.Adicionar("Vidas Secas", 49.9m);
Console.WriteLine("adiciona Iracema");
vm.Adicionar("Iracema", 25m);
Console.WriteLine("finaliza");
vm.Finalizar.Execute(null);
Console.WriteLine($"fim: {vm.Mensagem}");

class CaixaViewModel : INotifyPropertyChanged
{
    private readonly List<(string Titulo, decimal Preco)> _itens = [];
    private string _mensagem = "";

    public CaixaViewModel() => Finalizar = new ComandoSimples(FinalizarVenda, () => _itens.Count
> 0);

    public event PropertyChangedEventHandler? PropertyChanged;
    public ComandoSimples Finalizar { get; }
    public decimal Total => _itens.Sum(i => i.Preco);
    public string Mensagem
    {
        get => _mensagem;
        private set { _mensagem = value; Avisar(); }
    }

    public void Adicionar(string titulo, decimal preco)
    {
        _itens.Add((titulo, preco));
        Avisar(nameof(Total));
        Finalizar.ReavaliarEstado();
    }

    private void FinalizarVenda()
    {
        Mensagem = $"venda de {_itens.Count} itens, R$ {Total}";
        _itens.Clear();
        Avisar(nameof(Total));
        Finalizar.ReavaliarEstado();
    }

    private void Avisar([CallerMemberName] string? nome = null) => PropertyChanged?.Invoke(this,
new(nome));
}
```

```
class ComandoSimples(Action executar, Func<bool> podeExecutar) : ICommand
{
    public event EventHandler? CanExecuteChanged;
    public bool CanExecute(object? parametro) => podeExecutar();
    public void Execute(object? parametro) => executar();
    public void ReavaliarEstado() => CanExecuteChanged?.Invoke(this, EventArgs.Empty);
}
```

Saída

```
início: total 0, pode finalizar? False
adiciona Vidas Secas
  notificou: Total
  Finalizar habilitado? True
adiciona Iracema
  notificou: Total
  Finalizar habilitado? True
finaliza
  notificou: Mensagem
  notificou: Total
  Finalizar habilitado? False
fim: venda de 2 itens, R$ 74.9
```

O `[CallerMemberName]` preenche o nome da propriedade sozinho no `set` de `Mensagem`; para o `Total`, que é calculado, o aviso precisa ser explícito. É exatamente isso que a tela usaria: um `TextBlock` ligado a `Total` se redesenha a cada `notificou: Total`, e o botão ligado a `Finalizar` se habilita e desabilita pelo `CanExecuteChanged`. Na prática, o pacote CommunityToolkit.Mvvm gera esse código com `[ObservableProperty]` e `[RelayCommand]`.

Q77 Vendas offline

*O caixa precisa continuar vendendo quando a internet cai. Escreva uma `FilaOffline` que guarde as vendas pendentes, e um método `SincronizarAsync(Func<Venda, Task<bool>> enviar)` que tente enviar cada uma **em ordem**, parando na primeira falha (para não enviar a venda 3 antes da 2), e removendo da fila só as que foram aceitas. Simule uma queda no meio da sincronização.*

```
var fila = new FilaOffline();
for (int i = 1; i <= 4; i++)
    fila.Registrar(new Venda(Guid.NewGuid(), i, 10m * i));
Console.WriteLine($"pendentes: {fila.Pendentes}");

int tentativas = 0;
async Task<bool> ApiInstavel(Venda v)
{
    await Task.Delay(10);
    return ++tentativas != 3;  // a terceira chamada falha: "a internet caiu"
}

await fila.SincronizarAsync(ApiInstavel);
Console.WriteLine($"pendentes depois da queda: {fila.Pendentes}");
await fila.SincronizarAsync(ApiInstavel);
Console.WriteLine($"pendentes depois de voltar: {fila.Pendentes}");

record Venda(Guid Id, int Numero, decimal Total);

class FilaOffline
{
    private readonly Queue<Venda> _pendentes = new();
    public int Pendentes => _pendentes.Count;

    public void Registrar(Venda v) => _pendentes.Enqueue(v);

    public async Task SincronizarAsync(Func<Venda, Task<bool>> enviar)
    {
        while (_pendentes.TryPeek(out var v))  // olha sem remover
        {
            if (!await enviar(v))
            {
                Console.WriteLine($"  venda {v.Numero}: falhou, sincronização interrompida");
                return;
            }
            _pendentes.Dequeue();  // só sai da fila depois de aceita
            Console.WriteLine($"  venda {v.Numero}: enviada");
        }
    }
}
```

Saída

```
pendentes: 4
  venda 1: enviada
  venda 2: enviada
  venda 3: falhou, sincronização interrompida
pendentes depois da queda: 2
  venda 3: enviada
  venda 4: enviada
pendentes depois de voltar: 0
```

O `TryPeek` seguido de `Dequeue` só depois do sucesso é o que garante que nenhuma venda se perde: se o programa fechar no meio do envio, ela continua na fila. O risco oposto é enviar **duas vezes** (a API gravou, mas a resposta não chegou); por isso cada venda leva um `Guid` gerado no caixa, e a API ignora um id que já recebeu (idempotência, como no consumidor do módulo 6). No aplicativo real, a fila fica num SQLite local, para sobreviver ao fechamento.

MÓDULO 9

Jogos

Unity, Godot e MonoGame; o game loop do zero; nós, cenas e sinais; física e colisões; arquitetura sem alocar no loop; áudio, interface e save game.

Artigos 63 a 69

#63 C# nos jogos: Unity, Godot, MonoGame e o game loop
#64 Um game loop do zero com MonoGame: janela, sprites, entrada e os primeiros livros caindo
#65 Godot com C#: nós, cenas e sinais
#66 Input, física e colisões: fazendo o jogo parecer justo
#67 Arquitetura de jogo e por que não alocar dentro do loop
#68 Áudio, interface e save game: o que faz um jogo parecer pronto
#69 Projeto: a Biblioteca em Apuros completa, do documento de design ao export

Perguntas conceituais — respostas

Q78 Unity, Godot ou MonoGame

Compare Unity, Godot e MonoGame quanto ao que cada um oferece pronto e ao modelo de licença. Por que o curso começa o game loop com MonoGame e depois passa para Godot?

- **Unity**: motor completo (editor, física, animação, loja de recursos), com C# como linguagem de *scripts*. Proprietário, com planos pagos acima de certo faturamento.
- **Godot**: motor completo e **livre** (licença MIT), leve, com editor próprio e suporte a C# (via .NET) além do GDScript.
- **MonoGame**: um *framework*, não um motor: dá janela, desenho, entrada e áudio, e o resto (cenas, colisão, câmera) é com você. Livre e de código aberto.

Começar pelo MonoGame mostra **o que um motor faz por dentro**: o laço de `Update` e `Draw`, o `GameTime`, o carregamento de recursos. Depois, no Godot, os nós, as cenas e os sinais deixam de ser mágica e passam a ser conveniências cujo funcionamento você já conhece.

Q79 Por que não alocar no loop

Um jogo roda a 60 quadros por segundo e cria, a cada quadro, uma `List<Inimigo>` temporária e algumas strings para o placar. Por que isso causa "engasgos" periódicos, e que técnicas evitam o problema?

Cada alocação vai para a geração 0 do *heap*. Alocando a cada quadro, a geração 0 enche a cada poucos segundos e o coletor de lixo roda; uma coleta, mesmo curta, pode levar alguns milissegundos, e um quadro a 60 FPS tem só 16,7 ms. O resultado é um engasgo perceptível em intervalos regulares. Técnicas:

- **reaproveitar** coleções: uma `List` como campo, com `Clear()` a cada quadro, em vez de `new` (a capacidade interna é mantida);
- **pool de objetos**: projéteis e partículas voltam para um *pool* em vez de virar lixo;
- **structs** para dados pequenos, como vetores e cores;

- **evitar strings no loop**: atualizar o texto do placar só quando a pontuação muda, ou formatar num `Span<char>`.

Q80 **Sinais do Godot**

> *No Godot, um nó `Moeda` emite o sinal `Coletada` quando o jogador a toca, e o nó `Placar` reage a ele. Que vantagem isso tem sobre a moeda chamar `GetNode<Placar>("/root/Fase/Placar").Somar(10)` diretamente?*

Com a chamada direta, a moeda **conhece** o placar e o caminho exato dele na árvore: mover o placar na cena, reaproveitar a moeda numa fase sem placar ou acrescentar um som de coleta exige mexer no código da moeda. Com o sinal, que é o padrão Observer (o mesmo dos eventos do C#), a moeda só anuncia o que aconteceu, e quem se interessa se conecta: o placar, o tocador de som e o sistema de conquistas, cada um por conta própria. A regra do Godot resume: "chame para baixo, sinalize para cima": um nó chama os filhos diretamente e avisa o pai e os irmãos por sinais.

Exercícios de código — respostas

Q81 **Um pool de projéteis**

> *Escreva `class PoolDeProjeteis` que crie 3 projéteis de antemão, entregue um inativo em `Disparar()` (ou `null` se todos estiverem em uso) e os devolva ao pool quando saem da tela. Simule 10 quadros disparando a cada quadro e mostre que o número de objetos criados não passa de 3. Compare as alocações com uma versão que cria um projétil novo a cada disparo.*

```
var pool = new PoolDeProjeteis(capacidade: 3);
for (int quadro = 1; quadro <= 10; quadro++)
{
    var p = pool.Disparar();
    pool.Atualizar();
    Console.WriteLine($"quadro {quadro,2}: {(p is null ? "sem projétil livre" : $"disparou #
{p.Id}")}, ativos {pool.Ativos}");
}
Console.WriteLine($"projéteis criados no total: {Projetil.Criados}");

class Projetil
{
    public static int Criados;
    public int Id { get; } = ++Criados;
    public bool Ativo;
    public float X;
}

class PoolDeProjeteis
{
    private readonly Projetil[] _itens;
    public PoolDeProjeteis(int capacidade) => _itens = [.. Enumerable.Range(0,
capacidade).Select(_ => new Projetil())];
    public int Ativos => _itens.Count(p => p.Ativo);

    public Projetil? Disparar()
    {
        foreach (var p in _itens)
            if (!p.Ativo)
            {
                p.Ativo = true;
                p.X = 0;
                return p;
            }
        return null;  // pool esgotado: o jogo decide (ignorar o tiro, ou crescer o pool)
    }

    public void Atualizar()
    {
        foreach (var p in _itens)
            if (p.Ativo && (p.X += 30) > 100)  // saiu da tela: volta para o pool
                p.Ativo = false;
    }
}
```

Saída

```
quadro  1: disparou #1, ativos 1
quadro  2: disparou #2, ativos 2
quadro  3: disparou #3, ativos 3
quadro  4: sem projétil livre, ativos 2
quadro  5: disparou #1, ativos 2
quadro  6: disparou #2, ativos 2
quadro  7: disparou #3, ativos 3
quadro  8: sem projétil livre, ativos 2
quadro  9: disparou #1, ativos 2
quadro 10: disparou #2, ativos 2
projéteis criados no total: 3
```

Cada projétil vive quatro quadros (0 → 30 → 60 → 90 → 120) e, com três no *pool*, a cada quatro quadros um disparo fica sem projétil livre. É uma decisão de projeto: um *pool* que cresce sob demanda nunca recusa, mas pode alocar no meio do jogo. O `Ativos` com `Count` percorre o *array* inteiro; com milhares de objetos, mantém-se um contador. Num jogo real, o *pool* é medido no perfilador (o do Visual Studio, o `dotnet-counters` ou o próprio editor do motor), e não adivinhado.

Q82 Save game com versão

Escreva `record SaveGame(int Versao, int Fase, int Pontos, List<string> Itens)` e funções para salvar em JSON e carregar, migrando saves antigos: a versão 1 não tinha `Itens`. Carregue um save da versão 1 escrito à mão, migre para a versão 2 e recuse um arquivo corrompido sem derrubar o jogo.

```
#:property PublishAot=false
using System.Text.Json;

const int VersaoAtual = 2;

SaveGame? Carregar(string json)
{
    try
    {
        var s = JsonSerializer.Deserialize<SaveGame>(json, JsonSerializerOptions.Web);
        if (s is null)
            return null;
        if (s.Versao == 1)
            s = s with { Versao = 2, Itens = s.Itens ?? [] };  // migração: v1 não tinha itens
        return s.Versao == VersaoAtual ? s : null;
    }
    catch (JsonException e)
    {
        Console.WriteLine($"  save corrompido ({e.GetType().Name}); começando um jogo novo");
        return null;
    }
}

var novo = new SaveGame(VersaoAtual, 3, 1200, ["chave dourada"]);
string json = JsonSerializer.Serialize(novo, JsonSerializerOptions.Web);
Console.WriteLine($"salvo: {json}");

string Descrever(SaveGame? s) => s is null ? "nenhum" : $"v{s.Versao}, fase {s.Fase}, {s.Pontos}
pontos, itens [{string.Join(", ", s.Itens ?? [])}]";
Console.WriteLine($"recarregado: {Descrever(Carregar(json))}");
Console.WriteLine($"da versão 1: {Descrever(Carregar("""
{"versao":1,"fase":2,"pontos":300}"""))}");
Console.WriteLine($"corrompido: {Descrever(Carregar("""{"versao":2,"fase":"""))}");

record SaveGame(int Versao, int Fase, int Pontos, List<string>? Itens);
```

Saída

```
salvo: {"versao":2,"fase":3,"pontos":1200,"itens":["chave dourada"]}
recarregado: v2, fase 3, 1200 pontos, itens [chave dourada]
da versão 1: v2, fase 2, 300 pontos, itens []
  save corrompido (JsonException); começando um jogo novo
corrompido: nenhum
```

O campo `Versao` é a decisão que mais economiza dor de cabeça: sem ele, um save antigo é indistinguível de um corrompido. A migração encadeia versões (1 → 2, 2 → 3), cada uma pequena e testável. Para não perder o save se o jogo cair durante a gravação, grave num arquivo temporário e troque com `File.Move(temp, destino, overwrite: true)`, a mesma técnica do banco chave-valor do caderno de C++. O `JsonSerializerOptions.Web` (.NET 9) é uma instância pronta com as opções da web, que evita criar opções novas a cada chamada.

MÓDULO 10

Arquitetura, IA e desempenho

Clean Architecture e DDD com moderação, C# e IA com Microsoft.Extensions.AI e busca semântica, BenchmarkDotNet, Native AOT e ferramentas de linha de comando.

Artigos 70 a 73

#70 Arquitetura: Clean Architecture, DDD com moderação e os padrões que já usamos
#71 C# e IA: Microsoft.Extensions.AI, busca semântica e um assistente com ferramentas
#72 Performance e distribuição: BenchmarkDotNet, Native AOT e ferramentas de linha de comando
#73 Capstone: o jogo, a API, o banco e o site num só fluxo, e a retrospectiva do curso

Perguntas conceituais — respostas

Q83 A regra da dependência

> *Na Clean Architecture, as camadas são Domínio, Aplicação, Infraestrutura e Apresentação. Para que lado apontam as referências entre os projetos, e por que o `Livraria.Dominio` não deve referenciar o pacote do EF Core?*

As dependências apontam **para dentro**, em direção ao domínio: a Apresentação (API) e a Infraestrutura (EF Core, e-mail, fila) dependem da Aplicação, que depende do Domínio, e o Domínio não depende de ninguém. A Aplicação define **interfaces** (`ILivroRepositorio`), e a Infraestrutura as implementa: é a inversão de dependência do módulo 5. Se o Domínio referenciasse o EF Core, as regras de negócio ficariam presas a uma tecnologia de persistência: trocar de banco, testar sem banco ou reaproveitar o domínio num *worker* exigiriam arrastar o EF junto. O "com moderação" do curso vale aqui: num CRUD pequeno, quatro projetos podem ser cerimônia demais, e uma API com o `DbContext` direto é uma escolha honesta.

Q84 Ferramentas para um modelo de IA

> *Num assistente da livraria feito com `Microsoft.Extensions.AI`, o modelo pode chamar a função `BuscarLivros(string termo)`. Quem executa a função de fato, e que cuidados de segurança isso exige?*

O **modelo não executa nada**: ele responde com um pedido estruturado ("chame `BuscarLivros` com `termo = 'sertão'`"). Quem executa é o **seu código** (o `UseFunctionInvocation()` do `ChatClientBuilder` automatiza o ciclo), que devolve o resultado ao modelo para ele montar a resposta final. Cuidados: tratar os argumentos como **entrada não confiável** (validar e usar parâmetros no SQL, como qualquer formulário); expor só funções com o mínimo de poder necessário (buscar, sim; apagar pedidos, não, ou só com confirmação humana); aplicar a autorização do **usuário** que está conversando, e não a do sistema; e lembrar que um texto vindo de fora (a sinopse de um livro, um e-mail) pode conter instruções que tentam desviar o modelo (*prompt injection*).

Q85 Medir antes de otimizar

> *Por que um laço com `Stopwatch` em volta de uma chamada é uma forma pouco confiável de comparar duas implementações, e o que o BenchmarkDotNet faz de diferente?*

Uma medição ingênua mistura o tempo do **JIT** (a primeira chamada compila o método), sofre com o compilador eliminando código cujo resultado não é usado, mede o ruído do sistema (outros processos, frequência variável da CPU, coletas de lixo) e costuma ser feita em modo *Debug*. O BenchmarkDotNet roda cada caso num processo separado, em *Release*, faz o aquecimento, repete até a medida estabilizar, consome os resultados para que não sejam eliminados e relata média, desvio e, com `[MemoryDiagnoser]`, os bytes alocados por operação. É a diferença entre uma impressão e uma medida.

Q86 **Native AOT: o que se ganha e o que se perde**

> *O que muda ao publicar uma ferramenta de linha de comando com `PublishAot=true`? Cite dois ganhos e duas restrições.*

O código é compilado para código de máquina **antes** da execução, num executável único, sem JIT e sem precisar do *runtime* instalado. Ganhos: **partida quase instantânea** (milissegundos, contra dezenas ou centenas com JIT), ótima para ferramentas de terminal e funções *serverless*; e **menor uso de memória**, com um executável que não exige o .NET instalado. Restrições: **reflexão dinâmica e geração de código em execução** ficam limitadas (é por isso que os programas de arquivo único deste caderno precisaram de `PublishAot=false` para JSON por reflexão e EF Core); e o executável é **específico de uma plataforma** (um *build* para Linux x64, outro para Windows ARM64). Bibliotecas que dependem de reflexão precisam de alternativas com *source generators*.

Exercícios de código — respostas

Q87 **Um objeto de valor Dinheiro**

> *No espírito do DDD, escreva `readonly record struct Dinheiro(decimal Valor, string Moeda)` que recuse valores com mais de duas casas, impeça somar moedas diferentes (lançando `InvalidOperationException`) e ofereça `Dividir(int partes)`, que reparte sem perder centavos (o resto vai para as primeiras partes). Divida R$ 100,00 em 3.*

```
var conta = new Dinheiro(100m, "BRL");
var partes = conta.Dividir(3);
Console.WriteLine($"{conta} em 3: {string.Join(" + ", partes)} = {partes.Aggregate((a, b) => a +
b)}");

Console.WriteLine(new Dinheiro(10.5m, "BRL") + new Dinheiro(0.25m, "BRL"));
try { _ = new Dinheiro(1m, "BRL") + new Dinheiro(1m, "USD"); }
catch (InvalidOperationException e) { Console.WriteLine($"recusado: {e.Message}"); }
try { _ = new Dinheiro(0.001m, "BRL"); }
catch (ArgumentException e) { Console.WriteLine($"recusado: {e.Message}"); }

readonly record struct Dinheiro
{
    public decimal Valor { get; }
    public string Moeda { get; }

    public Dinheiro(decimal valor, string moeda)
    {
        if (decimal.Round(valor, 2) != valor)
            throw new ArgumentException($"{valor} tem mais de duas casas decimais");
        Valor = valor;
        Moeda = moeda;
    }

    public static Dinheiro operator +(Dinheiro a, Dinheiro b) =>
        a.Moeda == b.Moeda ? new(a.Valor + b.Valor, a.Moeda)
                           : throw new InvalidOperationException($"não dá para somar {a.Moeda}
com {b.Moeda}");

    public Dinheiro[] Dividir(int partes)
    {
        long centavos = (long)(Valor * 100);
        long cada = centavos / partes, resto = centavos % partes;
        var moeda = Moeda;
        return [.. Enumerable.Range(0, partes).Select(i => new Dinheiro((cada + (i < resto ? 1 :
0)) / 100m, moeda))];
    }

    public override string ToString() => $"{Moeda} {Valor:F2}";
}
```

Saída

```
BRL 100.00 em 3: BRL 33.34 + BRL 33.33 + BRL 33.33 = BRL 100.00
BRL 10.75
recusado: não dá para somar BRL com USD
recusado: 0.001 tem mais de duas casas decimais
```

Dividir 100 por 3 com `decimal` daria 33,3333..., e arredondar cada parte para 33,33 somaria 99,99: um centavo sumiria. Trabalhar em centavos inteiros e distribuir o resto garante que as partes **sempre** somam o total. Como objeto de valor, `Dinheiro` não tem identidade, é imutável e se valida ao nascer: um "valor com moeda" inválido não existe no domínio. A forma não posicional, com construtor explícito, deixa a validação acontecer antes de qualquer atribuição; e como as propriedades não têm `init`, nem um `with` consegue

criar um `Dinheiro` sem passar por ela. O único furo é o `default(Dinheiro)`, que todo `struct` tem: valor 0 e moeda nula.

Q88 Busca semântica com vetores

> *Uma busca semântica compara embeddings: vetores que um modelo de IA gera para cada texto, próximos quando os significados são próximos. Sem chamar modelo nenhum, use vetores de 4 dimensões já prontos para três sinopses e para a pergunta "livro sobre a seca no sertão", calcule a* ***similaridade de cosseno*** *com* `TensorPrimitives.CosineSimilarity` *e ordene os resultados.*

```
#:package System.Numerics.Tensors@10.0.13
using System.Numerics.Tensors;

// Vetores de brinquedo. Um modelo de embeddings real gera centenas ou milhares de dimensões.
(string Titulo, float[] Vetor)[] acervo =
[
    ("Vidas Secas",       [0.9f, 0.8f, 0.1f, 0.0f]),  // seca, sertão
    ("Dom Casmurro",      [0.0f, 0.1f, 0.9f, 0.7f]),  // ciúme, cidade
    ("Os Sertões",        [0.7f, 0.9f, 0.0f, 0.3f]),  // sertão, guerra
];
float[] pergunta = [0.8f, 0.7f, 0.0f, 0.1f];  // "livro sobre a seca no sertão"

foreach (var (titulo, sim) in acervo.Select(l => (l.Titulo,
TensorPrimitives.CosineSimilarity(pergunta, l.Vetor)))
                                    .OrderByDescending(r => r.Item2))
    Console.WriteLine($"{titulo,-13} {sim:F3}");
```

Saída

```
Vidas Secas   0.992
Os Sertões    0.969
Dom Casmurro  0.115
```

A similaridade de cosseno mede o **ângulo** entre os vetores: 1 é a mesma direção, 0 é sem relação, e o tamanho dos vetores não importa. Com `Microsoft.Extensions.AI`, os vetores viriam de um `IEmbeddingGenerator<string, Embedding<float>>`, e o acervo ficaria num banco com índice vetorial (o `pgvector` do PostgreSQL, por exemplo), para não comparar a pergunta com cada livro, um por um. O `TensorPrimitives` usa instruções SIMD do processador, várias multiplicações por instrução.

PARA IR ALÉM

Desafios integradores

Projetos maiores, que juntam vários módulos. Não há uma única resposta certa: o gabarito mostra um caminho.

D1

Uma calculadora de terminal

*Escreva uma calculadora de terminal com sete operações: soma, subtração, multiplicação, divisão, potência (expoente inteiro), porcentagem (`15 % 200` é 15% de 200) e raiz quadrada (`raiz 2`). Ela deve: (a) aceitar vírgula ou ponto como separador decimal; (b) guardar o último resultado em `ans`, utilizável na conta seguinte; (c) recusar divisão por zero, raiz de negativo, expoente fracionário, resultado grande demais e texto que não é número com uma mensagem clara, **sem encerrar o programa**; (d) ter os comandos `hist` (histórico) e `sair`. Separe o núcleo (uma classe `Calculadora` sem `Console`) da interface, e use `decimal` para que `0,1 + 0,2` dê exatamente `0,3`.*

Caminho da solução

- **Núcleo**: `Calculadora.Avaliar(string linha)` devolve um `Resultado` (um *record* com o valor ou a mensagem de erro), sem lançar exceções para erros do usuário e sem nenhuma referência ao `Console`. Num projeto real, ela fica numa biblioteca `Calculadora.Core`, usada pelo terminal, por uma janela e pelos testes.
- **Números**: `decimal.TryParse` com `CultureInfo.InvariantCulture`, depois de trocar a vírgula por ponto. Com `decimal`, as quatro operações são exatas para valores decimais; a raiz quadrada não tem versão `decimal` na biblioteca, então passa por `double` (e perde a exatidão, o que a saída mostra).
- **Potência**: calculada por multiplicações repetidas em `decimal`, dentro de um bloco `checked`, que transforma o estouro em `OverflowException`, tratada como "resultado grande demais".
- **Interface**: o laço lê com `Console.ReadLine`, mostra os resultados com vírgula (cultura `pt-BR`) e guarda o histórico numa `List<string>`.

```
using System.Globalization;

var calc = new Calculadora();
var historico = new List<string>();
var br = CultureInfo.GetCultureInfo("pt-BR");
Console.WriteLine("Calculadora: + - * / ^ % e raiz. Comandos: hist, sair.");
while (true)
{
    Console.Write("> ");
    var linha = Console.ReadLine()?.Trim();
    if (linha is null or "sair")
        break;
    if (linha == "hist")
    {
        for (int i = 0; i < historico.Count; i++)
            Console.WriteLine($"  {i + 1}. {historico[i]}");
        continue;
    }
    if (linha == "")
        continue;
    var r = calc.Avaliar(linha);
    if (r.Erro is null)
    {
        historico.Add($"{linha} = {r.Valor.ToString(br)}");
        Console.WriteLine(historico[^1]);
    }
    else
    {
        Console.WriteLine($"erro: {r.Erro}");
    }
}

record Resultado(decimal Valor, string? Erro)
{
    public static Resultado Ok(decimal v) => new(v, null);
    public static Resultado Falha(string e) => new(0, e);
}

class Calculadora
{
    public decimal Ans { get; private set; }

    public Resultado Avaliar(string linha)
    {
        var p = linha.Split(' ', StringSplitOptions.RemoveEmptyEntries);
        Resultado r = p switch
        {
            ["raiz", var x] => Numero(x) is { } v ? Raiz(v) : Resultado.Falha("operando não é um
número"),
            [var a, var op, var b] => Numero(a) is { } va && Numero(b) is { } vb ? Operar(op, va,
vb)
                                                                                  :
Resultado.Falha("operando não é um número"),
```

```
            _ => Resultado.Falha("use \"a op b\" ou \"raiz a\""),
        };
        if (r.Erro is null)
            Ans = r.Valor;  // só um resultado válido substitui o ans
        return r;
    }

    private decimal? Numero(string texto) =>
        texto == "ans" ? Ans
        : decimal.TryParse(texto.Replace(',', '.'), NumberStyles.Number,
CultureInfo.InvariantCulture, out var v) ? v
        : null;

    private static Resultado Raiz(decimal x) =>
        x < 0 ? Resultado.Falha("raiz de número negativo") :
Resultado.Ok((decimal)Math.Sqrt((double)x));

    private static Resultado Operar(string op, decimal a, decimal b)
    {
        try
        {
            return op switch
            {
                "+" => Resultado.Ok(a + b),
                "-" => Resultado.Ok(a - b),
                "*" => Resultado.Ok(a * b),
                "/" => b == 0 ? Resultado.Falha("divisão por zero") : Resultado.Ok(a / b),
                "%" => Resultado.Ok(a * b / 100),
                "^" => b != decimal.Truncate(b) ? Resultado.Falha("o expoente precisa ser
inteiro") : Resultado.Ok(Potencia(a, (int)b)),
                _ => Resultado.Falha("operação desconhecida (use + - * / ^ % ou raiz)"),
            };
        }
        catch (OverflowException)
        {
            return Resultado.Falha("resultado grande demais");
        }
    }

    private static decimal Potencia(decimal a, int n)
    {
        if (n < 0)
            return 1 / Potencia(a, -n);
        decimal r = 1;
        checked
        {
            for (int i = 0; i < n; i++)
                r *= a;  // decimal estoura com OverflowException
        }
        return r;
    }
}
```

Entrada digitada:

```
2 + 3
0,1 + 0,2
ans * 10
10 / 0
2 ^ 0,5
raiz -4
raiz 2
ans * ans
10 / 3
15 % 200
abc + 1
2 ^ 200
2 $ 3
hist
sair
```

Saída

```
Calculadora: + - * / ^ % e raiz. Comandos: hist, sair.
> 2 + 3 = 5
> 0,1 + 0,2 = 0,3
> ans * 10 = 3,0
> erro: divisão por zero
> erro: o expoente precisa ser inteiro
> erro: raiz de número negativo
> raiz 2 = 1,4142135623731
> ans * ans = 2,0000000000000014004103 60361
> 10 / 3 = 3,33333333333333333333333333
> 15 % 200 = 30
> erro: operando não é um número
> erro: resultado grande demais
> erro: operação desconhecida (use + - * / ^ % ou raiz)
>   1. 2 + 3 = 5
  2. 0,1 + 0,2 = 0,3
  3. ans * 10 = 3,0
  4. raiz 2 = 1,4142135623731
  5. ans * ans = 2,0000000000000014004103 60361
  6. 10 / 3 = 3,33333333333333333333333333
  7. 15 % 200 = 30
>
```

Com a entrada redirecionada, o terminal não ecoa o que foi digitado, e cada resposta aparece logo depois do `>`.

Compare com os cadernos de C e de C++: lá, `0,1 + 0,2` só "dava" 0,3 porque a impressão arredondava; aqui o `decimal` guarda exatamente 0,3. Ele também guarda a **escala**: `0,3 * 10` dá `3,0`, com uma casa, e não `3`. Mas a exatidão do `decimal` é **decimal**, não absoluta: `10 / 3` tem 28 dígitos e um arredondamento no fim, e a raiz de 2 passou por `double` e, na conversão para `decimal`, ficou com 14 algarismos significativos; por isso `ans * ans` não volta a dar 2. O `decimal` também tem alcance menor (até cerca de $7{,}9 \times 10^{28}$), e por isso `2 ^ 200` estoura aqui, enquanto com `double` daria $1{,}6 \times 10^{60}$.

Uma versão gráfica com **Avalonia** (módulo 8) reaproveita a classe `Calculadora` sem mudar uma linha: um `CalculadoraViewModel` com a propriedade `Visor`, um `ICommand` para cada botão que acrescenta o símbolo ao visor e um comando "=" que chama `Avaliar` e mostra o resultado ou o erro. O mesmo ViewModel serve a uma versão .NET MAUI para o celular, e testes xUnit do núcleo valem para as três interfaces.

D2

A livraria de ponta a ponta

Junte as peças do curso numa versão mínima da livraria: uma API (Minimal API) com `GET /livros`, `GET /livros/{id}` e `POST /pedidos`, com EF Core e SQLite; um worker que consome os pedidos criados de um `Channel` e "envia" o e-mail de confirmação; e um cliente de terminal que lista o catálogo e faz um pedido pela API. Inclua testes de integração da API com `WebApplicationFactory`.

Caminho da solução

- **Projetos**: `Livraria.Dominio` (entidades e regras, sem dependências), `Livraria.Api` (*endpoints*, EF Core, o *worker*), `Livraria.Cliente` (terminal com `HttpClient`) e `Livraria.Testes`. Use DTOs nos *endpoints*, nunca as entidades (módulo 6).
- **Pedido**: o `POST /pedidos` valida (livro existe, quantidade positiva, estoque suficiente), grava o pedido e a baixa de estoque **na mesma transação** (`SaveChanges` único), publica `PedidoCriado` no `Channel` e responde `201 Created`. O estoque usa concorrência otimista (`[Timestamp]`), para que dois pedidos simultâneos do último exemplar não sejam ambos aceitos.
- **Worker**: um `BackgroundService` lê o canal com `ReadAllAsync(stoppingToken)` e registra o envio em log. Se o processo cair entre gravar o pedido e publicar o evento, o e-mail se perde: a correção profissional é o padrão *outbox* (gravar o evento numa tabela na mesma transação e publicá-lo depois).
- **Cliente**: `HttpClient` com `AddStandardResilienceHandler` (repetição e *timeout*), `GetFromJsonAsync<List<LivroResponse>>` e tratamento do `ProblemDetails` nos erros 400 e 409.
- **Testes**: `WebApplicationFactory<Program>` sobe a API em memória; troque a conexão do EF por um SQLite em memória por teste (`ConfigureServices` no `WithWebHostBuilder`). Teste o caminho feliz do pedido, o 400 de quantidade inválida, o 404 de livro inexistente e o conflito de estoque com dois pedidos concorrentes do último exemplar.
- **Execução**: o .NET Aspire (módulo 7) sobe a API, o *worker* e um PostgreSQL com um só `dotnet run` do projeto *AppHost*, com o painel de *logs* e *traces* do OpenTelemetry. Sem Aspire e sem Docker, rode a API com `dotnet run --project Livraria.Api` e o cliente com `dotnet run --project Livraria.Cliente` em dois terminais, com o SQLite em arquivo.

Fim do gabarito.