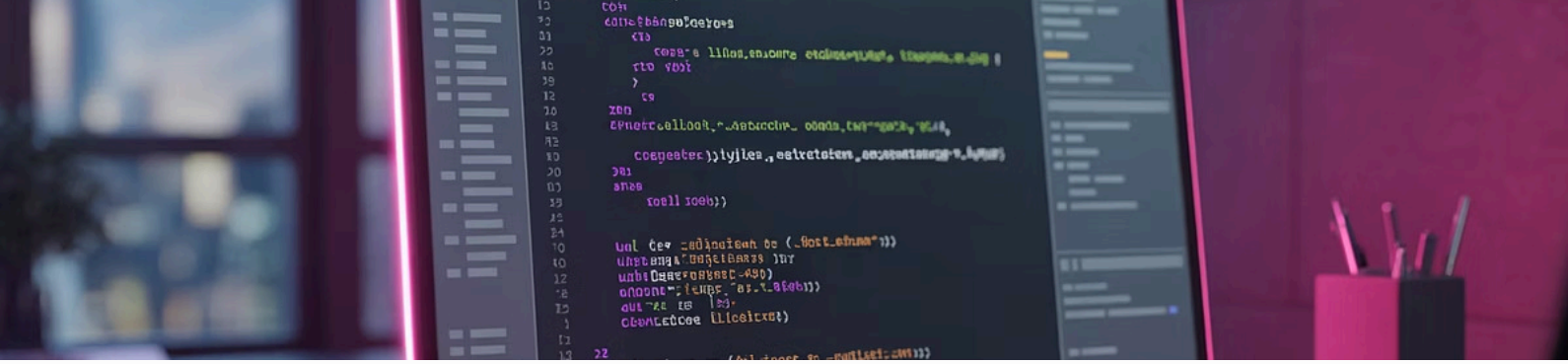


Tipos Nuláveis e Não Nuláveis em C# — Fundamentos, Migração com NRT e Labs

Continuidade do trilho didático: explicação profunda, exemplos do cotidiano, mapeamento para C# e prática incremental (fundamentos → migração → boas práticas).

Objetivos de aprendizagem

1 **Distinguir valor nulo de valor ausente**

Aprender a modelar corretamente estes conceitos no domínio da aplicação

2 **Usar tipos por valor nuláveis e tipos de referência nuláveis**

Trabalhar com `int?`, `DateTime?` e NRT (Nullable Reference Types)

3 **Habilitar e trabalhar com análise de nulidade do compilador**

Usar `enable`, `#nullable enable` para melhorar a segurança do código

4 **Aplicar operadores de nulidade e pattern matching**

Dominar `?.`, `??`, `??=`, `!` e `is null`

5 **Migrar código legado com estratégia**

Reduzir warnings e evitar null reference exceptions

6 **Documentar contratos de nulidade com anotações/atributos**

Usar `NotNull`, `MaybeNull`, etc.

Nulo no cotidiano (sem código)

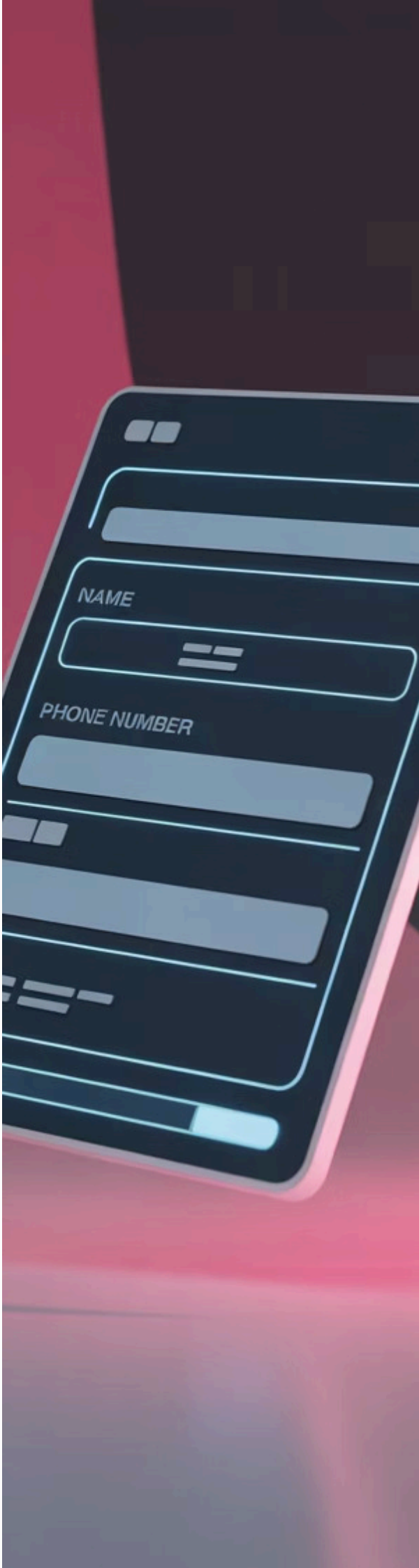
Exemplos práticos:

- **Telefone opcional** no cadastro: pode não existir
- **Segundo nome** (middle name): pode estar ausente
- **Data de retorno** em uma consulta: desconhecida até ser marcada

Modelagem:

Só permita null quando **ausência** for um estado legítimo no domínio. Caso contrário, **exija valor** (não-nulável) e garanta isso no construtor/required.

> A modelagem correta de nulidade reflete a semântica do seu domínio!



Tipos por valor × Tipos por referência

Tipos por valor (struct)

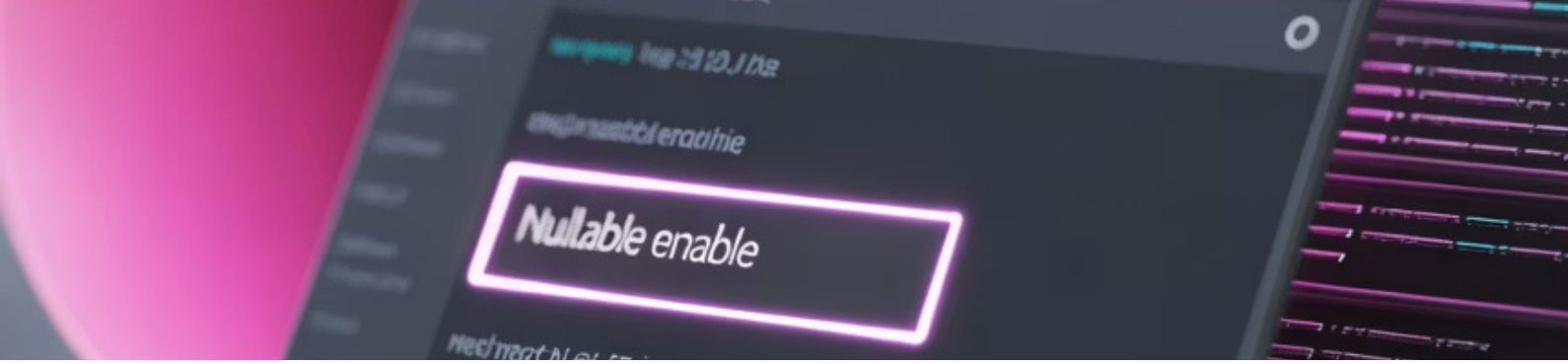
`int`, `double`, `DateTime`, `decimal`...

- **Não aceitam null** por padrão
- Use **T?** para permitir null (ex.: `int?`, `DateTime?`)

Tipos por referência (class)

`string`, classes suas, arrays...

- **Aceitam null** por natureza
- Com NRT ativado você **declara a intenção**:
- `string` (não nulável): o compilador exige que você **inicialize** e garanta não-nulo
- `string?` (nulável): pode ser null, e o compilador exige **checas** antes do uso



Habilitando NRT (Nullable Reference Types)

Ative a análise do compilador:

No projeto (.csproj):

```
<PropertyGroup>
  <Nullable>enable</Nullable>
  <!-- Opcional: trate avisos como erros -->
  <!-- <TreatWarningsAsErrors>true</TreatWarningsAsErrors> -->
</PropertyGroup>
```

Por arquivo/região:

```
#nullable enable
// ...código analisado...
#nullable disable
```

Estados disponíveis:

- `enable` (recomendado)
- `disable`
- `annotations`
- warnings

Declarações, inicialização e invariantes

```
public class Aluno
{
  public string Nome { get; } // não-nulável
  public string? Apelido { get; private set; } // nulável (opcional)
  public DateTime? DataDeRetorno { get; private set; } // nulável (pode
                                                       // não existir)

  public Aluno(string nome)
  {
     if (string.IsNullOrWhiteSpace(nome))
       throw new ArgumentException("Nome inválido");
     Nome = nome; // garante não-nulo
  }
}
```

> **Regra prática:** prefira **construtores/required** para garantir não-nuláveis. Evite inicializar com `= null!;` (operador "forgiving") — só use para cenários muito específicos (ex.: ORMs com *materialization* e você tem certeza da atribuição posterior).

Operadores de nulidade e padrões

Acesso condicional `?.`

Encadeia chamadas apenas se não for null.

```
var tamanho = aluno.Apelido?.Length; // int? (pode ser nulo)
```

Coalescência `??`

Valor padrão quando for null.

```
string exibicao = aluno.Apelido ?? aluno.Nome;
// se Apelido é null, usa Nome
```

Atribuição por coalescência `??=`

```
aluno.Apelido ??= "Sem apelido";
// só define se estiver null
```

Null-forgiving `!` (use com parcimônia)

```
string textoCerteza = aluno.Apelido!;
// diz ao compilador: "eu garanto que não é null"
```

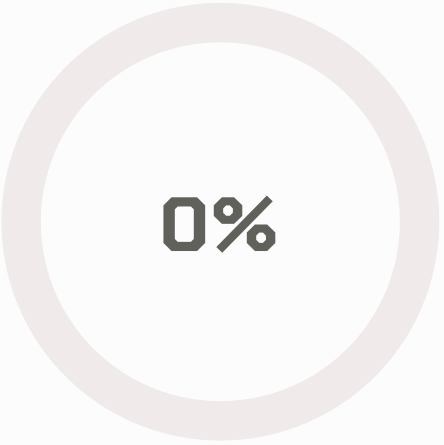
Pattern matching com `is`

```
if (aluno.Apelido is null) {/* tratar ausência */}
if (aluno.Apelido is not null)
 Console.WriteLine(aluno.Apelido.Length);
```

Tipos por valor nuláveis (Nullable)

```
int? idade = null;
idade = 20;
if (idade.HasValue)
  Console.WriteLine(idade.Value);
int valor = idade ?? 0; // coalesce para valor padrão
```

Conversões e operações: operadores aritméticos propagam null (se algum lado é null, resultado pode ser null). Use coalescência para defaults.



Compatibilidade

Tipos por valor nuláveis são compatíveis com todas as versões do C#

Overhead

Praticamente zero overhead de performance em comparação com tipos não-nuláveis

Contratos de nulidade com atributos

Importe `System.Diagnostics.CodeAnalysis` e/ou `System.Diagnostics.Contracts` (alguns atributos variam por versão do .NET).

```
using System.Diagnostics.CodeAnalysis;

public static class Guard
{
   public static void AgainstNull([NotNull] T? value, string paramName)
   {
      if (value is null)
        throw new ArgumentNullException(paramName);
   }
   public static bool TryParseNonEmpty(string? s,
       [NotNullWhen(true)] out string? result)
   {
      if (!string.IsNullOrWhiteSpace(s)) {
        result = s; return true;
      }
      result = null; return false;
   }
}
```

- `[NotNull]` → após a chamada, o compilador sabe que **não é nulo**.
- `[NotNullWhen(true)]` → se retornar **true**, o out não é nulo.
- Outros: `[MaybeNull]`, `[AllowNull]`, `[DisallowNull]`, `[MemberNotNull]`, `[NotNullIfNotNull]`.

Padrões de domínio (boas práticas)

Faça

- Modele **opcionais** com T? (referência) ou T? (valor) **somente** quando "ausência" faz sentido.
- Prefira coleções **vazias** a coleções null (evita if em toda parte).
- Valide **fronteiras** (DTOs, input) e converta para tipos **não-nuláveis** dentro do domínio.
- Use required (C# 11+) para forçar inicialização:

```
public class Produto
{
    public required string Nome { get; init; }
    public decimal Preco { get; init; }
}
```

Evite

- Retornar null quando puder retornar **valor neutro** (ex.: `string.Empty`, `Enumerable.Empty()`).
- Espalhar `!` (null-forgiving) para calar o compilador.
- Tornar tudo `?` por preguiça de modelar o domínio.

Migração de código legado (estratégia prática)

Ative NRT no projeto

Ou por arquivo em módulos sensíveis

Trate warnings de fora para dentro

Comece pelas bordas e avance para o domínio

Endpoints/DTOs

Marque campos realmente opcionais com `?`

Mappers

Transforme opcionais em **valores obrigatórios** (aplique defaults/validação)

Domínio

Mantenha propriedades **não-nuláveis**; construtores/required garantem invariantes

Substitua retornos null

Use valores neutros quando fizer sentido

Acrescente guards e atributos

Ajude o analisador com `AgainstNull` e atributos de nulidade

Use ! com cautela

Só quando **tiver certeza** do fluxo (e prefira registrar essa garantia no tipo/contrato)

Interoperabilidade e frameworks

EF Core

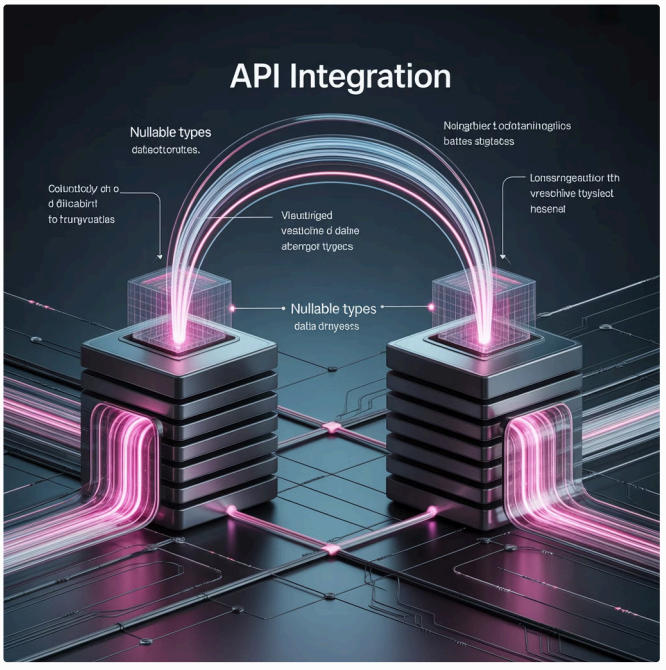
A nulidade da propriedade mapeia para NULL da coluna. Ajuste **migrations** ao tornar `string?` → `string` (ou vice-versa).

APIs (JSON)

Campos opcionais → `?` no DTO; valide na borda antes de chegar ao domínio.

Bibliotecas externas sem NRT

O compilador trata como "*oblivious*" (sem informação). Faça checagens manuais ou encapsule.



Labs guiados (incrementais)

Use um projeto AulaNullability.

1

Lab A — Fundamentos

1. Ative NRT no projeto.
2. Declare `string nome` sem inicializar: observe o *warning* e corrija.
3. Declare `string? apelido` e trate com `??` / `?.` antes de imprimir.

2

Lab B — Guards e atributos

1. Implemente `Guard.AgainstNull([NotNull] T? value, string paramName)`.
2. Crie um método que retorna bool + out com `[NotNullWhen(true)]` e comprove no IntelliSense que o out é não-nulo após check.

3

Lab C — DTO → Domínio

1. Crie `AlunoDto { string? Nome; DateTime? Retorno; }`.
2. No mapper, valide Nome e converta para Aluno com Nome **não-nulável**.
3. Registre decisão: Retorno null significa **"a definir"**.

4

Lab D — Valor nulável

1. Use `DateTime? proximaConsulta`.
2. Se null, imprima "Sem data"; caso contrário, formate a data.

5

Lab E — EF/Infra (conceitual)

1. Modele `Produto.Nome` (não-nulável) e `Produto.Descricao?` (opcional).
2. Discuta o impacto na migration (coluna Descricao NULL).

Exercícios com "resultado esperado"

Exercício 1

```
string? s = null;
Console.WriteLine(s?.Length ?? 0);
```

Resultado esperado: 0

Exercício 2

Explique diferença: `if (s == null)` vs `if (s is null)`

Resposta: mesma semântica, `is` evita *operator overload* customizado.

Exercício 3

Quando usar `!`?

Resposta: somente quando você **garante** não-nulo pelo fluxo e não consegue expressar isso no tipo.

Exercício 4

Transforme `List? Itens` em **nunca nulo**

Resposta: inicialize com `= new();` e garanta invariantes no construtor.

Exercício 5

```
int? x = null;
var r = x ?? -1;
```

Resultado esperado: -1

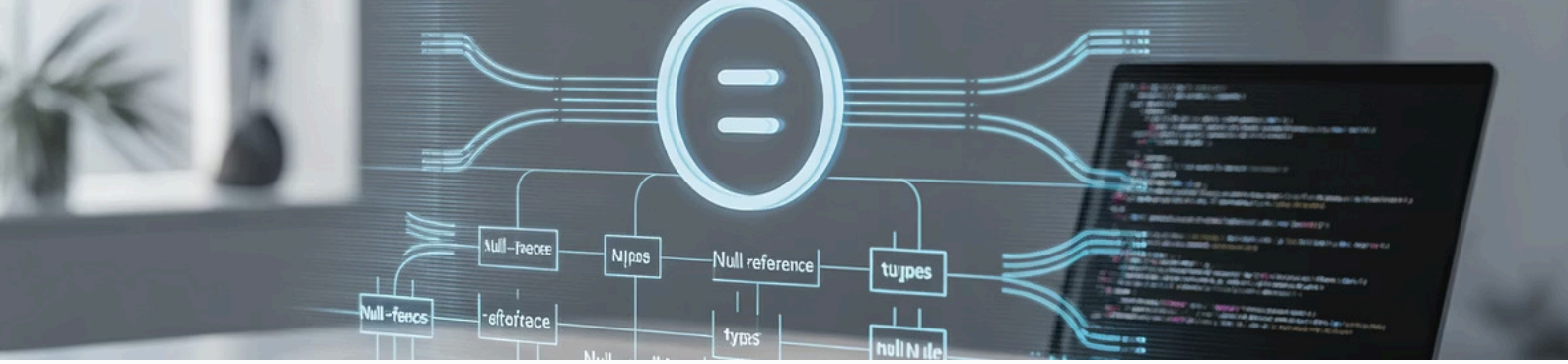
Checklist de consolidação

- [ ] **Sei quando modelar ? no domínio**

  Compreendo a semântica de ausência e quando aplicar tipos nuláveis

- [ ] **Ativei NRT e sei interpretar/zerar warnings comuns**

  Consigo configurar o projeto e resolver avisos do compilador



- [ ] **Domino ?., ??, ??=, ! e is null**

  Aplico os operadores de nulidade com confiança e precisão

- [ ] **Sei usar atributos de nulidade para documentar contratos**

  Utilizo NotNull, MaybeNull e outros para melhorar a análise estática

- [ ] **Consegui migrar DTOs e repositórios mantendo o domínio não-nulável**

  Aplico estratégias de migração em código legado com sucesso

Próximos passos

Records imutáveis

Com required e with-expressions para construir invariantes.

```
public record Produto(string Nome, decimal Preco)
{
    public string? Descricao { get; init; }
}

// Uso com with-expression
var produtoComDescricao = produto with
{
    Descricao = "Detalhes..."
};
```

Result/Option types

Padrão funcional quando o domínio pede explicitamente "valor ou ausência/erro".

Analisadores Roslyn

Integração com analisadores para políticas de nulidade do time.

Fechamento

> Nulidade é sobre **semântica de ausência**. Ao declarar **explicitamente** o que pode ou não ser null, você ensina ao compilador (e ao time) as regras do domínio.

Com NRT, operadores de nulidade e uma estratégia de migração clara, seu código:

Reduz NullReferenceException

Menos erros em tempo de execução

Ganha legibilidade

Intenções claras no código

Fica mais fácil de evoluir

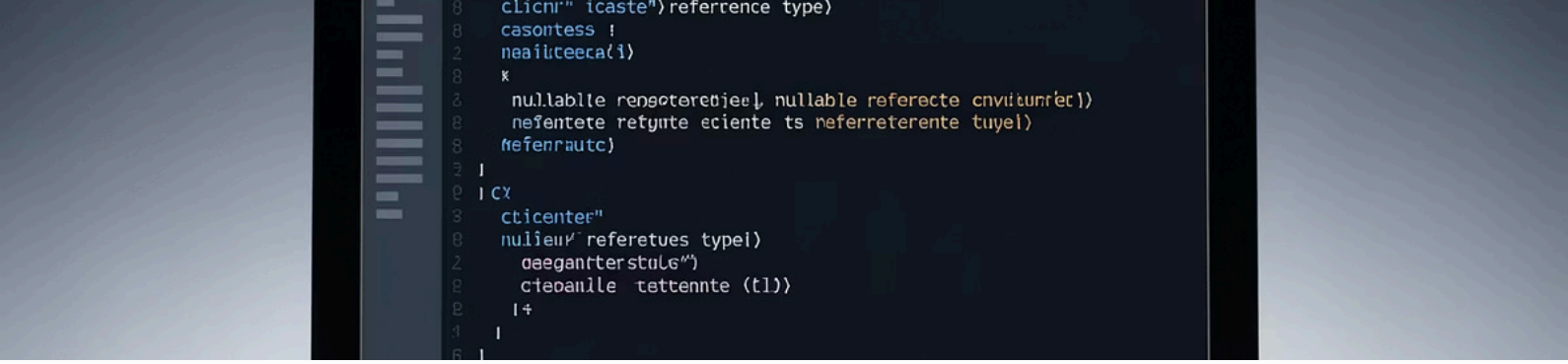
Contratos explícitos facilitam manutenção

Exemplos práticos de uso

Antes (sem NRT)

```
public class Cliente
{
    public string Nome { get; set; }
    public string Email { get; set; }
    public List Pedidos { get; set; }

    // Risco de NullReferenceException
    public int TotalPedidos => Pedidos.Count;
}
```



Depois (com NRT)

```
public class Cliente
{
 public required string Nome { get; init; }
 public string? Email { get; init; }
 public List Pedidos { get; init; } = new();

 // Seguro - nunca será null
 public int TotalPedidos => Pedidos.Count;

 // Uso seguro de nullable
 public string EmailDisplay => Email ?? "Não informado";
}
```

Recursos adicionais

Documentação oficial

Microsoft Learn: Nullable Reference Types

https://learn.microsoft.com/pt-br/dotnet/csharp/nullable-references

Ferramentas

- ReSharper/Rider: suporte avançado para análise de nulidade
- NullGuard.Fody: injeção de verificações em tempo de compilação

Comunidade

Stack Overflow: tag [c#-nullable]

GitHub: exemplos de projetos com NRT habilitado

Lembre-se: nulidade bem modelada reflete a semântica do seu domínio e torna seu código mais expressivo e seguro!