C#: Igualdade de Objetos com Equals

Fundamentos, Problemas e Implementações

Material didático para aula de Programação Orientada a Objetos em C#, abordando fundamentação teórica, justificativas práticas, riscos de implementação inadequada e exemplos detalhados com código.

Objetivos de Aprendizagem

1 Distinguir Identidade de Igualdade

Compreender a diferença entre identidade de objeto (referência em memória) e igualdade de valor (equivalência no domínio de negócio).

2 Dominar o Contrato de Equals

Explicar o contrato de Equals e sua relação fundamental com GetHashCode para garantir consistência.

3 Implementar Igualdade Corretamente

Usar IEquatable, operadores == / != e comparadores em classes de domínio de forma adequada.

4 Identificar e Evitar Armadilhas

Reconhecer problemas com coleções Hash*, campos mutáveis e herança, aplicando boas práticas como imutabilidade e records.

O Conceito de Igualdade em Software

Dois Tipos Fundamentais de Igualdade

Identidade (Referência)

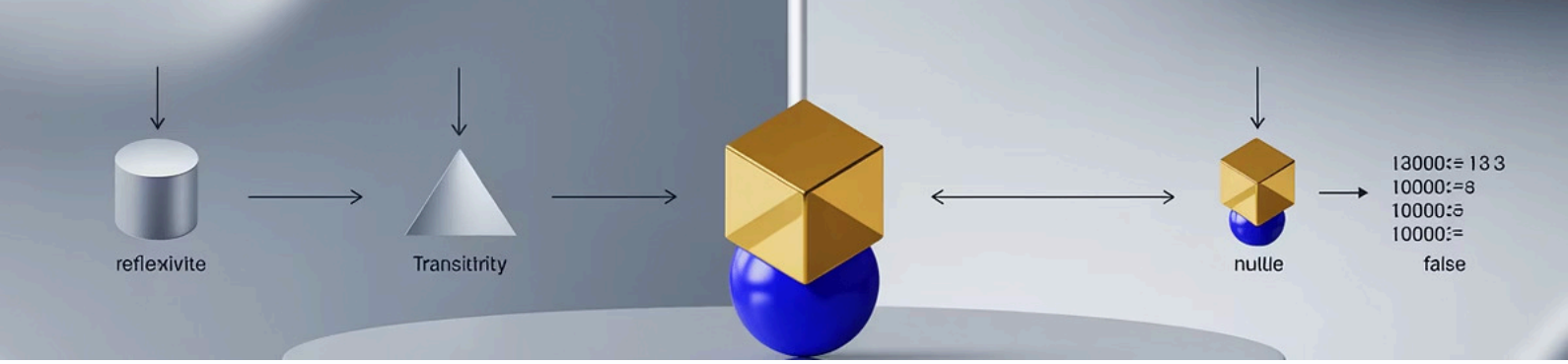
"São o mesmo objeto?"

Duas variáveis apontam para a *mesma* instância em memória. Verificada com `object.ReferenceEquals(a, b)`.

Igualdade (Valor/Semântica)

"Representam o mesmo conceito?"

Duas instâncias diferentes, mas **equivalentes segundo regras de negócio** (mesmo CPF, SKU, número de pedido). Verificada com `a.Equals(b)`.



Por Que Redefinir Equals?

Domínio Real

Dois produtos com o mesmo **SKU** são o mesmo item de catálogo, mesmo com descrições diferentes. A igualdade deve refletir a realidade do negócio.

Coleções Eficientes

`HashSet`, `Dictionary` e métodos como `Contains` dependem de `Equals`/`GetHashCode` para evitar duplicatas e localizar itens rapidamente.

Testes Claros

Testes de unidade ficam mais legíveis com regras claras de igualdade, ao invés de comparar campo por campo manualmente.

Performance e Corretude

Sem igualdade correta, você pode inserir duplicatas em estruturas de busca, degradando performance e quebrando a lógica.

Consequências de NÃO Implementar

Duplicatas Invisíveis

HashSet aceita "produtos iguais" se Equals/GetHashCode não refletirem a identidade de negócio, causando dados inconsistentes.

Busca que Falha

`lista.Contains(produto)` retorna false para um item "igual" sem igualdade adequada, gerando bugs difíceis de detectar.

Chaves Perdidas em Dicionários

Alterar um campo usado no GetHashCode após inserir em Dictionary torna a chave **irrecuperável**, causando vazamentos de memória.

Quebra do Contrato

Implementar Equals sem obedecer reflexividade, simetria, transitividade, consistência e tratamento de null gera bugs sutis e difíceis.

01

Reflexivo

`x.Equals(x)` é sempre **true**

02

Simétrico

`x.Equals(y)` ⇔ `y.Equals(x)`

03

Transitivo

Se `x.Equals(y)` e `y.Equals(z)`, então `x.Equals(z)`

04

Consistente

Múltiplas chamadas com os mesmos valores retornam o mesmo resultado

05

Tratamento de Null

`x.Equals(null)` é sempre **false**

06

Compatível com GetHashCode

Se `x.Equals(y)`, então x.GetHashCode() == y.GetHashCode()

Peças do Quebra-Cabeça da Igualdade

object.Equals(object?)

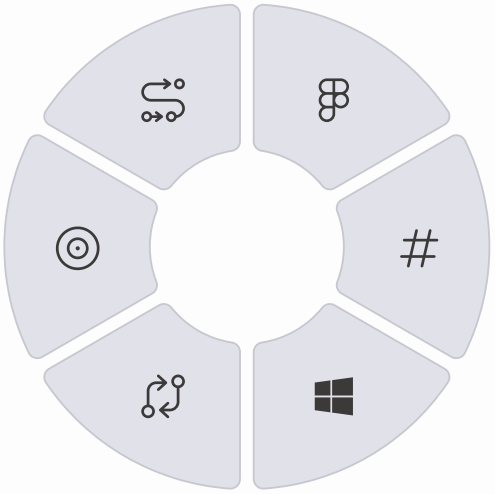
Método virtual base que você pode **sobrescrever** para definir igualdade personalizada

Tipos record

Geram **igualdade de valor** automaticamente com base nos componentes

IEqualityComparer

Comparadores externos para regras alternativas (ex: case-insensitive)



IEquatable<T>.Equals(T?)

Comparação **fortemente tipada** que evita boxing e é mais rápida

GetHashCode()

Deve ser **compatível** com Equals para uso em estruturas de hash

Operadores == / !=

Opcionais, mas quando definidos devem ser **consistentes** com Equals

Exemplos do Cotidiano

CPF

Duas carteiras com o mesmo número representam a **mesma pessoa**, mesmo que a foto seja diferente.

Placa do Veículo

Placas iguais identificam o **mesmo veículo**, ainda que a cor do carro mude.

SKU de Produto

Rótulos podem variar, mas o **SKU** identifica o item do cadastro.

Código de Barras

Duas caixas com mesmo código EAN/UPC representam o **mesmo produto** para o sistema.

Ingresso de Cinema

Dois ingressos para o mesmo assento **não são iguais** se os IDs/QR codes forem diferentes.

A regra de igualdade depende do conceito de negócio!

Boas Práticas Essenciais

Defina a Identidade

Identifique um campo ou combinação que representa a identidade do tipo e **mantenha-a imutável** após construção.

Sincronize Equals e GetHashCode

Ao sobrescrever Equals, **sempre** sobrescreva GetHashCode com a **mesma** base de comparação.

Implemente IEquatable<T>

Prefira implementar IEquatable e delegar Equals(object?) para ele, garantindo performance e type safety.

Operadores Consistentes

Se criar operadores == / !=, faça-os chamar Equals para manter consistência.

Práticas Avançadas

Tratamento de Strings

Para strings, use **StringComparison** adequados:

- `OrdinalIgnoreCase` para identificadores
- Evite dependência cultural não intencional
- Use `StringComparer` em coleções

Objetos de Valor

Para tipos como CPF, Money, Coordenada, considere **record/record struct** com igualdade automática.

Herança e Igualdade

Igualdade em hierarquias é complexa:

- Prefira tipos **sealed** ou **record**
- Projete cuidadosamente para manter contratos
- Evite alterar regras de igualdade em subclasses

Template de Implementação Segura

```
public sealed class Produto : IEquatable<Produto>
{
 public string Sku { get; } // identidade imutável
 public string Name { get; private set; }
 public decimal UnitPrice { get; private set; }

 public Produto(string sku, string name, decimal unitPrice)
 {
   if (string.IsNullOrWhiteSpace(sku))
     throw new ArgumentException("SKU inválido");
   Sku = NormalizeSku(sku);
   Name = name ?? string.Empty;
   UnitPrice = unitPrice < 0 ?
     throw new ArgumentOutOfRangeException(nameof(unitPrice)) :
       unitPrice;
 }

 private static string NormalizeSku(string s) =>
     s.Trim().ToUpperInvariant();

 // IEquatable<T>
 public bool Equals(Produto? other) =>
   other is not null && Sku == other.Sku;
```

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

 public override int GetHashCode() =>
    Sku.GetHashCode(StringComparison.Ordinal);

 public static bool operator ==(Produto? a, Produto? b) =>
    a is null ? b is null : a.Equals(b);

 public static bool operator !=(Produto? a, Produto? b) => !(a == b);
}
```

Uso Prático em Coleções

```
var a = new Produto("abc-123", "Café 250g", 18.90m);
var b = new Produto("ABC-123", "Café torrado", 19.50m);

// DEDUPLICAÇÃO: apenas 1 item
var set = new HashSet<Produto> { a, b }; // Count == 1

// USO COMO CHAVE
var estoque = new Dictionary<Produto, int> { [a] = 10 };
estoque[b] += 5; // recupera a mesma chave; total 15
```

Resultado: O HashSet contém apenas um produto, pois os SKUs são equivalentes após normalização. O Dictionary reconhece ambos os objetos como a mesma chave.

Perigo: Campos Mutáveis

⊗ Exemplo do que NÃO fazer:

```
// ANTI-PADRÃO: NÃO FAÇA ASSIM
public sealed class Tag : IEquatable<Tag>
{
    public string Codigo { get; private set; } // usado na igualdade

    public Tag(string codigo) => Codigo = codigo;
    public void Renomear(string novo) => Codigo = novo; // PERIGO!
```

```
    public bool Equals(Tag? other) =>
        other is not null && Codigo == other.Codigo;
    public override int GetHashCode() => Codigo.GetHashCode();
}

var t = new Tag("A1");
var dict = new Dictionary<Tag, string> { [t] = "ok" };
t.Renomear("B2");
var achou = dict.ContainsKey(t); // pode ser false → chave "sumiu"
```

> **Regra de Ouro:** Campos que participam da igualdade **devem ser imutáveis** após inserção em estruturas de hash.

Comparação de Strings Correta

StringComparison Adequado

```
var s1 = "arquivo";
var s2 = "ARQUIVO";

bool eq1 = string.Equals(s1, s2,
    StringComparison.OrdinalIgnoreCase); // true

bool eq2 = string.Equals(s1, s2,
    StringComparison.CurrentCulture); // pode variar

var set = new HashSet<string>(
    StringComparer.OrdinalIgnoreCase) { s1 };
set.Add(s2); // não duplica
```

> **Dica:** Para identificadores técnicos (códigos, SKUs), use **Ordinal/OrdinalIgnoreCase** para evitar problemas culturais.

Igualdade por Composição: Money

```
public readonly record struct Money(decimal Amount, string Currency)
{
    public override string ToString() => $"{Currency} {Amount:N2}";
}
```

```
var m1 = new Money(10m, "BRL");
var m2 = new Money(10m, "BRL");
var m3 = new Money(10m, "USD");

// m1 == m2 (true); m1 == m3 (false)
```

record/record struct já implementam igualdade de valor automaticamente - todos os componentes contam para a comparação.

0	100%
Linhas de Código	Compatibilidade
Necessárias para implementar igualdade com records	Com todas as regras do contrato de Equals

Objetos de Valor: CPF

```
public readonly record struct Cpf
{
    public string Number { get; }

    public Cpf(string number)
    {
        if (string.IsNullOrWhiteSpace(number))
            throw new ArgumentException("CPF inválido");

        // normaliza: só dígitos
        Number = new string(number.Where(char.IsDigit).ToArray());

        if (Number.Length != 11)
            throw new ArgumentException("CPF deve ter 11 dígitos");
    }

    public override string ToString() =>
        Convert.ToUInt64(Number).ToString(@"000\.000\.000\-00");
}

var c1 = new Cpf("123.456.789-09");
var c2 = new Cpf("12345678909");
bool iguais = c1 == c2; // true (record struct cuida da igualdade)
```

O record struct automaticamente implementa igualdade baseada no valor, normalizando diferentes formatos de entrada para a mesma representação interna.

Igualdade de Sequências

```
var a = new[] { "A", "b", "c" };
var b = new[] { "a", "B", "C" };

bool mesmaOrdem = a.SequenceEqual(b, StringComparer.OrdinalIgnoreCase); // true
```

Para comparar listas e arrays elemento por elemento, use `SequenceEqual` com o comparador apropriado.

01

Mesmo Tamanho

Verifica se as sequências têm o mesmo número de elementos

02

Mesma Ordem

Compara elementos na mesma posição usando o comparador especificado

03

Todos Iguais

Retorna true apenas se todos os pares de elementos forem iguais

Todos Iguais

Herança: Por Que É Difícil

```
public class Pessoa : IEquatable<Pessoa>
{
    public Cpf Cpf { get; }
    public string Nome { get; }

    public Pessoa(Cpf cpf, string nome) { Cpf = cpf; Nome = nome; }

    public virtual bool Equals(Pessoa? other) =>
        other is not null && Cpf == other.Cpf;
```

```
    public override bool Equals(object? obj) => Equals(obj as Pessoa);
    public override int GetHashCode() => Cpf.GetHashCode();
}

public class Aluno : Pessoa
{
    public string Ra { get; }

    public Aluno(Cpf cpf, string nome, string ra) : base(cpf, nome)
    {
        Ra = ra;
    }

    // NÃO mude a base da igualdade; senão quebra simetria/transitividade
}
```

⚠ **Sugestão:** Se a identidade é a mesma na hierarquia, **não altere** a regra nas subclasses. Prefira **tipos selados** ou **records** quando possível.

Checklist de Revisão

Identidade Definida

Qual é a **identidade** do objeto? Um campo específico ou uma composição de campos?

Imutabilidade

Os campos da identidade são **imutáveis** após a construção do objeto?

IEquatable Implementado

A classe implementa **IEquatable<T>** para comparação tipada e eficiente?

Delegação Correta

`Equals(object?)` está **delegando** para `Equals(T?)`?

GetHashCode Compatível

GetHashCode usa **as mesmas partes** que Equals para manter consistência?

Operadores Consistentes

Os operadores == / != são **consistentes** com o método Equals?

StringComparison Apropriado

Usou StringComparison adequado para comparações de string?

Casos de Coleção

Considerou o comportamento em **HashSet** e **Dictionary**?

Benefícios e Referências

Benefícios ao Aplicar Corretamente

Coerência de Domínio

O código "fala a língua do negócio" com regras claras de identidade

Coleções Eficientes

Sem duplicatas, buscas rápidas e estruturas de dados corretas

Testes Expressivos

Asserções de alto nível ao invés de comparar campo por campo

Menos Bugs

Contratos claros evitam falhas sutis e difíceis de detectar

Referências Rápidas

- `object.Equals`, `object.ReferenceEquals`
- `IEquatable<T>`, `IEqualityComparer<T>`
- `StringComparer` e `StringComparison`
- `record` e `record struct`
- `Enumerable.SequenceEqual`

> **Regra de Ouro:** Igualdade é uma decisão de modelagem. Defina-a no momento de criar o tipo e mantenha-a coerente, estável e alinhada ao domínio.