



# Presente e Futuro do C# 12

O C# é uma linguagem de programação extremamente poderosa e versátil, com uma excelente maturidade mas que continua evoluindo e atraindo novos usuários a cada dia.

Para entender um pouco mais sobre o presente e futuro da linguagem, precisamos primeiro analisar como ela começou, suas motivações e principalmente por que algumas mudanças são *extremamente* necessárias.

## Índice

[Índice](#)

[História do C#](#)

[Backward Compatibility](#)

[C# e .NET andam juntos](#)

[Como está o C#?](#)

[Por que mudar?](#)

[O que vem por aí?](#)

[Novidades do C# 12](#)

[Construtores Primários](#)

[O que é um construtor?](#)

[Language Preview](#)

[Como instalar o .NET 8](#)

[Verificando a versão instalada](#)

[O que são Primary Constructors?](#)

[Primary Constructors para classes](#)

[Herança e Primary Constructors](#)

[Construtores Parameterless e Primary Constructors](#)

[Mudanças no Using](#)

[Global Using](#)

[Implicit Using](#)

[Customizando o Global Using](#)

[Alias](#)

[Using no C# 12](#)

[Valores padrão em expressões Lambda](#)

[Conclusão](#)

[Sobre o autor](#)

[André Baltieri](#)

## História do C#



O C# teve apareceu ao mercado em meados de 2000, com a chegada de Anders Hejlsberg que saía da Borland (Empresa por trás do Delphi, uma das IDEs mais aclamadas da época).

É difícil saber o quão real é esta história, mas dizem por aí que o próprio Bill Gates foi quem contratou ele e outros engenheiros da Borland na época.

Com um convite irrecusável deste, Anders fez um pedido inusitado, criar uma nova linguagem e plataforma, que integraria (E substituiria) a família do Visual Basic.

Se é um mito ou verdade não sabemos, mas de fato uma nova plataforma foi construída, chamada de .NET por conta da movimentação que havia na época para aplicações Web.

Nasceu então o C#, uma linguagem tipada, compilada, gerenciada e orientada à objetos, com uma sintaxe mais próxima ao C/Java do que ao VB que era a principal linguagem da Microsoft na época.

## Backward Compatibility

Mas se o C# é tão antigo (23 anos), ainda vale a pena estudar esta linguagem? Não está ultrapassada? Como me mantenho atualizado sobre ela?

São ótimas perguntas e um dos pontos positivos do .NET/C# é a retrocompatibilidade, ou Backward Compatibility.

As versões do C# são bem compatíveis, ou seja, se você escrever um código que fazia no C# 3 ou 4, é bem capaz que ele ainda funcione, embora hajam melhorias que talvez tornem seu código mais limpo e legível.

Além disso, muitas das mudanças de performance, não afetam design, ou seja, seu código fica mais rápido sem necessariamente alterar nada.

O C# também tem a capacidade de compilar para diferentes versões, ou seja, mesmo em uma versão mais antiga, você pode optar por utilizar recursos mais novos durante a compilação, o que garante pelo menos a parte de performance.

## C# e .NET andam juntos

O .NET é um conjunto de bibliotecas (Framework) que suporta C# e outras linguagens do ecossistema Microsoft.

Um ponto interessante é que o C# e o .NET andam juntos, ou seja, para você utilizar o C# 11, basta instalar o .NET 7.

Então, não temos como usar o C# sem o .NET, não existe um instalador apenas do C#, ele depende do ecossistema .NET para funcionar.

Isto ocorre por conta do C# ser uma linguagem gerenciada, que significa que toda alocação de memória fica por conta de um Runtime, ou seja, nada de `malloc` e `free`.

### IMPORTANTE

Embora C# seja uma linguagem gerenciada, podemos ter código não gerenciado utilizando o escopo `unsafe`, porém a recomendação é utilizá-lo apenas quando você realmente sabe o que está fazendo.

O Runtime que gerencia o C#, também gerencia todas as linguagens do ecossistema Microsoft, o que inclui VB.NET, F# e até Cobol.NET.

Por conta desta vasta compatibilidade, quando compilamos uma aplicação C#, ela é antes traduzida para uma linguagem intermediária (Intermediate Language) e só depois gerenciada pelo Runtime.

Desta forma o Runtime ganha o nome de CLR (Common Language Runtime), já que ele gerencia todas as linguagens do ecossistema.

## Como está o C#?

Bem, obrigado! Brincadeiras a parte, o C# não parou no tempo e muitas das mudanças que vem ocorrem, são decorrentes da decisão de se tornar Open Source que ocorreu em meados de 2015.

Com código aberto, aumentou o número de contribuições e ideias para melhorar a linguagem, além de estritar a relação entre comunidade e Microsoft.

Este crescimento se estende a adoção da linguagem pelo mercado, o que dá ao C# a quinta posição no ranking do GitHub e quebra o mito que ele só é utilizado em projetos legados.

Com um .NET novo e remodelado, rodando em Windows, Mac e Linux, o C# com certeza está no catálogo de muitas grandes empresas na hora de iniciar um projeto.

## Por que mudar?



“Em time que está ganhando, não se mexe” correto? Só que não! Ouvi do próprio Mads Torgersen, que é o responsável pelo C# hoje, que eles querem melhorar e modernizar ainda mais.

Particularmente acredito que precisamos nos modernizar e ter a mente aberta. O tempo todo novas pessoas estão chegando no mundo da programação, temos que atendê-las também.

É necessário manter o “velho” que funciona, mas também ter a mente aberta para novas funcionalidades e perspectivas de linguagens modernas.

Recursos como o Top-Level Statement que vieram com C# 10 são fundamentais para isto. Implicit Using, melhorias nas funções Lambda e por aí vai. São recursos que tornam o C# mais fácil e simples de utilizar.

## O que vem por aí?

Performance e simplicidade. O .NET segue no topo dos Frameworks com mais performance, no topo mesmo, perdendo em alguns cenários apenas para o Rust e sendo até 10x mais rápido que o NodeJs.

A principal imagem do .NET 8, que sai em Novembro de 2023 é um carro de corrida, ou seja, mostra o comprometimento com a performance.

Do lado do design, melhorias no trabalho com listas, construtores primários, ainda mais suporte a using e melhorias nas lambdas também trazem novidades de design.

Novamente, são funcionalidades opcionais, que você pode testar hoje (Só seguir neste paper) ou manter seu código como está. Você não é obrigado a acompanhar, mas eu recomendo fortemente.

Para as próximas versões do C# eu espero mais facilidades, mais simplicidade, mantendo a maturidade e performance que o C# sempre teve.

## Novidades do C# 12

O C# 12 está previsto para ser lançado em Novembro de 2023, junto com o .NET 8, mas já tivemos uma prévia das suas novidades no Preview 4 do .NET 8 e vamos conferir aqui.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
</Project>
```

## Construtores Primários

Se você já trabalha com C# e está atualizado sobre as melhorias na linguagem, provavelmente já utilizou o recurso de construtores primários nas `structs` e/ou `records`.

Porém, no C# 12 teremos esta novidade inclusa também nas classes. Isto mesmo, agora podemos simplificar muito a criação de classes utilizando construtores primários.

```
// Exemplo de construtor primário
// em Records
public record Person(int Id, string Name);

// Exemplo de construtor primário
// em Structs
public struct Person(int Id, string Name);

// Exemplo de construtor primário
// em Classes (* novidade do C# 12)
public class Person(int Id, string Name);
```

## O que é um construtor?

Construtor ou Constructor em inglês é o nome dado a um método especial de uma classe, que é executado toda vez que um objeto com base nesta classe é criado e que obrigatoriamente deve ter o mesmo nome da classe.

Então, supondo que temos uma classe chamada `Name`, podemos ter um **Construtor** chamado `Name` nesta classe.

```
public class Name
{
    public Name()
    {
        // Método Construtor
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Desta forma, toda vez que instanciarmos um objeto a partir desta classe, o método `Name` será chamado. Note que ele não tem retorno e pode ou não conter parâmetros.

```
var name = new Name();  
// Neste momento o construtor é chamado  
  
var name = new Name {  
    FirstName = "André",  
    LastName = "Baltieri",  
};  
// Outro exemplo
```

Sempre que criamos um construtor sem parâmetros, ele recebe o nome de ***Parameterless***, que significa exatamente **Sem Parâmetros** em inglês, porém, é plenamente possível criar um construtor com parâmetros.

```
public class Name  
{  
    public Name(string firstName, string lastName)  
    {  
        FirstName = firstName;  
        LastName = lastName;  
    }  
  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
}
```

O único detalhe é que uma vez definido um (E apenas um) construtor com parâmetros, somos obrigados a informar os valores já na instância do objeto.

```
var name = new Name();  
// ❌ ERRO - Precisa informar os valores  
  
var name = new Name("André", "Baltieri");  
// ✅ OK - Informou os valores
```

Por fim, podemos combinar construtores, tendo mais de um para nossas classes, incluindo um com e outro sem parâmetros.

```
public class Name  
{  
    public Name() { }  
  
    public Name(string firstName, string lastName)  
    {  
        FirstName = firstName;  
        LastName = lastName;  
    }  
  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
}
```

No exemplo acima, temos uma sobrecarga de método, dando a possibilidade de informar ou não argumentos na chamada do método.

```
var name = new Name();  
// ✅ OK - Passa pelo Parameterless  
  
var name = new Name("André", "Baltieri");  
// ✅ OK - Informou os valores
```

É comum também, vermos construtores como o modificador `protected` ao invés do `public`. Isto ocorre por que o Entity Framework por exemplo, exige um construtor `parameterless` para instância de objetos.

```
public DbSet<Product> Products { get; set; }  
// ❌ - Vai falhar se Product não tiver um construtor parameterless
```

Entretanto, uma boa recomendação é explicitar no construtor da sua classe, quais propriedades são obrigatórias para mesma funcionar.

Para resolver esta situação, podemos utilizar o `protected` no construtor `parameterless`, afinal ele só será usado por uma classe proxy que derivará da classe base no EF.

```
public class Name  
{  
    // 📌 Agora é protected  
    protected Name() { }  
  
    public Name(string firstName, string lastName)  
    {  
        FirstName = firstName;  
        LastName = lastName;  
    }  
  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
}
```

Desta forma podemos evitar que nossa classe seja instanciada sem valores for do seu escopo, mas ainda permitimos que classes filhas faça uso do construtor `parameterless`.

```
var name = new Name();  
// ❌ Erro - Não pode instanciar sem passar valores  
  
var name = new Name("André", "Baltieri");  
// ✅ OK - Informou os valores  
  
public DbSet<Product> Products { get; set; }  
// ✅ - Proxy das classes vão funcionar
```

Para finalizar, podemos fazer uso dos `Optional Parameters`, para ter um efeito parecido com os de cima.

```
public class Measurement  
{  
    public Measurement(  
        int width = 0, // 📌 Parâmetro opcional, já possui um valor  
        int height = 0) // 📌 Parâmetro opcional, já possui um valor  
    {  
    }  
  
    public int Width { get; set; }  
    public int Height { get; set; }  
}
```

Isto torna a instância do objeto mais maleável, podendo ou não informar os valores definidos no construtor.

```
// Inicia com os valores padrão  
var measurement = new Measurement();
```

```
// Inicia com 10 para width e 0 para height
var measurement = new Measurement(10);

// Inicia com 10 para width e 10 para height
var measurement = new Measurement(10, 10);
```

De qualquer forma, as possibilidades são muitas, vai do que você precisa para o seu cenário e de como quer tratar a instância do seu objeto.

## Language Preview

É importante salientar que este recurso está disponível no .NET 8 Preview 3, ou seja é apenas uma pré-visualização do que poderá ser este recurso.

Já vimos casos no passado, como o `!!` que foram removidos da versão final, então não se apegue muito aos novos recursos em si até a versão final.

É importante frisar também que o .NET 8 só sai em Novembro de 2023, muita coisa pode mudar até lá e você precisa instalar a versão Preview 3 ou superior para testar estas funcionalidades.

## Como instalar o .NET 8

Você pode fazer o download do .NET 8 no site oficial da Microsoft, utilizando a URL abaixo.

 <https://dotnet.microsoft.com/en-us/download/dotnet/8.0>

## Verificando a versão instalada

Com o .NET instalado, basta fechar seus terminais e abrir novamente, seguido pela execução do comando abaixo para verificar a versão instalada.

```
dotnet --version
## 8.0.100-preview.3.23178.7
```

Além disso, o .NET 8 ainda mantém o C# 11 como linguagem padrão, então é preciso habilitar uma funcionalidade chamada **Language Preview** no arquivo `.csproj` do seu projeto.

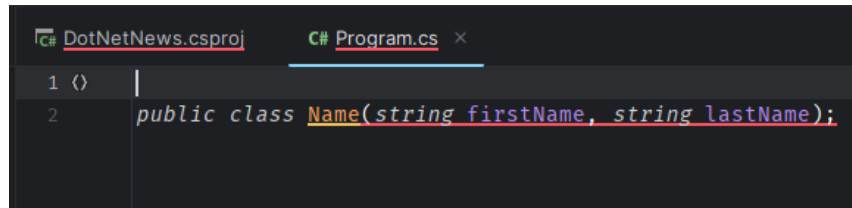
Isto é feito pela configuração chamada **LangVersion** cujo temos que atribuir o valor `preview`, como mostrado abaixo.

```
<Project Sdk="Microsoft.NET.Sdk">

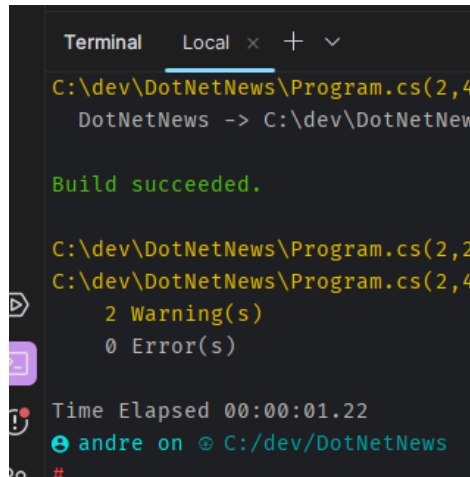
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework> ☞ Garantir que o .NET está na V8
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <LangVersion>preview</LangVersion> ☞ Adiciona esta linha
  </PropertyGroup>

</Project>
```

Além disso, provavelmente sua IDE não vai reconhecer os comandos novos, identificando eles como possíveis erros.



Mas o Build não mente, e como podemos ver, mesmo com a IDE informando que a sintaxe não é válida, nosso programa compila corretamente.



## O que são Primary Constructors?

Legal, mas supondo que eu tenha uma classe com duas propriedades apenas, porém, estas propriedades devem ser obrigatórias, meu código ficaria assim?

```
public class Name
{
    public Name(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Pois é! Muito código para algo realmente simples, não acha? Inclusive, é sempre uma boa recomendação ver se os **records** ou **structs** podem fazer o trabalho pela sua classe.

```
public record Name(string FirstName, string LastName);
public struct Name(string FirstName, string LastName);
```

Como podemos ver acima, tanto os records quanto os structs possuem um recurso chamado **Primary Constructors** ou construtores primários.



Este recurso permite que sequer precisemos abrir chaves para criação da classe, ou seja, tudo é definido na estrutura inicial.

```
var name = new Name();  
// ❌ ERRO - Precisa informar os valores  
  
var name = new Name("André", "Baltieri");  
// ✅ OK - Informou os valores
```

Então, um construtor primário, nada mais é do que a possibilidade de explicitar alguns valores obrigatórios já na definição da classe, simplificando o código.

Desta forma, no C# 12, teremos a inclusão dos **Primary Constructors** para classes, tornando a sintaxe ainda mais enxuta.

```
// Podemos usar a sintaxe reduzida  
public class Name(string FirstName, string LastName);  
  
// Ou mesmo expandir e ter comportamento na classe  
public class Name(string FirstName, string LastName)  
{  
  
}
```

Como resultado, temos um comportamento similar aos anteriores, onde somos obrigados a informar os valores na instância do objeto.

```
var name = new Name();  
// ❌ CS7036: There is no argument given that corresponds to the required parameter 'FirstName' of 'Name.Name(string, string)'  
  
var name = new Name("André", "Baltieri");  
✅ - OK
```

## Primary Constructors para classes

Existe uma diferença nos construtores primários para as classes em relação a mesma funcionalidade para records e structs.

No caso dos Records e Struts, ao utilizar o recurso de construtor primário temos automaticamente uma propriedade pública com o mesmo nome, criada de forma automática.

```
public record Name(string FirstName, string LastName);  
  
var name = new Name("André", "Baltieri");  
Console.WriteLine(name.FirstName);  
Console.WriteLine(name.LastName);
```

Neste mesmo exemplo, ao alterar de record para class, mudamos o comportamento, visto que os parâmetros dos **Primary Constructors** são identificados como membros privados em uma classe.

### IMPORTANTE

Este comportamento pode ser alterado até a versão final do C# 12 que está prevista para Novembro de 2023.

```
// 🔄 utilizando class ao invés de record  
public class Name(string FirstName, string LastName);
```

```
var name = new Name("André", "Baltieri");
Console.WriteLine(name.FirstName); // ✗ FirstName não existe
Console.WriteLine(name.LastName); // ✗ LastName não existe
```

Para ter o mesmo comportamento de um `record` ou `struct`, precisamos exportar os argumentos, explicitando o acesso dos mesmos com o modificador `public`.

```
public class Name(string firstName, string lastName)
{
    public string FirstName => firstName;
    public string LastName => lastName;
}
```

Desta forma, agora temos dois parâmetros de entrada, que são privados por padrão e consequentemente dois membros para saída, que são públicos.

```
var name = new Name("André", "Baltieri");
Console.WriteLine(name.FirstName); // ✓
Console.WriteLine(name.LastName); // ✓
```

Note também que não há necessidade de nomear os parâmetros com inicial maiúscula, já que eles sempre serão privados.

```
public record Name(string FirstName, string LastName);
// Recomendo usar maiúscula já que são membros públicos

var name = new Name();
Console.WriteLine(name.FirstName);
Console.WriteLine(name.LastName);

// No caso das classes o comportamento inverte
// recomendo usar minúscula pois são privados
public class Name(string firstName, string lastName)
{
    public string FirstName => firstName;
    public string LastName => lastName;
}

var name = new Name();
Console.WriteLine(name.FirstName);
Console.WriteLine(name.LastName);
```

## Herança e Primary Constructors

Caso haja necessidade de herdar de uma classe que contém um **Primary Constructor**, precisamos fornecer os dados para mesma, assim como já fazemos hoje com construtores que não são *parameterless*.

```
public class Student(string firstName, string lastName, int grade)
    : Name(firstName, lastName)
{
}
```

Note que ao utilizar a herança “:” somos obrigados a informar o `firstName` e `lastName`, e neste caso, precisamos receber eles também no `Student`.

Um ponto importante e interessante aqui, é que a classe `Student` possui uma propriedade a mais, chamada `Grade` e que não está sendo exportada, logo, esta propriedade não poderá ser acessada.

```
var student = new Student("André", "Baltieri", 10);

Console.WriteLine(student.FirstName); // ✓ - Publica no Name
Console.WriteLine(student.LastName); // ✓ - Publica no Name
Console.WriteLine(student.Grade); // ✗ - Somente Student conhece
```

## Construtores Parameterless e Primary Constructors

Outro ponto interessante que testei foi o cenário onde temos a herança de uma classe com um construtor primário mas queremos ter um construtor **parameterless**.

Mais especificamente, vamos supor que a classe `Name` tenha um construtor primário mas a classe `Student` que herda dela tenha um construtor **parameterless**.

```
public class Student(string firstName, string lastName, int grade)
    : Name(firstName, lastName)
{
    // Construtor com valores padrões
    public Student() : this("First", "Last", 5)
    {
    }

    public int Grade => grade;
}
```

Neste caso, podemos utilizar o `this` para inicializar o construtor **parameterless** e informar os valores padrão para `Name`, visto que eles são obrigatórios.

## Mudanças no Using

Nós organizamos nossos projetos em C# em pastas (Físico) e **Namespaces** (Lógico), ou seja, um **Namespace** nada mais é do que uma separação lógica, virtual, do nosso código.

Feita esta divisão, podemos utilizar o `using` para ter acesso ao código de um determinado **Namespace**.

Podemos por exemplo, fazer uso do `using` para referenciar outros projetos, namespaces ou mesmo para simplificar o uso de alguns tipos, como por exemplo:

```
System.Console.WriteLine("Hello World");
```

Neste caso acima, o famoso `Console.WriteLine` que usamos, está dentro do namespace `System`, e usando o `using System`, encurtamos o caminho.

```
using System;

Console.WriteLine("Hello world");
```

## Global Using

Desde do C# 10 há uma funcionalidade chamada **Global Usings** que nos permite importar namespaces que são de uso comum (Que tem muito uso) em um arquivo na raiz da aplicação, comumente chamado de **GlobalUsings.cs**.

Para utilizar esta funcionalidade, basta criar um arquivo na raiz do seu projeto (Se quiser nomear ele como **GlobalUsings.cs** fica legal) e adicionar os **usings** que você deseja com prefixo `global`.

```
global using System;
```

Desta forma, o namespace `System` será automaticamente importado em todos os arquivos do projeto, dispensando aquela linha de código extra.

```
// using System; <!-- ✗ não precisa mais  
Console.WriteLine("Hello world");
```

Você pode importar quantos namespaces quiser, só precisa ficar atento pelo fato deles serem importados em todos os arquivos, incluindo arquivos em sub-pastas.

Se você não pretende usar o namespace com muita frequência, pode optar pelo caminho normal, sem o **Global Using**.

## Implicit Using

Além do recurso de **Global Using**, outra funcionalidade inclusa desde o C# 10 é o **Implicit Using**, que vem habilitado por padrão em projetos .NET 6 ou superior através da seguinte configuração no `.csproj`.

```
<ImplicitUsings>enable</ImplicitUsings>
```

Com esta funcionalidade habilitada, o .NET irá gerar um arquivo em tempo de compilação via **Source Generator** contendo os seguintes **Global Usings**.

```
// <auto-generated/>  
global using global::System;  
global using global::System.Collections.Generic;  
global using global::System.IO;  
global using global::System.Linq;  
global using global::System.Net.Http;  
global using global::System.Threading;  
global using global::System.Threading.Tasks;
```

Note que este arquivo contém uma linha descrevendo que ele foi gerado automaticamente e não devemos modificá-lo.

### Source Generated Files

Arquivos que contém a extensão “.g.cs” são gerados por um recurso do .NET chamado Source Generator, que permite o compilador gerar arquivos adicionais para otimizar nosso código.

No caso, como meu projeto de exemplo se chamava **CsharpTwelve**, o arquivo gerado foi `CsharpTwelve.GlobalUsings.g.cs`, que pode ser localizado neste caminho:

```
obj/Debug/net8.0/CsharpTwelve.GlobalUsings.g.cs
```

### IMPORTANTE

Estas importações mudam de acordo com o projeto gerado. No caso do Console são estas exibidas acima, no caso do Web serão outras e assim por diante.

## Customizando o Global Using

Nós também podemos customizar as importações que o .NET fará através de uma configuração no `.csproj`.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>

  <!-- Adiciona ou remove as importações-->
  <ItemGroup>
    <Using Remove="System.IO"/>
    <Using Include="System.Text"/>
  </ItemGroup>

</Project>
```

No caso, utilizamos o `Include` para adicionar um namespace e o `Remove` para excluir um namespace do **Global Usings**. Como resultado temos o seguinte código:

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
// global using global::System.IO; ❌ REMOVIDO
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Text; // ➡️ ADICIONADO
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

## Alias

Um alias é como um apelido, um nome que podemos utilizar em diferentes lugar no C#. `string` (Com S minúsculo) por exemplo é um Alias para `String` (com S maiúsculo).

No caso dos `using`, podemos criar um alias para facilitar o entendimento de alguma importação, como por exemplo substituir o `Console` por `Terminal`.

```
using Terminal = System.Console;
Terminal.WriteLine(x.Units);
```

Os alias também são uteis quando temos o mesmo nome de um type replicado em diferentes namespaces, como no caso dos `Commands` do `CQRS`.

```
using Balta.Core.Account.UseCases.Create.Command;
using Balta.Core.Account.UseCases.Update.Command;
using Balta.Core.Account.UseCases.Delete.Command;
```

Podemos então criar um alias para cada um deles e assim identificar melhor cada ação.

```
using CreateCommand = Balta.Core.Account.UseCases.Create.Command;
using UpdateCommand = Balta.Core.Account.UseCases.Update.Command;
using DeleteCommand = Balta.Core.Account.UseCases.Delete.Command;
```

```
var createCommand = new CreateCommand();  
await _handler.HandleAsync(createCommand);
```

## Using no C# 12

O C# 12 pretende expandir ainda mais esta funcionalidade, tornando possível criar alias para tipos primitivos, arrays, nullables e até tuplas.

```
using Measurement = (string Units, int Distance);  
using PathOfPoints = int[];  
using DatabaseInt = int?;  
  
Measurement x = new Measurement("inchs", 3);
```

Nas novas versões será possível criar um alias associando um tipo imediatamente, como por exemplo:

```
using DurationInMinutes = int;  
DurationInMinutes duration = 10;
```

Além disso, o uso de nullables e arrays também é permitido, mas o melhor é o uso de Tuplas e quem sabe Records ou Structs no futuro.

```
using Name = (string FirstName, string LastName);  
  
var name = new Name("André", "Baltieri");  
Console.WriteLine(name.FirstName); // André
```

## Valores padrão em expressões Lambda

Expressões Lambda é um recurso indispensável no C# e que utilizamos diariamente. Não só as listas vem evoluindo como as expressões também.

A partir do C# 12 (Preview 3 do .NET 8), podemos definir um valor padrão para uma expressão lambda como mostrado abaixo.

```
var addWithDefault = (int addTo = 2) => addTo + 1;  
addWithDefault(); // 3  
addWithDefault(5); // 6
```

Como os valores ficam no metadata, eles podem ser recuperados de forma fácil caso necessário.

```
var addWithDefault = (int addTo = 2) => addTo + 1;  
addWithDefault.Method.GetParameters()[0].DefaultValue; // 2
```

## Conclusão

O C# tem muita história para contar mas está longe de ser uma linguagem que parou no tempo, e o melhor de tudo é que possui uma retrocompatibilidade incrível.

E aí, bora aprender mais sobre C#? 🚀

## Sobre o autor



### **André Baltieri**

Me dedico ao desenvolvimento de software desde 2003, sendo minha maior especialidade o Desenvolvimento Web. Durante esta jornada pude trabalhar presencialmente aqui no Brasil e Estados Unidos, atender remotamente times da Índia, Inglaterra e Holanda, receber **10x Microsoft MVP** e realizar diversas consultorias em empresas e projetos de todos os tamanhos.