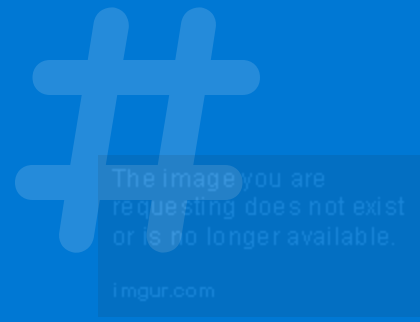
Fundamentos de .NET e C#

Uma introdução completa ao desenvolvimento com a plataforma .NET e a linguagem C# para iniciantes e desenvolvedores



O que é .NET (Framework vs Core)

Evolução e características da plataforma de desenvolvimento

.NET Framework

Framework original lançado em 2002, exclusivo para Windows, com foco em aplicações desktop e Web Forms.

.NET Core (Atual .NET)

Plataforma moderna, open source, multiplataforma (Windows, Linux, macOS) lançada em 2016, otimizada para cloud e containers.

Característica	.NET Framework	.NET Core/.NET
Plataformas	Windows	Windows, Linux, macOS
Open Source	Parcial	Completo
Performance	Bom	Superior
Versão Atual	4.8.1 (Final)	.NET 8.0 (2023)

Evolução da Plataforma

Em 2020, o .NET 5 unificou o ecossistema, eliminando a distinção entre Core e Framework. Hoje, existe apenas ".NET", seguindo uma versão principal por ano (6, 7, 8...).

CLR e CLI - Arquitetura Interna

Entendendo o núcleo da plataforma .NET

Common Language Runtime (CLR)

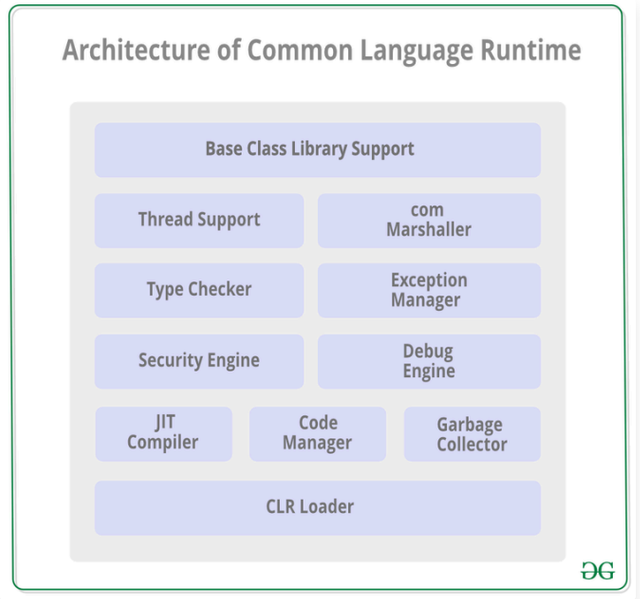
Ambiente de execução que gerencia código, memória e executa o garbage collection para aplicações .NET.

Common Language Infrastructure (CLI)

Especificação que permite múltiplas linguagens (C#, VB.NET, F#) compilarem para o mesmo bytecode (IL).

Componentes Principais

- CTS (Common Type System): Sistema de tipos unificado
- JIT Compiler: Converte IL para código nativo
- Garbage Collector: Gerencia memória automaticamente



Processo de Execução

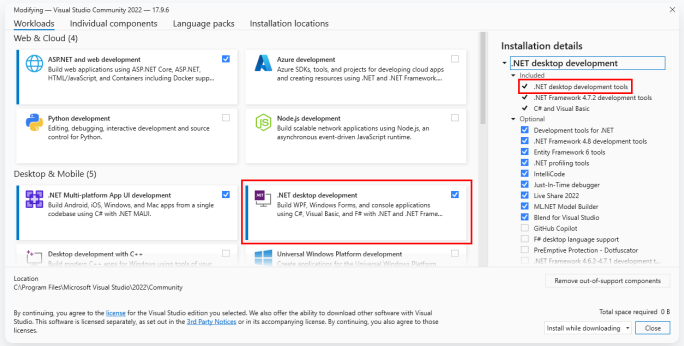
1. Código compilado para IL (Intermediate Language)
2. IL empacotado em assemblies (.dll/.exe)
3. CLR executa o IL via JIT em tempo de execução

Instalação do Ambiente (Visual Studio/VS Code)

Configurando seu ambiente de desenvolvimento para .NET

Visual Studio

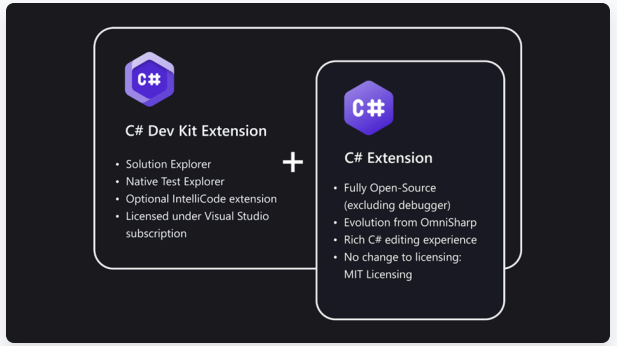
IDE completa com ferramentas integradas para desenvolvimento .NET



1. Baixe o Visual Studio no site da Microsoft
2. Selecione workload "**.NET Desktop Development**"
3. Opcional: "ASP.NET and Web Development"

Visual Studio Code

Editor leve e extensível para múltiplas plataformas



1. Instale o VS Code (code.visualstudio.com)
2. Instale o .NET SDK (dotnet.microsoft.com)
3. Extensão: "**C# Dev Kit**" no VS Code

Dica

Verifique a instalação executando **dotnet --version** no terminal após a configuração.

Primeiro Hello World (Console App)

Criando e entendendo sua primeira aplicação console em C#

Código de exemplo do Hello World:

```
using System;

namespace MeuPrimeiroApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadKey(); // Aguarda tecla
        }
    }
}
```

Como executar:

1. No terminal: `dotnet run`
2. No VS: Pressione F5

using System

Importa namespaces com funcionalidades básicas do .NET.

namespace

Organiza o código e evita conflitos de nomes.

Main method

Ponto de entrada da aplicação. O runtime inicia aqui.

Console.WriteLine()

Exibe texto no console e move para próxima linha.

Tipos de Valor vs Tipos de Referência

Como a memória é gerenciada em C#

Stack (Rápido, Tamanho Fixo)

Heap (Dinâmico, Gerenciado pelo GC)

Tipos de Valor

Armazenados diretamente na **Stack**. Contêm o valor real.

Exemplos:

`int` `float` `bool` `struct` `enum`

```
int x = 10;

int y = x; // y é cópia (10)

x = 20; // y continua 10
```

Tipos de Referência

Armazenados no **Heap**. A variável contém apenas um endereço.

Exemplos:

`class` `string` `array` `interface` `object`

```
Person p1 = new Person();

Person p2 = p1; // mesma referência

p1.Name = "João"; // afeta ambos
```

Importância na Prática

Entender essa diferença é essencial para performance, passagem de parâmetros e comportamento do código. Tipos de valor são mais rápidos, enquanto tipos de referência oferecem mais flexibilidade.

Operadores em C#

Aritméticos, Lógicos e de Comparação

± Operadores Aritméticos

Operador	Descrição	Exemplo
`+`	Adição	`int soma = 5 + 3;`
`-`	Subtração	`int diff = 10 - 4;`
`*`	Multiplicação	`int prod = 3 * 5;`
`/`	Divisão	`int div = 10 / 2;`

≠ Operadores de Comparação

Operador	Descrição	Exemplo
`==`	Igual a	`if (idade == 18)`
`!=`	Diferente de	`if (resp != "S")`
`<`	Menor que	`if (temp < 0)`
`>`	Maior que	`if (valor > 100)`

Operadores Lógicos

Operador	Descrição	Exemplo
`&&`	E lógico	`if (a > 0 && b > 0)`
`||`	OU lógico	`if (a > 0 || b > 0)`
`!`	Negação	`if (!condicao)`

</> Uso em Expressões

Tipo	Exemplo
Expressão aritmética	`double media = (nota1 + nota2) / 2;`
Expressão condicional	`string status = (media >= 7) ? "Aprovado" : "Reprovado";`

Estruturas Condicionais (if, else, switch)

Controlando o fluxo do programa com decisões lógicas

if / else / else if

Estrutura para tomada de decisões baseada em condições.

```
int idade = 18;

if (idade >= 18)

Console.WriteLine("Maior de idade");

else

Console.WriteLine("Menor de idade");
```

Operador Ternário

Forma concisa para condições simples:

```
string msg = idade >= 18 ? "Adulto" : "Menor";
```

switch / case

Compara uma expressão com múltiplos casos possíveis.

```
int dia = 3;

switch (dia) {

case 1:

Console.WriteLine("Domingo");

break;

case 3:

Console.WriteLine("Terça");

break;

default:

Console.WriteLine("Outro dia");

break;

}
```

Dicas

Use **if** para condições lógicas e **switch** para comparar uma variável com valores constantes.

Escopo de Variáveis

Como a acessibilidade de variáveis é determinada em C#

Escopo de Classe

Variáveis declaradas no nível da classe (campos ou propriedades) são acessíveis em toda a classe.

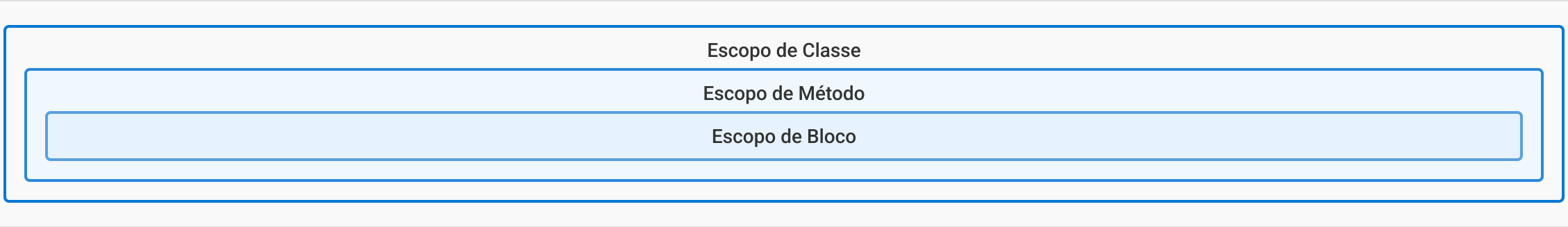
Escopo de Método

Variáveis declaradas dentro de um método só são acessíveis dentro deste método.

Escopo de Bloco

Variáveis declaradas dentro de um bloco (entre chaves {}) são acessíveis apenas neste bloco e blocos aninhados.

```
class Programa {
  int escopo_classe = 10; // Acessível em toda a classe
  void Metodo() {
    int escopo_metodo = 20; // Acessível só no método
    if (true) {
      int escopo_bloco = 30; // Acessível só no bloco if
    }
  }
}
```



Laços de Repetição (for, while, do-while)

Estruturas para executar código repetidamente

Laços de repetição permitem executar um bloco de código múltiplas vezes, conforme uma condição específica:

Loop for

Útil quando se conhece o número de iterações.

```
for (int i = 0; i < 5; i++) {
Console.WriteLine(i);
}
```

Inicializador; Condição; Incremento

Loop while

Executa enquanto uma condição for verdadeira.

```
int j = 0;
while (j < 5) {
Console.WriteLine(j++);
}
```

Verifica a condição antes de executar

Loop do-while

Executa pelo menos uma vez, depois verifica.

```
int k = 0;
do {
Console.WriteLine(k++);
} while (k < 5);
```

Garante ao menos uma execução

Quando usar cada um

for: Quando você conhece o número exato de iterações (arrays, coleções).

while: Quando a condição pode mudar a qualquer momento e você não sabe quantas iterações ocorrerão.

do-while: Quando precisa executar o código pelo menos uma vez, independente da condição.

Instruções de Salto (break, continue)

Alterando o fluxo de execução em laços

break

Interrompe a execução do laço completamente, saindo imediatamente.

```
for (int i = 0; i < 10; i++)
{
if (i == 5) break;
Console.WriteLine(i);
}
// Saída: 0 1 2 3 4
```

Quando usar:

Quando encontrar o que procura ou precisar sair do loop.

continue

Pula para a próxima iteração, ignorando o código restante.

```
for (int i = 0; i < 10; i++)
{
if (i % 2 == 0) continue;
Console.WriteLine(i);
}
// Saída: 1 3 5 7 9
```

Quando usar:

Para pular elementos que não atendem a certos critérios.

Diferenças principais

O **break** encerra o laço, enquanto o **continue** pula para a próxima iteração. Funcionam em todos os tipos de laço (for, foreach, while, do-while).

Instrução	Efeito	Casos de uso
break	Termina o laço	Encontrar elemento, condições de erro, otimização
continue	Pula para próxima iteração	Filtrar elementos, ignorar casos específicos

Métodos - Conceitos e Assinatura

Organização e modularização de código em C#

O que são Métodos?

Blocos de código reutilizáveis que executam tarefas específicas. Fundamentais para organizar a lógica em unidades coesas.

Anatomia de um Método

`tipo de retorno` `nome do método` (`lista de parâmetros`)

Assinatura do Método

Inclui nome e lista de parâmetros (tipos e ordem). O tipo de retorno **não** faz parte da assinatura.

Tipo de Retorno

Define o tipo de dado retornado. Use void quando não há retorno.

```
// Retorna um inteiro
int Somar(int a, int b) {
return a + b;
}
// Não retorna valor
void ExibirMensagem(string msg) {
Console.WriteLine(msg);
}
```

Boas Práticas

• Nomes descritivos (CalcularTotal) • Responsabilidade única • Limitar parâmetros (3-4 máx) • Documentação clara

Parâmetros (Por Valor e Por Referência)

Diferentes formas de passar argumentos para métodos em C#

Por Valor padrão

O método recebe uma **cópia** da variável. Modificações dentro do método não afetam a variável original.

Por Referência ref

O método recebe uma **referência** à variável original. A variável deve ser inicializada antes da chamada do método.

Por Saída out

Similar ao ref, mas a variável **não precisa** ser inicializada antes. O método deve obrigatoriamente atribuir um valor ao parâmetro.

```
Por Valor (padrão)
int x = 10; Alterar(x); Console.WriteLine(x); // Saída: 10 (não muda)
void Alterar(int n) { n = 20; }
```

```
Por Referência (ref)
int y = 10; AlterarRef(ref y); Console.WriteLine(y); // Saída: 20
(valor alterado) void AlterarRef(ref int n) { n = 20; }
```

```
Por Saída (out)
int z; // Não precisa inicializar ObterValor(out z);
Console.WriteLine(z); // Saída: 42 void ObterValor(out int n) { n =
42; } // Deve atribuir valor
```

Característica	Por Valor	ref	out
Inicialização da variável	Opcional	Obrigatória	Não necessária
Atribuição no método	Opcional	Opcional	Obrigatória
Altera variável original	Não	Sim	Sim

Overloading de Métodos

Múltiplas versões do mesmo método com assinaturas diferentes

O que é Overloading (Sobrecarga)?

Permite criar múltiplos métodos com o mesmo nome, mas parâmetros diferentes. O compilador determina qual versão chamar com base nos argumentos fornecidos.

```
class Calculadora {
public int Somar(int a, int b) {
return a + b;
}
public int Somar(int a, int b, int c) {
return a + b + c;
}
public double Somar(double a, double b) {
return a + b;
}
}
```

```
var calc = new Calculadora();
int r1 = calc.Somar(5, 10); // Primeira versão: 15
int r2 = calc.Somar(5, 10, 15); // Segunda versão: 30
double r3 = calc.Somar(2.5, 7.5); // Terceira versão: 10.0
```

Regras de Assinatura

A assinatura inclui nome e lista de parâmetros (tipos e ordem), mas **não** inclui o tipo de retorno.

> **Importante:** Métodos com mesmos tipos de parâmetros mas diferentes retornos **não são** sobrecargas válidas!

Quando Usar

Ideal para realizar a mesma operação com diferentes tipos ou quantidades de dados, mantendo a clareza do código.

Benefícios

Melhora legibilidade, reduz duplicação de código e permite APIs mais flexíveis com opções para diferentes cenários.

Conclusão e Próximos Passos

Resumo e caminhos para aprofundar seus conhecimentos

Tópicos Abordados

- .NET Framework, .NET Core/Modern, CLR e CLI
- Configuração do ambiente com Visual Studio e VS Code
- Hello World em Console App
- Tipos de valor e referência, operadores
- Estruturas condicionais, laços e escopo
- Métodos, parâmetros e overloading

Próximos Passos

Aprofundamento em POO

Classes, objetos, herança, polimorfismo, encapsulamento, interfaces e classes abstratas

Entity Framework Core

Modelagem de dados, migrations, LINQ e operações CRUD

ASP.NET Core

Desenvolvimento web com MVC, Web API e Blazor

Continue aprendendo! O ecossistema .NET é vasto e oferece diversas oportunidades profissionais. Pratique com projetos pessoais, participe da comunidade e consulte a documentação oficial da Microsoft para se manter atualizado.