Manual da 3935_2/N Programação em C#

Adriana Teles

7 Janeiro 2025

Este manual de formação em C# foi desenvolvido para capacitar os formandos com uma base sólida na linguagem de programação, abordando desde os fundamentos até técnicas avançadas. Destina-se a indivíduos sem experiência prévia em programação e proporciona um percurso progressivo e prático. O conteúdo inclui conceitos fundamentais, como Programação Orientada a Objetos (POO), estruturas de dados, organização de código, e técnicas avançadas. A finalidade é preparar os participantes para aplicarem as competências adquiridas no desenvolvimento de aplicações modernas, robustas e escaláveis em C#.

Índice

1. Introdução

1.1. Objetivo Geral

O principal objetivo deste curso é dotar os participantes de competências sólidas em programação utilizando a linguagem C#. Este curso pretende desenvolver habilidades práticas e teóricas que permitam aos formandos conceber, implementar e manter aplicações modernas, robustas e escaláveis. No final do curso, os formandos deverão estar aptos a:

- Compreender os conceitos fundamentais da linguagem C#.
- Desenvolver aplicações orientadas a objetos, aplicando boas práticas de programação.
- Explorar técnicas avançadas para melhorar a eficiência e a modularidade do código.
- Integrar ferramentas e frameworks do ecossistema .NET no desenvolvimento de software.

1.2. Estrutura do Curso

Este curso foi desenhado para garantir um progresso gradual no domínio da linguagem C#, com uma abordagem prática e orientada a projetos. A sua estrutura organiza-se da seguinte forma:

1. **Introdução ao Ambiente de Desenvolvimento:**
   - Configuração do ambiente de trabalho com o .NET Framework e Visual Studio.
   - Familiarização com as ferramentas essenciais para desenvolvimento em C#.
2. **Fundamentos da Linguagem C#:**
   - Aprendizagem dos conceitos básicos, como tipos de dados, operadores, controlo de fluxo e gestão de variáveis.
3. **Programação Orientada a Objetos (POO):**
   - Introdução a conceitos como classes, objetos, encapsulamento, herança e polimorfismo.
   - Aplicação prática destes conceitos no desenvolvimento de software modular.
4. **Estruturas de Dados:**
   - Manipulação de arrays, propriedades e indexadores para organização eficiente de informações.
5. **Organização de Código:**
   - Utilização de namespaces e implementação de operadores e eventos personalizados para código limpo e reutilizável.
6. **Técnicas Avançadas:**
   - Introdução a métodos avançados, genéricos, Garbage Collector e criação de atributos personalizados.

7. **Projeto Prático:**
    - Desenvolvimento de um projeto que consolida todos os conhecimentos adquiridos.

1.3. Metodologia

O curso combina métodos de ensino teórico e prático, incluindo:

- **Exposições teóricas:** Para apresentar os conceitos essenciais de cada módulo.
- **Exercícios práticos:** Para reforçar os conteúdos abordados, promovendo a aprendizagem ativa.
- **Projeto final:** Para aplicação integrada de todos os tópicos num cenário realista.

Com esta estrutura, os formandos terão acesso a um percurso de aprendizagem dinâmico e eficaz, permitindo-lhes consolidar as suas competências ao longo das 50 horas do curso.

2. Ambiente de Desenvolvimento

2.1. .NET Framework

O **.NET Framework** é uma plataforma de desenvolvimento criada pela Microsoft para suportar a criação de aplicações em diversas linguagens, incluindo C#. Esta plataforma fornece um ambiente robusto para a execução de programas e inclui ferramentas que simplificam o desenvolvimento e a manutenção de software.

2.1.1. Características principais:

- **Common Language Runtime (CLR):** O CLR é o núcleo do .NET Framework, responsável pela execução de programas .NET. Ele gere a memória, segurança, e tratamento de exceções, garantindo que o código seja executado de forma eficiente e segura.
- **Base Class Library (BCL):** Um vasto conjunto de bibliotecas que disponibilizam funcionalidades como acesso a ficheiros, manipulação de strings e conectividade a bases de dados.
- **Multi-Linguagem:** Suporte para múltiplas linguagens de programação, como C#, VB.NET e F#.

Exemplo prático:

O seguinte programa demonstra uma aplicação simples em C# utilizando o .NET Framework:

```
using System;

class Program {
    static void Main() {
        Console.WriteLine("Bem-vindo ao .NET Framework!");
    }
}
```

2.1.2. Como funciona:

- O método `Main` é o ponto de entrada do programa.

- `Console.WriteLine` faz parte da biblioteca BCL e é usado para escrever no terminal.

2.2. Visual Studio .NET

O **Visual Studio** é a principal IDE (Ambiente de Desenvolvimento Integrado) para desenvolvimento em C#. Esta ferramenta oferece funcionalidades que tornam o processo de programação mais rápido, eficiente e organizado.

Funcionalidades principais:

- **IntelliSense:** Um sistema de sugestão de código que ajuda a escrever de forma mais rápida e evita erros comuns.
- **Debugging:** Ferramentas avançadas para depurar aplicações e corrigir erros de execução.
- **Templates de Projetos:** Facilita a criação de diferentes tipos de aplicações, como aplicações de consola, desktop ou web.

Configuração do Ambiente:

1. **Instalar o Visual Studio:** Certifique-se de instalar a versão mais recente do Visual Studio.
2. **Configurar o projeto:** Crie um novo projeto selecionando o template "Aplicação de Consola (.NET Framework)".
3. **Escrever código:** Adicione o código de exemplo acima na função `Main`.
4. **Executar:** Pressione `F5` para executar o programa.

Exemplo prático com depuração:

```
using System;

class Program {
    static void Main() {
        int numero = 10;
        Console.WriteLine("O número é: " + numero);
        Console.WriteLine("Fim do programa.");
    }
}
```

Passos para depurar:

1. Coloque um **breakpoint** na linha `Console.WriteLine("O número é: " + numero);` clicando na margem esquerda da linha.
2. Execute o programa em modo de depuração pressionando `F5`.
3. Inspecione o valor da variável `numero` quando o programa parar no breakpoint.
4. Use `F10` para avançar para a próxima linha de código.

Benefícios do .NET Framework e Visual Studio

- **Eficiência no desenvolvimento:** Ferramentas integradas para escrita, teste e depuração de código.
- **Segurança:** Gestão de exceções e controlo de acesso robustos.
- **Flexibilidade:** Suporte a vários tipos de aplicações e linguagens de programação.

3. Fundamentos de C#

3.1. Memória: Stack e Heap

A gestão de memória é um conceito essencial em C#, uma vez que a eficiência e o desempenho de um programa estão diretamente ligados a como os dados são armazenados e manipulados. Em C#, a memória está dividida em dois principais locais: **stack** e **heap**.

3.1.1. Stack

A stack é uma área de memória organizada como uma pilha (stack) de dados. Os dados armazenados aqui são geralmente de tamanho fixo e têm tempos de vida previsíveis. Cada vez que uma função é chamada, um bloco de memória é alocado na stack para armazenar variáveis locais e parâmetros.

- **Características:**
  - Acesso rápido.
  - Memória alocada e desalocada automaticamente.
  - Usada para armazenar tipos simples (Value-Type) e referências para objetos.

```
int a = 10; // Alocado na stack
```

3.1.2. Heap

A heap é uma área de memória usada para armazenar dados que têm tamanhos dinâmicos ou tempos de vida incertos. Objetos e outros Reference-Types são alocados aqui.

- **Características:**
  - Acesso mais lento em comparação com a stack.
  - Memória gerida pelo Garbage Collector.
  - Usada para armazenar objetos criados com new.

```
string nome = new string("João"); // Alocado na heap
```

Diferenças entre Stack e Heap

Característica	Stack	Heap
Velocidade	Mais rápida	Mais lenta
Alocação de memória	Automática	Dinâmica
Dados armazenados	Variáveis locais e tipos simples	Objetos e dados complexos

Compreender a diferença entre stack e heap é essencial para otimizar o uso da memória e identificar potenciais problemas de desempenho.

3.2. Tipos de Dados (Value-Type e Reference-Type)

Os tipos de dados em C# podem ser divididos em dois grandes grupos: **Value-Type** e **Reference-Type**. Estes conceitos são fundamentais para compreender como a linguagem gere a memória e o desempenho das aplicações.

3.2.1. Value-Type

Os Value-Type armazenam os seus dados diretamente na memória stack. Este tipo de dados é mais eficiente em termos de desempenho, uma vez que a stack é mais rápida que o heap. Exemplos de Value-Type incluem:

- `int` (inteiros)
- `float` (números de ponto flutuante)
- `bool` (valores booleanos)
- `struct` (estruturas)

```
int idade = 25;
bool ativo = true;
Console.WriteLine($"Idade: {idade}, Ativo: {ativo}");
```

3.2.2. Reference-Type

Os Reference-Type armazenam uma referência ao valor real, que está localizado na memória heap. São usados para tipos mais complexos, como objetos e strings. Exemplos de Reference-Type incluem:

- `string`
- `class`
- `object`

```
string nome = "João";
Console.WriteLine($"Olá, {nome}!");
```

3.3. Operadores

Os operadores em C# são símbolos que realizam operações em variáveis ou valores. Podem ser divididos em várias categorias:

- **Aritméticos:** `+`, `-`, `*`, `/`, `%`
- **Relacionais:** `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Lógicos:** `&&`, `||`, `!`
- **Atribuição:** `=`, `+=`, `-=`, `*=`, `/=`

```
int a = 10, b = 20;
Console.WriteLine(a + b); // Soma: 30
Console.WriteLine(a > b); // Resultado: False
```

3.4. Declaração de Variáveis

As variáveis são usadas para armazenar dados temporariamente na memória enquanto o programa está a ser executado. Em C# é necessário declarar o tipo de dado que a variável armazenará.

```
int idade = 30;
double altura = 1.75;
string nome = "Ana";
Console.WriteLine($"Nome: {nome}, Idade: {idade}, Altura: {altura}");
```

3.5. Statements e Controle de Fluxo

Os statements de controlo de fluxo permitem alterar o curso de execução de um programa com base em condições ou repetições.

3.5.1. `if` e `else`

```
int numero = 10;
if (numero > 5) {
    Console.WriteLine("Maior que 5");
} else {
    Console.WriteLine("Menor ou igual a 5");
}
```

3.5.2. switch

```
string dia = "segunda";
switch (dia) {
    case "segunda":
        Console.WriteLine("Início da semana!");
        break;
    case "sexta":
        Console.WriteLine("Fim da semana!");
        break;
    default:
        Console.WriteLine("Dia normal");
        break;
}
```

3.5.3. Ciclos (for, while, do-while)

3.5.3.1. for:

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

3.5.3.2. while:

```
int contador = 0;
while (contador < 3) {
    Console.WriteLine($"Contador: {contador}");
    contador++;
}
```

3.5.3.3.do-while:

```
int x = 0;
do {
    Console.WriteLine($"Valor de x: {x}");
    x++;
} while (x < 2);
```

4. Programação Orientada a Objetos (POO)

A Programação Orientada a Objetos (POO) é um dos paradigmas mais importantes na programação moderna. Este paradigma organiza o software em torno de objetos, que combinam dados e comportamentos numa única estrutura.

4.1. Definições Fundamentais

4.1.1. Classe:

Uma classe é um modelo ou estrutura que define os atributos (dados) e métodos (comportamentos) de um objeto. É o ponto de partida para a criação de objetos em POO.

```
class Pessoa {
    public string Nome;
    public int Idade;

    public void Apresentar() {
        Console.WriteLine($"Olá, o meu nome é {Nome} e tenho {Idade} anos.");
    }
}
```

4.1.2. Objeto:

Um objeto é uma instância de uma classe. Representa uma entidade única com os dados e comportamentos definidos pela classe.

Exemplo:

```
Pessoa pessoa = new Pessoa();
pessoa.Nome = "João";
pessoa.Idade = 30;
pessoa.Apresentar();
```

4.1.3. Atributo:

Um atributo é uma variável que representa as características de uma classe.

Exemplo: `Nome` e `Idade` são atributos da classe `Pessoa` acima.

4.1.4. Método:

Um método é uma função definida dentro de uma classe que implementa os comportamentos dos objetos.

Exemplo: `Apresentar` é um método da classe `Pessoa`.

4.1.5. Superclasse e Subclasse:

Superclasse: Uma classe que serve como base para outras classes.

Subclasse: Uma classe que herda atributos e métodos de uma superclasse.

```
class Animal {
    public void Comer() {
        Console.WriteLine("Este animal está a comer.");
    }
}

class Cachorro : Animal {
    public void Latir() {
        Console.WriteLine("Au au!");
    }
}
```

4.1.6. Interface:

Uma interface é um contrato que define métodos que uma classe deve implementar. As interfaces são usadas para garantir que diferentes classes partilhem a mesma estrutura.

```
interface IVeiculo {
    void Mover();
}

class Carro : IVeiculo {
    public void Mover() {
        Console.WriteLine("O carro está a mover-se.");
    }
}
```

4.2. Principais Conceitos de POO

4.2.1. Encapsulamento:

Refere-se à ocultação dos detalhes internos de uma classe, expondo apenas o que é necessário através de métodos ou propriedades. Ajuda a proteger os dados e a manter o código modular.

```
class ContaBancaria {
    private double saldo;

    public void Depositar(double valor) {
        saldo += valor;
    }

    public void ConsultarSaldo() {
        Console.WriteLine($"Saldo: {saldo}");
    }
}
```

4.2.2. Herança:

Permite que uma classe reutilize os atributos e métodos de outra classe, promovendo a reutilização de código e reduzindo a redundância.

```
class Animal {
    public void Comer() {
        Console.WriteLine("Este animal está a comer.");
    }
}

class Cachorro : Animal {
    public void Latir() {
        Console.WriteLine("Au au!");
    }
}
```

4.2.3. Polimorfismo:

Permite que diferentes classes usem métodos com o mesmo nome, mas com comportamentos distintos, dependendo do contexto.

```
class Forma {
    public virtual void Desenhar() {
        Console.WriteLine("Desenhar uma forma genérica.");
    }
}

class Circulo : Forma {
    public override void Desenhar() {
        Console.WriteLine("Desenhar um círculo.");
    }
}
```

4.2.4. Abstração:

Enfatiza os aspectos essenciais de uma entidade, ignorando detalhes desnecessários. Em C#, pode ser implementada com classes abstratas ou interfaces.

Exemplo:

```
abstract class Veiculo {
    public abstract void Mover();
}

class Carro : Veiculo {
    public override void Mover() {
        Console.WriteLine("O carro está a mover-se.");
    }
}
```

5. Estruturas de Dados

As estruturas de dados são fundamentais na programação, pois permitem armazenar, organizar e manipular dados de forma eficiente. Em C#, temos várias opções de estruturas, sendo os **arrays**, **propriedades** e **indexadores** algumas das mais utilizadas.

5.1. Arrays

Os arrays são coleções de elementos do mesmo tipo, armazenados em posições contíguas na memória. Cada elemento é identificado por um índice, começando pelo zero.

Características principais:

- Podem ser unidimensionais, multidimensionais ou jagged (arrays de arrays).
- São úteis para armazenar conjuntos de dados relacionados.

Exemplo de array unidimensional:

```
int[] numeros = { 1, 2, 3, 4, 5 };

foreach (int numero in numeros) {
    Console.WriteLine(numero);
}
```

Saída:

1
2
3
4
5

Exemplo de array multidimensional:

```
int[,] matriz = {
    { 1, 2, 3 },
    { 4, 5, 6 }
};

Console.WriteLine(matriz[1, 2]); // Saída: 6
```

Exemplo de array jagged:

```
int[][] jaggedArray = new int[2][];
jaggedArray[0] = new int[] { 1, 2 };
jaggedArray[1] = new int[] { 3, 4, 5 };

Console.WriteLine(jaggedArray[1][2]); // Saída: 5
```

5.2. Propriedades

As propriedades são membros de uma classe que fornecem uma interface para acessar ou modificar os atributos. Elas permitem encapsular a lógica de acesso e proteger os dados internos da classe.

Exemplo:

```
class Produto {
    private double preco;

    public double Preco {
        get { return preco; }
        set {
            if (value > 0) {
                preco = value;
            } else {
                Console.WriteLine("Preço inválido.");
            }
        }
    }
}

class Program {
    static void Main() {
        Produto produto = new Produto();
        produto.Preco = 50.0;
        Console.WriteLine(produto.Preco);
    }
}
```

Saída:

50

5.3. Indexadores

Os indexadores permitem que uma classe seja indexada como se fosse um array. São úteis para criar coleções personalizadas.

Exemplo:

```
class Colecao {
    private string[] itens = new string[10];

    public string this[int index] {
        get { return itens[index]; }
        set { itens[index] = value; }
    }
}

class Program {
    static void Main() {
        Colecao colecao = new Colecao();
        colecao[0] = "Item 1";
        colecao[1] = "Item 2";

        Console.WriteLine(colecao[0]); // Saída: Item 1
        Console.WriteLine(colecao[1]); // Saída: Item 2
    }
}
```

6. Organização de Código

A organização de código em C# é essencial para manter o código limpo, modular e fácil de manter. Esta secção aborda conceitos como **namespaces**, **operadores personalizados** e **eventos**, que são fundamentais para estruturar aplicações de forma eficiente.

6.1. Namespaces

Os **namespaces** permitem organizar o código em agrupamentos lógicos, evitando conflitos de nomes em projetos grandes e facilitando a reutilização de código.

Exemplo de namespace:

```
namespace Aplicacao.Exemplo {
    class MinhaClasse {
        public void MostrarMensagem() {
            Console.WriteLine("Namespace a funcionar!");
        }
    }
}

class Program {
    static void Main() {
        Aplicacao.Exemplo.MinhaClasse obj = new Aplicacao.Exemplo.MinhaClasse();
        obj.MostrarMensagem();
    }
}
```

Saída:

Namespace a funcionar!

Benefícios dos Namespaces:

- Organização do código em módulos.
- Prevenção de conflitos de nomes.
- Facilitação da navegação e manutenção do código.

6.2. Operadores Personalizados

Os operadores personalizados permitem redefinir o comportamento de operadores como +, -, *, etc., para tipos definidos pelo programador. Isto é útil para tornar classes mais intuitivas e expressivas.

Exemplo:

```
class Complexo {
    public int Real { get; set; }
    public int Imaginario { get; set; }

    public static Complexo operator +(Complexo c1, Complexo c2) {
        return new Complexo {
            Real = c1.Real + c2.Real,
            Imaginario = c1.Imaginario + c2.Imaginario
        };
    }

    public override string ToString() {
        return $"{Real} + {Imaginario}i";
    }
}

class Program {
    static void Main() {
        Complexo c1 = new Complexo { Real = 1, Imaginario = 2 };
        Complexo c2 = new Complexo { Real = 3, Imaginario = 4 };

        Complexo resultado = c1 + c2;
        Console.WriteLine(resultado);
    }
}
```

Saída:

4 + 6i

6.3. Eventos

Os **eventos** são mecanismos que permitem notificar outras partes do programa quando uma ação específica ocorre. São frequentemente usados no padrão de design Publisher/Subscriber.

Exemplo:

```
using System;

class Notificador {
    public event Action AlgoAconteceu;

    public void AcionarEvento() {
        Console.WriteLine("Evento a ser acionado...");
        AlgoAconteceu?.Invoke();
    }
}

class Program {
    static void Main() {
        Notificador notificador = new Notificador();

        notificador.AlgoAconteceu += () => Console.WriteLine("O evento foi acionado!");

        notificador.AcionarEvento();
    }
}
```

Saída:

Evento a ser acionado...

O evento foi acionado!

Benefícios dos Eventos:

- Permitem comunicação eficiente entre diferentes partes do programa.
- Facilitam a implementação de padrões de design como Observer.

7. Técnicas Avançadas

Nesta secção, exploramos técnicas avançadas que ampliam as capacidades da linguagem C# e permitem criar aplicações mais eficientes e modulares. Estas incluem o uso de **métodos e parâmetros**, **variáveis reference-type**, **métodos estáticos e de extensão**, **atributos personalizados**, o **Garbage Collector**, e **genéricos**.

7.1. Métodos e Parâmetros

Os métodos são blocos de código que realizam uma tarefa específica. Os parâmetros permitem passar dados para os métodos, tornando-os mais flexíveis e reutilizáveis.

Tipos de Parâmetros:

1. **Valor (Default):** Passa uma cópia do valor para o método.
2. **Referência (`ref`):** Passa a referência para a variável original, permitindo a sua modificação.
3. **Saída (`out`):** Passa um valor de saída do método.
4. **Parâmetros variáveis (`params`):** Permitem passar um número variável de argumentos.

Exemplo:

```
class Program {
    static void Somar(params int[] numeros) {
        int soma = 0;
        foreach (int numero in numeros) {
            soma += numero;
        }
        Console.WriteLine($"Soma: {soma}");
    }

    static void Main() {
        Somar(1, 2, 3, 4); // Saída: Soma: 10
    }
}
```

7.2. Variáveis Reference-Type

Além dos Value-Type, as Reference-Type armazenam referências a objetos na memória heap. Classes, strings e arrays são exemplos comuns de Reference-Type.

Exemplo:

```
class Program {
    static void AlterarTexto(ref string texto) {
        texto = "Texto alterado!";
    }

    static void Main() {
        string textoOriginal = "Texto original";
        AlterarTexto(ref textoOriginal);
        Console.WriteLine(textoOriginal); // Saída: Texto alterado!
    }
}
```

7.3. Métodos Estáticos e de Extensão

7.3.1. Métodos Estáticos

Os métodos estáticos pertencem a uma classe e não a uma instância específica. São úteis para funcionalidades que não dependem do estado do objeto.

Exemplo:

```
class Matematica {
    public static int Somar(int a, int b) {
        return a + b;
    }
}

class Program {
    static void Main() {
        Console.WriteLine(Matematica.Somar(5, 10)); // Saída: 15
    }
}
```

7.3.2. Métodos de Extensão

Permitem adicionar funcionalidades a classes existentes sem modificar o código original.

Exemplo:

```
public static class Extensoes {
    public static int Dobrar(this int numero) {
        return numero * 2;
    }
}

class Program {
    static void Main() {
        int numero = 5;
        Console.WriteLine(numero.Dobrar()); // Saída: 10
    }
}
```

7.4. Atributos Personalizados

Os atributos permitem associar metadados a classes, métodos e outros elementos do código. Podem ser usados para fornecer informações adicionais ao compilador ou ao runtime.

Exemplo:

```
[Obsolete("Este método será removido na próxima versão.")]
static void MetodoAntigo() {
    Console.WriteLine("Método antigo.");
}

class Program {
    static void Main() {
        MetodoAntigo(); // Gera um aviso de obsolescência
    }
}
```

7.5. Garbage Collector (GC)

O Garbage Collector é responsável por gerir automaticamente a memória em C#, libertando objetos que não são mais utilizados.

Exemplo:

```
class Program {
    static void Main() {
        CriarObjetos();
        GC.Collect(); // Força a execução do Garbage Collector
        Console.WriteLine("GC executado.");
    }

    static void CriarObjetos() {
        for (int i = 0; i < 1000; i++) {
            var obj = new object();
        }
    }
}
```

7.6. Genéricos

Os genéricos permitem criar classes, métodos e interfaces que funcionam com qualquer tipo de dado, aumentando a reutilização de código e a segurança de tipo.

Exemplo:

```
class Caixa<T> {
    private T conteudo;

    public void Colocar(T item) {
        conteudo = item;
    }

    public T Retirar() {
        return conteudo;
    }
}

class Program {
    static void Main() {
        Caixa<int> caixaDeInteiros = new Caixa<int>();
        caixaDeInteiros.Colocar(10);
        Console.WriteLine(caixaDeInteiros.Retirar()); // Saída: 10

        Caixa<string> caixaDeStrings = new Caixa<string>();
        caixaDeStrings.Colocar("Olá Mundo");
        Console.WriteLine(caixaDeStrings.Retirar()); // Saída: Olá Mundo
    }
}
```

Glossário de termos

Atributo	Uma variável definida dentro de uma classe que armazena o estado ou as propriedades de um objeto.
Classe	Um modelo ou estrutura que define os atributos (dados) e métodos (comportamentos) de um objeto. Serve como a base para criar objetos.
Encapsulamento	O conceito de esconder os detalhes internos de uma classe, permitindo o acesso apenas através de métodos ou propriedades públicas.
Evento	Mecanismo que permite notificar partes do programa quando uma ação específica ocorre. Usado no padrão Publisher/Subscriber.
Garbage Collector (GC)	Componente do runtime do .NET responsável por gerir automaticamente a memória, removendo objetos que já não são utilizados.
Genéricos	Recurso que permite criar classes, métodos ou interfaces que funcionam com diferentes tipos de dados, promovendo reutilização de código
Heap	Área de memória onde objetos e dados dinâmicos são armazenados. Utilizado por tipos de referência (Reference-Type).
Herança	Mecanismo que permite que uma classe derive atributos e métodos de outra classe.
Indexador	Recurso que permite que uma classe seja acedida como se fosse um array.
Interface	Um contrato que define métodos que uma classe deve implementar, promovendo padronização e flexibilidade no design.
Método	Uma função definida dentro de uma classe que implementa o comportamento dos objetos.
Namespace	Agrupamento lógico de classes e outros elementos para organizar o código e evitar conflitos de nomes.
Objeto	Uma instância de uma classe que representa uma entidade com atributos e métodos definidos pela classe.

Operador Personalizado	Redefinição de operadores padrão para tipos definidos pelo programador.
Polimorfismo	Capacidade de métodos com o mesmo nome se comportarem de forma diferente dependendo do contexto.
Propriedade	Membro de uma classe que fornece uma interface para ler ou modificar os atributos de forma controlada.
Reference-Type	Tipos que armazenam uma referência ao valor, localizado na memória heap.
Stack	Área de memória usada para armazenar variáveis locais e tipos simples (Value-Type). Acesso rápido e memória alocada automaticamente.
Subclasse	Classe que herda atributos e métodos de uma superclasse.
Superclasse	Classe que serve como base para outras classes.
Value-Type	Tipos que armazenam os seus dados diretamente na memória stack.

Bibliografia

1. Microsoft Documentation. "C# Language Reference." Acesso em https://learn.microsoft.com/en-us/dotnet/csharp/.
2. Albahari, Joseph, e Albahari, Ben. "C# 10 in a Nutshell." O'Reilly Media, 2022.
3. Richter, Jeffrey. "CLR via C#." Microsoft Press, 2012.
4. Skeet, Jon. "C# in Depth." Manning Publications, 2019.
5. Troelsen, Andrew, e Japikse, Philip. "Pro C# 10 and the .NET 6 Platform." Apress, 2022.
6. Freeman, Adam. "Essential C# 8.0." Apress, 2020.
7. Pluralsight. "C# Fundamentals." Acesso em https://www.pluralsight.com/.
8. Stack Overflow. "C# Questions and Answers." Acesso em https://stackoverflow.com/questions/tagged/c%23.
9. Skeet, Jon. "C# Documentation Best Practices." Artigo publicado no site https://csharpindepth.com.