Link do vídeo no youtube: https://youtu.be/N7O_LwOK8BI

Alunos: Guilherme Soranzo, Lucas H. Giraldi, Victor G. Hermann

Introdução à Linguagem de Programação C#

A linguagem de programação C# (pronunciada como "C sharp") é uma das linguagens mais populares e amplamente usadas no mundo do desenvolvimento de software. Criada pela Microsoft, a linguagem C# foi introduzida em 2000 como parte da plataforma .NET, e desde então tem desempenhado um papel crucial na criação de uma ampla gama de aplicativos, desde aplicativos de desktop até aplicativos móveis e serviços web.

1. Principais Características da Linguagem C#

1.1 Orientação a Objetos

A linguagem C# é fortemente orientada a objetos, o que significa que ela suporta conceitos como classes, objetos, herança, polimorfismo e encapsulamento. Isso facilita a criação de código modular e reutilizável.

1.2 Tipagem Estática

C# é uma linguagem de tipagem estática, o que significa que você deve declarar explicitamente o tipo de uma variável antes de usá-la. Isso ajuda a identificar erros de tipo em tempo de compilação, tornando o código mais robusto.

1.3 Gerenciamento Automático de Memória

C# utiliza um coletor de lixo (garbage collector) para gerenciar automaticamente a alocação e liberação de memória, tornando mais fácil evitar vazamentos de memória e simplificando o gerenciamento de recursos.

1.4. Bibliotecas Poderosas

A linguagem C# possui uma rica biblioteca de classes, conhecida como .NET Framework (ou .NET Core, dependendo da versão), que oferece uma ampla gama de funcionalidades para desenvolvimento de software, desde manipulação de arquivos até comunicação de rede.

1.5 Multithreading

C# oferece suporte a programação multithread, permitindo que os desenvolvedores criem aplicativos que executem várias tarefas simultaneamente, melhorando o desempenho e a responsividade.

2. Ambientes de Desenvolvimento

Existem vários ambientes de desenvolvimento disponíveis para programar em C#, incluindo:

2.1 Visual Studio

A IDE (Ambiente de Desenvolvimento Integrado) da Microsoft, conhecida como Visual Studio, oferece uma experiência de desenvolvimento completa para C#. Ela inclui um editor de código avançado, depuração, criação de interface gráfica e outras ferramentas úteis.

2.2 Visual Studio Code

Uma alternativa mais leve ao Visual Studio é o Visual Studio Code, uma IDE de código aberto que também oferece suporte sólido para desenvolvimento em C#.

3. Forma de Compilação e Execução

C# é uma linguagem compilada, o que significa que o código-fonte é traduzido em código de máquina antes da execução. O processo de compilação envolve a transformação do código C# em um arquivo executável (geralmente com extensão .exe) ou em uma biblioteca de classes (DLL) que pode ser usada por outros programas.

O processo de compilação pode ser realizado usando ferramentas como o compilador da linguagem C# (csc.exe) ou diretamente a partir de uma IDE, como o Visual Studio.

A execução de um programa C# ocorre quando o arquivo executável é iniciado pelo usuário ou por outro software. O código de máquina gerado é interpretado pelo sistema operacional e executado.

3.1 Linguagem Compilada ou Interpretada

Como mencionado anteriormente, C# é uma linguagem compilada. Isso significa que o código-fonte é traduzido para código de máquina antes da execução. No entanto, o .NET Framework também inclui uma máquina virtual chamada Common Language Runtime (CLR), que ajuda na execução do código compilado, fornecendo recursos como gerenciamento de memória e coleta de lixo.

Portanto, pode-se dizer que C# está em um ponto intermediário entre linguagens puramente compiladas e linguagens puramente interpretadas.

4. O Surgimento da Linguagem C#

A história da linguagem C# começa no final da década de 1990, quando a Microsoft estava em busca de uma linguagem moderna e poderosa para fazer parte da plataforma .NET, que tinha como objetivo revolucionar a forma como aplicativos de software eram desenvolvidos e executados.

Nesse contexto, a equipe de desenvolvimento liderada por Anders Hejlsberg, um renomado engenheiro de software dinamarquês, foi encarregada de criar uma linguagem que fosse robusta, segura e eficiente. A equipe tinha em mente a ideia de construir uma linguagem que aproveitasse o melhor das linguagens existentes e que fosse especialmente adequada para desenvolver aplicativos Windows e serviços web.

4.1 Influências na Criação da Linguagem

Ao criar o C#, a equipe de desenvolvimento de Anders Hejlsberg se inspirou em várias linguagens de programação existentes, incluindo:

4.1.1 C e C++

O C# herdou muitos conceitos de linguagens como C e C++, incluindo a sintaxe básica e o uso de ponteiros.

4.1.2 Java

O C# também foi influenciado pelo Java em termos de segurança, portabilidade e gerenciamento automático de memória. Isso tornou o C# uma alternativa viável para o desenvolvimento de aplicativos em plataforma cruzada.

4.1.3 Delphi

A experiência da equipe de desenvolvimento com a linguagem Delphi, que era usada para desenvolver aplicativos Windows, também influenciou a criação do C#. O Delphi era conhecido por sua facilidade de desenvolvimento de aplicativos de desktop, e a equipe queria replicar essa facilidade no C#.

5. Estruturas de Controle Simples em C#

5.1 Leitura de dados

Para ler dados do usuário a partir do teclado, você pode utilizar a classe *Console*, que faz parte do *namespace System*. O método Console.*ReadLine*() é usado para ler uma linha de texto da entrada padrão (geralmente o teclado) e armazená-la em uma variável.

```
Console.WriteLine("Digite um número:");
string input = Console.ReadLine();
int numero = int.Parse(input);
```

Imagem 1 - Exemplo leitura de dados

5.2 Escrita de dados

Para exibir dados na tela, você pode usar o método console.WriteLine(), que escreve uma linha de texto no console.

```
int resultado = 42;
Console.WriteLine("O resultado é: " + resultado);
```

Imagem 2 - Exemplo escrita de dados

5.3 Atribuição de dados

A atribuição de dados em C# é feita usando o operador de atribuição =. Você pode atribuir valores a variáveis de diferentes tipos de dados, como inteiros, strings, booleanos, etc.

```
int idade = 30;
string nome = "Alice";
bool eEstudante = true;
```

Imagem 3 - Exemplo atribuição de dados

5.4 Estrutura condicional

A estrutura condicional em C# permite que você tome decisões com base em condições. O bloco *if*, juntamente com as cláusulas *else if* e *else*, é usado para criar estruturas condicionais.

```
int idade = 18;

if (idade < 18)
{
    Console.WriteLine("Você é menor de idade.");
}
else if (idade >= 18 && idade < 60)
{
    Console.WriteLine("Você é adulto.");
}
else
{
    Console.WriteLine("Você é um idoso.");
}
```

Imagem 4 - Exemplo condições

5.5 Estrutura de repetição

A estrutura de repetição em C# permite que você execute um bloco de código repetidamente enquanto uma condição for verdadeira. Duas estruturas de repetição comuns são o *for* e o *while*.

```
int contador = 0;

while (contador < 5)
{
    Console.WriteLine("Contador: " + contador);
    contador++;
}
```

Imagem 5 - Exemplo estrutura WHILE

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

Imagem 6 - Exemplo estrutura FOR

6. Tipos de Dados Primitivos

C# possui uma série de tipos de dados primitivos, que são usados para armazenar valores simples, como números inteiros, números de ponto flutuante, caracteres e valores booleanos. Alguns dos tipos de dados primitivos mais comuns incluem:

6.1 Int

Armazena números inteiros, como 1, 2, -3, 0, etc.

Exemplo:

```
int idade = 25;
```

6.2 float e double

Armazenam números de ponto flutuante, como 3.14, 2.71828, etc.

Exemplo:

```
float pi = 3.14f;
double e = 2.71828;
```

6.3 Char

Armazena um único caractere, como 'A', 'b', '$', etc.

Exemplo:

```
char letra = 'A';
```

6.4 Bool

Armazena valores booleanos, verdadeiro (true) ou falso (false).

Exemplo:

```
bool isTrue = true;
```

7. Variáveis Compostas do Tipo Produto Cartesiano (Registros)

Em C#, as variáveis compostas do tipo produto cartesiano são representadas por estruturas (structs). As estruturas permitem que você crie tipos de dados personalizados que consistem em várias variáveis de diferentes tipos.

Exemplo de declaração de uma estrutura em C#:

```
struct Pessoa
{
public string Nome; public int Idade;
}
```

Uso da estrutura:

```
Pessoa pessoa1;
pessoa1.Nome = "Alice";
pessoa1.Idade = 30;
```

8. Variáveis Compostas do Tipo Mapeamento (Vetores e Matrizes)

C# oferece suporte a variáveis compostas do tipo mapeamento, como vetores (arrays) e matrizes.

Exemplo de declaração de um vetor em C#:

```
int[] numeros = new int[5]; // Vetor de inteiros com 5 elementos
numeros[0] = 1;
numeros[1] = 2;
```

Exemplo de declaração de uma matriz em C#:

```
int[,] matriz = new int[3, 3]; // Matriz 3x3 de inteiros
matriz[0, 0] = 1;
matriz[0, 1] = 2;
```

9. Elaboração de Ponteiros

Em C#, a elaboração de ponteiros é restrita e só é possível em contextos de código não gerenciado usando a palavra-chave `unsafe`. O uso de ponteiros é considerado avançado e não é comum na programação C#.

Exemplo de uso de ponteiro (em um contexto `unsafe`):

```
unsafe
{
int numero = 42;
int* ponteiroParaNumero = &numero;
Console.WriteLine(*ponteiroParaNumero); // Imprime o valor apontado pelo
ponteiro
}
```

10. Lista de Palavras-Reservadas da Linguagem

C# possui várias palavras-reservadas que não podem ser usadas como identificadores (nomes de variáveis, métodos, etc.) em seu código. Algumas das palavras-reservadas em C# incluem:

- class: Define uma classe.
- int: Tipo de dado inteiro.
- string: Tipo de dado para armazenar cadeias de caracteres.
- if: Início de uma estrutura condicional.
- else: Parte de uma estrutura condicional.
- for: Início de um loop "for".
- while: Início de um loop "while".
- return: Utilizado para retornar um valor de uma função.
- struct: Define uma estrutura.
- new: Cria uma nova instância de um objeto.
- static: Define membros estáticos de uma classe.
- using: Declara o uso de um namespace.

11. Variáveis Locais e Globais

Variáveis Locais são aquelas definidas dentro de um método, função ou bloco específico e têm um escopo limitado a esse contexto. Isso significa que elas só são acessíveis dentro do bloco onde foram declaradas.

```
public void ExemploMetodo()
{
    int idade = 30; // Variável local
    Console.WriteLine(idade);
}
```

Imagem 7 - Exemplo variável local

Variáveis Globais, por outro lado, são definidas fora de qualquer método, geralmente no nível de classe, e são acessíveis a partir de qualquer parte da classe. Elas têm um escopo que abrange toda a classe em que são definidas.

```
public class MinhaClasse
{
    int idadeGlobal; // Variável global

    public void DefinirIdade(int novaIdade)
    {
        idadeGlobal = novaIdade; // Acessando a variável global
    }

    public void ExibirIdade()
    {
        Console.WriteLine(idadeGlobal); // Acessando a variável global
    }
}
```

Imagem 8 - Exemplo variáveis globais

Neste exemplo, a variável idadeGlobal é uma variável global porque é definida no nível de classe. Ela pode ser acessada por qualquer método dentro da classe MinhaClasse.

12. Variáveis temporárias

Variáveis temporárias são aquelas que existem apenas durante a execução de um método ou função e são descartadas quando o método ou função é concluído. Elas são usadas principalmente para armazenar dados temporários ou intermediários durante a execução de um algoritmo. As variáveis temporárias são frequentemente declaradas dentro de blocos de código, como funções ou métodos, e têm escopo limitado a esses blocos.

Neste exemplo, a variável resultado é uma variável temporária que existe apenas dentro do método CalcularSoma e é descartada quando o método termina sua execução.

```
public void CalcularSoma(int a, int b)
{
    int resultado = a + b; // Variável temporária
    Console.WriteLine("A soma é: " + resultado);
    // 'resultado' será descartada quando a função terminar
}
```

Imagem 9 - Variável temporária

13. Variáveis persistentes

Variáveis persistentes são usadas para armazenar dados que devem ser mantidos além do escopo de um método ou função e, portanto, têm uma vida útil mais longa. Isso inclui variáveis que armazenam informações em arquivos ou em bancos de dados, por exemplo. Para criar variáveis persistentes, geralmente são usados recursos de armazenamento externos, como arquivos, bancos de dados ou variáveis de instância de classe que têm escopo mais amplo.

Neste exemplo, a variável arquivo é uma variável que permite escrever dados em um arquivo. Os dados persistirão no arquivo mesmo após a conclusão do método.

```
using System.IO;

public void SalvarDadosEmArquivo(string dados)
{
    // Abre um arquivo para escrita
    using (StreamWriter arquivo = new StreamWriter("dados.txt"))
    {
        arquivo.WriteLine(dados);
        // O arquivo persistirá após o término deste método
    }
}
```

Imagem 10 - Exemplo variável persistente

14. Vinculação de variáveis

Em C#, as vinculações nas declarações de variáveis são estáticas, o que significa que o tipo da variável deve ser definido no momento da sua declaração e não pode ser alterado posteriormente. O tipo da variável é inferido com base no tipo do valor que está sendo atribuído a ela no momento da declaração.

É importante observar que, uma vez que o tipo de uma variável é definido, ele não pode ser alterado posteriormente para um tipo diferente na linguagem C#. Isso é conhecido como vinculação estática e ajuda a prevenir erros de tipo em tempo de compilação.

15. Sobrecarga de operadores

A linguagem de programação C# permite a sobrecarga de operadores, o que significa que você pode definir comportamentos personalizados para operadores em classes definidas pelo usuário. Isso permite que um operador assuma diferentes operações dependendo do contexto da classe em que está sendo usado. A sobrecarga de operadores é útil quando você está criando classes que representam tipos personalizados e deseja que essas classes se comportem de forma intuitiva com operadores.

Exemplo de sobrecarga do operador de adição:

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

    public Complexo(double real, double imaginario)
    {
        Real = real;
        Imaginario = imaginario;
    }

    // Sobrecarga do operador de adição (+)
    public static Complexo operator +(Complexo c1, Complexo c2)
    {
        return new Complexo(c1.Real + c2.Real, c1.Imaginario + c2.Imaginario);
    }
}

// Uso da sobrecarga do operador de adição
Complexo numero1 = new Complexo(3, 2);
Complexo numero2 = new Complexo(1, 7);
Complexo resultado = numero1 + numero2;
```

Imagem 11 - Exemplo sobrecarga de adição

Exemplo de sobrecarga do operador de igualdade:

```
public class Ponto
{
    public int X { get; set; }
    public int Y { get; set; }

    public Ponto(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Sobrecarga do operador de igualdade (==)
    public static bool operator ==(Ponto p1, Ponto p2)
    {
        return (p1.X == p2.X) && (p1.Y == p2.Y);
    }

    // Sobrecarga do operador de desigualdade (!=)
    public static bool operator !=(Ponto p1, Ponto p2)
    {
        return !(p1 == p2);
    }
}

// Uso da sobrecarga do operador de igualdade
Ponto ponto1 = new Ponto(1, 2);
Ponto ponto2 = new Ponto(1, 2);
bool saoIguais = (ponto1 == ponto2); // Resultado: true
```

Imagem 12 - Exemplo sobrecarga igualdade

16. Funções

Em C#, os procedimentos e funções, são definidos dentro de classes. Eles são essenciais para estruturar e organizar o código, permitindo a reutilização de lógica e modularização do programa.

Exemplo de estrutura básica de uma função em C#:

```
[modificadores] tipoRetorno NomeDoProcedimento(parâmetros)
{
    // Corpo do procedimento
}
```

Imagem 13 - Exemplo esqueleto função

16.1 Modificadores

Opcional. Pode incluir palavras-chave como public, private, static, etc., para modificar a visibilidade e o comportamento do procedimento.

16.2 tipoRetorno

Obrigatório para funções, mas omitido para procedimentos (void indica que o procedimento não retorna um valor).

16.3 NomeDoProcedimento

O nome escolhido para o procedimento.

16.4 Parâmetros

Opcional. Lista de parâmetros que o procedimento aceita.

17. P.O.O em C#

A linguagem de programação C# é fortemente orientada a objetos, o que significa que ela suporta os conceitos fundamentais da programação orientada a objetos, como classes, objetos, métodos e herança.

17.1 Classes

Uma classe em C# é um modelo para criar objetos. Ela define as propriedades (atributos) e comportamentos (métodos) que os objetos criados a partir dela terão. As classes em C# são definidas usando a palavra-chave class.

```
public class Pessoa
{
    // Atributos da classe
    public string Nome { get; set; }
    public int Idade { get; set; }

    // Método da classe
    public void Apresentar()
    {
        Console.WriteLine("Olá, meu nome é " + Nome + " e tenho " + Idade +
    }
}
```

Imagem 14 - Exemplo classe

17.2 Objetos

Um objeto é uma instância de uma classe. Quando você cria um objeto, ele herda todas as propriedades e métodos definidos na classe. Você pode criar vários objetos a partir da mesma classe, cada um com seu próprio conjunto de dados.

Neste exemplo, pessoa1 e pessoa2 são objetos da classe Pessoa, cada um com seus próprios valores para as propriedades Nome e Idade.

```
Pessoa pessoa1 = new Pessoa();
pessoa1.Nome = "Alice";
pessoa1.Idade = 30;

Pessoa pessoa2 = new Pessoa();
pessoa2.Nome = "Bob";
pessoa2.Idade = 25;

// Chamando o método Apresentar para cada objeto
pessoa1.Apresentar();
pessoa2.Apresentar();
```

Imagem 15 - Exemplo Objetos

17.3 Métodos

Métodos são funções definidas em uma classe que descrevem o comportamento dos objetos criados a partir dessa classe. Eles permitem que você execute ações específicas em objetos da classe.

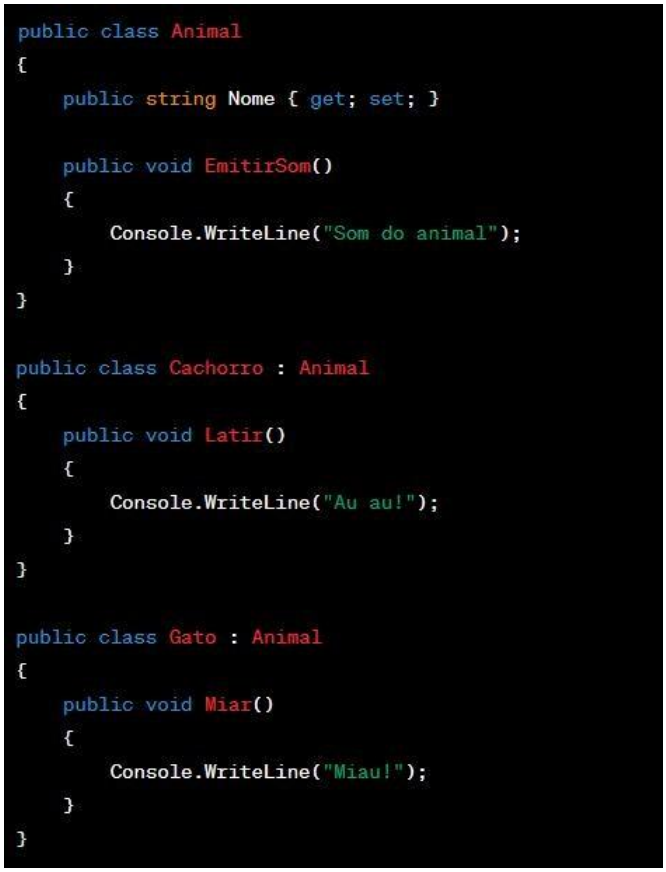
```
Pessoa pessoa = new Pessoa();
pessoa.Nome = "Carlos";
pessoa.Idade = 35;

pessoa.Apresentar(); // Chamando o método Apresentar da classe Pessoa
```

Imagem 16 - Exemplo Métodos

17.4 Herança

A herança é um conceito fundamental na programação orientada a objetos que permite criar novas classes (classes derivadas) com base em classes existentes (classes base). As classes derivadas herdam as propriedades e métodos das classes base e podem adicionar ou modificar seu comportamento.

Neste exemplo, a classe Animal é a classe base e as classes Cachorro e Gato são classes derivadas que herdam as propriedades e métodos da classe base. Cada classe derivada também pode ter seus próprios métodos e comportamentos específicos.



```
public class Animal
{
    public string Nome { get; set; }

    public void EmitirSom()
    {
        Console.WriteLine("Som do animal");
    }
}

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

public class Gato : Animal
{
    public void Miar()
    {
        Console.WriteLine("Miau!");
    }
}
```

Imagem 17 - Exemplo Herança

18. Referências

BILLWAGNER. Documentação do C# – introdução, tutoriais, referência. Disponível em: <https://learn.microsoft.com/pt-br/dotnet/csharp/>.

C# Orientado a Objetos: Introdução. Disponível em: <https://www.devmedia.com.br/csharp-orientado-a-objetos-introducao/29539>. Acesso em: 25 set. 2023.

BILLWAGNER. Programação orientada a objeto (C#) - C#. Disponível em: <https://learn.microsoft.com/pt-br/dotnet/csharp/fundamentals/tutorials/oop>. Acesso em: 25 set. 2023.

C# - Aplicando os conceitos da POO na prática. Disponível em: <https://www.macoratti.net/16/08/c_ooprat1.htm>. Acesso em: 25 set. 2023.

A evolução da linguagem de programação C#. Disponível em: <https://www.devmedia.com.br/a-evolucao-da-linguagem-de-programacao-csharp/28639#:~:text=A%20cria>. Acesso em: 25 set. 2023.