

CENTRO DO TREINAMENTO PROFISSIONAL UBUNTU SOCIEDADE

MÓDULO I

LINGUAGEM DE P. C#

AUTOR: Bernardo Kinavuidi Paulo Lukoki, *Engº*

CAPITULO I – FUNDAMENTOS DE LINGUAGEM DE PROGRAMAÇÃO

I.1. FUNDAMENTOS

As **linguagens de programação** foram criadas para solucionar qualquer tipo de problema na área tecnológica computacional.

Acompanhando todas as tendências da tecnologia, elas estão sempre em constante evolução, buscando oferecer as mais modernas ferramentas para as mais complexas tarefas.

Cada linguagem possui suas particularidades e, no conteúdo de hoje, conheceremos algumas delas e suas principais aplicações.

I.2. LÍNGUAGEM DE PROGRAMAÇÃO

É por onde o hardware (máquina) e o programador se comunicam.

É uma linguagem formal que funciona por meio de uma série de instruções, símbolos, palavras-chave, regras semânticas e sintáticas.

A **linguagem de programação** permite que um programador crie programas a partir de um conjunto de ordens, ações consecutivas, dados e algoritmos.

Esse conjunto faz o controle do comportamento físico e lógico de uma máquina. Existem diversas linguagens, pois também existem diversas formas de transmitir um mesmo comando para alcançar um mesmo objetivo.

I.2.1. Paradigma de programação

Um paradigma de programação é uma metodologia que proporciona ao programador visualizar a estrutura e execução do programa, existem vários, nomeadamente:

- Paradigma Procedural ou Imperativo

Conceito de programação que envolve as linguagens mais ensinadas, como C, C+, Java e Pascal. Define softwares como uma sequência de comandos para serem executados.

- Paradigma Orientado a Objetos

A OOP é o paradigma de programação mais popular. É baseado no uso de componentes individuais, os objetos, que fazem parte da composição do software. Reúne linguagens como PHP, Java, Ruby, C# e Python.

As linguagens de programação podem ser classificados segundo vários critérios, nomeadamente: Quanto a gerações, quanto ao nível, etc.

Quanto ao nível, temos:

- Alto Nível

As linguagens de programação também são classificadas em níveis, **alto ou baixo**.

De modo geral, as linguagens de Alto Nível são aquelas que mais se aproximam à linguagem humana e, por isso, são as que mais facilitam a vida do programador.



Elas chegam à máquina por meio do tradutor ou do compilador.

Tradutor

Como o nome indica, o tradutor interpreta os programas escritos em uma linguagem de programação, traduzindo para a linguagem de máquina do computador. A execução ocorre à medida que são traduzidos.

Compilador

O compilador é um sistema que tem como objetivo traduzir um sistema em linguagem de alto nível para outro de linguagem simbólica.

Ele deve conseguir desempenhar as etapas de análise (analisar o código fonte) e síntese (sintetizar a linguagem).

- Baixo nível

A linguagem de baixo nível é mais próxima da linguagem de máquina. Essas linguagens têm o objetivo de se comunicar com o computador mais rapidamente, de uma forma mais otimizada.

Linguagem de máquina

É o primórdio das linguagens. É uma sequência de bits, ou dígitos, que é quase impossível de ser entendida pelo ser humano.

A linguagem de máquina é formada por códigos binários (0 e 1). É também chamada de linguagem de primeira geração.

Linguagem Assembly

A linguagem Assembly é mais próxima da usada por seres humanos, e, portanto, mais compreensível pelos programadores.

É importante entender essa linguagem, pois cada família de processadores possui sua própria linguagem assembly.

Para que serve a linguagem de programação?

A função das linguagens de programação é conectar a comunicação entre humanos e computadores.

Elas são conjuntos de instruções padronizadas, que servem para o computador entender os comandos.

Quais são as principais linguagens de programação?

Com essa diversidade de linguagens de programação, destacamos abaixo as principais, conheça a importância de cada uma:

- Java

Java é uma linguagem muito conhecida, desenvolvida na década de 1990. É orientada a objetos.

Essa linguagem permite desenvolver softwares que podem ser executados e distribuídos em diferentes plataformas (MAC, Linux, Windows etc.), sem que haja a necessidade de modificá-los ou de focar na arquitetura da máquina.

- JavaScript

JavaScript pode ser confundida com Java, mas lembre-se: essas são linguagens totalmente diferentes uma da outra.

JavaScript é uma linguagem de programação criada para navegadores, e é a programação Web mais popular do mercado.

- Python

Python é uma linguagem de programação utilizada para fins diversos. Classificada como uma linguagem de altíssimo nível, suporta diferentes paradigmas de programação e conta com grandes recursos.

O destaque fica por conta da legibilidade do código e da sintaxe moderna. É uma das principais linguagens de programação.

- Linguagem C

Criada em 1972, a linguagem C é derivada das suas antecessoras ALGOL 68 e BCPL. Foi criada por conta da necessidade de escrever programas de forma mais fácil que a linguagem Assembly.

A Linguagem C simplifica o processo de desenvolvimento por ser estruturada e pela sua portabilidade. Com recursos de baixo nível, permite a incorporação de códigos Assembly.

- Linguagem C++

A linguagem C++ é baseada em C, e foi criada na década de 80. Ela tornou-se uma linguagem muito potente, com capacidade de resolução de problemas altamente complexos.

Ainda muito utilizada nos dias de hoje, é aplicada nas mais variadas frentes, como editores de imagem ou de texto e jogos. É multiparadigma, tem boa performance e portabilidade.

- **Linguagem C#**

A linguagem C# (“c sharp”) foi desenvolvida pela Microsoft e lançada em 2002. Um dos recursos da plataforma .NET, esta linguagem foi criada com o objetivo de aprimorar a comunicação entre diferentes tecnologias utilizadas pela empresa.

É uma linguagem orientada a objetos. A sintaxe foi baseada em C++, Java e Object Pascal.

- **PHP**

PHP é uma linguagem de programação de livre distribuição, que oferece boa performance, portabilidade, e suporte tanto à programação estruturada, quanto à orientação a objetos.

É uma linguagem de código aberto, liberado para que a comunidade de programadores tenha condições de trabalhar na sua evolução e consultar problemas já resolvidos.

É utilizada em todo o mundo para criação de sistemas web dinâmicos.

- **SQL**

A Linguagem de Consulta Estruturada - SQL ou Structured Query Language - é um dos recursos mais conhecidos do mundo.

É a linguagem padrão para trabalhar com bancos de dados. Suas principais características tiveram como inspiração a álgebra relacional.

Alguns dos principais sistemas que utilizam SQL são: Oracle, PostgreSQL, Firebird, MySQL, entre outros.

- **Ruby**

A linguagem Ruby foi criada em 1995. É uma linguagem de programação orientada a objetos e de sintaxe simples.

A proposta do seu criador era criar uma linguagem legível, fácil e agradável. Tem código aberto, que é mantido por uma ativa comunidade de desenvolvedores de todo o mundo.

- **Assembly**

Assembly é uma linguagem de programação de baixo nível, criada por volta dos anos 50, quando os computadores ainda funcionavam com válvulas.

É conhecida como linguagem de montagem ou código de máquina, e permite trabalhar diretamente com as instruções do processador.

- Perl

Perl é uma linguagem de programação de alto nível. É uma multiplataforma, com código aberto, e de fácil aprendizado. Utilizada para jogos, aplicações web, processamento de textos, programação de redes e assim em diante.

Esta linguagem conta com recursos que facilitam a manipulação de textos, o que se torna ideal para o desenvolvimento rápido de scripts e para realização de diversas tarefas.

- Google Go

A Google Go foi lançada em 2009, como solução para atender a diversas necessidades do próprio Google.

De alta performance, é multiplataforma e objetiva ótimos desempenhos tanto da compilação quanto de processamento da aplicação. Tem suporte para Linux, Windows, MacOS e outros.

- Swift

Swift é uma linguagem de programação criada pela Apple e voltada ao desenvolvimento de aplicativos para as plataformas da marca, como Mac OS, iOS, Apple Watch e Apple TV.

Essa linguagem tem código aberto e busca proporcionar liberdade para os programadores. Possui sintaxe simples e possibilidade de incorporar códigos em Objective-C.

- Visual Basic

Visual Basic é uma linguagem de programação produzida pela Microsoft, como parte integrante do pacote Microsoft Visual Studio.

Foi um dos primeiros sistemas que trouxeram a praticidade de escrever programas para o sistema operacional Windows.

- Linguagem R

R é uma linguagem de programação destinada à computação estatística, baseada na linguagem S. De código aberto, é uma linguagem constantemente aprimorada.

É bastante utilizada por cientistas, estatísticos e cientistas de dados. É multiplataforma, com suporte para sistemas operacionais Linux, Windows e Mac.

- Objective-C

Linguagem de programação orientada a objetos que pertence à Apple, a Objective-C é utilizada para o desenvolvimento de aplicações para a plataforma iOS, que engloba iPhone, iPad e iPod Touch.

1.3. PROGRAMAÇÃO

É o ato de programar, ou seja escrever códigos (algoritmos).

2. INTRODUÇÃO A LINGUAGEM C#

O C# (pronuncia-se "C Sharp") é uma linguagem de programação moderna, orientada a objeto e fortemente tipada. O C# permite que os desenvolvedores criem muitos tipos de aplicativos seguros e robustos que são executados no .NET. O C# tem suas raízes na família de linguagens C e os programadores em C, C++, Java e JavaScript a reconhecerão imediatamente.



C# é uma linguagem de programação orientada a objetos e **orientada a componentes**. C# fornece construções de linguagem para dar suporte diretamente a esses conceitos, tornando C# uma linguagem natural para criação e uso de componentes de software. Desde sua origem, o C# adicionou recursos para dar suporte a novas cargas de trabalho e práticas emergentes de design de software. Em sua essência, o C# é uma linguagem **orientada a objeto**.

2.1. Arquitetura do .NET

Programas C# são executados no .NET, um sistema de execução virtual chamado CLR (Common Language Runtime) e um conjunto de bibliotecas de classes. O CLR é a implementação pela Microsoft da CLI (Common Language Infrastructure), um padrão internacional. A CLI é a base para criar ambientes de execução e desenvolvimento nos quais as linguagens e bibliotecas funcionam em conjunto perfeitamente.

O código-fonte escrito em C# é compilado em uma [IL \(linguagem intermediária\)](#) que está em conformidade com a especificação da CLI. O código IL e os recursos, como bitmaps e cadeias de caracteres, são armazenados em um assembly, normalmente com uma extensão de *.dll*. Um assembly contém um manifesto que fornece informações sobre os tipos, a versão e a cultura.

Quando o programa C# é executado, o assembly é carregado no CLR. Em seguida, o CLR executará a compilação JIT (Just-In-Time) para converter o código de IL em instruções nativas da máquina. O CLR também oferece outros serviços relacionados à coleta automática de lixo, tratamento de exceções e gerenciamento de recursos. O código que é executado pelo CLR é, às vezes, chamado de "código gerenciado". "Código não gerenciado" é compilado em linguagem de máquina nativa e visa uma plataforma específica.

Interoperabilidade de linguagem é um recurso importante do .NET. O código IL produzido pelo compilador C# está em conformidade com a CTS (Common Type Specification). O código IL gerado a partir do C# pode interagir com o código que foi gerado a partir das versões do .NET do F#, do Visual Basic, do C++. Há mais de 20 outras linguagens compatíveis com CTS. Um único assembly pode conter vários módulos gravados em diferentes idiomas do .NET. Os tipos podem fazer referência uns aos outros como se fossem escritos na mesma linguagem.

Além dos serviços de tempo de execução, o .NET também inclui bibliotecas extensas. Essas bibliotecas dão suporte a muitas cargas de trabalho diferentes. Eles são organizados em namespaces que fornecem uma ampla variedade de funcionalidades úteis. As bibliotecas incluem desde a entrada e a saída do arquivo até a manipulação de cadeia de caracteres até a análise de XML até estruturas de aplicativos Web para controles de Windows Forms. O aplicativo típico em C# usa bastante a biblioteca de classes para lidar com tarefas comuns de "conexão".

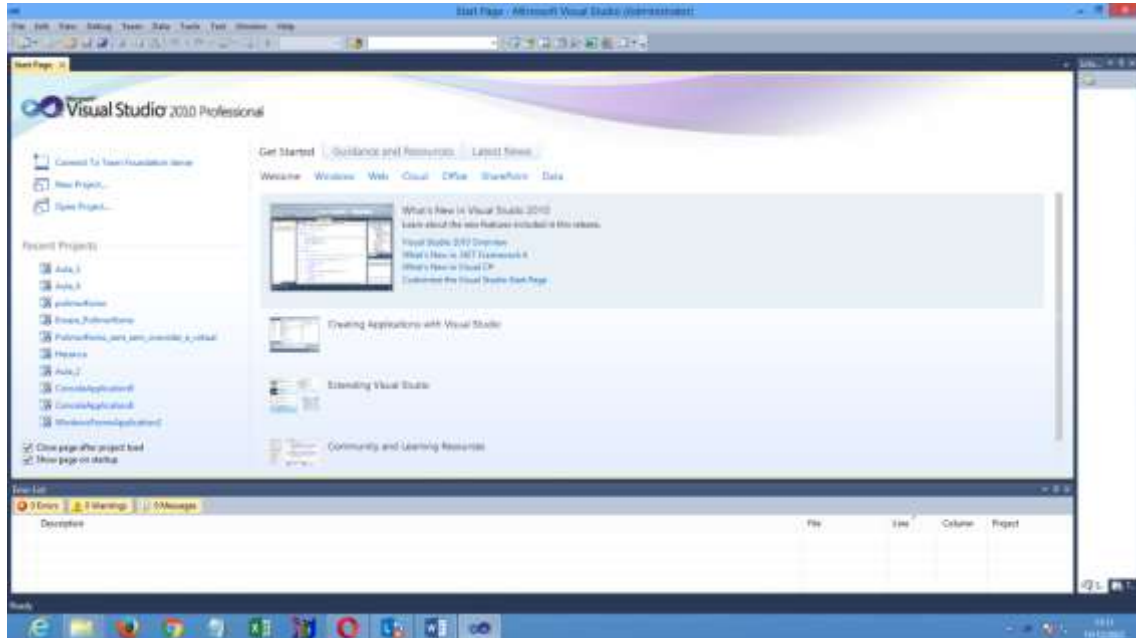


Figura 1: Ubuntu-SD

Primeiro Programa <Hello world>

O programa "Hello, World" é usado tradicionalmente para introduzir uma linguagem de programação. Este é para C#:

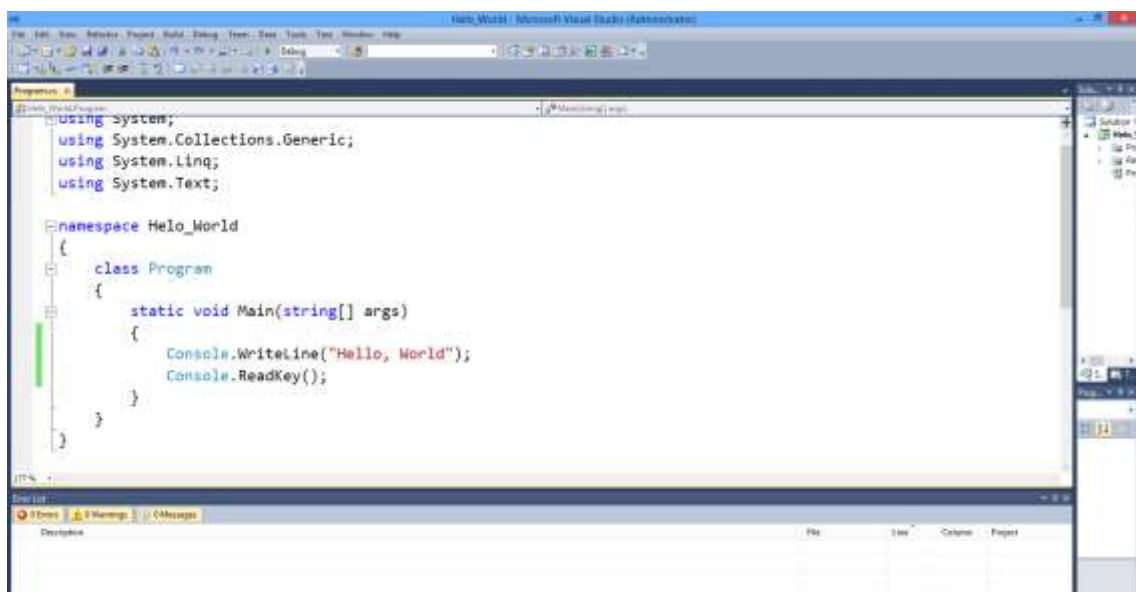


Figura 2: Ubuntu-SD

O programa "Hello, World" começa com uma diretiva `using` que faz referência ao namespace `System`. Namespaces fornecem um meio hierárquico de organizar bibliotecas e programas em `C#`. Os namespaces contêm tipos e outros namespaces — por exemplo, o namespace `System` contém uma quantidade de tipos, como a classe `Console` referenciada no programa e diversos outros namespaces, como `IO` e `Collections`. A diretiva `using` que faz referência a um determinado namespace permite o uso não qualificado dos tipos que são membros desse namespace. Devido à diretiva `using`, o programa pode usar `Console.WriteLine` como um atalho para `System.Console.WriteLine`.

A classe `Hello` declarada pelo programa "Hello, World" tem um único membro, o método chamado `Main`. O método `Main` é declarado com o modificador `static`. Embora os métodos de instância possam fazer referência a uma determinada instância de objeto delimitador usando a palavra-chave `this`, métodos estáticos operam sem referência a um objeto específico. Por convenção, um método estático denominado `Main` serve como ponto de entrada de um programa `C#`.

A saída do programa é produzida pelo método `WriteLine` da classe `Console` no namespace `System`. Essa classe é fornecida pelas bibliotecas de classe padrão, que, por padrão, são referenciadas automaticamente pelo compilador.

2.2. TIPOS E VARIÁVEIS

2.2.1. VARIÁVEIS

São espaços reservados na memória do computador para receberem e armazenarem valores. Ainda, se pode afirmar que, são endereços que referenciam valores alocados em espaços na memória do computador.

Uma *variável* é um rótulo que se refere a uma instância de um tipo específico.

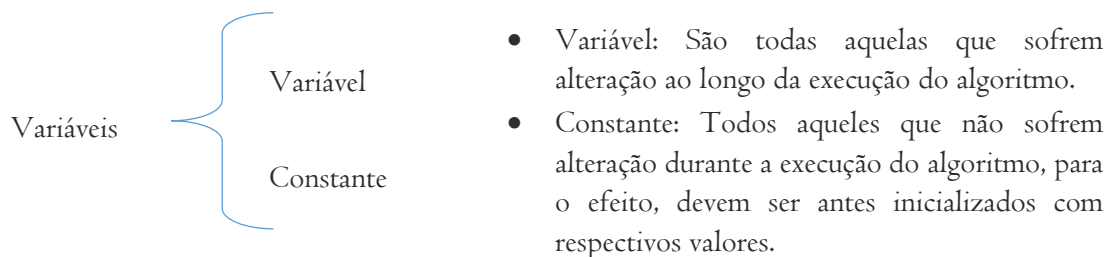
2.2.2. TIPOS

Um *tipo* define a estrutura e o comportamento de qualquer dado em `C#`. A declaração de um tipo pode incluir seus membros, tipo base, interfaces implementadas e operações permitidas para esse tipo.

Há dois tipos em `C#`: *tipos de referência* (*Variados*) e *tipos de valor* (*Constantes*). Variáveis de tipos de **valor** (**Constantes**) contêm diretamente seus dados.

Variáveis de tipos de **referência** (*Variados*) armazenam referências a seus dados.

Veja detalhadamente as duas grandes famílias:



Um *identificador* é um nome de variável.

Exemplo: nome, idade, anonascimento, genero, etc.

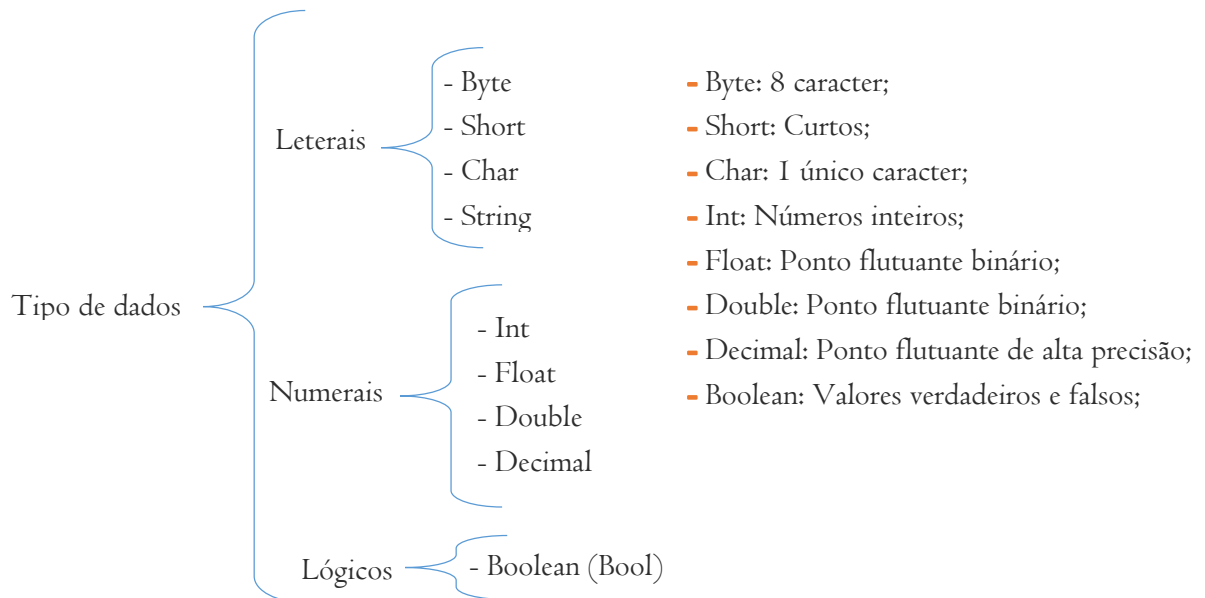
Regras na utilização de um identificador:

- Não pode conter nenhum espaço em branco.
- Não pode conter caracteres especiais;
- Não pode começar com um número;

- **Tipos de dados**

Os tipos de valor do C# são divididos em *tipos simples*, *tipos de enumeração*, *tipos struct* e *tipos de valor anulável* e *tipos de valor de tupla*.

Sendo que as variáveis armazenam valores, os valores são agrupados em tipos como vemos abaixo:



2.3. DECLARANDO VARIÁVEIS EM C#

Uma variável é utilizada para guardar valores temporariamente e é essencial para qualquer programa de computador. Em C#, uma variável é declarada desta forma:

`<tipo de dados> <nome da variáveis>;`

EXEMPLO:

```
String nomeFuncionario;
String departamento;
int anoDeNascimento;
double salario;
```

2.4. METODOS DE ENTRADA E SAÍDA

Em C# os metodos de entrada de dados e saída de informações são:

2.4.1. Entrada

Read()

ReadLine()

2.4.2. Saída

Write()

WriteLine()

Veja um exemplo;

```
String nomeFuncionario;  
String departamento;  
int anoDeNascimento;  
double salario;  
Console.WriteLine("Olá! Informe o nome de um funcionário da  
Etec:");  
nomeFuncionario = Console.ReadLine();  
Console.WriteLine("Em qual departamento este funcionário  
trabalha?");  
departamento = Console.ReadLine();  
Console.WriteLine("Obrigado! O(a) funcionário(a) informado(a)  
foi " + nomeFuncionario + " que trabalha no(a) " +  
departamento );
```

2.5. CONVERSÕES DE TIPOS DE DADOS

A leitura de variáveis do tipo numérica (*int* e *double*) são um pouquinho diferentes, elas precisam ser convertidas. Ou seja quando o usuário digita algo no teclado o computador sempre irá entender tudo que for digitado como texto, ou seja, ele entende tudo como *String*. Mesmo que você digite um número, o computador irá entender como sendo apenas um símbolo qualquer.

Para que o computador possa entender que nem tudo que digitamos é *String*, devemos então informar a ele o tipo de valor que queremos informar e para isso precisamos fazer a conversão usando o comando *Especificados* na tabela abaixo

<i>Inteiro -> String</i>	<code>Convert.ToString()</code>	<code>String.Parse()</code>
<i>String -> Inteiro</i>	<code>Convert.ToInt32()</code>	<code>Int.Parse()</code>
<i>Inteiro -> Double</i>	<code>Convert.ToDouble()</code>	<code>Double.Parse()</code>
<i>Double -> Inteiro</i>	<code>Convert.ToInt32()</code>	<code>Int.Parse()</code>
<i>Char -> String</i>	<code>Convert.ToString()</code>	<code>String.Parse()</code>
<i>String -> Char</i>	<code>Convert.ToChar()</code>	<code>Char.Parse()</code>

Exemplo de converção de tipo de dados

```
Console.WriteLine("Informe o ano de nascimento do  
funcionário:");  
anoDeNascimento = Convert.ToInt32(Console.ReadLine());
```

```
Console.WriteLine("Informe o salário do funcionário:");  
salario = Convert.ToDouble(Console.ReadLine());
```

2.6. OPERADORES

Os operadores indicam o tipo de operação que será executada, gerando assim novos valores a partir de um ou mais operandos (itens à direita ou à esquerda do operador). Geralmente, o resultado é do tipo booleano (**boolean**) ou numérico (**int**, **decimal**, **double**). É importante sabermos como usá-los e é isso que vamos aprender a partir de agora.

2.6.1. Operador de Atribuição

O símbolo igual (=) representa a atribuição de um valor à uma variável, onde ela e o valor atribuído devem obrigatoriamente ser de tipos compatíveis, ou seja, uma variável do tipo **int** por exemplo não pode receber um valor do tipo **string** (a menos que usemos as conversões, mas vamos deixar isso mais para frente).

Veja a sintaxe: `variável = atribuição;`

```
//variável idade recebe o valor 21  
int idade = 21;  
  
//declaro variável maiorIdade como boolean  
bool maiorIdade;  
  
//exibe ambos em tela, atribuindo valor ao booleano  
Console.WriteLine("\n" + idade + "\n");  
Console.WriteLine(maiorIdade = true);  
  
Console.ReadKey();
```

Podemos também atribuir uma variável primitiva a outra variável primitiva. Veja no exemplo da **Listagem 2**.

```
int a = 10;  
int b = a;
```

Os conceitos acima ilustra os Tipos por **Valor** e por **Referência**.

2.6.2. Operadores Aritméticos

Os operadores aritméticos descritos na **Tabela 1** são os mesmos usados em cálculos matemáticos.

<i>Operadores</i>	<i>Discrição</i>
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (Resto da divisão)

```
int a = 5, b = 10, c = 15, d = 20; // declaramos quatro variáveis do tipo int  
Console.WriteLine(a + d); // operação de soma  
Console.WriteLine(c - a); // operação de subtração  
Console.WriteLine(b * c); // operação de multiplicação  
Console.WriteLine(d / b); // operação de divisão  
Console.WriteLine(c % b); // operação de módulo (resto de divisão)  
Console.ReadKey();
```



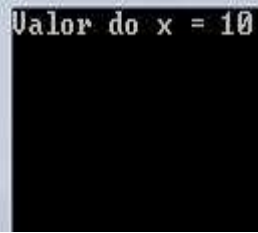
25
10
150
2
5

2.6.2.I. Operadores Aritméticos de Atribuição Reduzida

Esses operadores são usados para compor uma operação aritmética e uma atribuição

<i>Operadores</i>	<i>Discrição</i>
+ =	Mais igual
- =	Menos igual
* =	Vezes igual
/ =	Dividido igual
% =	Módulo igual

```
int x = 5;
x += 5; // é a mesma coisa que x = x + 5
Console.WriteLine("Valor do x = " + x);
Console.ReadKey();
```



Valor do x = 10

2.6.2.2. Operadores incrementais e decrementais

Os operadores incrementais e decrementais têm a função de aumentar ou diminuir exatamente em **1** o valor de uma variável. Eles podem ser pré ou pós **incremental** e pré ou pós **decremental**. Veja os conceitos de cada um deles e um exemplo prático a seguir:

2.6.2.2.1. Incremental (++):

- **Pré incremental ou prefixo** – Significa que, se o sinal for colocado **antes** da variável, primeiramente será somado o valor **I** para esta variável, continuando em seguida a resolução da expressão.
- **Pós incremental ou sufixo** – Significa que, se o sinal for colocado **após** a variável, é resolvido primeiro a expressão, seja ela adição, subtração, multiplicação ou qualquer outra, para em seguida ser adicionado o valor **I** à variável.

2.6.2.2.2. Decremental (--)

- **Pré incremental ou prefixo** – Significa que, se o sinal for colocado **antes** da variável, primeiramente será subtraído o valor **I** para esta variável, continuando em seguida a resolução da expressão.
- **Pós incremental ou sufixo** – Significa que, se o sinal for colocado **após** a variável, é resolvido primeiro a expressão, seja ela adição, subtração, multiplicação ou qualquer outra, para em seguida ser subtraído o valor **I** à variável.

```

Console.WriteLine("Pré-Incremento:\n");

int x = 0;
Console.WriteLine("x = " + x);

Console.WriteLine("\n++x +20 = \n");
Console.WriteLine(++x +20 + "\n");

Console.WriteLine("\n Pós-Incremento:\n");
x = 0;

Console.WriteLine("\n x++ +20 = \n");
Console.WriteLine(x++ + 20 + "\n");

Console.WriteLine("\n Pré-Decremento:\n");
x = 0;

Console.WriteLine("x = " + x);

Console.WriteLine("\n--x +20 = \n");
Console.WriteLine(--x + 20 + "\n");

Console.WriteLine("\n Pós-Decremento:\n");
x = 0;

Console.WriteLine("\n x-- +20 = \n");
Console.WriteLine(x-- + 20 + "\n");

Console.ReadKey();

```

```

Pré-Incremento:
x = 0
++x +20 =
21

Pós-Incremento:
x++ +20 =
20

Pré-Decremento:
x = 0
--x +20 =
19

Pós-Decremento:
x-- +20 =
20

```

2.6.2.3. Operadores Relacionais

Os operadores relacionais comparam dois valores e retornam um valor booleano (**true** ou **false**).

<i>Operadores</i>	<i>Discrição</i>
==	Igual a
!=	Diferente de
>	Maior que
<	Menor que
>=	Maior do que ou igual a
<=	Menor do que ou igual a


```
int a = 10, b = 25, c = 50, d = 100; // declaramos quatro variáveis
de tipo int
```

```
Console.WriteLine(a == d); // avaliamos a igualdade entre a e d
Console.WriteLine(b != c); // avaliamos a desigualdade entre b e c
Console.WriteLine(a > b); // avaliamos se a é maior que b
Console.WriteLine(c < d); // avaliamos se c é menor que d
Console.WriteLine(c >= a); // avaliamos se c é maior ou igual que a
Console.WriteLine(d <= b); // avaliamos se d é menor ou igual que b
```

```
Console.ReadKey();
```

```
False
True
False
True
True
False
```

2.6.2.4. Operadores Lógicos

Os operadores lógicos trabalham como operandos booleanos e seu resultado também será booleano (`true` ou `false`). Eles são usados **somente** em expressões lógicas

Operadores	Discrição
<code>&&</code>	AND “e”
<code>//</code>	OR “ou”
<code>!</code>	NOT “não”

NOTA BEM

Assim, em um teste lógico usando o operador **&&** (**AND**), o resultado somente será verdadeiro (**true**) se **todas** as expressões lógicas forem avaliadas como verdadeiras.

Já, se o operador usado for o **||** (**OR**), basta que apenas uma das expressões lógicas seja verdadeira para que o resultado também seja verdadeiro.

Completando, o operador lógico **!** (**NOT**) é usado para gerar uma negação. Desta forma, é invertida toda a lógica da expressão.

```
int a = 5, b = 10, c = 15, d = 20;
// declaramos quatro variáveis do tipo int
Console.WriteLine(a == 5 && d == 10);
// avaliamos se a é igual a 5 e se d é igual a 10
Console.WriteLine(c < b || d == 20);
// avaliamos se c é menor que b ou se d é igual a 20
Console.WriteLine(! (b > a));
// negamos que b é maior que a
Console.ReadKey();
```

False
True
False

Exercícios de aplicação

- Aritméticos

10 + 5 = []

5 - 2 = []

2 * 8 = []

20 * 5 = []

5 % 2 = []

- Relacionais

10 > 5 = []

5 < 2 = []

2 <= 8 = []

20 >= 5 = []

5 <> 2 = []

- Lógicos

0 e 1 = []

1 ou 1 = []

0 e 0 = []

1 ou 0 = []

0 e ~1 = []

EXEMPLO DE IMPLEMENTAÇÃO

```
String nomeFuncionario;  
String departamento;  
int anoDeNascimento;  
double salario;  
  
Console.WriteLine("Olá! Informe o nome de um funcionário da Etec:");  
nomeFuncionario = Console.ReadLine();  
  
Console.WriteLine("Em qual departamento este funcionário trabalha?");  
departamento = Console.ReadLine();  
  
Console.WriteLine("Obrigado! O(a) funcionário(a) informado(a) foi " + nomeFuncionario + "  
que trabalha no(a) " + departamento);  
  
Console.WriteLine("Informe o ano de nascimento do funcionário:");  
anoDeNascimento = Convert.ToInt32(Console.ReadLine());  
  
Console.WriteLine("Informe o salário do funcionário:");  
salario = Convert.ToDouble(Console.ReadLine());  
  
Console.WriteLine("A idade do funcionário é " + (2020 - anoDeNascimento));  
Console.WriteLine("O salário reajustado é de " + (salario * 1.05));  
  
Console.ReadKey();
```

2.7. ESTRUTURAS DE CONTROLO DE FLUXOS DE DADOS

As estruturas condicionais estão diretamente ligadas ao processo de tomada de decisão dentro de um algoritmo, são elas que nos permitem alterar o fluxo de execução do código dependendo dos valores com os quais estamos trabalhando.

Já as estruturas de repetição assim como o nome já diz nos permitem repetir certos trechos de códigos um número determinado ou indeterminado de vezes, dependendo sempre de qual a condição de parada para tal estrutura.

As estruturas de controlo, mais usados são:

- Estrutura de sequência
- Estrutura condicional
- Estrutuea de repetição

2.7.1. ESTRUTURA DE SEQUÊNCIA

Um algoritmo com estrutura totalmente sequencial é composto por um conjunto de comandos que são executados de maneira sequencial, seguindo a ordem em que aparecem.

A sua sintaxe é:

```
1. Algoritmo EstruturaSequencial;  
2. Variáveis  
3.   ...; //declaração de variáveis  
4. Início  
5.   comando 1;  
6.   comando 2;  
7.   ...  
8.   comando i;  
9.   ...  
10.  comando n;  
11. Fim.
```

Exemplo: implementação de algoritmo que calcula a média de duas notas.

```
1. Algoritmo Media;  
2. Variáveis  
3.   notal, nota2, media: real; //declaração de variáveis  
4. Início  
5.   leia(notal);  
6.   leia(nota2);  
7.   media <- (notal + nota2)/2;  
8.   escreva(media);  
9. Fim.
```

2.7.2. ESTRUTURA CONDICIONAL

As estruturas condicionais possibilitam ao programa tomar decisões e alterar o seu fluxo de execução. Isso possibilita ao desenvolvedor o poder de controlar quais são as tarefas e trechos de código executados de acordo com diferentes situações, como os valores de variáveis.

As estruturas condicionais geralmente analisam expressões booleanas e, caso estas expressões sejam verdadeiras, um trecho do código é executado. No caso contrário, outro trecho do código é executado.

Portanto, as estruturas condicionais estão subdivididas em:

- Estruturas condicionais simples;
- Estruturas condicionais aninhadas;
- Estruturas condicionais de múltiplas escolhas.

2.7.2.I. ESTRUTURA CONDICIONAL IF & ELSE

Estrutura if/else

O **if/else** é uma estrutura de condição em que uma expressão booleana é analisada. Quando a condição que estiver dentro do **if** for verdadeira, ela é executada. Já o **else** é utilizado para definir o que é executado quando a condição analisada pelo **if** for falsa. Caso o **if** seja verdadeiro e, consequentemente executado, o **else** não é executado.

O **if** pode ser utilizado em conjunto com o **else** ou até mesmo sozinho, caso necessário.

Abaixo, temos um exemplo onde o `if` é utilizado em conjunto com o `else`.

A sua sintaxe:

```
if (Condição)
{
    Instruções / bloco de instruções;
}
else
{
    Instruções / bloco de instruções;
}
```

Exemplo

```
int idade = 18;
if (idade >= 18)
{
    // Se a variável idade for maior ou igual à 18 esse trecho de código será executado
    Console.WriteLine("Você é maior de idade!");
}
else
{
    // Se a variável idade não for maior ou igual à 18 esse trecho de código será executado
    Console.WriteLine("Você é menor de idade!");
}
```

1

Em alguns problemas é necessário encadear vários testes de condições. Isto ocorre geralmente quando existem um grande número de possibilidade ou combinações de situações para que um conjunto de comando possa ser executado.

A sua sintaxe é:

```
if (Condição)
{
    Instrução / bloco de instruções
}
else if (Condição)
{
    Instrução / bloco de instruções
}
else
{
    Instrução / bloco de instruções
}
```

Exemplo

```
int idade = 18;
if (idade >= 18)
{
    Console.WriteLine("Você é maior de idade!");
}
else if (idade >= 16)
{
    Console.WriteLine("Você não é maior de idade, mas já pode votar!");
}
else
{
    Console.WriteLine("Você é menor de idade!");
}
```

Exercício 2: Criar algoritmo abaixo em C#

```

1.  Algoritmo Triangulo;
2.  Variáveis
3.    11, 12, 13: inteiro;
4.  Início
5.    leia(11, 12, 13);
6.    se (11 < 12 + 13) e (12 < 11 + 13) e (13 < 11 + 12) então
7.      se (11 = 12) e (12 = 13) então
8.        escreva("TRIÂNGULO EQUILATERO");
9.      senão
10.     se (11 = 12) ou (11 = 13) ou (12 = 13) então
11.       escreva("TRIÂNGULO ISÓSCELES");
12.     senão
13.       escreva("TRIÂNGULO ESCALENO");
14.     fimse;
15.   fimse;
16.   senão
17.     escreva("NÃO FORMAM UM TRIÂNGULO");
18.   fimse;
19. Fim.

```

2.7.2.3. ESTRUTURA DE MÚLTIPLAS ESCOLHAS SWITCH & CASE

A estrutura condicional switch/case vem como alternativa em momentos em que temos que utilizar múltiplos if's no código. Múltiplos if/else encadeados tendem a tornar o código muito extenso, pouco legível e com baixo índice de manutenção.

O switch/case testa o valor contido em uma variável, realizando uma comparação com cada uma das opções. Cada uma dessas possíveis opções é delimitada pela instrução case.

Podemos ter quantos casos de análise forem necessários e, quando um dos valores corresponder ao da variável, o código do case correspondente será executado. Caso a variável não corresponda a nenhum dos casos testados, o último bloco será executado, chamado de default (padrão).

A análise de cada caso também precisa ter seu final delimitado. Essa delimitação é feita através da palavra reservada break. A sintaxe é:

```

switch (condição)
{
    case "expressão de critério_1": comando / bloco de comandos;
    break;
    case "expressão de critério_2": comando / bloco de comandos;
    break;
    case "expressão de critério_3": comando / bloco de comandos;
    break;
    .
    .
    .

    case "expressão de critério_n": comando / bloco de comandos;
    break;
}

```


Exemplo:

```
int mes = 2;
switch (mes)
{
    case 1:
        Console.WriteLine("O mês é Janeiro!");
        break;
    case 2:
        Console.WriteLine("O mês é Fevereiro");
        break;
        .
    .
    .

    case 9:
        Console.WriteLine("O mês é Setembro");
        break;
    .
    .
    .

    case 12:
        Console.WriteLine("O mês é Dezembro");
        break;
    default:
```

2.7.3. ESTRUTURA DE REPETIÇÃO

Estruturas de repetição, também conhecidas como loops (laços), são utilizadas para executar repetidamente uma instrução ou bloco de instrução enquanto determinada condição estiver sendo satisfeita. As principais estruturas de repetição na maioria das linguagens são o for e o while, no C# além dessas duas também temos o do/while e o foreach.

2.7.3.I. INSTRUÇÃO WHILE

O while é uma estrutura de repetição que irá repetir o bloco de código enquanto uma determinada condição for verdadeira, geralmente o while é utilizado quando não sabemos quantas vezes o trecho de código em questão deve ser repetido.

A sintaxe:

```
while (<condição>)\n{\n    // comando ou bloco de comandos;\n}
```

Exemplo

```
int numero = 1;\nwhile (numero <= 10)\n{\n    Console.WriteLine(numero);\n    Numero ++;\n}
```

A estrutura `do/while` é bem semelhante a estrutura `while`, a diferença é que na estrutura `do/while` a condição é testada apenas ao final do loop, ou seja, o código será executado ao menos uma vez, mesmo que a condição seja falsa desde o início.

Sintaxe:

```
Do\n{\n    // comando ou bloco de comandos;\n}\nwhile (<condição>;
```

Exemplo

```
int contador = 15;\ndo\n{\n    Console.WriteLine("O contador vale: " + contador);\n    contador++;\n} while (contador <= 10);
```

Exercício: Calcular a média de vários alunos do algoritmo abaixo em C#.

```

1.  Algoritmo MediaTurma;
2.  Variáveis
3.      nota1, nota2, media: real;
4.      numalunos: inteiro;
5.      nome: literal;
6.  Inicio
7.      leia(numalunos);
8.      enquanto (numalunos > 0) faça
9.          leia(nome);
10.         leia(nota1);
11.         leia(nota2);
12.         media <- (nota1 + nota2)/2;
13.         se (media >= 5) então
14.             escreva("O aluno ", nome, " está APROVADO");
15.         senão
16.             escreva("O aluno ", nome, " está REPROVADO");
17.         fimse;
18.         numalunos <- numalunos - 1;
19.     fimenquanto;
20. Fim.

```

2.7.3.3. INSTRUÇÃO FOR

O for é uma estrutura de repetição que trabalha com uma variável de controle e que repete o seu bloco de código até que um determinada condição que envolve essa variável de controle seja falsa, geralmente utilizamos a estrutura for quando sabemos a quantidade de vezes que o laço precisará ser executado.

A sintaxe:

```

for (<tipo de dado><variável de controle>; <condição de parada>; <incremento/decremento da
variável de controle>)
{
    // comando ou bloco de comandos;
}

```

Exemplo:

```

for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("A variável i agora vale " + i);
}

```

```

1.  Algoritmo ProgressaoGeometrica;
2.  variáveis
3.      Contador,
4.      Termo,
5.      Soma : inteiro;
6.  Início
7.      Contador <- 1;
8.      Soma <- 0;
9.      leia(Termo);
10.     repita
11.         escreva (Termo, "\n");
12.         Soma <- Soma + Termo;
13.         Termo <- Termo * 3;
14.         Contador <- Contador + 1;
15.     até (Contador > 25);
16.     escreva (Soma);
17. Fim.

```

2.7.3.4. INSTRUÇÃO FOREACH

Além da estrutura `for` no C# também temos a estrutura `foreach`, basicamente a estrutura `foreach` serve para que possamos percorrer os elementos de uma estrutura de dados de forma mais simples e menos verbosa.

O `foreach` foi adicionado pois na maioria das vezes que precisamos percorrer os elementos de uma coleção não nos interessa o índice daquela coleção e sim o seu valor e antes da existência do `foreach` era necessário utilizar o `for` e percorrer os elementos de uma coleção utilizando os índices da mesma.

A sintaxe:

Exemplo:

```

string[] alunos = new string[4] { "Cleyson", "Anna", "Autobele", "Hayssa" };
for (int i = 0; i < 4; i++)
{
    Console.WriteLine(alunos[i]);
}

```

Obs: Não que seja menos importante, em C# é normal falar e usar comandos que permitão o programa se mater executando mesmo quando haja erros de estruturais ou de sintaxe. Para tal, usamos as exceções.

2.7.4. ESTRUTURAS DE EXCEPÇÕES (*exceptions*)

Exceções em C# fornecem uma maneira estruturada, uniforme e segura de manipulação, tanto em nível de sistema e erros condicionais de aplicação. *Exceptions* em C# são similares a exceções em C++, mas com algumas importantes diferenças:

- Em C#, todas as *exceptions* representam um instância de classe derivada de `System.Exception`.

- Em C# o Bloco `finally` pode ser escrito ao final do código para executar em ambas as situações de exceção ou execução normal.

2.7.4.I. Exceções possuem as principais propriedades

- Um bloco `try` em torno de uma instrução pode lançar exceções;
- Após a execução do bloco `try`, o fluxo de controle vai para o primeiro manipulador de exceção associado que está presente em qualquer lugar na pilha de chamadas. O `catch` é usado para definir o manipulador de exceção;
- Se nenhum manipulador de exceção para uma determinada exceção estiver presente, o programa apresenta erro.

Exceções podem ser explicitamente geradas por um programa usando a palavra-chave `throw`.

Exemplo:

```
static void TestCatch2()
{
    System.IO.StreamWritersw = null;
    try
    {
        sw = newSystem.IO.StreamWriter(@"C:\test\test.txt");
        sw.WriteLine("Hello");
    }
    catch (System.IO.FileNotFoundException ex)
    {
        // Put the more specific exception first.
        System.Console.WriteLine(ex.ToString());
    }
    catch (System.IO.IOException ex)
    {
        // Put the less specific exception last.
        System.Console.WriteLine(ex.ToString());
    }
    finally
    {
        sw.Close();
    }
    System.Console.WriteLine("Done");
}
```

3. ESTUDO SOBRE CLASSES

Em programação e na orientação a objetos, uma **classe** é um Tipo abstrato de Dados (TAD); ou seja, uma descrição que abstrai um conjunto de objetos com características similares (um projeto do objeto), é um código da linguagem de programação orientada a objetos que define e implementa um novo tipo de objeto, que terão características (atributos) que guardaram valores e, também funções específicas para manipular estes.^[1] Formalmente, é um conceito que encapsula abstrações de dados e procedimentos que descrevem o conteúdo e o comportamento de entidades do mundo real, representadas por objetos.^[2] De outra forma, uma classe pode ser definida como uma descrição das propriedades ou estados possíveis de um conjunto de objetos, bem como os comportamentos ou ações aplicáveis a estes mesmos objetos.

A classe é um elemento primordial de um diagrama de classes; modelagem importante na programação orientada a objetos.

3.1. Estrutura da classe

Uma classe comumente define o estado e o comportamento de um objeto implementando [atributos](#) e [métodos](#).

- **Os atributos** (por vezes referidos como "campos", "membros de dados" ou "propriedades"), indicam as possíveis informações armazenadas por um objeto de uma classe, representando o estado de cada objeto.
- **Os métodos** (por vezes referidos como "operações" ou serviços) são procedimentos que formam os comportamentos e ações oferecidos por objetos de uma classe, sendo responsáveis por alterar o estado ou fornecer informações sobre um objeto.

Um método é um bloco de código que contém uma série de instruções. Um programa faz com que as instruções sejam executadas chamando o método e especificando os argumentos de método necessários. No C#, todas as instruções executadas são realizadas no contexto de um método.

O método Main é o ponto de entrada para todos os aplicativos C# e é chamado pelo CLR (Common Language Runtime) quando o programa é iniciado. Em um aplicativo que usa [instruções de nível superior](#), o Main método é gerado pelo compilador e contém todas as instruções de nível superior.

Assinaturas de método

Os métodos são declarados em uma [classe](#), [struct](#) ou [interface](#) especificando o nível de acesso, como public ou private, modificadores opcionais, como abstract ou sealed, o valor retornado, o nome do método e os parâmetros de método. Juntas, essas partes são a assinatura do método.

Os parâmetros de método estão entre parênteses e separados por vírgulas. Parênteses vazios indicam que o método não requer parâmetros. Essa classe contém quatro métodos:

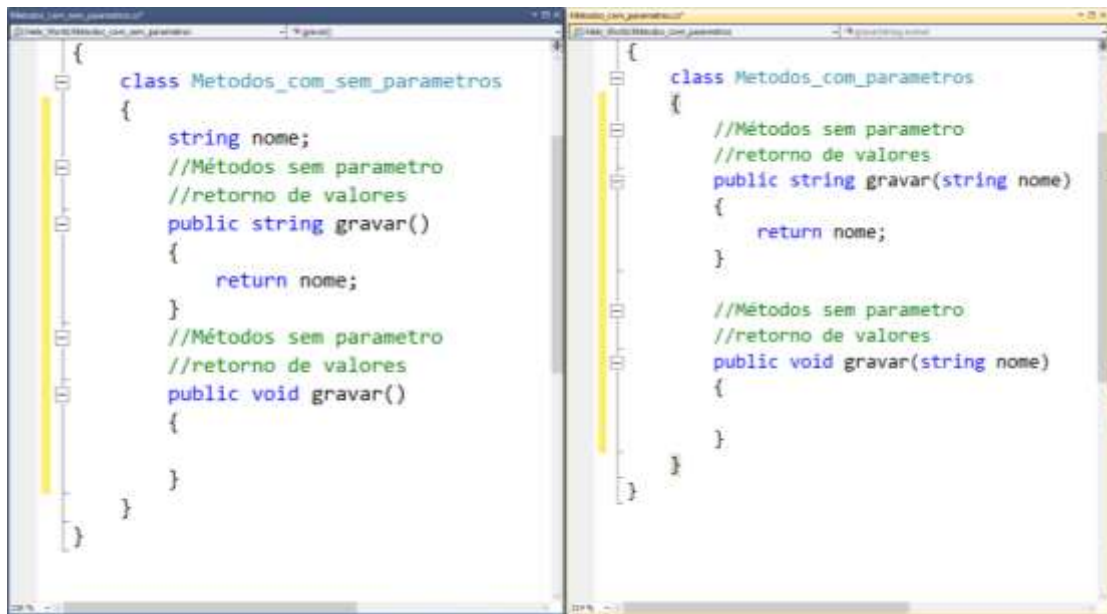


Figura 3: Ubuntu-SD

Exercício

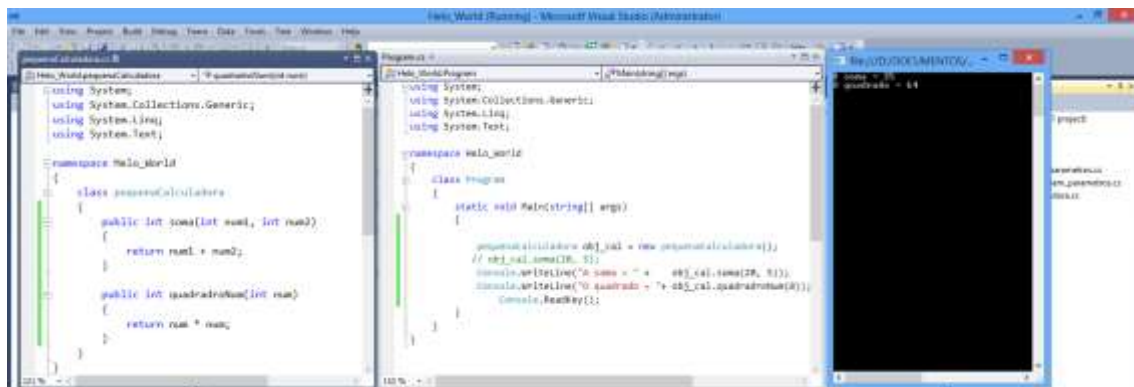


Figura 4: Ubuntu-SD

Outros elementos associados a uma classe são:

- **Construtor e destrutor** - métodos especiais que definem o comportamento do objeto de uma classe no momento da sua criação e destruição. Em algumas linguagens, como em C++, um método destrutor é utilizado para liberar recursos do sistema (como memória),^[3] já em outras, como em Java e C#, isto é realizado de modo automático pelo coletor de lixo.
- **Propriedade** - define o acesso a um estado do objeto.
- **Evento** - define um ponto em que o objeto pode chamar outros procedimentos de acordo com seu comportamento e estado interno.

3.2. Encapsulamento

No paradigma de orientação a objetos, é possível encapsular o estado de um objeto. Em termos práticos, isso se realiza limitando o acesso a atributos de uma classe exclusivamente através de seus métodos. Para isso, as linguagens orientadas a objeto oferecem limitadores de acesso para cada membro de uma classe.

Tipicamente os limitadores de acesso são:

- público (*public*) - o membro pode ser acessado por qualquer classe. Os membros públicos de uma classe definem sua interface
- protegido (*protected*) - o membro pode ser acessado apenas pela própria classe e suas sub-classes
- privado (*private*) - o membro pode ser acessado apenas pela própria classe

No exemplo abaixo, implementado a classe *Pessoa* permite o acesso ao atributo *nome* somente através dos métodos *setNome* e *getNome*.

```
public class Pessoa {  
    private String nome;  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

3.3. Herança

A herança é um relacionamento pelo qual uma classe, chamada de sub-classe, herda todos comportamentos e estados possíveis de outra classe, chamada de super-classe ou classe base. É permitido que a sub-classe estenda os comportamentos e estados possíveis da super-classe (por isso este relacionamento também é chamado de extensão). Essa extensão ocorre adicionando novos membros a sub-classe, como novos métodos e atributos.

É também possível que a sub-classe altere os comportamentos e estados possíveis da super-classe. Neste caso, a sub-classe sobrescreve membros da super-classe, tipicamente métodos.

Quando uma classe herda de mais de uma super-classe, ocorre uma herança múltipla. Esta técnica é possível em C++ e em Python, mas não é possível em Java e C#, no entanto estas linguagens permitem múltipla tipagem através do uso de interfaces.

Exemplo de aplicação de Herança

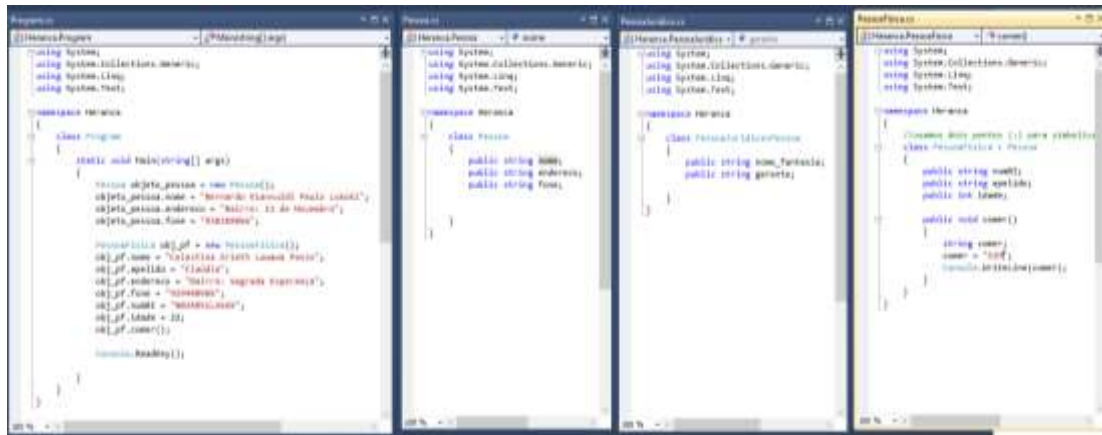


Figura 5: Ubuntu-SD

3.4. Polimorfismo

Ver artigo principal: [Polimorfismo](#)

Na [programação orientada a objetos](#), o **polimorfismo** permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam. Assim, um mesmo método pode apresentar várias formas, de acordo com seu contexto. O polimorfismo é importante pois permite que a semântica de uma interface seja efetivamente separada da implementação que a representa. O termo polimorfismo é originário do grego e significa 'muitas formas' (poli = muitas, morphos = formas).

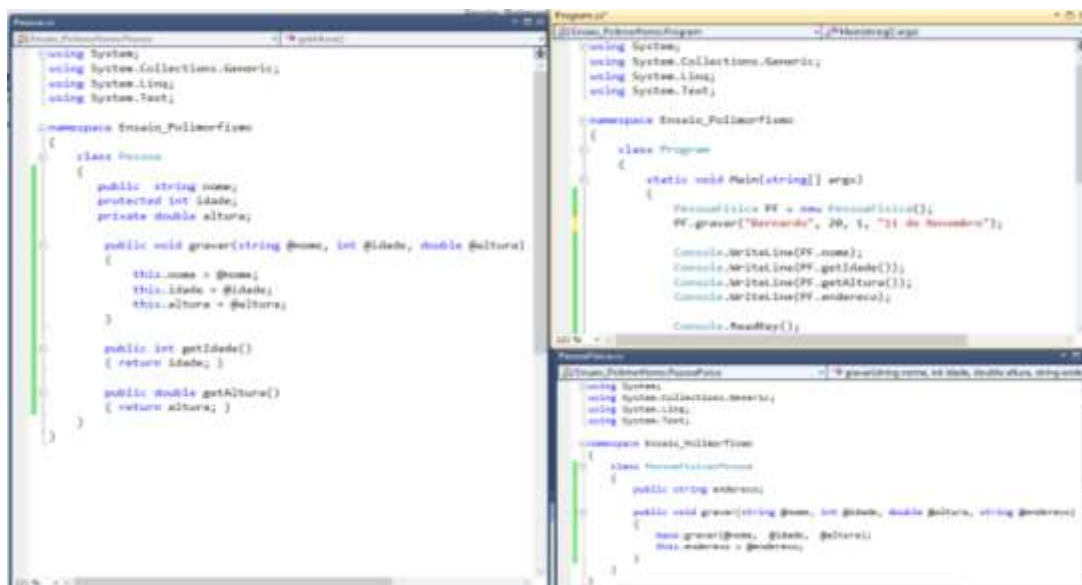


Figura 6: Ubuntu-SD

3.5. Classes abstratas e concretas

Uma classe abstrata é desenvolvida para representar entidades e conceitos abstratos. A classe abstrata é sempre uma superclasse que não possui instâncias. Ela define um modelo (*template*) para uma funcionalidade e fornece uma implementação incompleta - a parte genérica dessa funcionalidade - que é compartilhada por um grupo de classes derivadas. Cada uma das classes derivadas completa a funcionalidade da classe abstrata adicionando um comportamento específico.

Uma classe abstrata normalmente possui métodos abstratos. Esses métodos são implementados nas suas classes derivadas concretas com o objetivo de definir o comportamento específico. O método abstrato define apenas a assinatura do método e, portanto, não contém código.

Por outro lado, as classes concretas implementam todos os seus métodos e permitem a criação de instâncias. Uma classe concreta não possui métodos abstratos e, geralmente, quando utilizadas neste contexto, são classes derivadas de uma classe abstrata.

Métodos abstratos

Um método abstrato é aquele com apenas uma assinatura e sem corpo de implementação. É frequentemente utilizado para especificar que uma subclasse deve fornecer uma implementação do método. Métodos abstratos são usados para especificar interfaces em algumas linguagens de computador.

Métodos especiais

Os métodos especiais são muito específicos da linguagem e uma linguagem pode suportar nenhum, alguns ou todos os métodos especiais definidos aqui. O compilador de uma linguagem pode gerar automaticamente métodos especiais padrão ou um programador pode ter permissão para definir métodos especiais opcionalmente. A maioria dos métodos especiais não pode ser chamada diretamente, mas, em vez disso, o compilador gera código para chamá-los nos momentos apropriados.

Métodos estáticos

Os métodos estáticos devem ser relevantes para todas as instâncias de uma classe, e não para qualquer instância específica. Nesse sentido, são semelhantes a variáveis estáticas. Um exemplo seria um método estático para somar os valores de todas as variáveis de cada instância de uma classe. Por exemplo, se houvesse uma classe Produto, ela poderia ter um método estático para calcular o preço médio de todos os produtos.

3.6. FUNÇÕES (MÉTODOS / PROCEDIMENTOS) PRÉ-DEFINIDAS

São funções já prontas, disponibilizadas pela biblioteca da própria linguagem de programação.

Portanto, as principais funções pré-definidas do C# (elas auxiliam a sua tarefa de programar) por enquanto, considere apenas que uma função pré-definida do C# tem como objetivo realizar uma tarefa específica dentro em programas, facilitando o trabalho do programador. O C# possui uma grande quantidade de funções pré-definidas.

3.6.1. FUNÇÕES NUMÉRICAS E TRIGONOMÉTRICAS

Função `Math.Abs(x)`: Resulta no valor absoluto de um número qualquer (representado por `x`).

Função `Math.Sqrt(x)`: Resulta na raiz quadrada de um número positivo qualquer (representado por `x`).

Função `Math.Pow(x,y)`: Resulta no valor de `x` elevado a `y`.

Função `Math.Round(x)`: Resulta no valor de `x` elevado arredondado.

Função `Math.PI`: Funciona como se você tivesse digitado o valor de π (pi).

Função `Math.Sin(x)`: Resulta no valor do seno de um ângulo qualquer (representado por `x`) em radianos.

Função `Math.Cos(x)`: Resulta no valor do cosseno de um ângulo qualquer (representado por `x`) em radianos.

Função `Math.Tan(x)`; Resulta no valor da tangente de um ângulo qualquer (representado por `x`) em radianos.

Obs.: Para converter graus para radianos: $\text{radianos} = \text{graus} * \text{Math.PI} / 100$; Obs.: Para converter graus para radianos: $\text{radianos} = \text{graus} * \text{Math.PI} / 100$.

Exemplo

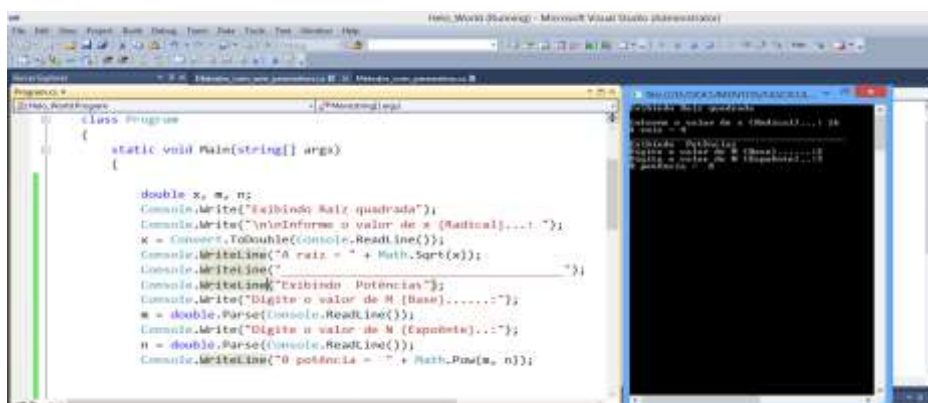


Figura 7: Ubuntu-SD

CENTRO DE TREINAMENTO PROFISSIONAL UBUNTU-SD

Firma: "BERFELI - COMÉRCIO GERAL E PRESTAÇÃO DE SERVIÇOS (SU), LDA."

NIF: 5001282433

SEDE: Na Província de Zaire, Município de M'banza-Kongo, Bairro II de Novembro, Rua s/n, Casa

3.6.2. FUNÇÕES DE TEXTOS

Você pode utilizar uma série de funções pré-definidas para trabalhar com variáveis do tipo string (texto). Veja a seguir códigos exemplificando o uso das principais funções pré-definidas para string/texto. a)

Função Substring: Permite pegar uma parte de uma string. **Obs.:** Considera-se que o primeiro caractere de um texto/string ocupa a posição zero (0).

Exemplo: A palavra informática tem as seguintes posições: **i n f o r m á t i c a** (são 11 caracteres e 11 posições: da 0 a 10).

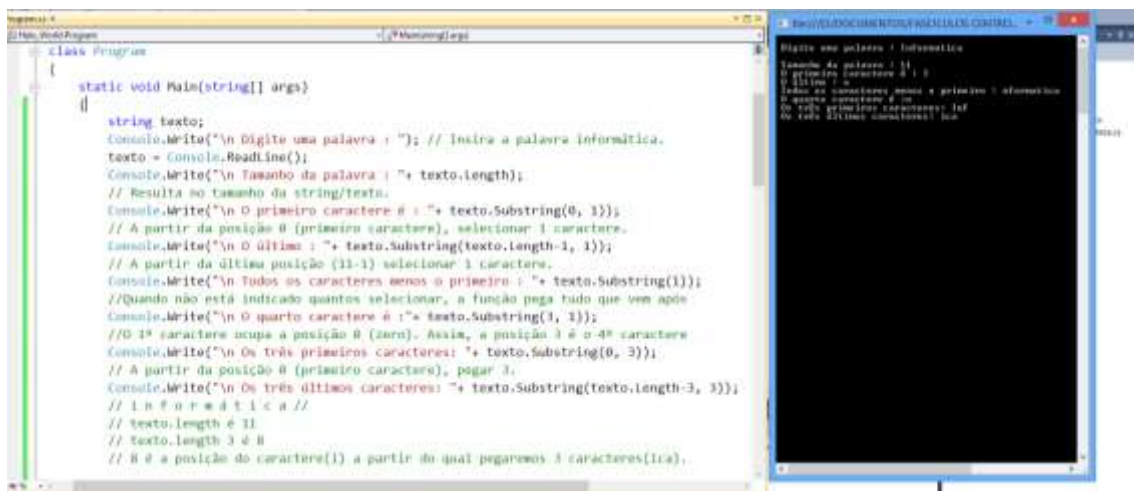


Figura 8: Ubuntu-SD

3.6.3. FUNÇÕES DE REMOÇÃO DE ESPAÇOS

Função TrimStart remove os espaços em branco do início da string.

Função TrimEnd remove os espaços em branco do final da string.

Função Trim remove os espaços em branco do início e do fim da string.

Exemplo:

```

string nome = "    João Gomes    ";
Console.WriteLine(nome); nome = nome.Trim();
//Console.WriteLine(nome); nome = nome.TrimEnd();
//Console.WriteLine(nome); nome = nome.TrimStart();
Console.WriteLine(nome);
    
```

3.6.4. FUNÇÕES DE PREENCHIMENTO

As funções `PadLeft` e `PadRight` servem para preencher uma string a esquerda ou a direita com um caractere especificado.

Exemplo

```
string nome = "João";
    nome = nome.PadLeft(10, ' ');
    // Ficarà como " João"
    Console.WriteLine(nome);
string codigo = "123";
    codigo = codigo.PadLeft(6, '0');
    // Ficarà como "000123"
    Console.WriteLine(codigo);
```

4. INTRODUÇÃO A MATRIZES E VETORES

Vetores e **Matrizes** são estruturas de dados muito simples que podem nos ajudar muito quando temos muitas variáveis do mesmo tipo em um algoritmo. Imagine o seguinte problema: Você precisa criar um algoritmo que lê o nome e as 4 notas de 50 alunos, calcular a média de cada aluno e informar quais foram aprovados e quais foram reprovados.

Conseguiu imaginar quantas variáveis você vai precisar? Muitas né? Vamos fazer uma conta rápida: 50 variáveis para armazenar os nomes dos alunos, $(4 * 50 =)$ 200 variáveis para armazenar as 4 notas de cada aluno e por fim, 50 variáveis para armazenar as médias de cada aluno. 300 variáveis no total, sem contar a quantidade de linhas de código que você vai precisar para ler todos os dados do usuário, calcular as médias e apresentar os resultados. Mas eu tenho uma boa notícia pra você. Nós não precisamos criar 300 variáveis! Podemos utilizar **Vetores** e **Matrizes** (também conhecidos como **ARRAYs**)!

4.1. MATRIZES UNIDIMENSIONAIS

(**array** uni-dimensional) é uma variável que armazena várias variáveis do mesmo tipo. Ou seja, é uma variável composta homogênea unidimensional formada por uma sequência de variáveis, todas do mesmo tipo.

Sintaxe:

<tipo de dados> [] vetor = novo <tipo de dados> [tamanho];

```
int[] array = new int[10];
```

1	23	44	67	95	12	34	74	76	89
---	----	----	----	----	----	----	----	----	----

array [5] = 95 // array[8] = 74

Essa matriz contém os elementos de array[0] a array[9]. Os elementos da matriz são inicializados para o **valor padrão** do tipo de elemento, 0 para inteiros.

Matrizes podem armazenar qualquer tipo de elemento que você especificar, como o exemplo a seguir que declara uma matriz de cadeias de caracteres:

```
string[] stringArray = new string[6];
```

Inicialização de Matriz

Você pode inicializar os elementos de uma matriz ao declarar a matriz. O especificador de comprimento não é necessário porque é inferido pelo número de elementos na lista de inicialização. Por exemplo:

```
int[] arrayI = new int[] { 1, 3, 5, 7, 9 };
```

O ícone a seguir mostra uma declaração de uma matriz de cadeia de caracteres em que cada elemento da matriz é inicializado por um nome de um dia:


```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Você pode evitar a expressão `new` e o tipo de matriz ao inicializar uma matriz após a declaração, conforme mostrado no código a seguir. Isso é chamado de [matriz tipada implicitamente](#):

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Recuperar dados da matriz

Você pode recuperar os dados de uma matriz usando um índice. Por exemplo:

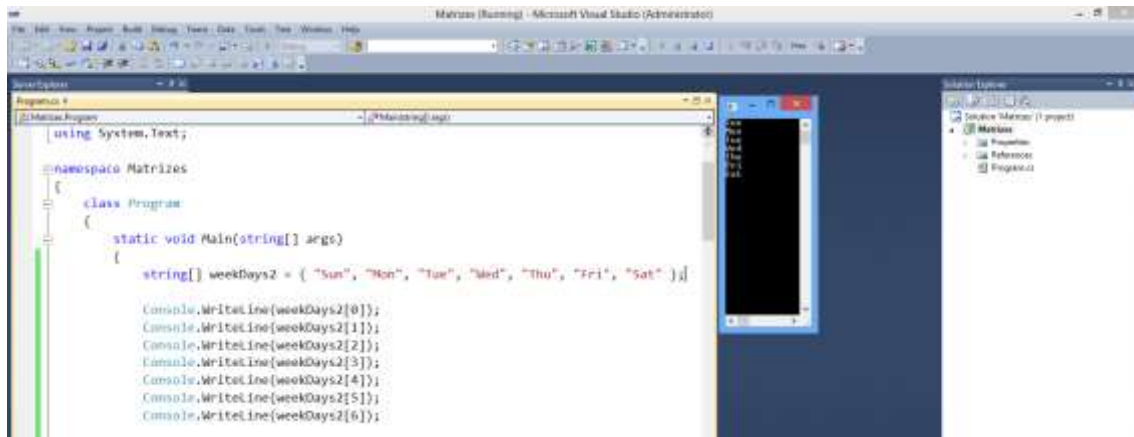


Figura 9: Ubuntu-SD

Você pode declarar uma variável de matriz sem criá-la, mas deve usar o operador `new` ao atribuir uma nova matriz a essa variável. Por exemplo:

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 }; // OK
//array3 = { 1, 3, 5, 7, 9 }; // Error
```

4.2. MATRIZES MULTIDIMENSIONAIS

As matrizes podem ter mais de uma dimensão. Por exemplo, a declaração a seguir cria uma matriz bidimensional de quatro linhas e duas colunas.

Sintaxe:

<tipo de dados> [] vetor = novo <tipo de dados> [tamanho1(**linhas**), tamanho2(**colunas**)];

```
int[,] array = new int[4, 2];
```

Exemplo

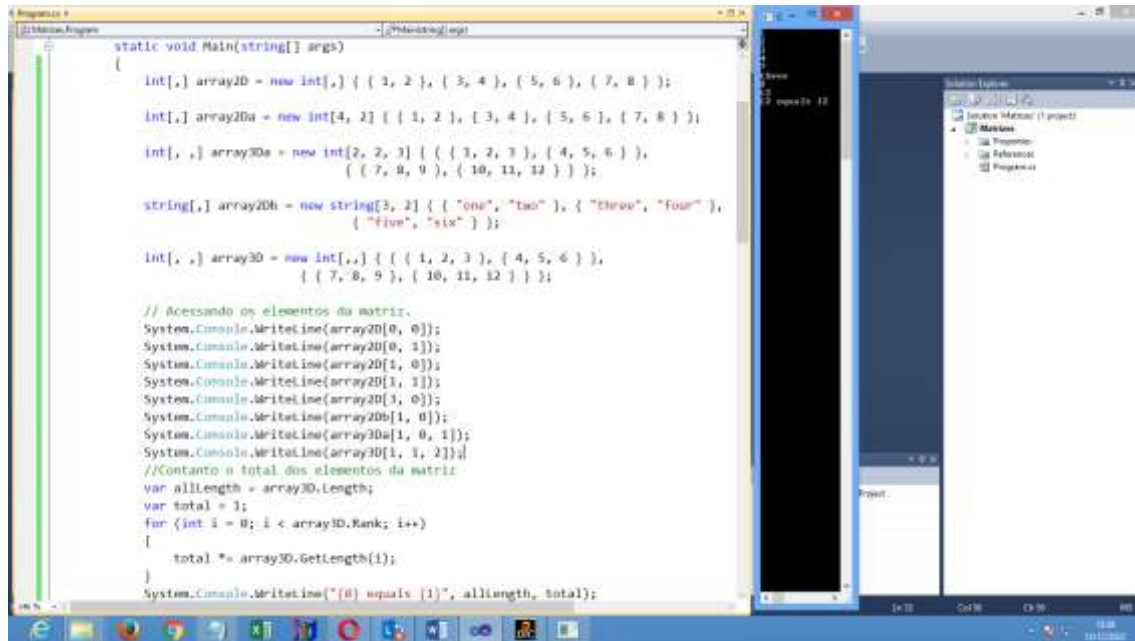


Figura 10: Ubuntu-SD

Exemplo

Algoritmo "semnome"

```
// Disciplina : [Curso de Lógica de Programação]
// Formador : Bernardo Kinavuidi Paulo Lukoki
// Autor(a) : CTP-Ubuntu-SD
// Data atual : 26/II/2022
Var
// Seção de Declarações das variáveis
vetA: vetor [1..10] de inteiro
i: inteiro
Inicio
// Seção de Comandos, procedimento, funções,
operadores, etc...
para i <- 1 ate 10 faca // varrer a linha do vetor
Escreva("Digite um numero para ser armazenado:")
vetA[i] <- 0
fimpara
para i <- 1 ate 4 faca // varrer a linha da matriz
para j <- 1 ate 3 faca // varrer a coluna da matriz
Escreva(matrizA[i,j])
fimpara
escreva("")
fimpara
Fimalgoritmo
```

Algoritmo "semnome"

```
// Disciplina : [Curso de Lógica de Programação]
// Formador : Bernardo Kinavuidi Paulo Lukoki
// Autor(a) : CTP-Ubuntu-SD
// Data atual : 26/II/2022
Var
// Seção de Declarações das variáveis
vetA: vetor [1..10] de inteiro
i: inteiro
Inicio
// Seção de Comandos, procedimento, funções,
operadores, etc...
para i <- 1 ate 10 faca // varrer a linha do vetor
Escreva("Digite um numero para ser armazenado:")
leia(vetA[i])
fimpara
para i <- 1 ate 4 faca // varrer a linha da matriz
para j <- 1 ate 3 faca // varrer a coluna da matriz
Escreva(matrizA[i,j])
fimpara
escreva("")
fimpara
Fimalgoritmo
```

4.3. MATRIZES

Matriz (array multi-dimensional) é um **vetor** de **vetores**. No nosso problema, imagine uma matriz para armazenar as 4 notas de cada um dos 50 alunos. Ou seja, um vetor de 50 posições, e em cada posição do vetor, há outro vetor com 4 posições. Isso é uma matriz.

Sintaxe:

<lista-de-variáveis> : vetor "["<lista-de-intervalos>"]" de <tipo-de-dado>

matrizA: vetor [1..2,1..4] de inteiro

1	2	45	98
33	71	23	59

matrizA[1,2] <- 2 // matrizA[2,3] <- 23

EXEMPLO

Algoritmo "semnome"

```
// Disciplina : [Curso de Lógica de Programação]
// Formador : Bernardo Kinavuidi Paulo Lukoki
// Autor(a) : CTP-Ubuntu-SD
// Data atual : 26/11/2022
```

Var

```
// Seção de Declarações das variáveis
```

matrizA: vetor [1..2,1..2] de inteiro

i, j: inteiro

Início

```
// Seção de Comandos, procedimento, funções,
operadores, etc...
```

para i <- 1 ate 2 faca // varrer a linha da matriz

para j <- 1 ate 2 faca // varrer a coluna da matriz

matrizA[i,j]<-3

fimpara

fimpara

para i <- 1 ate 2 faca // varrer a linha da matriz

para j <- 1 ate 2 faca // varrer a coluna da matriz

Escreva(matrizA[i,j])

fimpara

escreva("")

fimpara

Fimalgoritmo

Algoritmo "semnome"

```
// Disciplina : [Curso de Lógica de Programação]
// Formador : Bernardo Kinavuidi Paulo Lukoki
// Autor(a) : CTP-Ubuntu-SD
// Data atual : 26/11/2022
```

Var

```
// Seção de Declarações das variáveis
```

matrizA: vetor [1..4,1..3] de inteiro

i, j: inteiro

Início

```
// Seção de Comandos, procedimento, funções,
operadores, etc...
```

para i <- 1 ate 4 faca // varrer a linha da matriz

para j <- 1 ate 3 faca // varrer a coluna da matriz

leia(matrizA[i,j])

fimpara

fimpara

para i <- 1 ate 4 faca // varrer a linha da matriz

para j <- 1 ate 3 faca // varrer a coluna da matriz

Escreva(matrizA[i,j])

fimpara

escreva("")

fimpara

Fimalgoritmo

algoritmo "MediaDe5Alunos"

// Função : Calcular a média das notas de 10 alunos e apresentar quem foi aprovado ou reprovado

// Autor : Gustavo

// Seção de Declarações

var

nomes: vetor [1..5] de caractere

notas: vetor [1..5,1..4] de real

medias: vetor [1..5] de real

contadorLoop1, contadorLoop2: inteiro

inicio

//Leitura dos nomes e as notas de cada aluno

PARA contadorLoop1 DE 1 ATE 5 FACA

ESCREVA("Digite o nome do aluno(a) número ", contadorLoop1, " de 5: ")

LEIA(nomes[contadorLoop1])

PARA contadorLoop2 DE 1 ATE 4 FACA

ESCREVA("Digite a nota ", contadorLoop2, " do aluno(a) ",
nomes[contadorLoop1], ": ")

LEIA(notas[contadorLoop1, contadorLoop2])

FIMPARA

//CÁLCULO DAS MÉDIAS

medias[contadorLoop1] := (notas[contadorLoop1, 1] + notas[contadorLoop1, 2] +
notas[contadorLoop1, 3] + notas[contadorLoop1, 4]) / 4

FIMPARA

//APRESENTAÇÃO DOS RESULTADOS

PARA contadorLoop1 DE 1 ATE 5 FACA

SE medias[contadorLoop1] >= 6 ENTAO

ESCREVAL("O aluno(a) ", nomes[contadorLoop1], " foi aprovado com as notas ("
notas[contadorLoop1, 1], ", ", notas[contadorLoop1, 2], ", ", notas[contadorLoop1, 3], ", ",
notas[contadorLoop1, 4], ") e média: ", medias[contadorLoop1])

SENAO

ESCREVAL("O aluno(a) ", nomes[contadorLoop1], " foi reprovado com as notas ("
notas[contadorLoop1, 1], ", ", notas[contadorLoop1, 2], ", ", notas[contadorLoop1, 3], ", ",
notas[contadorLoop1, 4], ") e média: ", medias[contadorLoop1])

FIMSE

FIMPARA

fimalgoritmo

Digite o nome do aluno(a) número 1 de 5: Gustavo

Digite a nota 1 do aluno(a) Gustavo: 9

Digite a nota 2 do aluno(a) Gustavo: 10

Digite a nota 3 do aluno(a) Gustavo: 9,5

Digite a nota 4 do aluno(a) Gustavo: 8

Digite o nome do aluno(a) número 2 de 5: João

Digite a nota 1 do aluno(a) João: 5

Digite a nota 2 do aluno(a) João: 6

Digite a nota 3 do aluno(a) João: 4,5

Digite a nota 4 do aluno(a) João: 7

Digite o nome do aluno(a) número 3 de 5: Pedro

Digite a nota 1 do aluno(a) Pedro: 7

Digite a nota 2 do aluno(a) Pedro: 8,5

Digite a nota 3 do aluno(a) Pedro: 6

Digite a nota 4 do aluno(a) Pedro: 7

Digite o nome do aluno(a) número 4 de 5: Luciana

Digite a nota 1 do aluno(a) Luciana: 10

Digite a nota 2 do aluno(a) Luciana: 7

Digite a nota 3 do aluno(a) Luciana: 7,5

Digite a nota 4 do aluno(a) Luciana: 8

MUITO
OBRIGADO

!!!