INF-0990 - Programação em C#

Aula Prática 1

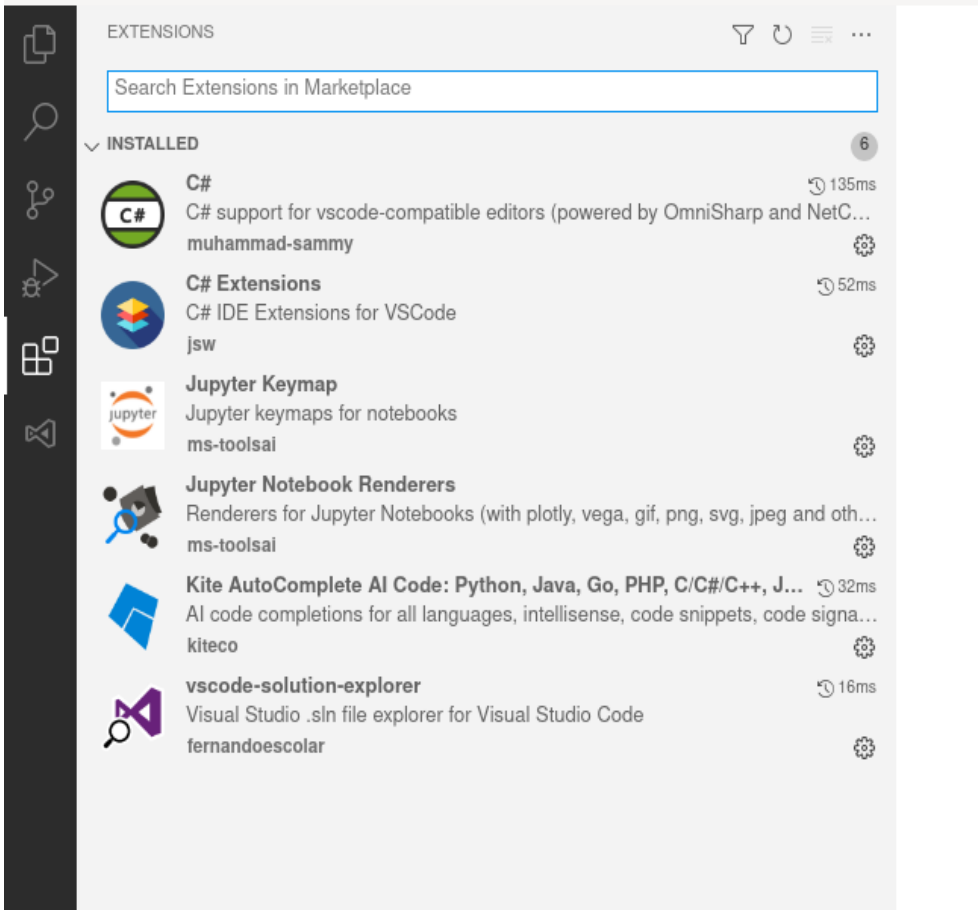
1) Instalação Visual Studio Code + Microsoft .NET 6

Inicialmente, instalaremos o Microsoft .NET SDK 6, acesse o link https://dotnet.microsoft.com/en-us/download/dotnet/6.0 e baixe a versão apropriada para o seu sistema operacional.

Instalaremos também o editor Visual Studio Code, acesse o link https://code.visualstudio.com/download e baixe a versão para o seu sistema operacional.

Uma vez instalados, abra o Visual Studio Code, acesse a aba lateral "Extensões" e instale as seguintes extensões:

- C#
- C# Extensions
- vscode-solution-explorer



Para criar um projeto C#, precisamos criar uma ***solution*** e associar pelo menos um **projeto** a ela. Podemos fazer isso diretamente por meio do vs-code ou utilizando a CLI do dotnet. O uso da CLI precisará que trabalhemos utilizando um Command Shell. Se estivermos utilizando o ambiente Windows, podemos usar o Windows PowerShell. Se estivermos em um ambiente MacOS ou Linux, podemos usar um outro shell, como por exemplo o bash. O primeiro passo, portanto, para usarmos a CLI é abrir o Command Shell. Uma descrição completa dos comandos disponíveis na CLI do dotnet podem ser visualizados aqui.

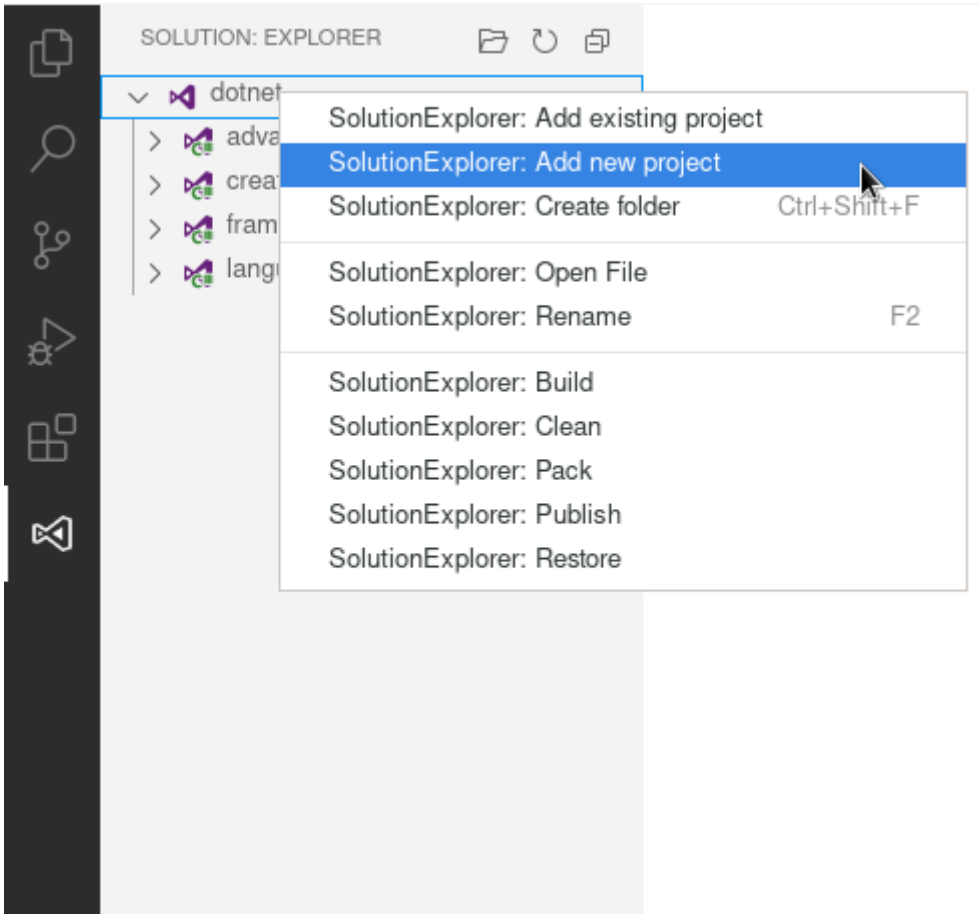
Inicialmente criamos um diretório, nos movemos para ele e podemos então criar uma solution:

```
mkdir Teste
cd Teste
dotnet new sln
```

Observe que um novo arquivo com a extensão .sln foi criado no diretório corrente. Esse arquivo descreve a solution criada e listará posteriores projetos que serão associados a ela.
Em seguida, podemos criar um novo projeto do tipo console e acomodá-lo no subdiretório <DIR>

```
dotnet new console --output <DIR>
```

Ou então diretamente no Visual Code:



Se não utilizarmos a extensão **--output <DIR>**, o novo projeto é criado no próprio diretório da solution. No caso de um único projeto por solution, isso não será um problema. Entretanto, caso a solution integre mais de um projetos, isso pode levar a uma estrutura desorganizada de diretórios. De um modo geral, como estratégia de engenharia de software, devemos sempre acomodar diferentes projetos em diferentes diretórios.
Observe que dois arquivos e um diretório foram criados. Um dos arquivos, com a extensão .cs é um template para seu programa C#. Você pode modificar seu conteúdo, desenvolvendo seu programa C# nele. O outro arquivo é um arquivo com a extensão .csproj, que é um arquivo XML que descreve detalhes do projeto a ser construído. Você pode customizar detalhes do projeto a ser construído, inserindo e modificando diretivas XML neste arquivo. Maiores informações sobre as diretivas que podem ser utilizadas podem ser encontradas aqui.
Por fim, é necessário acrescentar esse projeto à solution criada anteriormente:

```
dotnet sln add <DIR>
```

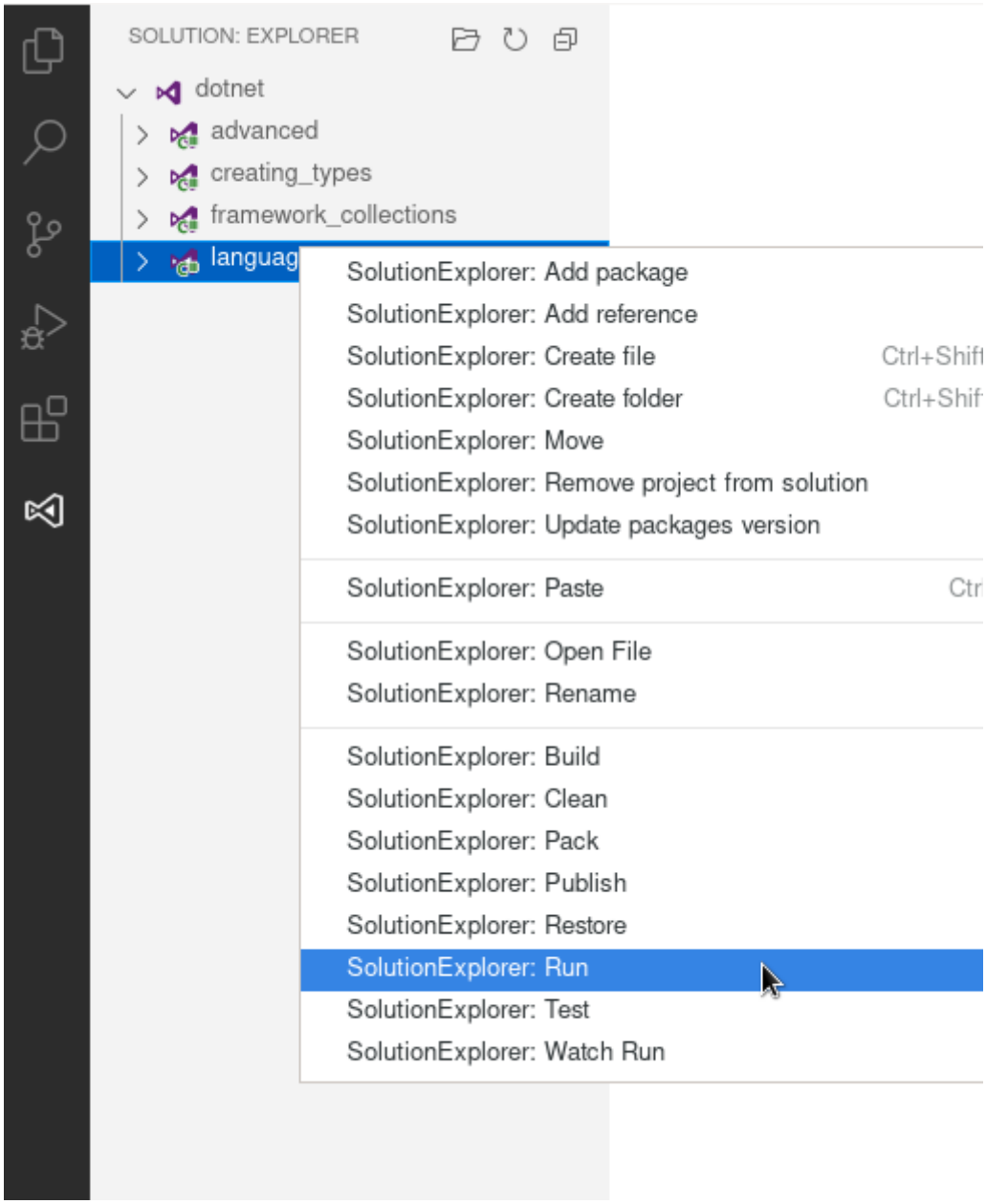
Uma vez que a solution tenha sido devidamente criada, e seus projetos configurados, é possível compilá-la usando:

```
dotnet build
```

Observe o resultado do build ... investigue os novos diretórios que foram criados e os arquivos que foram criados. Tente localizar onde está o arquivo executável e execute-o.

Observe que esse mesmo procedimento para criar solutions e projetos, e para compilá-lo, pode ser executado por meio dos menus do vs-code. Tente descobrir como fazê-lo.

Uma dica, clique com o botão direito do mouse sobre o projeto e várias opções de comandos aparecerão:



Da mesma forma, o vs-code disponibiliza um Command Shell para o teste das aplicações desenvolvidas por seu intermédio. Esse Command Shell pode ser disponibilizado no vs-code selecionando-se Terminal -> Novo Terminal. Durante a codificação, você poderá usar o atalho Ctrl+Espaço para abrir o assistente de escrita de código.

Por fim, um recurso muito valioso para a programação em C# é o .NET API Browser. Utilizando essa referência, podemos encontrar a documentação sobre todas as classes oferecidas no BCL (Base Class Library) do .NET.

2) Hello World em C#

Com o Visual Studio Code aberto, nós iremos codificar nosso primeiro programa em C#. Para isso, crie uma solução e dentro dessa solução adicione um novo projeto do tipo *Console App*. A seguir, abra o arquivo Program.cs gerado no vs-code, compile e verifique seu funcionamento. Esse é um exemplo de um programa mínimo em C#.

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

Apesar desse código ser um código válido em C#, essa não é a estrutura típica de um programa C#. Programas desse tipo, semelhantes ao de linguagens de script como Python ou Javascript só se tornaram possíveis a partir de versões mais modernas do C#. Para a aula de hoje, nos serviremos de programas desse tipo, somente para propósito didático. Na aula que vem, veremos como é a estrutura de um programa C# típico, e entenderemos como o compilador C# transforma programas simples como esse, na estrutura padrão de um programa C# típico, de maneira automática, dando a ilusão de que a estrutura básica não é necessária. Programas como os que construiremos na aula de hoje são importantes, entretanto, para o caso de scripts de apoio, muito úteis em diversas situações em que não queremos o trabalho de construir de fato um programa orientado a objetos mais complexo.

O comando *Console.WriteLine()* é usado para imprimir valores na tela. Não vamos entrar em detalhes, no momento, de como esse comando funciona. Para entendermos corretamente seu funcionamento, é necessário a compreensão de como funcionam os objetos e classes, assunto da nossa aula da semana que vem. Do ponto de vista técnico, Console é o que chamamos de uma classe estática, que possui um método WriteLine, que imprime valores na tela. O método WriteLine pode ser invocado por meio da sintaxe Console.WriteLine. Por enquanto, é tudo que precisamos saber. Na aula que vem, entenderemos melhor como esse comando funciona.

3) Tipos de Dados Primitivos

C# fornece vários tipos de dados primitivos como *int*, *double*, *long*, *bool*, *string*, entre outros. C# é originalmente uma linguagem fortemente tipada, o que significa que todas as variáveis devem ser precedidas pelo tipo a qual pertencem. No entanto, como veremos posteriormente, pode-se declarar variáveis sem especificar o tipo, e deixar que o compilador determine durante a compilação.

Os tipos de dados primitivos são, normalmente, o que chamamos de ***tipos por valor*** (*value types*). São usados tipicamente para representar variáveis contendo dados que não são objetos. Objetos, ao contrário, possuem um **tipo por referência** (*reference types*). A grande maioria dos tipos de dados primitivos são tipos por valor. O tipo *string*, entretanto, por razões históricas, é um tipo por referência.

3.1) Tipos de Dados Numéricos

Os tipos de dados inteiros com sinal são *sbyte*, *short*, *int*, *long*:

```
sbyte s = 20;
short t = -32;
int i = -1;
long l = -3000;
```

Os tipos de dados intereiros sem sinal são *byte*, *ushort*, *uint* e *ulong*:

```
byte b = 255;
ushort us = 2;
uint ui = 5;
ulong ul = 5000;
```

Os tipos de dados reais são *float*, *double* e *decimal*:

```
float f = 3.14F;
double d = 1.23;
decimal g = 2.32M;
```

A diferença entre eles é o número de bytes utilizado para codificar a notação em ponto flutuante. O *float* usa 4 bytes, o *double* usa 8 bytes e o *decimal* usa 16 bytes.

Sufixos podem ser usados para especificar o tipo de dado. Sendo eles: F = float, D = double, M = decimal, U = uint, L = long, UL = ulong.

```
long i = 5;            // Nenhum sufixo é necessário: Conversão implícita de int para
 long
double x = 4.0;        // O sufixo D é redundante
float f = 4.5F;        // Não irá compilar sem o sufixo F
decimal d = -1.23M;    // Não irá compilar sem o sufixo M
```

Tipos numéricos inteiros podem usar notação decimal ou hexadecimal; hexadecimal é denotado com o prefixo 0x:

```
int x = 127;
long y = 0x7F;
```

C# também permite especificar números binários com o prefixo 0b:

```
int b = 0b1010_1011_1100_1101_1110_1111;
```

Podemos inserir um *underscore* para deixar a leitura mais fácil:

```
int million = 1_000_000;

double doubleMillion = 1E06; // Notação exponencial é permitida com o sufixo E
```

Conversões são implícitas sempre que o destinatário pode representar o valor da origem. Caso contrário, uma conversão explícita é requerida:

```
int x = 12345;       // int é um inteiro de 32-bits
long y = x;          // Conversão implícita para inteiro de 64-bits
short z = (short)x;  // Conversão explícita para inteiro de 16-bits

// Todos os tipos inteiros podem ser implicitamente convertidos para tipos de ponto fl
utante:
int i = 1;
float f = i;

// O processo reverso deve ser explícito
int iExplicit = (int)f;
```

No entanto, implicitamente converter um tipo de dado inteiro para um tipo de ponto flutuante mantém a magnitude do valor preservada, porém há perda de precisão:

```
int i1 = 100000001;
float f1 = i1;          // Magnitude preservada, precisão foi perdida
int i2 = (int)f1;       // 100000000
```

Relembrando sempre que:

```
// O resto da divisão por inteiros é truncado
int a = 2 / 3;      // 0

// Divisão por zero lança exceção
int b = 0;
int c = 5 / b;      // throws DivisionByZeroException

// Por padrão, operações aritméticas overflow silenciosamente:
int a = int.MinValue;
a--;
Console.WriteLine(a == int.MaxValue);  // True
```

3.2) Tipos de Dados Booleanos

Tipos de dados booleanos podem ser implementados como:

```
bool a = true;
bool b = false;
```

Operadores de igualdade == e desigualdade != testam qualquer tipo de dado, mas sempre retornam um valor booleano:

```
int x = 1;
int y = 2;
int z = 1;

Console.WriteLine(x == y);    // False
Console.WriteLine(x != y);    // True
Console.WriteLine(x == z);    // True

Console.WriteLine(x < y);    // True
Console.WriteLine(x >= z);    // True
```

Operadores de interseção &&, união || e de exclusão ! operam sobre tipos booleanos:

```
bool UseUmbrella(bool rainy, bool sunny, bool windy)
{
  return !windy && (rainy || sunny);
}

UseUmbrella(true, false, false);  // True
UseUmbrella(true, true, true);    // False
```

Operadores && e || são essenciais para permitir a avaliação de expressões que possam retornar valores nulos, sem que haja exceções *NullReferenceException*:

```
StringBuilder sb = null;

if (sb != null && sb.Length > 0)
  Console.WriteLine("sb has data");
else
  Console.WriteLine("sb is null or empty");
```

O operador ternário possui a seguinte forma geral *q ? a : b*. Se a condição *q* é verdadeira, *a* é avaliado, caso contrário *b* é avaliado:

```
int Max(int a, int b)
{
  return (a > b) ? a : b;
}

Max(2, 3);   // 3
Max(3, 2);   // 3
```

3.3) Tipos de Dados Literais

Tipos de dados literais *char* representam um caracter Unicode e ocupam dois bytes em memória:

```
char c = 'A';       // Caracter simples
```

Uma string é representada por aspas duplas "". Strings são tipos de dados de referência (ponteiro), ao invés de valor:

```
string h = "Heat";
```

No entanto, seu operador de igualdade segue a mesma lógica semântica como se fosse uma variável de valor:

```
string a = "test";
string b = "test";
Console.WriteLine (a == b);  // True
```

Sequências de escape expressam caracteres que não podem ser interpretados literalmente, como quebras de linha, tabs, entre outros. Uma sequência de escape é representada pelo caractere \ seguido por outro caractere com significado próprio:

```
char newLine = '\n';  // Quebra de linha
char backSlash = '\\';

string t = "Here's a tab:\t"; // Tab
string escaped  = "First Line\r\nSecond Line"; // Escaped
```

Uma string verbatim, precedida pelo símbolo @, pode ocupar multiplas linhas:

```
string verbatim = @"First Line
                    Second Line";

// Adição de aspas duplas é feita escrevendo-as duplicadas:
string xml = @"id=""123""";
```

O operador + concatena duas strings. Quando o operando a direita não é uma string, o método *ToString* é chamado para fazer a conversão:

```
string s1 = "a" + "b"; // ab

string s2 = "a" + 5;   // a5
```

Uma string precedida pelo caractere $ é uma string interpolada:

```
int x = 4;
Console.WriteLine($"A square has {x} sides");    // A square has 4 sides

string s = $"255 in hex is {byte.MaxValue:X2}";  // X2 = 2-digit Hexadecimal

x = 2;
s = $@"this spans {
        x} lines";
```

Interpolação de strings e constantes são permitidas a partir do C# 10:

```
const string greeting = "Hello";
const string message = $"{greeting}, world";

Console.WriteLine(message);
```

3.4) Tipos de Dados Matriciais (*arrays*)

Um vetor representa um conjunto de elementos de um tipo particular. O tamanho de um vetor é sempre definido na sua inicialização:

```
char[] vowels = new char[5];    // Um vetor de tamanho 5 do tipo char
```

Para acessar um elemento específico do vetor, usamos o processo de indexação, no qual a posição desejada é informada:

```
vowels[0] = 'a';
vowels[1] = 'e';
vowels[2] = 'i';
vowels[3] = 'o';
vowels[4] = 'u';
Console.WriteLine(vowels[1]);      // e
```

O primeiro elemento de um vetor possui o índice 0. Vetores também podem ser inicializados de um modo mais simples:

```
char[] easy = {'a','e','i','o','u'};
```

Os valores dos elementos de um vetor são sempre inicializados com valor 0 por padrão:

```
int[] a = new int[1000];
Console.Write(a[123]);    // 0
```

C# permite o uso de índices mais sofisticados para acessar determinados elementos:

```
char[] vowels = new char[] {'a','e','i','o','u'};
char lastElement  = vowels[^1];   // 'u'
char secondToLast = vowels[^2];   // 'o'

Index first = 0;
Index last = ^1;
char firstElement = vowels[first];   // 'a'
char lastElement2 = vowels[last];    // 'u'
```

Também são permitidos o uso de intervalos para acessar uma fatia do vetor:

```
char[] vowels = new char[] { 'a', 'e', 'i', 'o', 'u' };

char[] firstTwo = vowels[..2];     // 'a', 'e'
char[] lastThree = vowels[2..];    // 'i', 'o', 'u'
char[] middleOne = vowels[2..3];   // 'i'

char[] lastTwo = vowels[^2..];     // 'o', 'u'

Range firstTwoRange = 0..2;
char[] firstTwo2 = vowels[firstTwoRange];   // 'a', 'e'
```

Matrizes podem ser instanciadas como blocos de memória com n-dimensões:

```
int[,] matrix = new int[3, 3];    // Vetor de inteiros com duas dimensões

for (int i = 0; i < matrix.GetLength(0); i++) // O metodo GetLength retorna o tamanho
 de um vetor
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix[i, j] = i * 3 + j;
```

Matrizes também podem ser inicializadas com valores:

```
int[,] matrix2 = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

Formas simplificadas de inicialização de variáveis removem o operador *new*:

```
char[] vowels = {'a','e','i','o','u'};

int[,] rectangularMatrix =
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};

int[][] jaggedMatrix =
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

Todos os vetores/matrizes são checados com relação a sua indexação:

```
int[] arr = new int[3];
arr[3] = 1;               // Exceção IndexOutOfRangeException
```

4) Operadores e Expressões

4.1) Operador null

Variáveis com tipo por referência podem assumir um valor *null*. Isso significa que não estão apontando para um objeto válido.

```
string s1 = null;
string s2 = s1 ?? "nothing";   // s2 é avaliado para "nothing"
```

O operador ?? é chamado de null-coalescing e caso tenhamos a ?? b ele retorna a se a não for nulo, ou b, caso contrário.

```
string s1 = null;
s1 ??= "something";
Console.WriteLine(s1);  // something

s1 ??= "everything";
Console.WriteLine(s1);  // something
```

O operador ??= é semelhante a ?? mas se temos a ??= b caso a seja null, ele substitui seu valor por b, mantendo o valor de a original, caso contrário.

```
string sb = null;
string s = sb?.ToString();              // Sem erros, nesse caso, s é avaliado para nu
lo
```

Nesse caso, caso a variável sb seja null, ele não invoca o método, retornando null

```
string sb = null;
int? length = sb?.ToString().Length;      // OK : int? pode ser nulo
string s = sb?.ToString() ?? "nothing";   // s é avaliada para "nothing"
```

Exercícios

1) Reescreva os trechos de código abaixo usando a notação do operador nulo.

```
if (valor != null)
{
     valor = valor.Trim().ToUpper();
}
```

```
int? tamanho = (pessoa != null) ? (int?)pessoa.Length : null;
```

4.2) Definição de Variáveis

Variáveis de um mesmo tipo podem ser declaradas na mesma linha, desde que separadas por vírgula:

```
string someWord = "rosebud";
int someNumber = 42;
bool rich = true, famous = false;
```

Constantes são declaradas com a palavra-chave *const*. Após declaradas, constantes não podem ser mais alteradas:

```
const double c = 2.99792458E08;
```

A seguir, apresentamos algumas expressões comumente usadas:

```
x = 1 + 2;                 // Atribuição
x++;                       // Incremento após uso
x--;                       // Decremento após uso
++x;                       // Incremento antes do uso
--x;                       // Decremento antes do uso
y = Math.Max(x, 5);        // Atribuição
Console.WriteLine(y);      // Chamada de método
sb = new StringBuilder();  // Atribuição
new StringBuilder();       // Instanciação de objeto
```

Exercícios

1) Declare uma variável inteira sem atribuir nenhum valor a ela. Tente imprimir na tela. O que acontece?

2) Faça um programa que altera o valor de uma constante. O que acontece?

3) Faça um programa que incremente o valor de um *char*. O que acontece?

4.3) Tipagem Implícita (var)

C# fornece uma forma de omitir a tipagem de variáveis em código e deixar que o compilador faça a inferência do tipo durante a compilação. Para isso usamos a palavra-chave *var*:

```
var i = 3;           // i é implicitamente do tipo int
var s = "sausage";   // s é implicitamente do tipo string

var rectMatrix = new int[,]    // rectMatrix é implicitamente do tipo int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};

var vowels = new[] {'a','e','i','o','u'};   // Compilador infere o tipo char[]
var x = new[] { 1, 10000000000 };           // Legal - todos os elementos são converti
dos para o tipo long
```

Exercícios

1) Declare e inicialize uma variável inteira sem especificar o tipo. Tente atribuir uma valor literal a essa variável. O que acontece?

2) Declare e inicialize uma variável inteira sem especificar o tipo. Tente somar um *char* a essa variável. O que acontece?

5) Laços de Controle de Fluxo e de Repetição

5.1) If, else

O comando de controle de fluxo mais simples e utilizado é o *if*. No qual uma expressão booleana é avaliada no início do laço, caso retorne verdadeira, o corpo do laço é executado:

```
if(5 < 2 * 3)
{
  Console.WriteLine("true");       // True
}
```

Se a expressão avaliada retornar falso, podemos direcionar a execução para o trecho de código alternativo, definido dentro da cláusula *else*:

```
if(2 + 2 == 5)
{
  Console.WriteLine("Does not compute");
}
else
{
  Console.WriteLine("false");      // False
}
```

Caso mais de uma expressão precise ser avaliada, podemos adicionar comandos *else if* conforme necessário:

```
if(2 + 2 == 5)
  Console.WriteLine("Does not compute");
else if(2 + 2 == 4)
  Console.WriteLine("Computes");   // Computes
```

Não será necessário definir o escopo do laço com chaves { } quando o corpo do laço possuir apenas um comando.

Exercícios

Para os exercícios a seguir, utilizaremos outro método da classe Console, que é o ReadLine. Com ele, podemos ler um string digitado pelo usuário. O método ReadLine retorna uma string. Para usarmos os valores numericamente, podemos converter essa string em um valor, usando o método Parse, como no exemplo a seguir:

```
string meustring = Console.ReadLine();
int n = int.Parse(meustring);
Console.WriteLine(n);
```

Tente fazer agora os exercícios a seguir:

1) Faça um programa que leia o ano de nascimento do usuário, imprima na tela se ele poderá votar esse ano.

2) Faça um programa que leia três números binários, imprima na tela o maior e o menor deles.

3) Faça um programa que leia as medidas de um triângulo (decimal inteiro), imprima na tela se ele é Equilátero, Isósceles ou Escaleno. Imprima também a área do triângulo com 3 casas decimais de precisão.

5.2) Switch

O comando *switch* fornece um modo mais claro e simples de controlar o fluxo de dados, ao invés do encadeamento de múltiplos comandos *if*:

```
ShowCard (5);
ShowCard (11);
ShowCard (13);

static void ShowCard(int cardNumber)
{
  switch(cardNumber)
  {
    case 13:
      Console.WriteLine("King");
      break;
    case 12:
      Console.WriteLine("Queen");
      break;
    case 11:
      Console.WriteLine("Jack");
      break;
    case -1:
      goto case 12;
    default:
      Console.WriteLine(cardNumber);
      break;
  }
}
```

Quando a avaliação de mais de um valor leva a mesma execução, é possível listar vários *cases* sequencialmente:

```
int cardNumber = 12;

switch(cardNumber)
{
  case 13:
  case 12:
  case 11:
    Console.WriteLine("Face card");
    break;
  default:
    Console.WriteLine("Plain card");
    break;
}
```

Exercícios

1) Faça um programa que leia dois números do teclado, juntamente com a operação desejada (soma, subtração, divisão, multiplicação). Execute a operação e mostre na tela. Use o comando *switch* na sua solução.

2) Faça um programa que imprima a estação do ano (verão, inverno, outono, primavera) de acordo com o número do mês informado. Use o comando *switch* na sua solução.

3) Faça um programa de conversão de base numérica, leia um número decimal inteiro e a base desejada para conversão (octal, hexadecimal, binária). Use o comando *switch* na sua solução.

5.3) While, do

Em um laço de repetição *while*, a expressão de controle é avaliada antes que o corpo do laço seja executado:

```
int i = 0;
while(i < 3)
{
  Console.WriteLine(i);
  i++;
}
```

Em um laço *do-while*, a expressão de controle é avaliada somente no final. Assim, o corpo do laço executa ao menos uma vez:

```
int i = 0;
do
{
  Console.WriteLine(i);
  i++;
}
while(i < 3);
```

Exercícios

1) Faça um programa que leia um número do teclado até encontrar um número menor ou igual a 1. Ao final, mostre a soma de todos os números digitados.

2) Faça um programa que verifica se duas strings são iguais.

5.4) For, foreach

O laço de repetição *for* permite definir quantas vezes um determindo trecho de código será executado:

```
for(int i = 0; i < 3; i++)
  Console.WriteLine(i);
```

É possível adicionar mais de uma variável na inicialização do laço:

```
for(int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
{
  Console.WriteLine(prevFib);
  int newFib = prevFib + curFib;
    prevFib = curFib; curFib = newFib;
}
```

O comando *foreach* permite a iteração sobre elementos enumeráveis, neste caso, *System.String* implementa a interface *IEnumerable*:

```
foreach(char c in "beer")   // c é a variável de iteração
  Console.WriteLine(c);
```

Exercícios

1) Faça um programa que imprima a tabuada de 1 até 9.

2) Faça um programa que imprima todos os números primos até 100.

3) Faça um programa que determina se uma string é um palíndromo, palavra que ser lida indiferentemente da esquerda para a direita ou vice-versa. Implemente sua solução com o comando *foreach*.

5.5) Break, continue

O comando *break* termina a execução do laço de repetição, usualmente usado dentro de laços *for*, *while* e *switch*:

```
int x = 0;
while(true)
{
  if(x++ > 5)
    break ;      // quebra o laço
}
```

O comando *continue* faz com que os comandos subsequentes do laço sejam desconsiderados. Assim, o laço reinicia na próxima iteração:

```
for(int i = 0; i < 10; i++)
{
  if((i % 2) == 0)    // se i é par
    continue;         // continua para a próxima iteração

  Console.Write(i + " ");
}
```

Exercícios

1) Faça um programa que imprima o primeiro número, entre 1 e 1000, que seja divisível por 7, 13, 17. Implemente sua solução com o comando *break*.

2) Faça um programa que some todos os números de 1 até 1000, exceto os múltiplo de 5. Implemente duas versões: com e sem o comando *continue*.

Última atualização: sexta, 26 Ago 2022, 23:36

INF-0990

Participantes

Emblemas

Competências

Notas

Plano de Desenvolvimento da Disciplina

Atividade 1: 27/08/2022 08:30-12:30

Aula Teórica 1

Aula Prática 1

Atividade 2: 03/09/2022 – 08:30-12:30

Atividade 3: 10/09/2022 – 08:30-12:30

Atividade 4: 10/09/2022 – 13:30-17:30

INF-0991

INF-0992

INF-0993

INF-0994

INF-0995

INF-0996

INF-0997

INF-0998

INF-0999

ADMINISTRAÇÃO

Administração do curso
