cencal centro de formação profissional para a indústria de cerâmica

UFCD 3935

Programação em C#

Formador:

António Ramos

Ação:

Programador de Informática - 481TI827.22

Co-Financiado por:

IEFP APICER associação portuguesa das indústrias de cerâmica e cristalaria ARO Associação Empresarial da Região Oeste | REPÚBLICA PORTUGUESA POCH PORTUGAL 2020 UNIÃO EUROPEIA Fundo Social Europeu

Índice

Co-Financiado por:

Co-Financiado por:

Co-Financiado por:

O que é?

É uma linguagem de programação orientada a objetos desenvolvida pela Microsoft, sendo inicialmente parte da *Framework* .NET e com o propósito de ser uma linguagem de programação genérica que permita ser usada para o desenvolvimento de diferentes tipos de aplicações, desde a área *Mobile*, Gaming, *Console*, *Desktop* ou *Web*.

É utilizado um *compiler* que vai converter o código para a "linguagem máquina" para depois ser executado pelo computador.

Vantagens?

O C# tem funcionalidades e uma *sintax* semelhantes a outras linguagens de programação. Adicionalmente está integrado com a *framework opensource* .NET, que inclui uma biblioteca de pacotes, componentes e de *namespaces* que permitem aos programadores reutilizarem códigos desenvolvidos nativamente pela Microsoft, permitindo desenvolver as aplicações mais eficiente e rapidamente.

O C# é uma linguagem de Programação Orientada a Objetos (POO), permitindo partir o programa em objetos interativos entre eles.

.NET

Inicialmente o .NET era uma *framework* de código fechado, proprietária da Microsoft. A partir de 2014, esta foi disponibilizado publicamente e tornada *opensource*, sendo rescrita para uma *framework* aberta que permitisse ser executada em todos os sistemas operativos.

Tipos de Aplicações

Com .NET podem ser desenvolvidas aplicações de forma rápida, com base em bibliotecas já existentes. Assim, podem ser desenvolvidas os seguintes tipos de apps:

- *Web*
- *Mobile*
- *Desktop*
- *Microservices*

Co-Financiado por:

- *Cloud*
- *Machine Learning*
- *Game Development*
- *Internet of Things*

Co-Financiado por:

Elementos de um programa em C#

Namespaces

C# contém um conjunto de blocos de código e métodos já desenvolvidos e divididos em diferentes e contextualizados *namespaces*. Por exemplo, o *namespace System*, contém métodos que nos permite interagir com os utilizadores.

Podemos também criar os nossos *namespaces* e uma vantagem de utilizarmos os *namespaces* é a prevenção de conflitos de nomenclatura, desde que o método com o mesmo nome pertença a outro *namespace*.

Using directives

Permite simplificar a utilização de *namespaces* sem especificar o *namespace* qualificado desse tipo. Ao usarmos, estamos a dar indicação ao compilador que o nosso programa utiliza o respetivo *namespace*.

Método *Main*()

O método *Main*() é o ponto de entrada de todas as Aplicações de C*onsole*. Quando a aplicação Console é iniciada, o método *Main*() é o primeiro a ser chamado.

Dentro dos parêntesis do método *Main*(), pode ser ou não indicados os argumentos passados para dentro do método.

Comentários

As linhas de comentários não são parte do programa, sendo ignoradas pelo compilador. Estas servem para os programadores efetuarem "comentários" tornando os blocos de código mais legíveis para os programadores.

Co-Financiado por:

Variáveis e Operadores

O que são variáveis?

As variáveis representam localizações na memória. Cada variável tem um tipo que determina que valores podem ser guardados nessa variável. Basicamente, são nomes que se dão a dados para serem guardados e manipulados nos programas.

A declaração inicial identifica o tipo da variável, seguindo do seu nome. Após a sua declaração, o programa aloca na memória do computador uma parte do espaço necessário para alocar a informação da variável. Posteriormente estas podem ser acessíveis e modificadas sendo efetuada a referência ao seu nome.

Tipos de variáveis

A listagem abaixo refere aos tipos de variáveis mais comuns na generalidade dos programas desenvolvidos em C#:

- *Int*
  - Números inteiros (sem casas decimais ou partes fracionadas), vai desde o número -2 147 483 648 até ao 2 147 483 647
- *Byte*
  - Utilizado para números inteiros, mas com menor diferença (0 a 255) reservando assim um espaço inferior na memória.
- *Float*
  - Usado para números decimais, com uma aproximação de 7 casas decimais
- *Double*
  - Usado para números decimais, mas com uma aproximação maior até 10 casas decimais.
  - Este tipo é o *default data type* para números com casas decimais pelo C#.
- *Decimal*
  - Utilizado para programas com um elevado grau de precisão, este tipo de dados guarda números até 28 casas decimais. No entanto, oferece um menor alcance que os tipos anteriores.
- *Char*
  - É utilizado para guardar caracteres *Unicode*.
- *String*
  - É utilizado para guardar texto. Internamente o texto é guardado numa coleção sequencial de objetos *Chars*.

Co-Financiado por:

- *Bool*
  - É usado para estados binários e apenas guarda um dos dois valores (verdadeiro ou falso).

Inicialização de variáveis

A inicialização da variável é efetuada na declaração de uma nova variável, para que a esta seja atribuída um valor inicial. É boa prática as variáveis serem inicializadas, podendo estas serem alteradas ao longo do programa.

Operadores

Além da atribuição de um valor ou atribuição de outra variável, podem ser efetuadas operações matemáticas. Vejamos os exemplos abaixo:

```
// Atribuição de um valor
x = 5

// Atribuição de uma variável
x = y

// Adição
x + y

// Subtração
x - y

// Multiplicação
x * y

// Divisão
x / y

// Modulo
x % y
```


Co-Financiado por:

Arrays, Strings e Lists

Array

O *Array* é uma coleção/grupo de dados relacionados entre si. Em vez de serem guardados dados individualmente em variáveis, estas são guardadas num *array*.

Este é declarado e iniciado pela seguinte forma:

```
int[] usersAge = {18, 19, 21, 26, 30};
```

Os parêntesis retos [] indicam ao compilador que esta variável é um *array* em vez de uma variável dita normal. Dentro dos parêntesis { }, estão os 5 inteiros que serão guardados dentro do *array*.

A declaração e inicialização da variável pode ser efetuada separadamente, sendo efetuada primeira a declaração e depois a inicialização, no entanto é necessário saber na declaração o número de valores a guardar.

```
int[] usersAge = new int[5];
usersAge = new [] {18, 19, 21, 26, 30};
```

Os valores individuais dentro do *array*, são acessíveis através do seu índice, sendo que estes iniciam sempre no zero. Isto é, o primeiro valor do *array* (no exemplo acima 18) é acessível pelo acesso ao índice ZERO.

```
Console.WriteLine(usersAge[0]);
```

Propriedades e Métodos

O C# contém um conjunto de propriedades e métodos que se podem utilizar com os arrays. É utilizado o ponto final (.) para acedermos aos métodos ou propriedades das classes, sendo que no caso dos métodos, são abertos uns parênteses.

Length

A propriedade *Length* devolve o número de itens do array,

Sort()

O método Sort() permite ordenar os itens do array.

Co-Financiado por:

IndexOf()

Este método é utilizado para determinar se certo valor existe dentro do *array*, devolvendo o número do *index* onde este está. Caso não exista, este devolve -1;

String

O *data type String* pode ser considerado como um pedaço de texto. Para declarar e iniciar a variável usamos atribuição com o valor dentro de aspas. O sinal (+) permite concatenar duas *Strings*, juntando assim os vários pedaços de texto.

```
string mensagem = "Hello World";
```

é igual

```
string mensagemConcat = "Hello" + " " + "World";
```

Propriedades e Métodos

Length

Devolve o número de caracteres que a *String* contém.

Substring()

Método que permite extrair parte de uma String. Esta requer dois argumentos, a posição inicial do caracter a extrair e o comprimento de caracteres.

Equals()

Método que verifica se duas stings são iguais.

Co-Financiado por:

Lists

As listas são usadas para quando é necessária mais flexibilidade. Estas servem para guardar valores (como um array), mas como não é necessário indicar o número de valores a serem guardados no array ou inicializá-los na sua declaração, torna as listas mais versáteis.

```
List<int> usersAgeList = new List<int>();
```

Propriedades e Métodos

Add()

Método que permite adicionar novos elementos no fim da lista.

Count

Propriedade que devolve o número de elementos da lista.

Insert()

Método que permite adicionar novos elementos na posição indicada via parâmetro.

Clear()

Método que remove todos os itens da lista.

Co-Financiado por:

Tipos de valor vs. Tipos de referência

Todos os tipos de dados em C# podem ser classificados em "Tipos de valor" ou "Tipos de referência"

O "Tipo de valor" são as variáveis que guardam o seu próprio valor. Enquanto que o "Tipo de referência", não guarda o valor mas sim, uma referência à informação. Na prática, este diz ao compilador onde ir buscar o valor e não o valor em si.

O tipo "String" é um reference type, enquanto a maioria dos outros são nativamente de valor. No caso da String, a variável irá conter o endereço de memória onde está guardado o valor.

As variáveis do "Tipo de Referência" por defeito têm o valor de *null*, que indica que o valor da variável ainda não refere nenhum valor.

Operadores de comparação

A estrutura de seleção condicional envolve a utilização de declarações condicionais. O programa irá diferenciar se certa condição é (ou não) correspondida ou não.

Desde a comparação de variáveis e se os valores são maiores ou menores, estas e outros operadores podem são usados nas condições.

Não é Igual (!=)

Retorna "*true*" se a variável da esquerda é igual à da direita

Maior que (>)

Retorna "*true*" se o valor da esquerda for superior ao da direita

Menor que (<)

Retorna "*true*" se o valor da esquerda for menor ao da direita

Maior ou igual do que (>=)

Retorna "*true*" se o valor da esquerda for maior ou igual ao valor da direita

Co-Financiado por:

Smaller than or equal to (<=)

Retorna "*true*" se o valor da esquerda for menor ou igual ao valor da direita

Operador E (&&)

Operador que permite combinar múltiplas condições. Este Retorna "*true*" se todas as condições forem correspondidas

Operador OU (||)

Operador que permite combinar múltiplas condições. Retorna "*true*" se pelo menos uma condição for verdadeira.

Instruções de Controlo de Fluxo

As declarações de controlo, funcionam em conjunto com as operadores de comparação, permitindo assim adicionarmos lógica e controlar o fluxo do nosso programa.

Instrução IF

Permite ao programa avaliar se certa condição é correspondida, sendo posteriormente efetuada a ação a que esta corresponde.

```
if (condition){
  // block of code to be executed if the condition is True
}
```

IF "inline" (versão curta)

Também conhecido por operador ternário, a instrução "inline" pretende simplificar se é conveniente utilizar ou atribuir um valor dependendo do resultado da condição.

```
variable = (condition) ? expressionTrue : expressionFalse;
```


Co-Financiado por:

Instrução Switch

A instrução Switch é semelhante à anterior, no entanto esta é indicada para casos onde a comparação é directa a um único valor, e não uma variedade de condições. O caso "*default*" é opcional e apenas corre se nenhuma das condições anteriores for correspondida.

Quando alguma condição for satisfeita, o que estiver dentro do "*case*" é executado pelo programa. A keyword "*break*" é usada para o compilador saltar a condição.

```
switch(expression)
{
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
    break;
}
```

Ciclo For

O ciclo "*for*" executa um bloco de código repetidamente até que a condição não seja válida. Utilizamos quando sabemos o número de vezes que queremos correr a condição.

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


Co-Financiado por:

A condição que compõem o ciclo for é dividida em 3 partes pelo ponto e vírgula (;). A primeira inicializa e declara a variável a zero, servindo de contador.

A segunda parte, testa e verifica se a variável é menor de 5. Caso positiva, então irá correr o bloco de código, sendo a variável declarada na condição passível de ser usada dentro do bloco.

A terceira parte, é ação que ocorre em cada ciclo "*for*". No exemplo acima, estamos a indicar que a variável irá ser incrementada por um.

Ciclo Foreach

Ciclo que pode ser usado em listas e arrays. É vantajoso quando queremos obter informação da lista, sem efetuar qualquer alteração.

```
foreach (type variableName in arrayName){
  // code block to be executed
}
```

Ciclo While

Permite executar blocos de código dentro do ciclo enquanto a condição permanecer válida.

```
while (condition){
  // code block to be executed
}
```

É necessário existir uma variável que sirva de contador, para que esta seja reduzida e que cumpra a condição dentro do while. Caso contrário, este bloco de código corre infinitamente.

Ciclo Do

É semelhante ao ciclo While, no entanto o bloco código dentro dos parênteses curvos apenas é executado uma vez.

```
do{
  // code block to be executed
```


Co-Financiado por:

```
}
while (condition);
```

A condição while deste ciclo deve cumprir apenas uma vez, para que garanta a sua execução.

Instruções de salto

As instruções de salto indicam ao programa para desviar do fluxo normal da sequência e saltar para outro bloco de código. Estes são normalmente usados em ciclos ou em fluxos de controlo.

Break

A *keyword* "*break*" faz sair do *loop* quando certa condição é correspondida. Esta pode ser utilizada nos ciclos

Continue

A *keyword "continue"* dá indicação ao programa para avançar para o ciclo seguinte quando a keyword é atingida.

```
for (int i = 0; i<5; i++)
{
    Console.WriteLine("i = {0}", i);
    if (i == 2) continue;
    Console.WriteLine("I will not be printed if i=2.\n");
}
```

Return

Termina a execução da função e retorna o controlo ou o resultado para a função para a execução onde foi chamada a função.

```
Console.WriteLine("First call:");
```


Co-Financiado por:

```
DisplayIfNecessary(6);

void DisplayIfNecessary(int number){

    if (number % 2 == 0)

    {

        return;

    }

    Console.WriteLine(number);

}
```

Gestão de erros e exceções

As exceções deverão ser prevenidas no nosso programa para evitarmos que estes parem ou que tenham resultados inesperados, para registar eventos de erros e/ou permitir que estes continuem uma correta execução após surgirem.

O C# fornece apoio para tratar as exceções através de blocos *try-catch-finally* que controlam como o programa prossegue quando o erro ocorre.

O código abaixo exemplifica uma aplicação simples da gestão de erros, no entanto a instrução *Catch* pode tratar erros específicos.

```
try {
    // put the code here that may raise exceptions
}
catch {
    // handle exception here
}
finally {
    // final cleanup code
}
```

Gestão de exceções específicas

A gestão especifica de erros, é útil quando pretendemos lógica quando o erro acontece (ex.: mostrar uma mensagem de erro).

Co-Financiado por:

```
int choice = 0;
int[] numbers = { 10, 11, 12, 13, 14, 15 };
Console.Write("Please enter the index of the array: ");

try
{
    choice = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("numbers[{0}] = {1}", choice, numbers[choice]);
}
catch (IndexOutOfRangeException)
{
    Console.WriteLine("Error: Index should be from 0 to 5.");
}
catch (FormatException)
{
    Console.WriteLine("Error: You did not enter an integer.");
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

No bloco de código acima, a primeira exceção (*IndexOutOfRangeException*) é tratada no primeiro bloco de *try-catch.* Esta acontece quando é acedemos ao *index* inexistente no *array*.

No caso do segundo erro (*FormatException*) ocorre caso o utilizador algo, que não consiga ser convertido para inteiro (Ex.: Ola).

Por fim, o último bloco, apanha todos os outros erros não especificados anteriormente e mostra ao utilizador.

Co-Financiado por:

Programação Orientada a Objetos

O que é?

É uma abordagem que divide o nosso programa em vários objetos que se vão interagir entre si. Os objetos serão criados através de "*templates*" ou "plantas" que se chamam classes. Ao construirmos o nosso programa, vamos instanciar várias "plantas" que depois irão compor o nosso programa.

Criar uma classe

Para criar uma classe, usamos a keyword "class" antes do nome da mesma. É boa prática utilizar a convenção PascalCasing para a nome das classes. No interior da classe, é definido todo o conteúdo, campos, propriedades e métodos da classe.

```
class Pessoa {
    // Campos, Propriedades e Métodos
    private string nomeDaPessoa;
    private int numeroFiscal;
}
```

Campos

A declaração dos campos das classes, é definindo como variáveis, sendo posteriormente acessíveis ou não consoante a keyword que se coloca antes da definição da variável. No exemplo acima, a keyword "*private*" indica ao programa que esta variável apenas é acessível dentro da mesma classe, não sendo acessível por outras. Os campos podem ainda ser "*public*" e "*protected*" ou "*internal*". Estas keywords são denominadas como "*access modifier*".

Propriedades

São utilizadas para dar acesso a campos privados quando são precisos por outras classes. Desta forma, temos mais controlo sobre o direito à informação aos campos "*private*".

Declaramos as propriedades indicando o *access modifier* e o *data type* da propriedade.

Co-Financiado por:

```
public int NumeroFiscal
{
    // get
    // set
}
```

As propriedades têm dois métodos especiais denominados como "*accessors*", o *getter* e o *setter*. O getter retorna o valor do campo e o setter define o valor do campo.

```
get{
    return numeroFiscal;
}

set{
    numeroFiscal = value;
}
```

A keyword ***value*** refere-se ao valor dado na instrução de atribuição do setter.

Por defeito o getter e setter tem o mesmo nível de acesso que a propriedade, no entanto podemos declarar os mesmos como privados usando a keyword *private*. Assim tornamos a propriedade apenas de leitura fora da classe principal.

Propriedades auto-implementadas

Estas forma de implementar as propriedades utilizam-se quando não é necessário utilizar lógica adicional ao getter e setter. Usando a versão "shorthand" não é necessário definir os campo "*private*", pois o compilador cria-os automaticamente.

```
public int NumeroFiscal { get; set; }
```

Métodos

Os métodos são blocos de códigos que efetuam uma determinada tarefa. Estes são criados dentro da classe, sendo primeiro indicado o nível de acesso do método e depois o tipo de dados que o método retorna.

Co-Financiado por:

```
public void BoasVindas()
{
    Console.WriteLine("Bem-vindo!! :)");
}
```

Os parêntesis à frente do método, indica ao compilador que esta função é um método que não recebe parâmetros. No entanto, esta pode receber N parâmetros e no método colocarmos a lógica que pretendermos.

Overloading

Overloading refere-se à utilização de métodos com o mesmo nome, mas com assinaturas diferentes (assinaturas entende-se como os parâmetros de entrada do método).

```
public void BoasVindas(string nome)
{
    Console.WriteLine("Bem-vindo {0}!! :)", nome);
}
```

A assinatura do primeiro método é BoasVindas() enquanto a assinatura do segundo método é BoasVindas(string nome), assim podemos utilizar qualquer um dos métodos consoante os parâmetros pretendidos e com lógicas diferentes.

Construtores

São métodos específicos para a criação de objetos do tipo da classe. É o primeiro método a ser chamado para criar o objeto e são utilizados normalmente para inicializar os métodos das classes.

Este deve ter sempre o nome da Classe e não deve retornar nenhum valor (não necessita de declarar a keyword do datatype de retorno).

Co-Financiado por:

```
public Pessoa(int nif)
{
      numeroFiscal= nif;
      Console.WriteLine($"NIF = {numeroFiscal}");
}
```

Caso não seja declarado um construtor, o compilador cria um automaticamente, sendo que inicializa todos os campos da classe ao valor por defeito (ex: 0 a números inteiros, strings vazias).

Instanciar um objeto

Dentro do nosso programa, devemos instanciar as classes que serão utilizadas. O dataType será o nome da classe, seguindo da keyword "new" e invocado o método costrutor da classe.

```
NomeClasse nomeObjectoX = new NomeClasse(argumentos);
```

Após a criação do objeto, usamos o ponto final para acedermos aos campos, propriedades ou métodos da classe.

Keyword "Static"

Para a criação de classes ou métodos/campos sem a criação de objetos, utilizamos a keyword "*static*". No entanto, caso a classe seja "*static*" todos os membros da classe deverão ser estáticos.

```
public static class Calculadora
{

}
```

Parâmetros (Tipos de Valor VS Tipos de Referência)

Quando passamos uma variável do tipo de valor para um método, qualquer alteração dessa variável será feita apenas dentro do mesmo método. Quando o programa sai do método, a alteração deixa de ser válida.

Co-Financiado por:

Em contrário, se passamos a referência do variável, qualquer alteração é válida, mesmo depois do término do método.

```
public void PassagemPorValor(int a)
{
    a = 10;
    Console.WriteLine("valor de a = {0}", a);
}

public void PassagemPorReferência(int[] b)
{
    b[0] = 5;
    Console.WriteLine("valor de b[0] = {0}", b[0]);
}
```

Caso seja pretendido que variáveis do tipo de valor sejam passadas como referência, deve ser usado a keyword "*ref*" no parâmetro do método.

```
public void PassagemPorValor(ref int a)
{
    a = 10;
    Console.WriteLine("valor de a = {0}", a);
}
```

Herança

Permite a criação de uma classe, a partir de uma classe já existente, permitindo a extensão, reutilização e modificação do comportamento definido em outras classes.

A classe cujos membros são herdados é definido como classe "*base*" e a classe que herda esses membros é classe "*derivada*".

```
class Membro
{
    protected int feeAnual;
    private string nome;
    private int membroID;
    private int membroDesde;
}
```


Co-Financiado por:

Classes derivadas (Subclasses)

As classes derivadas herdam todas os membros públicos e "*protected*" da classe Mãe ("base").

Para indicarmos que a classe é derivada de outra, usamos os dois pontos (:) a seguir ao nome da classe derivada.

```
class MembroVIP : Member
{
}
```

Polimorfismo

Polimorfismo é o conceito de dar várias formas aos métodos das classes derivadas.

Tem como princípio, em que as classes derivadas são capazes de invocar os mesmos métodos da classe base, mas com uma implementação diferente.

Para indicarmos que o nosso método da classe base é passível de ser alterado nas classes derivadas, usamos a keyword "*virtual*".

```
public class Veiculo{
    public string Tipo {get;set;}

    public Veiculo(string tipoVeiculo)
    {
        this.Tipo = tipoVeiculo;
    }

    public virtual void Mover(){
    }
    public virtual void Parar(){
    }
}
```

Na classe derivada usamos a keyword "*override*" no método com a implementação diferente da classe base.

```
public class Carro : Veiculo {

  public override void Mover(){
    Console.WriteLine("Carro anda!");
  }

  public override void Parar(){
    Console.WriteLine("Carro pára!");
  }
}
```


Co-Financiado por:

Além disso, temos a possibilidade de definir uma lista das classes mãe, instanciando objetos das classes derivadas.

```
Membro[] membrosGinasio = new Membro[5];
membrosGinasio[0] = new MembroNormal("Special Rate", "Jose", 30);
membrosGinasio[1] = new MembroVIP("Carolina", 20);
membrosGinasio[2] = new MembroVIP("Maria", 21);
```

Para identificar o tipo da classe, usa-se o método GetType(). A função typeOf() retorna o tipo da classe que pretendemos.

```
if (membrosGinasio[0].GetType() == typeof(MembroVIP))
```

Métodos e classes "Abstratas"

As classes abstratas são utilizadas apenas para servirem de base às classes derivadas. Estas não podem ser instanciadas. Estas podem conter campos, propriedades e métodos como as outras classes, excepto que não podem ter membros estáticos.

```
abstract class AMinhaClasseAbstrata
{

}
```

Os métodos abstratos deverão ser criados obrigatoriamente na declaração das classes derivadas. Os métodos podem conter lógica e pode ser declarado um construtor na classe abstrata.

```
public abstract void MeuMetodoAbstrato();
```

Os métodos abstratos ao contrário dos outros, devem conter o ponto e virgula no final da sua declaração.

Estes podem ser utilizados para o desenho de componentes, com funcionalidades pré-estabelecidas ou até utilizados para a criação de funcionalidades comuns ao longo das classes derivadas.

Interfaces

Semelhante às classes abstratas, estas não são instanciadas e só podem ser "herdadas". No entanto, esta apenas contém as declarações e não possuem lógica. Todos os métodos existentes nas interfaces devem se obrigatoriamente especificados nas classes derivadas.

Co-Financiado por:

Além disso, as classes podem conter várias "*interfaces*" mas apenas podem herdar uma superclasse.

É aconselhável e é boa prática identificar as classes interface iniciando com a Letra I.

```
public interface ILogicaNegocio
{
   void Inicializacao();
}
```

Para implementar a classe interface, utiliza-se os dois pontos.

```
public class LogicaNegocio : ILogicaNegocio
{
   public void Inicializacao()
   {
       //Some code
   }
}
```


Co-Financiado por:

Enum e Struct

Os tipos de dados “Enum” permitem ao programador utilizar nomes friendly para constantes numerárias. Para declarar o enum usa-se a keyword “enum” antes do nome da variável, sendo os valores passados dentro dos parênteses separados por virgula

```
enum WeekDayNumber
{
  Sun = 1, Mon = 2, Tues = 3, Wed = 4, Thurs = 5, Fri = 6, Sat = 7
}
```

Os “structs” representam estruturas de dados, podem conter campos, métodos, propriedades como as classes.

Estes são utilizados para definir estruturas de dados complexas, pares de chave-valor, entre outros.

```
public struct Money
{
    public string Currency { get; set; }
    public double Amount { get; set; }
}

public struct PhoneNumber
{
    public int Extension { get; set; }
    public int CountryCode { get; set; }
    public int RegionCode { get; set; }
}


Money money = new Money();
Console.WriteLine(money.Currency);
Console.WriteLine(money.Amount);
```


Co-Financiado por:

Classes Dictionary<TKey,TValue>

Os dicionários permitem criar listas/coleções genéricas de pares valor-chave. As suas chaves devem ser únicas e os seus valores são acessíveis pela chave myDictionary[key]

Podem ser utilizados os métodos Add(), Remove() para manipular as coleções ao longo do programa.

```
// Create a new dictionary of strings, with string keys.
Dictionary<string, string> openWith =
    new Dictionary<string, string>();

// Add some elements to the dictionary. There are no
// duplicate keys, but some of the values are duplicates.
openWith.Add("txt", "notepad.exe");
openWith.Add("bmp", "paint.exe");
openWith.Add("dib", "paint.exe");
openWith.Add("rtf", "wordpad.exe");
```

Para verificar se a chave já existe no dicionário utiliza-se o método ContainsKey().

```
if (openWith.ContainsKey("txt"))

{

    Console.WriteLine(openWith["txt"]);

}
```


Co-Financiado por:

LINQ

Funcionalidade que permite efetuar *queries* aos dados no nosso programa, podendo efetuar filtragens em listas de dados.

```
List<int> numbers = new() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

IEnumerable<int> filteringQuery =

from num in numbers

where num < 3 || num > 7

select num;
```

A Query é constituída por 3 expressões:

- from num in numbers
  - Indica que estamos a efetuar a query na lista numbers. A variavel num vai representar o valor individual no array
- where num < 3 || num > 7
  - Testa o variável individual se corresponde à condição
- select num;
  - Seleciona todos os elementos que satisfaçam a condição.

O LINQ também pode ser utilizado através de métodos, como demonstra os exemplos abaixo:

```
var average = numbers.Average();

var concatenationQuery = numbers.Concat(numbers);

var largeNumbersQuery = numbers.Where(c => c > 8);
```


Co-Financiado por:

Referências Bibliográficas

Chan, J. (2015). *C#: Learn C# in One Day and Learn It Well.* : LCF Publishing. doi:B016Z18MLG

Documentação C#. (01 de 10 de 2022). Obtido de Microsoft Learn: https://learn.microsoft.com/pt-pt/dotnet/csharp/

Troelsen, A., & Japikse, P. (2022). *Pro C# 10 with .NET 6: Foundational Principles and Practices in Programming.* New York: Apress Media LLC. doi:978-1-4842-7869-7

W3Schools. (01 de 10 de 2022). Obtido de C# Tutorial: https://www.w3schools.com/cs/index.php

Co-Financiado por: