

Elys Sanntos

**C# Collections parte 1: Listas, arrays, listas ligadas, dicionários e conjuntos**

SUMÁRIO

[Introdução 4](#_Toc62147086)

[**Declarando e Populando Arrays** 5](#_Toc62147087)

[**Atribuindo Valores A Um Array** 12](#_Toc62147088)

[**Modificando um valor de um elemento de um array** 12](#_Toc62147089)

[**Operações com Arrays** 13](#_Toc62147090)

[**Copiando dados de um array** 17](#_Toc62147091)

[**Mão na Massa: Começando com Arrays** 18](#_Toc62147092)

[Declarando um array 18](#_Toc62147093)

[Imprimindo um array 19](#_Toc62147094)

[Acessando o primeiro item do array 20](#_Toc62147095)

[Acessando o último item do array 21](#_Toc62147096)

[Modificando item de um array 21](#_Toc62147097)

[Acessando o primeiro ou último item de um array que satisfaz uma condição 22](#_Toc62147098)

[Revertendo a ordem do array 22](#_Toc62147099)

[Redimensionando um array 23](#_Toc62147100)

[Ordenando um array 24](#_Toc62147101)

[Copiando um array 25](#_Toc62147102)

[Clonando um array 25](#_Toc62147103)

[Limpando arrays 26](#_Toc62147104)

[**Para saber mais** 26](#_Toc62147105)

[**O que aprendemos?** 26](#_Toc62147106)

[**Consolidando seu conhecimento** 27](#_Toc62147107)

[**Introdução às Listas** 27](#_Toc62147108)

[**Operações com Listas** 33](#_Toc62147109)

[**Listas de Objetos** 39](#_Toc62147110)

[**Ordenando uma lista** 47](#_Toc62147111)

[**Lista Somente Leitura** 48](#_Toc62147112)

[**Lista somente-leitura** 52](#_Toc62147113)

[**Ordenando e Totalizando** 53](#_Toc62147114)

[**Totalizando valores** 58](#_Toc62147115)

[**Mão na Massa: Lista, a Coleção Flexível** 58](#_Toc62147116)

[Declarando uma lista 59](#_Toc62147117)

[A sintaxe de uma lista de strings em C 59](#_Toc62147118)

[Declarando e inicializando uma lista 59](#_Toc62147119)

[Inicializando uma lista vazia 60](#_Toc62147120)

[A lista como um array dinâmico 60](#_Toc62147121)

[Adicionando elementos 61](#_Toc62147122)

[Imprimindo listas 61](#_Toc62147123)

[Índices 63](#_Toc62147124)

[Acessando elementos de uma lista 63](#_Toc62147125)

[Procurando elementos 64](#_Toc62147126)

[Encontrando a primeira ocorrência 64](#_Toc62147127)

[Encontrando a última ocorrência 65](#_Toc62147128)

[Revertendo lista 65](#_Toc62147129)

[Removendo um elemento 66](#_Toc62147130)

[Ordenando lista 67](#_Toc62147131)

[Copiando lista 67](#_Toc62147132)

[Clonando lista 68](#_Toc62147133)

[**O que aprendemos?** 68](#_Toc62147134)

[**xxxxxxxx** 69](#_Toc62147135)

[**C# parte 8: List, lambda, linq** 70](#_Toc62147136)

[**Classe List do dotNet** 70](#_Toc62147137)

[**xxxxxxxx** 76](#_Toc62147138)

[Principais comandos 77](#_Toc62147139)

[Definir pontos de interrupção no código-fonte 77](#_Toc62147140)

[Conflitos 78](#_Toc62147141)

[Conclusão 79](#_Toc62147142)



# Introdução

#### **C# Collections parte 1: Listas, arrays, listas ligadas, dicionários e conjuntos**

* Aprenda os fundamentos das coleções C# através dos arrays.
* Explore o poder e flexibilidade das listas.
* Aprenda a trabalhar com conjuntos.
* Comece a usar dicionários para ler e gravar valores facilmente a partir de chaves.
* Entenda como funciona uma lista ligada e como construí-la através dos seus nós.
* Aplique em seus projetos coleções especializadas com prioridade de saída, através de pilhas e filas.

#### Começando com Arrays

Faremos uma introdução acerca dos assuntos que serão abordados:

Começando pelo tipo de coleção mais básica e primitiva do .NET Framework, as **matrizes**, ou ***arrays***. Aprenderemos a modificar e acessá-los por meio do índice, e a copiar e cloná-los também.

Em seguida, iremos explorar o poder das **listas**, os **arrays dinâmicos**, com atenção especial por se tratar de coleções poderosas.

Veremos também como **ordenar uma lista tendo por base elementos de uma classe customizada**, preparando esta lista para que os elementos sejam comparáveis com outros elementos.

Criaremos objetos que expõem coleções do tipo "Somente leitura" para impedir que suas informações sejam manipuladas externamente a este objeto.

Aprenderemos a copiar e clonar listas, totalizando os valores contidos nos objetos da coleção. Veremos como trabalhar com conjuntos no .NET Framework (objetos baseados no conceito matemático de conjuntos).

Aprenderemos a lidar com ***hashing***, e como implementar os métodos necessários para a definição deste espalhamento.

Para ler ou gravar valores facilmente a partir de uma chave, aprenderemos sobre a **coleção dicionário**, a **lista ligada** (*linked list*), e como seus nós se conectam durante a criação desta lista.

Também veremos como trabalhar com a coleção do tipo **pilha**.

# **Declarando e Populando Arrays**

Começaremos pelo tipo mais básico de coleção existente no .NET Framework, o **Array**. Supondo que temos 3 aulas a serem colocadas em uma coleção, representadas por 3 strings diferentes, vamos declará-las, armazenando-as em uma coleção (array).

Faremos isto em uma sintaxe especial, colocando inicialmente o tipo do elemento a ser contido no array, que no caso será uma string. Há duas maneiras de se declarar um array, e uma delas é quando já sabemos quais elementos ele irá conter.

Neste caso, sabemos que haverá 3 elementos, as três aulas, então poderemos simplesmente colocar o valor das três variáveis:

namespace CSharpCollections

{

class Program

{

static void Main(string[] args)

{

string aulaIntro = "Introdução às Coleções";

string aulaModelando = "Modelando a Classe Aula";

string aulaSets = "Trabalhando com Conjuntos";

string[] aulas = new string[]

{

aulaIntro,

aulaModelando,

aulaSets

};

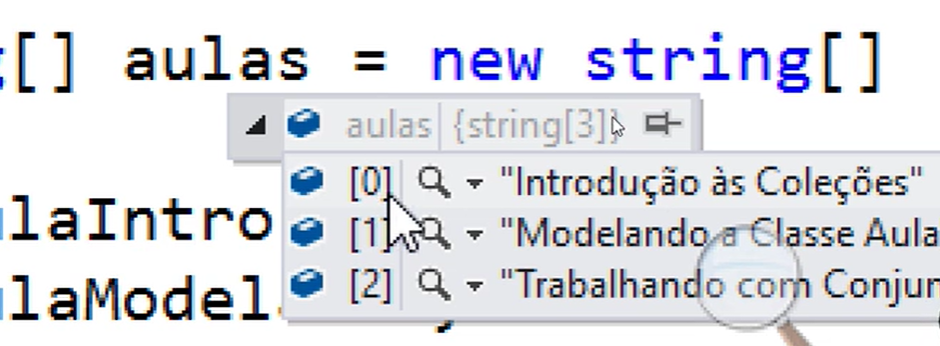
}

}

}

Vamos rodar a aplicação colocando um *breakpoint (*[*Veja aqui a documentação sobre BreakPoint*](https://docs.microsoft.com/pt-br/visualstudio/debugger/using-breakpoints?view=vs-2019%23:~:text=Voc%C3%AA%20tamb%C3%A9m%20pode%20selecionar%20a,interrup%C3%A7%C3%A3o%20%3E%20Inserir%20ponto%20de%20interrup%C3%A7%C3%A3o)*.)* na linha que fecha as strings. Veremos que em aulas há três elementos ("Introdução às Coleções", "Modelando a Classe Aula" e "Trabalhando com Conjuntos"), como gostaríamos.

O primeiro deles começa no índice 0, o segundo no 1 e o último no 2.

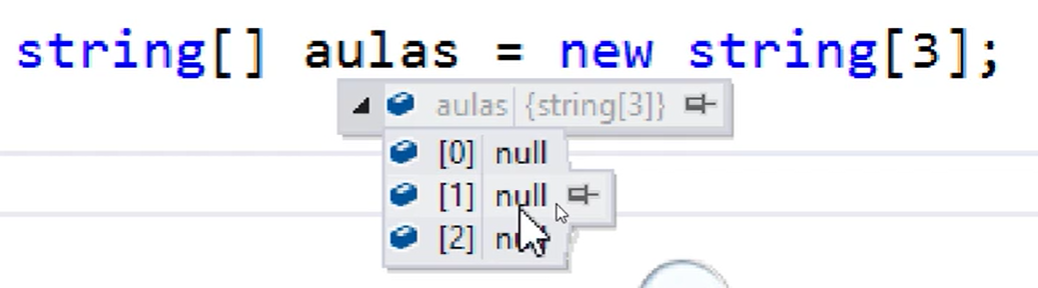
Ou seja, **o índice de um elemento será igual à sua posição menos um**. Esta é a primeira forma de declararmos um array no caso de já sabermos de antemão quais elementos ele irá conter.

A outra forma de declararmos um array consiste em incluirmos o tamanho que ele irá ocupar, algo que foi declarado implicitamente na primeira forma. Desta vez o faremos explicitamente.

É importante lembrarmos que um array é uma coleção de tamanho fixo!

string[] aulas = new string[3];

Rodando a app desta forma, veremos que em aulas estão armazenadas três posições, cada uma com valor null (nulo), e estes elementos ainda são desconhecidos, porém já temos o array declarado.

Completaremos o código:

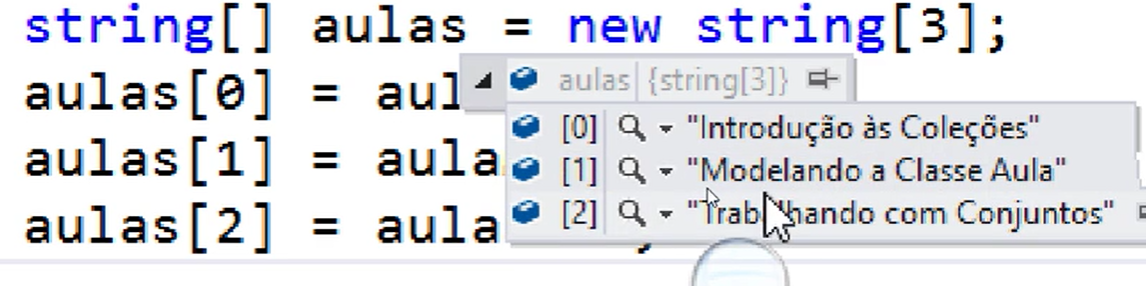
string[] aulas = new string[3];

aulas[0] = aulaIntro;

aulas[1] = aulaModelando;

aulas[2] = aulaSets;

Vamos rodar o código mais uma vez. Veremos que o array aulas possui os três elementos preenchidos. Imprimiremos isto na tela



acrescentando Console.WriteLine(); - que também pode ser feito com o atalho "cw + Tab + Tab":

string[] aulas = new string[3];

aulas[0] = aulaIntro;

aulas[1] = aulaModelando;

aulas[2] = aulaSets;

Console.WriteLine(aulas);

Ao rodarmos a aplicação para verificar o resultado, temos impresso na tela System.String[], o que não é exatamente o que queríamos. Queremos imprimir os elementos do array de aulas e, para isto, varreremos esta coleção (aulas) e imprimiremos cada um dos itens.

Uma das maneiras de se fazer isso é utilizar a instrução foreach para informar que, para cada aula em aulas, se faça algo (que no caso é a impressão).

string[] aulas = new string[3];

aulas[0] = aulaIntro;

aulas[1] = aulaModelando;

aulas[2] = aulaSets;

Console.WriteLine(aulas);

foreach (var aula in aulas)

{

Console.WriteLine(aula);

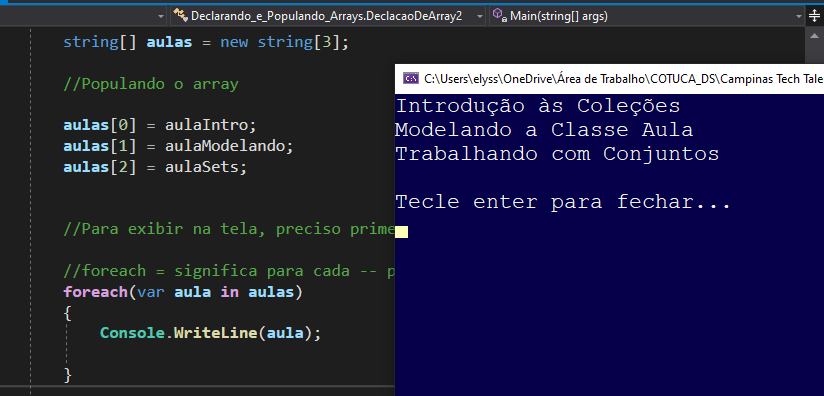
}

Rodando novamente o código, obteremos como *Output* o seguinte:

Introdução às Coleções

Modelando a Classe Aula

Trabalhando com Conjuntos



Pegaremos esta instrução e a jogaremos a outro método para a reutilização conforme nossa necessidade. Utilizando "Ctrl + . (ponto)" extrairemos o método denominado Imprimir(). Nele, teremos nosso laço foreach.

Como segunda forma de imprimirmos estes elementos, utilizaremos outro laço o for que varrerá o comprimento do nosso array de um índice até a posição final. Comentaremos o código do laço foreach

{

// strings omitidos

Console.WriteLine(aulas);

Imprimir(aulas);

}

private static void Imprimir(string[] aulas)

{

//foreach (var aula in aulas)

//{

// Console.WriteLine(aula);

//}

for (int i = 0; i < aulas.Length; i++)

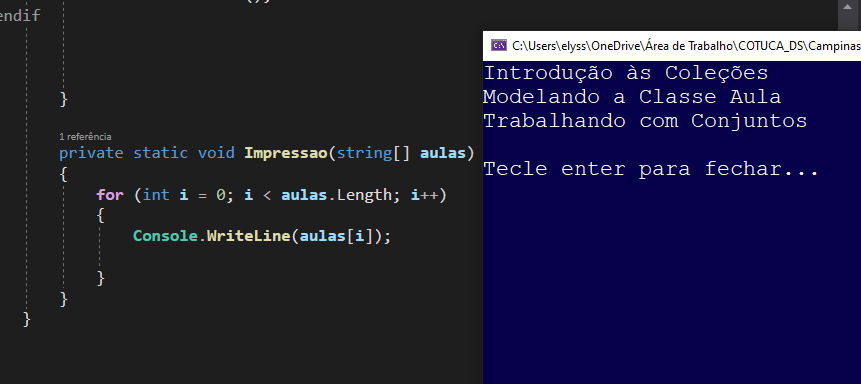
{

Console.WriteLine(aulas[i]);

}

}

Feito isto, podemos rodar a aplicação mais uma vez. O resultado é exatamente o mesmo que o anterior, embora a forma de chegarmos a ele seja outra.



Com o laço foreach não precisávamos de muitas informações, bastando passar o nome do array. Com o for, conseguimos controlar o avanço do índice. Neste caso, é necessário sabermos seu comprimento e o índice que está sendo impresso.

Para imprimirmos o primeiro elemento do nosso array, usaremos Console.WriteLine(); novamente, acessando este elemento, cujo índice é 0.

{

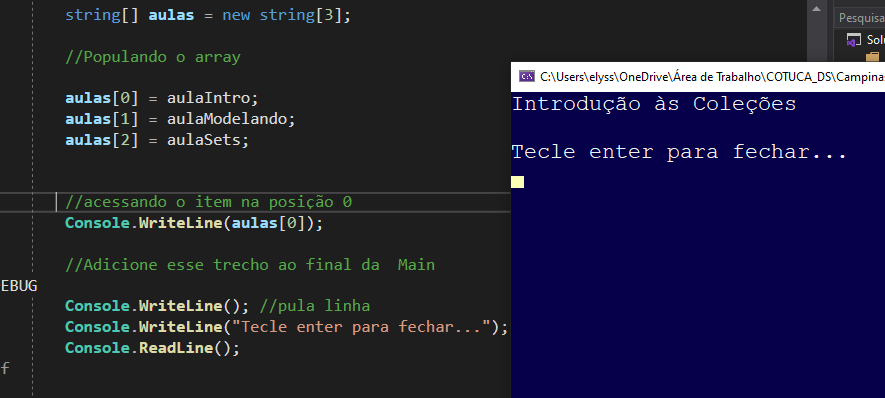
// strings omitidos

Console.WriteLine(aulas);

Imprimir(aulas);

Console.WriteLine(aulas[0]);

}



Rodando a app, veremos a impressão de "Introdução às Coleções" na tela. Para acessarmos o último elemento, precisaremos realizar uma conta:

{

// strings omitidos

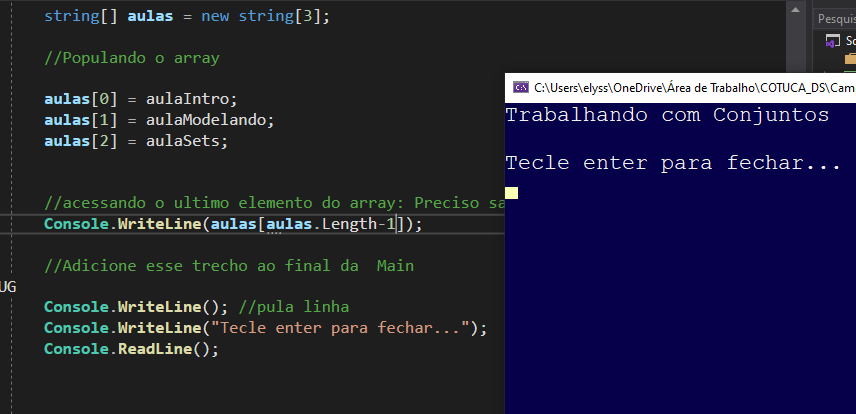
Console.WriteLine(aulas);

Imprimir(aulas);

Console.WriteLine(aulas[0]);

Console.WriteLine(aulas[aulas.Length - 1]);

}



Verificaremos que tudo funciona conforme esperado, quando rodamos a app vemos "Trabalhando com Conjuntos" no Output. Se quisermos trocar um elemento deste array, como faremos?

{

// strings omitidos

Console.WriteLine(aulas);

Imprimir(aulas);

Console.WriteLine(aulas[0]);

Console.WriteLine(aulas[aulas.Length - 1]);

aulaIntro = "Trabalhando com Arrays";

Imprimir(aulas);

}

Rodaremos a aplicação e não teremos nenhuma mudança! Isso ocorre pois aulaIntro não está sendo armazenado no nosso array de aulas. O que armazenamos ali foi o **valor copiado de** aulaIntro, portanto não adianta mudarmos seu valor.

Precisaremos acessar diretamente a primeira posição do nosso array de aulas. Para atribuirmos um valor a esta posição, de índice 0, deixaremos o código assim:

{

// strings omitidos

Console.WriteLine(aulas);

Imprimir(aulas);

Console.WriteLine(aulas[0]);

Console.WriteLine(aulas[aulas.Length - 1]);

aulaIntro = "Trabalhando com Arrays";

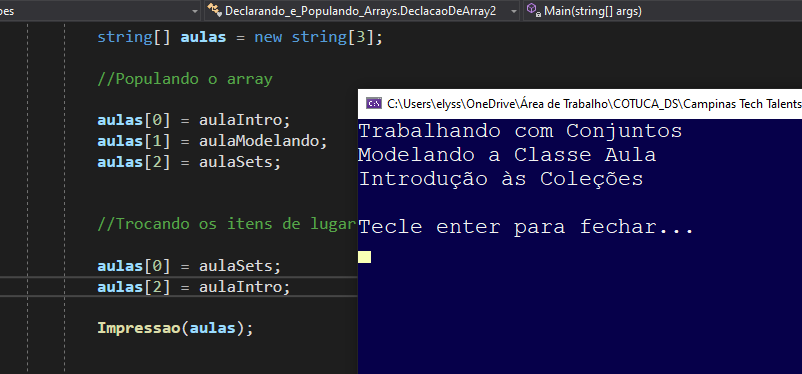
Imprimir(aulas);

aulas[0] = "Trabalhando com Arrays";

Imprimir(aulas);

}

Rodaremos a aplicação e veremos que conseguimos trocar com sucesso o primeiro elemento do nosso array.



# **Atribuindo Valores A Um Array**

Manoel precisa declarar uma coleção de nomes de meses do ano. Para criá-la, ele decide declarar um array com tamanho suficiente para armazenar os nomes de todos os meses do ano (na ordem correta). Em seguida, Manoel tem que definir no array o valor para o mês de janeiro.

O que Manoel pode fazer? Escolha a melhor alternativa.

string[] meses = new string[12];

meses[0] = "Janeiro";

Isso aí!

# **Modificando um valor de um elemento de um array**

Dê uma olhada no seguinte trecho de código:

//os elementos que entrarão no array

string aulaIntro = "Introdução às Coleções";

string aulaModelando = "Modelando a Classe Aula";

string aulaSets = "Trabalhando com Conjuntos";

//declarando um array populado

string[] aulas = new string[]

{

aulaIntro,

aulaModelando,

aulaSets

};

aulaModelando = "Trabalhando com Arrays";

Qual o valor de aulas[1] quando você executa o código acima?

"Modelando a Classe Aula"

Quando o array foi populado, o conteúdo aulaModelando foi copiado para para o array **por valor**, e naquele momento o conteúdo de aulaModelando era "Modelando a Classe Aula".

# **Operações com Arrays**

Agora vamos procurar um elemento no array, a aula "Modelando a Classe Aula", por exemplo. Para isso, imprimiremos o texto "Aula modelando está no índice " concatenado com o índice onde ele foi encontrado, mostrando onde teve a ocorrência da nossa string.

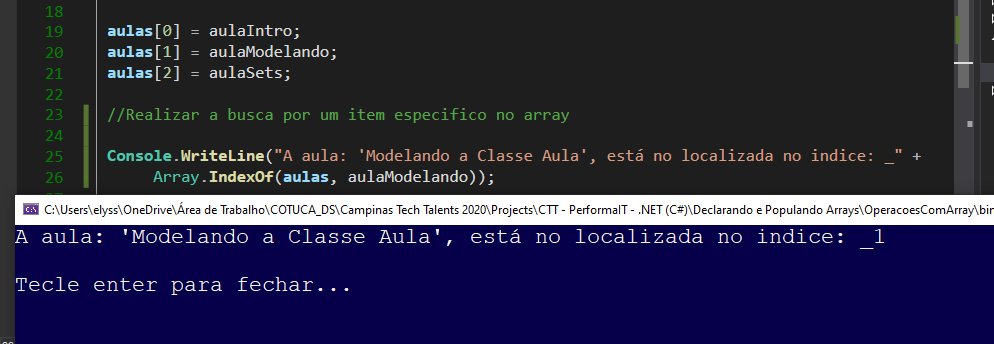
Utilizaremos um método estático IndexOf da classe Array que buscará a primeira ocorrência de uma string, retornando o índice onde foi encontrado:

aulas[0] = "Trabalhando com Arrays";

Imprimir(aulas);

Console.WriteLine("Aula modelando está no índice " + Array.IndexOf(aulas, aulaModelando));

Ao rodarmos a aplicação, será exibido o texto "A aula modelando está no índice 1". É a primeira ocorrência desta string no array de strings.



Digamos que este texto se repita algumas vezes no nosso array, então é possível procurá-la assim:

aulas[0] = "Trabalhando com Arrays";

Imprimir(aulas);

Console.WriteLine("Aula modelando está no índice " + Array.IndexOf(aulas, aulaModelando));

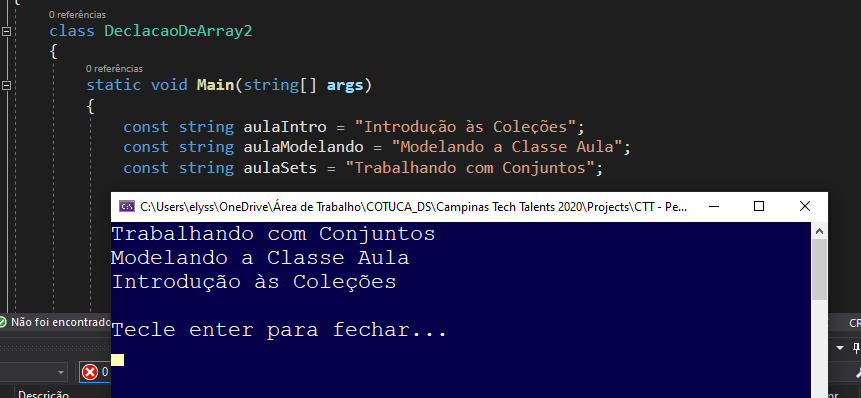
Console.WriteLine(Array.LastIndexOf(aulas, aulaModelando));

Rodando a app mais uma vez, a aula será encontrada no índice 1 por tratar-se da primeira e última ocorrência desta string no array. Prosseguindo, faremos uma operação para trocar a ordem dos elementos, colocando o último como primeiro, e vice-versa.

Para revertermos o array, chamaremos o método estático do Array chamado Reverse():

Console.WriteLine("Aula modelando está no índice " + Array.IndexOf(aulas, aulaModelando));

Console.WriteLine(Array.LastIndexOf(aulas, aulaModelando));



Array.Reverse(aulas);

Imprimir(aulas);

Em contrapartida, para voltar o Array para sua ordem original, faremos a reversão mais uma vez:

Console.WriteLine("Aula modelando está no índice " + Array.IndexOf(aulas, aulaModelando));

Console.WriteLine(Array.LastIndexOf(aulas, aulaModelando));

Array.Reverse(aulas);

Imprimir(aulas);

Array.Reverse(aulas);

Imprimir(aulas);

Vimos, então, que o método Reverse() funciona como um **método idempotente**, ou seja, se ele for rodado duas vezes, volta-se ao estado original. Supondo que tenhamos que redimensionar um array, reduzindo seu tamanho, a classe Array também permite que se faça isto, por meio do método estático denominado "Redimensionar", ou "***Resize***", não esquecendo de imprimir aulas de novo:

Array.Reverse(aulas);

Imprimir(aulas);

Array.Reverse(aulas);

Imprimir(aulas);

Array.Resize(ref aulas, 2);

Imprimir(aulas);

No entanto, como conseguimos redimensionar um array, se ele é um tipo de coleção de tamanho fixo? O que acontece é que, internamente, o Resize() cria uma nova coleção (array) com o tamanho definido (neste caso, 2) e, em seguida, copia os primeiros elementos da nossa classe original para este array de destino. No final, troca-se a referência da variável aulas apontando-a para o novo array.

Ou seja, sem que tivéssemos visto, uma cópia interna é criada. Este é o pequeno "truque" que acontece no método Resize(). Se quiséssemos aumentar o nosso array novamente, para o tamanho 3, poderemos tentar:

Array.Reverse(aulas);

Imprimir(aulas);

Array.Resize(ref aulas, 2);

Imprimir(aulas);

Array.Resize(ref aulas, 3);

Imprimir(aulas);

Rodaremos de novo a app e teremos exatamente o mesmo resultado. Para confirmar que esta alteração foi de fato realizada, colocaremos um *breakpoint* nesta última linha que acabamos de criar, rodaremos a aplicação e inspecionaremos o array aulas.

Ao fazermos isto, veremos 3 elementos, cujo último deles, de índice 2, está com valor nulo, o que ocorre porque este elemento (ou índice) está vago, ou seja, não há nenhum valor dentro dele.

É preciso, então, preenchermos esta última posição com um valor. Digamos que queiramos colocar a última aula como sendo a de conclusão. Para atribuirmos um valor a ela, colocaremos entre colchetes o índice que queremos usar.

O índice da última aula é 2, portanto o código ficará assim:

Array.Resize(ref aulas, 3);

Imprimir(aulas);

aulas[2] = "Conclusão";

Se tivermos um array cujo tamanho desconhecemos e quisermos preencher a última posição deste array, não poderemos simplesmente colocar 2, que no caso seria o número mágico, e sim uma expressão que sempre acesse a última posição do array.

Sendo o índice sempre n - 1, poderemos colocar o número do array, que é aulas.Length e - 1, garantindo que estamos sempre acessando a última posição do array. O código ficará assim, portanto:

aulas[aulas.Length - 1] = "Conclusão";

Imprimir(aulas);

Vamos rodar a aplicação e verificar seu funcionamento. Agora sim, conseguimos preencher a última posição com a string "Conclusão".

Para ordenarmos este array alfabeticamente, poderíamos utilizar o método estático da classe Array "Ordenar", que em inglês é "***Sort***", a partir do qual passaremos nosso array, imprimindo-o em seguida.

Array.Sort(aulas);

Imprimir(aulas);

Rodando o código, desta vez teremos as aulas em ordem alfabética, como gostaríamos:

Conclusão

Modelando a Classe Aula

Trabalhando com Arrays

O método Sort() não pode ser revertido, porque não é idempotente como o Reverse().

Para copiarmos um array a outro, criaremos um de destino, acrescentando seu novo tamanho, 2 (duas posições). Queremos duplicar as duas últimas aulas do nosso array de aulas para esta cópia.

Acessaremos o método estático da classe Array "cópia" (em inglês, "***Copy***"), passando o array de origem (aulas), o índice inicial a partir do qual copiaremos o array de aulas (1, que equivale à segunda posição), o nome do array de destino, a posição inicial (índice 0), e a quantidade de elementos que iremos copiar (2):

string[] copia = new string[2];

Array.Copy(aulas, 1, copia, 0, 2);

Imprimir(copia);

Rodaremos a aplicação e veremos que copiamos as duas últimas aulas. Para fazer o mesmo com todos os elementos, do início ao fim, poderíamos optar por uma maneira mais simples utilizando um método de **clonagem**.

Declararemos um novo array denominado clone. E, em vez de declararmos uma nova string, colocaremos a referência do nosso array inicial (aulas) chamando um método muito conveniente: Clone.

string[] clone = aulas.Clone();

Porém o Visual Studio está "reclamando" por não conseguir converter o tipo object para string[], então teremos que fazer um *cast*:

string[] clone = aulas.Clone() as string[];

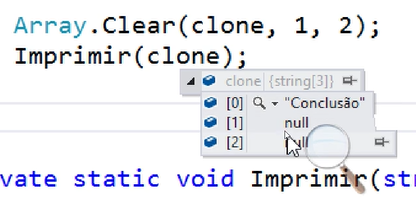
Imprimir(copia);

Rodaremos a aplicação e obteremos o resultado do nosso array clonado, que é exatamente o nosso original aulas. Para terminar, veremos como limpar alguns elementos do array clone, neste caso, os dois últimos.

Acessaremos o método estático da classe Array, de limpeza, do inglês ***clear***, bem como o índice da segunda posição (1) e o número de elementos que queremos limpar (2), não esquecendo de imprimir este array clone:

Array.Clear(clone, 1, 2);

Imprimir(clone);



Feito isto, rodaremos o código e veremos que conseguimos limpar os dois últimos elementos deste array. Se colocarmos um breakpoint na linha de código que acabamos de criar e rodarmos novamente a aplicação, confirmaremos isso.

Ao inspecionarmos o array, veremos que o primeiro elemento permanece intacto (índice 0, "Conclusão"), e os outros dois passaram a ter valor nulo.

É importante lembrarmos que o array é um tipo muito básico de coleção do .NET, então o utilizaremos em casos específicos, como em transferências de arquivos, manipulação de *buffer* ou imagens, em arquivos de baixo nível. Isso porque normalmente, embora faça parte de várias coleções do .NET, o array não é tão usado quanto outro tipo de coleção (como o List). Assim sendo, se for usar um tipo de coleção, busque sempre ver se é possível optar pela lista no lugar de um array, pois ela fornece métodos muito mais convenientes do que um simples array.

# **Copiando dados de um array**

Imagine um array declarado como decimal[] saldosPorDiaDoMes.

Esse array saldosPorDiaDoMes representa todos saldos diários de uma conta bancária ao longo de um determinado mês do ano.

Detalhe: você não sabe qual mês esse array representa. Pode ser qualquer mês!

Qual o código necessário para copiar os saldos dos dois últimos dias do mês para um novo array?

decimal[2];

Array.Copy(saldosPorDiaDoMes, saldosPorDiaDoMes.Length - 2, copia, 0, 2);

O método Array.Copy está copiando corretamente os valores dos 2 últimos elementos do array, a partir do penúltimo elemento do array, que é representado pela expressão saldosPorDiaDoMes.Length - 2, que sempre funcionará para qualquer mês do ano.

# **Mão na Massa: Começando com Arrays**

Vamos botar a mão na massa para trabalhar com o tipo de coleção mais básica do .NET Framework: as matrizes, ou arrays, como são conhecidas.

# Declarando um array

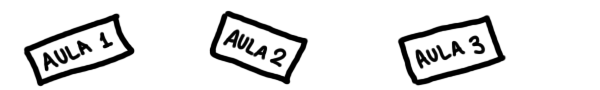
Imagine um grupo de 3 variáveis de strings, cada uma representando um nome de aula:

string aulaIntro = "Introdução às Coleções";

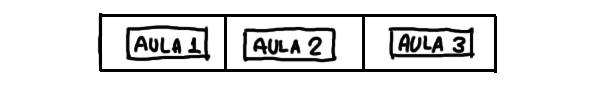
string aulaModelando = "Modelando a Classe Aula";

string aulaSets = "Trabalhando com Conjuntos";

Muitas vezes iremos trabalhar com essas informações reunidas num mesmo grupo, numa mesma coletânea de aulas, numa mesma... coleção.



Por quê? Ora, porque é mais fácil trabalhar com informações "no atacado", isto é, se elas são parte de uma estrutura maior. Essa estrutura maior é a coleção.



E uma coleção matriz, ou array, pode ser declarada assim:

string[] aulas = new string[]

{

aulaIntro,

aulaModelando,

aulaSets

};

Note a sintaxe especial: primeiro escrevemos string, e em seguida os colchetes. Isso indica ao programa que estamos querendo trabalhar com um array de strings.

Note também que escrevemos as chaves, e dentro delas os elementos que vão preencher o array.

Importante: Nunca declaramos um array de strings com Array<string>!

Mas essa não é a única forma. Também podemos declarar o array assim:

string[] aulas =

{

aulaIntro,

aulaModelando,

aulaSets

};

Mais simples, não?

Mas nos exemplos acima, precisamos todos os itens do array na declaração. E se precisarmos criar o array e só depois (ou muito depois) preencher esse array? É aí que entra a terceira forma de declaração: passando o tamanho do array explicitamente:

string[] aulas = new string[3];

E agora, quando formos preencher o array, temos que informar a posição a ser preenchida.

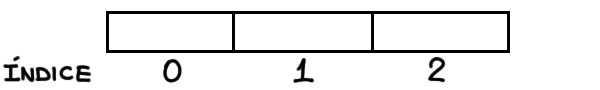
aulas[0] = aulaIntro;

aulas[1] = aulaModelando;

aulas[2] = aulaSets;

Como dá pra ver, essa sintaxe exige o índice dentro dos colchetes. O índice é igual a posição menos um:

**ÍNDICE = POSIÇÃO - 1**



# Imprimindo um array

Em situações em que você precise imprimir um array, isto é, imprimir todos os seus elementos no console, você se sentirá tentado a fazer isto:

Console.WriteLine(aulas);

Mas o resultado não será o esperado! Você verá aparecer no console:

System.String[]

Por quê? Quando tentamos imprimir o array diretamente, o .NET Framework irá acessar o método ToString() da classe object e imprimir o namespace e o nome da classe.

Então nesse caso vamos varrer, isso é, iterar, enumerar a coleção de alunos para imprimir cada um deles. Mas para isso vamos usar a instrução foreach, que pega cada um dos elementos:

foreach (var aula in aulas)

{

Console.WriteLine(aula);

}

Note que foreach é uma instrução bem simples, que não exige saber o tamanho do array nem precisa de acesso via indexador.

Existe outra maneira de fazer a mesma varredura do array, mas desta vez usando a instrução for:

for (int i = 0; i < aulas.Length; i++)

{

Console.WriteLine(aulas[i]);

}

Perceba que o tamanho do array aulas é dado pela propriedade Length.

Note que a instrução for exige saber o tamanho do array como também exige o acesso via indexador. Dá mais trabalho usar for do que foreach, porém ele permite mais controle sobre a varredura.

Para reaproveitar esse código, vamos colocar em um novo método:

private static void Imprimir(string[] aulas)

{

Console.Clear();

//foreach (var aula in aulas)

//{

// Console.WriteLine(aula);

//}

for (int i = 0; i < aulas.Length; i++)

{

Console.WriteLine(aulas[i]);

}

}

# Acessando o primeiro item do array

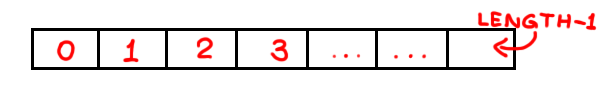
Como vimos, um array é uma coleção base-zero. Isto é, a primeira posição sempre tem índice zero. Então, para obtermos o primeiro item, temos que usar índice zero:

Console.WriteLine(aulas[0]);

# Acessando o último item do array

Se você quer pegar o último item do array, então tem que calcular o índice como a última posição menos um, isto é, o total de elementos menos um.

Console.WriteLine(aulas[aulas.Length - 1]);



# Modificando item de um array

Imagina que você precise modificar um item agora, a aula de introdução.

aulaIntro = "Trabalhando com Arrays";

Imprimir(aulas);

Qual o resultado? O item não é modificado! Mas por quê?

Simples: quando criamos o array, a variável aulaIntro foi copiada para o array **por valor**, e não por referência! Logo, qualquer mudança na variável aulaIntro não afetará o array.

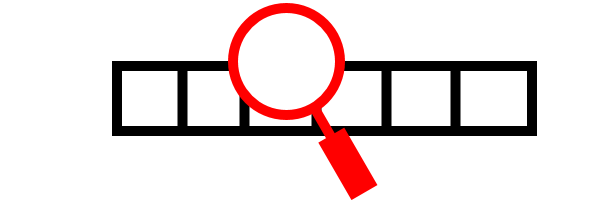
E como podemos mudar um valor dentro do array? Acessando o indexador, a mesma forma que fazemos quando estamos **obtendo** um valor.

aulas[0] = "Trabalhando com Arrays";

Imprimir(aulas);

Então tanto a leitura quanto a gravação de um item de um array é feito via indexador.

# Acessando o primeiro ou último item de um array que satisfaz uma condição



Quando você precisa varrer um array para buscar um item que satisfaz uma condição, você pode usar um laço, como **foreach** ou **for**, como vimos.

Porém, existe um jeito mais fácil. Você pode encontrar o índice de uma ocorrência, com o método IndexOf:

Console.WriteLine("Aula modelando está no índice " + Array.IndexOf(aulas, aulaModelando));

No exemplo acima, encontramos o índice onde está a aula "Modelando a Classe Aula". E para encontrar o elemento em si, basta usar o indexador com esse índice.

E se temos que pegar o último item que satisfaz a condição: Usamos o método LastIndexOf, que é análogo a IndexOf:

Console.WriteLine(Array.LastIndexOf(aulas, aulaModelando));

# Revertendo a ordem do array



Se quisermos revertar a ordem natural do array, podemos usar o método Reverse():

Array.Reverse(aulas);

Imprimir(aulas);

Note que o método Reverse acima não se aplica ao objeto em si, mas sim à classe Array. Ele é um método estático.

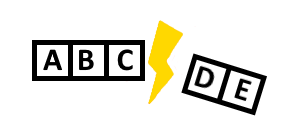
Se revertermos novamente, temos a ordem original do array:

Array.Reverse(aulas);

Imprimir(aulas);

# Redimensionando um array

Pode ser que você queira encolher, diminuir, truncar um array, para ele conter menos elementos. Mas um array é uma coleção de tamanho fixo. O que fazer?



Você pode usar o método estático de redimensionamento da classe Array.

Observe como reduzimos o tamanho de um array, de 3 para 2 posições:

Array.Resize(ref aulas, 2);

Imprimir(aulas);

Note que o método Resize toma como argumento:

* o array a ser redimensionado (por referência)
* o novo tamanho do array

Assim, teremos um novo array com tamanho 2.

Por outro lado, se quisermos aumentar o tamanho do array, podemos usar o mesmo método Resize, e dessa vez vamos voltar o array ao seu tamanho original, 3:

Array.Resize(ref aulas, 3);

Imprimir(aulas);

Mas note que, com isso, agora temos uma nova posição, ao final do array, sem nenhum valor preenchido.

Vamos preencher essa posição acessando nosso array com o indexador, e indicando a última posição, isto é, aulas.Length - 1:

aulas[aulas.Length - 1] = "Conclusão";

Imprimir(aulas);



Agora que aprendemos como redimensionar um array, fica a pergunta: como isso é possível, já que o array é uma coleção de tamanho fixo?

Na verdade, um array não é "redimensionado" diretamente pelo método Resize. O que acontece é que o método Resize **copia** o conteúdo do array para um novo array com o novo tamanho definido, e em seguida a variável aulas passa a referenciar esse novo array.

# Ordenando um array



Um array pode ser ordenado através do método estático Sort da classe Array:

Array.Sort(aulas);

Imprimir(aulas

Note que Sort não retorna valor algum. Em vez disso, ele modifica o próprio parâmetro, que é o array que foi ordenado.

Importante notar também que o critério de ordenação do método Sort nesse exemplo é a ordem natural dos elementos: como estamos trabalhando com strings, então a ordem é alfabética.

# Copiando um array



Existem situações em que queremos copiar todos os itens de um array, ou parte deles, para um novo array. Felizmente, existe o método Copy, que é estático e pertence à classe Array, para nos ajudar nisso.

Primeiro declaramos o array de destino, ou seja, aquele que vai receber a cópia.

string[] copia = new string[2];

Em seguida chamamos o método Copy, passando o array de origem, o de destino, assim como as posições (índices) de início da cópia, e também a quantidade de itens a serem copiados.

string[] copia = new string[2];

Array.Copy(aulas, 1, copia, 0, 2);

Imprimir(copia);

# Clonando um array

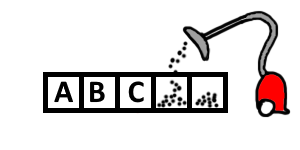


Se olharmos a sintaxe do método Copy acima, vemos que ela é um pouco complicada, principalmente se quisermos fazer uma cópia simples de um array para outro. Para esse tipo de cenário, o melhor mesmo é utilizar o método Clone, que obtém uma cópia exata do array de origem sem muita cerimônia:

string[] clone = aulas.Clone() as string[];

Imprimir(clone);

# Limpando arrays



Da mesma forma como preenchemos arrays, podemos também limpá-los, para deixar todas as posições (ou parte delas) do array vazias (isto é, com valor default, que pode ser zero, null, etc.).

Para isso existe o método estático Clear, que toma o array a ser afetado, além da posição (índice) inicial e a quantidade de elementos afetados:

Array.Clear(clone, 1, 2);

Imprimir(clone);

# **Para saber mais**

C# suporta vários tipos de arrays:

* array unidimensional
* array multidimensional (array retangular)
* array de arrays (arrays "denteados" ou "serrilhados")

Saiba mais sobre declaração e uso de arrays neste tutorial:

# **O que aprendemos?**

**O que aprendemos?**

* Declarando um array
* A sintaxe de um array em C#
  + Declarando e inicializando um array
  + Inicializando um array vazio com uma dimensão
* Imprimindo elementos de um array
* Índices
* Procurando índices de um elemento
  + Encontrando a primeira ocorrência
  + Encontrando a última ocorrência
* Revertendo arrays
* Redimensionando arrays
* Acessando elementos pelo índices
* Ordenando arrays
* Copiando arrays
* Clonando arrays
* Limpando elementos de um array

# **Consolidando seu conhecimento**

Chegou a hora de você seguir todos os passos realizados durante esta aula. Caso já tenha feito, excelente. Se ainda não, é importante que você execute o que foi visto no vídeo para poder continuar com o próximo capítulo.

#### Lista, a Coleção Flexível

# **Introdução às Listas**

Faremos algo muito parecido com o que fizemos na aula de arrays, porém utilizando outro tipo de coleção do .NET Framework, a **Lista**. Começaremos colando o código inicial dessa aula, com aquelas três strings usadas na criação da primeira coleção.

A forma de declarar uma lista é bem diferente de um array. Primeiro colocamos o nome da classe List, em seguida o tipo do elemento que ela irá conter, entre os símbolos de <string>. A instância segue o mesmo padrão new List<string>. Assim, teremos a primeira declaração de uma coleção Lista no nosso código, podemos iniciá-la com as string's que já deixamos preparada. Vamos imprimi-la e rodar a aplicação.

static void Main(string[] args)

{

string aulaIntro = "Introdução às Coleções";

string aulaModelando = "Modelando a Classe Aula";

string aulaSets = "Trabalhando com Conjuntos";

List<string> aulas = new List<string>

{

aulaIntro,

aulaModelando,

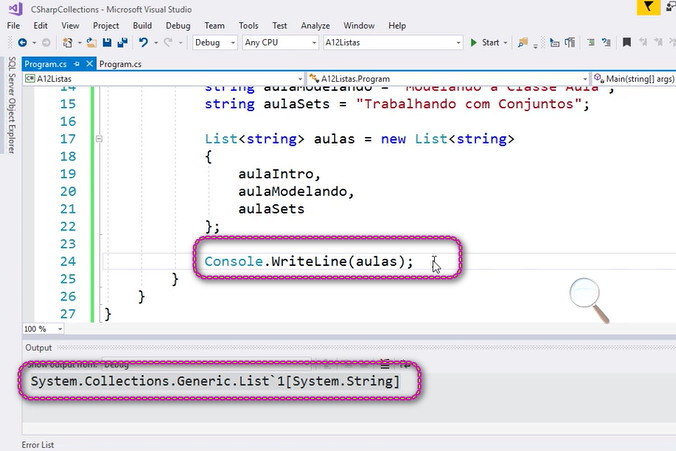
aulaSets

};

Console.WriteLine(aulas);

}

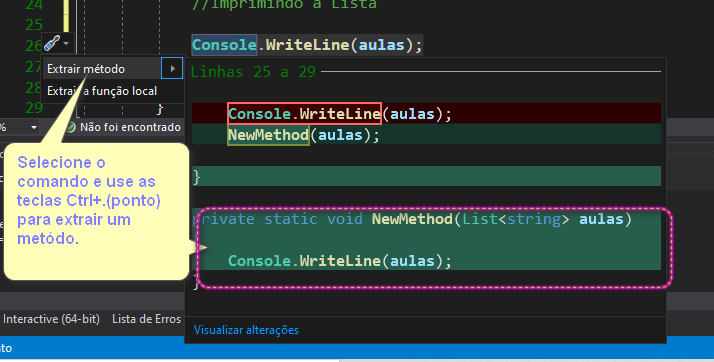
Na impressão, obtereremos o seguinte:



System.Collections.Generic.List`1[System.String]

Não temos a listagem de elementos pois, como vimos com os arrays, com Console.WriteLine(), o nome da classe será pego no método do string, e não seu conteúdo. Os elementos da lista de strings serão exibidos.

Precisaremos imprimir cada um dos elementos individualmente, extraindo o método Console.WriteLine(), optando pelo uso de um laço.



static void Main(string[] args)

{

string aulaIntro = "Introdução às Coleções";

string aulaModelando = "Modelando a Classe Aula";

string aulaSets = "Trabalhando com Conjuntos";

List<string> aulas = new List<string>

{

aulaIntro,

aulaModelando,

aulaSets

};

Imprimir(aulas);

}

private static void Imprimir(List<string> aulas)

{

foreach (var aula in aulas)

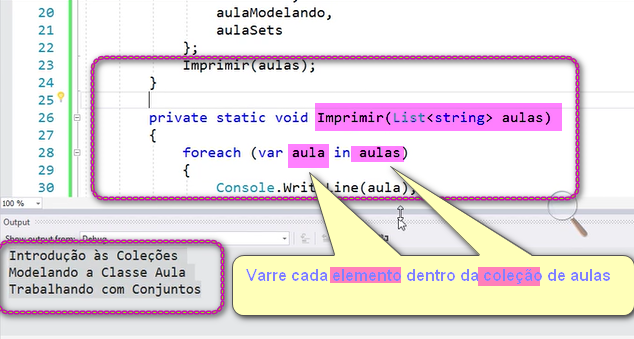
{

Console.WriteLine(aula);

}

}

Feito isso, rodaremos a app e obteremos a impressão das três aulas:



Introdução às Coleções

Modelando a Classe Aula

Trabalhando com Conjuntos

Poderemos obter a declaração de aulas de outra forma, pois neste caso já conhecemos os elementos iniciais da nossa lista. No entanto, pode ser que não conheçamos. Deixaremos o código assim:

static void Main(string[] args)

{

string aulaIntro = "Introdução às Coleções";

string aulaModelando = "Modelando a Classe Aula";

string aulaSets = "Trabalhando com Conjuntos";

//List<string> aulas = new List<string>

//{

// aulaIntro,

// aulaModelando,

// aulaSets

//};

List<string> aulas = new List<string>();

Imprimir(aulas);

}

private static void Imprimir(List<string> aulas)

{

foreach (var aula in aulas)

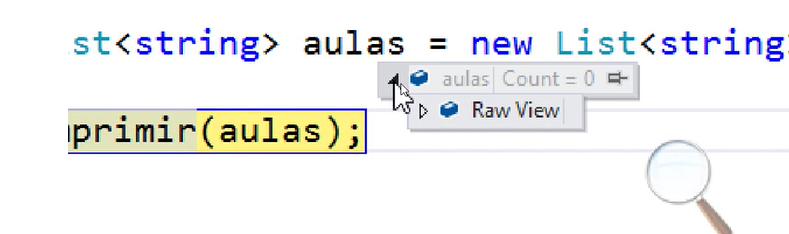
{

Console.WriteLine(aula);

}

}

Investigaremos o conteúdo da lista colocando um breakpoint no momento da impressão de aulas. Quando colocamos a lupa em cima dela, veremos que a lista de aulas possui uma quantidade 0 de elementos, ou seja, está vazia.



Vimos uma lista com 3 elementos e depois uma com nenhum elemento. **Uma lista contém internamente um array**, sendo assim chamada também de **array dinâmico** por "se acomodar" conforme a quantidade de elementos contidos.

Caso se trabalhe com um array com 10 elementos, é possível eliminar alguns, porém suas posições ficam vazias (nulas). No caso de uma lista, isto não acontece; conforme o acréscimo ou eliminação de elementos, a lista vai expandindo ou reduzindo.

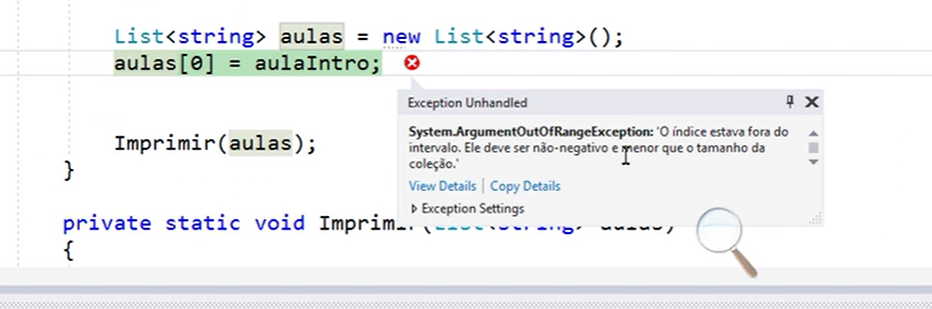
Já declaramos a lista, vazia, então adicionaremos cada um dos elementos necessários, o que significa que isso pode ser feito aos poucos, sem a declaração dos elementos iniciais:

List<string> aulas = new List<string>();

aulas[0] = aulaIntro;

Imprimir(aulas);

Ao rodarmos a aplicação, teremos um erro! Um System.ArgumentOutOfRangeException ("O índice estava fora do intervalo.



Ele deve ser não-negativo e menor que o tamanho da coleção"). O que acontece é que estamos acessando o índice, mas ele é inexistente na lista, pois está vazia. É necessário fazermos isto de outra forma.

Em se tratando de listas, não a atribuímos desta maneira para acrescentarmos elementos. Para isto, na variável de aulas, acessaremos o método de adicionar Add():

List<string> aulas = new List<string>();

aulas.Add(aulaIntro);

aulas.Add(aulaModelando);

aulas.Add(AulaSets);

Imprimir(aulas);

Rodaremos a app de novo para vermos o resultado... Muito bem, conseguimos enxergar os três elementos adicionados! Faremos uma modificação no método Imprimir(), por conta do laço foreach, feito com base em uma lista. Poderemos usar o for, assim como fizemos com o array.

Varreremos a lista a partir do índice 0 até a quantidade total de seus elementos e acessaremos a variável aulas e a propriedade length. No entanto, quando digitamos, esta palavra não é encontrada, pois não faz parte do objeto List<string>. Utilizaremos Count em seu lugar, que é seu equivalente no uso de listas:

private static void Imprimir(List<string> aulas)

{

//foreach (var aula in aulas)

//{

// Console.WriteLine(aula);

//}

for (int i = 0; i < aulas.Count; i++)

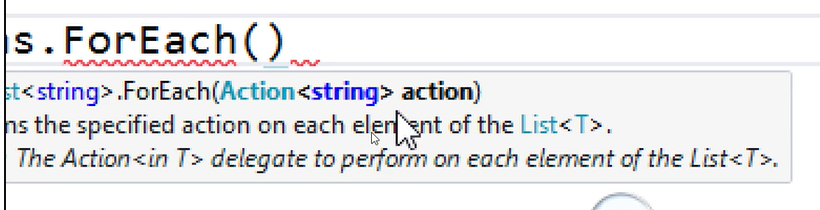
{

Console.WriteLine(aulas[i]);

}

}

Rodaremos e imprimiremos mais uma vez, e tudo ocorrerá sem problemas. Há ainda uma terceira forma de fazermos isso: em vez de utilizarmos uma instrução como foreach ou for, usaremos um método (ForEach()).



private static void Imprimir(List<string> aulas)

{

//foreach (var aula in aulas)

//{

// Console.WriteLine(aula);

//}

//for (int i = 0; i < aulas.Count; i++)

//{

// Console.WriteLine(aulas[i]);

//}

aulas.ForEach(aula =>

{

Console.WriteLine(aula);

});

}

Rodaremos o código para verificar se o resultado é o mesmo, o que se confirma. Neste momento, acessaremos o primeiro elemento das aulas, o que significa imprimir a primeira aula. Colocaremos um Console.WriteLine("A primeira aula é " + aula[0]);.

List<string> aulas = new List<string>();

aulas.Add(aulaIntro);

aulas.Add(aulaModelando);

aulas.Add(AulaSets);

Imprimir(aulas);

Console.WriteLine("A primeira aula é " + aulas[0]);

Rodando a aplicação, teremos o resultado:

A primeira aula é Introdução às Coleções

Caso queiramos a última aula, copiaremos a linha que acabamos de criar, trocaremos "primeira" por "última" e, no lugar do índice 0, colocaremos o último índice que, como vimos anteriormente, será aulas.Count - 1:

Console.WriteLine("A primeira aula é " + aulas[0]);

Console.WriteLine("A última aula é " + aulas[aulas.Count - 1]);

Rodando a app novamente, teremos:

A última aula é Trabalhando com Conjuntos

Outra maneira de acessarmos estes elementos - o último e o primeiro - se dá por meio de Last e First ("último" e "primeiro" em inglês), respectivamente:

Console.WriteLine("A primeira aula é " + aulas.First());

Console.WriteLine("A última aula é " + aulas.Last());

Rodaremos o código para confirmar esta alteração.

Neste vídeo, vimos alguns métodos interessantes da classe List, declarando-a de modos diferentes. E também aprendemos a visualizar impressões pelo método ForEach().

# **Operações com Listas**

Desta vez queremos trocar o nome da primeira aula acessando a lista de aulas e a primeira posição, de índice 0. Atribuiremos a ela um novo valor string.

Console.WriteLine("A última aula é " + aulas[aulas.Count - 1]);

Console.WriteLine("A última aula é " + aulas.Last());

aulas[0] = "Trabalhando com Listas";

Imprimir(aulas);

Rodaremos a aplicação e verificaremos que a alteração foi feita corretamente.

Para exemplificarmos uma busca por determinada palavra nesta lista, pesquisaremos pela primeira que contiver a palavra "trabalhando". Já vimos que First() retornará o primeiro elemento independentemente do conteúdo. No entanto, este método possui sobrecarga ("*overload*"), o que é bastante útil.



Existe a opção de passarmos um parâmetro para First(), um predicado possível de ser utilizado por meio de uma expressão lambda, a qual define, neste caso, uma função anônima. Utilizaremos também o método Contains(), que trará true ou false dependendo do resultado da busca.

Console.WriteLine("A primeira aula 'Trabalhando' é:" + aulas.First(aula => aula.Contains("Trabalhando")));

Vamos rodar a aplicação. Obteremos o seguinte:

A primeira aula 'Trabalhando' é: Trabalhando com Listas

Faremos algo parecido para buscarmos a última aula. Copiaremos as linhas de código recém criadas e substituiremos "primeira" por "última" e "*First*" por "*Last*":

Console.WriteLine("A primeira aula 'Trabalhando' é:" + aulas.First(aula => aula.Contains("Trabalhando")));

Console.WriteLine("A última aula 'Trabalhando' é:" + aulas.Last(aula => aula.Contains("Trabalhando")));

Feito isto, obteremos a mensagem:

A última aula 'Trabalhando' é: Trabalhando com Conjuntos

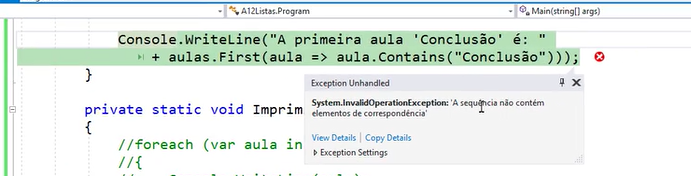
Para buscarmos a palavra "Conclusão", copiaremos e colaremos o código novamente, trocando "Trabalhando" por "Conclusão":

Console.WriteLine("A primeira aula 'Trabalhando' é:" + aulas.First(aula => aula.Contains("Trabalhando")));

Console.WriteLine("A última aula 'Trabalhando' é:" + aulas.Last(aula => aula.Contains("Trabalhando")));

Console.WriteLine("A primeira aula 'Conclusão' é:" + aulas.First(aula => aula.Contains("Conclusão")));

Obteremos um erro de InvalidOperationException: "A sequência não contém elementos de correspondência". **Muito cuidado quando for fazer uma busca deste tipo, cujo resultado não é encontrado, pois será necessário estar preparado para tratar este tipo de situação**.



Neste caso, em vez de First(), usaremos um método similar, FirstOrDefault(), o qual retorna o primeiro elemento de uma sequência ou valor default, se a sequência não contém elementos. Ou seja, quando nada é encontrado, o valor será substituído por um "*null*", ou "nulo".

Console.WriteLine("A primeira aula 'Conclusão' é:" + aulas.FirstOrDefault(aula => aula.Contains("Conclusão")));

Ao rodarmos este novo código, nenhum erro aparece, e sim o seguinte texto:

A primeira aula 'Conclusão' é:

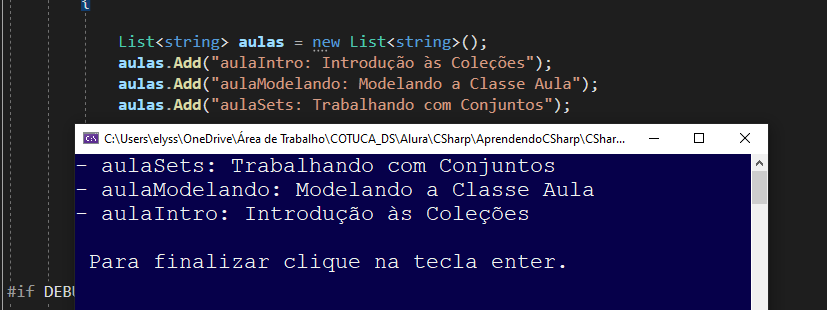
Se quisermos reverter esta lista para trocarmos sua ordem, assim como fizemos com os arrays, utilizaremos aulas.Reverse(); sem termos que acessar o método da classe Array como fazíamos anteriormente. A reversão é feita diretamente no objeto aulas.

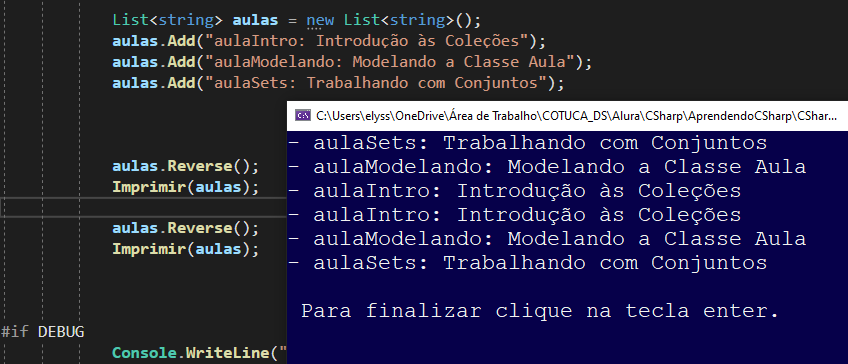
aulas.Reverse();

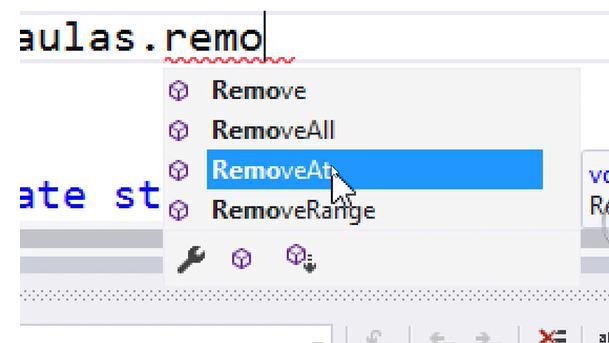
Imprimir(aulas);

Teremos o que queríamos e, para voltarmos ao estado inicial, basta copiarmos e colarmos estas mesmas linhas.

Assim como fizemos com os arrays, que tal removermos o último elemento da lista?



Acessaremos o objeto aulas e usaremos o método Remove(), que nos traz outras opções, como RemoveAt() (remoção de uma parte específica), para o qual também precisaremos passar o índice desejado.



Neste caso, precisaremos saber o tamanho da nossa lista por meio de aulas.Count, imprimindo-a em seguida:

aulas.RemoveAt(aulas.Count - 1);

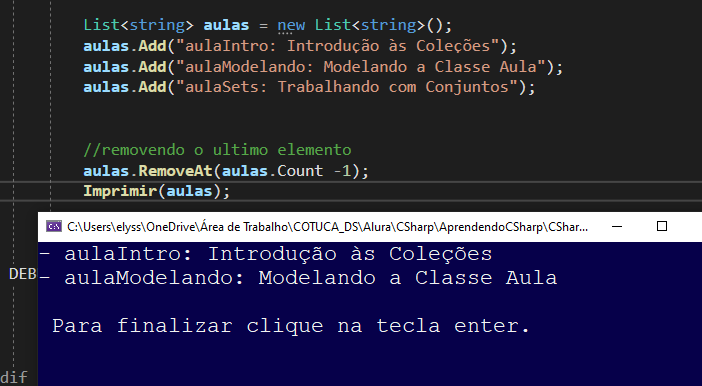
Imprimir(aulas);

Obteremos:

Trabalhando com Listas

Modelando a Classe Aula

O "Trabalhando com Conjuntos" sumiu, pois foi removido.

Temos dois elementos, diferentemente de um array, cuja remoção do último elemento se dá pelo Resize() e, ao redimensionarmos de volta, a lista fica com valor nulo.

Aqui, quando removemos elementos, a lista vai diminuindo e expandindo, conforme se vai adicionando ou removendo itens. Para acrescentarmos mais uma aula à nossa lista, poderemos utilizar o método Add():

aulas.Add("Conclusão");

Imprimir(aulas);

Para ordenar a lista de forma alfabética, precisaremos do método Sort(), como visto com arrays, porém sem o método estático na classe List, acessando-se diretamente pelo objeto aulas:

aulas.Sort();

Imprimir(aulas);

Para copiarmos os dois últimos elementos da lista de aulas, escreveremos o seguinte código:

List<string> copia = aulas.GetRange(aulas.Count - 2, 2)

Imprimir(copia);

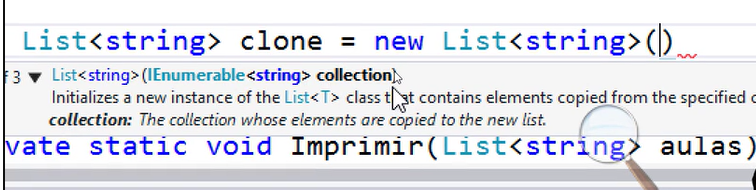
Rodando a aplicação, teremos a impressão da cópia, com as duas últimas aulas que tínhamos em nossa coleção original:

Modelando a Classe Aula

Trabalhando com Listas

Agora faremos o equivalente à clonagem do array. Aqui, não temos um método para isto. Declararemos a lista denominada clone, uma nova lista, não mais do tipo string. Simplesmente obteremos uma lista a partir da nossa lista original, acessando um construtor diferente, já que o padrão não possui parâmetros.

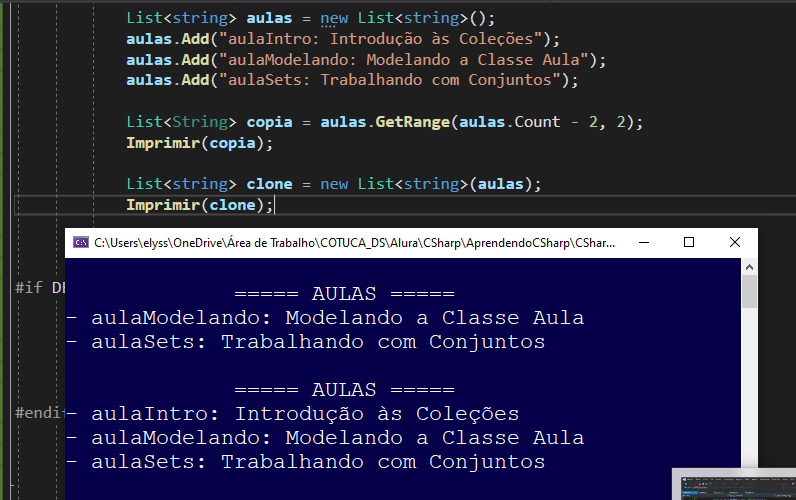
Há outra variação, com capacidade (o tamanho) da lista inicial, que também não nos interessa. A terceira variação do construtor recebe uma coleção possível de ser nossa lista original:



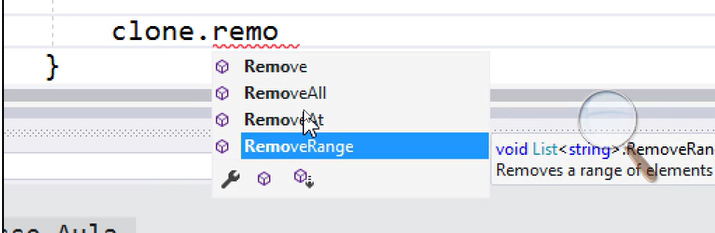
List<string> clone = new List<string>(aulas);

Imprimir(clone);

Com isto, instanciaremos nosso clone a partir dos elementos contidos em aulas e teremos sua cópia exata.



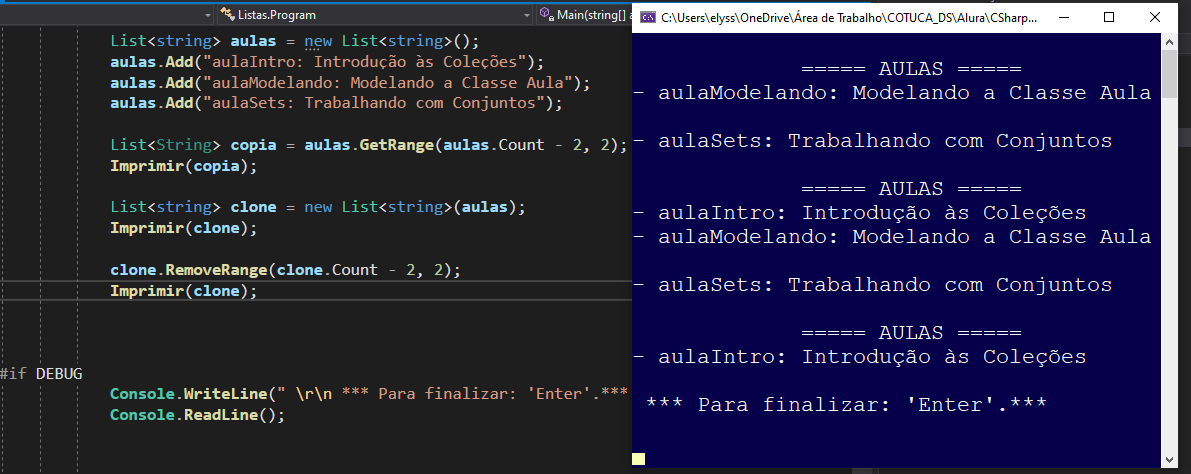
Para finalizar, removeremos os dois últimos elementos do nosso clone, mencionando-o e escolhendo a opção RemoveRange (remover uma faixa de elementos),

passando o índice da primeira posição que queremos remover como primeiro parâmetro e, como segundo, a quantidade que queremos remover, que é 2.

clone.RemoveRange(clone.Count - 2, 2);

Imprimir(clone);

Imprimiremos este clone novamente para vermos o resultado.

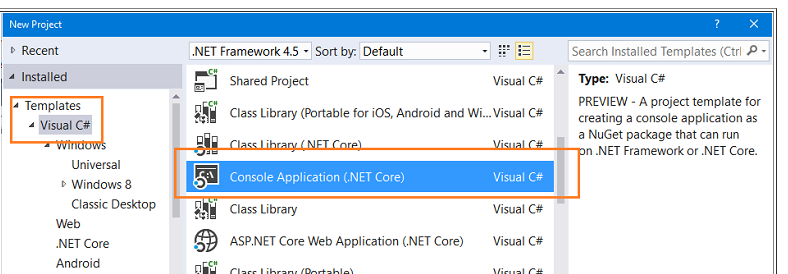


Neste vídeo percebemos como as listas são poderosas, desempenhando um avanço em relação aos arrays por ser dinâmica e não exigir que se conheça tão especificamente o tamanho da coleção.

Como critério para o desenvolvimento de uma aplicação, sempre que pensar em coleções, pense primeiro em listas, partindo para outro tipo de coleção apenas se ela não for o suficiente.

# **Listas de Objetos**

Neste vídeo estamos trabalhando com um novo projeto, um *Console Application*, e veremos as três strings do vídeo anterior, aquelas três aulas:



namespace A23ListasDeObjetos

{

class Program

{

static void Main(string[] args)

{

string aulaIntro = "Introdução às Coleções";

string aulaModelando = "Modelando a Classe Aula";

string aulaSets = "Trabalhando com Conjuntos";

}

}

}

Além destes nomes (as strings), precisaremos trabalhar com a duração destas aulas, uma lista que as armazene. Trata-se de uma lista de objetos do tipo Aula. Vamos declarar a classe:

class Aula

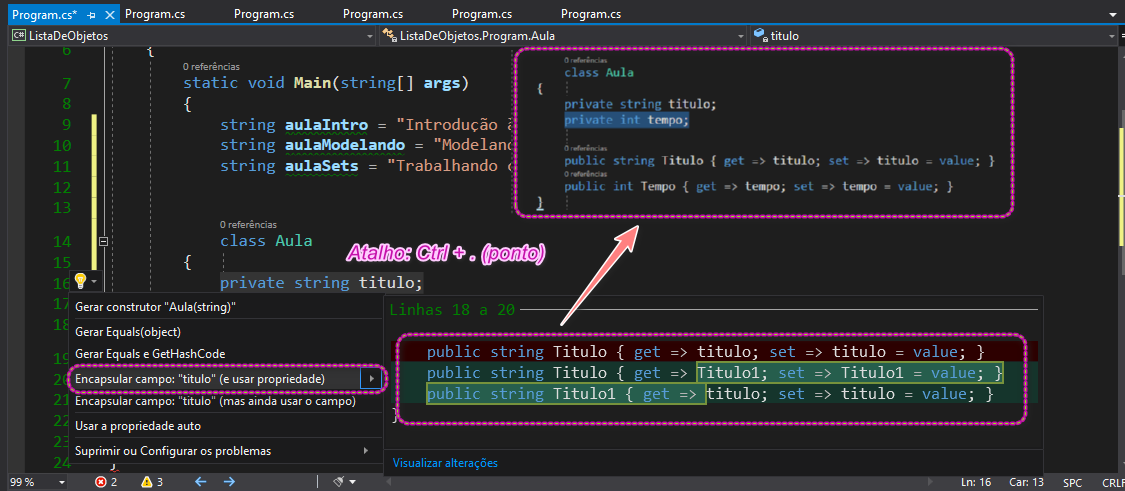
{

private string titulo;

private int tempo;

}

A partir destes dois campos, criaremos propriedades com o atalho "Ctrl + . [ponto]", selecionando "Encapsular campo: 'titulo'" para o Visual Studio gerar uma propriedade homônima. Faremos o mesmo para tempo.



Feito isto, selecionaremos os dois campos, titulo e tempo, clicando em "Ctrl + ." e selecionando "Gerar construtor 'Aula(string, int)'".

class Aula

{

private string titulo;

private int tempo;

public Aula(string titulo, int tempo)

{

this.titulo = titulo;

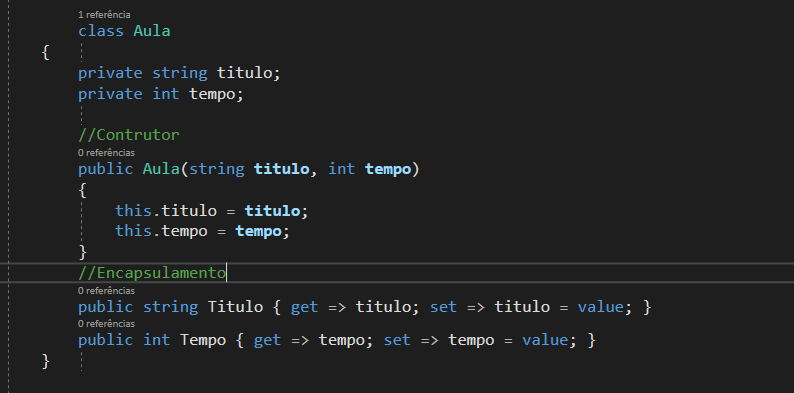
this.tempo = tempo;

}

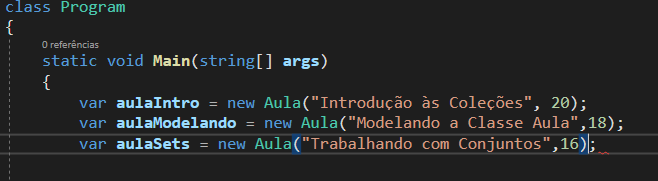
public string Titulo { get => titulo; set => titulo = value; }

public int Tempo { get => tempo; set => tempo = value; }

}



Trocaremos as strings por instâncias da classe Aula, substituindo todas elas por var, passando também o tempo de duração de cada aula em minutos.



Criaremos também a instância da nossa listagem utilizando List, uma classe genérica de tipo Aula. Declararemos a variável aulas, que armazenará a lista com todas as aulas declaradas anteriormente.

Supondo que queiramos acrescentar mais uma aula ("Conclusão") em nossa lista, desta vez uma string, digitaremos:

static void Main(string[] args)

{

var aulaIntro = new Aula("Introdução às Coleções", 20);

var aulaModelando = new Aula("Modelando a Classe Aula", 18);

var aulaSets = new Aula("Trabalhando com Conjuntos", 16);

List<Aula> aulas = new List<Aula>();

aulas.Add(aulaIntro);

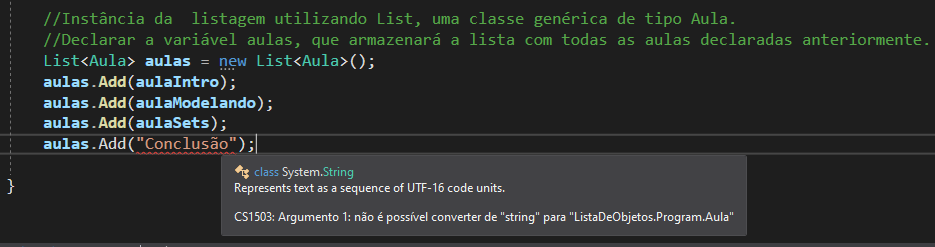
aulas.Add(aulaModelando);

aulas.Add(aulaSets);

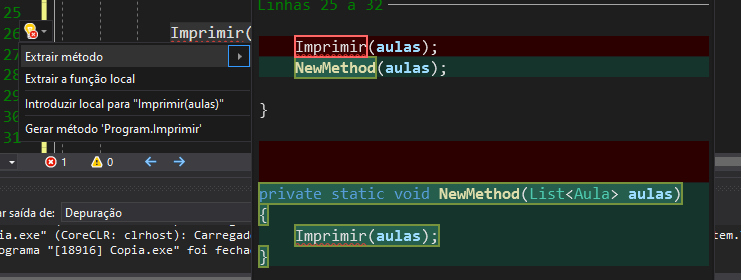
aulas.Add("Conclusão");

}

Porém, ao fazermos isto,

 o Visual Studio nos informa que não é possível converter uma string para uma instância da classe Aula. Quando se define uma lista como sendo de um determinado tipo, só é possível adicionar elementos cujo tipo seja daquela instância, a menos que seja uma herança - ou derivada - daquele tipo utilizado como elemento.

Portanto, por ora, comentaremos esta linha. Continuando, para imprimirmos chamaremos um método, passando como argumento a nossa lista de aulas. Como Imprimir() ainda não existe, solicitaremos ao Visual Studio a criação deste método.



static void Main(string[] args)

{

var aulaIntro = new Aula("Introdução às Coleções", 20);

var aulaModelando = new Aula("Modelando a Classe Aula", 18);

var aulaSets = new Aula("Trabalhando com Conjuntos", 16);

List<Aula> aulas = new List<Aula>();

aulas.Add(aulaIntro);

aulas.Add(aulaModelando);

aulas.Add(aulaSets);

//aulas.Add("Conclusão");

Imprimir(aulas);

}

private static void Imprimir(List<Aula> aulas)

{

Console.Clear();

foreach (var aula in aulas)

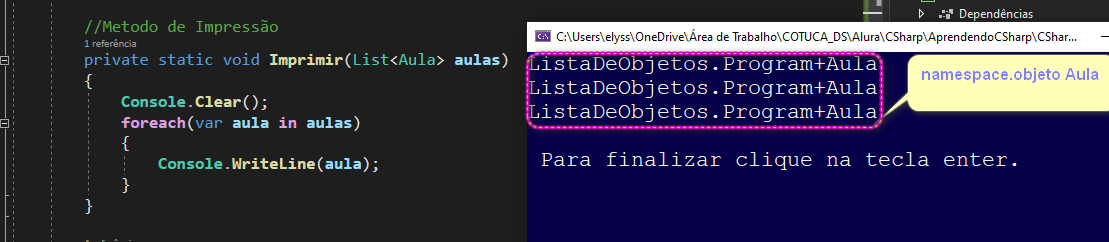
{

Console.WriteLine(aula);

}

}

Com "Ctrl + F5", rodaremos a aplicação. Obteremos não os nomes das aulas, e sim três linhas contendo o namespace do objeto Aula. Não é o que gostaríamos!

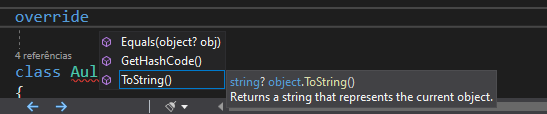
 Isso ocorre porque quando pedimos a impressão, o Console.WriteLine() vai até o objeto Aula, em busca de um método chamado ToString(), inexistente na classe Aula.

No entanto, como a classe Aula naturalmente herda da classe Object, o método ToString() é buscado na classe Object, que é de onde vem a impressão final, o resultado.

Precisaremos sobrepor este método da classe Object para que se imprima aquilo que queremos.

Para realizar este *override*, implementaremos um método ToString() na classe Aula.

Digitando override e dando espaço, obtém-se a opção ToString. Clicando nele, o programa já nos cria este método.



Para concatenarmos as informações provenientes da classe Aula, utilizaremos a sintaxe **interpolação de strings**, que faz parte do C#, marcada pelo uso de $ e chaves ({}).

public override string ToString()

{

return $"[título: {titulo}, tempo: {tempo} minutos]";

}

Rodaremos a aplicação novamente, e obteremos o seguinte:



[título: Introdução às Coleções, tempo: 20 minutos]

[título: Modelando a Classe Aula, tempo: 18 minutos]

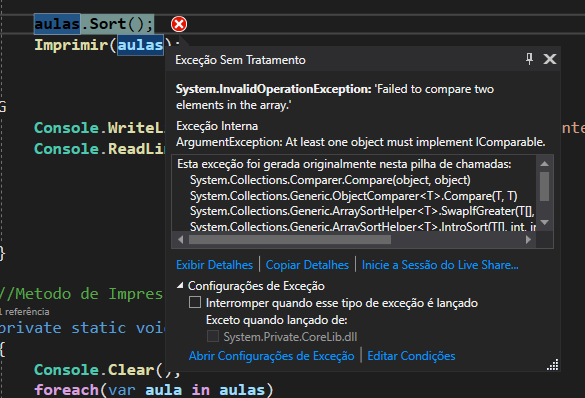
[título: Trabalhando com Conjuntos, tempo: 16 minutos]

Prosseguindo, faremos outra operação com a listagem de aulas, a ordenação alfabética. Para isso, chamaremos o método Sort() pois, como vimos anteriormente, serve bem para este propósito.

aulas.Sort();

Imprimir(aulas);

Rodando a aplicação, teremos um erro, de **falha ao comparar dois elementos na matriz, pois pelo menos um deles deve implementar IComparable**.

Isto significa que a classe Aula não está implementando a interface necessária para realizar esta operação, que exige uma interface chamada IComparable.

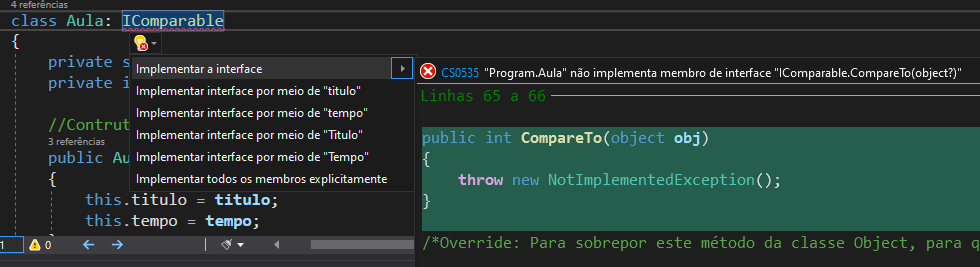
Clicando em cima disto e apertando "Ctrl + .", a interface é de fato implementada.

class Aula : IComparable

{

//...

}



O método CompareTo() foi gerado para comparar dois elementos que fazem parte da classe Aula, uma vez que o método Sort() precisa chamar CompareTo() para saber que algoritmo precisará ser usado para a ordenação.

Acrescentaremos um return para que o título seja comparado com o nome do outro objeto, o parâmetro object obj.

O CompareTo() recebe como parâmetro este objeto da classe Aula, e poderíamos passar obj chamando obj.titulo.

No entanto, isto não existe, pois object é um objeto, sendo necessário, então, fazermos um *cast*, uma conversão para que isso possa ser trabalhado.

O código ficará assim:

public int CompareTo(object obj)

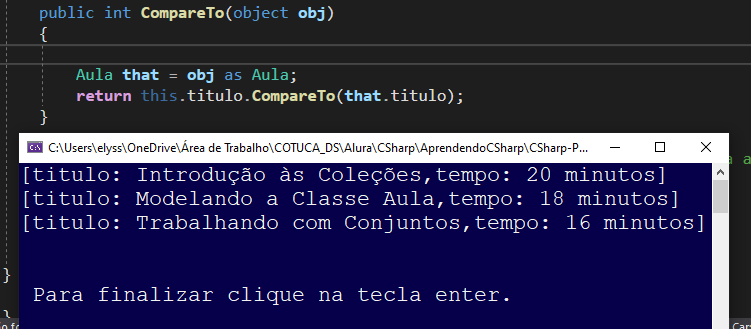
{

Aula that = obj as Aula;

return this.titulo.CompareTo(that.titulo);

}

Vamos rodar a aplicação de novo para verificar se tudo funciona corretamente. Muito bem, as aulas aparecem listadas em ordem alfabética.



Agora, de que forma poderemos, sem que a classe Aulas seja alterada, mudar a forma de ordenação?

Ordenando-se não pelo título, mas pelo tempo de duração, por exemplo? Ao tentarmos utilizar o método Sort() verificaremos que há um *overload*, que pode receber parâmetros.

É possível, assim, compararmos dois elementos ou instâncias de uma mesma classe. No método Sort(), usaremos uma expressão lambda, a receber dois parâmetros, elementos iniciais e finais, com nomes que você quiser, como este e outro.

Iremos usar CompareTo() mais uma vez, para compararmos este com outro:

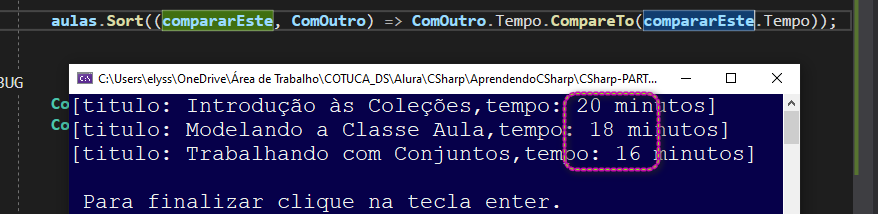
aulas.Sort();

Imprimir(aulas);

aulas.Sort((este, outro) => este.Tempo.CompareTo(outro.Tempo));

Imprimir(aulas);

Rodaremos o código e veremos que a listagem de aulas foi ordenada do menor para o maior tempo de duração com sucesso!



# **Ordenando uma lista**

Um colega de Pedro desenvolveu um código C# para ordenar uma lista, através do método Sort, conforme abaixo:

public class Program

{

public static void Main()

{

var aulaIntro = new Aula("Introdução às Coleções", 20);

var aulaModelando = new Aula("Modelando a Classe Aula", 18);

var aulaSets = new Aula("Trabalhando com Conjuntos", 16);

List<Aula> aulas = new List<Aula>();

aulas.Add(aulaIntro);

aulas.Add(aulaModelando);

aulas.Add(aulaSets);

aulas.Sort();

}

static void Imprimir(List<Aula> aulas)

{

Console.WriteLine();

foreach(var aula in aulas)

{

Console.WriteLine(aula);

}

}

}

public class Aula

{

private string titulo;

private int tempo;

public Aula(string titulo, int tempo)

{

this.titulo = titulo;

this.tempo = tempo;

}

public string Titulo { get { return titulo; } set { titulo = value; } }

public int Tempo { get { return tempo; } set { tempo = value; } }

public override string ToString()

{

return "[título: " + titulo + ", tempo: " + tempo + " minutos]";

}

}

Ao executar o código, Pedro se depara com um erro!

Sem saber o que fazer, Pedro pede a sua ajuda. Então ele mostra que o programa está lançando uma exceção na linha:

aulas.Sort();

Será que você pode dar uma força ao Pedro? Use o que aprendeu na aula para apontar a causa mais provável do problema!

A classe Aula não implementa a interface IComparable e, por consequência, não implementa o método CompareTo.

A interface IComparable exige uma implementação do método CompareTo, que precisa ser chamado pelo algoritmo interno do método Sort() da classe List<T>.

# **Lista Somente Leitura**

Criar um novo projeto *Console Application*, mantendo a classe Aula, contida em um arquivo separado (Aula.cs). Em Program.cs, mantivemos nosso método Imprimir():

private static void Imprimir(List<Aula> aulas)

{

Console.Clear();

foreach (var aula in aulas)

{

Console.WriteLine(aula);

}

}

Aumentaremos um pouco mais a arquitetura para criarmos uma nova classe com os cursos cujas aulas serão incluídas.

Esta classe se chamará Curso.cs e terá três propriedades: a lista de aulas (como visto anteriormente), o nome do curso e instrutor.

Utilizaremos o atalho "propfull + TAB + TAB" para criar a estrutura da propriedade necessária, com get e set. Declararemos seu tipo, e a propriedade será pública, denominada Aulas. Faremos o mesmo para o nome do curso e instrutor.

namespace A24ListaSomenteLeitura

{

class Curso

{

private List<Aula> aulas;

public List<Aula> Aulas

{

get { return aulas; }

set { aulas = value; }

}

private string nome;

public string Nome

{

get { return nome; }

set { nome = value; }

}

private string instrutor;

public string Instrutor

{

get { return instrutor; }

set { instrutor = value; }

}

}

}

Criaremos também o construtor da classe recém criada. Para fazê-lo de maneira mais prática, usando as teclas "Alt + seta para cima", deixaremos a linha private string instrutor; logo abaixo de private string nome;.

Então selecionaremos estas duas linhas para que o construtor seja gerado. Com "Ctrl + ." clicaremos em Generate constructor 'Curso(string, string)', no qual instanciaremos a lista de aulas, pois inicialmente ela estará nula.

Inicialmente, definiremos esta lista como nova, a qual, a partir de agora, poderemos acessar sem problema algum em relação à referência nula.

private string nome;

private string instrutor;

public Curso(string nome, string instrutor)

{

this.nome = nome;

this.instrutor = instrutor;

this.aulas = new List<Aula>();

}

Voltando a Program.cs, declararemos uma variável Curso de nome csharpColecoes, pelo qual acessaremos a propriedade pública Aulas. Adicionaremos uma nova aula a esta lista com o método Add(), passando como parâmetro uma nova instância da classe Aula.

static void Main(string[] args)

{

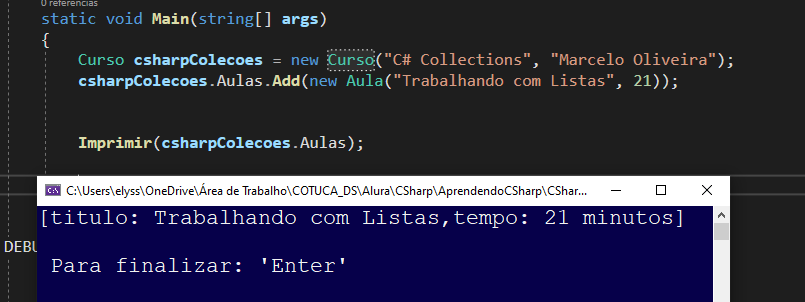
Curso csharpColecoes = new Curso("C# Collections", "Marcelo Oliveira");

csharpColecoes.Aulas.Add(new Aula("Trabalhando com Listas", 21));

Imprimir(csharpColecoes.Aulas);

}

Imprimiremos o resultado no console e rodaremos a aplicação com "Ctrl + F5". Obteremos o seguinte:



[título: Trabalhando com Listas, tempo: 21 minutos]

Conseguimos criar a classe Curso com sucesso, populando e imprimindo as aulas. No entanto, há um problema não muito evidente, conhecido por ***code smell* na programação**, ou "mau cheiro no código", que neste caso implica no fato de acessarmos csharpColecoes.Aulas e a lista de aulas da classe Curso adicionando elementos nela.

Isso significa que estamos "passando por cima" da responsabilidade da classe Curso de manter suas propriedades, e estamos manipulando diretamente esta lista de aulas.

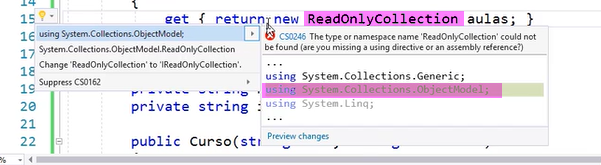
O correto seria chamarmos um método da classe Curso para podermos adicionar estes elementos. Caso contrário, esta classe não tem nenhum controle sobre as suas aulas, pois esta listagem está exposta.

Trata-se de um *code smell* por ser uma "exposição indecente". Para evitarmos que uma propriedade fique exposta à manipulações externas, é necessário encapsularmos esta funcionalidade de adicionar elementos na lista.

Então, primeiramente, teremos que bloquear esta lista interna para evitar a possibilidade de manipulação por código externo. Acessando-se a classe Curso, modificaremos a propriedade Aulas, em que encontramos get e set.

Removeremos o set, pois do jeito que está, qualquer um pode alterá-lo para um valor nulo, e a classe Curso perde qualquer chance de barrar esta modificação. Feito isso, retornaremos não mais uma lista de aulas (List<Aula>) em get, mas outro tipo de objeto, uma **coleção somente leitura** (*read only collection*).

ReadOnlyCollection se localiza no .NET com *namespace* System.Collections.ObjectModel, e é uma classe genérica, portanto necessita também do tipo de elemento (no caso, a classe Aula). Precisaremos passar no construtor um parâmetro que é uma lista, a própria referência das aulas.



Aulas é o campo privado que está sendo protegido, porém o Visual Studio não consegue converter ReadOnlyCollection para um List<Aula>, declarado na propriedade. Portanto, precisaremos trocá-lo por um IList<Aula>, que é o que o ReadOnlyCollection implementa. O código ficará desta forma:

class Curso

{

private IList<Aula> aulas;

public IList<Aula> Aulas

{

get { return new ReadOnlyCollection<Aula>(aulas); }

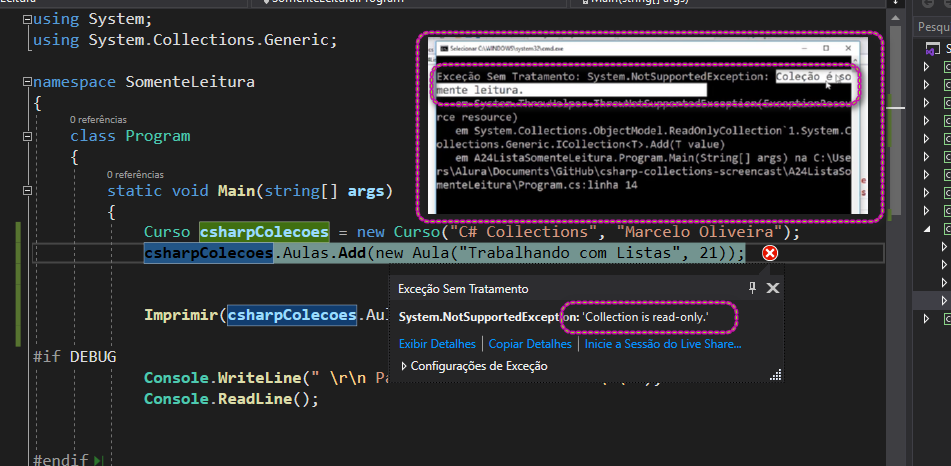
}

}

Veremos outro problema, desta vez na hora de chamarmos o método Imprimir(), pois estamos passando uma propriedade de tipo IList<>, e nosso método recebe List<>. Precisaremos fazer a devida alteração.

Feito isso, rodaremos a aplicação!

Há um erro:

System.NotSupportedException "Coleção é somente leitura". Protegemos nossa coleção para que ela fosse somente leitura, e assim, o código externo fica impedido de modificar esta coleção.

É preciso encapsularmos esta funcionalidade de adicionar algum elemento na lista de aulas em algum método da classe Curso, porque aí sim, ela terá o poder de controlar esta funcionalidade.

Para facilitar, como temos csharpColecoes.Aulas.Add(), poderemos trocar por csharpColecoes.Adiciona(). No entanto, este novo método ainda não existe. Com "Ctrl + .", clicaremos no atalho Generate method 'Curso.Adiciona'.

Com F12, entraremos no método para o implementarmos:

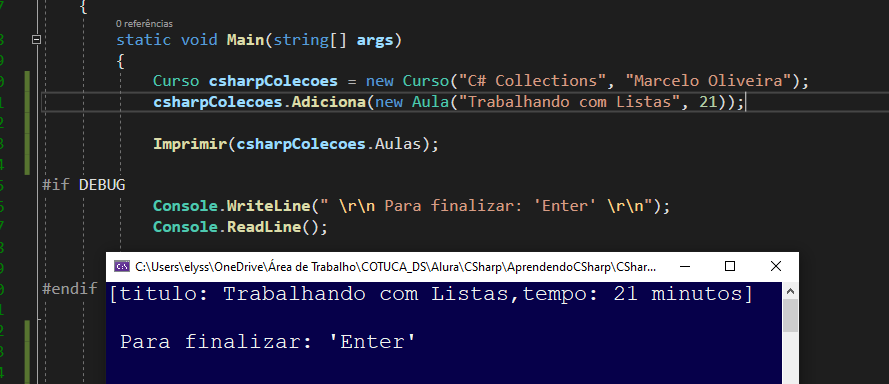
internal void Adiciona (Aula aula)

{

this.aulas.Add(aula);

}

Rodando novamente a aplicação, teremos:



[título: Trabalhando com Listas, tempo: 21 minutos]

# **Lista somente-leitura**

Pronto para um momento de reflexão? Em que situação e como você usaria uma lista somente-leitura? Escolha a melhor alternativa!

Uma classe possui uma lista List<T>, que é um campo privado da classe, e essa lista precisa ser protegida contra gravação de fora da classe. Então uma propriedade pública pode expor para os clientes da classe uma nova instância de uma lista somente-leitura (ReadOnlyCollection<T>), usando como origem de dados a lista interna que precisa ser protegida.

Mantendo uma lista interna privada como List<T> permitirá que a classe tenha total controle sobre essa lista e evitará acesso externo. Para clientes dessa classe, uma cópia da lista interna através de uma nova instância de ReadOnlyCollection<T> servirá para leitura dos dados mas deixará claro aos clientes que essa lista não pode ser modificada.

# **Ordenando e Totalizando**

No último vídeo aprendemos a proteger a coleção de aulas do curso, modificando o retorno como sendo de **coleção somente leitura**, ou *read only collection*. O que faremos agora são algumas operações em cima desta lista de aulas.

Inicialmente, adicionaremos duas aulas novas neste curso:

static void Main(string[] args)

{

Curso csharpColecoes = new Curso("C# Collections", "Marcelo Oliveira");

csharpColecoes.Aulas.Add(new Aula("Trabalhando com Listas", 21));

Imprimir(csharpColecoes.Aulas);

//adicionar 2 aulas

csharpColecoes.Adiciona(new Aula("Criando uma Aula", 20));

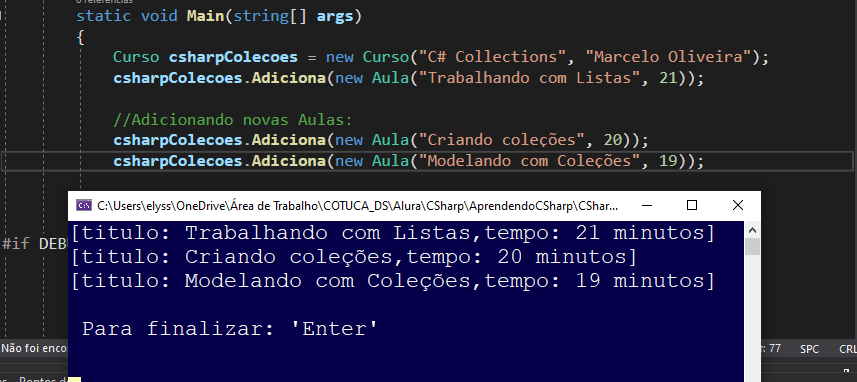
csharpColecoes.Adiciona(new Aula("Modelando com Coleções", 19));

//imprimir

Imprimir(csharpColecoes.Aulas);

}

Rodaremos com "Ctrl + F5" e obteremos como resultado:



[título: Trabalhando com Listas, tempo: 21 minutos]

[título: Criando uma Aula, tempo: 20 minutos]

[título: Modelando com Coleções, tempo: 19 minutos]

Em seguida, colocaremos estas aulas em ordem alfabética:

static void Main(string[] args)

{

Curso csharpColecoes = new Curso("C# Collections", "Marcelo Oliveira");

csharpColecoes.Aulas.Add(new Aula("Trabalhando com Listas", 21));

Imprimir(csharpColecoes.Aulas);

//adicionar 2 aulas

csharpColecoes.Adiciona(new Aula("Criando uma Aula", 20));

csharpColecoes.Adiciona(new Aula("Modelando com Coleções", 19));

//imprimir

Imprimir(csharpColecoes.Aulas);

//ordenar a lista de aulas

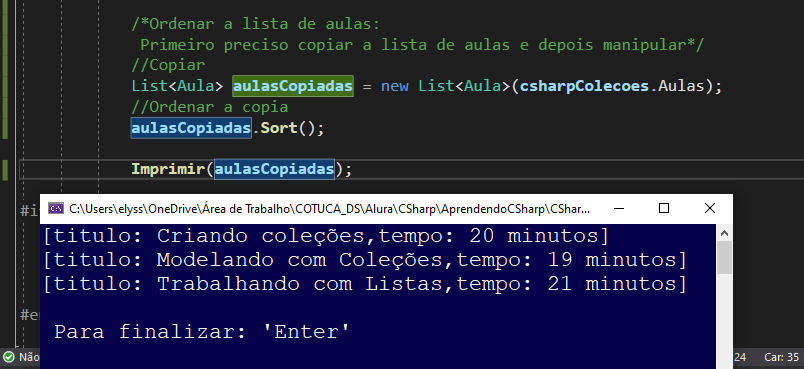
csharpColecoes.Aulas.Sort();

}

Ao rodarmos o código acima, ocorre um erro que indica que IList<Aula> não contém a definição para Sort(), pois quando vamos à definição de csharpColecoes.Aulas com a tecla F12, veremos que ela foi declarada como um IList<>, interface que não dá suporte ao método de ordenação.

Em Program.cs, tentaremos ordenar de outra maneira, copiando a lista para outra, por exemplo. Só que desta vez não usaremos uma lista somente leitura, e sim alguma que permita, além da ordenação, outros tipos de modificação.

Esta definição de aulas copiadas irá trazer uma lista vazia, e não é isto que queremos. Queremos instanciar uma lista com todas as aulas existentes no curso. Para criar uma lista previamente populada com estas aulas, passaremos a listagem no construtor. Feito isso, poderemos ordená-la propriamente:



//ordenar a lista de aulas

//csharpColecoes.Aulas.Sort();

//copiar a lista para outra lista

List<Aula> aulasCopiadas = new List<Aula>(csharpColecoes.Aulas);

//ordenar a cópia

aulasCopiadas.Sort();

Imprimir(aulasCopiadas);

Rodaremos com "Ctrl + F5" e veremos que tudo foi feito como gostaríamos.

A próxima tarefa será totalizar o tempo do curso. Em Curso.cs, acrescentaremos o seguinte:

public string Instrutor

{

get { return instrutor; }

set { instrutor = value; }

}

public int TempoTotal

{

get

{

int total = 0;

foreach (var aula in aulas)

{

total += aula.Tempo;

}

return total;

}

}

E em Program.cs teremos:

//totalizar o tempo do curso

Console.WriteLine(csharpColecoes.TempoTotal);

Rodando a aplicação, o resultado da soma é 60, como total do curso. Poderemos consegui-lo de outra forma: há um conjunto de técnicas chamado **LINQ**, criado pela Microsoft como um conjunto de extensões que dá suporte a uma série de consultas que podem ser feitas em cima de coleções, possibilitando também o trabalho com banco de dados, XML, e outros.

O que fazemos com nosso acumulador para totalizar o tempo do curso, na verdade, é uma consulta, que pode ser feita de modo completamente diverso, mais simples e elegante.

O método Sum() exige que passemos um seletor, uma expressão lambda e a propriedade da aula que nos interessa, Tempo. Em Curso.cs, então:

get

{

//int total = 0;

//foreach (var aula in aulas)

//{

// total += aula.Tempo;

//}

//return total;

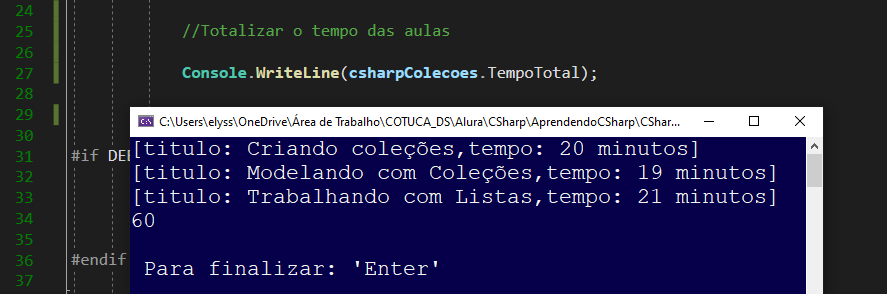
//LINQ = Language Integrated Query

//Consulta Integrada à Linguagem

return aulas.Sum(aula => aula.Tempo);

}

Voltaremos a Program.cs e rodaremos novamente para confirmar o mesmo resultado que já tínhamos (60). Para finalizar, imprimiremos este curso,

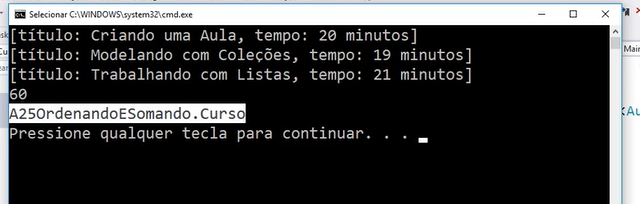
 acrescentando a variável csharpColecoes:

//totalizar o tempo do curso

Console.WriteLine(csharpColecoes.TempoTotal);

Console.WriteLine(csharpColecoes);

Rodando, teremos como resultado o namespace A25OrdenandoESomando.Curso, porém queremos o nome, duração e demais especificidades do curso.

 Neste caso, teremos que implementar um método, o ToString().

Quando imprimimos um objeto qualquer que ainda não implementou esta propriedade, chamamos o método ToString() do Object do .NET, que imprimirá o namespace e o nome da classe.

Utilizaremos override, uma operação de sobrescrita de um método cuja classe base já existe (no caso, Object). Para as aulas, usaremos o string.Join juntará as strings da coleção, separando-as por vírgulas. Teremos:

public override string ToString()

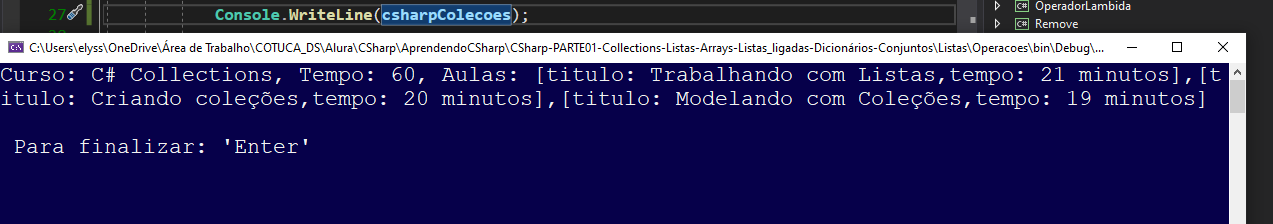
{

return $"Curso: {nome}, Tempo: {TempoTotal}, Aulas: {string.Join(",", aulas)}";

}

Em Program.cs, rodaremos novamente a aplicação. O resultado é:

Curso: C# Collections, Tempo: 60, Aulas: [título: Trabalhando com Listas, tempo: 21 minutos],[título: Criando uma Aula, tempo: 20 minutos],[título: Modelando com Coleções, tempo: 19 minutos]



Conseguimos o que gostaríamos!

# **Totalizando valores**

Lembra da classe Aula? Nós vamos usá-la de novo neste exercício:

public class Aula

{

private string titulo;

public string Titulo { get { return titulo; } set { titulo = value; } }

private int tempo;

public int Tempo { get { return tempo; } set { tempo = value; } }

public Aula(string titulo, int tempo)

{

this.titulo = titulo;

this.tempo = tempo;

}

}

O que vamos fazer com a classe Aula?

Criamos uma lista de aulas, como declarada abaixo:

var aulaIntro = new Aula("Introdução às Coleções", 20);

var aulaModelando = new Aula("Modelando a Classe Aula", 18);

var aulaSets = new Aula("Trabalhando com Conjuntos", 16);

List<Aula> aulas = new List<Aula>();

aulas.Add(aulaIntro);

aulas.Add(aulaModelando);

aulas.Add(aulaSets);

**Agora é com você**:

Nós precisamos totalizar o tempo dessas aulas. Responda abaixo, escolhendo a melhor alternativa!

var tempoTotal = aulas.Sum(aula => aula.Tempo);

O método Sum() toma como argumento um selector, que pode ser uma expressão lambda recebendo como parâmetro um elemento da coleção e retornando o valor da propriedade Tempo, que será totalizada.

# **Mão na Massa: Lista, a Coleção Flexível**

Vamos botar a mão na massa mais uma vez, desta vez para trabalhar com o tipo de coleção mais poderosa e flexível .NET Framework: a classe List<T>, também conhecida como array dinâmico.

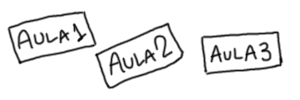
# Declarando uma lista

Lembra daquelas 3 variáveis de strings da última aula? Vamos usá-las novamente:

string aulaIntro = "Introdução às Coleções";

string aulaModelando = "Modelando a Classe Aula";

string aulaSets = "Trabalhando com Conjuntos";



Agora vamos declarar nossa lista.

# A sintaxe de uma lista de strings em C

Diferente de um array, uma lista (List<T>) não possui uma sintaxe especial em C#. Na declaração, usamos a classe genérica List<T>, onde o tipo T especifica a natureza dos seus elementos.

# Declarando e inicializando uma lista



Pode ser que você queira criar uma vazia. Mas também pode ser que você queira declarar E preencher de uma vez só. Nesse caso, usamos a sintaxe com as chaves {...} incluindo uma lista de itens iniciais.

List<string> aulas = new List<string>

{

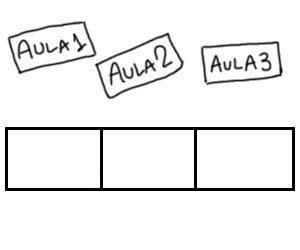
aulaIntro,

aulaModelando,

aulaSets

};

# Inicializando uma lista vazia



Já uma lista vazia tem a sintaxe normal de uma declaração de objeto em C#:

List<string> aulas = new List<string>();

# A lista como um array dinâmico

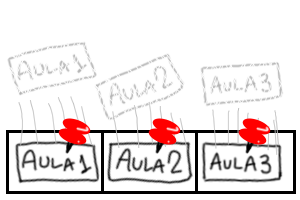
Como assim, array dinâmico?

Simples: um array é uma coleção de tamanho fixo, com número estático de elementos, certo?

Então a lista (List<T>) é parecida, só que quando você adiciona/remove elementos dela, o tamanho da lista aumenta ou diminui, respectivamente.

E é por isso mesmo que podemos iniciar uma lista vazia, com tamanho zero, adicionando elementos posteriormente conforme desejado.

# Adicionando elementos



O jeito mais fácil de alimentar uma lista é adicionando elementos. E isso fazemos com o método Add(), que significa "adicionar" em inglês.

aulas.Add(aulaIntro);

aulas.Add(aulaModelando);

aulas.Add(aulaSets);

Com esse método, um elemento é sempre adicionado ao final da lista.

Ou seja, a lista funciona **como uma sequência**.

# Imprimindo listas

Lembra de quando tentamos imprimir um array no console?

Que tal fazer o mesmo com uma lista?

Console.WriteLine(aulas);

Como antes, esse código não imprime os elementos da lista individualmente. Em vez disso, ele só imprime o namespace seguido do nome da classe:

System.Collections.Generic.List`1[System.String]

Então, novamente, precisamos imprimir os elementos da lista um a um. Vamos criar o método Imprimir() para isso:

private static void Imprimir(List<string> aulas)

{

foreach (var aula in aulas)

{

Console.WriteLine(aula);

}

Console.WriteLine();

}

O código acima utiliza um laço com a instrução foreach. Assim como com arrays, podemos também usar um laço com instrução for:

private static void Imprimir(List<string> aulas)

{

for (int i = 0; i < aulas.Count; i++)

{

Console.WriteLine(aulas[i]);

}

Console.WriteLine();

}

Mas uma List<T> ainda possui mais uma maneira de varrer a coleção: o método ForEach():

private static void Imprimir(List<string> aulas)

{

aulas.ForEach(aula =>

{

Console.WriteLine(aula);

});

Console.WriteLine();

}

O que esse método faz?

Ele recebe uma Action, que é um **delegate**, ou delegado, que encapsula um método que não retorna valor (método void).

Note que, acima, a Action é declarada como **expressão lambda**:

aula =>

{

Console.WriteLine(aula);

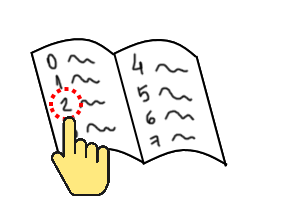
}

Uma **expressão lambda** é uma função anônima (isto é, um método sem nome) que permite criar uma expressão inline, ou em linha, sem a necessidade de referenciar um método externo.

Uma **expressão lambda** contém três partes:

* o parâmetro (aula)
* o operador lambda (=>)
* o corpo do método anônimo ({ Console.WriteLine(aula); })

# Índices



Uma List<T> também possui indexador, e é por isso que podemos usar a sintaxe que vimos em arrays:

Console.WriteLine("A primeira aula é " + aulas[0]);

Note que a mesma sintaxe de índice pode ser usada tanto para ler quanto para modificar items:

aulas[0] = "Trabalhando com Listas";

# Acessando elementos de uma lista

Para pegar o primeiro elemento da lista (posição 1, índice 0), basta acessar o índice adequado:

Console.WriteLine("A primeira aula é " + aulas[0]);

Mas também há um outro jeito de fazer isso, utilizando o método First():

Console.WriteLine("A primeira aula é " + aulas.First());

Note que o método First() não faz parte da classe List<T>: ele é fornecido pela extensão LINQ (Language INtegrated Queries, ou Consultas Integradas à Linguagem), que existe num namespace separado, o namespace System.Linq.

Mas não há muita diferença entre usar índice ou usar First para acessar o primeiro item, não é verdade? Mas o mesmo não podemos dizer quando queremos pegar o último elemento:

Na sintaxe de índice temos:

Console.WriteLine("A última aula é " + aulas[aulas.Count - 1]);

Já usando LINQ, o método Last() se torna bem mais legível!

Console.WriteLine("A última aula é " + aulas.Last());

Gostou do LINQ? Aprenda o que há por trás dessa tecnologia de consultas de coleções, fazendo nossos cursos [LinQ parte 1](https://cursos.alura.com.br/course/linq-c-sharp) e [LinQ parte 2](https://cursos.alura.com.br/course/linq-c-sharp-parte-2) aqui na Alura!

# Procurando elementos



Aproveitando um pouco mais das extensões do LINQ, vamos procurar agora algums elementos pelo seu conteúdo.

# Encontrando a primeira ocorrência

**"Qual é a primeira aula da lista que contém a palavra 'Trabalhando'"?**

Pra responder essa pergunta vamos usar o método First() (lembra dele)?

aulas.First()

Mas desta vez, vamos utilizar uma sobrecarga, ou overload, ou seja, uma variação deste método, uma que aceita como parâmetro um predicate (predicado) que descreve qual condição deve ser satisfeita nessa busca.

aulas.First(aula => ...))

E acima, de novo, usamos expressão lambda... melhor você se acostumar com ela!

A condição é: **que contém a palavra 'Trabalhando'**.

Traduzindo isso para a nossa querida linguagem C#, fica assim:

aulas.First(aula => aula.Contains("Trabalhando"))

Colocando tudo junto, imprimimos o resultado no console:

Console.WriteLine("A primeira aula 'Trabalhando' é: "

+ aulas.First(aula => aula.Contains("Trabalhando")));

# Encontrando a última ocorrência

E pra pegar a última aula da lista com a palavra "Trabalhando"?

Mesma coisa, só que usamos agora o método Last():

Console.WriteLine("A última aula 'Trabalhando' é: "

+ aulas.Last(aula => aula.Contains("Trabalhando")));

**Mas há um problema nesses métodos**: se nenhum elemento for encontrado, tomaremos um erro!

System.InvalidOperationException: 'A sequência não contém elementos de correspondência'

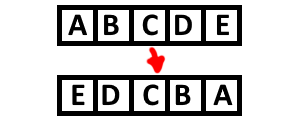
Então, nesse caso, podemos resolver o problema com um método similar, que devolve o valor default caso não haja ocorrências:

Console.WriteLine("A primeira aula 'Conclusão' é: "

+ aulas.FirstOrDefault(aula => aula.Contains("Conclusão")));

E o que seria, exatamente, esse valor default? Depende. Em um objeto ou string, seria um null. Em um decimal, double, etc. seria um zero. E assim por diante.

# Revertendo lista



Para reverter a ordem de uma lista, usamos o método conhecido, Reverse:

aulas.Reverse();

Imprimir(aulas);

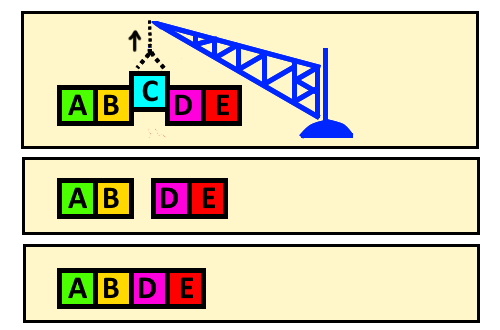
Mas qual a diferença para o método que usamos em arrays? Apenas o fato de que esse método não é estático.

E se quisermos voltar a lista para sua ordem original, basta executar o método novamente:

aulas.Reverse();

Imprimir(aulas);

# Removendo um elemento



Podemos remover o elemento de uma lista pela sua posição.

Nesse caso, passando o índice desejado. Se é o último, então índice = aulas.Count - 1.

aulas.RemoveAt(aulas.Count - 1);

Imprimir(aulas);

Se você precisar remover uma região de uma lista, o caminho é o método RemoveRange.

Vamos mostrar um exemplo de remoção da última e penúltima aulas da lista:

aulas.RemoveRange(aulas.Count - 2, 2);

Imprimir(clone);

O método acima remove uma região (range) que começa no penúltimo elemento (aulas.Count - 2) e abrange 2 elementos.

Mas também podemos remover elemento pelo seu conteúdo. Se nossa lista é de strings, temos que passar uma string como parâmetro:

aulas.Remove("Conclusão");

Imprimir(aulas);

Lembre-se de que uma lista é uma coleção dinâmica! Perceba que todos esses métodos para remover elementos acabam afetando o tamanho da lista.

# Ordenando lista



Listas podem ser ordenadas com o método Sort. Uma lista de strings será posta em ordem alfabética:

aulas.Sort();

Imprimir(aulas);

Mas o método Sort tem um outro overload ("sobrecarga") que recebe um IComparison (comparação), que que define o critério de ordenação entre os elementos.

Num exemplo maluco, podemos ordenar nossa lista de aulas por ordem de tamanho da string:

aulas.Sort((a, b) => a.Length.CompareTo(b.Length));

Imprimir(aulas);

# Copiando lista



Para copiar uma lista, a abordagem é bem diferente daquela que vimos com arrays.

A classe List<T> possui um método chamado GetRange(), para "pegar uma região" de uma lista original e assim gerar uma nova lista com esses elementos copiados:

List<string> copia = aulas.GetRange(aulas.Count - 2, 2);

Imprimir(copia);

# Clonando lista



Mas assim como em arrays, se o que você precisa é apenas uma cópia de todos os elementos de uma lista, em vez de usar o método Clone(), você pode clonar criando uma nota instância de List<T> passando como parâmetro do construtor a lista a ser clonada:

List<string> clone = new List<string>(aulas);

Imprimir(clone);

Ufa, quanta coisa!

Listas são coleções importantíssimas em .NET Framework. Dominar os fundamentos dessa coleção é vital para um bom aproveitamento da programação com C#.

E sobre o LINQ? Você pode turbinar suas aplicações com essa incrível ferramenta de consultas em C#, fazendo os cursos [LinQ parte 1](https://cursos.alura.com.br/course/linq-c-sharp) e [LinQ parte 2](https://cursos.alura.com.br/course/linq-c-sharp-parte-2) aqui na Alura!

A seguir, vamos começar a estudar um outro tipo de coleção: os **conjuntos**. Até à próxima!

# **O que aprendemos?**

**O que aprendemos?**

* Declarando uma lista
* A lista como um array dinâmico
* A sintaxe de uma lista de strings em C#
  + Declarando e inicializando uma lista
  + Inicializando uma lista vazia
* Índices
* Acessando elementos de uma lista
* Adicionando elementos
* Removendo elementos
* Procurando elementos
  + Encontrando a primeira ocorrência
  + Encontrando a última ocorrência
* Verificando se um elemento está contido ou não
* Revertendo lista
* Acessando elementos pelo índices
* Removendo elemento pelo índice
* Ordenando lista
* Copiando lista
* Clonando lista
* Limpando elementos de uma lista
* Trabalhando com listas de objetos custom
* Sobreescrevendo o método ToString()
* Implementando a interface IComparable
* Implementando o método CompareTo()
* Utilizando o método Sort()
* Expressão lambda como argumento do método Sort()
* A interface IList
* Listas somente-leitura com ReadOnlyCollection
* Acumulando valores de objetos de uma lista
* Somando valores com o método Sum()

# **xxxxxxxx**

#### O Poder dos Sets

#### Dicionários, Trabalhando com Chave e Valor

#### Lista Ligada, Pilha e Fila

#### Arrays e Tipos genéricos

#### Collections

#### Qual Coleção Usar

# **C# parte 8: List, lambda, linq**

Olá! Sou Guilherme Costa e dou as boas-vindas à parte 8 da série de cursos de C#. Nela, estudaremos List<T>, que consiste na implementação de lista, dentro do .NET.

Perceberemos que esse tipo também lida com os métodos Add() e Remove(). No entanto, List<T> não implementa o método AdicionarVarios(). Teremos que criá-lo e chamá-lo por meio de uma referência de List<T>. Durante esse processo, aprenderemos o que é um método de extensão, desenvolvendo o nosso próprio que, no caso, será um método genérico. Entenderemos como ele e a inferência de tipos do compilador funcionam para métodos como AdicionarVarios().

Ainda sobre inferência de tipo, estudaremos os benefícios de utilizar var, uma palavra reservada da linguagem C#. Exploraremos o mecanismo do inicializador de listas e a organização delas, um problema natural de lidar com diversos itens.

Sendo assim, conheceremos meios de ordenar listas de string, int e a própria lista do tipo ContaCorrente. Para ordenarmos ContaCorrente, passaremos por diversas interfaces do .NET, como IComparable(), entendendo como ela funciona e como devemos implementar o método CompareTo() e suas particularidades.

Utilizaremos uma implementação específica e padrão para ContaCorrente, mas acabaremos querendo ordenar de outras formas. Isso nos levará à interface IComparer(), que é genérica no tipo que queremos ordenar.

Ao longo do curso, nos depararemos com muitos códigos, descobrindo outras formas de fazer isso, utilizando operadores de link, conhecendo o mecanismo de OrderBy e Where, além de entender códigos diferentes de uma expressão *lambda*, como (conta => conta.Numero) e (conta => conta != null), que retorna um Booleano — verdadeiro ou falso.

#### ListT e métodos de extensão

# **Classe List do dotNet**

Vamos continuar os estudos? Para começar a parte 8 da série de cursos de C#, analisaremos o trabalho — muito legal — que fizemos em Program.cs, criando a seguinte lista genérica:

Lista<int> idades = new Lista<int>();

idades.Adicionar(1);

idades.Adicionar(5);

idades.Adicionar(14);

idades.Adicionar(25);

idades.Adicionar(38);

idades.Adicionar(61);

for (int i = 0 < idades.Tamanho; i++)

{

Console.WriteLine(idades[i]);

}

Console.ReadLine();

Nela, temos várias chamadas para o método Adicionar(). Se executarmos a aplicação, teremos as idades listadas como retorno:

1

5

14

25

38

61

Abaixo da lista de Adicionar(), podemos inserir Remover() para retirar algum item. No caso, removeremos 5:

Lista<int> idades = new Lista<int>();

idades.Adicionar(1);

idades.Adicionar(5);

idades.Adicionar(14);

idades.Adicionar(25);

idades.Adicionar(38);

idades.Adicionar(61);

idades.Remover(5);

Ao executar, nossa expectativa será atendida: a lista de idades aparecerá sem o número 5:

1

14

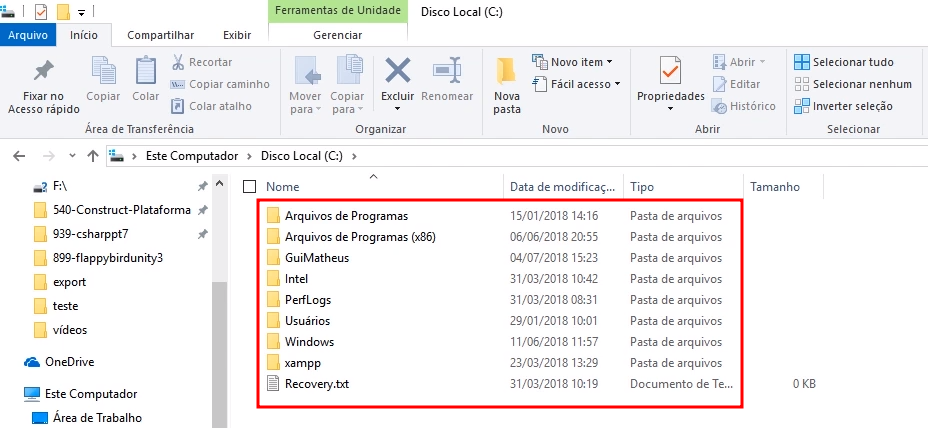
25

38

61

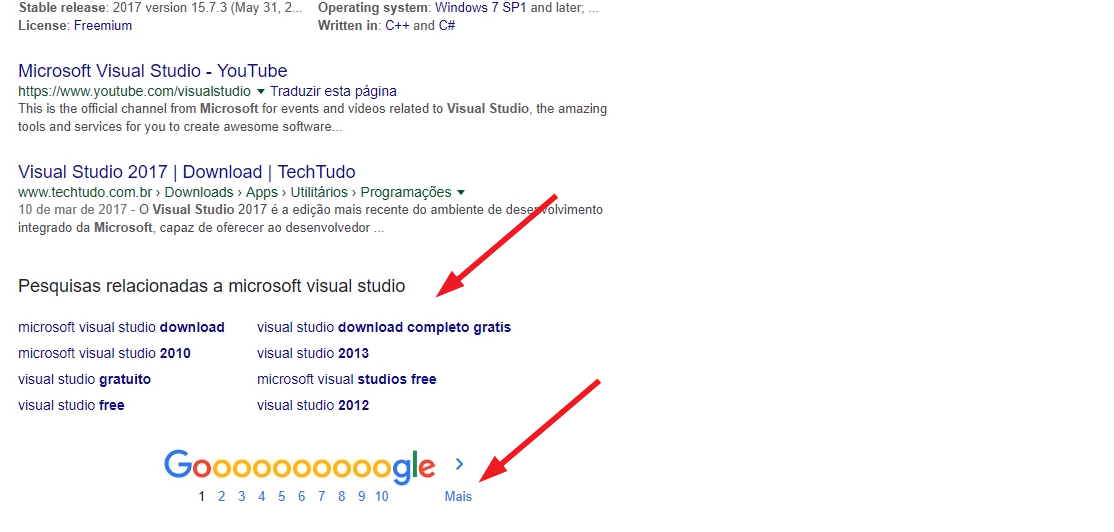
Essa foi a lista genérica que criamos com Adicionar(), método genérico que espera um argumento do tipo genérico também.

Mas, será que nosso problema em lidar com arrays e manipular esses dados até criarmos a lista é um problema tão raro? Na verdade, não, pois isso faz parte de vários programas que acabamos usando no cotidiano. Por exemplo, se abrirmos o Windows Explorer, no explorador de arquivos, encontramos uma lista de itens.



No caso do exemplo, esses itens seriam as pastas e o arquivo. Contudo, para gerar essa visualização na tela, temos uma lista, com várias pastas e um arquivo.

Outro exemplo de lista bastante comum no nosso dia a dia, é o que encontramos quando realizamos qualquer busca na internet. Se acessarmos o [Google](http://google.com/) e digitarmos microsoft visual studio, teremos uma [lista de resultados](https://www.google.com.br/search?q=microsoft+visual+studo&oq=microsoft+visual+studo&aqs=chrome..69i57.6375j0j1&sourceid=chrome&ie=UTF-8). Ainda no final da página, abaixo dos resultados, encontramos uma lista de sugestões relacionadas e outra lista de páginas com mais resultados.



Ou seja, lista é algo muito comum. Mas se é tão comum assim, será que todo desenvolvedor e toda desenvolvedora que desenvolve com C# precisa criar sua própria lista, repetidas vezes? Na verdade, existe uma lista, dentro do .NET, que é muito semelhante à que criamos. Inclusive no nome: List, que é Lista em português, como temos no código acima.

Se substituirmos Lista por List:

List<int> idades = new List<int>();

O Visual Studio já encontrará essa classe genérica, que permite o uso de int como argumento genérico. No entanto, perceba que Adicionar(), Remover() e Tamanho ficarão sublinhados em vermelho, por estarem em português.

Se os alterarmos para inglês, o sublinhado vermelho abaixo deles desaparecerá. Sendo assim, aplicaremos as respectivas substituições:

* Adicionar por Add;
* Remover por Remove;
* Tamanho por Count, propriedade do .NET responsável pela contagem de itens que compõem a lista.

List<int> idades = new List<int>();

idades.Add(1);

idades.Add(5);

idades.Add(14);

idades.Add(25);

idades.Add(38);

idades.Add(61);

idades.Remove(5);

for (int i = 0 < idades.Count; i++)

{

Console.WriteLine(idades[i]);

}

Console.ReadLine();

Para nos certificarmos de que as substituições são equivalentes, executaremos a aplicação novamente:

1

14

25

38

61

Obtivemos a mesma saída, indicando que o código é executado, sem problemas. Se comentarmos a linha de Remove(), adicionando duas barras (//) no início dela:

// idades.Remove(5);

O número 5 voltará a ser impresso na lista:

1

5

14

25

38

61

Na série do curso [C# Collections Parte 1](https://www.alura.com.br/curso-online-csharp-collections) você encontra mais conteúdo sobre lista, array, sets e outros tipos de coleções do .NET.

Continuando, constatamos que o List possui Add() e Remove(). Mas há outro método bastante interessante em Lista.cs, o AdicionarVarios():

public void AdicionarVarios(params T[] itens)

{

foreach (T item in itens)

{

Adicionar(item);

}

}

O bacana desse método é o argumento params, que permite chamar AdicionarVarios() de uma forma bastante natural, sem exigir a criação manual de um array.

Será que o List do .NET possui esse método? Para testar, poderíamos adicionar AddRange(), abaixo do trecho de Add(), em Program.cs. No entanto, ele não espera um argumento params.

Portanto, não seria possível utilizar números como parâmetros, esperando que o compilador crie um array e depois o passe como argumento, porque AddRange() não funciona dessa forma. Ou seja, implementa-lo como demonstrado no trecho abaixo, não funcionaria:

List<int> idades = new List<int>();

idades.Add(1);

idades.Add(5);

idades.Add(14);

idades.Add(25);

idades.Add(38);

idades.Add(61);

idades.AddRange(1, 2, 3, 9);

// idades.Remove(5);

for (int i = 0 < idades.Count; i++)

{

Console.WriteLine(idades[i]);

}

Console.ReadLine();

Se criarmos um array de int, utilizando aquela sintaxe que estudamos de inicialização de array\* , na qual não precisamos nos preocupar com número inicial de elementos, nem com a adição de item por item no \*array, da seguinte forma:

idades.AddRange(new int[] {1, 2, 3, 9});

Aplicando essa sintaxe, AddRange() é compilado, sem problemas. Ao executarmos o código após a alteração, obteremos a lista com os números esperados:

1

5

14

25

38

61

1

2

3

9

Mas poderia estar melhor, igual à lista que criamos:

idades.AddRange(1, 2, 3, 9);

No entanto, AddRange() não é compilado dessa maneira, ficando sublinhado em vermelho. Seria legal se houvesse uma forma de estender a classe List.cs e colocar o nosso método funcionando na lista também. Pensando nisso, temos uma alternativa, que já estudamos: criar uma classe estática, com métodos auxiliares.

Sendo assim, criaremos uma abrindo o "Gerenciador de Soluções", à direita da tela, e selecionando "ByteBank.SistemaAgencia > Adicionar > Classe...". A nomearemos como ListExtensoes.cs, já que armazenaremos as extensões da classe List.cs do .NET nela.

Como ela somente armazenará extensões, ou seja, não possuirá estado, nem será instanciada, a definiremos como estática (static) e colocaremos AdicionarVarios() no escopo dela, mantendo List<int> — adiante nos preocupamos em torná-lo genérico — como argumento:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.SistemaAgencia

{

public static class ListExtensoes

{

public void AdicionarVarios(this List<int> listaDeInteiros, params int[] itens)

{

foreach(int item in itens)

{

listaDeInteiros.Add(item);

}

}

}

}

Inserimos params como segundo argumento de AdicionarVarios(), assim basta coletarmos todos os elementos do array itens e adicioná-los, um a um, à listaDeInteiros. No corpo de AdicionarVarios(), poderíamos utilizar:

* for e variar i de 0 até a quantidade de itens;
* foreach(), que não traz o índice.

Como queremos somente varrer item por item, atribuindo uma ação para cada um deles — sem nos preocuparmos com o índice — utilizamos foreach(), estabelecemos int item in itens como argumento e inserimos listaDeInteiros.Add(item) no escopo dele. Com o que escrevemos, AdicionarVarios() não é compilado, ficando sublinhado em vermelho. Faltou adicionar o modificador static a ele.

Lembrando que em uma classe estática, os membros também devem ser estáticos.

Sendo assim, ao defini-lo como static, o código será compilado:

namespace ByteBank.SistemaAgencia

{

public static class ListExtensoes

{

public static void AdicionarVarios(this List<int> listaDeInteiros, params int[] itens)

{

foreach(int item in itens)

{

listaDeInteiros.Add(item);

}

}

}

}

Com AdicionarVarios() declarado na classe ListExtensoes.cs, como podemos utilizá-lo? De volta a Program.cs, abaixo de AddRange(), adicionaremos:

ListExtensoes.AdicionarVarios(idades, 1, 5687, 1987, 1567, 987);

Definimos idades — que contém nossa lista de itens — como primeiro argumento, seguido por números aleatórios, sem nos preocuparmos com a sinaxe de criar array, passar os índices e colocar colchetes ([]). Ficou muito mais simples. Vamos executar para certificar que o código funciona:

1

5

14

25

38

61

1

5687

1987

1567

987

Funcionou, todos os itens foram adicionados. Mas ainda falta naturalidade em chamar um método estático e passar a lista como argumento. Seria melhor continuar escrevendo o nome da lista (idades) e, então, AdicionarVarios(), utilizando a sintaxe que aprendemos:

idades.AdicionarVarios(5454, 46, 54);

Mas, da forma que fizemos, isso não é possível. AdicionarVarios() fica sublinhado em vermelho, indicando que o código não é compilado. Adiante, aplicaremos os ajustes necessários.

# **xxxxxxxx**

#### Método de extensão genérico

#### Usando VAR e método Sort

#### IComparable e IComparer

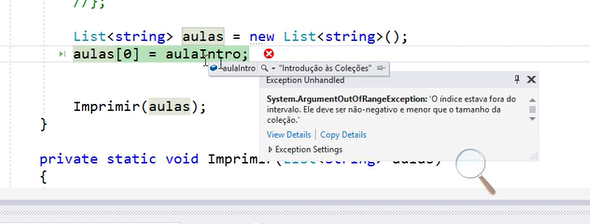
#### OrderBy e expressões Lambda

#### Linq e operador Where

# Principais comandos

|  |  |
| --- | --- |
| Definir pontos de interrupção no código-fonte | clique na margem da extrema esquerda ao lado de uma linha de código. Você também pode selecionar a linha e pressionar **F9**, selecionar > **ponto de interrupção de alternância** de depuração, ou clicar com o botão direito do mouse e selecionar **ponto de** interrupção > **Inserir ponto** de interrupção. O ponto de interrupção aparece como vermelho na margem esquerda. |
| Cw + tab + tab | Atalho para digitar Console.WriteLine() |
| Ctrl . | \*\*Selecione o trecho de código que se tornará um metodo\*\* Para extrair o metodo |
| WriteLine(); -- ,\n | Para pular linha |
| "propfull + TAB + TAB" | para criar a estrutura da propriedade necessária, com get e set |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Conflitos



Ao rodarmos a aplicação, teremos um erro! Um System.ArgumentOutOfRangeException ("O índice estava fora do intervalo. Ele deve ser não-negativo e menor que o tamanho da coleção"). O que acontece é que estamos acessando o índice, mas ele é inexistente na lista, pois está vazia. É necessário fazermos isto de outra forma.

Em se tratando de listas, não a atribuímos desta maneira para acrescentarmos elementos. Para isto, na variável de aulas, acessaremos o método de adicionar Add():

List<string> aulas = new List<string>();

aulas.Add(aulaIntro);

aulas.Add(aulaModelando);

aulas.Add(AulaSets);

Imprimir(aulas);

# Conclusão