PROGRAMAÇÃO I

AULA 3

Prof. Alan Matheus Pinheiro Araya

CONVERSA INICIAL

LINQ, TIPOS ANÔNIMOS E LAMBDA FUNCTIONS

O objetivo desta aula será abordar um conjunto de recursos e funções da linguagem C# que são primordiais para o dia a dia do desenvolvedor .NET com coleções e em plataformas Web/Desktop/Mobile.

Recursos como LINQ e Lambda Functions são tão comuns quanto o uso de um simples foreach. Nesta aula, vamos destrinchar esses assuntos e torná-los mais simples possíveis para sua utilização.

TEMA 1 – INTRODUÇÃO AO LINQ E LAMBDAS FUNCTIONS

LINQ é um acrônimo para *Language Integrated Query*, sendo um conjunto de sublinguagem de consulta e funções do framework .NET para escrever consultas em objetos ou fontes remotas (como bancos de dados). Originalmente introduzido na versão 3.0 e 3.5 do .NET Framwork (Albahari, 2017, p. 369).

Os recursos do LINQ permitem que você consulte qualquer coleção que implemente IEnumerable <T>, seja uma Array, Lista ou XML DOM, bem como fontes de dados remotas, como tabelas em um banco de dados SQL. O LINQ unifica os benefícios “tipagem” em tempo de compilação e composição de consultas dinâmicas em coleções (Albahari, 2017, p. 369).

O LINQ possui dois tipos diferentes de expressões da linguagem, como a seguir.

- ***Fluent Syntax*** – faz o uso de *Lambdas Functions* e da concatenação de métodos formando uma “pipeline” de processamento. Provê uma sintaxe típica de linguagens funcionais (Griffiths, 2019, p. 444).
- ***Query Expressions*** – é uma forma de expressar sua consulta de forma similar a uma query SQL. Pode executar as mesmas funções *da Fluent Syntax*, mas é mais verbosa (possui uma escrita mais extensa). É muito útil de ser utilizada em Joins (união de conjuntos) e outras situações complexas de querys (Albahari, 2017, p. 363).

Antes de entendermos como utilizar as expressões LINQ, vamos conceituar Lambdas Functions.

1.1 Lambda function

Uma expressão lambda é um **método sem nome**. Analisando a documentação oficial e definindo em termos formais, o compilador converte imediatamente a expressão lambda em (Albahari, 2017, p. 153):

- Uma instância delegada;
- Uma árvore de expressão (*Expression Tree*), do tipo Expression<TDelegate>, representando o código dentro da expressão lambda em um modelo chamado de "*Object Model*" – e isso permite que a expressão lambda possa ser interpretada posteriormente em tempo de execução.

Para simplificar o entendimento de expressões lambdas, não vamos conceituar a fundo o que é um "*delegate*" ou "*expression tree*".

Vamos simplificar da seguinte maneira: uma função lambda é **um método, sem nome, sem modificadores de acesso** e, muitas vezes, com o "body" (corpo) simplificado em uma única linha.

Uma expressão lambda deve possuir uma das seguintes "anatomias":

- *(input-parameters) => expression.*; e
- *(input-parameters) => { <sequência de códigos> }*.

Dentro do universo de expressões lambdas vamos utilizar dois Types especiais que representam tipos especiais (*delegates* no C#) e que delegam a execução de um método:

- **Action e Action<T>** – encapsula métodos, que podem receber parâmetros e **não** retornam valores; e
- **Func<T>** – encapsula métodos, que podem receber parâmetros e **retornam valores**.

A seguir, veja um exemplo de Action e de Func no C#:

```
public class ExemploLambda
{
    public ExemploLambda()
    {
        //cria uma Action (delegate) para o método MostrarData() - sem par
ametros
        Action acaoMostrarData = MostraData;

        //Executa a action
        //como a Action apenas é um delegate (encapsula) o método original
        //ela executará o método MostrarData
        acaoMostrarData();

        //Cria uma nova action que recebe um parâmetro DateTime
        //Repare que estamos usando o método MostrarData
        //Porém o compilador já entendeu que deve usar o overload (versão
do método): MostrarData(Datetime data)
        Action<DateTime> acaoMostrarDataProximoDia = MostraData;

        //Cria uma função que retorna um Datetime
        //Observe que a declaração da função segue o modelo Lambda:
        //•      (input-parameters) => expression
        //No nosso caso é uma função sem parâmetros, então apenas
        //declaramos os parênteses vazios, assim: () => expression
        Func<DateTime> funcDataProx = () => CalculaDataDoProximoDia();

        //Aqui executamos a função e pegamos seu retorno em uma variável
        DateTime dataProximoDia = funcDataProx();

        //Executamos a Action que recebe um Datetime como parâmetro
        //Observe que a chamada se assemelha a um método qualquer
        //porém esta encapsulado pelo delegate (Action<T>)
        acaoMostrarDataProximoDia(dataProximoDia);
    }

    public void MostraData()
    {
        Console.WriteLine($"A data atual é: {DateTime.Now}");
    }

    public void MostraData(DateTime data)
    {
        Console.WriteLine($"A data é: {data}");
    }


    public DateTime CalculaDataDoProximoDia()
    {
        DateTime dataProximoDia = DateTime.Now.AddDays(1);
        return dataProximoDia;
    }
}
```

Observe, nesse exemplo, a *Action* e a *Action<Datetime>* apenas encapsulam (gerando apenas uma referência) o método real. Também podemos ver uma lambda expression sendo criada para executar o método *CalculaDataDoProximoDia*() e retornando seu valor para uma *Func* (*function*),

que pode ser executada de forma similar a um método ou Action. Com a diferença de que uma *Func<T>* sempre espera um método que retorne um valor (diferente de void).

No exemplo a seguir, repare que criamos uma função (método anônimo) que multiplica dois valores e uma Action para imprimir valores no console:

```
public class ExemploLambda2
{
    public ExemploLambda2()
    {
        //Expressão lambda sem body (sem "{}"). O return é implícito
        Func<int, int, int> multiplicador = (int x, int y) => x * y;

        //Passando a function como referência
        ProcessaNumeros(5, multiplicador, (int result) =>
        {
            Console.WriteLine("resultado: " + result);
        });
        //Output no console:
        //resultado: 0
        //resultado: 33
        //resultado: 132
        //resultado: 300
        //resultado: 532

        //Passando a func e action anonimamente
        ProcessaNumeros(5,
        //func anonima
        (int x, int y) =>
        {
            return x * y;
        },
        //action anonima
        (int result) =>
        {
            Console.WriteLine("resultado: " + result);
        });
    }

    public void ProcessaNumeros(int quantidade, Func<int, int, int> proces
sador,  Action<int> outputAction)
    {
        for (int i = 0; i < quantidade; i++)
        {
            outputAction(processador(i, i * 100 / 3));
        }
    }
}
```

Depois, criamos um método que processa números, fazendo um loop de cinco iterações nas quais para cada iteração é **chamada a função** de multiplicação e um **método anônimo**, que escreve o resultado no console. Observe que podemos chamar esse método passando uma variável com a

referência (*Func*) ou podemos simplesmente declarar ela anonimamente direto no parâmetro do método *ProcessarNumeros*. Ambas são equivalentes.

Observando atentamente a notação das *Lambda Functions*, você verá que se assemelham muito mesmo a métodos clássicos em termos de sintaxe. Recebem parâmetros dentro de “()” (parênteses), podem ter um body dentro ou fora de “{}” (chaves) e podem retornar valores ou não. Não aplicando-se apenas, modificadores de acesso, métodos de extensão entre outros para as Lambda.

Neste último exemplo vamos unir *Lambdas Functions* com métodos de extensão e ver como podem ser poderosos. Primeiro, vamos criar uma classe chamada “ListExtensions” com um método de extensão para qualquer Type que implemente a interface IEnumerable<string>. Nesta classe, vamos criar um método de extensão que tem como objetivo encontrar o maior valor numérico em uma lista de strings que podem ou não conter valores numéricos em seus elementos. Para isso, esse método receberá uma função cujo objetivo é encontrar o valor numérico do elemento (de uma string) e outra função cujo o objetivo é encontrar o maior valor dentro de uma lista de inteiros.

Agora vamos montar um cenário para utilizá-lo:

```
public static class ListExtensions
{
    public static int BuscaMaiorValorEmTextosComNumeros
(this ICollection<string> textos, Func<string, int> GetValor, Func<List<i
nt>, int> GetMax)
    {
        var valoresInt = new List<int>(textos.Count);
        foreach (var texto in textos)
        {
            var valorInt = GetValor(texto);
            if (valorInt > 0)
            {
                valoresInt.Add(valorInt);
            }
        }

        return GetMax(valoresInt);
    }
}
```

```
public class ExemploLambda3
{
    public ExemploLambda3()
    {
        var textos = new List<string>();
        textos.Add("texto 1");
        textos.Add("texto 2");
        textos.Add("texto");
        textos.Add("texto 10");
        textos.Add("xxxx 150");
        textos.Add("12");

        var maiorValor = textos.BuscaMaiorValorEmTextosComNumeros(
            (texto) =>
            {
                var numeros = new string(texto.ToCharArray()
                                               .Where(c => char.IsDigit(c))
                                               .ToArray());

                return string.IsNullOrEmpty(numeros)? 0 : int.Parse(numeros);
            },
            (lista) =>
            {
                lista.Sort();

                return lista.Last();
            });


        Console.WriteLine("Maior valor: " + maiorValor);
    }
}
```

Observe que criamos uma lista com alguns valores de texto e números dentro. Depois chamamos nosso método de extensão (que vimos anteriormente como funciona), passando duas funções anônimas como parâmetro.

A primeira função recebe uma string, que chamamos de “texto”, então converte ela para um Array de *char*, usando um método de extensão do próprio C#, o *ToCharArray*(). Em sequência, chama um outro método de extensão **muito** utilizado dentro do C#, o “**Where**”. Passando a ele outra lambda (função) que recebe um char e retorna um booleano (vamos abordar este método detalhadamente nos próximos temas desta aula). Com seu resultado, inicializamos uma nova String com os caracteres que possuem dígitos apenas (números apenas – tudo isso usando-se de métodos de extensão e Lambdas Function). Depois fazemos um “if ternário” para ver se dentro da string existe algum valor. Caso sim, convertemos para int. Caso não, retornamos 0.

A segunda função recebe uma lista de inteiros. Organiza-se ela utilizando-se do método “Sort” que todo Array possui. A partir de então, usa o método de extensão “Last” para obter o último valor da lista, ou seja, o maior valor encontrado.

1.2 Linq To Objects

O LINQ pode ser utilizado com vários “**providers**”, pois ele é uma especialização de linguagem de consulta e veremos que a forma como ele é executado permite que possamos aplicar **querys** (consultas) LINQ em basicamente qualquer fonte. A fonte mais comum e mais útil no seu dia a dia será a “*LINQ To Objects*” (LINQ para objetos). Isso significa que, na prática, você pode fazer consultas em listas de objetos, tais como você faria em um banco de dados, por exemplo. Isso porque o “provider” padrão do LINQ é o IEnumerable<T>. E como vimos anteriormente, essa interface é a interface base de todas as coleções do C# (Griffiths, 2019, p. 445).

Como vimos no tema 1, temos dois tipos de estrutura de consultas LINQ. Vamos abordar as duas neste tema. Iniciando pela *Fluent Syntax*. Mas antes vamos abordar os métodos de extensão LINQ mais úteis e utilizados (métodos LINQ também são conhecidos como “**Operadores LINQ**”) (Albahari, 2017, p. 354).

- *Where* – Recebe uma lambda function que tem como input o Type T da lista e que retorna um booleano. Retorna uma IEnumerable<T> (lista de objetos) em que a cláusula do Where seja satisfeita (verdadeira). Se assemelha muito com a clausula Where do banco de dados.
- *First* / *FirstOrDefault* – Recebe uma lambda function que tem como input o Type T da lista e que retorna um booleano. Mesmo retorno do método Where, mas nesse caso apenas retorna o primeiro elemento da sequência que satisfaça a condição. No caso do *First*, salvo nenhum elemento seja encontrado, uma exceção será lançada (pois é como tentar pegar o primeiro índice de um array vazio). E no caso do *FirstODefault,* o valor “**default**” do Type será retornado (no caso de objetos, o valor **null**).
- *Single* / *SingleOrDefefault* – Mesmo padrão do *First/FirstOrDefault*, com a diferença de que, além de retornar apenas o primeiro elemento, esse método irá lançar uma exceção, caso o resultado do seu filtro encontre mais de um elemento na coleção que satisfaça a clausula da Lambda.

- *Any* – Possui duas versões, com e sem uma função lambda. A versão sem parâmetro retorna booleano se a coleção possuir pelo menos um elemento e false se a coleção estiver vazia. É uma forma elegante de checar se a coleção possui dados. Troca-se o uso do *if(lista.Count > 0)* por: *if(lista.Any())*.
- *Max / Min* – Retorna o maior e menor valor, respectivamente, de uma coleção. É muito útil e fácil de usar em coleções numéricas, como List<int> ou IEnumerable<float>. Retorna 0 se a coleção estiver vazia.

No trecho de código a seguir você poderá observar alguns exemplos dos métodos citados anteriormente sendo utilizados em situações típicas:

```
var listaValores = new List<int>();

//popula a lista com alguns valores
for (int i = 0; i < 10; i++)
{
    listaValores.Add(i);
}

//Exemplo de filtro com o WHERE:
var listaFiltradaWhere = listaValores.Where(p => p > 5); //6,7,8,9

//Exemplo de First encontrando o elemento:
var elemento2 = listaValores.First(v => v == 2); // 2

//Exemplo de FirstOrDefault de um elemento não existente:
var elementoDefault = listaValores.FirstOrDefault(v => v > 10); // 0
(valor default do int)

//Exemplo do Any e Max
if (listaValores.Any())
{
    var maxValue = listaValores.Max(); // 9
}
```

TEMA 2 – FLUENT SYNTAX

Como vimos nos exemplos anteriores, as instruções LINQ podem receber *Lambda Functions* e realizar operações com elas para cada elemento da coleção, retornando uma nova coleção de objetos ou mesmo valores discretos (únicos) como resultado.

A *Fluent Syntax* é a mais flexível e fundamental. Nesta seção, descrevemos como usá-la para encadear operadores de consulta formando

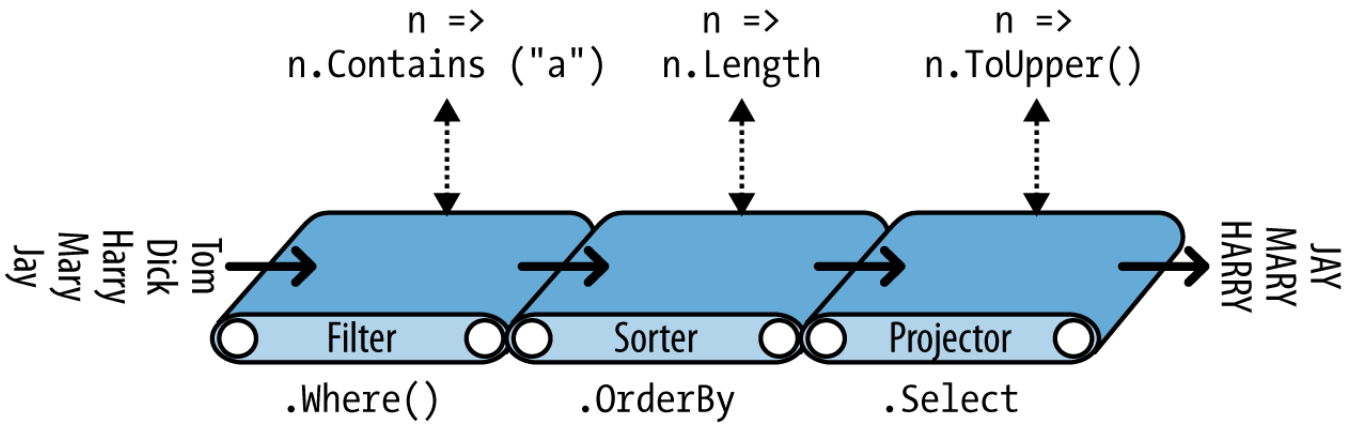
consultas mais complexas e mostrar por que os *Extensions Methods* são tão importantes para esse processo (Albahari, 2017, p. 354).

Na seção anterior, mostramos algumas consultas e filtros simples, cada uma compreendendo um único **Operador LINQ**. Para construir consultas mais complexas, você deve acrescentar operadores de consulta encadeados na expressão, criando uma sequência de operadores. Para ilustrar, a consulta a seguir extrai todas as strings contendo a letra "a", classifica-as por comprimento e, em seguida, converte os resultados para strings maiúsculas:

```
//Exemplo Query encadeada - Where + OrderBy + Select
string[] nomes = { "Tom", "Huck", "Harry", "Mary", "Jay" };

IEnumerable<string> query = nomes.Where(n => n.Contains("a"))
                                 .OrderBy(n => n.Length)
                                 .Select(n => n.ToUpper());

foreach (string nome in query)
{
    Console.WriteLine(nome);
}
```

Já introduzimos o operador *Where*, que emite uma versão filtrada da lista de entrada. O operador *OrderBy* emite uma versão classificada de sua lista de entrada; o método *Select*, por sua vez, emite uma sequência em que cada elemento de entrada é transformado ou projetado (*projection*) com uma determinada expressão lambda (n.ToUpper (), neste caso). Os dados fluem da esquerda para a direita por meio da cadeia de operadores, então os dados são primeiro filtrados, depois classificados e, em seguida, projetados (Albahari, 2017, p. 354). “Um operador de consulta nunca altera a sequência de entrada; ao invés disso retorna uma nova sequência (coleção). Isso é consistente com o paradigma funcional de programação, a partir do qual o LINQ foi inspirado” (Albahari, 2017, p. 355).

Quando os operadores de consulta são encadeados como no exemplo a seguir, a sequência de saída de um operador é a sequência de entrada do próximo. A consulta completa se assemelha a uma esteira de linha de produção (pipeline) (Albahari, 2017, p. 355):



Fonte: Albahari, 2017, p. 355.

Podemos construir a mesma query usando uma forma “progressiva”, fazendo o seguinte:

```
string[] names = { "Tom", "Huck", "Harry", "Mary", "Jay" };

IEnumerable<string> query = names.Where(n => n.Contains("a"))
                                 .OrderBy(n => n.Length)
                                 .Select(n => n.ToUpper());

IEnumerable<string> filtered = names.Where(n => n.Contains("a"));
IEnumerable<string> sorted = filtered.OrderBy(n => n.Length);
IEnumerable<string> finalQuery = sorted.Select(n => n.ToUpper());
```

A variável “finalQuery” conterá os mesmos elementos que a variável “query”. Ambas são equivalentes e, se enumeradas, produzirão o mesmo resultado.

Os operadores de input e output, comuns parâmetros Generics em querys, no LINQ serão (Albahari, 2017, p. 357):

- *TSoruce* – type de origem do elemento de input;
- *TResult* – type de retorno (output) da função (utilizando apenas para quando for diferente do Tsoruce); e
- *TKey* – type da chave de agrupamento (quando houver) para operações de *sort, group ou join.*

O operador LINQ “**Select**” é um tipo de operador especial que realiza operações “***map & reduce***” no C#. Esse tipo de operação é comum em linguagens como Phyton e Javascript, mas não é tão “natural” em linguagens tipadas e orientadas a objeto. Por exemplo: a expressão *Func <TSource, TResult>* corresponde a uma expressão lambda **TSource** => **TResult**, ou seja, aquela que mapeia um elemento de entrada para um elemento de saída.

TSource e TResult podem ser diferentes tipos, então uma expressão lambda pode alterar o tipo de cada elemento. Além disso, a expressão lambda determina o Type de saída. A seguinte consulta usa o operador "*Select*" para transformar

```
//Exemplo de operador Select
string[] nomes = { "Tom", "Huck", "Harry", "Mary", "Jay" };

//Utilizando o select para fazer uma Projeção (modificando o tipo de retor
no da lista)
//de uma lista de string, transformamos em uma lista de inteiros
//onde cada elemento representa o comprimento da string de seu indice
IEnumerable<int> queryResultSelect = nomes.Select(n => n.Length);
foreach (int length in queryResultSelect)
{
    Console.Write(length + "|"); //Output no console:  3|4|5|4|3|
}
```

elementos do tipo string em elementos do tipo inteiro durante uma "**projeção**" LINQ (veremos mais sobre ela no próximo tema) (Albahari, 2017, p. 357):

O compilador pode inferir o tipo do TResult a partir do valor de retorno da expressão lambda. Neste caso, "n.Length" retorna um valor int, então TResult é inferido como int.

O operador de consulta *Where* é mais simples e não requer inferência do Type de saída, uma vez que os elementos de entrada e saída são do mesmo Type. Isso faz sentido porque o operador apenas filtra elementos e não os transforma (Albahari, 2017, p. 358).

A ordem natural dos elementos importa. A ordem original dos elementos em uma sequência de entrada é significativa no LINQ. Alguns operadores de consulta dependem dessa ordem, como *Take*, *Skip* e *Reverse*. O operador *Take*, por exemplo, retorna os "N" primeiros elementos de uma coleção (Albahari, 2017, p. 359):

```
//Exemplo Take, Skip e Reverse
int[] numerosExemplo = { 10, 9, 8, 7, 6 };
IEnumerable<int> primeirosTres = numerosExemplo.Take(3); // { 10, 9, 8 }
```

O operador *Skip* ignora/pula os "N" primeiros elementos e retorna os demais (enumerando a coleção em ordem):

```
IEnumerable<int> ultimosDois = numerosExemplo.Skip(3); // { 7, 6 }
```

O operador *Reverse* inverte a ordem dos elementos da lista em seu output (nunca modificando a própria lista de input, isso é uma premissa de imutabilidade do LINQ):

```
IEnumerable<int> ultimosDois = numerosExemplo.Skip(3); // { 7, 6 }
```

TEMA 3 – QUERY EXPRESSION

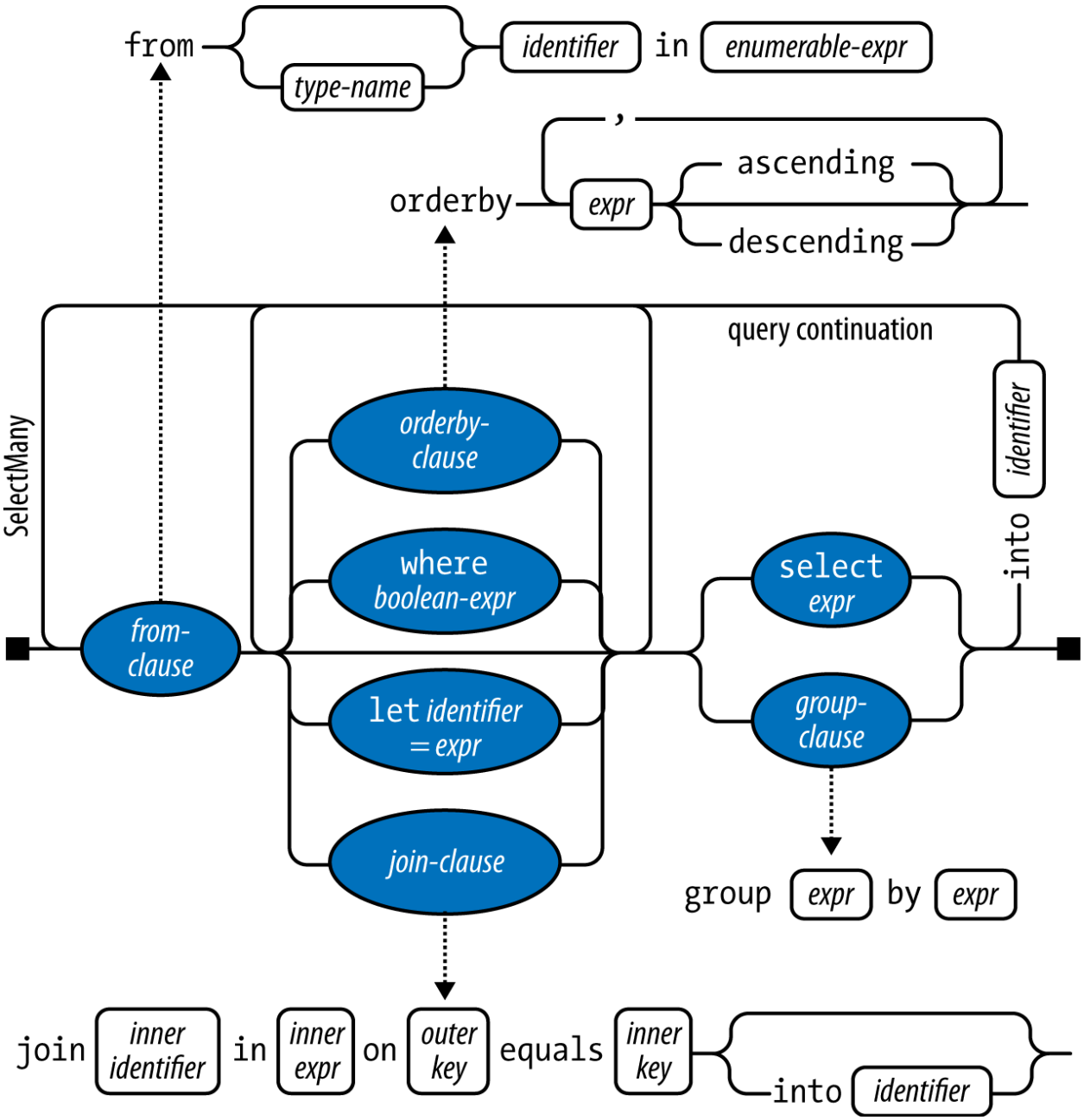
O C# também prove uma outra sintaxe para escrita de *querys* LINQ, chamada de "**Query Expression**". Essa sintaxe, ao contrário do que muita gente imagina à primeira vista, não é como escrever SQL no C#. Na verdade, seu design original se inspira em linguagens funcionais como o LISP e o Haskell, em um conceito chamado de "*list comprehesions*", buscando pouca influência de nomes do SQL para facilitar a migração dos desenvolvedores .NET para esse estilo de escrita de consultas. Como veremos em outras ocasiões, as consultas LINQ podem ser feitas "direto no banco de dados" (passando pelos devidos *providers*) (Albahari, 2017, p. 360).

No tema anterior, escrevemos uma consulta em *Fluent Syntax* para extrair strings contendo a letra "a", classificá-las por comprimento e convertê-las em maiúsculas. A seguir, podemos observar um exemplo com o mesmo resultado, usando a sintaxe "*Query Expression*":

```
string[] nomes = { "Tom", "Huck", "Harry", "Mary", "Jay" };
IEnumerable<string> query = from n in nomes
                            where n.Contains("a") // Filtra os elementos
                            orderby n.Length // Ordena pelo tamanho
                            select n.ToUpper(); // Converte para string
ToUpper (projeção)

foreach (string nome in query)
{
    Console.WriteLine(nome);
}
```

As consultas utilizando LINQ "Query Expressions" sempre começam com uma cláusula **from** e terminam com um **select** ou cláusula de **group by**. A cláusula **from** declara uma variável de intervalo (neste caso o **n**), será o nome da variável utilizada na "enumeração" - **como no foreach** – [***foreach(var n in nomes)***] . A imagem a seguir representa o pipeline (fluxo) que a CLR utiliza para processar um Query Expression.

Deve-se observar que a leitura da imagem a seguir deve começar pela esquerda e depois seguir para os elementos da direita, seguindo a sequência como um trilho de trem. Por exemplo, após a cláusula **from** obrigatória, você pode incluir opcionalmente um *orderby*, *where*, *let* ou *join*. Depois disso, você pode continuar com uma cláusula *select* ou *groupby*, ou voltar e incluir outra cláusula *from*, *orderby*, *where*, *let* ou *join* e assim sucessivamente (Albahari, 2017, p. 361):



Fonte: Albahari, 2017, p. 361.

Em nossa query usando a sintaxe “Query Expression”, os operadores *Where*, *OrderBy* e *Select* são resolvidos pela CLR usando as mesmas regras que uma consulta escrita em “*Fluent Syntax*”. Neste caso, esses operadores se ligam a métodos de extensão na classe *Enumerable* do namespace System.Linq.

3.1 Variáveis de intervalo (*Range Variables*)

O identificador imediatamente após a palavra-chave **from** é chamado de: variável de intervalo (*range variable*). Uma variável de intervalo se refere ao

```
from n in nomes // n é nossa variável de intervalo
where n.Contains("a") // n = elemento direto do array
orderby n.Length // n = elemento já filtrado pelo where
select n.ToUpper() // n = elemento já ordenado pelo orderby
```

elemento atual na coleção em que a operação/enumeração será realizada. Em nossos exemplos, a variável de intervalo **n** aparece em todas as cláusulas da consulta. E, de fato, a variável **representa um objeto diferente** para cada enumeração realizada nos operadores/segmentos da query:

Isso fica claro quando olhamos para forma como a query pode ser escrita usando *Fluent Syntax* e como o compilador entende cada "**variável n**" no seu escopo (observe como cada variável **n** pertence ao seu escopo local de *Lambda Function*):

Logo, escrever as Querys LINQ em *Fluent Syntax* ou *Query Expression* não faz nenhuma diferença da performance ou execução. O compilador irá converter toda e qualquer *Query Expression* para *Fleunt Syntax* em momento de

```
nomes.Where(n => n.Contains("a")) // variável de escopo local n
     .OrderBy(n => n.Length) // variável de escopo local n
     .Select(n => n.ToUpper()) // variável de escopo local n
```

compilação (*build*), e não em momento de execução.

Existem, como vimos no tema 1, algumas vantagens e desvantagens de cada sintaxe na facilidade de escrever as Querys LINQ para determinados cenários. De forma geral, é mais comum e prático utilizar a *Fluent Syntax* para Querys simples, sem muitos *groupbys*, *joins* e projeções complexas (usando *Select* e *Let*). Do contrário, a sintaxe Query Expression pode ser uma boa saída para tornar o código mais legível e fácil de dar manutenção.

3.2 Misturando Query Expression com Fluent Syntax

Existem muitos operadores que não possuem palavras-chave na sintaxe *Query Expression*. Esses exigem que você use a *Fluent Syntax,* ou pelo menos em parte. No caso, qualquer operador que não esteja dentro dos abaixo, lhe obrigará a implementar uma *Lambda Function* à parte ou recorrer a um mix entre *Fluent* e *Query Syntax*:

- Where;
- Select;
- SelectMany;
- OrderBy;
- ThenBy;
- OrderByDescending;
- ThenByDescending;
- GroupBy;
- Join; e
- GroupJoin.

Se um operador da sintaxe *Query Expressions* não possuir suporte para *Fluent Syntax*, você pode misturar as sintaxes em uma única query. **A única restrição é que cada componente de sintaxe de consulta deve ser completo**. Por exemplo, se você começar com uma cláusula **from**, deve terminar com uma cláusula **select** ou **group** (Albahari, 2017, p. 363).

No trecho de código exemplo a seguir, observe as misturas entre os diferentes tipos de operadores das duas sintaxes:

```
//Mix -> Fluent e Query Expression
string[] nomes = { "Tom", "Dick", "Harry", "Mary", "Jay" };

var quantidadeLetraA = (from n in nomes
                        where n.Contains("a")
                        select n).Count(); // 3
var primeiroElementoDescrescente = (from n in nomes
                                    orderby n descending
                                    select n).First(); // Tom

Console.WriteLine(quantidadeLetraA);
Console.WriteLine(primeiroElementoDescrescente);

// Query mais complexa com mix entre as sintaxes, subqueries e projection
var mixQueries = (from n in nomes
                  join nomeY in nomes.Where(y => y.Contains("y")) on n
equals nomeY
                  where n.Count() > 3
                  select n.ToUpper())
                  .OrderByDescending(x=> x); // MARY, HARRY
foreach (var item in mixQueries)
{
    Console.WriteLine(item); // MARY, HARRY
}
```

TEMA 4 – EXECUÇÃO TARDIA E CONCEITOS LINQ

Uma das funcionalidades mais importantes dos operadores de query LINQ é sua capacidade de execução tardia (***deferred execution/lazy execution***). Na prática, queremos dizer que um operador comum é executado apenas quando a query é enumerada, e não quando ela é construída. Em um primeiro momento, esse fato pode não parecer tão simples de ser compreendido, por isso vamos detalhar bem essa importante funcionalidade do LINQ neste tema.

Considere o exemplo de código a seguir (Albahari, 2017, p. 364):

```
//lista de números com um único elemento
var numeros = new List<int> { 1 };

//construímos uma query LINQ com o operador de projeção SELECT
//observe que estamos multiplicando cada elemento da coleção por 10
var query = numeros.Select(n => n * 10); // construção da query

numeros.Add(2); // Adicionamos 1 elemento a mais na coleção

foreach (int n in query)
{
    Console.Write(n + "|"); //Output no console: 10|20|
}
```

O número extra que adicionamos na lista após construir a consulta é incluído no resultado, quando a query é enumerada. Mesmo que houvesse um filtro *Where* ou outro operador de projeção/filtro, essas instruções de query não são executadas quando construídas, e sim quando a query é enumerada. Isso é chamado de execução tardia (***deferred execution*** ou ***lazy execution***) e é o mesmo que acontece com os tipos "*Delegate*", como o caso das "*Actions"* e "*Func"*, que já vimos no tema 1. Observe o exemplo de *Lambda Function* a seguir:

```
//Exemplo 2:
Action a = () => Console.WriteLine("Teste");

// Até esta linha nada foi executado ou escrito no console ainda!
//apenas quando executarmos a Action!
a(); // Execução tardia (Deferred execution)!
```

Quase todos os operadores de query LINQ (Where, GroupBy, Select etc.) fornecem essa funcionalidade de execução tardia, com exceção de dois grupos de operadores (Albahari, 2017, p. 365):

- operadores que retornam um único elemento ou um valor discreto, como "First", "Last" e "Count"; e

- os operadores de conversão de tipos de coleção – “*ToArray*”, “*ToList*”, “*ToDictionary*”, “*ToLookup*”.

Estes operadores citados são executados imediatamente, sem o mecanismo de “*deferred execution*”, pois esses operadores causam a enumeração das coleções. E, dessa forma, eles não podem suportar o mecanismo de execução tardia. O *Count*, por exemplo, realiza uma contagem de elementos na lista. Essa operação ocorre no mesmo momento em que ela é construída, não suportando a execução tardia.

A execução tardia é importante porque separa a construção da execução de uma query LINQ. Isso permite que você construa uma query em várias etapas, além de fazer possíveis consultas de banco de dados (veremos isso em outras ocasiões, quando vamos trabalhar com outros *providers* para o LINQ) (Albahari, 2017, p. 366).

4.1 Reavaliação

A execução tardia traz outras consequências interessantes, como quando você reavalia ou reexecuta uma query LINQ. Observe no exemplo a seguir que vamos construir uma lista com dois números, criar uma query de projeção, na qual a lambda de projeção multiplica os elementos por 10. Ao enumerarmos, visualizamos o resultado aplicado aos dois elementos. Mas após limparmos a lista, se executarmos novamente a query LINQ **veremos que não há nenhum output**:

```
//Exemplo 3:
var numeros2 = new List<int>() { 1, 2 };
var queryExemplo2 = numeros2.Select(n => n * 10);

foreach (int n in query)
{
    Console.Write(n + "|"); //Output no console: 10|20|
}

//Limpamos a lista!
numeros2.Clear();

foreach (int n in query)
{
    Console.Write(n + "|"); //Output no console: <nothing>
}
Console.Write(query.Count()); //Output no console: 0
```

Esse comportamento nem sempre é desejado. Seja porque a query pode ser computacionalmente intensiva ou porque queremos “congelar” os resultados

em “certo ponto no tempo”. Para “**contornar**” a execução tardia, chamando os **operadores de conversão**, como *ToList* ou *ToArray*. Esses operadores vão enumerar nossa query (executar ela) e copiar os elementos para uma nova lista (ou array) de saída (Griffiths, 2019, p. 456).

Podemos observar isso no exemplo a seguir:

```
//Exemplo 4:
var numeros3 = new List<int>() { 1, 2 };

// o ToList força a enumeração e executa a query imediatamente
var listaMultiplicadaPor10 = numeros3.Select(n => n * 10).ToList();

// limpamos a lista
numeros3.Clear();

//Observe que a listaMultiplicadaPor10 continua com 2 elementos
Console.WriteLine(listaMultiplicadaPor10.Count); // 2

foreach (int n in listaMultiplicadaPor10)
{
    Console.Write(n + "|"); //Output no console: 10|20|
}
```

4.2 Variáveis de contexto

Um dos aspectos mais interessantes do LINQ e das Lambda Functions é sua capacidade de compartilhar determinados elementos do contexto global com seu contexto local de execução. Isso significa que uma Lambda pode utilizar uma variável declarada previamente em um método ou classe dentro de si. E o mais interessante, a execução tardia irá utilizar-se desse valor somente no momento da enumeração. Observe o exemplo:

```
int[] numeros = { 1, 2 };
int fator = 10;

//construímos a query, onde factor nesse momento
//esta com o valor de 10
var query = numeros.Select(n => n * fator);

//alteramos o valor do factor para 20
fator = 20;

//Enumeramos a query
foreach (int n in query)
{
    Console.Write(n + "|"); //Output no console: 20|
}
```

Quando bem utilizado, variáveis de contexto podem trazer um poder extra para suas expressões e querys Lambda. Porém, caso o desenvolvedor não entenda o que está construindo e ignore o mecanismo de execução tardia ou

não saiba quais operadores fazem uso desse mecanismo, **pode acabar construindo um código defeituoso e certamente causará problemas (um bug).**

Veja o exemplo a seguir e observe um bug não tão simples de ser notado em uma leitura rápida de código. Esse trecho só irá gerar uma ***exception*** em momento de execução, o que significa que o compilador não irá apontar erros durante o processo de build. O objetivo desse código a seguir seria: remover do texto os parênteses e todo e qualquer caractere numeral (de 0-9). Tente identificar onde está o problema:

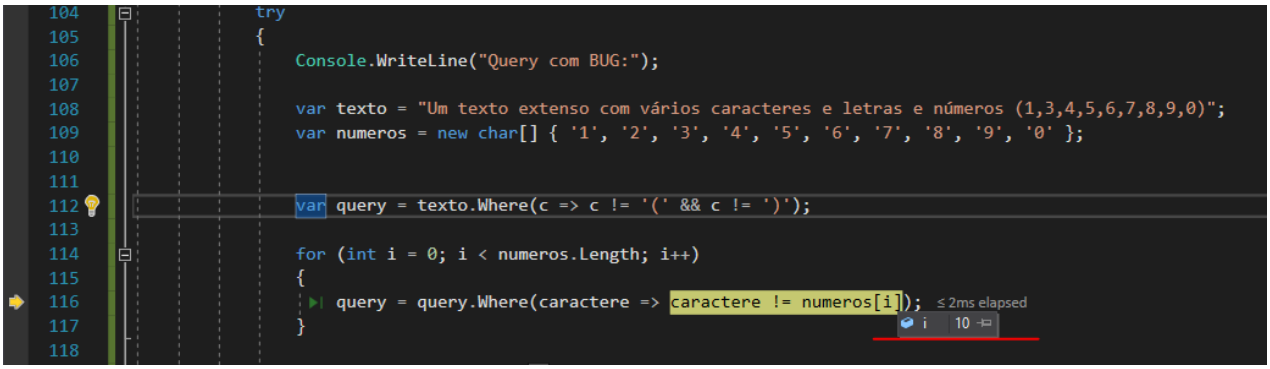
```
try
{
    Console.WriteLine("Query com BUG:");

    var texto = "Um texto extenso com vários caracteres e letras e números
(1,3,4,5,6,7,8,9,0)";
    var numeros = new char[] { '1', '2', '3', '4', '5', '6', '7', '8', '9',
 '0' };


    var query = texto.Where(c => c != '(' && c != ')');

    for (int i = 0; i < numeros.Length; i++)
    {
        query = query.Where(caractere => caractere != numeros[i]);
    }

    foreach (var caractere in query)
    {
        Console.Write(caractere);
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}
```

Esse código utiliza uma query para inicialmente filtrar os parênteses “()” do texto. Depois, possui um for iterando pelo array de números. Observe que o array possui 10 elementos. Após entrar no for, é feito um processo de filtragem em que somente ficamos com o “resultado” da lista onde não possuímos o dígito numérico do índice “i”, removendo esse caractere da query. Depois deste loop, é feita a enumeração da query.

O bug irá ocorrer somente em momento de execução. Pois quando ela for enumerada, percorrendo seus elementos, para cada elemento, o C# irá executar a lambda que está dentro do for. Por conta da execução tardia. Como já percorremos esse for, o “i” está com o valor 10. E o array “números” possui

apenas 10 elementos (como o array inicia no índice 0, o índice do último elemento é 9).

Logo, o C# irá lançar uma exceção:



Crédito: Araya (2021).

Para corrigir esse comportamento, devemos tomar o cuidado de salvar a variável correspondente ao caractere numeral em uma variável de escopo local primeiro. Veja o exemplo a seguir e observe que a única diferença é que

```
Console.WriteLine("Query com Sem Bug:");

var texto = "Um texto extenso com vários caracteres e letras e números (1,3,4,5,6
,7,8,9,0)";
var numeros = new char[] { '1', '2', '3', '4', '5', '6', '7', '8', '9', '0' };

var query = texto.Where(c => c != '(' && c != ')');

for (int i = 0; i < numeros.Length; i++)
{
    char digito = numeros[i];
    query = query.Where(caractere => caractere != digito);
}

foreach (var caractere in query)
{
    Console.Write(caractere);
}
```

salvamos o valor do “números[i]” em uma variável antes de executar a query:

Essa pequena alteração fará com que a Lambda Function executada pelo Where grave o “dígito” dentro de seu contexto. E preserve-o para a enumeração tardia. É claro que haveria formas mais simples e corretas de filtrar números de um texto, mas essa situação de exemplo mostra o poder e, ao mesmo tempo, o cuidado que se deve ter com querys LINQ, reavaliação das querys e variáveis de contexto ao mesmo tempo.

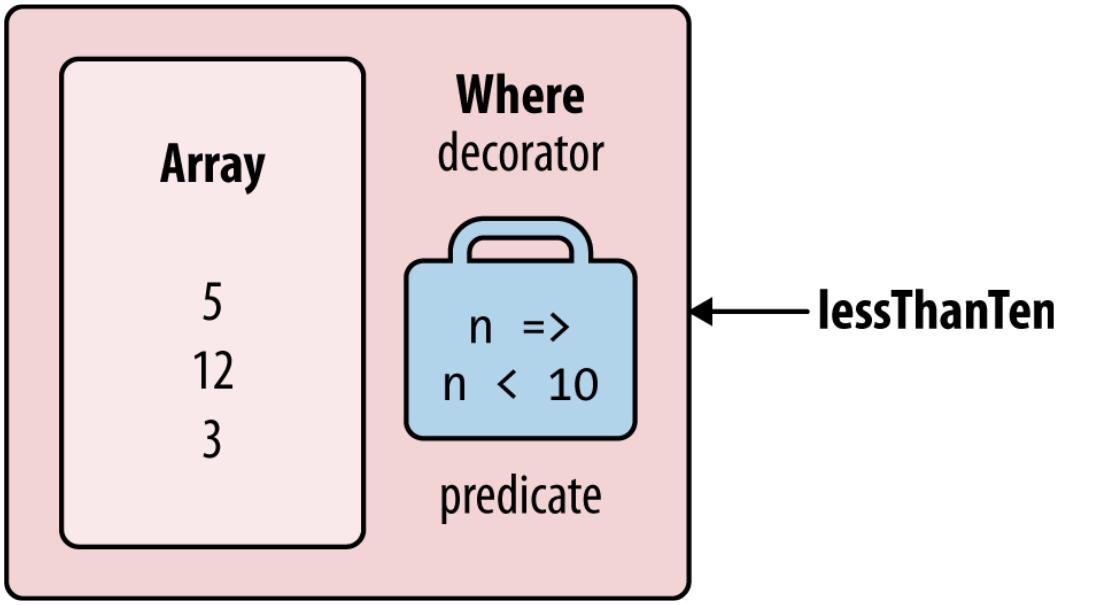
4.3 Como a execução tardia funciona?

A execução tardia funciona dentro do C# criando o que chamamos de "***decorator sequences***". Isso significa que quando uma query LINQ é criada e não foi enumerada ainda, ela gera um elemento "bolha" que a encapsula (Griffiths, 2019, p. 456).

Ao contrário de uma classe de coleção tradicional, como um array ou lista, um "**decorator**" (em geral) não tem estrutura de suporte própria para armazenar elementos. Em vez disso, ele envolve sua coleção em uma "bolha", além das variáveis que você fornece de input para a Lambda e que venham de um contexto externo. Sempre que você solicitar dados de um "**decorator**", ele, por sua vez, deve solicitar os dados à coleção que ele armazena (Albahari, 2017, p. 385).

A imagem a seguir ilustra como é armazenada a estrutura do "decorator" para a query LINQ:

```
IEnumerable<int> lessThanTen = new int[] { 5, 12, 3 }.Where(n => n < 10);
```

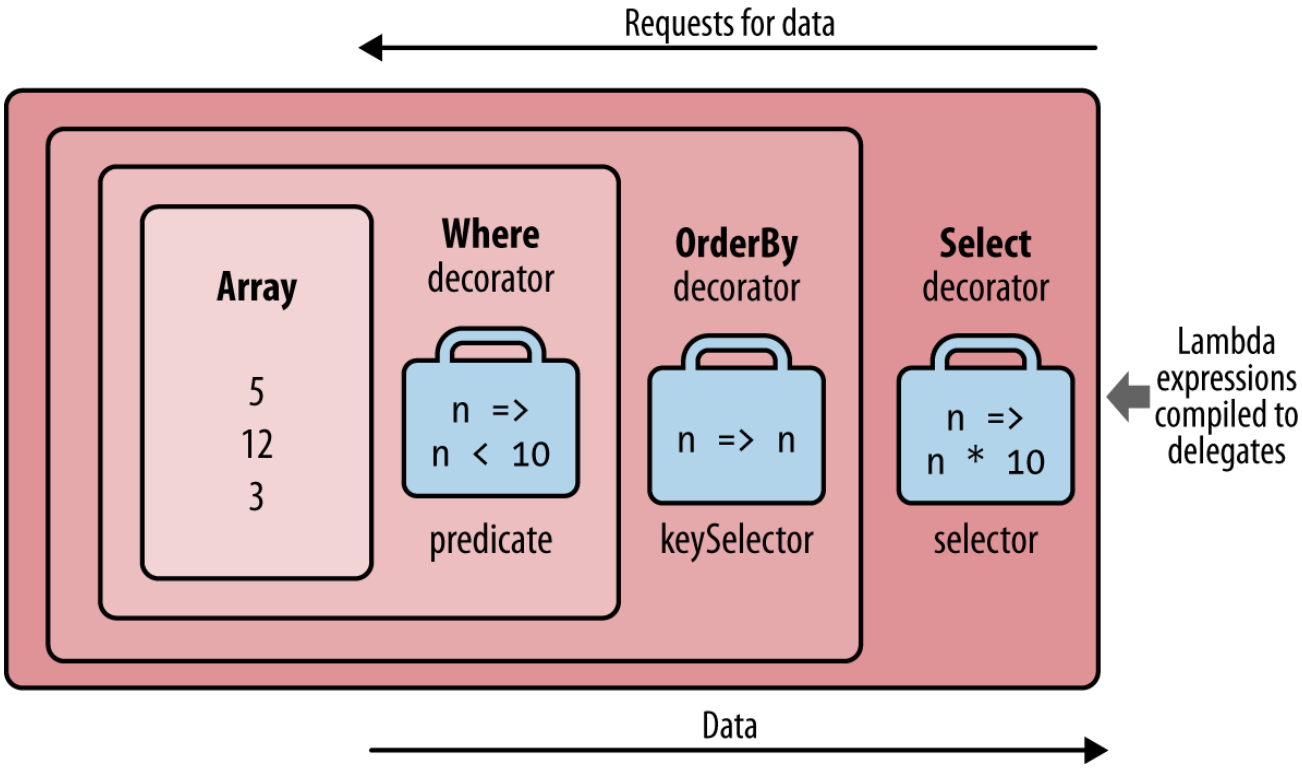


Fonte: Albahari, 2017, p. 385.

Quando encadeamos várias querys LINQ uma seguida da outra, estamos criando "bolhas" umas sobre as outras, em que existe uma ordem de execução que será da primeira para a última. Desconstruindo a estrutura de "decorators" criados.

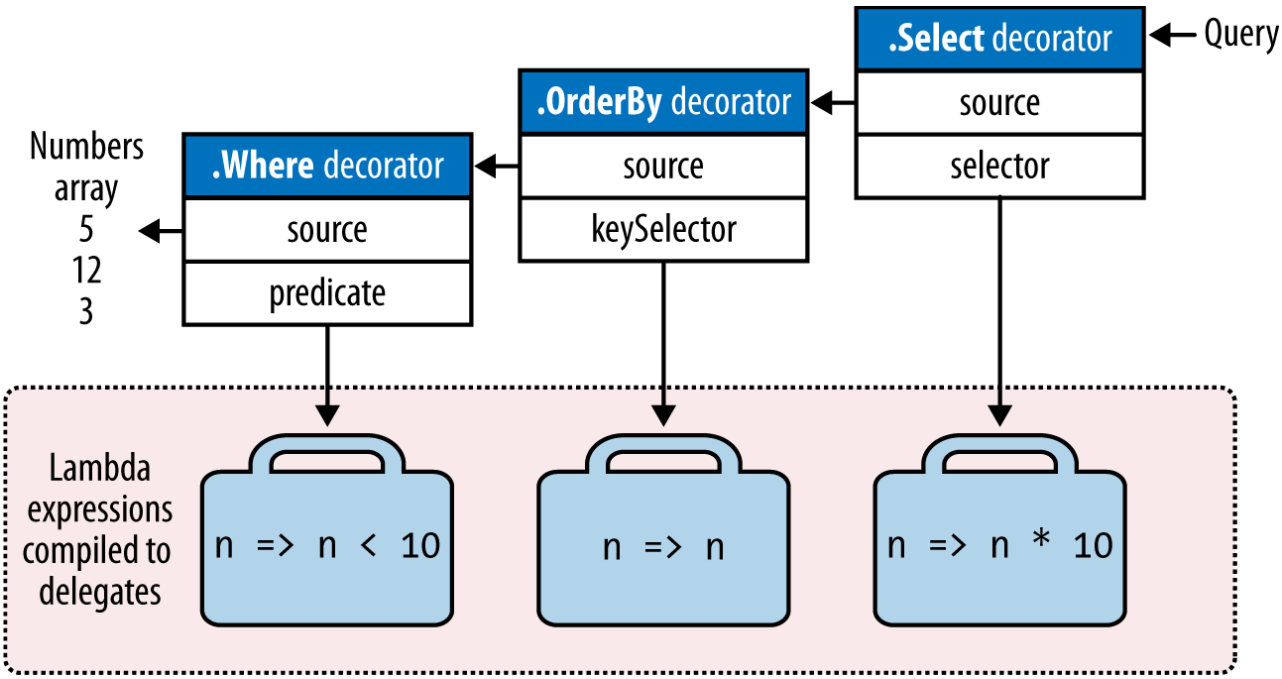
Veja o exemplo a seguir no qual encadeamos alguns operadores de query LINQ (Albahari, 2017, p. 386):

```
IEnumerable<int> query = new int[] { 5, 12, 3 }.Where(n => n < 10)
                                               .OrderBy(n => n)
                                               .Select(n => n * 10);
```



Quando você enumera uma query LINQ, está consultando o array/lista original, transformado por meio de uma camada ou cadeia de decoradores (*decorators*).

A imagem a seguir mostra a mesma composição de objeto na sintaxe UML. O *decorator* Select faz referência ao *decorator* OrderBy, que faz referência ao *decorator* Where, que faz referência ao Array original (Albahari, 2017, p. 387):



Fonte: Albahari, 2017, p. 387

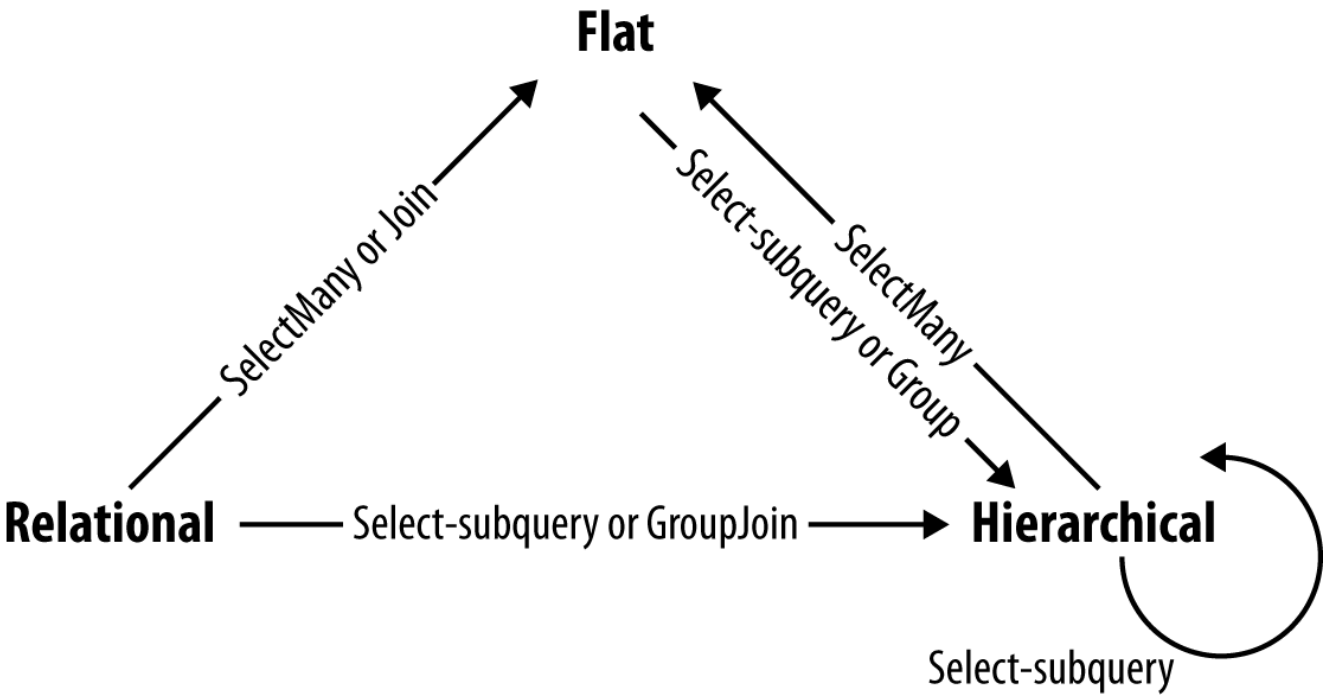
TEMA 5 – OPERADORES LINQ

Neste tema, iremos explorar um pouco mais dos operadores de query LINQ e quando os utilizar, servindo também de um pequeno guia de referência para operadores comuns.

Os operadores podem ser agrupados em três principais categorias (Albahari, 2017, p. 427):

- **“coleção in”, “coleção out”** (operadores que recebem uma coleção de input e devolvem outra coleção de output);
- “**coleção in”, “elemento out”** (operadores que recebem uma coleção de input e devolvem um único elemento ou valor discreto de output); e
- **sem parâmetros de entrada, “coleção out”** (operadores que não recebem parâmetros de input e devolvem uma coleção de output).

O tipo mais comum de operadores para as querys LINQ é o primeiro, em que o operador (método) recebe uma coleção de input e retorna uma coleção de output.



Fonte: Albahari, 2017, p. 427.

A figura mostrada ilustra bem a situação destes tipos de operadores. Onde eles podem realizar projeções usando *Select, Join, Group* e outras para criar estruturas relacionais ou hierárquicas. Por exemplo, o agrupamento de objetos irá gerar uma estrutura similar a uma árvore. Onde existem chaves que agrupam listas (coleções) de elementos. Onde essas coleções podem ter outras N coleções vinculadas.

5.1 Filtros

IEnumerable<TSource> -> IEnumerable<TSource>

Os operadores a seguir representam as possibilidades de filtro existentes no LINQ:

Método	Descrição	Equivalência em SQL
Where	Retorna um subset (subcoleção) de elementos que satisfazem uma determinada condição	WHERE
Take	Retorna os primeiros N elementos e descarta o restante	WHERE ROW_NUMBER() or TOP n subquery
Skip	Ignora os primeiros N elementos e retorna o restante	WHERE ROW_NUMBER()... or NOT IN (SELECT TOP n...)
TakeWhile	Retorna elementos até que a sua função condicional retorne falso	N/A
SkipWhile	Ignora os primeiros elementos até que a condição retorne falso, a partir de então enumera e retorna o restante dos elementos	N/A
Distinct	Retorna uma coleção excluindo elementos duplicados	SELECT DISTINCT ...

Fonte: Albahari, 2017, p. 430. Traduzida e adaptada.

Os operadores/métodos de filtro recebem um Type "*TSource*" como input e retornam o mesmo Type como output. Além disso, uma característica deste tipo de operador é que eles sempre vão lhe retornar uma coleção com menos elementos ou igual à coleção original, nunca com mais elementos.

5.2 Projeção

IEnumerable<TSource> -> IEnumerable<TResult>

Os operadores a seguir representam as possibilidades de projeção existentes no LINQ:

Método	Descrição	Equivalência em SQL
Select	Transforma cada elemento de entrada em outro elemento de saída utilizando uma lambda function	SELECT
SelectMany	Transforma cada elemento de entrada em outro elemento de saída utilizando uma lambda function. Além disso, concatena os elementos e uma única coleção de saída, utilizando uma técnica chamada de "flattern". O resultado é similar ao de uma query SQL com JOIN, onde unifica-se o resultado de 2 tabelas em única tabela de saída. Esta, podendo conter duplicatas causadas pelo "flattern"	INNER JOIN, LEFT OUTER JOIN, CROSS JOIN

Fonte: Albahari, 2017, p. 434.

Com o *Select*, sempre é obtido o mesmo número de elementos da coleção inicial. Cada elemento, entretanto, pode ser transformado dentro da função lambda. No exemplo de código a seguir usamos o poder de “projeção” do select para selecionar todas as línguas/culturas da máquina e listá-las. Observe que estamos criando um tipo anônimo no retorno, convertendo o objeto *culture* em outro objeto novo declarado em “tempo de execução”:

```
var queryCulturas = CultureInfo.GetCultures(CultureTypes.AllCultures).Select
(culture => new
{
    Nome = culture.DisplayName,
    NomeTecnico = culture.Name,
    NomeEmIngles = culture.EnglishName,
    HashCode = culture.GetHashCode()
});

foreach (var cultura in queryCulturas)
{
    Console.WriteLine($"Nome: {cultura.Nome} - Cultura:
        {cultura.NomeTecnico} - HashCode: {cultura.HashCode}");
}
```

5.3 Combinação (Join)

Os operadores a seguir representam as possibilidades de Join entre coleções existentes no LINQ:

Método	Descrição	Equivalência em SQL
Join	Aplica uma estratégica chamada de "Lookup" para dar match em elementos das duas coleções, produzindo como output uma coleção "plana" (flat).	INNER JOIN
GroupJoin	Similar ao Join, com a diferença que produz um output de coleção hierárquica.	INNER JOIN, LEFT OUTER JOIN
Zip	Enumera duas coleções simultaneamente, aplicando uma técnica conhecida como "Zipper" (por lembrar um zíper de roupa). Aplicando uma Lambda Function que combina os elementos das coleções. Produz um retorno utilizando tuplas, onde cada tupla é a combinação do elemento da primeira coleção com o elemento da segunda coleção, conforme a Lambda fornecida.	N/A

Fonte: Albahari, 2017, p. 446.

Os operadores/métodos de Join do Linq podem ser muito úteis para combinar resultados de coleções diferentes. Você deve usá-los quando possuir situações em que o resultado que precisa ser obtido deve ser fruto da combinação entre coleções. Similar à necessidade do uso da expressão "Join" na linguagem SQL para banco de dados.

Segundo Albahari, a regra de sintaxe para Query Expression é a seguinte:

"**from** outer-var **in** outer-enumerable

join inner-var **in** inner-enumerable **on** outer-key-expr **equals** inner-key-expr

[into identifier]”

Considere o seguinte exemplo de classes:

```
public class Cliente
{
    public int Id { get; set; }
    public string Nome { get; set; }
}

public class Pedido
{
    public int Id { get; set; }
    public int IdClinte { get; set; }
    public DateTime DataPedido { get; set; }
}
```

Agora, vamos popular duas listas com clientes e pedidos e depois realizar um Join entre clientes e pedidos e escrever o resultado no console:

```
var clientes = new List<Cliente>() { new Cliente() { Id = 1, Nome = "Tiago"
}, new Cliente() { Id = 2, Nome = "Pedro" }, new Cliente() { Id = 3, Nome =
"Maria" }, };
var pedidos = new List<Pedido>(100);


int idClienteAuxiliar = 0;
//gera 100 pedidos distribuindo para os 3 clientes
for (int i = 0; i < 100; i++)
{
    idClienteAuxiliar++; //incrementa o auxiliar para o prox. cliente

    var pedido = new Pedido() { Id = i, IdClinte = idClienteAuxiliar, DataPe
dido = DateTime.Now.AddDays(i * -1) };
    if (idClienteAuxiliar >= 3)
        idClienteAuxiliar = 0;

    pedidos.Add(pedido); //adiciona o pedido na lista
}

//produz um resultado usando LINQ unificando as duas
//coleções de clientes e pedidos:
var queryJoin = from p in pedidos
                join c in clientes on p.IdClinte equals c.Id
                select new
                {
                    IdPedido = p.Id,
                    IdCliente = c.Id,
                    NomeCliente = c.Nome,
                    DataPedido = p.DataPedido
                };

//enumera resultado
foreach (var item in queryJoin)
{
    Console.WriteLine($"IdPedido: {item.IdPedido} | IdCliente: {item.IdClien
te} | NomeCliente: {item.NomeCliente} | DataPedido: {item.DataPedido}");
}
```

O resultado será parecido com:

```
//Output no console:
//IdPedido: 0 | IdCliente: 1 | NomeCliente: Tiago | DataPedido: 16 / 05 / 2021
//IdPedido: 1 | IdCliente: 2 | NomeCliente: Pedro | DataPedido: 15 / 05 / 2021
//IdPedido: 2 | IdCliente: 3 | NomeCliente: Maria | DataPedido: 14 / 05 / 2021
//IdPedido: 3 | IdCliente: 1 | NomeCliente: Tiago | DataPedido: 13 / 05 / 2021
//IdPedido: 4 | IdCliente: 2 | NomeCliente: Pedro | DataPedido: 12 / 05 / 2021
//IdPedido: 5 | IdCliente: 3 | NomeCliente: Maria | DataPedido: 11 / 05 / 2021
//IdPedido: 6 | IdCliente: 1 | NomeCliente: Tiago | DataPedido: 10 / 05 / 2021
//....
```

5.4 Ordenação

IEnumerable<TSource> -> IOrderedEnumerable<TSource>

Os operadores a seguir representam as possibilidades de operadores para ordenação existente no LINQ:

Método	Descrição	Equivalência em SQL
OrderBy, ThenBy	Ordena a coleção em forma descendente	ORDER BY
OrderByDescending, ThenByDescending	Ordena a coleção em forma descendente	ORDER BY ... DESC
Reverse	Retorna uma nova coleção com seus elementos em ordem reversa	N/A

Fonte: Albahari, 2017, p. 436. Traduzida e adaptada.

Operadores de ordenação retornam sempre os mesmos elementos, porém, em ordens diferentes.

Exemplos:

```
string[] nomes = { "João", "Silva", "Paulo", "Antonio", "Maria", "Joana" };

//exemplo de ordenação ascendente
var listaAsc = nomes.OrderBy(a=> a).ToList();

//exemplo de ordenação descendente
var listaDesc = nomes.OrderByDescending(a=> a).ToList();

//exemplo de ordenação pelo tamanho da string e depois por ordem alfabética
var ordenacao2 = nomes.OrderBy(a => a.Length).ThenBy(l => l);
```

5.5 Conversão

Os métodos de conversão LINQ podem ser aplicados a todos os tipos de coleção, pois todas herdam sumariamente de IEnumerable<T>. Existem alguns operadores que podem modificar o Type de “destino” (target) das coleções de origem:

Método	Descrição
OfType	Enumera a coleção, para cada membro de origem Tsoruce, convertendo-os usando CAST para o Type de destinho TResult. Descarta o elemento quando o Cast falha
Cast	Enumera a coleção, para cada membro de origem Tsoruce, convertendo-os usando CAST para o Type de destinho TResult. Lança uma exceção quando o Cast falha
ToArray	Converte a coleção para T[] (array do type T)
ToList	Converte a coleção para List<T>
ToDictionary	Converte a coleção para Dictionary<Tkey,Tvalue>
ToLookup	Converte a coleção para um Ilookup<Tkey,Telement>
AsEnumerable	Retorna um IEnumerable<T> de uma coleção
AsQueryable	Executa um Cast ou Converte a coleção para um IQueryable<T>

Fonte: Albahari, 2017, p. 461.

Operadores de conversão sempre recebem uma coleção como input e retornam uma outra coleção com a mesma quantidade de elementos, com a única exceção sendo o método “*OfType*”.

Todos os operadores de conversão causam a enumeração imediata da query LINQ. Isso significa que eles não suportam a execução tardia.

5.6 Agregação

IEnumerable<TSource> -> valor discreto

Os operadores a seguir representam as possibilidades de operadores para agregação existentes no LINQ:

Método	Descrição	Equivalência em SQL
Count, LongCount	Retorna a quantidade de elementos na coleção. Pode executar um "Where" interno se for passado uma Lambda Function de filtro como parâmetro	Count(....)
Min, Max	Retorna o menor ou maior elemento de uma coleção	MIN (...), MAX(...)
Sum, Average	Calcula a somatória ou a média dos valores de elementos de uma coleção	SUM (...), AVG(...)
Aggregate	Executa uma agregação matemática. Recebendo uma Lambda Function como parâmetro para computá-la a cada elemento	N/A

Fonte: Albahari, 2017, p. 467.

Operadores de agregação sempre retornaram um valor discreto, isto é, um valor numérico simples (nunca uma coleção) como resultado.

Exemplos:

```
string[] nomes = { "João", "Silva", "Paulo", "Antonio", "Maria", "Joana" };

// equivalente ao Lenght ou Count
int qtdColecao = nomes.Count();

// Conta os a minusculos
int qtdLetraA = nomes.Sum(n => n.Count(c => c == 'a'));

// Max e Min
int[] numeros = { 28, 32, 14 };
int menor = numeros.Min(); // 14;
int maior = numeros.Max(); // 32
```

FINALIZANDO

Nesta aula abordamos um recurso de linguagem muito útil e comum do C# e da plataforma .NET, o LINQ (*Language Integrated Query*). Como vimos, o recurso do LINQ ajuda muito a obtermos dados de coleções sem precisar criar loops e interações manuais complexas. Utilizando o poder das Lambda

Functions e da querys LINQ, você pode poupar muitas linhas de código e ser mais objetivo.

Você pode conhecer os princípios das Lambdas Functions, sua origem, como utilizar e como elas podem ser úteis no dia a dia. Abordamos as duas sintaxes de consulta LINQ do C#, a Fluent Syntax e a Query Expression, mostrando as diferenças entre elas. Com a Fluent Syntax sendo habitualmente a mais utilizada e preferida pelos desenvolvedores em geral, inclusive reforçamos que toda query escrita utilizando Query Syntax é convertida para Fluent Syntax pelo compilador. Podemos destacar o uso e preferência da Query Syntax para join entre coleções. Por ela ter uma proximidade com a linguagem SQL, muitos desenvolvedores optam por ela para essas situações.

Por último, analisamos juntos como funciona o mecanismo de execução tardia das querys LINQ e como isso é importante para o funcionamento correto desse recurso. Avaliamos também as armadilhas que esse recurso pode trazer caso usado de forma errônea por alguém que negligencie seu comportamento.

No tema 5, abordamos os principais operadores LINQ existentes, deixando um guia rápido de consulta para que você possa tirar suas dúvidas e construir querys LINQ tirando o melhor proveito possível dos operadores existentes.

Caso você precise consultar exemplos de querys LINQ, a Microsoft disponibiliza uma url em que mantém mais de 100 exemplos de querys e situações reais utilizando o LINQ. Tanto em Fluent Sintax quanto em Query Expression. O link pode ser acessado na documentação oficial ou no Github.

LINQ é um assunto comum na linguagem, mas muito extenso. Recomendamos a leitura das referências e materiais auxiliares deixados nesta aula para que você possa aprofundar seus conhecimentos.

Disponível em:

- <https://docs.microsoft.com/en-us/samples/dotnet/try-samples/101-linq-samples/>;
- <https://github.com/dotnet/try-samples#basics>; e
- <https://github.com/dotnet/try-samples/blob/main/101-linq-samples/index.md>.

Acesso em: 6 set. 2021.

REFERÊNCIAS

ALBAHARI, J.; ALBAHARI, B. **C# 7.0 in a Nutshell**. 7. ed. United States of America: O'Reilly Media Inc, 2017.

GRIFFITHS, I. **Programming C# 8.0**. 2. ed. United States of America: O'Reilly Media Inc, 2019.

MICROSOFT. **C# System.Action – Lambda Functions**. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/api/system.action>. Acesso em: 18 maio 2021.

MICROSOFT. **C# Func – Lambda Functions**. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/api/system.func-1>. Acesso em: 18 maio 2021.

MICROSOFT. **Lambda Expressions no C#**. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/operators/lambda-expressions>. Acesso em: 16 maio 2021.

MICROSOFT. **LINQ 101 Samples**. Disponível em: <https://docs.microsoft.com/en-us/samples/dotnet/try-samples/101-linq-samples/>. Acesso em: 20 maio 2021.

MICROSOFT. **Tipos anônimos**. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/fundamentals/types/anonymous-types>. Acesso em: 20 maio 2021.