Aula 3

Programação I

Prof. Alan Matheus Pinheiro Araya

1

Conversa Inicial

2

LINQ

- **Nesta aula vamos abordar um recurso da linguagem C# muito comum no dia a dia do desenvolvedor .NET, independentemente da plataforma/versão: O Language Integrated Query (LINQ)**

3

Introdução ao LINQ e Lambda Functions

4

Introdução ao LINQ

- **"Os recursos do LINQ (Language Integrated Query) 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)

5

- **O LINQ possui duas "notações" ou "sintaxes" distintas no C#:**
  - **Fluent Syntax**
  - **Query Expression**

6

Introdução à Lambda Functions

- São expressões que representam métodos sem nome e sem modificadores de acesso. Muitas vezes com o corpo (body) representado em uma única linha. Uma expressão lambda possui a seguinte anatomia:
  - (input-parameters) → expression
  - (input-parameters) → { <sequência de códigos> }

7

- Dentro do universo de expressões lambdas, vamos utilizar dois Types especiais (delegates) no C#

8

- Action e Action<T> – Encapsula métodos que podem receber parâmetros e não retornam valores
- Func<T> – Encapsula métodos que podem receber parâmetros e retornam valores

9

- Vamos ver na prática os delegates funcionando!

10

LINQ – Fluent Syntax

11

Sintaxe e operadores

- Podemos entender a Fluent Syntax LINQ com a seguinte definição: "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)
- Formalmente, nos referimos aos métodos LINQ como "operadores" LINQ

12

```
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 operador WHERE:
var listaFiltradaWhere = listaValores.Where(p => p > 5); //6,7,8,9

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

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

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

13

- **Podemos encadear os operadores uns com os outros de forma que eles representem uma "pipeline" (esteira ou caminho de fluxo). Com os operadores LINQ podemos:**
  - **Filtrar**
  - **Ordenar**
  - **Agregar**
  - **Agrupar**
  - **Unir**
  - **Converter**

14

- **Vejamos um exemplo de pipeline utilizando Fluent Syntax:**

```
//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);
}
```

```
//Output no console:
JAY
MARY
HARRY
```

15

- **No exemplo anterior, o operador "Where" recebe um Lambda Function, com a seguinte assinatura:**
  - **Where<TSource>(Func<TSource, bool> predicate);**
- **O seu resultado é uma nova lista do mesmo Type, contendo apenas os elementos que deram "match"**

16

- **Depois temos o operador "OrderBy" que produz uma nova versão da lista ordenada:**
  - **OrderBy<TSource>(Func<TSource, TKey> keySelector);**
- **O seu resultado é uma nova lista do mesmo Type, com seus elementos ordenados, segundo o valor retornado pela Func: "keySelector"**

17

- **Depois temos o operador "Select" que realiza um projeção para os elementos da lista. Para cada elemento, ele executa uma Lambda Function que pode transformar o valor do elemento (trataremos dos detalhes dele mais adiante)**
- **O seu resultado é uma nova lista, com os elementos convertidos para maiúsculo (ToUpper())**

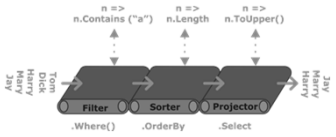
18

Entendendo a pipeline

- **"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)

19

- **Quando os operadores de consulta LINQ são encadeados, a consulta completa se assemelha a uma esteira de linha de produção (pipeline), onde a coleção de saída de um operador é a coleção de entrada para o próximo:**



20

- **Um dos operadores mais versáteis e complexos é o operador Select. Isso porque seu resultado pode ser de um Type diferente do Type da coleção original**
- **O operador Select pode realizar uma transformação do elemento para outro Type**

21

- **O Select pode receber uma Lambda de um Type "T" e pode retornar um Type "Y":**

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

//Utilizando o select para fazer uma Projeção (modificando o tipo de retorno da lista)
//de uma lista de string, transformamos em uma lista de inteiros
IEnumerable<int> queryResultSelect = nomes.Select(n => n.Length);

foreach (int length in queryResultSelect)
{
    Console.Write(length + "|"); //Output no console:  3|4|5|4|3|
}
```

22

LINQ – Query Expression

23

A sintaxe Query Expression

- **O C# também provê uma outra sintaxe para a 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# (...)**

24

- **Vejamos o mesmo exemplo de query LINQ que fizemos anteriormente em Fluent Syntax agora em 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);
}
```

```
//Output no console:
JAY
MARY
HARRY
```

25

- **Vamos entender melhor: uma das diferenças notáveis entre as sintaxes nesse nosso exemplo é o uso do "from" seguido por uma variável, em nosso caso o "n". Essa variável recebe o nome formal de: "variável de intervalo". Ela representa um elemento do array, assim como no "foreach":**
  - **foreach(var n in nomes){...}**

```
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
```

```
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
```

26

- **Toda query escrita em Query Expression pode ser convertida em uma query usando Fluent Syntax – o próprio compilador do C# faz isso durante o *build* do código. O contrário, porém, não é verdadeiro (...)**

27

- **Veja um exemplo de mix de sintaxes:**

```
string[] nomes = { "Tom", "Huck", "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

// 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
```

28

Execução tardia

29

Deffered exectuion

- **Uma das funcionalidades mais importantes dos operadores de query LINQ é sua capacidade de execução tardia (deferred execution/lazy execution)**

30

- Vamos ver na prática o comportamento de "Defered Execution/Lazy Execution" funcionando!

31

Reavaliação/reexecução

- A execução tardia traz outras consequências interessantes, como uma query LINQ não executa no momento que é construída, podemos reutilizá-la várias vezes!

```
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: <nada>
}
```

32

- O comportamento de execução tardia nem sempre é desejado. Seja por que a query pode ser computacionalmente intensiva, seja porque queremos "congelar" os resultados em "certo ponto no tempo" (...)

33

- Para "contornar" a execução tardia, chamamos os operadores de conversão, como "ToList" ou "ToArray". Esses operadores forçam a enumeração imediata da query, produzindo uma nova coleção

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

34

Operadores LINQ

35

Categorias de operadores

- O LINQ possui três principais categorias de operadores:
  - "Coleção in", "coleção out"
  - "Coleção in", "elemento out"
  - Sem parâmetros de entrada, "coleção out"

36

Operadores de filtro

- **Os operadores/métodos de filtro recebem um Type "TSource" como input e retornam o mesmo Type como output**
- **Uma característica desse tipo de operador é: sempre vão lhe retornar uma coleção com menos elementos ou igual à coleção original, nunca com mais elementos!**

37

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: O autor, 2021

38

Operadores de projeção

- **Os operadores de projeção podem transformar o tipo do elemento de entrada em um novo elemento na saída**

39

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 se unificam os resultados de duas tabelas em única tabela de saída. Esta podendo conter duplicatas causadas pelo "flattern"	INNER JOIN, LEFT OUTER JOIN, CROSS JOIN

Fonte: O autor, 2021

40

- **O operador Select pode ainda retornar um objeto especial que chamamos de "tipo anônimo" (anonymous type). Um tipo anônimo é um objeto (Sytem.Object) cuja classe não possui nome nem método, apenas propriedades criadas em tempo de execução pelo operador Select quando combinado com a palavra-chave "new {...}"**

41

```
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}");
}
```

```
//Exemplo de output do console:
//Nome: Portuguese - Cultura: pt - HashCode: -1525760948
//Nome: português(Angola) - Cultura: pt - AO - HashCode: 2058183436
//Nome: Portuguese(Brazil) - Cultura: pt - BR - HashCode: -96748476
//Nome: português(Suíca) - Cultura: pt - CH - HashCode: 1345421020
//Nome: português(Cabo Verde) - Cultura: pt - CV - HashCode: 67000352
//Nome: português(Guiné Equatorial) - Cultura: pt - GQ - HashCode: 504403706
//Nome: português(Guiné - Bissau) - Cultura: pt - GW - HashCode: -1070921970
//Nome: português(Luxemburgo) - Cultura: pt - LU - HashCode: -2086256120
//Nome: português(RAE de Macau) - Cultura: pt - MO - HashCode: 625552102
//Nome: português(Moçambique) - Cultura: pt - MZ - HashCode: -1039145868
//Nome: Portuguese(Portugal) - Cultura: pt - PT - HashCode: -1789600550
```

42

Operadores de combinação (Join)

- **Os operadores/métodos de "Join" do LINQ podem ser muito úteis para combinar resultados de coleções diferentes. Você deve usá-los em situações nas quais o resultado que precisa seja fruto da combinação entre coleções**

43

Método	Descrição	Equivalência em SQL
Join	Aplica um estratégia chamada de "Lookup" para dar match em elementos das duas coleções, produzindo como output um 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 ziper de roupa); aplicando uma Lambda Function que combina os elementos das coleções. Produz um retorno utilizando tuplas, no qual cada tupla é a combinação do elemento da primeira coleção com o elemento da segunda coleção, conforme a Lambda fornecida	N/A

Fonte: O autor, 2021

44

- **Para operações de combinação (Join), o uso da sintaxe de consulta LINQ Query Expression é muito simples, vamos um exemplo prático**

45

Operadores de ordenação

- **Retornam sempre os mesmos elementos, porém em ordens diferentes**

Método	Descrição	Equivalência em SQL
OrderBy, ThenBy	Ordena a coleção em forma descendente	ORDER BY
OrderByDescending, ThenByDescending	Ordena a forma de	
Reverse	Retorna coleção elemento reversa	

Fonte: O autor, 2021

```
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 orde
m alfabética
var ordenacao2 = nomes.OrderBy(a => a.Length).ThenBy(l => l);
```

46

Operadores de conversão

- **Os operadores de conversão LINQ podem ser aplicados a todos os tipos de coleção, pois sumariamente são heranças de IEnumerable<T>. Existem alguns operadores que podem modificar o Type de "destino" (target) das coleções de origem**

47

Método	Descrição
OfType	Enumera a coleção para cada membro de origem Tsoruce, convertendo-os usando CAST para o Type de destino 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 um 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: O autor, 2021

48

Operadores de agregação

- **Sempre retornaram um valor discreto, isto é, um valor numérico simples (nunca uma coleção) como resultado**

49

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 co	MIN( ) MAX( )
Sum, Average	Calcula de elem	
Aggregate	Execut recebe parâm elemen	

Fonte: O autor, 2021

```
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
```

50