Programação Web

Aulas Teóricas – Capítulo 1 – 1.4

1º Semestre - 2023/2024

Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra/Instituto Politécnico de Coimbra

Programação Web

C# – Conceitos Avançados - 4ª Parte

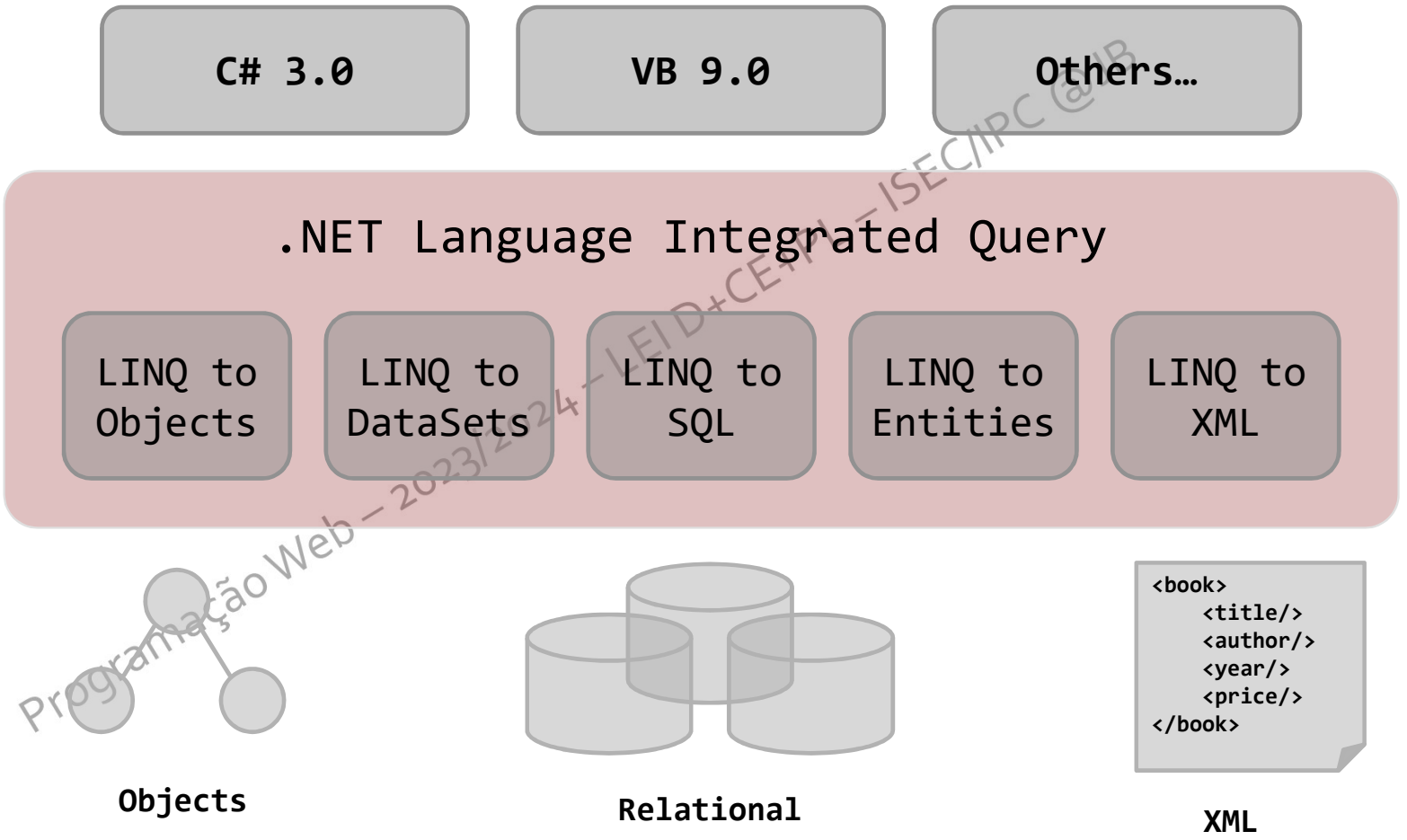
Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra/Instituto Politécnico de Coimbra

LINQ
Language Integrated Query

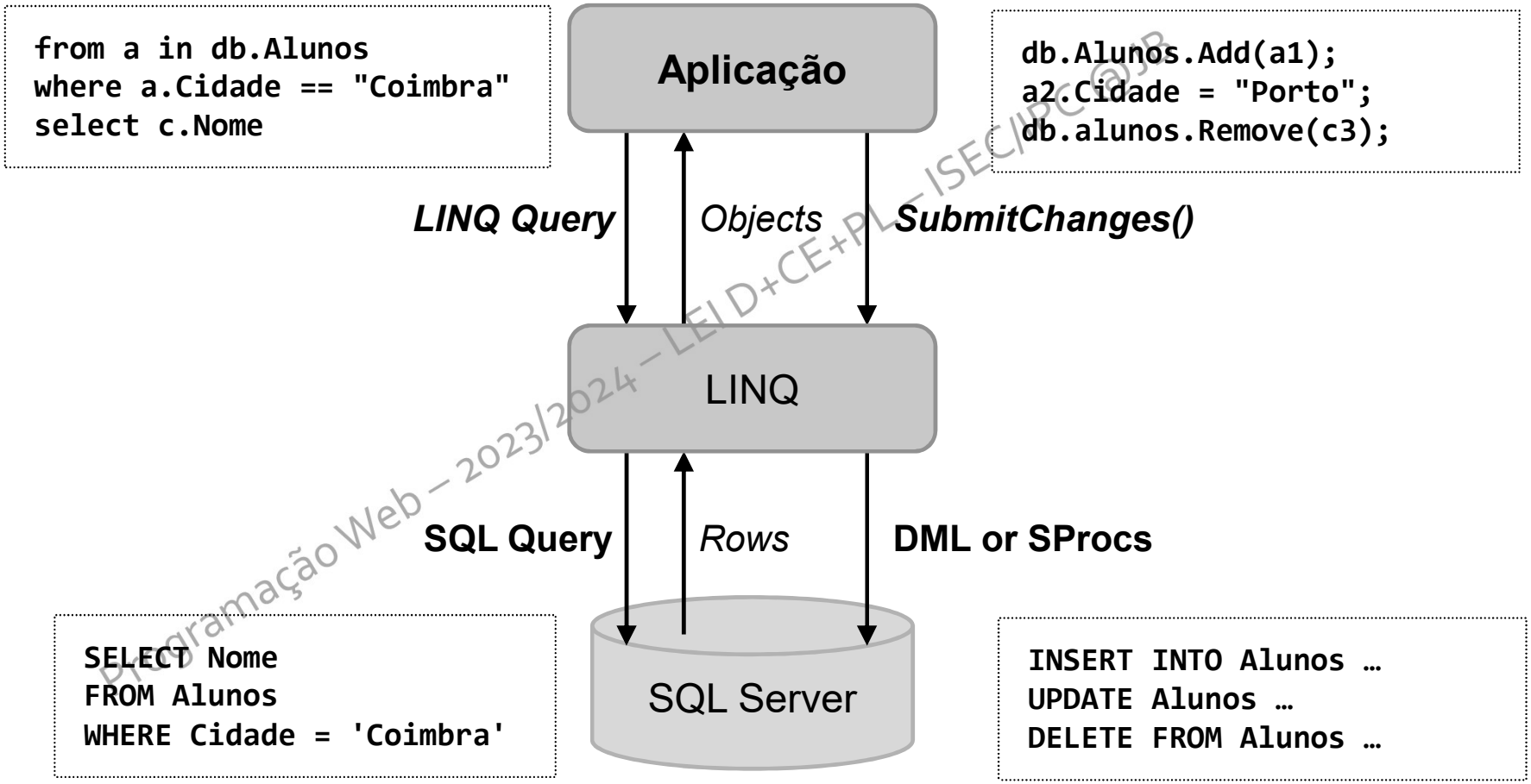
LINQ - Language Integrated Query

- Introduzida com o .Net 3.5 e VS 2008
- Permite efetuar pesquisas em C# ou VB a conjuntos de dados:
  - Objectos em memória – Collections (*LINQ to Objects*)
  - Databases (***LINQ to Entities***) – **Entity Framework**
  - XML (*LINQ to XML*)
  - ADO.NET DataSets (*LINQ to DataSet*)
  - … (com implementação do `IQueryable, IQueryable<T>`)

LINQ *Project*



Arquitectura - *LINQ to Entities*



LINQ

- *Queries* LINQ são *strongly typed*
- Permite a simplificação substantial de código
- Namespace
  - `System.Linq`
- LINQ API inclui duas classes principais **`Enumerable`** & **`Queryable`**.
- *LINQ Operators apresentam um fluent interface*

LINQ

- Vantagens
  - Permite identificação de erros em tempo de compilação
  - Reutilização de código
  - Suporta filtros, ordenação, agrupamentos com menos esforço de implementação

LINQ

- Existem 2 tipos de sintaxe *LINQ*:
  - ***a) Query Operators | Query Syntax***
    - ❖ Semelhante à linguagem *SQL* (*Structured Query Language*)
    - ❖ Inicia com a palavra chave **`from`** para especificar de onde os dados provem e termina com a clausula **`select`** ou **`group`**
    - ❖ Introduz uma variável que pode ser utilizada para filtrar, agrupar

```
var alunosExcelentes = from a in db.Alunos
                       where a.Media >= 18
                       orderby a.Nome
                       select a.Nome
```

LINQ

- Existem 2 tipos de sintaxe *LINQ* (cont.):
  - ***b) Extension Methods – mais "modernos"***
    - ❖ Usa uma *extensão de métodos* e *expressões Lambda*
    - ❖ Recorre a extensão de métodos das classes estáticas **Enumerable** ou **Queryable**.
    - ❖ **<u>Inclui operadores que não estão disponíveis na *Query Syntax*</u>**

```
var alunosExcelentes = db.Alunos.Where(a => a.Media >= 18)
                                .OrderBy(a => a.Nome)
                                .Select(a => a.Nome);
```

LINQ

- ***Preferencialmente devemos utilizer os Extension Methods – mais "modernos"***
  - Instruções muito mais simples
  - Instruções muito mais compactas
  - **<u>Inclui operadores que não estão disponíveis na *Query Syntax*</u>**
    - *A MS tem somente introduzido novos "operadores" nos métodos de extensão e não tem "actualizado" a Query Syntax*
    - …

Com *LINQ* vs Sem *LINQ*

```
IList<string>  PWList = new List<string>() {
        "C#",
        "Entity Framework",
        "Aprender C#",
        "ASP.NET Core com C#" ,
        "Linq",
        "Lambda" ,
        "Identity" ,
};
```

```
var PWListCSharp = new List<string>();
foreach (string x in PWList)
{
    if (x.Contains("C#"))
        PWListCSharp.Add(x);
}
```

Sem LINQ

```
var listaCShap = from s in PWList
                 where s.Contains("C#")
                 select s;
```

Query Syntax

```
var listaCShap = PWList.Where(l=>l.Contains("C#"));
```

Extension Methods

Com LINQ

LINQ

- Seleccionar todas as keywords que incluem C# ?

```
IList<string>  PWList = new List<string>() {
        "C#",
        "Entity Framework",
        "Aprender C#",
        "ASP.NET Core com C#" ,
        "Linq",
        "Lambda" ,
        "Identity" ,
};
... ??????
foreach (string x in listaCShap)
    Console.WriteLine(x);
```

Extension Methods vs Query Operators

```
IList<string>  PWList = new List<string>() {
        "C#",
        "Entity Framework",
        "Aprender C#",
        "ASP.NET Core com C#" ,
        "Ling"
```

```
var listaCShap = from s in PWList
                 where s.Contains("C#")
                 select s;
```

LINQ Query Syntax

```
foreach (string x in listaCShap)
    Console.WriteLine(x);
```

Extension Methods vs Query Operators

```
IList<string> PWList = new List<string>() {
        "C#",
        "Entity Framework",
        "Aprender C#",
        "ASP.NET Core com C#" ,
        "Linq",
```

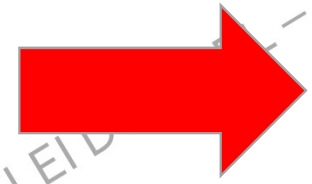
```
var listaCShapEMS =  PWList.
                     Where(x=>x.Contains("C#"));
```

LINQ Extension Methods

```
foreach (string x in listaCShap)
    Console.WriteLine(x);
```

LINQ Extension Methods

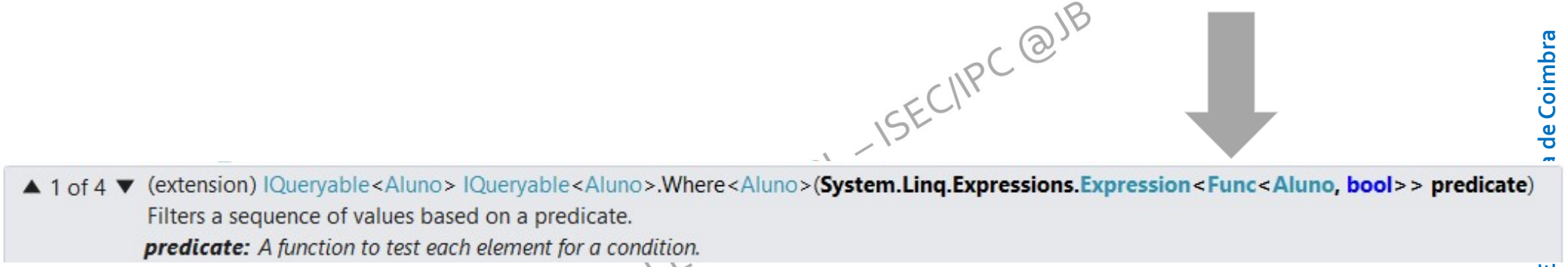
```
var queryAlunos = alunos.
```



- Contains<>
- CopyTo
- Count<>
- DefaultIfEmpty<>
- Distinct<>
- ElementAt<>
- ElementAtOrDefault<>
- Equals
- Except<>

- *Using System.Linq*
- Recorrem a expressões *Lambda*

LINQ Extension Methods



```
▲ 1 of 4 ▼ (extension) IQueryable<Aluno> IQueryable<Aluno>.Where<Aluno>(System.Linq.Expressions.Expression<Func<Aluno, bool>> predicate)
Filters a sequence of values based on a predicate.
predicate: A function to test each element for a condition.
```

Predicate

Expressão Lambda

```
var alunosExcelentes = db.Alunos.Where(a => a.Media >= 18);
```

Operadores LINQ

Restriction	`Where`
Projection	`Select, SelectMany`
Ordering	`OrderBy, ThenBy`
Grouping	`GroupBy`
Joins	`Join, GroupJoin`
Quantifiers	`Any, All`
Partitioning	`Take, Skip, TakeWhile, SkipWhile`
Sets	`Distinct, Union, Intersect, Except`
Elements	`First, Last, Single, ElementAt`
Aggregation	`Count, Sum, Min, Max, Average`
Conversion	`ToArray, ToList, ToDictionary`
Casting	`OfType<T>, Cast<T>`

...

LINQ x busca *case-insensitive*

```
IList<string>  PWList = new List<string>() {
        “c#",
        "Entity Framework",
        "Aprender C#",
        "ASP.NET Core com C#" ,
        "Linq",
        "Lambda" ,
        "Identity" ,
};
```

No exemplo anterior, a pesquisa de caracteres através do LINQ não tinha em conta se esses caracteres eram maiúsculos ou minúsculos e por isso só eram retornados os caracteres que corresponderiam exactamwente ao padrão indicado, ou seja neste exemplo o “C”!
Neste exemplo, utiliza-se também o **StringComparison.OrdinalIgnoreCase** e todas as ocorrências do caracter “C” serão obtidas quer sejam maiúsculas quer minúsculas!

```
var listaCSharp = from s in PWList
                  where s.Contains("C#“, StringComparison.OrdinalIgnoreCase )
                  select s;
```

```
var listaCSharp = PWList.Where(l=>l.Contains("C#“,StringComparison.OrdinalIgnoreCase));
```

LINQ x Expressões Regulares

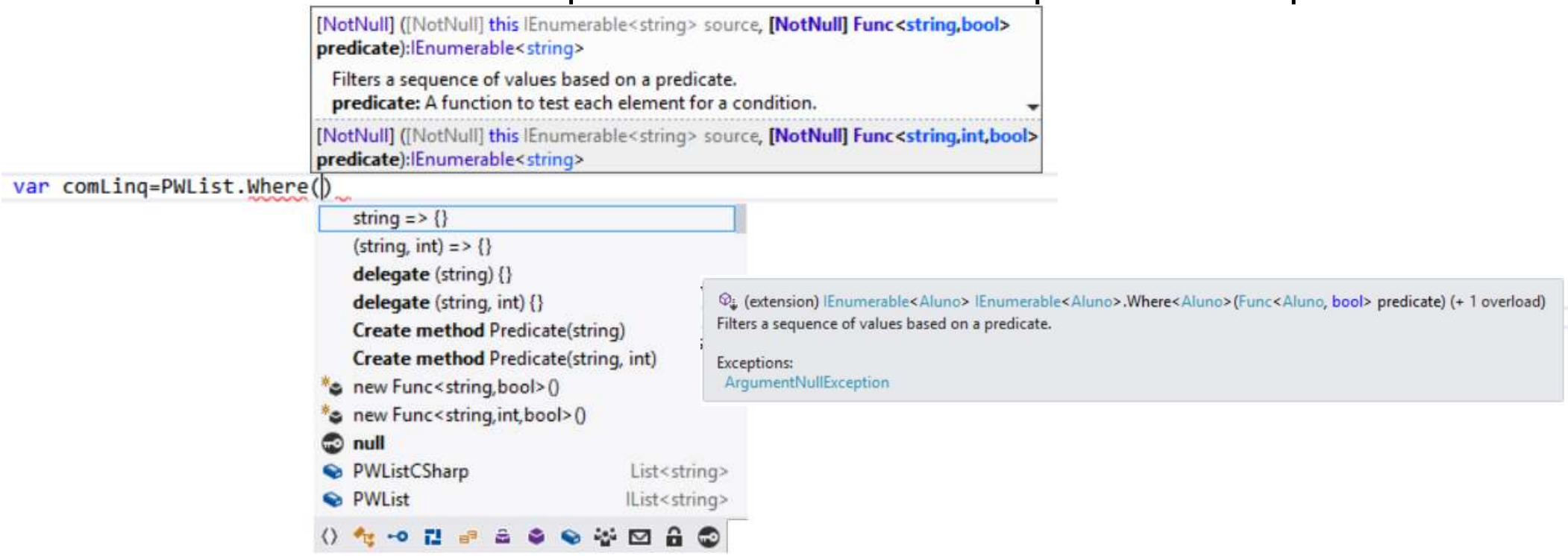
```
string acentuados = @"[^a-zA-Z]";

Regex regex = new Regex(acentuados);

List<string> nomes = new List<string>() { "Cláudio", "Flávia", "Inácio", "Miguel", "Renato", "Ana", "Aurélio", "Francisco",
"Jorge", "Patrícia" };
var resultado = from n in nomes  where regex.IsMatch(n) select n;

Console.WriteLine("Nomes que possuem caracteres acentuados:");
      foreach (var item in resultado)
      {
            Console.WriteLine(item);
      }
```

A utilização de Expressões regulares em conjunto com o LINQ permite a selecção de padrões de caracteres que estejam de acordo com a expressão regular definida para padrão de comparação!

LINQ
Restrição Where

LINQ: Restrição

- ***Where()***
  - Permite filtrar uma sequência de valores tendo por base um predicado



LINQ: Exemplo

```
public class Aluno
{
    public int Numero { get; set; }
    public string Nome { get; set; }
    public int Idade { get; set; }
}
```

Considere esta *class* e o *array de strings* e obtenha o nome dos alunos com 18<=idade<=23 usando *QS e EM*

```
Aluno[] alunos = {
        new Aluno() { Numero = 1, Nome= "Jose Antunes", Idade= 18 },
        new Aluno() { Numero = 2, Nome = "Filipa Moreira",  Idade = 21 },
        new Aluno() { Numero = 3, Nome = "Cristina Frias",  Idade = 25 },
        new Aluno() { Numero = 4, Nome = "Dinis Campos" , Idade = 20 },
        new Aluno() { Numero = 5, Nome = "Soraia Goncalves" , Idade = 31 },
        new Aluno() { Numero = 6, Nome = "Lena Pinheiro",  Idade = 17 },
        new Aluno() { Numero = 7, Nome = "Filipe Cruz",Idade = 19  },
 };
```

LINQ Query Operators

- Seleccionar todos os alunos com idade entre 18 e 23 com QS

```
Aluno[] alunos = {
        new Aluno() { Numero = 1, Nome= "Jose Antunes", Idade= 18 },
        new Aluno() { Numero = 2, Nome = "Filipa Moreira",  Idade = 21 },
        new Aluno() { Numero = 3, Nome = "Cristina Fria",  Idade = 25 },
        new Aluno() { Numero = 4, Nome = "Dinis Campos" , Idade = 20 },
        new Aluno() { Numero = 5, Nome = "Soaraia Goncalves" , Idade = 31 },
};
```

```
var alunosIdade18e23 = from a in alunos
                       where a.Idade > 18 &&
                             a.Idade < 23
                       select a;
```

LINQ Query Operators

- Seleccionar todos os alunos com idade entre 18 e 23 com QS

```
var alunosIdade18e23 = from a in alunos
                       where a.Idade > 18 && a.Idade < 23
                       select a;
```

Será possível especificar a restrição num bloco à parte?

LINQ Query Operators

- Seleccionar todos os alunos com idade entre 18 e 23 com QS

```
bool idadeEntre18e23(Aluno a) {
            return a.Idade > 18 && a.Idade < 23;
}
```

```
var alunosIdade18e23 = from a in alunos
                       where idadeEntre18e23(a)
                       select a;
```

LINQ Query Operators

- Seleccionar todos os alunos com idade entre 18 e 23?

```
var alunosIdade18e23 = from a in alunos
                       where a.Idade > 18 && a.Idade < 23
                       select a;
```

```
Func<Aluno, bool> idadeEntre18e23 = delegate (Aluno s) {
                    return s.Idade > 18 && s.Idade< 23;
};

var alunosIdade18e23 = from s in alunos
                       where idadeEntre18e23(s)
                       select s;
```

LINQ Extension Methods

- Seleccionar todos os alunos com idade entre 18 e 23 com EM?

```
Aluno[] alunos = {
        new Aluno() { Numero = 1, Nome= "Jose Antunes", Idade= 18 },
        new Aluno() { Numero = 2, Nome = "Filipa Moreira",  Idade = 21 },
        new Aluno() { Numero = 3, Nome = "Cristina Fria",  Idade = 25 },
        new Aluno() { Numero = 4, Nome = "Dinis Campos" , Idade = 20 },
        new Aluno() { Numero = 5, Nome = "Soaraia Goncalves" , I
        new Aluno() { Numero = 6, Nome = "Lena Pinheiro",  Idade
        new Aluno() { Numero = 7, Nome = "Filipe Cruz",Idade = 1
```

Jose Antunes
Lena Pinheiro
Filipe Cruz

```
var alunosIdade18e23 = alunos
                .Where(a=>a.Idade>18 && a.Idade<23);
```

Extension Methods Syntax

LINQ Extension Methods

- Seleccionar os alunos com média superior ou igual a 18?

```
...
List<Aluno> alunosExcelentes = new List<Aluno>();
foreach (var aluno in alunos)
{
    if (aluno.Media >= 18)
        alunosExcelentes.Add(aluno);
}
```

Sem LINQ

```
var alunosExcelentes = alunos.Where(a => a.Media >= 18);
```

LINQ EM

LINQ - *Where*

```
int[] numeros = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

var resultado = from n in numbers
                where n < 5
                select n;
```

```
string[] digits = { "zero", "one", "two", "three", "four",
                    "five", "six", "seven", "eight", "nine" };

var shortDigits = digits.Where((digit, index) =>
                               digit.Length < index));
```

```
five
six
seven
eight
nine
```

https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b

Exercício:

- Seleccionar os alunos cujo nome é “Filipa Moreira” usando LINQ EM

```
Aluno[] alunos = {
        new Aluno() { Numero = 1, Nome= "Jose Antunes", Idade= 18 },
        new Aluno() { Numero = 2, Nome = "Filipa Moreira",  Idade = 21 },
        new Aluno() { Numero = 3, Nome = "Cristina Fria",  Idade = 25 },
        new Aluno() { Numero = 4, Nome = "Dinis Campos" , Idade = 20 },
        new Aluno() { Numero = 5, Nome = "Soaraia Goncalves" , Idade = 31 },
        new Aluno() { Numero = 6, Nome = "Lena Pinheiro",  Idade = 17 },
        new Aluno() { Numero = 7, Nome = "Filipe Cruz",Idade = 19  },
 };
```

```
var filipa = alunos.Where(a => a.Nome == "Filipa Moreira");
```

- Seleccionar os alunos com número 5 usando LINQ EM

```
Aluno aluno5 = alunos.Where(a => a.Numero==5);
```

Exercício:

- Obter os números pares da lista

```
IList<int> listInts= new List<int>() {
                        12,15,4,23,17,18,13
                    };
```

```
var numPares = listInts.Where(n => n % 2 == 0);
```

Exemplo: Operador *OfType*

- Obter apenas os números inteiros


```
object[] numeros =  {
      12,15.6,4.2,23,17.1,18,13
};
```



```
var apenasInteiros = numeros.OfType<int>();
```

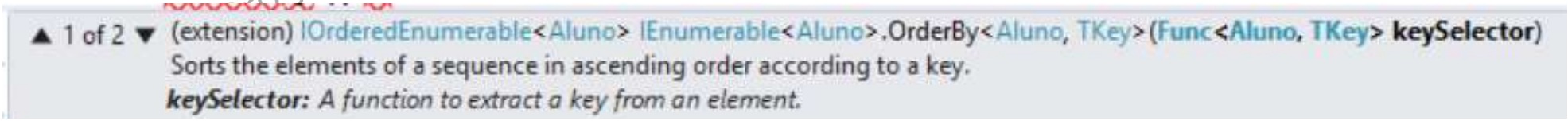
OfType - retorna os elementos que podem ser convertidos com segurança no tipo indicado

LINQ
Ordenação: OrderBy, ThenBy

Ordenação - OrderBy

- *OrderBy()*
  - Permite ordenar uma collection por um determinado valor
- *OrderByDescending()*
  - Permite ordenar de forma descendente

```
var alunosOrdenados =alunos
                     .OrderBy(a => a.Nome);
```



Ordenação - OrderBy

```
var alunosExcelentes = from a in Alunos
                       where a.Media >= 18
                       orderby a.Nome
                       select a;
```

```
var alunosExcelentes = db.Alunos.Where(a => a.Media >= 18)
                                .OrderBy(a => a.Nome);
```

```
var alunosExcelentes = from a in Alunos
                       where a.Naluno == 18
                       orderby a.Nome, a.Media, a.DataNascimento
                      descending
                       select a;
```

LINQ Extension Methods - Ordenação

- *ThenBy() / ThenByDescending()*
  - Permite ordenação de outros campos

```
var alunosExcelentes = alunos.Where(a => a.Media >= 18)
                             .OrderBy(a => a.Nome).ThenBy(a => a.Media);
```

```
var alunosExcelentes = alunos.Where(a => a.Media >= 18)
                             .OrderByDescending(a => a.Nome)
                             .ThenByDescending(a => a.Media);
```

LINQ
Projecção: Select, SelectMany

LINQ: Projeção

- *Select()*
  - Permite aplicar uma transformação a todos os elementos em qualquer estrutura de dados que implemente a *IEnumerable*


```
var alunos = from a in Alunos
             where a.NAluno == 1
             orderby a.Nome, a.DataNascimento descending
             select new {
                        Nome = a.Nome,
                        DataNasc = a.DataNascimento
                 };
```

LINQ Extension Methods - Projeção

- Seleccionar apenas o nome dos alunos com média igual ou superior a 18. Devem ser ordenados pelo nome.

```
var alunosExcelentes =  alunos.Where(a => a.Media >= 18)
                              .OrderBy(a => a.Nome)
                              .Select(a => a.Nome);
```

```
var alunosExcelentes = db.Alunos.Where(a => a.Media >= 18)
```

(local variable) IQueryable<string> alunosExcelentes

```
                        .Select(a=>a.Nome);
```

Projeção

- Algum problema aqui ?

```
IList<Aluno> alunosExcelentes =  alunos.Where(a => a.Media >= 18)
                                      .OrderBy(a => a.Nome)
                                      .Select(a => a.Nome);
```

```
IList<Aluno> alunosExcelentes =  alunos.Where(a => a.Media >= 18)
                                      .OrderBy(a => a.Nome)
                                      .Select(a => a.Nome);
```

Projeção

- Algum problema aqui ?

```
Var alunosExcelentes =  alunos.Where(a => a.Media >= 18)
                                .Select(a => a.Nome)
                               .OrderBy(a => a.Nome);
```



```
Var alunosExcelentes =  alunos.Where(a => a.Media >= 18)
                                .Select(a => a.Nome)
                               .OrderBy(a => a.Nome);
```

LINQ Extension Methods - Projeção

```
var alunosExcelentes = db.Alunos.Where(a => a.Media >= 18)
                                .OrderBy(a => a.Nome)
                                .Select(a => a.Nome);
```

```
foreach (var aluno in alunosExcelentes)
    Console.WriteLine(aluno.Nome);
```

```
foreach (var aluno in alunosExcelentes)
    Console.WriteLine(aluno);
```

Exercício 2

- Obter os quadrados dos números impares

```
IList<int> listInts= new List<int>() {
                        12,15,4,23,17,18,13
                    };
```

```
var quadradoImpares = listInts.Select(x => x * x)
                              .Where(y => y % 2 != 0);
```

```
var quadradoImpares = listInts.Where(y => y % 2 != 0)
                              .Select(x => x * x);
```

Resultado é igual?
Fazem o mesmo?

Exercício 2

- Obter os quadrados dos números impares

```
IList<int> listInts= new List<int>() {
                        12,15,4,23,17,18,13
                    };
```

```
var quadradoImpares = listInts.Select(x => x * x)
                              .Where(y => y % 2 != 0);
```

```
var quadradoImpares = listInts.Where(y => y % 2 != 0)
                              .Select(x => x * x);
```

LINQ Extension Methods - Projeção

```
var alunos = alunos.Where(a => a.Media >= 18)
                   .OrderBy(a => a.Nome)
                   .ThenByDescending(a => a.Media)
                   .Select(a => new {
                                      NomeAluno = a.Nome,
                                      NomeCurso=a.Curso.Nome
                                    } );
```

LINQ Extension Methods

```
var tes = alunos.Where(a => a.Media >= 15)
                .OrderBy(a => a.Nome)
                .ThenByDescending(a => a.Media)
                .Select(a => new
                    {
                        Nome= a.Nome,
                        Disciplinas= a.Disciplinas
                    }
                );
```

LINQ Extension Methods

```
var disAlunos = alunos.Where(a => a.Media >= 15)
                      .OrderBy(a => a.Nome)
                      .ThenByDescending(a => a.Media)
                      .Select(a => a.Disciplinas);
```

(local variable) IEnumerable<List<Disciplina>> disAlunos

```
foreach (var a in disAlunos)
    foreach (var b in a)
        Console.Write(b.Nome + ",");
```

Projecção - SelectMany

- *SelectMany()*
  - Permite efetuar o *select* de uma colecção de colecções e "passa" de um `IEnumerable<IEnumerable<T>>` para `IEnumerable<T>`


```
var disAlunos = alunos.Where(a => a.Media >= 15)
                      .OrderBy(a => a.Nome)
                      .ThenByDescending(a => a.Media)
                      .SelectMany(a => a.Disciplinas);
```

(local variable) IEnumerable<Disciplina> disAlunos

```
foreach (var a in disAlunos)
        Console.Write(a.Nome + ",");
```

LINQ
Agrupar: GroupBy

LINQ Query Syntax - Agrupar

- *GroupBy()*
  - Permite agrupar os dados em grupos diferentes

Com QS

```
var alunos = from a in alunos
             group a by a.Media
             into g
             select g;
```

```
foreach (var grupo in alunos){
    Console.Write("{0} - {1}", grupo.Key, grupo.Count());
    foreach (var aluno in grupo)
        Console.Write("\t{0}", aluno.Nome);
}
```

LINQ Extension Methods - Agrupar

- *GroupBy()*

Com EM

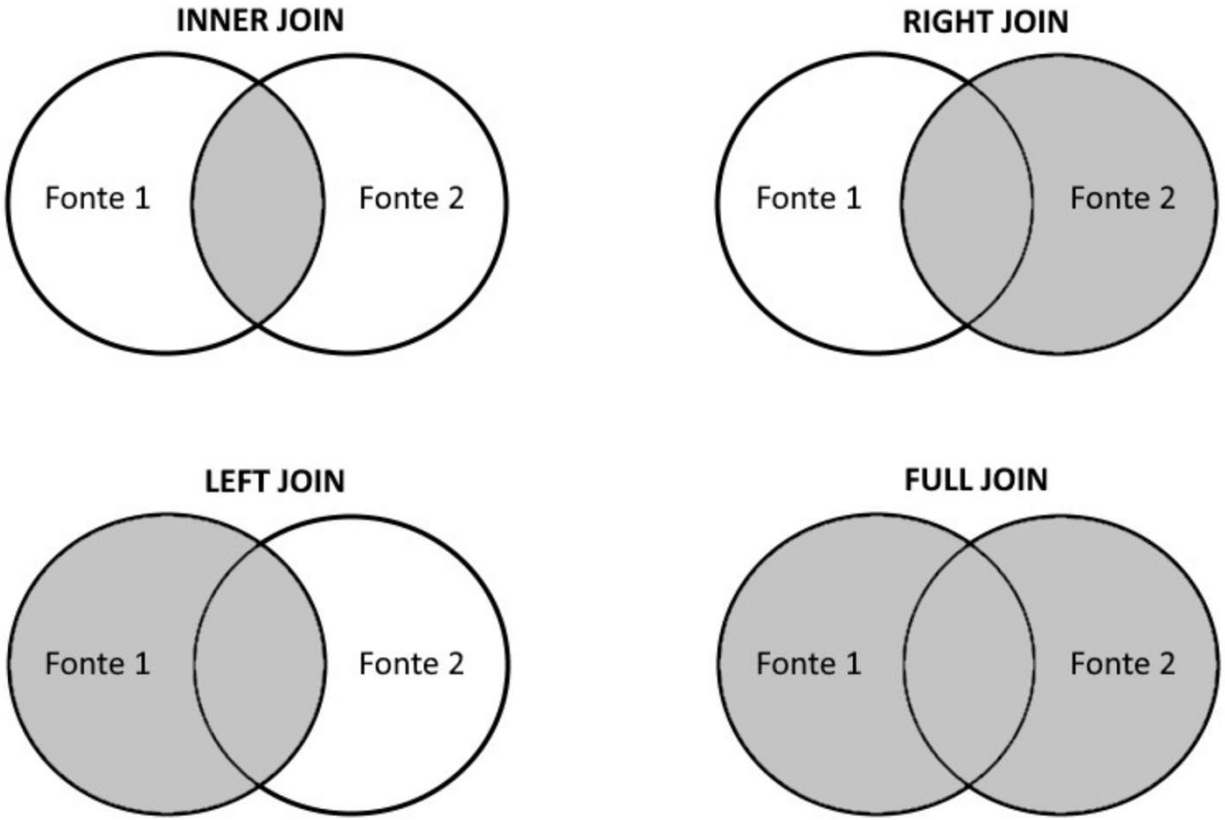
```
var grupos = alunos
                .GroupBy(a => a.Media);
foreach (var grupo in grupos)
{
    Console.Write(grupo.Key);
    foreach (var aluno in grupo)
        Console.Write("\t{0}", aluno.Nome);
}
```

LINQ
Joins: Join, GroupJoin

LINQ - Joins

- *Joins*
  - Utilizados para combinar diferentes listas ou tabelas com uma chave em comum.
- Existem vários tipos de Joins:
  - *Inner Join* - Equivalente ao Inner Join do SQL
  - *Group Join* - Não tem equivalência ao SQL
  - *Cross Join* - Igual ao Cross Join do SQL. Obtém-se o Produto Cartesiano

LINQ - Joins



Prc

LINQ: Inner Join

- **Exemplo**: Obter a lista comum entre as listas

```
var nomes1 = new List<string>() { "Jose", "Maria", "Filipa","Carlos" };
var nomes2 = new List<string>() { "Filipa", "Carlos", "Pedro","Joana","Nuno"};
```

```
var resultado = from n1 in nomes1
                join n2 in nomes2 on n1 equals n2
                select new { n1, n2 };
```

```
var  resultado = nomes1.Join(nomes2, n1 => n1, n2 => n2,
                                (n1, n2) => new { n1, n2 });
```

LINQ: Inner Join

- **Exemplo:** Obter a lista de todos os nomes dos alunos e do nome do respetivo curso

```
public class Aluno
{
    public int Numero { get; set; }
    public string Nome { get; set; }
    public int Idade { get; set; }
    public int Media { get; set; }
    public Curso Curso { get; set; }
}
```

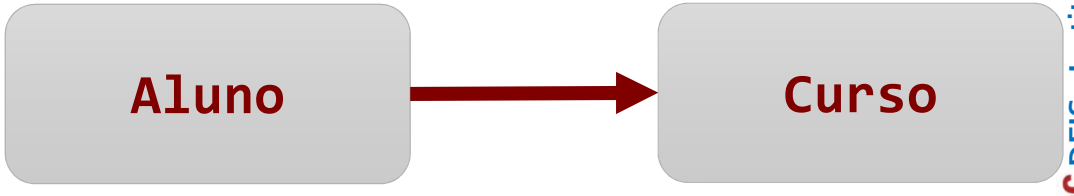
```
public class Curso
{
    public int CursoId { get; set; }
    public string Nome { get; set; }
}
```

LINQ: Inner Join

```
public class Aluno
{
    public int Numero { get; set; }
    public string Nome { get; set; }
    public int Idade { get; set; }
    public int Media { get; set; }
    public Curso Curso { get; set; }
}
```

```
public class Curso
{
    public int CursoId { get; set; }
    public string Nome { get; set; }
}
```

- Para ligação entre entidades, pode-se recorrer às propriedades específicas que as interligam



LINQ: Inner Join

- **Exemplo:** Obter a lista de todos os nomes dos alunos e do nome do respetivo curso


```
var alunos = from a in alunos
             select new { NomeAluno = a.Nome,
                          NomeCurso = a.Curso.Nome
                        };
```

```
var alunos = alunos.Select(a => new {
                          NomeAluno = a.Nome,
                          NomeCurso=a.Curso.Nome
      } );
```

LINQ Query Syntax: Group Join

- Total de alunos por curso

Curso	Alunos
Licenciatura em Engª Informática	4
Mestrado em Engenharia Informática	0


```
var alunos = from c in db.Cursos
             join a in db.Alunos on c.CursoId equals a.CursoId into g
             select new { NomeCurso = c.Nome, TotalAlunos = g.Count()};
```

LINQ Extension Methods: Group Join

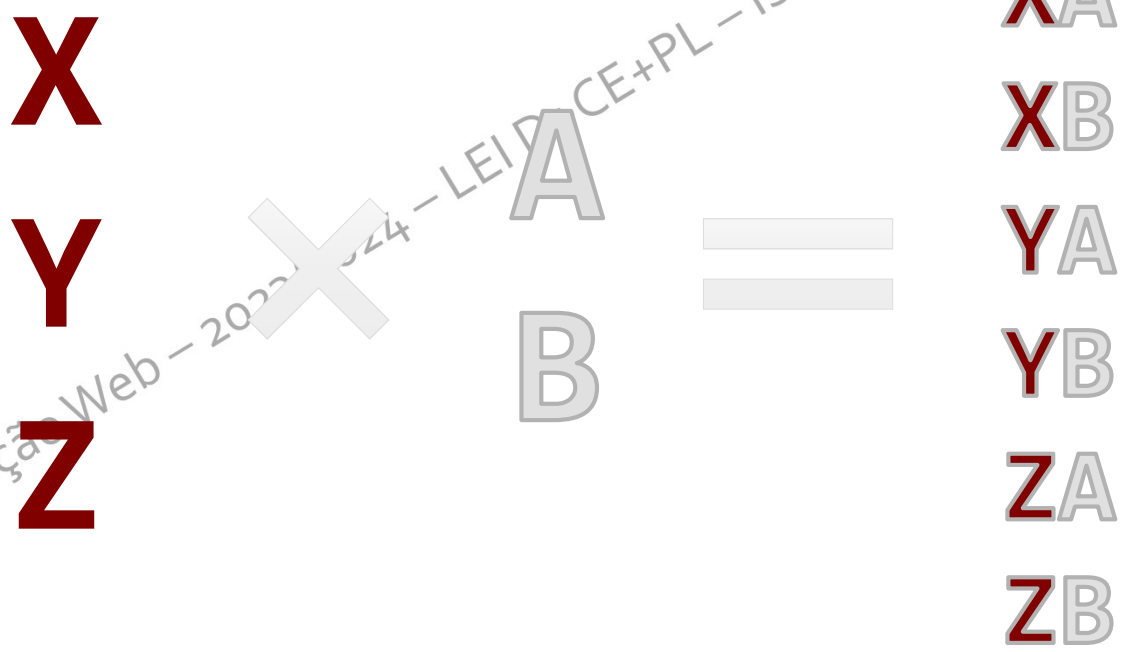
- Total de alunos por curso

Curso	Alunos
Licenciatura em Engª Informática e de Sistemas	4
Mestrado em Engenharia Informática	0

```
var alunosPorCurso = cursos
            .GroupJoin(alunos, c => c.CursoId, a=>a.CursoId,
                       (curso, alunos) => new {
                          NomeCurso = curso.Nome,
                          TotalAlunos=alunos.Count()
                       });
```

LINQ: Cross Join

- Exatamente igual ao *Cross Join* existente em SQL



LINQ Query Syntax: Cross Join

- Lista de todos os cursos de todos os alunos:


```
var alunosQ = from c in cursos
              from a in alunos
              select new {
                 NomeCurso = c.Nome,
                 NomeAluno = a.Nome
              };
```

LINQ Extension Methods: Cross Join

- Lista de todos os cursos de todos os alunos

```
var alunosEM = cursos.SelectMany(c => alunos,
                                 (curso, aluno) => new {
                                     NomeCurso = curso.Nome,
                                     NomeAluno = aluno.Nome
                                 });
```

Outros métodos

- *Any()*
  - Retorna um *boolean*
  - **Exemplo:** Existe algum aluno com média superior a 10?
- *All()*
  - Retorna um *boolean*
  - **Exemplo:** Todos os alunos tem média positiva?


```
var todosComMediaPositiva = alunos.All(a => a.Media > 10);
```

Outros métodos - Particionamento

- Úteis para paginação
  - *Skip()* – Permite "saltar", *skip,* um determinado número de elementos e retorna os restantes
  - *Take()* – Retorna um número específico de elementos

```
var lAlunos = alunos.Skip(3).Take(2);
```

Outros métodos - Particionamento

- Úteis para paginação
  - *TakeWhile* – Retorna elementos que satisfaçam uma determinada condição. Quando a condição já for verdadeira, termina.
  - *SkipWhile* – Permite fazer o skip de elementos em sequência baseados numa condição. Quando deixar de ser verdadeira, retorna os restantes elementos.

Exercício:

```
IList<string> keywordPWList = new List<string>(){
    "C#",
    "Aprender C#",
    "ASP.NET MVC com C#",
    "Entity Framework",
    "Linq",
    "Lambda",
    "Identity",
    "C# in the End",
    };

var result = keywordPWList.TakeWhile(x => x.Contains("C#"));
var result = keywordPWList.SkipWhile(x => x.Contains("C#"));
```

Qual o output?

Outros métodos

- *First*
- *FirstOrDefault*
- *Last*
- *LastOrDefault*
- *Single*
- *SingleOrDefault*

Permitem retornar um único valor

Podem ser invocados com ou sem predicado.

LINQ – First() / Last()

- *First()* / *Last()*
  - Retorna o primeiro/último elemento da sequência ou o primeiro/último elemento correspondente ao predicado especificado
    - Se não existirem elementos na sequência, é gerada uma excepção `InvalidOperationException` com a mensagem *"Sequence contains no elements"*.
    - Se não forem encontrados elementos que satisfaçam a condição do predicado é gerada uma `InvalidOperationException` com a mensagem "*Sequence contains no matching element*".

LINQ – …OrDefault()

- *FirstOrDefault()* / *LastOrDefault()*
  - Retorna o primeiro/último elemento da sequência ou o primeiro/último elemento correspondente ao predicado especificado
    - Se não existirem elementos na sequência ou se não forem encontrados elementos que satisfaçam a condição do predicado é retornado o valor por omissão do tipo da sequência usando *default (T)*

LINQ – Single()

- ***Single()***
  - Se a sequência contém exactamente um elemento, ou exactamente um elemento que satisfaça a condição especificada no predicado, então esse elemento é devolvido.
    - Se não existirem elementos na sequência, ou se não forem encontrados elementos que satisfaçam a condição do predicado, é gerada uma excepção *`InvalidOperationException`* com a mensagem *`"Sequence contains no elements"`*.
    - Se existir na sequência mais do que um elemento, ou se existir mais do que um elemento que satisfaça a condição do predicado é gerada a excepcao *`InvalidOperationException`* com a mensagem "*`Sequence contains more than one element`*".

LINQ – SingleOrDefault()

- *SingleOrDefault()*

  - Igual ao comportamento do Single(), no entanto se não existirem elementos ou se não existirem elementos que satisfaçam a condição do predicado, em vez de gerar excepção, é retornado o valor por omissão do tipo correspondente default(T).

LINQ – ElementAt, ElementAtOrDefault

- ***ElementAt()***
  - Retorna um elemento da sequência baseado na posição *index* especificada

- ***ElementAtOrDefault()***
  - Igual ao comportamento do Element(), no entanto se não existir elemento no indice especificado, retorna o valor por omissão

Exercício:

- Qual o resultado das seguintes instruções?

```
Aluno a1= alunos.First();
var b1= alunos.Last();
var a2= alunos.First(a=> a.Nome == "Filipa Moreira");
var b2= alunos.Last(a=> a.Nome == "Filipa Moreira");
var a3 = alunos.First(a => a.Nome == "Filipa Que?");
var a4 = alunos.FirstOrDefault(a => a.Nome == "Filipa Nao Tem");
var b4 = alunos.LastOrDefault(a => a.Nome == "Filipa Nao Tem");
```

```
Aluno[] alunos = {
        new Aluno() { Numero = 1, Nome= "Jose Antunes", Idade= 18 },
        new Aluno() { Numero = 2, Nome = "Filipa Moreira",  Idade = 21 },
        new Aluno() { Numero = 3, Nome = "Cristina Fria",  Idade = 25 },
        new Aluno() { Numero = 4, Nome = "Dinis Campos" , Idade = 20 },
        new Aluno() { Numero = 5, Nome = "Soaraia Goncalves" , Idade = 31 },
        new Aluno() { Numero = 6, Nome = "Filipa Moreira",  Idade = 26 },
        new Aluno() { Numero = 7, Nome = "Lena Pinheiro",  Idade = 17 },
        new Aluno() { Numero = 8, Nome = "Filipe Cruz",Idade = 19  },
 };
```

Exercício:

- Qual o resultado das seguintes instruções?

```
var aluno = alunos.Single(a =>a.Nome==“António Soares");
var aluno = alunos.Single(a =>a.Nome=="Cristina Fria");
var aluno = alunos.SingleOrDefault(a => a.Nome == “Carlos Afonso");
```

```
Aluno[] alunos = {
        new Aluno() { Numero = 1, Nome= “Carlos Antunes", Idade= 18 },
        new Aluno() { Numero = 2, Nome = "Filipa Moreira",  Idade = 21 },
        new Aluno() { Numero = 3, Nome = "Cristina Fria",  Idade = 25 },
        new Aluno() { Numero = 4, Nome = "Dinis Campos" , Idade = 20 },
        new Aluno() { Numero = 5, Nome = "Soaraia Goncalves" , Idade = 31 }
,
        new Aluno() { Numero = 6, Nome = "Filipa Moreira",  Idade = 26 },
        new Aluno() { Numero = 7, Nome = "Lena Pinheiro",  Idade = 17 },
        new Aluno() { Numero = 8, Nome = "Filipe Cruz",Idade = 19  },
 };
```

LINQ
Agregação: Count, Sum, Min, Max, Average

Funções de Agregação

- ***Count()*** - Contagem

```
var num = alunos.Count();
```

```
var quant = alunos.Where(a=>a.Media > 18)
                  .Count();
```

```
var quant = alunos.Count(a=>a.Media > 18);
```

Funções de Agregação

- ***Sum()*** - Somatório

```
var quant = alunos.Where(a=>a.Media)
                  .Sum();
```

```
var quant = alunos.Sum(a=>a.Media);
```

Funções de Agregação

- ***Min()*** / ***Max()***
  - Permite obter o valor mínimo/máximo de uma *list* ou *collection*

```
int[] num = {2,1,10,4,5,6,7,8,9};
int minimo = num.Min();
int maximo = num.Max();
```

```
var mediaMaior = alunos.Max(a => a.Media);
```

Funções de Agregação

- ***Average()***
  - Permite calcular a média

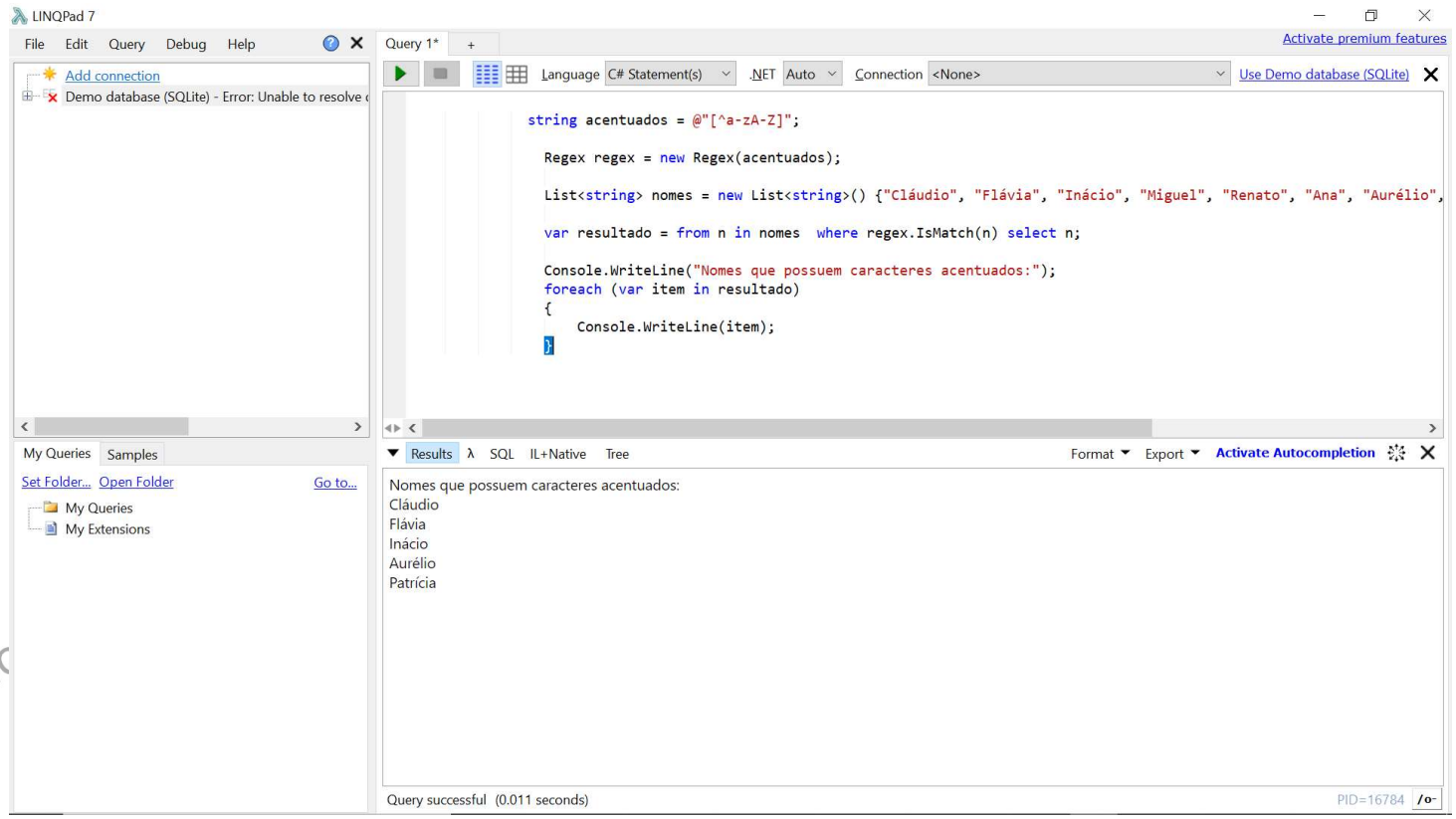
```
var quant = alunos.Select(a => a.Media).Average();
```

```
var quant = alunos.Average(a => a.Media);
```

Métodos para Conjuntos

- ***Union()***
  - Remove elementos repetidos
- ***Intersect()***
- ***Concat()***
  - Não remove os elementos repetidos
- ***Except()***
  - Implica distintos, portanto, remove elementos repetidos
- ***Distinct()***

Treinar LINQ



- Linqpad 7
  - Compatível com as últimas versões do C# e do .NET e Core 3.x
  - Gratuito disponível em https://www.linqpad.net/Download.aspx

Exercício:

- Seleccionar os alunos cujo nomes começam por “F”

```
Aluno[] alunos = {
        new Aluno() { Numero = 1, Nome= "Jose Antunes", Idade= 18 },
        new Aluno() { Numero = 2, Nome = "Filipa Moreira",  Idade = 21 },
        new Aluno() { Numero = 3, Nome = "Cristina Fria",  Idade = 25 },
        new Aluno() { Numero = 4, Nome = "Dinis Campos" , Idade = 20 },
        new Aluno() { Numero = 5, Nome = "Soaraia Goncalves" , Idade = 31 },
        new Aluno() { Numero = 6, Nome = "Lena Pinheiro",  Idade = 17 },
        new Aluno() { Numero = 7, Nome = "Filipe Cruz",Idade = 19  },
 };
```

```
var alunosF = alunos.Where( (a) => {
                return (a.Nome.StartsWith("F"));
            } );
```

```
var alunosF = alunos.Where( a =>
                a.Nome.StartsWith("F"));
```

Exercício:

- Seleccionar os alunos cujo ultimo nome começa por “F”

```
Aluno[] alunos = {
        new Aluno() { Numero = 1, Nome= "Jose Antunes", Idade= 18 },
        new Aluno() { Numero = 2, Nome = "Filipa Moreira",  Idade = 21 },
        new Aluno() { Numero = 3, Nome = "Cristina Fria",  Idade = 25 },
        new Aluno() { Numero = 4, Nome = "Dinis Campos" , Idade = 20 },
        new Aluno() { Numero = 5, Nome = "Soaraia Goncalves" , Idade = 31 },
        new Aluno() { Numero = 6, Nome = "Lena Pinheiro",  Idade = 17 },
        new Aluno() { Numero = 7, Nome = "Filipe Cruz",Idade = 19  },
 };
```

```
var alunosF = alunos.Where( a => a.Nome.Split(' ')
                                .Last()
                                .StartsWith("F"));
```

Exercício:

- Obter o aluno mais novo da turma?

```
var maisNovo = alunos.OrderBy(x => x.Idade).First();
```

- E este ? Funciona?

```
var maisNovo = alunos.Select(x => x.Idade).Min();
```

Exercício:

- Ordenar os alunos pela média descendentemente e depois pela idade ascentente

```
var ordenar = alunos.OrderByDescending(x => x.Media)
                    .ThenBy(x=>x.Idade);
```

Exercício:

- Obtenha, com uma simples instrução os alunos agrupados pela sua média.
- Apresente o resultado como se apresenta na figura


```
var gruposIdade = alunos.GroupBy(x => x.Media);

foreach (var item in gruposIdade)
{
    Console.WriteLine("Média = "+item.Key);
    foreach (var aluno in item)
        Console.WriteLine("   Nº "+aluno.Numero+": "+aluno.Nome);
}
```

```
Média = 12
   Nº 1: Jose Antunes
   Nº 6: Lena Pinheiro
   Nº 7: Filipe Cruz
Média = 16
   Nº 2: Filipa Moreira
   Nº 2: Filipa Moreira
Média = 18
   Nº 3: Cristina Fria
Média = 17
   Nº 4: Dinis Campos
Média = 19
   Nº 5: Soaraia Goncalves
```