Linguagem de Interrogação LINQ

Bases do LINQ

Programação em C#

```
int[] nums = new int[] { 1, 2, 4, 7, 12, 14, 15, 16, 18, 21 };
```

```
List<int> nums5a15 = new List<int>();

for (int i = 0; i < nums.Length; i++)
    if (nums[i] >= 5 && nums[i] <= 15)
        nums5a15.Add(nums[i]);

foreach( int i in nums5a15)
    Console.WriteLine("> " + i);
```

Programação em C# com LINQ

```
var res =  from i in nums where i >= 5 && i <= 15 select
i;


foreach (int x in res)
    Console.WriteLine("> " + x);
```

Em C# é introduzida uma linguagem de interrogação de dados – **LINQ** – que permite simplificar e uniformizar a forma de obter informação particular a partir de um conjunto de dados. O conjunto de dados pode ser por exemplo um *array*, uma coleção, um ficheiro XML ou uma base de dados. A linguagem **LINQ** encontra-se integrada no C# tendo uma sintaxe parecida com a linguagem SQL (mas 'não é SQL'). O formato básico mais simples de uma instrução LINQ toma a forma: `from variável in conjunto_de_dados select valor_seleccionado`.

No exemplo acima a fonte de dados é um *array* de inteiros de onde se vai extrair todos os valores entre 5 e 10 inclusive.

A maior parte das vezes as *queries* LINQ retornam objetos de classes que implementam a interface **`IEnumerable<T>`** mas cujo tipo em concreto pode ser difícil ou mesmo impossível de prever. Neste caso a solução e regra prática é guardar os resultados de uma *query* LINQ numa variável de tipo implícito – "**`var`**".

Execução diferida

Programação em C# com LINQ

```
List<Aluno> als = new List<Aluno> {
    new Aluno {Nome = "Joao Carlos", Numero = 1234},
    new Aluno {Nome = "Jose Antunes", Numero = 2345},
    new Aluno {Nome = "Joao Silva", Numero = 3456},
    new Aluno {Nome = "Ana Costa", Numero = 4567},
    new Aluno {Nome = "Ivo Tiago", Numero = 5678},
    new Aluno {Nome = "Manuel Jose", Numero = 6789},
    new Aluno {Nome = "Jose Jacinto", Numero = 7890}
};

Console.WriteLine("ALUNOS: \n");
    foreach (Aluno al in als)
        Console.WriteLine("> " + al);

// Selecciona alunos que têm 'Jose' no nome
var res = from a in als where a.Nome.Contains("Jose") select a.Nome;

Console.WriteLine("\n\nALUNOS com o nome 'Jose': \n");

foreach (var a in res)
    Console.WriteLine("> " + a);

Console.WriteLine("\n2 alunos adicionados");
    als.Add(new Aluno { Nome = "Carlos Costa", Numero = 8901 });
    als.Add(new Aluno { Nome = "Sa Jose Pinto", Numero = 9012 });

Console.WriteLine("\n\nALUNOS com o nome 'Jose': \n");
    foreach (var a in res)
        Console.WriteLine("> " + a);
```

A execução de uma *query* **LINQ** só se efetua quando o resultado é acedido. Neste caso diz-se que se efetua a **execução diferida** da *query*. Esta situação leva a que o resultado traga sempre os valores atualizados. No exemplo acima a segunda listagem de alunos feita último ciclo **`foreach`** já mostra o aluno "Sa Jose Pinto" introduzido imediatamente antes deste ciclo.

Execução imediata

Programação em C# com LINQ

```
// expressão LINQ com execução imediata
List<string> joses = (from a in als where a.Nome.Contains("Jose") select a.Nome).ToList();

Console.WriteLine("\n\nALUNOS com o nome 'Jose' (2): \n");
    foreach (string n in joses)
        Console.WriteLine("> " + n);
```

É possível executar imediatamente uma expressão LINQ chamando um método de extensão para o resultado entre os vários métodos disponíveis definidos para o tipo **`Enumerable`** (por exemplo **`ToArray<T>()`**, **`ToList<T>()`**, **`ToDictionary<TSource,TKey>()`**). Neste caso obtém-se uma imagem estática dos dados (*snapshot*). No exemplo acima chamou-se o método **`ToList()`** que faz a **execução imediata** da *query* e converte o resultado para uma lista de *strings*. (Nota: na realidade está-se a chamar o método **`ToList<string>()`** onde o tipo genérico foi omitido porque o compilador o consegue determinar automaticamente).

LINQ query operators

Programação em C# com LINQ

```
Console.WriteLine("\n1 - Alunos por ordem decrescente do nome");
var res1 = from a in als orderby a.Nome descending select a;
Console.WriteLine("\nALUNOS: \n");
foreach (var a in res1)
    Console.WriteLine("> " + a);


Console.WriteLine("\n2 - Nome dos alunos com número superior a 5000");
var res2 = from a in als where a.Numero>5000 select a;
Console.WriteLine("\nALUNOS: \n");
    foreach (var a in res2)
        Console.WriteLine("> " + a);
```

A linguagem integrada **LINQ** utiliza na sua sintaxe **operadores** como por exemplo os operadores **`from`** e **`in`** essenciais de qualquer *query*. Nos exemplos mostrados demonstra-se igualmente a utilização do operador **`select`** que define uma sequência de elementos a partir dos dados, o operador **`where`** que permite filtrar os dados a partir de uma expressão booleana, o operador **`orderby`** que permite uma ordenação dos dados e os operadores **`ascending`** e **`descending`** que permitem em conjunto com o **`orderby`** ordenar os resultados de forma ascendente ou descendente. Para além destes operadores existem vários outros, alguns semelhantes aos existentes em SQL, tais como **`join`**, **`on`**, **`equals`**, **`into`**, **`group`**, **`by`**, etc.
Para além destes operadores integrados na linguagem existem operadores sobre a forma de métodos de extensão genéricos que se aplicam diretamente aos resultados da *query* como, por exemplo, **`ToList<>()`** que converte o resultado numa lista de elementos, **`ToArray<>()`** que converte o resultado num *array* de elementos, **`Reverse<>()`** que inverte a ordem dos elementos, etc. Além dos mencionados encontram-se alguns que fazem operações sobre conjuntos de elementos (**`Distinct<>()`**, **`Union<>()`**, **`Intersect<>()`**, etc.), ou outros que agregam os resultados (**`Count<>()`**, **`Sum<>()`**, **`Min<>()`**, **`Max<>()`**, etc.). Recomenda-se a análise dos vários operadores disponíveis na documentação da linguagem C#.

Projeção de dados com tipos anónimos

Programação em C# com LINQ

```
// Projecção de dados com tipos anónimos
List<Disciplina> disc = new List<Disciplina>();

// Adição de várias disciplinas e alunos...
// ...

var res3 = from d in disc select new { Nome = d.Nome, NumeroAlunos = d.TotalAlunos };

Console.WriteLine("\nDISCIPLINAS: \n");
foreach (var r in res3)
    Console.WriteLine("> " + r.Nome + " com " + r.NumeroAlunos + " alunos.");
```

É possível obter como resultado de uma *query* LINQ um conjunto de dados de um tipo anónimo. Estes tipos podem ser úteis quando se pretende apenas analisar um subconjunto da informação existente nos dados. No exemplo dado foi criado um tipo anónimo com os campos **`Nome`** e **`NumeroAlunos`** que contém apenas a informação do nome e do total de alunos existente na disciplina.

LINQ queries sem a sintaxe integrada

Sintaxe simplificada LINQ	Sintaxe LINQ com métodos de extensão

`var res4 = als.Where(a => a.Nome.Contains("Jose")) .Select(a => a.Nome);`
 	
`var res5 = from a in als orderby a.Nome select a;`
 	
`var res6 = from a in als where a.Numero>5000 select a.Nome;`
	`var res4 = from a in als where a.Nome.Contains("Jose") select a.Nome;`
	
	`var res5 = als.OrderBy(a => a.Nome).Select(a => a);`
	
	`var res6 = als.Where( a => a.Numero>5000). Select(a => a.Nome);`

É possível não utilizar a sintaxe integrada do LINQ definida para a linguagem C#. Neste caso aplicam-se directamente os métodos de extensão sobre as fontes de dados. Esta é, de facto, a forma como o compilador C# executa as expressões LINQ sendo que a sintaxe normalmente usada é apenas uma forma simplificada da chamada aos respectivos métodos de extensão.
De referir ainda que muitos dos métodos de extensão têm como argumento referências para métodos que são habitualmente chamados para cada elemento do conjunto de dados.
Nos exemplos apresentados mostra-se a sintaxe LINQ usando operadores e a mesma operação usando os métodos de extensão. Os métodos que são passados como argumentos usam expressões lambda.