


Aspectos avançados da linguagem C#

Notas para a disciplina de Ambientes
Virtuais de Execução
Semestre de Inverno, 10/11

Sumário

- ◆ Enumeradores e Iteradores
- ◆ Métodos anónimos  a partir do C# 2.0
 - Captura de variáveis
- ◆ Construções de suporte ao LINQ (**L**anguage **I**ntegrated **Q**uery) (a partir do C# 3.0)
 - Métodos de extensão
 - Tipificação implícita
 - Tipos anónimos
 - Iniciadores
- ◆ Exemplos

Enumeráveis e enumeradores

- Quando um tipo é passível de ser enumerado, deve implementar a interface **IEnumerable<T>**, que contém o único método:

```
IEnumerator<T> GetEnumerator()
```

- O tipo usado para enumerar é **IEnumerator<T> : IDisposable**

```
bool MoveNext()  
T Current { get; }
```

- Admitindo que **en** é enumerável, a construção
foreach(T t in en){ *body* } é traduzida em:

```
using (IEnumerator<T> enumerator1 = en.GetEnumerator()) {  
  while (enumerator1.MoveNext()) {  
    T t = enumerator1.Current;  
    //body  
  }  
}
```

Interfaces genéricas e não genéricas

- ♦ **IEnumerable<T> : IEnumerable**
 - Método não genérico **IEnumerator GetEnumerator()**, com implementação de forma explícita
 - Método genérico **IEnumerator<T> GetEnumerator()**
- ♦ **IEnumerator<T> : IDisposable, IEnumerator**
 - Acrescenta a interface **IDisposable**
 - Métodos **Reset** e **MoveNext** são de **IEnumerator**, embora o método **Reset** não necessite de ser suportado
 - Duas propriedades **Current**
 - Genérica, retorna **T**
 - Não genérica, retorna **object** – implementada de forma explícita

Utilizações de IEnumerable/IEnumerator

- A utilização de enumeradores sobre tipos enumeráveis pode ter dois tipos de implementações/utilizações:



Sequências onde os elementos já estão calculados e armazenados numa estrutura de dados

Sequências onde os elementos são calculados apenas quando necessários – aquando da chamada do método MoveNext

Exemplo: Filtro (1)

- ◆ Dada uma sequência **s1** e um predicado **p**, calcular a sequência **s2** com os elementos de **s1** que satisfazem **p**

```
IEnumerable<T> FilterToList<T> (IEnumerable<T> seq, Predicate<T> pred)
```

- ◆ Solução 1 (*eager*)

- Criar uma estrutura de dados **s2** (ex. lista) que seja enumerável
- Percorrer **s1**, copiando para **s2** todos os elementos de **s1** que satisfazem **p**
- Retornar **s2**



```
public static IEnumerable<T> FilterToList<T>
    (IEnumerable<T> seq, Predicate<T> pred){
    List<T> result = new List<T>();
    foreach(T t in seq){
        if(pred(t)) result.Add(t);
    }
    return result;
}
```

Exemplo: Filtro (2)

♦ Solução 2 (*Lazy*)

- Retornar um objecto que implemente **IEnumerable<T>**, em que o enumerador associado possua as seguintes funcionalidades:
 - Contém um enumerador para **s1**
 - O método **MoveNext** avança o enumerador sobre **s1** enquanto o predicado é falso
 - A propriedade **Current**, retorna o elemento corrente de **s1**

Classe Filter: enumerável

```
class Filter<T> : IEnumerable<T>
{
    IEnumerable<T> enumerable;
    Predicate<T> pred;

    public Filter(IEnumerable<T> ie, Predicate<T> p)
    {
        enumerable = ie;
        pred = p;
    }

    public IEnumerator<T> GetEnumerator()
    {
        return new FilterEnumerator enumerable.GetEnumerator(),
            pred);
    }
}
```


Classe Filter: enumerador

```
class FilterEnumerator : IEnumerator<T> {
    IEnumerator<T> enumerator;
    Predicate<T> pred;

    public FilterEnumerator(IEnumerator<T> ie, Predicate<T> p) {
        enumerator = ie;
        pred = p;
    }
    public void reset() { enumerator.reset(); }

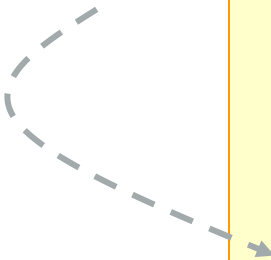
    public void Dispose() { enumerator.Dispose(); }

    public bool MoveNext() {
        bool b;
        while ((b = enumerator.MoveNext()) &&
            pred(enumerator.Current) == false);
        return b;
    }

    public T Current { get { return enumerator.Current; } }
}
```

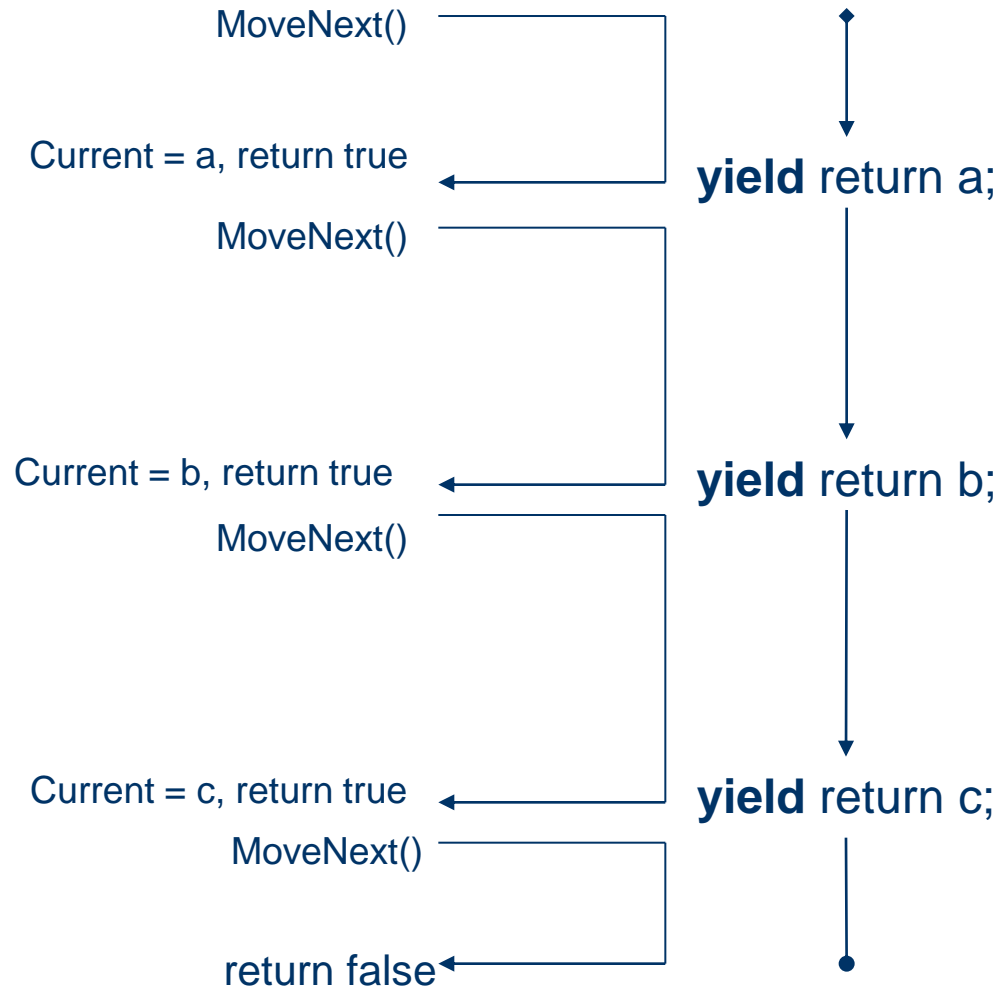
Classe Filter: comentários

- ♦ A segunda solução implica a criação de duas classes
 - **Filter**: implementação de **IEnumerable<T>**
 - **FilterEnumerator**: implementação de **IEnumerator<T>**
- ♦ É necessário passar o predicado e o enumerador entre a fonte e a classe enumerável, e entre esta e a classe enumeradora
- ♦ Faltam ainda os métodos das interfaces não genéricas
- ♦ Lógica do método **MoveNext** é mais complexa quando comparada com a implementação não *lazy*



```
public static IEnumerable<T> FilterToList<T>
    (IEnumerable<T> seq, Predicate<T> pred){
    List<T> result = new List<T>();
    foreach(T t in seq){
        if(pred(t)) result.Add(t);
    }
    return result;
}
```

Corrotinas



Utilização na definição de enumeradores

MoveNext()

Current = t

MoveNext()

```
public static IEnumerable<T>
Filter<T>(IEnumerable<T> seq,
          Predicate<T> pred){
    foreach (T t in seq)
    {
        if (pred(t))
            yield return t; // result.Add(t);
    }
}
```

Iteradores

```
public static IEnumerable<T> Filter<T>(IEnumerable<T> ie, Predicate<T> pred)
{
    foreach (T t in ie) { if (pred(t)) yield return t; }
}
```

- ♦ O método **Filter** retorna uma classe gerada pelo compilador e que implementa **IEnumerable<T>** e **IEnumerator<T>**
 - Os seus métodos, nomeadamente o **MoveNext**, reflectem a sequência de acções definida no corpo da função geradora
 - O *contexto da geração é capturado* para ser usado no método **MoveNext**
- ♦ Sintaxe e semântica
 - **yield return t** – sinaliza que o fio de execução (do **MoveNext**) termina com **true** e **Current = t**
 - **yield break** – sinaliza que o fio de execução (do **MoveNext**) termina com **false** (**Current** é indeterminado)

Iteradores: geradores de enumeradores

Classe gerada pelo compilador com base no corpo da função **Filter**

```
class X :  
IEnumerator<T>,IEnumerable<T>{
```

```
T Current { get {...}};
```

```
bool MoveNext(){  
    máquina de estados  
}
```

```
IEnumerator<T> ie;  
Predicate<T> pred;
```

```
IEnumerator<T> Filter<T>(ie, pred)  
{  
    foreach (T t in ie){  
        if (pred(t)) yield  
            return t;  
    }  
}
```

returns

Variáveis capturadas

Função Filter: código gerado (classe X)

```
public static IEnumerable<T> Filter<T>(IEnumerable<T> seq,
Predicate<T> pred) {
    X d = new X(-2);
    d._seq = seq;
    d._pred = pred;
    return d;
}
```

- ◆ Classe **X** gerada pelo compilador:
 - Contem campos com o contexto da geração, que neste caso são os parâmetros **seq** e **pred**
 - Implementa simultaneamente **IEnumerable<T>** e **IEnumerator<T>**, com otimização para o caso em que apenas é criado um enumerador
 - O método **MoveNext** implementado através duma máquina de estados
 - Estado -2: ainda não foi obtido o enumerador
 - Estado 0: enumerador no estado inicial
 - Estado -1: enumerador no estado final

Classe X: campos e construtor

- ♦ Classe X:
 - Campos para a implementação da máquina de estados
 - Campos com o contexto capturado para o enumerável e para o enumerador

```
private sealed class X : IEnumerable<T>, IEnumerator<T>{  
    // máquina de estados  
    private int state; // estado do enumerador  
    private T current; // elemento actual  
  
    // campos do enumerador  
    public IEnumerator<T> en; // enumerador fonte  
    public Predicate<T> pred; // predicado  
    public IEnumerable<T> seq; // enumerável fonte  
  
    // campos do enumerável  
    public Predicate<T> _pred; // predicado fonte  
    public IEnumerable<T> _seq; // enumerável fonte  
  
    public X(int _state){ state = _state;}
```


Classe X: método GetEnumerator

- ♦ Verificação, de forma atômica (**CompareExchange**), se não foi criado nenhum enumerador a partir deste enumerável
 - Em caso positivo, é aproveitada a mesma instância
 - Em caso negativo, é criada uma nova instância

```
IEnumerator<T> IEnumerable<T>.GetEnumerator()  
{  
    X d;  
    if (Interlocked.CompareExchange(ref this.state, 0, -2) == -2)  
        d = this;  
    else  
        d = new X(0);  
    d.seq = this._seq;  
    d.pred = this._pred;  
    return d;  
}
```

Classe X: método MoveNext

```
private bool MoveNext(){
    bool flag1;
    switch (state) {
        case 0: { break; }
        case 1: { goto state1; }
        case 2: { goto state2; }
        default: { goto state1; }
    }
    try {
        state = -1; en = this.seq.GetEnumerator(); state = 1;
        while (en.MoveNext())
        {
            T aux = en.Current;
            if (pred(aux) == false) { goto next; }
            current = aux; state = 2; return true;
            state = 1;
        }
        state = -1;
        if (en != null){ en.Dispose(); }
        state1: flag1 = false;
    }
    fault { ((IDisposable) this).Dispose(); }
    return flag1;
}
```

Saltar para o estado anterior

Algoritmo presente na função construtora

Exemplos de iteradores: concatenação e projecção de sequências

◆ Concatenação de duas sequências

```
public static IEnumerable<T> Append<T>(IEnumerable<T> seq1,
                                       IEnumerable<T> seq2)
{
    foreach(T t in seq1) yield return t;
    foreach(T t in seq2) yield return t;
}
```

◆ Projecção de sequências

```
public static IEnumerable<U>
Select<T,U>(IEnumerable<T> seq, Converter<T,U> selector) {
    foreach(T t in seq) yield return selector(t);
}
```

Métodos anónimos: motivação

- ♦ Seja o método:

```
public static IEnumerable<T>  
Where<T>(this IEnumerable<T> col, Predicate<T> p)
```

que retorna um enumerável com todos os elementos do enumerável `col` recebido, que satisfazem o predicado `p`. Tirando partido do método `Where`, pretende-se implementar o método `InRange`, à frente, que retorna um enumerável com todos elementos de `ts` entre `min` e `max`

```
public static IEnumerable<T> InRange<T>( this IEnumerable<T> ts, T min, T max)  
    where T : IComparable<T> {  
    return Where(ts, ?);  
}
```

Como definir o predicado?

Definição do predicado com recurso a uma classe auxiliar

- ♦ A solução passa por criar a classe genérica `RangeComparer<T>`
 - Campo `min` e `max` com a definição do intervalo
 - Construtor para a iniciação dos campos
 - Método (predicado) para verificar se um dado `T` passado como parâmetro está no intervalo entre `min` e `max`

```
class RangeComparer<T> where T : IComparable<T> {  
    private T min, max;  
    public RangeComparer(T mn, T mx) { min = mn; max = mx; }  
  
    public bool IsInRange(T t) {  
        return t.CompareTo(min) >= 0 && t.CompareTo(max) <= 0;  
    }  
}
```

Implementação de `InRange` com e sem métodos anónimos

Sem método anónimo

```
public static IEnumerable<T> InRange<T>( this IEnumerable<T> ts, T min, T max)
    where T : IComparable<T> {
    return Where(ts, new RangeComparer<T>(min, max).IsInRange);
}
```

Com método anónimo

```
public static IEnumerable<T> InRange2<T>( IEnumerable<T> ts, T min, T max)
    where T : IComparable<T> {
    return Where(ts, delegate(T t) { return t.CompareTo(min) >= 0 &&
t.CompareTo(max) <= 0; });
}
```

Destas duas formas deixa de ser necessária a classe auxiliar

```
public static IEnumerable<T> InRange2<T>( IEnumerable<T> ts, T min, T max)
    where T : IComparable<T> {
    return Where(ts, t => t.CompareTo(min) >= 0 && t.CompareTo(max) <= 0 ; );
}
```

Com expressão lambda

Métodos anónimos: classe e *delegate*

◆ Classe gerada automaticamente

```
private sealed class X{  
    public X();  
    public bool m(FileInfo x); // Método de  
    comparação  
    public string ext; // Estado  
}
```

◆ Instância do *delegate* gerado

```
public static void ListFilesByExt(string path, string ext){  
    Predicate<FileInfo> predicate1 = null;  
    X classe1 = new X();  
    classe1.ext = ext; // Acesso ao contexto  
    predicate1 = new Predicate<FileInfo>(classe1.m);  
    using (IEnumerator<FileInfo> enumerator1 = Global.Filter<FileInfo>(  
        Global.GetDirectoryEnumerator(  
            new DirectoryInfo(path)),  
            predicate1)  
        .GetEnumerator())
```

A assinatura do método anónimo é função do tipo *delegate*, que é inferido do contexto

Variáveis capturadas

- ◆ Variáveis externas: **variáveis locais**, **parâmetros valor** e **arrays** de parâmetros cujo **scope** inclua o método anónimo.
- ◆ Se o método anónimo estiver definido dentro dum método instância, então **this** também é uma variável externa
- ◆ As variáveis externas referidas pelo método anónimo dizem-se *capturadas*
- ◆ O compilador de C# cria uma classe com:
 - Um campo por cada variável capturada;
 - um método, correspondente ao método anónimo.

Variáveis capturadas (cont.)

- ♦ A instanciação de um método anónimo consiste na criação de uma instância da classe referida acima e na captura do contexto.
- ♦ A implementação dos métodos anónimos **não introduziu alterações** na CIL nem na CLI.
- ♦ No entanto, existem algumas limitações:
 - Parâmetros referência (**ref** e **out**) não podem ser capturados uma vez que não é possível garantir a validade das referências no momento da execução do *delegate*

C# 3.0: sumário

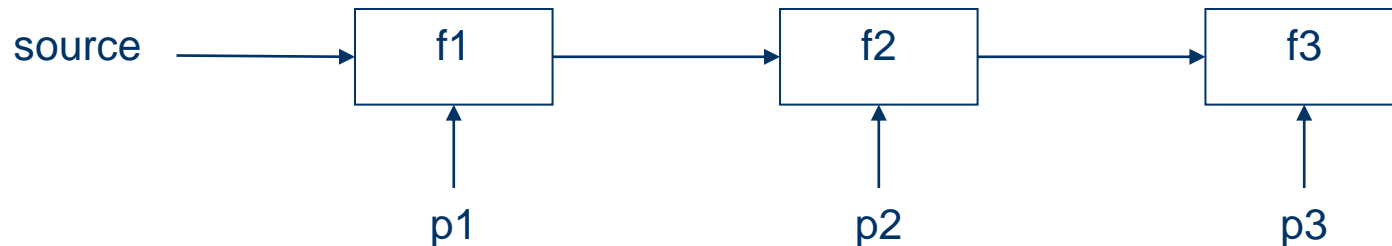
- ◆ Expressões lambda
 - ◆ Métodos extensão
 - ◆ Iniciadores
 - ◆ Tipos anónimos
 - ◆ Tipificação implícita
-
- ◆ Baseado em: Microsoft, “C# Version 3.0 Specification September 2005”, Setembro de 2005

Expressões lambda

- ♦ Método anónimo (C# 2.0): **delegate (int x){ return x>y;}**
 - Tipificação explícita do parâmetro
 - Corpo do método é um *statement*
- ♦ Expressão lambda: **x => x>y**
 - Tipificação implícita do parâmetro, obtida através do contexto
 - Corpo do método é uma expressão

Métodos de extensão

- ♦ Métodos estáticos invocáveis usando a sintaxe de método de instância
- ♦ Utilização: simplificar a sintaxe de construção de *pipelines*
- ♦ Com métodos estáticos
 - `f3(f2(f1(source,p1),p2),p3)`
- ♦ Com métodos instância
 - `source.f1(p1).f2(p2).f3(p3)`
- ♦ Métodos extensão (têm de estar presentes em classes estáticas)
 - `IEnumerable<T> f1(this IEnumerable<T>, P1 p1)`



System.Query

- ♦ O *assembly* **System.Query** define um conjunto de métodos de extensão sobre **IEnumerable<T>**
- ♦ Exemplos
 - Restrição: **Where**
 - Projecção: **Select, SelectMany**
 - Ordenação: **OrderBy, ThenBy**
 - Agrupamento: **GroupBy**
 - Quantificadores: **Any, All**
 - Partição: **Take, Skip, TakeWhile, SkipWhile**
 - Conjuntos: **Distinct, Union, Intersect, Except**
 - Elementos: **First, FirstOrDefault, ElementAt**
 - Agregação: **Count, Sum, Min, Max, Average**
 - Conversão: **ToArray, ToList, ToDictionary**

Exemplo

- ◆ Apresentar todos os ficheiros com extensão “.cs”, ordenados pela data/hora do último acesso de escrita

```
DirectoryInfo di = new DirectoryInfo(path);  
  
IEnumerable<FileInfo> fis = Global.GetDirectoryEnumerator(di)  
    .Where(fi => fi.Extension == ".cs")  
    .OrderBy(fi => fi.LastWriteTime);
```

```
foreach(FileInfo fi in fis){  
    Console.WriteLine(fi.FullName);  
}
```

Método que retorna um enumerável com representantes de todos os ficheiros presentes na sub-árvore de directorias com raiz em **di**

- ◆ Salienta-se
 - Métodos extensão – ordem do **Where** e do **OrderBy** é a ordem de execução
 - Expressões lambda – simplicidade do predicado e da selecção de ordenação

Tipificação implícita de variáveis locais

- ♦ Inferência do tipo das variáveis locais com base no tipo da sua iniciação
- ♦ Exemplos
 - **var** i = 5;
 - **var** s = new string("aaa");
 - **var** s = "aaa";
 - **var** point = {X = 3, Y = 4}
- ♦ Não é tipificação dinâmica, é tipificação estática com inferência do tipo em tempo de compilação
- ♦ Utilização
 - Tipos complexos
 - Tipos anónimos (ver slide seguinte)
- ♦ Só pode ser usada em variáveis locais
 - Não pode ser usada no tipo de retorno ou no tipo dos parâmetros


Tipos anónimos

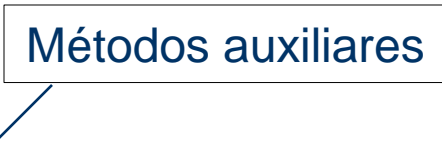
- ♦ Criação automática de tipos para o armazenamento de tuplos
 - `var point = new {X = 3, Y = 4};`
 - Resulta na criação duma classe anónima com as propriedades públicas **X** e **Y**, do tipo **int**, inferidas do iniciador **{X = 3, Y = 4}**
- ♦ Na mesma unidade de compilação, dois iniciadores iguais utilizam o mesmo tipo anónimo
- ♦ Forma alternativa

```
int X = 3;
int Y = 4
var point = new {X, Y}
```
- ♦ Tipo anónimo associado tem as propriedades **X** e **Y**

Exemplo

- ♦ Para cada ficheiro com extensão “.cs” apresentar o nome do ficheiro e o respectivo número de linhas

```
public static void ShowNumberOfLines(string path, string substring) {  
    DirectoryInfo di = new DirectoryInfo(path);  
    var res = GetDirectoryEnumerator(di)   
        .Where(fi => fi.Extension == ".cs")  
        .Select(fi => new { File = fi, Lines = CountLines(fi) });  
  
    foreach (var p in res) {  
        Console.WriteLine("file = {0}; lines = {1}",  
            p.File.FullName, p.Lines);  
    }  
}
```



O mesmo exemplo tirando partido da sintaxe Linq

- ◆ Para cada ficheiro com extensão “.cs” apresentar o nome do ficheiro e o respectivo número de linhas

```
public static void ShowNumberOfLines2(string path, string substring) {  
    DirectoryInfo di = new DirectoryInfo(path);  
    var res = from fi in GetDirectoryEnumerator(di)  
              where fi.Extension == ".cs"  
              select new { File = fi, Lines = CountLines(fi) };  
  
    foreach (var p in res) {  
        Console.WriteLine("file = {0}; lines = {1}",  
                           p.File.FullName, p.Lines);  
    }  
}
```