C# 2.0 e 3.0

Centro de Cálculo
Instituto Superior de Engenharia de Lisboa

Iteradores: sumário

- Enumeráveis e enumeradores
- Implementações *lazy*
- Corrotina
- Geração de enumeráveis e enumeradores
- Forma de implementação pelo compilador
- Exemplos de utilização

Enumeráveis e enumeradores

- **IEnumerable<T>** - tipo passível de ser enumerado
  - **IEnumerator<T> GetEnumerator()**
- **`IEnumerator<T> : IDisposable`** – tipo usado para enumerar
  - **`bool MoveNext()`**
  - **`T Current { get; }`**
- **`foreach(T t in en){ body }`** traduzido 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()**
    - Implementação de forma explícita
  - Método genérico **IEnumerator<T> GetEnumerator()**

- **IEnumerator<T>** : **IDisposable**, **IEnumerator**
  - Acrescenta a interface **IDisposable**
  - Métodos **Reset** e **MoveNext** são os de **IEnumerator**
    - Método **Reset** não necessita de ser suportado
  - Duas propriedades **Current**
    - Genérica, retorna **T**
    - Não genérica, retorna **object** – implementada de forma explícita

Utilizações de `IEnumerable/IEnumerator`

- Interface normalizada para o acesso sequencial a
  - Estruturas de dados *linearizáveis*
  - Qualquer objecto que represente uma sequência

- Implementações
  - 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)

- Enunciado: dado uma sequência **s1** e um predicado **p**, calcular a sequência **s2** com os elementos de **s1** que satisfazem **p**

- Solução 1
  - Criar uma estrutura de dados **s2** (ex. lista)
  - Percorrer **s1**, copiando para **s2** todos os elementos de **s1** que satisfazem **p**
  - Retornar **s2** como **IEnumerable<T>**

```
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
  - Retornar um objecto que implemente **IEnumerable<T>,** em que o enumerador associado:
    - Contém um enumerador para **s1**
    - Possui o método **MoveNext** que avança o enumerador sobre **s1** enquanto o predicado é falso
    - Contém a propriedade **Current**, que 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

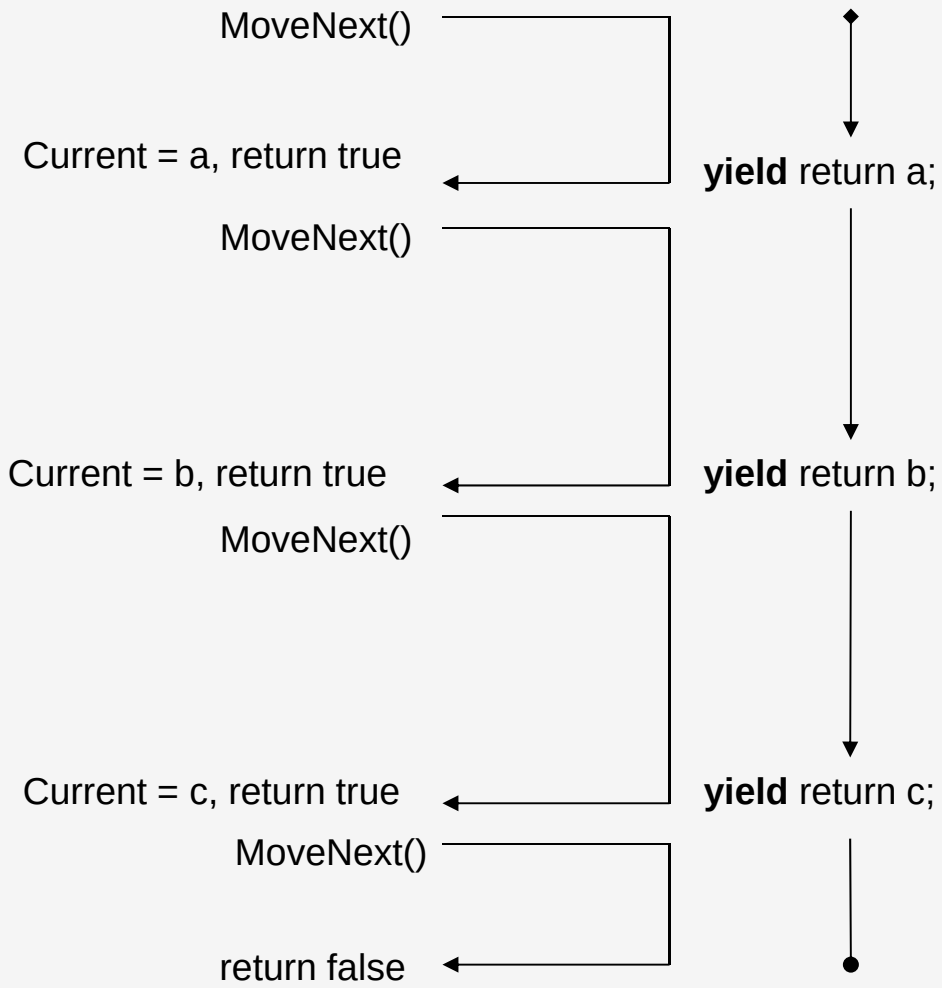
```
class FilterEnumerator : IEnumerator<T> {
    IEnumerator<T> enumerator;
    Predicate<T> pred;

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

    public void 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

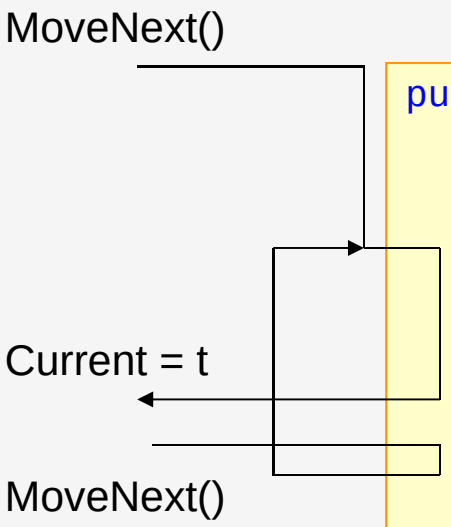
- Criação de duas classes
  - **Filter**: implementação de **IEnumerable<T>**
  - **FilterEnumerator**: implementação de **IEnumerator<T>**
- Passagem do predicado e do enumerador fonte entre a classe enumerável e a classe enumeradora
- Falta
  - Métodos das interfaces não genéricas
- Lógica do método **MoveNext**
- Complexo quando comparado 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


MoveNext()
Current = a, return true
yield return a;
MoveNext()
Current = b, return true
yield return b;
MoveNext()
Current = c, return true
yield return c;
MoveNext()
return false

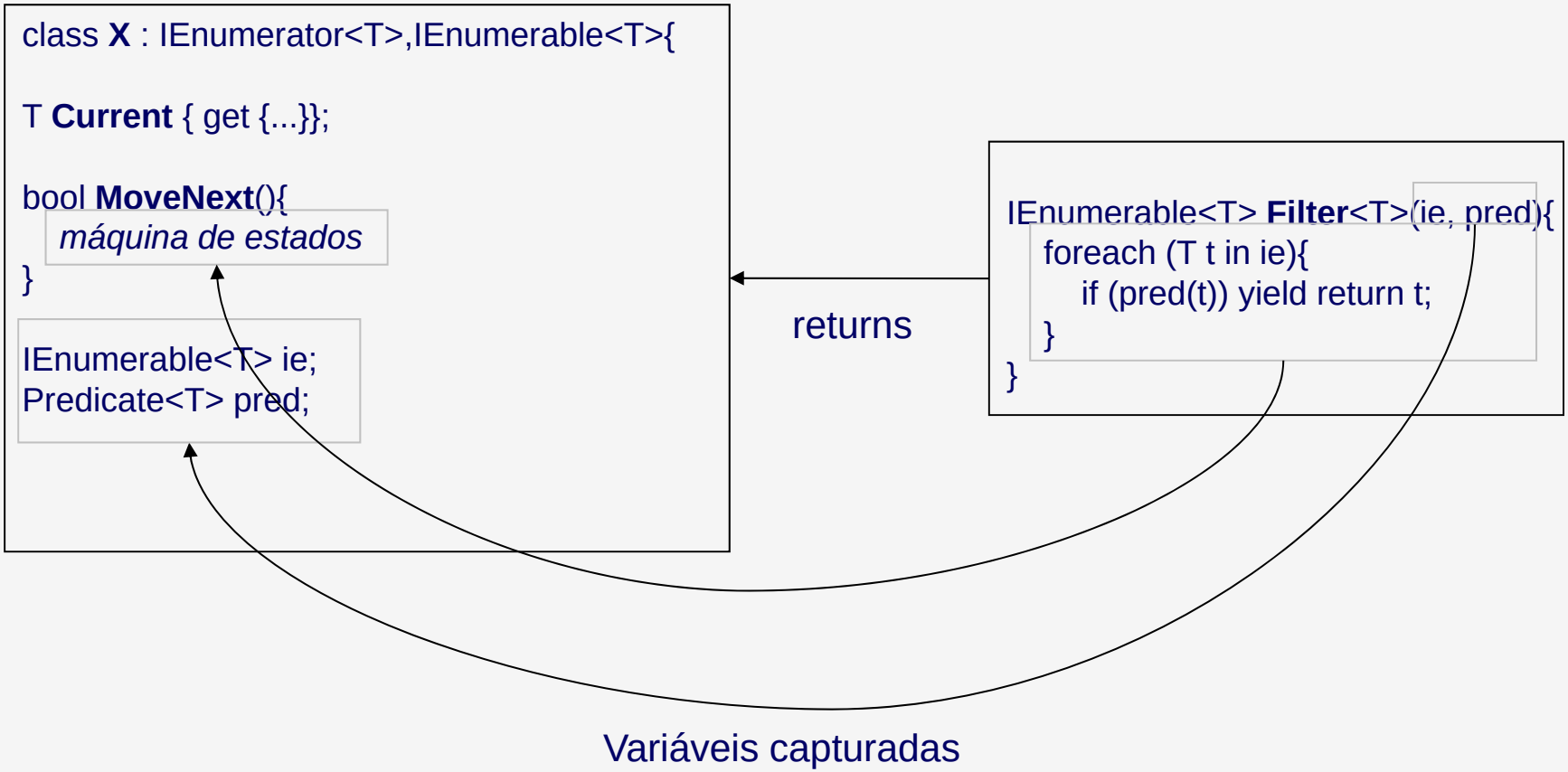
Utilização na definição de enumeradores




```
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: geradores de enumeradores

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



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)

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>**
    - Optimização para o caso em que apenas é criado um enumerador
  - 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 atómica (**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;
              state2: state = 1;
              next:;
         }
         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

Outro exemplo

```
public static IEnumerable<string>
      GetNameEnumerable(){

          yield return "Centro";
          yield return "Cálculo";
          yield return "Instituto";
          yield return "Superior";
          yield return "Engenharia";
          yield return "Lisboa";

}
```

Código gerado

```
private bool MoveNext(){
    switch (state){
        case 0: {
            state = -1; current = "Centro";
            state = 1; return true;
        }
        case 1:  {
            state = -1; current = "Cálculo";
            state = 2; return true;
        }
        …
        case 5: {
            state = -1; current = "Lisboa";
            state = 6; return true;
        }
        case 6: {
            state = -1; break;
        }
    }
    return false;
}
```

Exemplos

- 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);
}

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

Mais exemplos

- Ordenação usando o algoritmo *Selection Sort*
  - Por cada **MoveNext** é obtido o menor dos elementos restantes

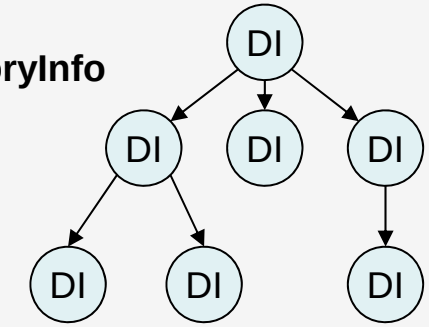
```
public static IEnumerable<T> OrderBy<T,U>(IEnumerable<T> seq, Converter<T,U> sortkey)
                                                        where U : IComparable<U>
{
    List<T> list = new List<T>(seq);
    for(int begin = 0; begin<list.Count ; ++begin){
        int minix = begin;
        for(int i = begin+1 ; i<list.Count ; ++i){
            if(sortkey(list[i]).CompareTo(sortkey(list[minix])) < 0){minix = i;}
        }
        if(minix != begin){
            T min = list[minix];
            list[minix] = list[begin];
            list[begin] = min;
        }
        yield return list[begin];
    }
}
```

Exemplo: enumeração de estruturas recursivas

- Realizar uma operação sobre todos os ficheiros presentes numa sub-árvore de directorias do sistema de ficheiros

```
public static void ForeachFileInfo(DirectoryInfo di, Action<FileInfo> action){
    foreach (FileInfo fi in di.GetFiles()) {
        action(fi);
    }
    foreach (DirectoryInfo childDi in di.GetDirectories()) {
        ForeachFileInfo(childDi,action);
    }
}
```

Árvore de **DirectoryInfo**

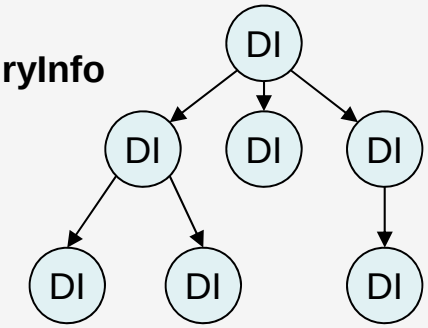


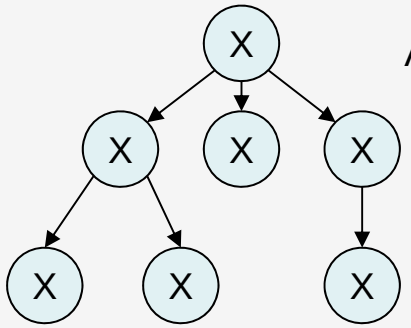
Exemplo: enumeração de estruturas recursivas

- Enumerar todos os ficheiros presentes numa sub-árvore de directorias do sistema de ficheiros

```
public static IEnumerable<FileInfo> GetDirectoryEnumerator(DirectoryInfo di) {
   foreach (FileInfo fi in di.GetFiles()) {
               yield return fi;
   }
   foreach (DirectoryInfo childDi in di.GetDirectories()) {
       foreach (FileInfo fi in GetDirectoryEnumerator(childDi)) yield return fi;
   }
}
```

Árvore de **DirectoryInfo**





Árvore de enumeráveis

Enumeração de estruturas recursivas: optimização

- Evitar a criação de novos enumeradores mantendo uma pilha com as directorias que falta visitar

```
public static IEnumerable<FileInfo> GetDirectoryEnumerator2(DirectoryInfo di) {
   Stack<DirectoryInfo> stack = new Stack<DirectoryInfo>();
   Stack<DirectoryInfo> aux = new Stack<DirectoryInfo>();
   stack.Push(di);
   while (stack.Count != 0) {
      DirectoryInfo cdi = stack.Pop();
      foreach (FileInfo fi in cdi.GetFiles()) {
         yield return fi;
      }
      foreach (DirectoryInfo d in cdi.GetDirectories())
      {
         aux.Push(d);
      }
      while (aux.Count != 0) { stack.Push(aux.Pop()); }
   }
}
```

Exemplo: *tokenize*

```
public static IEnumerable<string> Tokenize(IEnumerable<char> seq,
                                           Predicate<char> pred)  {
   using (IEnumerator<char> ienum = seq.GetEnumerator())
   {
       bool res = true;
       while (res)
       {
           while ((res = ienum.MoveNext()) && !pred(ienum.Current));
           if (res == false) yield break;

           StringBuilder token = new StringBuilder();
           token.Append(ienum.Current);
           while ((res = ienum.MoveNext()) && pred(ienum.Current))
           {
                token.Append(ienum.Current);
           }
           yield return token.ToString();
       }
   }
}
```

Métodos anónimos: sumário

- Motivação
- Forma de implementação
- Exemplos de utilização

Métodos anónimos: motivação

- Enunciado: dado uma sub-árvore e uma extensão, listar todos os ficheiros pertencentes a essa sub-árvore e que tenham extensão igual

```
public static void ListFilesWithExt(string path, string ext) {
        foreach (FileInfo fi in Filter(
                GetDirectoryEnumerator(new DirectoryInfo(path)), ? )) {
               Console.WriteLine(fi.FullName);
        }
}
```

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

- Problema: definição do predicado

Definição do predicado: classe auxiliar

- Classe ExtensionComparer
  - Campo **ext** com a extensão a comparar
  - Construtor para a iniciação do campo
  - Método para comparação da extensão do **FileInfo** passado como parâmetro com a *string* **ext**

```
public class ExtensionComparer {
   string ext;
   public ExtensionComparer(string _ext) { ext = _ext; }
   public bool Compare(FileInfo fi) { return fi.Extension == ext; }
}
```

Utilização de métodos anónimos

```
public static void ListFilesByExt(string path, string ext)
{
     foreach (FileInfo fi in
        Filter(GetDirectoryEnumerator(new DirectoryInfo(path)),
               new Predicate<FileInfo>(new ExtensionComparer(ext).Compare)))
     {
          Console.WriteLine(fi.FullName);
     }
}

public static void ListFilesByExt(string path, string ext)  {
     foreach (FileInfo fi in
          Filter(GetDirectoryEnumerator(new DirectoryInfo(path)),
                 delegate(FileInfo x) { return x.Extension == ext; }))
     {
          Console.WriteLine(fi.FullName);
     }
}
```

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 automaticamente

```
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())
```

Inferência da assinatura

- A assinatura do método anónimo é função do tipo *delegate*
- O tipo *delegate* é inferido do contexto

```
void m(Predicate<int> p);
m(delegate(int x) { return x > 0; })

Predicate<int> p = delegate(int x) { return x>0;};

Delegate d = delegate(int x) { return x > 0;};
```

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.
- 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.

Exemplo

```
class ContextExample  {
        public int aField;
        public Action AMethod(int aParam)
        {
            long aLocal;
            aLocal = DateTime.Now.Millisecond;
            return delegate
            {
                Console.WriteLine("aField = {0}, aParam = {1}, aLocal = {2}", aField, aParam, aLocal);
                aLocal += 1; aField += 1;
            };
        }
    }
    public static void ContextTest()
    {
        ContextExample ce1 = new ContextExample();
        ce1.aField = 1;
        Action mi1 = ce1.AMethod(2);
        mi1(); // aField = 1, aParam = 2, aLocal = 87
        ce1.aField = 3;
        mi1(); // aField = 3, aParam = 2, aLocal = 88
        Action mi2 = ce1.AMethod(20);
        mi2(); // aField = 4, aParam = 20, aLocal = 94
        mi1(); // aField = 5, aParam = 2, aLocal = 89
    }
}
```

Exemplo: implementação

```
public Action AMethod(int aParam){
    X class1 = new X();
    class1.aParam = aParam;
    class1._this = this;
    class1.aLocal = DateTime.Now.Millisecond;
    return new Action(class1.m);
}

private sealed class X{
    public ContextExample _this;
    public long aLocal;
    public int aParam;
    public void m(){
        Console.WriteLine("…", this._this.aField, this.aParam, this.aLocal);
        this.aLocal++;
        this._this.aField++;
    }
}
```

Limitações na captura de variáveis

- Parâmetros referência (**ref** e **out**) não podem ser capturados
- Não é possível garantir a validade das referências no momento da execução do *delegate*

Exemplo: **OrderBy**

- Método **OrderBy**: enumera de forma ordenada
- Ordenação segundo a chave produzida pela *função* **sortkey**
- Para usar o método **Sort** de **List<T>** é necessário criar uma *função* que compara as chaves de cada elemento
  - public void Sort (Comparison<T> comparison)

```
public static IEnumerable<T> OrderBy<T,U>(IEnumerable<T> seq, Converter<T,U> sortkey)
                                                                    where U : IComparable<U>{
    List<T> list = new List<T>(seq);
    list.Sort( delegate(T t1, T t2) {return sortkey(t1).CompareTo(sortkey(t2));} );
    return list;
}
```

Exemplo: derivada

- Utilização dos métodos anónimos para a definição de funções de ordem superior
- Exemplo:
  - função **Derivative** recebe um função real e retorna a sua derivada (que também é uma função real)

```
public delegate double RealFunction(double x);

public static RealFunction Derivative(RealFunction f, double dx){
    return delegate(double x){
        return (f(x+dx)-f(x))/dx;
    };
}
```

Exemplo: conjunção de predicados

- Note-se que o retorno do método é um *delegate*

```
public static Predicate<T> And<T>(params Predicate<T>[] preds) {
    return delegate(T t){
        foreach (Predicate<T> p in preds) {
            if (p != null && p(t) == false) return false;
        }
        return true;
    };
}
```

Exemplo: negação

- Exemplo: Negação dum predicado

```
public static Predicate<T> Not<T>(Predicate<T> p){
    return delegate(T t){ return p(t) == false;  };
}
```

- Exemplo de utilização
  - Partição duma sequência

```
public static Pair<IEnumerable<T>,IEnumerable<T>> Partition<T>(
                                IEnumerable<T> seq, Predicate<T> pred){
    return MakePair(
        Filter(seq,pred),
        Filter(seq,Not(pred))
    );
}
```

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

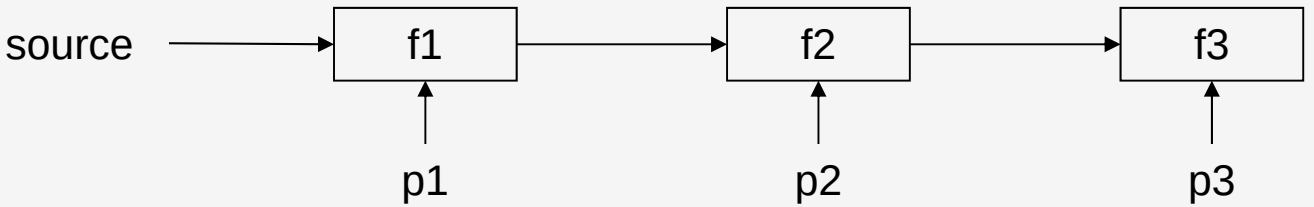
Expressões lambda

- Método anónimo (C# 2.0): **delegate (int x){ return x>y;}**
  - Tipificação explícita do parâmetro
  - Corpo do método é um *statement*
  - Sintaxe prolixa e imperativa

- Expressão lambda: **x => x>y**
  - Tipificação implícita do parâmetro, obtida através do contexto
  - Corpo do método é uma expressão
  - Sintaxe concisa e funcional

Métodos 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
  - IEnumerable<T> f1(**this** IEnumerable<T>, P1 p1)



System.Query

- O *assembly* **System.Query** define um conjunto de métodos 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);
}
```

- 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

Tipos anónimos

- Criação automática de tipos para o armazenamento de tuplos
  - 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

  Point = new {X, Y}
- Tipo anónimo associado tem as propriedades **X** e **Y**

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
- Só pode ser usada em variáveis locais
  - Não pode ser usada no tipo de retorno ou no tipo dos parâmetros

Exemplo

- Procurar **substring** em todos os ficheiros com extensão “.cs”, e para cada ocorrência apresentar o nome do ficheiro e a linha onde se encontra **substring**

```
public static void ShowAllWithString(string path, string substring){
    DirectoryInfo di = new DirectoryInfo(path);
    var res = Global.GetDirectoryEnumerator(di)
        .Where(fi => fi.Extension == ".cs")
        .SelectMany(fi => GetLines(fi).Select(s => new {File = fi, Line = s}))
        .Where( p => p.Line.Contains(substring));

    foreach(var p in res){
        Console.WriteLine("file = {0}; line = {1}",p.File.FullName, p.Line);
    }
}
```

Query expressions - LINQ

- Query LINQ (Language Integrated Query):

```
// Declaration and initialization of an array of anonymous types
var customers = new []{
                new { Name = "Marco", Discount = 4.5 },
                new { Name = "Paolo", Discount = 3.0 },
                new { Name = "Tom", Discount = 3.5 }
};

var query = from c in customers
                where c.Discount > 3
                orderby c.Discount
                select new { c.Name, Perc = c.Discount / 100 };

foreach (var x in query) {
        Console.WriteLine(x);
}
```

Query expressions - LINQ

- A *query expression* começa com a cláusula **from** e termina com uma cláusula **select** ou **group**
  - A cláusula **from** especifica o objecto sobre o qual as operações LINQ são efectuadas
    - Este deve ser uma instância de uma classe que implementa IEnumerable<T>
- O código anterior produz o seguinte resultado:

```
{ Name = Tom, Perc = 0,035 }
{ Name = Marco, Perc = 0,045 }
```

- No C# 3.0, o *query* anterior é interpretado como se fosse escrito na forma:

```
var query = customers
                .Where( c => c.Discount > 3)
                .OrderBy( c => c.Discount)
                .Select( c => new { c.Name, Perc = c.Discount / 100 } );
```