
Iteradores e Métodos anónimos

Agenda da sessão

- Iteradores
- Métodos anónimos
- Exemplos de utilização

Agenda da sessão

- Iteradores
 - Enumeráveis e enumeradores
 - Implementações lazy
 - Corrotina
 - Geração de enumeráveis e enumeradores
 - Forma de implementação pelo compilador (....)
 - Exemplos de utilização
- Métodos anónimos
- Exemplos de utilização

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:

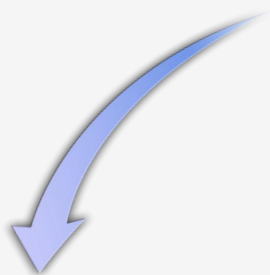
```
using (IEnumerator<T> enumerator1 = en.GetEnumerator()) {  
    while (enumerator1.MoveNext()) {  
        T t = enumerator1.Current;  
        //body  
    }  
}
```

Interfaces genéricas e não genéricas

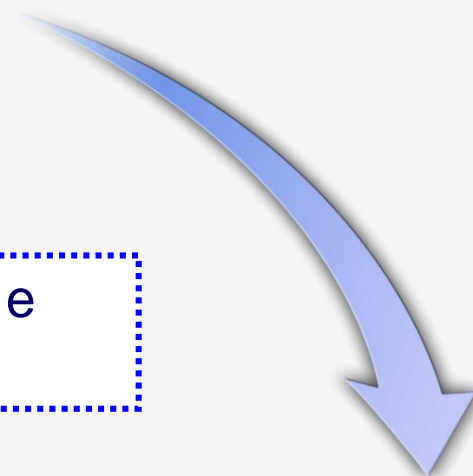
- **IEnumerable<T> : IEnumerable**
 - Método não genérico **IEnumerator GetEnumerator()**, com implementação de forma explícita
 - Método genérico **IEnumerator<T> GetEnumerator()**
- **IEnumerator<T> : IDisposable, IEnumerator**
 - Acrescenta a interface **IDisposable**
 - Métodos **Reset** e **MoveNext** são de **IEnumerator**, embora o método **Reset** não necessita 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;
    }

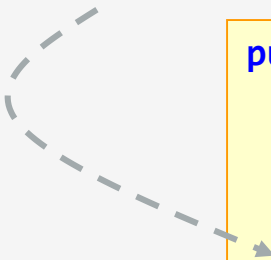
    public IEnumerator<T> GetEnumerator()
    {
        return new FilterEnumerator enumerable.GetEnumerator(), pred);
    }
}
```

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

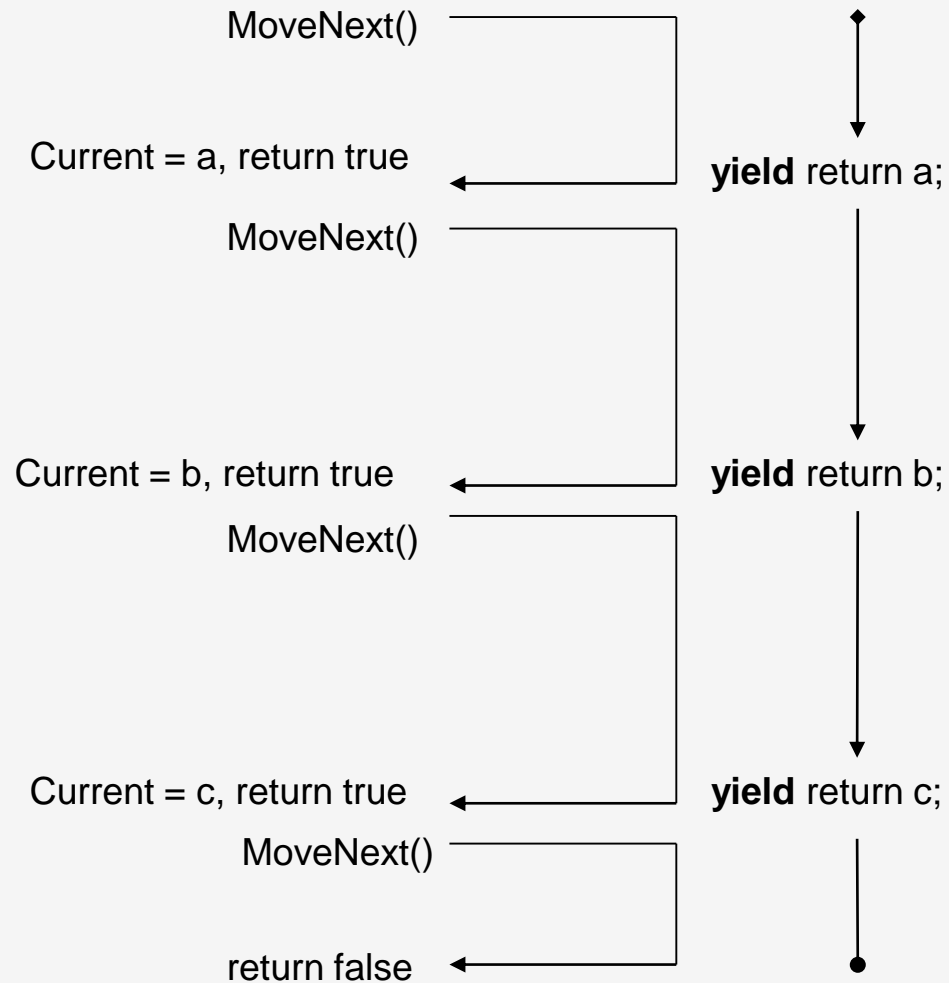
- A segunda solução implica a criação de duas classes
 - **Filter**: implementação de **IEnumerable<T>**
 - **FilterEnumerator**: implementação de **IEnumerator<T>**
- É necessário passar o predicado e o enumerador entre a fonte e a classe enumerável, e entre esta e a classe enumeradora
- Faltam ainda 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



Corrotinas



```
static IEnumerable<int> OneTwoThree()
{
    Console.WriteLine("Returning 1");
    yield return 1;

    Console.WriteLine("Returning 2");
    yield return 2;

    Console.WriteLine("Returning 3");
    yield return 3;
}
```

```
static void Main(string[] args)
{
    foreach (int number in OneTwoThree())
    {
        Console.WriteLine("Number: {0}", number);
    }
}
```

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

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

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

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

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

```
IEnumerable<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:
 - Contém 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 fonte  
    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() {  
    try {  
        switch (state) {  
            case 0: 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(); }  
                    break;  
            case 2: goto state2;  
        }  
        return false;  
    }  
    fault { ((IDisposable) this).Dispose(); } }
```

Algoritmo presente na
função construtora

-> Saltar para o estado anterior

Exemplos: 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;
}
```

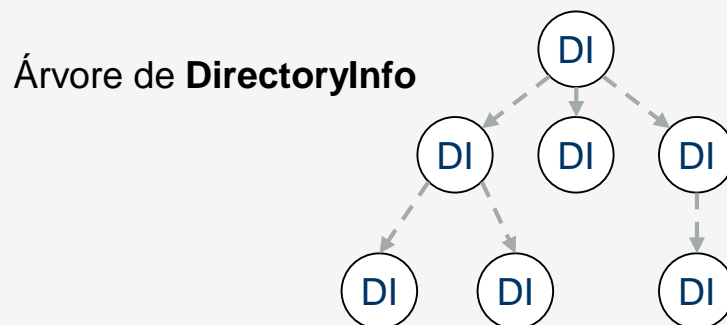
- Projecção de sequências

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

```
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; }
    }
}
```

Exemplos: enumeração de estruturas recursivas

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



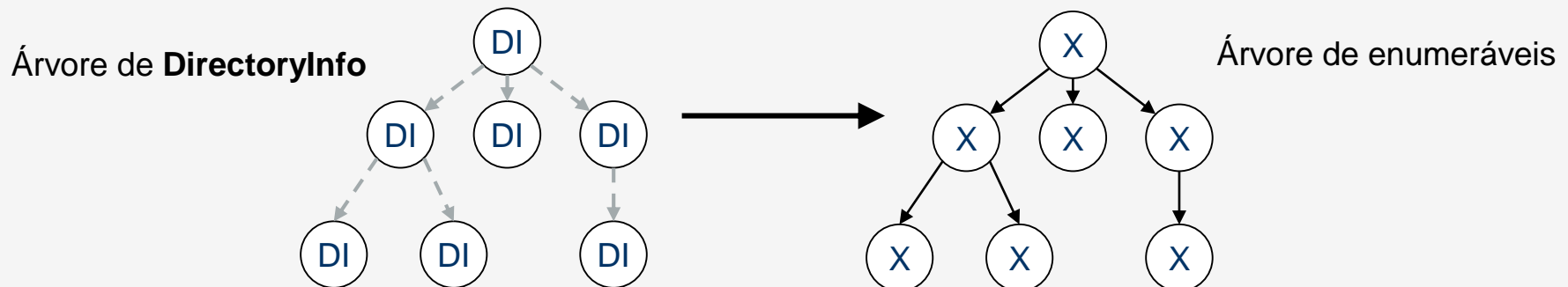
```
public static void ForeachFileInfo(DirectoryInfo di,
                                   Action<FileInfo> action)
{
    foreach (FileInfo fi in di.GetFiles())
        action(fi);

    foreach (DirectoryInfo childDi in di.GetDirectories())
        ForeachFileInfo(childDi, action);
}
```

Exemplos: enumeração de estruturas recursivas (cont.)

- 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;  
}
```



Agenda da sessão

- Iteradores
- Métodos anónimos
- Exemplos de utilização

Agenda da sessão

- Iteradores
 - Motivação
 - Forma de implementação
 - Exemplos de utilização
- Métodos anónimos
- Exemplos de utilização

Métodos anónimos: motivação

- Dada 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 com recurso a uma 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

Sem 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);
    }
}
```

Com métodos anónimos

```
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);
    }
}
```

Deixa de ser necessária a classe auxiliar

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())
```

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 anônimo consiste na criação de uma instância da classe referida acima e na captura do contexto.
- A implementação dos métodos anônimos não introduziu alterações na CIL nem na CLI.
- Limitação: Parâmetros referência (ref e out) não podem ser capturados

Exemplo

```
class ContextExample
{
    public int aField;
    public Action AMethod(int aParam)
    {
        long 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 = x
    ce1.aField = 3;
    mi1(); // aField = 3, aParam = 2, aLocal = x +1
    Action mi2 = ce1.AMethod(20);
    mi2(); // aField = 4, aParam = 20, aLocal = y
    mi1(); // aField = 5, aParam = 2, aLocal = x + 2
}
```


Exemplo: código gerado

```
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++;
    }
}
```

Exemplo: OrderBy

- O método **OrderBy** enumera de forma ordenada uma sequência, segundo a chave produzida pela *função sortkey*
- Para recorrer a uma instância de **List<T>** e usar o método `public void Sort(Comparison<T> comparison)` é necessário criar uma *função* que compara as chaves de cada elemento

```
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: negação e partição

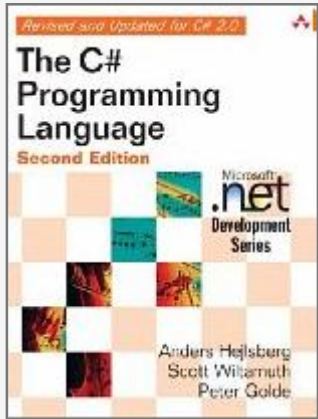
- Negação dum predicado

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

- Partição de uma 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))  
    );  
}
```

Bibliografia



Anders Hejlsberg, Scott Wiltamuth, Peter Golde
“The C# Programming Language, (2nd Edition)” ,
Addison-Wesley Professional, 2006