

Genéricos

ISEL/LEIC - Verão 2007/2008

2.º ano, 2.º Semestre

(Adaptado de acetatos
da autoria de Pedro Félix e João Trindade)

Nuno Leite

AVE: Ambientes Virtuais de Execução - Semestre de Verão 2007/08

<http://www.deetc.isel.ipl.pt/programacao/ave/>

Sumário

- Genéricos
- Interfaces e colecções genéricas
- Constrangimentos (*constraints*)
- Implementação de genéricos
- Tipos abertos e fechados
- Métodos genéricos
- Tipos *nullable*

Sem genéricos

```
public interface IBusinessStats {  
    // ...  
    Customer[] Top10Customers(IList customers);  
    Supplier[] Top10Suppliers(IList suppliers);  
    // ...  
}
```

- Limitações de expressividade, robustez e desempenho
 - A assinatura dos métodos especifica apenas uma lista
 - *Pressupõe-se* que os métodos serão invocados com listas homogêneas (de clientes ou de fornecedores)
 - Verificação de tipos em tempo de execução
- Limitações ao polimorfismo
 - Os dois métodos têm de ter nomes diferentes porque têm listas de parâmetros iguais

Com genéricos

```
public interface IBusinessStats {  
    // ...  
    Customer[] Top10(ICollection<Customer> customers);  
    Supplier[] Top10(ICollection<Supplier> suppliers);  
    // ...  
}
```

- As assinaturas dos métodos especificam o tipo dos elementos das listas
- Verificação em tempo de compilação
- Não existem pressupostos escondidos
- As listas são garantidamente homogéneas
- (e os dois métodos já podem ter o mesmo nome)

Exemplo: *Stack* não genérico

```
public class Stack {  
    int sp;  
    object[] items;  
    public Stack(int dim) { items = new object[dim]; sp = 0; }  
    public void Push(object item) { items[sp++] = item; }  
    public object Pop() { return items[--sp]; }  
}
```

// Main:

```
Stack strStack = new Stack(); // Stack de strings não se  
Stack intStack = new Stack(); // distingue de Stack de ints
```

```
string s; int i;
```

```
strStack.Push("X");
```

```
intStack.Push(8); // box
```

```
s = (string)strStack.Pop(); // cast
```

```
i = (int)intStack.Pop(); // unbox
```

```
intStack.Push("8"); // string por int
```

```
i = (int)intStack.Pop(); // exceção!
```

- Expressividade

- Desempenho

- Robustez

Exemplo: *Stack* genérico

```
public class Stack<T> {  
    int sp;  
    T[] items;  
    public Stack(int dim) { items = new T[dim]; sp = 0; }  
    public void Push(T item) { items[sp++] = item; }  
    public T Pop() { return items[--sp]; }  
}
```

// Main:

```
Stack<string> strStack = new Stack<string>(100); // Stacks de tipos distintos  
Stack<int> intStack = new Stack<int>(100);      + Expressividade  
string s;  
int i;  
strStack.Push("X");  
intStack.Push(8);          // sem box          + Desempenho  
s = strStack.Pop();        // sem cast  
i = intStack.Pop();        // sem unbox  
//intStack.Push("8");      // não compila!    + Robustez
```

Genéricos \neq *Templates* (1)

- *Templates* do C++
 - Não originam directamente código intermédio nem nativo
 - por cada instanciação de um *template*, a sua definição (presente no ficheiro *header*) é combinada com a dos tipos-parâmetro para gerar código nativo específico
 - Cada instanciação do *template* com um tipo diferente é compilada separadamente existindo uma versão do código para cada tipo: expansão de código (“code expansion”)
 - Os *templates* têm de ser programados em ficheiros *header* (.h) para que o compilador conheça as interfaces dos tipos-parâmetro e assim gerar código **em tempo de compilação**
 - Assim, bibliotecas genéricas em C++ não podem ser distribuídas já compiladas

Genéricos \neq *Templates* (2)

- Distribuição de bibliotecas em .NET
 - Na FCL .NET, os tipos genéricos são fornecidos já compilados em código intermédio
- Genéricos .NET
 - O código genérico é compilado para IL, que fica com informação genérica de tipos
 - representação intermédia ainda é genérica
 - genérico é usável na forma compilada CIL
 - O compilador não conhece a interface dos tipos que vão ser usados na instanciação do genérico
 - limita as acções realizáveis sobre objectos dos tipos-parâmetro

Genéricos \neq *Templates* (3)

- Templates C++
 - Verificação e instanciação em tempo de compilação na utilização
 - Expansão de código

- Genéricos Java
 - Verificação em tempo de compilação na declaração
 - Uma única instanciação (*type erasure*)
 - Partilha de código

Genéricos \neq *Templates* (4)

- Genéricos .NET
 - Verificação em tempo de compilação na declaração
 - *Dynamic Code Expansion and Sharing*
 - partilha de código intermédio
 - expansão de código nativo à medida
 - Instanciação em tempo de execução (JIT)
 - partilha de código nativo quando argumentos são tipos referência

Genéricos \neq *Templates* (5)

- No construtor de Stack:

```
public Stack(int dim) {  
    T aux = new T(); // Erro em tempo de compilação porque T  
                    // só é realmente construído em tempo de  
                    // execução; por isso, em tempo de  
                    // compilação o compilador não sabe se T  
                    // tem construtor público sem parâmetros
```

```
public class Stack<T> where T : new() {  
    // Informar o compilador de que T tem  
    // construtor sem parâmetros  
}
```

Constrangimentos (*constraints*)

- Tipos-parâmetro podem ter constrangimentos

```
public class SingletonFactory<T>
```

```
  where T: class, new() {
```

```
    private T instance = null;
```

```
    ...
```

```
  }
```

Restrições:

- tipo T tem que ter construtor sem parâmetros

- tipo T tem que ser um tipo referência (neste caso para garantir que a afectação corre bem)

Constrangimentos (*constraints*)

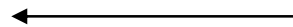
- As quatro restrições existentes são:
 - Classe base
 - Interfaces
 - Tipo referência ou tipo valor
 - Construtor sem parâmetros
- Vários tipos num genérico podem ter restrições diferentes:

```
public class SingletonFactory<T, U>  
  where T: MyBaseClass, new()  
  where U: struct {  
    ...  
  }
```

Métodos genéricos

- Não só as classes mas também os métodos podem ser genéricos
public delegate Boolean MyPredicate<T>(T item);

```
public static int CountIf<T>(T[] a, MyPredicate<T> p)
{
    int count = 0;
    foreach (T i in a)
    {
        if (p(i))
            ++count;
    }
    return count;
}
```

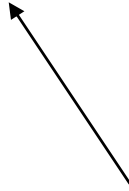


Count genérico para arrays
genéricos

Métodos genéricos

```
public static int CountIf<T>(IEnumerable<T> a, MyPredicate<T> p)
{
    // Mesmo código que o CountIf anterior
}
```

Count genérico para
sequências (arrays, listas, etc.)



Campos estáticos

- A definição de uma classe genérica representa um conjunto de tipos
 - cada um desses tipos é uma instância da classe genérica
 - `Stack<T> : { Stack<int>, Stack<string>, ... }`
- Cada instância de uma classe genérica tem um conjunto próprio de campos estáticos
- O construtor estático é chamado para cada instanciação da classe genérica

Campos estáticos

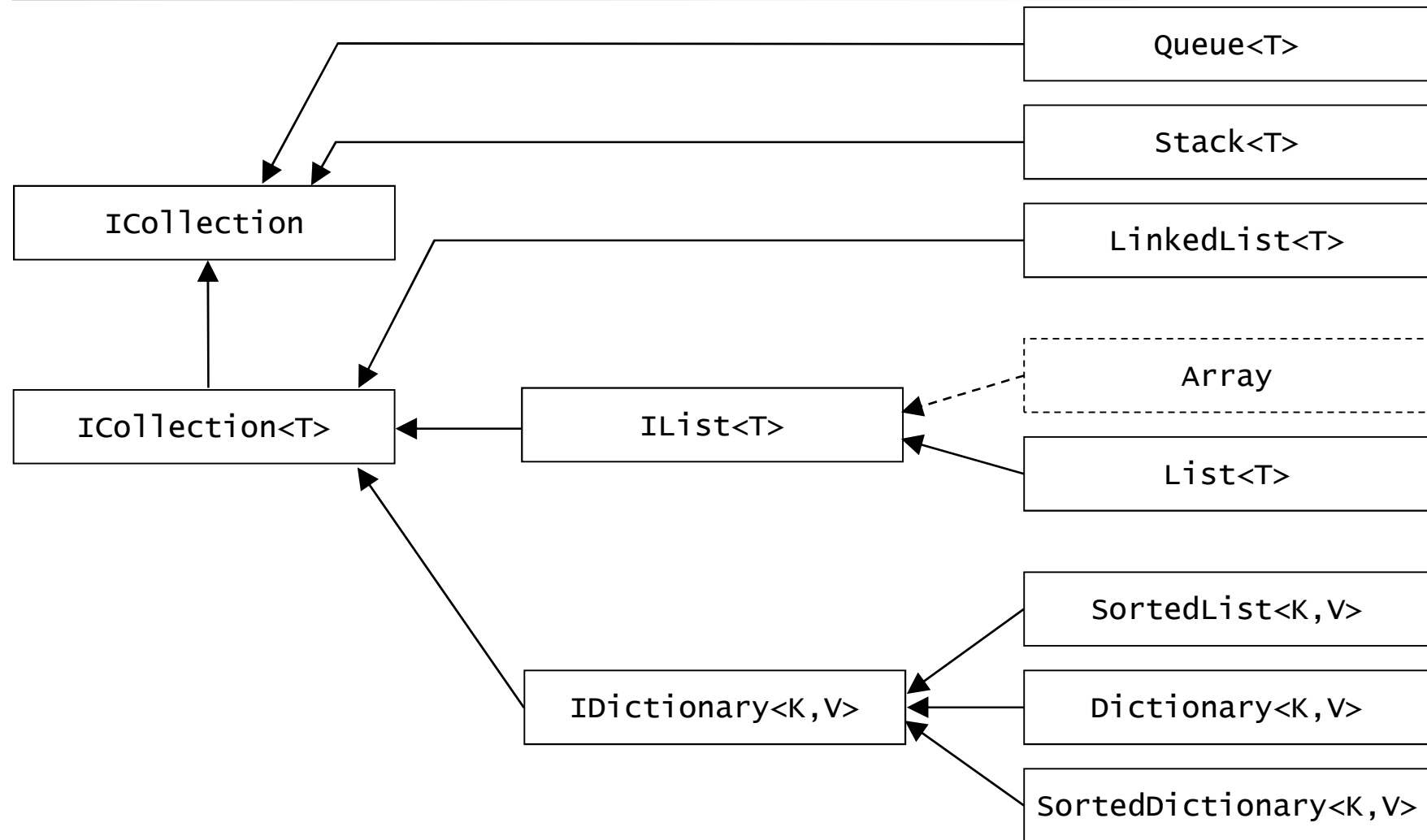
```
public static class Singleton<T> where T : new() {  
    static Singleton() {}  
    static public readonly T Instance = new T();  
}  
  
public static partial class Examples {  
    public static void SingletonExample() {  
        Singleton<StringBuilder>.Instance.Append("a");  
        Singleton<StringBuilder>.Instance.Append("b");  
        Console.WriteLine(Singleton<StringBuilder>.Instance); //“ab”  
    }  
}
```

System.Collections.Generic

- Novas versões genéricas de colecções
(implementam `ICollection<T>`)

<code>Queue<T></code>	Versão genérica de <code>Queue</code> (FIFO)
<code>Stack<T></code>	Versão genérica de <code>Stack</code> (LIFO)
<code>List<T></code>	Versão genérica de <code>ArrayList</code> (lista sobre <i>array</i>)
<code>LinkedList<T></code>	Lista duplamente ligada
<code>SortedList<K,V></code>	Versão genérica de <code>SortedList</code> sobre dois <i>arrays</i> (colecção ordenada de pares chave/valor)
<code>Dictionary<K,V></code>	Versão genérica de <code>HashTable</code> (tabela associativa de pares chave/valor)
<code>SortedDictionary<K,V></code>	Outra versão genérica de <code>SortedList</code> (colecção ordenada de pares chave/valor)

System.Collections.Generic



----- Ligação em tempo de execução

Retrocompatibilidade

- Versões genéricas de IEnumerable e de IEnumerator estendem as versões anteriores (não genéricas)

```
public interface IEnumerable<T> : IEnumerable
```

```
public interface IEnumerator<T> : IEnumerator, IDisposable
```

- Colecções genéricas implementam interfaces genéricas e não-genéricas

```
public class List<T> :
```

```
    IList<T>, ICollection<T>, IEnumerable<T>,
```

```
    IList, ICollection, IEnumerable
```

Suporte para genéricos na CLI

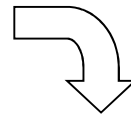
- Genéricos suportados directamente pela CLI
 - Interoperabilidade entre genéricos ao nível da plataforma
- Modificações
 - Suporte para tipos e métodos genéricos
 - Novos prefixos
 - Novas instruções
 - Alteração da semântica de instruções

Tipos genéricos na CLS

- Nomes de tipos genéricos têm o formato “*nome`aridade*”, em que
 - *nome* é o nome da classe genérica
 - *aridade* é o número de tipos-parâmetro declarados pela classe
- Nos tipos internos, a lista de tipos-parâmetro inclui os tipos-parâmetro do tipo externo

Tipos genéricos na CLS

```
public class A<T> {  
    public class B {}  
    public class C<U, V> {  
        public class D<W> {}  
    }  
}  
public class X {  
    public class Y<T> {}  
}
```



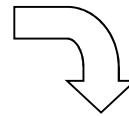
```
.class ... A`1<T> ... {  
    .class ... nested ... B<T> ... {}  
    .class ... nested ... C`2<T,U,V> ... {  
        .class ... nested ... D`1<T,U,V,W> ... {}  
    }  
}  
.class ... X ... {  
    .class ... nested ... Y`1<T> ... {}  
}
```

Indicação de restrições

- Restrições indicadas com o formato “[valuetype | class] [.ctor] [(*C1*, ..., *Cn*)] *T*”, em que
 - *T*: um tipo-parâmetro
 - **valuetype**: restrição de tipo-valor (C# `struct`)
 - **class**: restrição de tipo-referência (C# `class`)
 - **.ctor**: restrição de construtor sem parâmetros (C# `new()`)
 - (*C1*, ..., *Cn*) é a lista de restrições de classe base e de interfaces implementadas

Indicação de restrições

```
public class X<U,V>
  where U : SomeBaseClass, ISomeInterface
  where V : class, ISomeInterface, new() {
    ...
  }
```



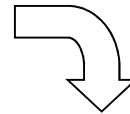
```
.class ... X`2<(SomeBaseClass, ISomeInterface) U,
  class .ctor (ISomeInterface) V> {
    ...
  }
```

Referência aos tipos-parâmetro

- Dentro da definição de um tipo genérico, os tipos parâmetro são referidos por índice ou por nome
 - quando referidos por índice, o primeiro tipo-parâmetro é referido por !0, o segundo por !1, etc.
 - as referências por índice para os tipos-parâmetros de métodos são !!0, !!1, etc.

Referência aos tipos-parâmetro

```
public class H<U,V> where V : class
{
    public bool Op<W>(W w) {
        return w is V;
    }
}
```



```
.class ... H`2<U,class V> ... {
    .method ... bool Op<W>(!!0 w) ... {
        .maxstack 8
        IL_0000: ldarg.1
        IL_0001: box
        IL_0006: isinst
        IL_000b: ldnull
        IL_000c: cgt.un
        IL_000e: ret
    }
}
```

Diagram illustrating the mapping of C# code to IL code:

- The parameter **W** is mapped to the IL instruction **box** (IL_0001).
- The type parameter **V** is mapped to the IL instruction **isinst** (IL_0006).

Alterações à CIL

- Novos prefixos

`constrained. T`

- aplicado à instrução `callvirt` para que esta possa ser usada uniformemente com tipos-referência e com tipos valor

`readonly.`

- aplicado à instrução `ldema` para que esta não faça verificação de tipos e retorne um *controlled-mutability managed pointer*

- Novas instruções

`ldem T / stem T`

- podem lidar com qualquer tipo `T`

- Passam a suportar tipos referência

`box`, `initobj`, `ldobj`, `stobj`, `cpobj`

Tipos anuláveis (1)

- Objectivo
 - Suportar tipos-valor nulos (sem valor atribuído)
- Instâncias de tipos-referência podem não ter objecto associado (valor nulo)
- Instâncias de tipos-valor têm sempre valor não nulo
- Pode ser necessário indicar que uma instância de um tipo-valor não contém um valor válido (ex.: campos NULL de uma base de dados)

Tipos anuláveis (2)

- No *namespace* `System` está definido o tipo genérico `Nullable<T>`
`public struct Nullable<T> where T : struct`
- `Nullable<T>` tem duas propriedades:
`HasValue : bool`
`Value : T`
 - Se `HasValue` vale `true`, então `Value` é um objecto válido.
 - Caso contrário, `Value` está indefinido e uma tentativa de acesso à propriedade resulta numa excepção (`InvalidOperationException`).

Tipos anuláveis (3)

- O C# 2.0 admite uma notação abreviada para os tipos anuláveis

- Modificador ? para declarar um tipo como anulável.

```
int? i = 0; // inteiro anulável com valor zero  
i = null; // passa a conter valor null
```

- Operador ?? (“coalesce”) para indicar o valor pré-definido numa atribuição de uma instância de um tipo anulável a um não-anulável.

$(a ?? 0)$ *equivalente a* $(a.HasValue ? a.Value : 0)$

- Comparação com null verifica HasValue.

$(a == null)$ *equivalente a* $!a.HasValue$

Tipos anuláveis (4)

```
Nullable<int> a = null;  
Nullable<int> b = 3;  
int? c = null;  
int? d = 5;
```

```
b += d;           // b <- 8  
d = a + b;        // d <- null (porque a vale null)
```

```
int e = (int)b;    // e <- 8  
int f = (int)c;    // exceção (porque c vale null)
```

```
int g = c ?? -1;   // g <- -1 (porque c vale null)
```

```
v? nv = null;    // v é um tipo valor  
Object o = nv;   // AVE verifica se o campo HasValue é  
false. Se for, não faz box e coloca null em 'o'
```