

Generics

Agenda da sessão

- Genéricos
- Tipos anuláveis

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 *ad-hoc*
 - Os dois métodos têm de ter nomes diferentes porque têm listas de parâmetros iguais

Exemplo: *Stack* não genérico

```
public class Stack {
    int sp = 0;
    object[] items = new object[100];
    public void Push(object item) { items[sp++] = item; }
    public object Pop() { return items[--sp]; }
}

public partial class Examples {
    public static void UseANonGenericStack() {
        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

Com genéricos

```
public interface IBusinessStats {  
    // ...  
    Customer[] Top10 (IList<Customer> customers);  
    Supplier[] Top10 (IList<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 homogêneas
- (e os dois métodos já podem ter o mesmo nome)

Exemplo: *Stack* genérico

```
public class Stack<T> {  
    int sp = 0;  
    T[] items = new T[100];  
    public void Push(T item) { items[sp++] = item; }  
    public T Pop() { return items[--sp]; }  
}  
  
public partial class Examples {  
    public static void UseAGenericStack() {  
        Stack<string> strStack = new Stack<string>(); // Stacks de tipos  
        Stack<int> intStack = new Stack<int>();       // distintos  
  
        string s;  
        int i;  
  
        strStack.Push("X");  
        intStack.Push(8); // sem box  
  
        s = strStack.Pop(); // sem cast  
        i = intStack.Pop(); // sem unbox  
  
        //intStack.Push("8"); // não compila!  
    }  
}
```

+ Expressividade

+ Desempenho

+ Robustez

Genéricos \neq *Templates*

- *Templates*
 - Não originam directamente código intermédio nem nativo
 - por cada instanciação de um *template*, a sua definição é combinada com a dos tipos-parâmetro para gerar código nativo específico
 - O compilador conhece as interfaces dos tipos-parâmetro usados na instanciação do *template*
 - verificações realizadas durante a compilação das várias instâncias
- Genéricos
 - 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 *≠* Templates

- 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 .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

Restrições

- Por omissão, os tipos-parâmetro só podem ser usados através da interface de `object`, já que é a única que é garantidamente implementada.
- Podem ser aplicadas restrições (*constraints*) aos tipos-parâmetro:
 - classe base
 - interfaces implementadas
 - existência de construtor sem parâmetros (`new()`)
 - tipo-referência (`class`) ou tipo-valor (`struct`)

Exemplo

```
public static partial class Utils {  
    public static T Min<T>(T[] tv) where T : IComparable<T> {  
        if (tv.Length == 0) throw new ArgumentException("Empty sequence");  
        T m = tv[0];  
        for (int i = 1; i < tv.Length; ++i) {  
            if (m.CompareTo(tv[i]) > 0) {  
                m = tv[i];  
            }  
        }  
        return m;  
    }  
}
```

- Para poder invocar o método `CompareTo` sobre os objectos do *array* é necessário que estes implementem a interface `IComparable<T>`
 - indicado como restrição
- A definição alternativa sem genéricos não admite *arrays* de tipos valor

```
public static object Min(IComparable[] vals)
```

Anatomia de um genérico

```
// Classe genérica com dois tipos-parâmetro: U e V
public class Generic<U,V>

    // O tipo U deve ter SomeBaseClass como classe base e implementar as
    // interfaces (também genéricas) ISomeInterface e IOneMoreInterface<V>
    where U : SomeBaseClass, ISomeInterface, IOneMoreInterface<V>

    // O tipo V deve ser um tipo-referência e implementar a interface
    // IAnotherInterface<U> e a interface genérica IYetAnotherInterface<V>
    where V : class, IAnotherInterface<U>, IYetAnotherInterface<V>, new()
{
    // Construtores não podem ser genéricos
    static Generic() { /* ... */ }
    public Generic() { /* ... */ }

    // Método não genérico: pode usar tipos-parâmetro da classe
    public void m1(U u, V v) { /* ... */ }

    // Método genérico com um tipo-parâmetro: X
    // Pode usar os seus tipos-parâmetro para além dos da classe
    public void m2<X>(X x, U u) { /* ... */ }

    // Classe interna com três tipos-parâmetro: U, V e W
    public class Nested<W> { public void m<Z>(V v, W w, Z z) { /* ... */ } }
}
```

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

```
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);  
    }  
}
```

Invariância de tipos genéricos

- *Invariant generic typing*

`List<string>` não é um subtipo de `List<object>`

```
public static partial class Examples {  
    public static void AddObjectToList(List<object> list) {  
        list.Add(new object()); // lista de strings conteria um object  
    }  
  
    public static void TestGenericInvariance() {  
        List<string> list = new List<string>();  
        list.Add("A"); list.Add("B");  
        AddObjectToList(list); // não compila: lista poderia ser modificada  
    }  
}
```

- Alternativa

- definição de método genérico

`void AddObjectToList<T>(List<T> list) [where T : SomeClass]opt`

Notas

- Não é possível ser subtipo de um tipo-parâmetro
 - tipos valor não podem ser estendidos
 - não é possível determinar a tabela de métodos

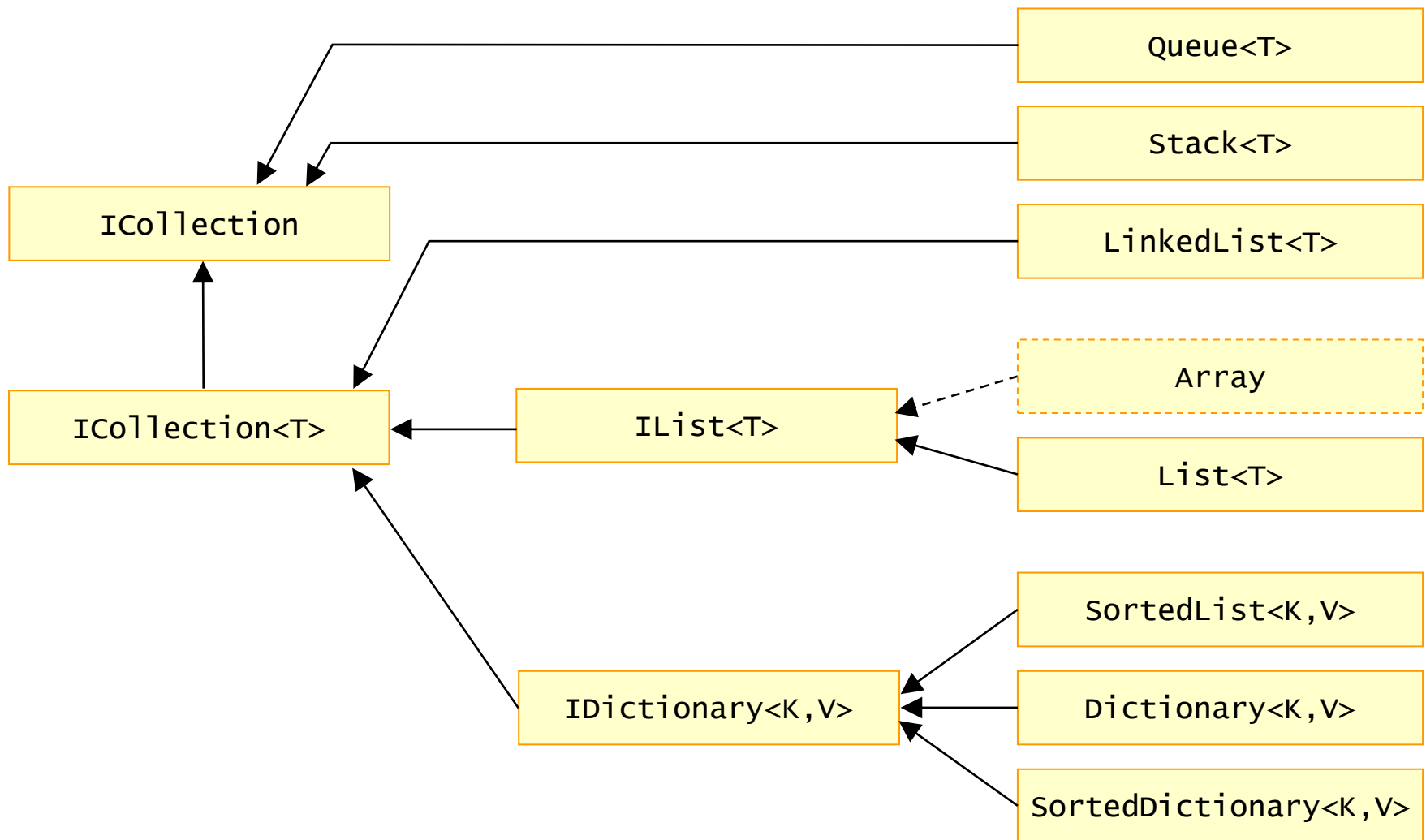
```
public class X<T> : T
```
- Tipos genéricos não podem estender `Attribute`
 - Atributos não podem ser genéricos

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
```

- Coleções genéricas implementam interfaces genéricas e não-genéricas

```
public class List<T> :
    IList<T>, ICollection<T>, IEnumerable<T>,
    IList, ICollection, IEnumerable
```

Comparadores

- Comparações através das interfaces `IComparer<T>` e `IEqualityComparer<T>`
- Comparadores pré-definidos:
 - `Comparer<T>.Default`: se `T` implementa `IComparable<T>`, retorna uma instância de `Comparer` que usa essa implementação; caso contrário, retorna um `Comparer` baseado em `IComparable`
 - `EqualityComparer<T>.Default`: se `T` implementa `IEquatable<T>`, retorna uma instância de `Comparer` que usa essa implementação; caso contrário, retorna um `EqualityComparer` que usa `Equals` e `GetHashCode`
- Para *strings*, usar `StringComparer`, em vez de `Comparer<string>`

Delegates e algoritmos genéricos pré-definidos

- No *namespace* `System` estão definidos 4 *delegates* genéricos:

```
public delegate void Action<T>(T obj)
public delegate int Comparison<T>(T x, T y)
public delegate TOutput Converter<TInput, TOutput>(TInput input)
public delegate bool Predicate<T>(T obj)
```

- As classes `System.Collections.Generic.List<T>` e `System.Array` disponibilizam um conjunto de métodos, parametrizados por funtores, para acesso aos seus dados. Ex.:

`List<T>`:

```
public int FindIndex(Predicate<T> match);
public List<T> FindAll(Predicate<T> match);
public bool TrueForAll(Predicate<T> match);
public void ForEach(Action<T> action);
...
```

`Array`:

```
public static T Find<T>(T[] array, Predicate<T> match);
public static bool Exists<T>(T[] array, Predicate<T> match);
public static void Sort<T>(T[] array, Comparison<T> comparison);
public static U[] ConvertAll<T, U>(T[] array,
                                   Converter<T,U> converter);
...
```

Exemplo

```
public static partial class Utils {
    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;
        }
    }

    public static T[] InRange<T>(T[] ts, T min, T max)
        where T : IComparable<T> {
        return Array.FindAll(ts, new RangeComparer<T>(min, max).IsInRange);
    }
}

public static partial class Examples {
    public static void TestInRangeExample() {
        Array.ForEach(
            Utils.InRange(new int[] { 10, 21, 32, 43 }, 20, 40),
            Console.WriteLine
        );
    }
}
```

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

```
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 (eq. struct)
 - **class**: restrição de tipo-referência (eq. class)
 - **.ctor**: restrição de construtor sem parâmetros (eq. new)
 - **(C1, ..., Cn)** é a lista de restrições de classe base e de interfaces implementadas

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



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

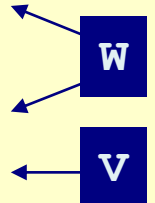
Referência aos tipos-parâmetro

- 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.

```
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          !!0
        IL_0006: isinst       !1
        IL_000b: ldnull
        IL_000c: cgt.un
        IL_000e: ret
    }
}
```



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, ldoobj, stobj, cpobj`

Terminologia

- Definição de tipo genérico: `public class X<T> {}`
- Lista de parâmetros de tipo genérico: `<T>`
- Parâmetro de tipo genérico: `T`
- Tipo genérico construído:
 - Tipo genérico aberto: `X<T>`
 - Tipo genérico fechado: `X<int>`
 - instância de tipo genérico
- Lista de argumentos de tipo genérico: `<int>`
- Argumento de tipo genérico: `int`
- Restrição de parâmetro de tipo genérico: `class, new(), SomeClass`
 - Restrições: classe base, interfaces implementadas
 - Atributos: `class, struct, new()`

Reflexão: System.Type

- Para obter uma instância de Type que represente uma definição de tipo genérico ou um tipo genérico fechado

```
Type td = typeof(Dictionary<,>);
```

```
Type ct = typeof(Dictionary<int,string>);
```

- Pode representar
 - tipo genérico aberto
 - tipo genérico fechado
 - parâmetro genérico
 - argumento genérico (não é um tipo genérico)

Reflexão : System.Type

- Novos métodos na classe Type
 - IsGenericType
 - definições de tipos genéricos, tipos genéricos construídos, parâmetros genéricos
 - *false* para *arrays* de tipos genéricos
 - IsGenericTypeDefinition
 - definições de tipos genéricos
 - ContainsGenericParameters
 - definições de tipos genéricos, tipos genéricos abertos, *arrays* de tipos genéricos abertos
 - GetGenericArguments
 - retorna um *array* de parâmetros ou de argumentos de um tipo genérico
 - IsGenericParameter
 - distinção entre parâmetro e argumento de um tipo genérico

(cont.)

Reflexão : System.Type

- Name
 - permite obter o nome de um parâmetro de um tipo genérico
- GenericParameterPosition
 - índice do tipo-parâmetro
 - usar DeclaringMethod e DeclaringType para distinguir In de !In
- GetGenericParameterConstraints
 - *array* de restrições (classe base e interfaces)
 - usar IsClass para distinguir restrição de classe base de restrição de interface
- GenericParameterAttributes
 - combinação de restrições (class, struct ou new())
- DeclaringMethod
 - MethodInfo do método genérico que o tipo parametriza
 - *null* para parâmetros de tipo
- DeclaringType
 - Tipo genérico que o tipo parametriza
 - Tipo a que pertence o método que o tipo parametriza

Tipos anuláveis: sumário

- Objectivos
- Classe genérica `Nullable<T>`
- Operadores

Tipos anuláveis

- 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

- 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

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

- Modificador ? para declarar um tipo como anulável.

```
typeof(int?) == typeof(Nullable<int>)
```

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

```
(a ?? 0) == (a.HasValue ? a.Value : 0)
```

- Comparação com null verifica HasValue.

```
(a == null) == !a.HasValue
```

Tipos anuláveis

```
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)  
int h = a ?? c ?? 0; // h <- 0 (porque a e c valem null)
```