Common Type System

Notas para a disciplina de Ambientes Virtuais de Execução

Semestre de Verão, 10/11

Delegates na CLI

- Mecanismo de suporte a *callbacks* fornecido pelo *Runtime*
- Generalização *type-safe* do conceito de ponteiro para função em C/C++, que permite usar como *callbacks* métodos estáticos ou métodos de instância.
- São suportados à custa de *Reference Types* que derivam directamente de **`System.MulticastDelegate`**

Callbacks em C/C++

```
typedef int (*Comparator)(const void *, const void *);


int compareInts(const void*i1, const void *i2) {
        return *((int *) i1) - *((int *) i2);
}



void testQSort() {
        int  vals[] = { 2, 8 , 13, 5, 4 };
        int nelems = sizeof(vals)/sizeof(int);
        qsort(vals, nelems, sizeof(int), compareInts);
        for (int i=0; i < nelems; ++i) {
                Console::WriteLine(vals[i]);
        }
}
```

Callbacks em C#

```
delegate int Comparator(object o1, object o2);

 class Program  {

        public static void sort(object[] vals, Comparator cmp)  {
        …
        }

        private static int intComparer(object o1, object o2) {
            int i1 = (int)o1;
            int i2 = (int)o2;
            return i1 - i2;
        }

        static void Main(string[] args) {
            object[] vals = { 3, 1, 8, 4, 5, 2 };

            sort(vals, intComparer);
            foreach (int i in vals) Console.WriteLine(i);
        }
}
```

Delegates: tipo gerado pelo compilador

- Em C#, a palavra reservada **delegate** define um novo tipo
  - Método **Invoke** com a assinatura usada na definição

```
class auto ansi sealed public Comparator
         extends [mscorlib]System.MulticastDelegate {

  .method … void .ctor(object 'object', native int 'method') runtime
managed { }

  .method … int32 Invoke(object i, object j) runtime managed { }
}
```

- Um *delegate* é um tipo referência (derivado de **MulticastDelegate**)
  - **Comparator p1 = f1;** é uma abreviação para
    Comparator p1 = new Comparator(f1);
  - **int i = p1(1, 2);** resulta em **int i = p1.Invoke(1,2);** (proibido em C#)

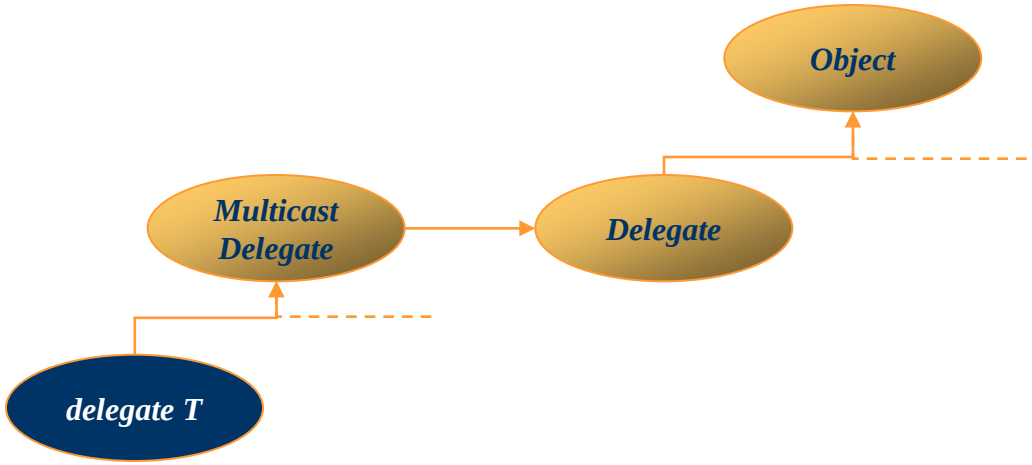
Loggers

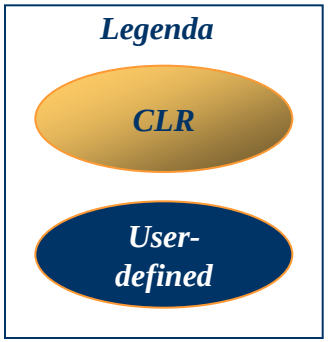
```
delegate void Logger(string msg, int code);

class MyStreamLogger {
         // callback em método de instância
         StreamWriter logStream;

         public MyStreamLogger(Stream logStream) {
            this.logStream = new StreamWriter(logStream);
         }

         public  void log(string s, int id) {
         logStream.WriteLine("MyStreamLogger: MSG={0}, ID={1}", s, id);
            logStream.Flush();
         }
}

class MyConsoleLogger {
         // callback em método estático
         public  static void log(string s, int id) {
                  System.Console.WriteLine("MyConsoleLogger:
                           MSG={0}, ID={1}", s, id);
         }
}
```

Using Loggers

```
class WorkerProcess {
     public Logger logger;

     virtual protected void doLog(string msg, int code) {
           if (logger != null)   logger(msg, code);
     }

     public void doWork() {
          // .... working....
          // logging error
            doLog("erro", 123);
     }
}

class Class1 {
          static void Main(string[] args) {
              WorkerProcess p = new WorkerProcess();
              MyStreamLogger fl = new MyStreamLogger(Console.OpenStandardOutput());
              // registar callbacks
              p.logger += new Logger(fl.log);
              p.logger += new Logger(MyConsoleLogger.log);
              p.doWork();
          }
}
```

Delegates no CLI: A classe Delegate






```
public abstract class Delegate {
    ...
    public static Delegate Combine(Delegate d1, Delegate d2);
    public static Delegate Remove(Delegate source, Delegate d);
    public virtual Delegate[] GetInvocationList();
    public MethodInfo Method { get; }
    public Object Target { get; }
    public static Boolean operator==(Delegate d1, Delegate d2);
    public static Boolean operator!=(Delegate d1, Delegate d2);

}
```

Delegates: combinação

- As instâncias de *delegates* são imutáveis
- Duas instâncias podem ser combinadas, dando origem a uma terceira instância
  - A chamada do método **Invoke** da terceira instância resulta na chamada dos métodos associados às duas instâncias originais
  - O valor de retorno é o retorno da segunda instância

```
Logger p1 = …
Looger p2 = f2;
p2 = (Logger) Delegate.Combine(p1, p2);
```

- Sintaxe simplificada em C# (algo enganadora)

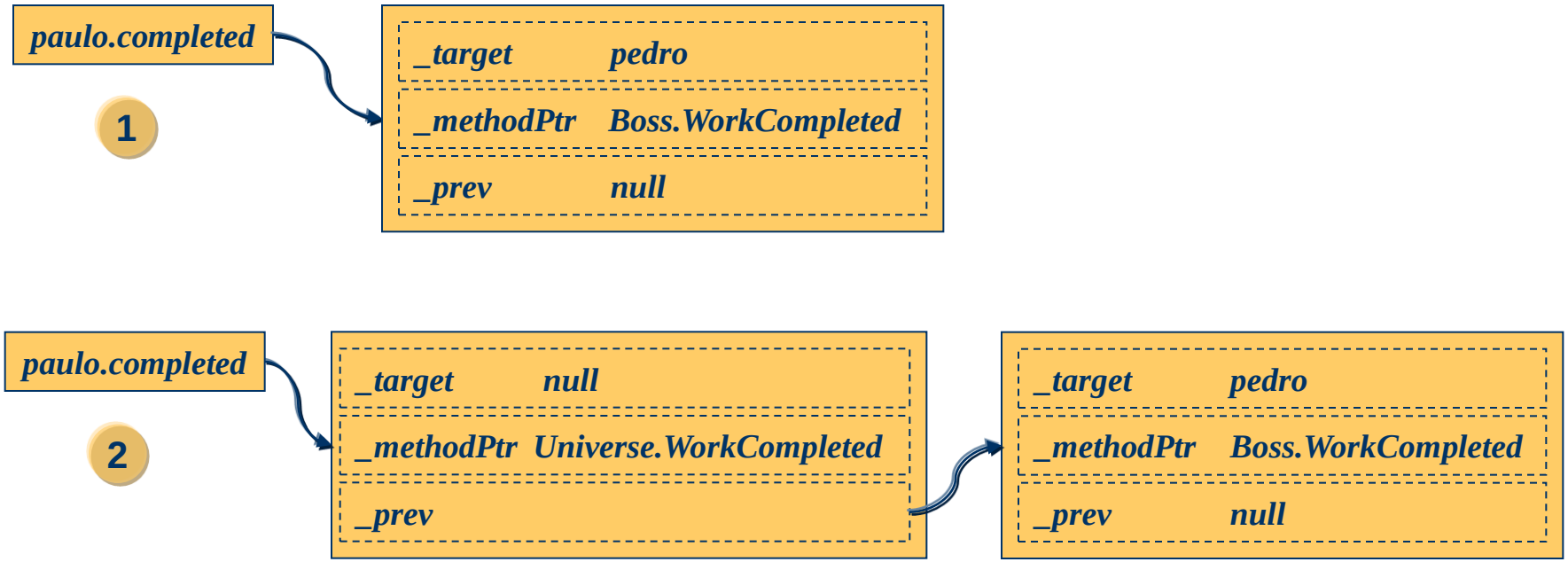
```
p2 += p1;
```

- Método **Delegate.Remove** (operador **-=** em C#) realiza a remoção

Implementação de delegates: lista intrusiva (1)

C#

```
...
paulo.completed = new WorkCompleted(pedro.WorkCompleted);   1
paulo.completed += new WorkCompleted(Universe.WorkerCompleted);   2
...
```



cadeias de delegates(2)

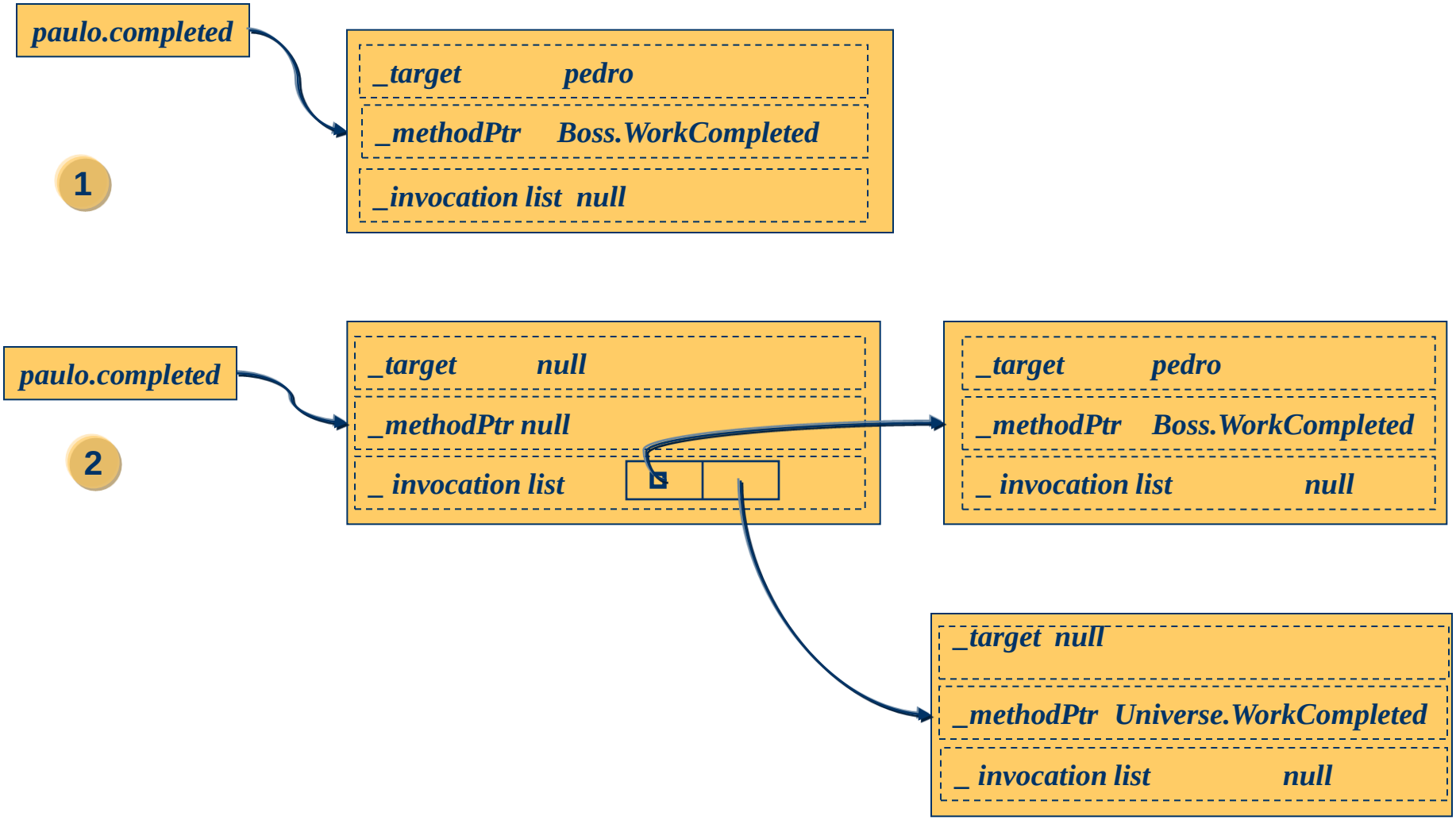
C#

```
public class Worker
{
   public void DoWork() { … int grade = completed(); … }
}
```

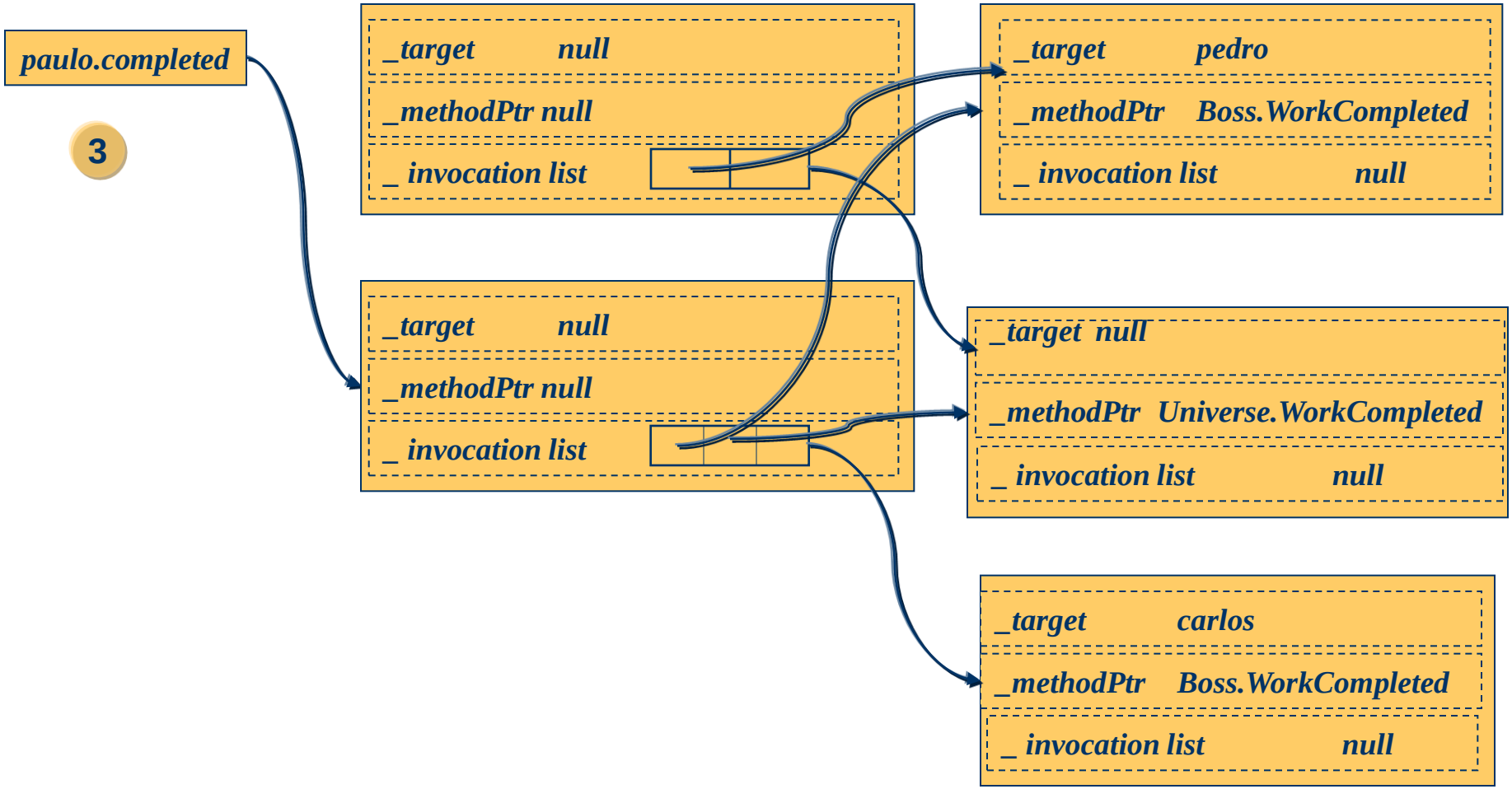
Pseudo código

```
public class WorkCompleted : System.MulticastDelegate
{
   …
   public virtual Int32 Invoke()
   {
      if(_prev != null) _prev.Invoke();
      return _target._methodPtr();
   }
   …
}
```

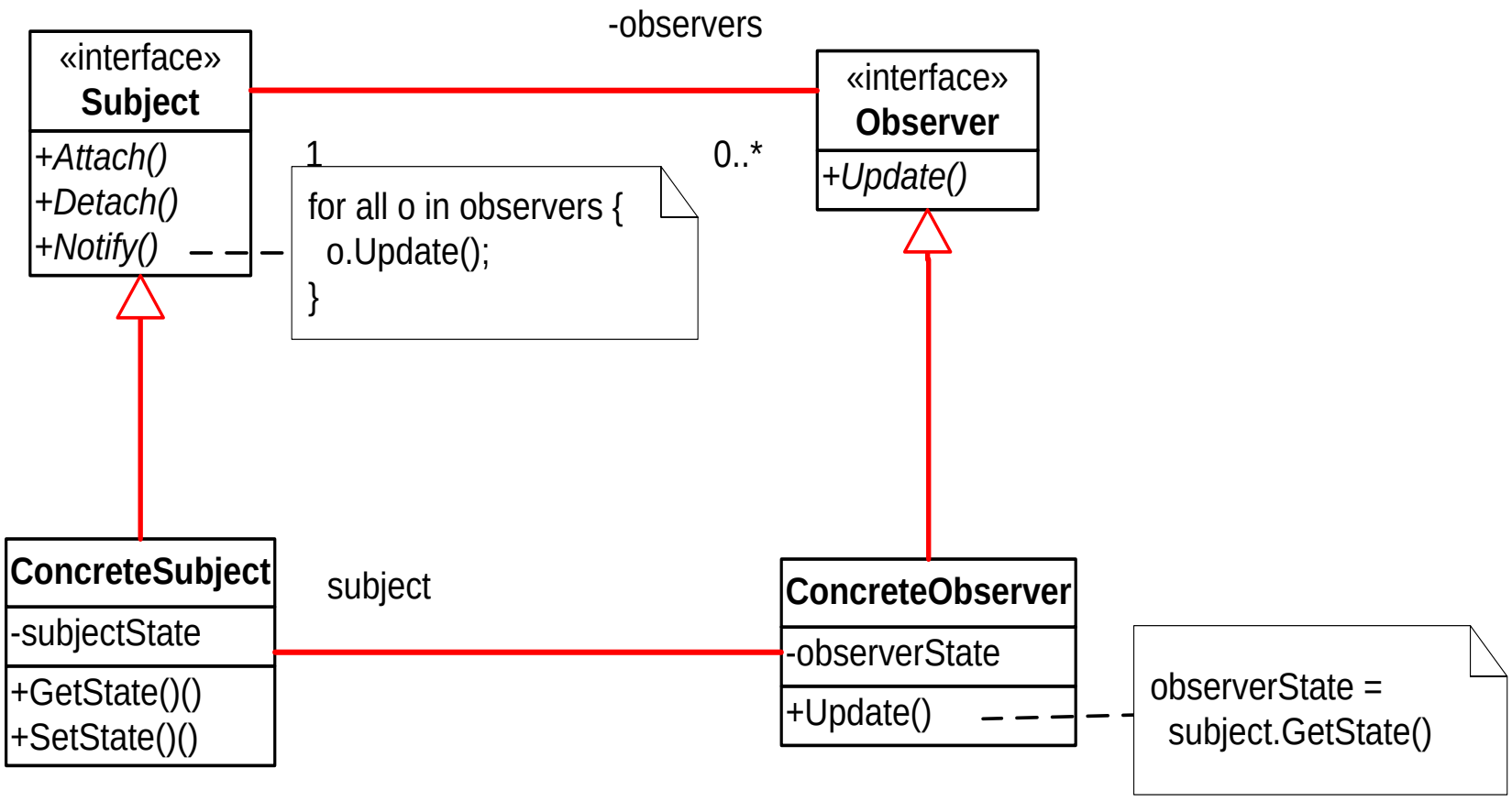
Cadeias de delegates (.NET 2.0)



Cadeias de delegates (.NET 2.0)



Os delegates suportam directamente o padrão *Observer*, aqui implementado usando interfaces



Eventos

- Membros que identificam o tipo como produtor de acontecimentos
- Cada membro “evento” é suportado por um par de métodos e um campo do tipo *delegate* correspondente
- A linguagem C# fornece dois operadores (**+=** e **-=**) cuja utilização é convertida, pelo compilador, em chamadas ao par de métodos

Definição em C#

```
public delegate void WorkStarted();

public class Worker {
  public event WorkStarted Started;
}
```

Código equivalente gerado pelo compilador

```
public class Worker {
  private WorkStarted Started;
  public void add_Started(WorkStarted value)
  { this.Started = Delegate.Combine(this.Started,value); }
  public void remove_Started(WorkStarted value)
  { this.Started = Delegate.Remove(this.Started,value); }

  /* mais entrada de metadata para representar o membro evento */
}
```

Definição *custom* dos métodos de evento

```
class Worker {
  public void DoWork() { ... }
  public event WorkStarted started;
  public event WorkCompleted completed;
  public event WorkProgressing progressing {
    add {
      if( DateTime.Now.Hour < 12 ) {
        _progressing += value;
      }
      else {
         string msg = "Must register before noon.";
         throw new InvalidOperationException(msg);
      }
    }

    remove {
       _progressing -= value;
    }
  }
  private WorkProgressing _progressing;
}
```

Enumerados

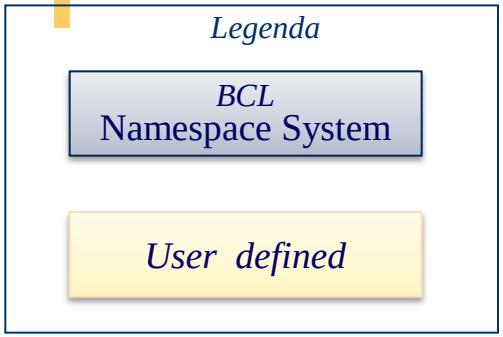
Definição em C#

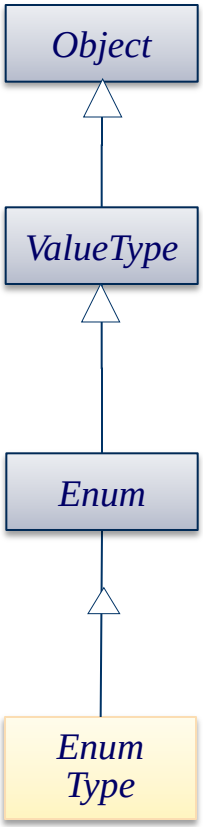
```
enum EstacoesDoAno {
                    Primavera, Verao, Outono, Inverno
};
```

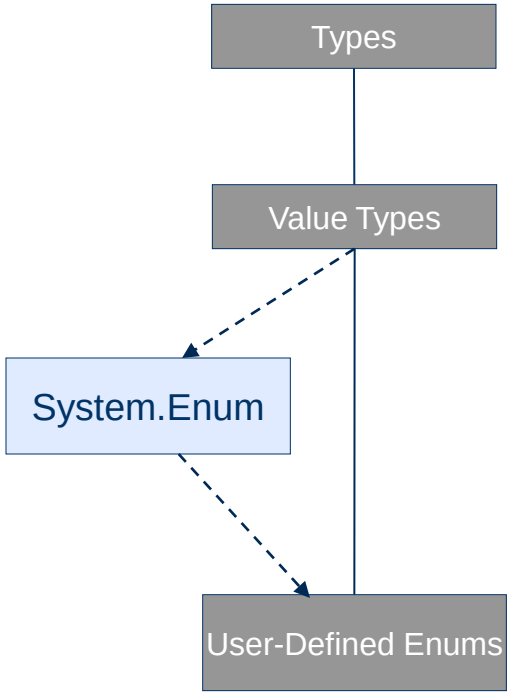
Código equivalente gerado pelo compilador

```
.class private auto ansi sealed  EstacoesDoAno
       extends [mscorlib]System.Enum {
  .field public specialname rtspecialname int32 value__
  .field public static literal valuetype
   EstacoesDoAno Primavera = int32(0x00000000)
  .field public static literal valuetype
   EstacoesDoAno Verao = int32(0x00000001)
  .field public static literal valuetype
   EstacoesDoAno Outono = int32(0x00000002)
  .field public static literal valuetype
   EstacoesDoAno Inverno = int32(0x00000003)
} // end of class Geometry.EstacoesDoAno
```

Linguagem C# - Excerto do Modelo de tipos
(enumerados)





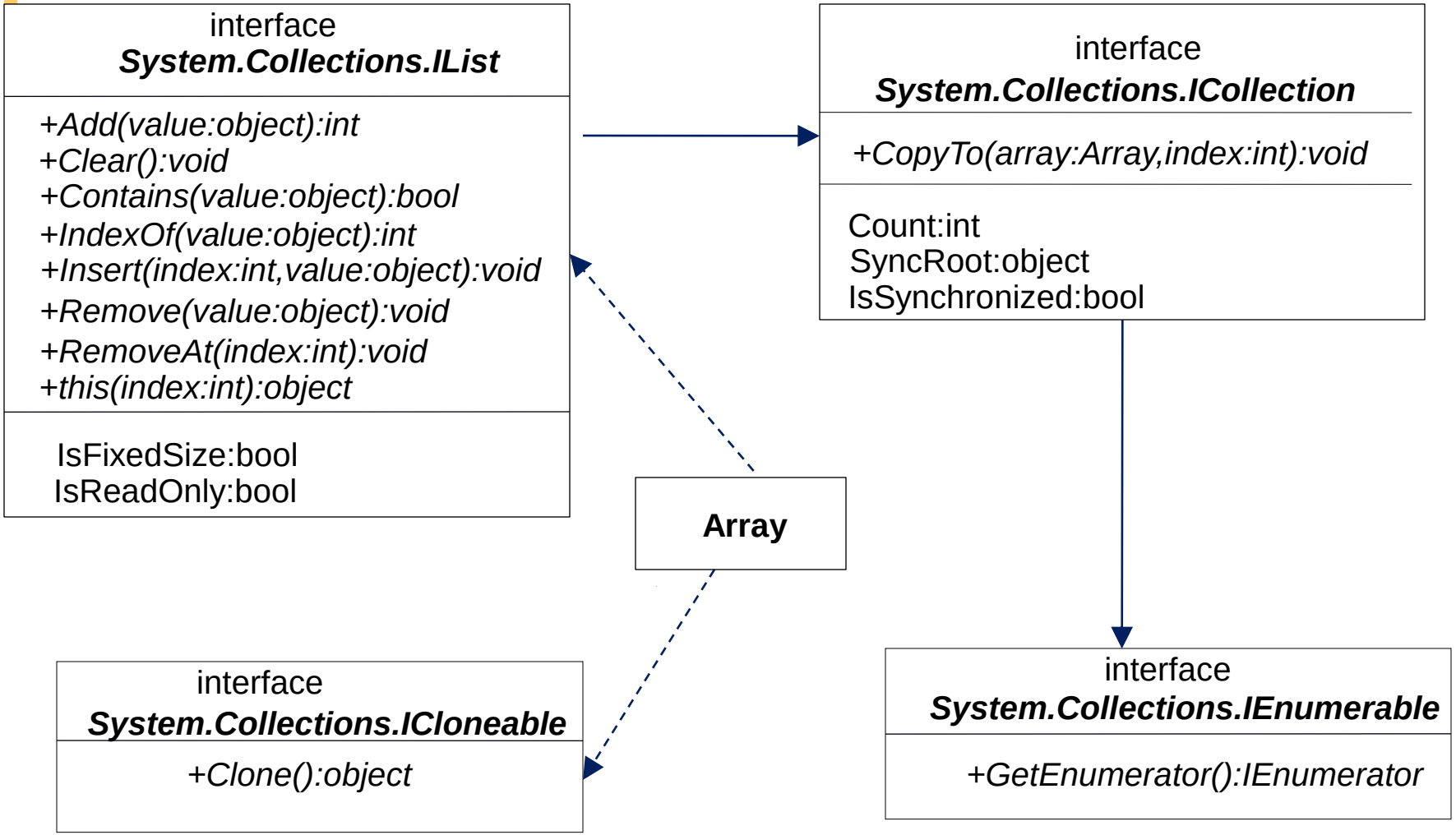


O tipo Enum (excerto)

Implementa IComparable

int CompareTo(object target)	Compares this instance to a specified object and returns an indication of their relative values.
static string Format(Type enumType, object value, string format)	Converts the specified value of a specified enumerated type to its equivalent string representation according to the specified format.
static string GetName(Type enumType, object value)	Retrieves the name of the constant in the specified enumeration that has the specified value.
static string[] GetNames(Type enumType)	Retrieves an array of the names of the constants in a specified enumeration.
static Type GetUnderlyingType(Type enumType) .	Returns the underlying type of the specified enumeration.
static Array GetValues(Type enumType)	Retrieves an array of the values of the constants in a specified enumeration.
static bool IsDefined(Type enumType,object value)	Returns an indication whether a constant with a specified value exists in a specified enumeration.
static object Parse(Type enumType,string value)	Overloaded. Converts the string representation of the name or numeric value of one or more enumerated constants to an equivalent enumerated object.

Arrays - Interfaces implementadas pela classe Array



O tipo Array - propriedades

Para além das propriedades relacionados com a implementação das interfaces `IList, IEnumerable e IClonable`

Length	Gets a 32-bit integer that represents the total number of elements in all the dimensions of the **Array**.
LongLength	Gets a 64-bit integer that represents the total number of elements in all the dimensions of the **Array**.
Rank	Gets the rank (number of dimensions) of the **Array**.

O tipo Array – métodos publicos (I)

Para além dos métodos relacionados com a implementação das interfaces `IList, IEnumerable e IClonable` e dos definidos em object

static int **BinarySearch(**Array, object)	Overloaded. Searches a one-dimensional sorted **Array** for a value, using a binary search algorithm.
static void **Copy**(Array, Array, int)	Overloaded. Copies a section of one **Array** to another **Array** and performs type casting and boxing as required.
static Array **CreateInstance**(Type, int)	Overloaded. Initializes a new instance of the **Array** class.
int **GetLength**(int dimension)	Gets a 32-bit integer that represents the number of elements in the specified dimension of the **Array**.
int **GetLowerBound**(int dimension)	Gets the lower bound of the specified dimension in the **Array**.

O tipo Array – métodos publicos (II)

public int **GetUpperBound**(int dimension)	Gets the upper bound of the specified dimension in the **Array**.
public object **GetValue(int**[] indexes)	Overloaded. Gets the value of the specified element in the current **Array**. The indexes are specified as an array of 32-bit integers.
public void **Initialize**()	Initializes every element of the value-type **Array** by calling the default constructor of the value type.
public **static** void **Reverse(Array**)	Overloaded. Reverses the order of the elements in a one-dimensional **Array** or in a portion of the **Array**.
public void **SetValue(object** value , int[] indexes)	Overloaded. Sets the specified element in the current **Array** to the specified value.
public static void **Sort(Array**)	Overloaded. Sorts the elements in one-dimensional **Array** objects.

Arrays Multidimensionais e Arrays de Arrays (*jagged arrays*)

Array Multidimensional

```
// declare reference to 2D array of Int32
int[,] matrix;
// allocate array of 3x4 elements
matrix = new int[3,4];
// touch all elements in order
for (int i = 0; i < matrix.GetLength(0); ++i)
  for (int j = 0; j < matrix.GetLength(1); ++j)
    matrix[i,j] = (i + 1) * (j + 1);
```

Arrays de Arrays

```
// declare reference to jagged array of Int32
int[][] matrix;
// allocate array of 3 elements
matrix = new int[3][];
// allocate 3 subarrays of 4 elements
matrix[0] = new int[4];
matrix[1] = new int[4];
matrix[2] = new int[4];
// touch all elements in order
for (int i = 0; i < matrix.Length; ++i)
  for (int j = 0; j < matrix[i].Length; ++j)
    matrix[i][j] = (i + 1) * (j + 1);
```

Genéricos

- Objectivo
  - Independência de tipos na definição de:
    - Tipos de dados (classes, estruturas, interfaces, *delegates)*
    - Algoritmos (métodos)
  - Sem perder:
    - Robustez
    - Desempenho
    - Expressividade
- Exemplo
  - Realizar uma fila homogénea de objectos, independente do tipo de objecto
  - Definição: **class Queue<T>{...}**
  - Utilização: **Queue<int> qi; Queue<string> qs;**

Genéricos

- Type Safety
  - Permite criar colecções homogéneas, validadas em tempo de compilação
- Aumento da legibilidade
  - O código não necessita de *cast´s* explícitos
- Aumento de performance
  - Instâncias de tipo valor não precisam de ser boxed para serem guardadas em colecções genéricas.

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();       // excepção!
   }
}
```

- Expressividade

- Desempenho

- Robustez

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

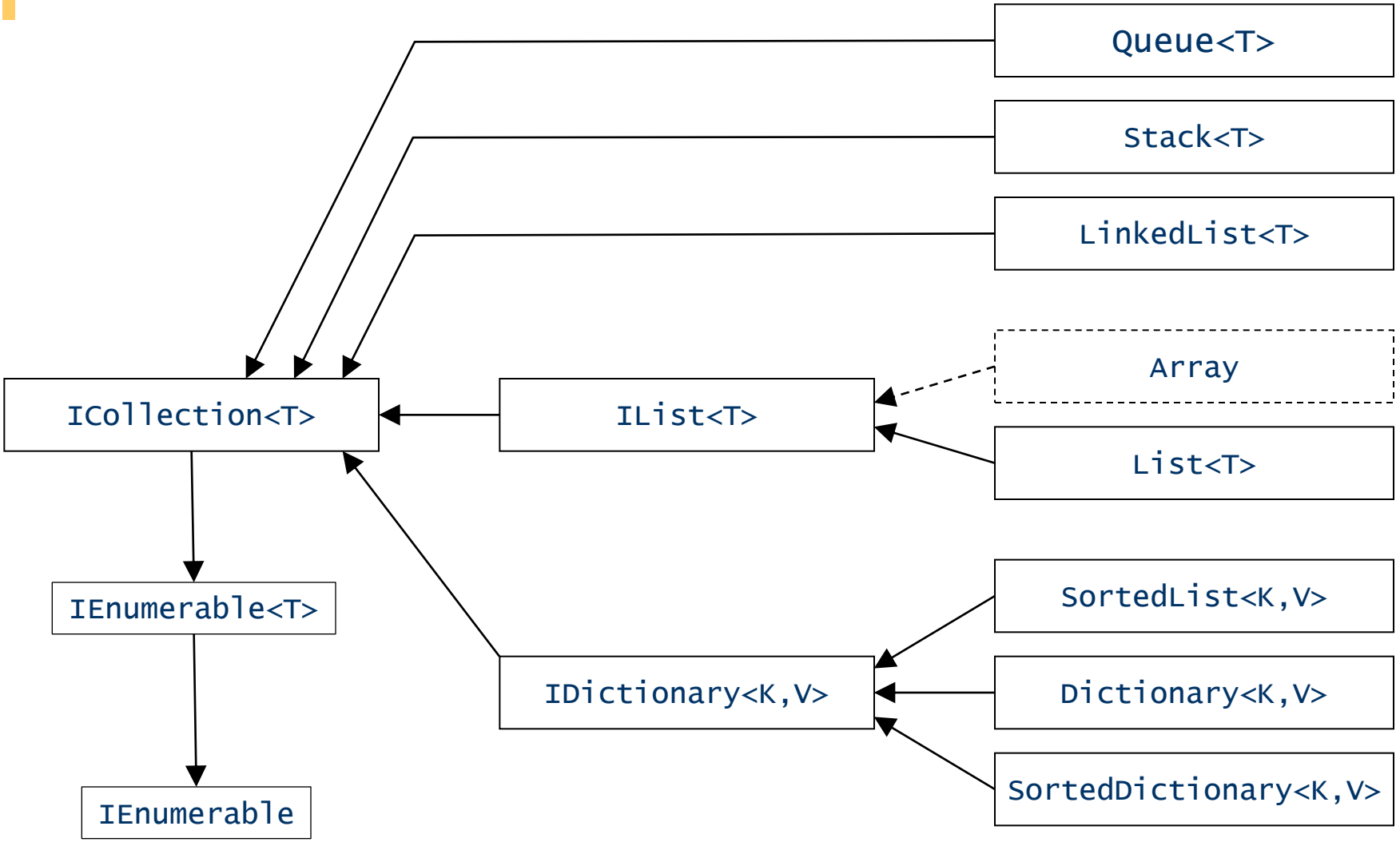
Namespace System.Collections.Generic

- Novas versões genéricas de colecções

  (implementam `ICollection<T>`)

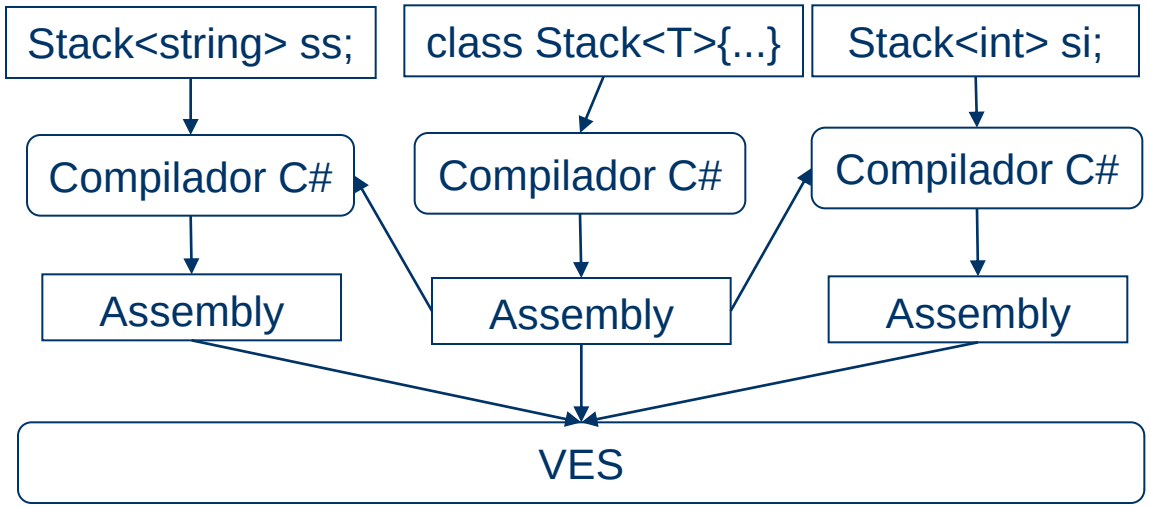
`Queue<T>`	Versão genérica de `Queue` (FIFO)
`Stack<T>`	Versão genérica de `Stack` (LIFO)
`List<T>`	Versão genérica de `ArrayList` (lista sobre *array*)
`LinkedList<T>`	Lista duplamente ligada
`SortedList<K,V>`	Versão genérica de `SortedList` (colecção ordenada de pares chave/valor) baseada em *arrays*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

System.Collections.Generic (interfaces e implementações)



Compilação de Genéricos

- 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



Tipos abertos e fechados

- A definição de um tipo genérico corresponde à criação de um novo tipo, mas do qual não se podem criar instâncias (tipo aberto)

- Para cada parametrização (completa) de um tipo aberto a máquina virtual (o *compilador JIT*) cria um correspondente tipo fechado do qual se podem criar instâncias

Instruções IL e prefixos de instruções modificados e acrescentadas devido aos genéricos (excerto)

- initobj
- newobj
- constrained.
- box
- unbox.any

Genéricos em Java

- A compilação da classe genérica Generic é equivalente ao código da direita (um processo que se denomina de erasure, mais informação de metadata que indica que a classe originalmente era genérica e os correspondentes parâmetros

```
class Generic<T> {
  T  value;
  Generic(T value) {
         this.value=value;
  }

  T  getValue() {
         return value;
  }

  void setValue(T t) {
         value=t;
  }
}
```

Erasure

```
class GenericObj {
   Object  value;
   public GenericObj(Object value) {
          this.value=value;
   }

   Object  getValue() {
          return value;
   }

   void setValue(Object t) {
          value=t;
   }
}
```

- Problemas:
  - Diferentes especializações partilham campos estáticos
  - Construções de objectos do tipo genérico
  - `extend` e `implements` de especializações do mesmo tipo ou interface genéricos

Constraints (Restrições)

- Nos tipos genéricos do CTS:
  - A compilação é realizada sem o conhecimento do tipo parâmetro
  - Para validar a utilização é necessário *algum* conhecimento sobre o tipo
- 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`)

Constraints (Restrições) II

```
class Utils {
    public static T max<T>(params T[] vals) where T : IComparable<T> {
       if (vals.Length == 0)
          throw new System.Exception("Invalid Argument List");
       T r = vals[0];
       for(int i=1; i < vals.Length; ++i) {
          if (r.CompareTo(vals[i]) < 0) r = vals[i];
       }
       return r;
    }
}
```

Constraints	Descrição
where T: <className>	**T** tem de derivar de <className>
where T:<interfaceName>	**T** tem de implementar <interfaceName>
where T : class	**T** tem de ser um tipo referência
where T: struct	**T** tem de ser um tipo valor
where T : new()	**T** tem de ter construtor sem parâmetros

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 functores, 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 void ForEach<T>(T[] array, Action<T> action);
   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
      );
   }
}
```

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`).

O tipo genérico Nullable<T>

```
public struct Nullable<T> where T: struct {
    private bool hasValue;
    internal T value;
    public Nullable(T value);
    public bool HasValue { get; }
    public T Value { get; }
    public T GetValueOrDefault();
    public T GetValueOrDefault(T defaultValue);
    public override bool Equals(object other);
    public override int GetHashCode();
    public override string ToString();
}
```

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; // excepção (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)