1

Introdução à Linguagem de Programação C#

João Pascoal Faria

FEUP

10 de Setembro de 2001

2

Objectivo

- Apresentar os elementos fundamentais da linguagem C#
- Enfatizar conceitos comuns ao *framework* Microsoft .NET
- Enfatizar novidades
- Deixar para leitura posterior alguns tópicos mais avançados (diapositivos com asterisco)

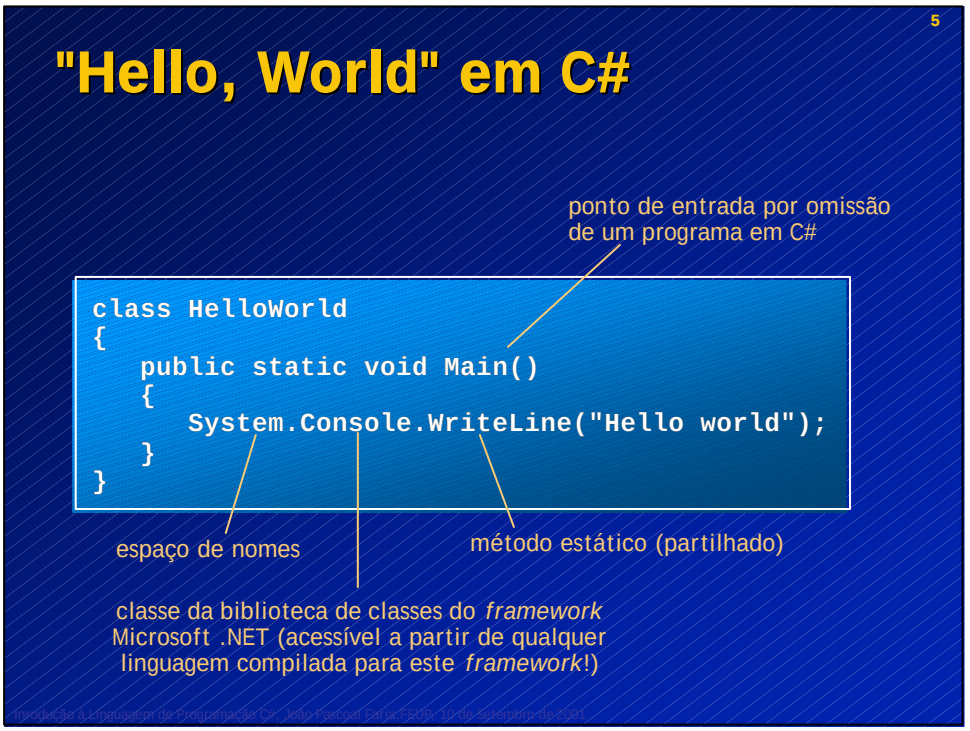
3

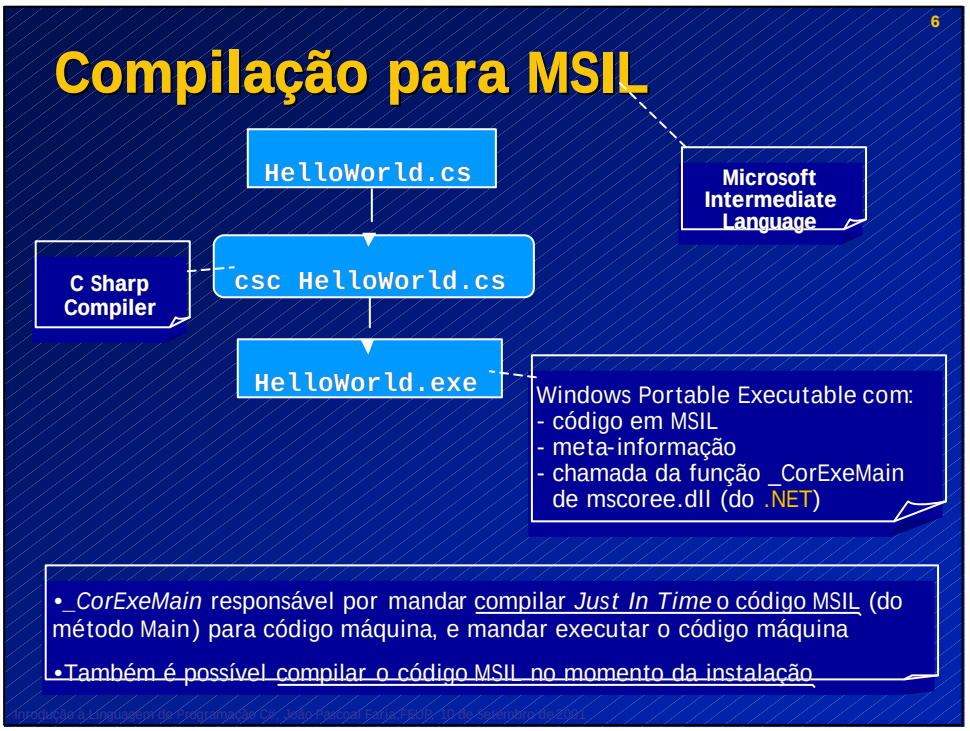
Índice

- O meu primeiro programa em C#
- Introdução ao *framework* Microsoft .NET
- Objectivos do C#
- Sistema de tipos
- Classes
- Estruturas, interfaces, enumerações e *arrays*
- Delegados (apontadores para funções OO) e eventos
- Operadores e instruções
- Tratamento de excepções
- Sobrecarga de operadores
- Atributos (anotações tipadas)
- Interoperação com código não gerido pelo .NET *runtime*
- Outros tópicos: documentação em XML, compilação condicional *, *assemblies* (componentes), reflexão, *multithreading* *

4

O meu primeiro programa em C#


5
"Hello, World" em C#
ponto de entrada por omissão de um programa em C#
class HelloWorld
{
public static void Main()
{
System.Console.WriteLine("Hello world");
}
}
espaço de nomes
método estático (partilhado)
classe da biblioteca de classes do framework Microsoft .NET (acessível a partir de qualquer linguagem compilada para este framework!)


6
Compilação para MSIL
HelloWorld.cs
Microsoft Intermediate Language
C Sharp Compiler
csc HelloWorld.cs
HelloWorld.exe
Windows Portable Executable com:
- código em MSIL
- meta-informação
- chamada da função _CorExeMain de mscoree.dll (do .NET)
•_CorExeMain responsável por mandar compilar Just In Time o código MSIL (do método Main) para código máquina, e mandar executar o código máquina
•Também é possível compilar o código MSIL no momento da instalação

7

"Hello, World" em MSIL *

Código do método Main obtido com o ildasm (Intermediate Language Disassembler):

```
.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size  11 (0xb)
  .maxstack  8
  IL_0000: ldstr  "Hello, World"
  IL_0005: call void ['mscorlib']System.Console.WriteLine(class System.String)
  IL_000a: ret
} // end of method 'Hello:Main'
```

carrega string para a stack

Assembly

tipo de parâmetro a retirar da stack

8

Introdução ao *Framework* Microsoft .NET

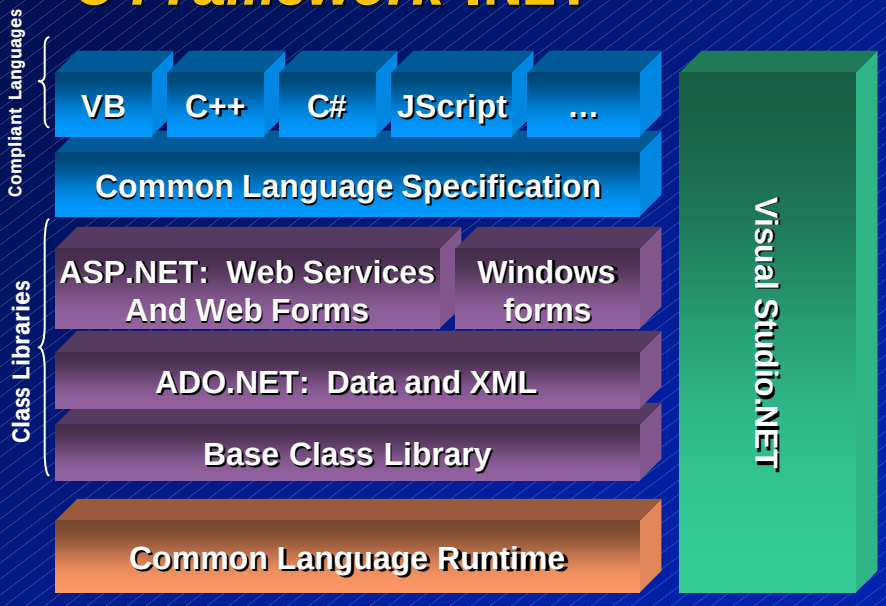
9

A Linguagem C# e o *Framework* Microsoft .NET

- C# nasceu com o .NET
- C# foi desenhado à medida do .NET
- C# é a linguagem que melhor reflecte as características do .NET
- Milhões de linhas de código do .NET escritas em C#

10

O *Framework* .NET



Fonte: Manuel Costa, Microsoft Portugal

11

Common Language Runtime (CLR)

- O CLR faz a gestão em tempo de execução do código conforme com o *Framework* .NET
  - Gestão automática de memória *(garbage collector* - GC)
  - Gestão de *threads*
  - Gestão de segurança
    - controlo de acesso a recursos sensíveis (ficheiros, rede, etc.)
  - Verificação de código
    - conformidade com o CTS (Common Type System)
    - *type safety* – código acede apenas a localizações de memória a que está autorizado a aceder (se verificação falhar, lança excepção)
  - Compilação de código
    - compilação *just in time* (JIT) de MSIL para código máquina

12

Programas Hospedeiros (*runtime hosts*)

- O .NET *Framework* inclui diversos programas hospedeiros (*runtime hosts*) para aplicações .NET
- Um programa hospedeiro é um componente não gerido que carrega o CLR para o seu processo e inicia a execução de código gerido
- Programas hospedeiros:
  - **ASP.NET** – carrega o *runtime* para o processo que irá tratar o pedido Web; cria um domínio aplicacional (processo lógico gerido pelo CLR) para cada aplicação Web que irá correr num servidor Web
  - **Internet Explorer** – cria domínios aplicacionais (por defeito, um para cada *Web site*) para correr controlos Windows Forms embebidos em documentos HTML; comunicação entre CLR e IE através de filtros MIME
  - **Executáveis shell** – controlo é transferido para código de hospedagem cada vez que um executável é lançado pela shell (ver HelloWorld)

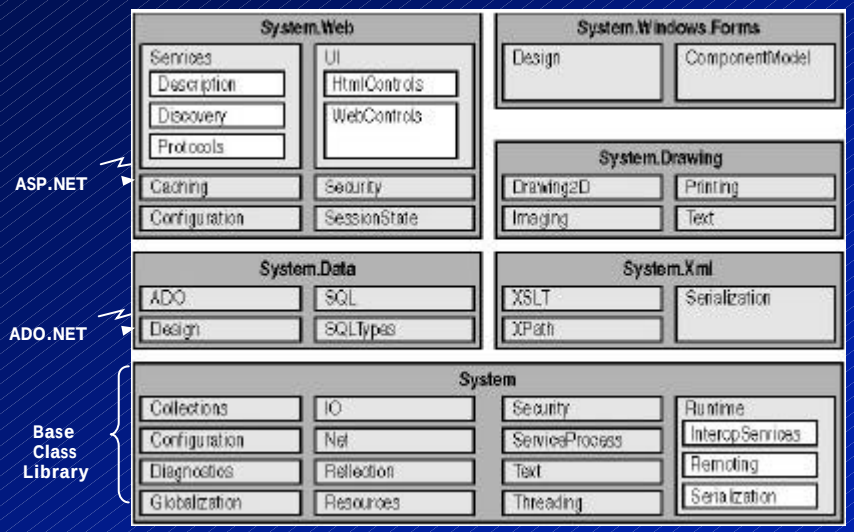
13

Common Language Specification (CLS)

- Especificação de subconjunto de *features* do CTS a que as linguagens e compiladores devem obedecer, para garantir a interoperabilidade de programas escritos em diferentes linguagens
  - Possibilidade de, numa linguagem, usar (ou até mesmo estender) uma classe ou outro tipo definido noutra linguagem
  - Apenas interessa para partes de um programa que precisam de interagir com outros programas escritos noutras linguagens

14

Bibliotecas de Classes



Fonte: Tom Archer, "Inside C#"

15

Objectivos do C#

16

Objectivos do C#

- Orientação a componentes
- Orientação a objectos
- Robustez
- Preservar investimentos

17

Orientação a Componentes *

- Apresentada como primeira linguagem "Orientada por Componentes" da família C/C++
- Considerações de evolução de versões pensadas na linguagem
- Componentes auto-descritivos
  - Metadados, incluindo atributos definidos pelo utilizador, consultados em tempo de execução através de reflexão
  - Documentação integrada em XML
- Suporte para propriedades, métodos e eventos
- Programação simples
  - Não existem .h, IDL, etc.
  - Pode ser utilizada em páginas ASP

18

Orientação a Objectos *

- Tudo é um objecto
- Herança simples de implementação e herança múltipla de interface (como em Java)
- Polimorfismo *a la carte* com métodos virtuais (como em C++)
- Membros estáticos (partilhados) e de instância (como C++ e Java)
- Vários tipos de membros:
  - campos, métodos, construtores e destrutores:
  - propriedades, indexadores, eventos e operadores (como C++)
- Não tem *templates*

19

Robustez *

- Gestão automática de memória *(garbage collection)*
  - Elimina problemas com fugas de memória e apontadores inválidos
  - Mas quem quiser pode trabalhar directamente com apontadores
- Excepções
  - Melhor tratamento de erros
- Segurança de tipos (*type-safety*)
  - Elimina variáveis não inicializadas, coerção insegura de tipos, etc.

20

Preservar Investimentos *

- Semelhanças com C++ e Java
  - Espaços de nomes, apontadores (em código "unsafe"), etc.
  - Nenhum sacrifício desnecessário
- Interoperabilidade
  - Cada vez mais importante
  - C# fala com XML, SOAP, COM, DLLs, e qualquer linguagem do .NET *Framework*
- Milhões de linhas de código C# no .NET
  - Pequena curva de aprendizagem
  - Melhor produtividade

21

Sistema de Tipos

22

Tipos Valor e Tipos Referência

- Tipos valor (*value types*)
  - Variáveis contêm directamente dados/instâncias
    - Não podem ser **`null`** (**`Nothing`** em VB)
    - Comparação e atribuição operam com os próprios valores (em C#)
    - Manipulação eficiente porque dados podem ser alocados na *stack*
- Tipos referência (*reference types*)
  - Variáveis contêm referências para objectos/instâncias (no *heap*)
    - Podem ser **`null`** (**`Nothing`** em VB)
    - Comparação e atribuição operam com as referências (em C#)
    - Gestão automática de memória (*garbage collector* do CLR)

```
int i = 123;
string s = "Hello, world";
```

23

Tipos Valor e Tipos Referência

- Tipos Valor
  - **Primitivos** int i; (de facto também são estruturas!)
  - **Enumerações** enum State { Off, On }
  - **Estruturas** struct Point { int x, y; }
- Tipos Referência
  - ***Arrays*** string[] a = new string[10];
  - **Classes** class Foo: Bar, IFoo {...}
  - **Interfaces** interface IFoo: IBar {...}
  - **Delegados** delegate double MathFunc(double x);

24

Tipos Pré-definidos em C#

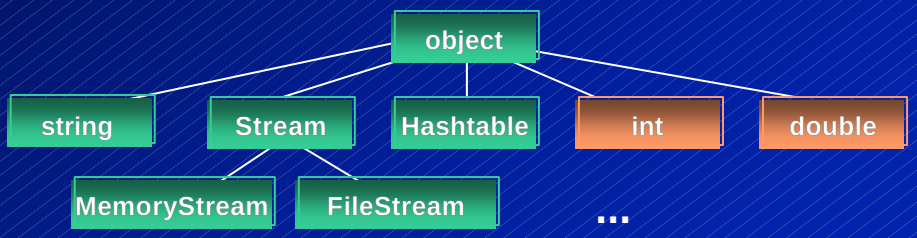
	CTS Type Name	C# Alias	Description
Classes	System.Object	object	Base class for all CTS types
	System.String	string	String
Estruturas	System.SByte	sbyte	Signed 8-bit byte
	System.Byte	byte	Unsigned 8-bit byte
	System.Int16	short	Signed 16-bit value
	System.UInt16	ushort	Unsigned 16-bit value
	System.Int32	int	Signed 32-bit value
	System.UInt32	uint	Unsigned 32-bit value
	System.Int64	long	Signed 64-bit value
	System.UInt64	ulong	Unsigned 64-bit value
	System.Char	char	16-bit Unicode character
	System.Single	float	IEEE 32-bit float
	System.Double	double	IEEE 64-bit float
	System.Boolean	bool	Boolean value (true/false)
	System.Decimal	decimal	128-bit data type exact to 28 or 29 digits mainly used for financial applications

```
int i = 1;
string s = i.ToString() + 123.ToString();
```

25

Tudo é um Objecto

- Todos os tipos (mesmo tipos valor) derivam de **`object`** e podem ser tratados como objectos
- Concilia eficiência com simplicidade
  - instâncias de tipos valor são guardadas na *stack*
  - não há *wrapper classes* com em Java (graças a mecanismo de *boxing* e *unboxing* – ver a seguir)
  - colecções funcionam com todos os tipos



26

Boxing e Unboxing

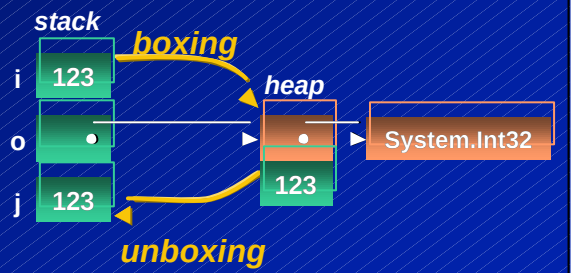
- *Boxing* – conversão de tipo valor para tipo referência
- *Unboxing* – conversão de tipo referência para tipo valor

```
int i = 123;

// upcast
object o = i;

// downcast explícito
int j = (int)o;

// outra forma
int j = o as int;
```



27

Espaços de Nomes (*namespaces*)

- Programa constituído por declarações de tipos (classes, interfaces, estruturas, etc.) dentro de espaços de nomes
  - código escrito *inline;* ordem de declaração irrelevante
- Um espaço de nomes pode conter tipos (classes, estruturas, etc.) e outros espaços de nomes
- Tipos declarados fora de qualquer espaço de nomes ficam num espaço de nomes global
- Vários ficheiros de código fonte podem contribuir para o mesmo espaço de nomes, mas cada tipo tem de ser definido num único ficheiro
- Para minimizar conflitos, é aconselhável usar o nome da empresa seguido do produto ou projecto (exemplo: Novabase.CSI.LockMngr)
- Instruções:
  - `using` ***nome-de-espaço-de-nomes;***
  - `using` ***alias = nome-de-classe;***
  - `namespace` ***nome-de-espaço-de-nomes***
    { declarações-de-*tipos-e-espaços-de-nomes* }

28

Espaços de Nomes
Exemplo *

```
namespace N1   // nome completo: N1
{
  class C1     // nome completo: N1.C1
  {
    class C2   // nome completo: N1.C1.C2
    { }
  }
  namespace N2 // nome completo: N1.N2
  {
    class C2   // nome completo: N1.N2.C2
    { }
  }
}
namespace N1.N2
{
  class C3    // N1.N2.C3
  { }
}
```

```
using MyC2 = N1.N2.C2;
using N1;
class myClass
{
   C1 c1;   // N1.C1
   MyC2 c2; // N1.N2.C2
}
```

29

Classes

30

Classes

- { Membros }
  - Campos, métodos, constantes, propriedades, indexadores, operadores, construtores, destrutores, eventos, tipos *nested*
  - Âmbito: estáticos (partilhados) ou de instância
  - Acessibilidade: `private` (por omissão), `protected`, `public` ou `internal` (mesma unidade de compilação; combinação ou c/ `protected`)
- : Bases
  - herança de 0 ou 1 uma classe base
    - por omissão é `System.Object`
    - são herdados todos os membros excepto construtores e destrutores
  - implementação de 0 ou mais interfaces base
- Modificadores
  - Acessibilidade: `public, private, protected, internal`
  - ...

31

Campos (*fields*)
Conceitos

- Guardam dados
- Âmbito:
  - campos de instância – alocados em cada instância da classe, para guardar dados da instância
  - campos estáticos (`static`) ou partilhados (VB) – alocados uma vez para toda a classe, para guardar dados partilhados por todas as instâncias
- Campo imutável (`readonly`) – guarda valor imutável definido pelo construtor (ou inicializador) no momento da execução
  - podem ser estáticos
- Podem ter inicializador
  - senão são inicializados com valor por omissão do respectivo tipo

32

Campos (*fields*)
Exemplo

```
using Date = System.DateTime;

public class Factura {
  // campos (private por omissão)
  static int ultimoNumero = 0;
  readonly int numero = ++ultimoNumero;
  Date data  = Date.Today;
  decimal valor;

  public Factura(decimal valor) // constructor
  { this.valor = valor; }

  public override string ToString() // método
  { return "Factura número=" + numero.ToString()
           + " data=" + data.ToShortDateString()
           + " valor=" + valor.ToString("C"); }
}
```

33

Campos (*fields*)
Exemplo (cont.)

```
class FacturaApp
{
   public static void Main()
   {
       System.Console.WriteLine(new Factura(10000));
       System.Console.WriteLine(new Factura(20000));
   }
}
```

```
Factura número=1 data=23-07-2001 valor=10.000$00 Esc.
Factura número=2 data=23-07-2001 valor=20.000$00 Esc.
```


34

Métodos
Conceitos

- Âmbito:
  - Estáticos (partilhados) (`static`)
    - invocados com *`classe.método`*
  - De instância (por omissão)
    - invocados com *`objecto.método`*
    - objecto é parâmetro implícito (`this`)
- *Overloading*
  - vários métodos com o mesmo nome e diferentes tipos de parâmetros

35

Métodos
Tipos de Parâmetros

- Parâmetros valor, só de entrada (*default*)
- Parâmetros referência de entrada e saída (`ref`)
  - obrigatório passar variáveis já inicializadas
- Parâmetros referência só de saída (`out`)
  - permite passar variáveis não inicializadas

```
class Calculator
{
  private double mem;
  public void Store(double x)    { mem = x; }
  public void Load(out double x) { x = mem; }
  public void Exchange(ref double x)
    { double old_mem = mem; mem = x; x = old_mem;}
}
```

36

Métodos
Argumentos Variáveis *

- Métodos podem aceitar um nº variável de argumentos do mesmo tipo (no limite `object`, i.e., qualquer tipo)
- Declara-se parâmetro do tipo *array* com `params`
  - Podem existir parâmetros anteriores normais (sem `param`)
  - Podem-se passar vários argumentos ou um array

```
class VarArgsApp
{ public static double Sum(params double[] nums)
  { double sum = 0;
    foreach (double x in nums)  sum += x;
    return sum;
  }
  public static void Main()
  { System.Console.WriteLine(Sum(1.0, 2.0, 3.0)); }
}
```

37

Herança e "Substitutabilidade"

```
using System;

class C1 {
 public void F1()
 {Console.WriteLine("F1");}
}

class C2: C1 {
 public void F2()
 {Console.WriteLine("F2");}
}

class C3: C2 {
 public void F3()
 {Console.WriteLine("F3");}
}
```

Herda implicitamente de System.Object

Herda membros de C1 e acrescenta F2

```
class InheritanceApp {
 public static
 void Main(){
  C1 c1 = new C1();
  C2 c2 = new C2();
  C3 c3 = new C3();
  // herança
  c2.F1();
  c3.F1();
  c3.F2();
  // substitutabilidade
  c2 = c3;
  c1 = c2;
  c1 = c3;
}
```

Onde se espera um objecto da classe base (C1) pode-se passar um objecto de uma classe derivada (C3)

38

Upcast e *Downcast* *

- *Upcast* – conversão de classe derivada para classe base (ao nível de referências)
  - Sempre possível de acordo com princípio da "substitutabilidade"
  - Não precisa de *cast* explícito
- *Downcast* – conversão de classe base para classe derivada (ao nível de referências)
  - Tem de se fazer um *cast* explícito com **`(type)`** ou **`as type`**
  - Só é possível se objecto referenciado for da classe derivada (ou de uma terceira classe derivada desta)
    - Se assim não for,
      - `"(type)objref"` dá excepção `System.InvalidCastException`
      - `"objref as type"` dá `null`

39

Métodos Virtuais e Polimorfismo

```
using System;

class A {
  public virtual void F() {Console.WriteLine("A.F");}
}

class B: A {
  public override void F() {Console.WriteLine("B.F");}
}

class PolymorphismApp {
  public static void Main(){
    B b = new B();
    A a = b;
    a.F(); // B.F
    b.F(); // B.F
  }
}
```

pode ser *overriden*

substitui implementação de método virtual herdado, para objectos da classe derivada

Utilização explícita de `override` reduz o problema da classe base frágil

método chamado depende do tipo do objecto referenciado (determinado em tempo de execução – *late binding*) e não do tipo da variável que guarda a referência (determinado em tempo de compilação – *early binding*)

40

Esconder Membros Herdados

- Modificador **new** – usado na definição de um novo membro (método, tipo *nested*, etc.) que esconde um membro herdado com o mesmo nome (e assinatura, no caso de métodos e similares)
  - Utilização explícita de **new** reduz problema da classe base frágil

```
class A { public virtual void F() { ... } }
class B: A { public new void F() { ... } }

class DemoApp {
  public static void Main(){
    B b = new B();
    A a = b;
    a.F(); // chama A.F (não há polimorfismo)
    b.F(); // chama B.F
  }
}
```

41

Classes e Métodos Abstractos

- Classe abstracta: não pode ser instanciada (só classes derivadas podem ser instanciadas)
- Método abstracto: não tem implementação na classe em que é declarado (só em classes derivadas)
  - Classe tem de ser abstracta
  - É implicitamente virtual (mas não leva `virtual`)

```
public abstract class Shape {
  public abstract void Resize(double factor);
  public void DoubleSize() { Resize(2.0); }
}

public class Box: Shape {
  public override void Resize(double factor)
  { ... }
}
```


42

Classes e Métodos Selados

- Classe selada: não pode ser usada como base doutra classe

```
sealed class X {}

class Y : X {} // Erro!
```

- Método selado: não pode ser *overriden*

```
class A { public virtual void F() {} }

class B : A { public sealed override void F() {} }

class C: B { public override void F() {} } // Erro!
```

43

Construtores de Instância

- São métodos com mesmo nome da classe, usados para inicializar novas instâncias da classe
- Podem ter parâmetros (passados a **new**)
- Podem ser *overloaded*
- Não têm valor de retorno (mas não levam `void`)
- Podem invocar no inicializador um construtor da classe base `base(...)` *ou* outro construtor da mesma classe `this(...)` - por omissão, é invocado `base()`

```
class A
{
  private int x;
  public A(int x)
  { this.x = x; }
}
```

```
class B : A
{
  public B(int x) : base(x)
  { }
}
```

inicializador

44

Construtores Estáticos *

- Cada classe pode ter um construtor estático (`static`) para inicializar a classe (normalmente inicializar campos estáticos)
- Construtor estático não tem tipo de retorno nem parâmetros

45

Destrutores *

- São métodos com nome da classe precedido de "~", usados para "destruir" um objecto que vai ser libertado de memória
  - São chamados pelo *garbage collector*
  - Um objecto pode ser libertado de memória a partir do momento em que não pode ser usado por nenhuma parte do código
- Destrutores não têm parâmetros nem valor de retorno (mas não levam `void`)
- Correspondem a método `Finalize` no CLR, implementado desde `System.Object`

46

Constantes

- Valor constante definido no momento da compilação
  - Difere de campo imutável, cujo valor pode ser definido no momento da execução
- Sintaxe: campo com prefixo **`const`** e inicialização
- São membros estáticos (partilhados), mas não levam `static`

```
namespace System
{
  public sealed class Math
  {
    public const double PI = 3.14159265358979323846;
    public const double E = 2.71828182845904523354;
    ...
  }
}
```

47

Propriedades

- São "smart fields"
  - Usados como campos
  - Implementados por métodos **`set`** e/ou **`get`**
  - Podem ser virtuais, abstractos, estáticos, etc. (como métodos)

```
public class Button: Control
{
   private string caption;

   public string Caption {
      get { return caption; }
      set { caption = value; Repaint(); }
   }
}
```

```
Button b = new Button();
b.Caption = "OK";
string s = b.Caption;
```


48

Indexadores (*indexers*)

- São "smart arrays"
  - Permitem usar objectos como *arrays*
  - Implementados por métodos **`set`** e/ou **`get`**
  - Podem ser *overloaded*
  - Podem ser virtuais, abstractos, etc. (como métodos)
  - Mapeados para propriedades com argumentos no CTS

```
public class ListBox: Control {
   private string[] items;
   public string this[int index] {
      get { return items[index]; }
      set { items[index] = value; Repaint(); }
   }
}
```

```
ListBox listBox = new ListBox();
listBox[0] = "hello";
Console.WriteLine(listBox[0]);
```

49

Estruturas, Interfaces, Enumerações e *Arrays*

50

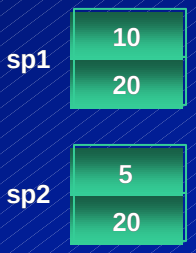
Estruturas

- Como classes, excepto
  - São tipos-valor em vez de tipos-referência
    - Atribuição copia valor (dados) em vez de referência
  - Sem herança (mas herdam implicitamente de `object` e podem implementar interfaces)
  - Sem destrutores
  - Campos de instância não podem ter inicializadores
- Ideal para objectos leves
  - int, float, double, etc., são estruturas
  - complex, point, rectangle, color, ...
- Utilização mais eficiente da memória

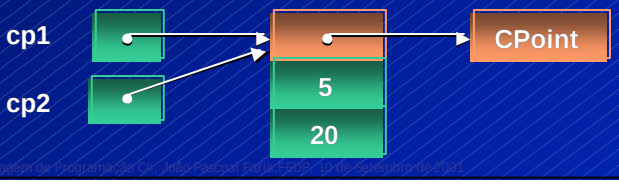
51

Estruturas *versus* Classes

```
struct SPoint { int x, y; ... }
SPoint sp1 = new SPoint(10, 20);
Spoint sp2 = sp1;
sp2.setX(5);
```




```
class CPoint { int x, y; ... }
CPoint cp1 = new CPoint(10, 20);
Cpoint cp2 = cp1;
cp2.setX(5);
```



52

Interfaces

- Um interface define um contracto ao qual tem de aderir qualquer classe ou estrutura que o implemente
- Membros: métodos, propriedades, eventos e indexadores
  - sem implementação
    - a fornecer por classes ou estruturas que implementam o interface
  - implicitamente públicos e de instância (não estáticos)
  - pode-se usar `new` para esconder membro herdado
- Herança múltipla:
  - um interface pode herdar de múltiplos interfaces
  - uma classe ou estrutura pode implementar múltiplos interfaces
- Nome: habitual começar com letra "**`I`**"
- Modificadores: acessibilidade e `new` (em tipos *nested*)

53

Interfaces
Exemplo

```
public delegate void StringListEvent(IStringList sender);

public interface IStringList {
  void Add(string s);
  int Count { get; }
  event StringListEvent Changed;
  string this[int index] { get; set; }
}

public class StringList : IStringList {
  private string[] strings;
  public void Add(string s) { ... }
  public int Count { get {return strings.Length} }
  public event StringListEvent Changed;
  public string this[int index] { get{...} set{...} }
}
```

54

Interfaces
Herança entre Interfaces

```
interface IControl {
  void Paint();
}
interface ITextBox: IControl {
  void SetText(string text);
}
interface IListBox: IControl {
  void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}

class MyComboBox :  IComboBox {
  public void Paint() { ... }
  public void SetText(string text) { ... }
  public void SetItems(string[] items) { ... }
}
```

55

Interfaces
Implementação explícita *

```
interface IDataBound
{
   void Bind(IDataBinder binder);
}

class EditBox: Control, IDataBound
{
   void IDataBound.Bind(IDataBinder binder) {...}
}
```

Nome completamente qualificado (com prefixo que indica interface em que foi especificado) – só pode ser acedido através de instância do interface

56

Enumerações

- Definem conjuntos de constantes simbólicas
- Conversão de/para inteiro, mas só explícita
- Operadores aplicáveis:
  - comparação: `==  >  <  >=  <=  !=`
  - bit-a-bit: `&  |  ^  ~`
  - outros: `+  -  ++  --  sizeof`
- Pode especificar-se o tipo subjacente (por omissão `int`)
  - `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` ou `ulong`
- Herdam implicitamente de `System.Enum`

```
enum Color: byte {
   Red   = 1,
   Green = 2,
   Blue  = 4,
   Black = 0,
   White = Red | Green | Blue
}
```

```
Color c = Color.Red;
```

57

Arrays uni-dimensionais

- Declaração com tipo de *array* e nome:
  - `int[] a; //` array uni-dimensional de elementos do tipo `int`
- Criação (alocação):
  - `a = new int[4]; // cria array de 4 inteiros`
  - Tamanho é determinado ao criar (em tempo de execução)
- Índices são inteiros e começam em 0
- Inicializados com valores por omissão do respectivo tipo, ou com valores (constantes) dados explicitamente:
  - `new int[] {1, 2, 3};` ou: `new int[3] {1, 2, 3};`
- Conversão entre *arrays* do tipo A[] e B[] realiza-se nas mesmas condições que conversão entre tipos A e B
  - `object[]` aceita qualquer *array* uni-dimensional

58

Arrays de Arrays *

- Declaração
  - `int[][] a; //`array uni-dimensional de elem's do tipo `int[]`
- Utilização
  - `int[] ai = a[i]; int aij = a[i][j];`
- Sub-arrays têm de ser criados um a um
  - `int[][] a = new int[100][5]; // Erro`
  - ```
    int[][] a = new int[100][];  // Ok
    for (int i = 0; i < 100; i++)
       a[i] = new int[5];
    ```
- Sub-arrays podem ter tamanhos diferentes
  - ```
    int[][] matrizTriang = new int[10];
    for (int i = 0; i < 10; i++)
         matrizTriang[i] = new int[i+1];
    ```

59

Arrays multi-dimensionais *

- Declaração
  - `int[,] m; //array bi-dimensional de int's`
- Criação
  - `m = new int[4,3]; // array 4x3`
- Inicialização
  - `new int[,] {{0, 1}, {2, 3}, {4, 5}};`
  - `new int[3, 2] {{0, 1}, {2, 3}, {4, 5}};//equiv.`
- Conversão
  - Só entre *arrays* com o mesmo nº de dimensões
  - Tipo `object[,]` aceita qualquer *array* bi-dimensional

60

Arrays
Métodos e propriedades

- Herdam implicitamente de `System.Array`
  - Métodos estáticos: **`BinarySearch`**, **`Clear`**, **`Copy`**, **`IndexOf`**, **`LastIndexOf`**, **`Reverse`**, **`Sort`**, **`CreateInstance`**, **`...`**
  - Propriedades de instância: **`Length`** (dá o nº total de elementos em todas as dimensões do array), **`Rank`** (nº de dimensões do array)
  - Métodos de instância: **`Clone`**, **`Equals`**, **`GetLength`** (nº de elementos numa dada dimensão), **`...`**

61

Delegados e Eventos

62

Delegados *(Delegates)*

- São apontadores para funções orientados por objectos
- Tipo de delegado
  - Sintaxe: assinatura de método com a palavra chave **`delegate`**
  - Implicitamente derivado de `System.Delegate`
- Instância de delegado
  - Encapsula zero ou mais entidades invocáveis - métodos estáticos ou de instância, com assinatura idêntica à definida no tipo de delegado
  - Usado como objecto (em atribuições, passagem de parâmetros, etc.) ou como método (para invocar indirectamente as entidades nele encapsuladas)

63

Delegados
Exemplo

```
using System;

class DelegateApp {
  // Declaração do tipo de delegado (assinatura)
  public delegate void IntFunc(int n);

  // Método a chamar indirectamente
  static void F(int n)
  { Console.WriteLine("F called with n={0}", n); }

  public static void Main() {
    // Instanciação do delegado
    IntFunc func = new IntFunc(F);

    // Invocação indirecta de método
    func(1);
  }
}
```


64

Delegados *
Composição e *multicasting*

```
class MulticastingApp
{
   delegate void IntFunc(int n);

   static void F1(int n)
   { System.Console.WriteLine("F1 called with n={0}", n); }

   static void F2(int n)
   { System.Console.WriteLine("F2 called with n={0}", n); }

   public static void Main() {
      // Instanciação do delegado com composição
      IntFunc funcs = (new IntFunc(F1)) + (new IntFunc(F2));

      // Chamada indirecta de F1(1) seguido de F2(1)
      // (multicasting)
      funcs(1);
  }
}
```

65

Eventos
Definir e disparar

```
// tipo de delegado que define a assinatura do evento
public delegate void EventHandler(object sender,
                                  EventArgs e);

public class Button {
   // campo que define o evento e memoriza handlers
   public event EventHandler Click;

   // método que dispara o evento
   // (normalmente escondido de clientes)
   protected void OnClick(EventArgs e) {
      if (Click != null)
               Click(this, e); // dispara aqui
   }
}
```

ou classe derivada

66

Eventos
Registar *handler* e tratar

```
using System.Windows.Forms;
public class MyForm: Form {
  Button okButton;
  public MyForm() {
    okButton = new Button();
    okButton.Text = "OK";
    okButton.Click += new EventHandler(OkButtonClick);
    Controls.Add(okButton);
    Show();
  }
  void OkButtonClick(object sender, EventArgs e)
  { MessageBox.Show("You pressed the OK button"); }
  public static void Main(string[] args)
  {Application.Run(new MyForm());}
}
```

regista *handler*

trata evento

67

Eventos
Detalhes *

- A assinatura do evento (ou melhor, dos métodos que tratam o evento) é definida por um delegado com dois parâmetros:
  - o objecto que disparou o evento
  - um objecto da classe `System.EventArgs` ou classe derivada, com informação sobre o evento
- O evento é definido num membro com a palavra-chave **`event`**, seguida do nome do tipo de delegado que define a assinatura do evento, seguido do nome do evento
  - esse campo guarda a lista de métodos que tratam o evento (*handlers*)
- Registar um método que trata o evento:
  - ***`evento`*** **`+= new`** ***`delegate-type`*****`(`*****`método`*****`)`**
- Disparar evento: ***`evento`*****`(`*****`obj,args`*****`)`**
  - métodos registados são invocados sequencialmente
  - previamente testa-se se não é **`null`** (i.e., se há subscritores)

68

Operadores e Instruções

69

Operadores e Instruções

- São basicamente os mesmos do C/C++, com alguns melhoramentos para evitar erros comuns
  - Instruções sem qualquer efeito são proibidas

```
i == 0; // Instrução não válida!
```

  - Condições em `if`, `while`, `for` têm de ser do tipo `bool` ou de um tipo convertível explicitamente para `bool` (não há conversão implícita de `int` para `bool`)

```
int i = 1;
if (i) ...       // Errado!
if (i != 0) .... // Certo!
```


70

Instrução `switch` *

- Não faz *fall-through*, requer "**`goto case`**" ou "**`goto default`**"

```
switch(opcao)
{
  case 'a':
  case 'A': // Certo
     adicionar();
     goto case 'R'; // Dá erro se não tiver isto!
  case 'R':
     refrescar();
     break; // Opcional no último case ou default
}
```

71

Instrução foreach

- `foreach (`*`type identifier`* `in` *`expression`*`)`
  `embedded-statement`
- Iteração sobre arrays

```
public static void Main(string[] args)
{
   foreach (string s in args)
      Console.WriteLine(s);
}
```

- Iteração sobre colecções

```
void Display(System.Collections.ArrayList words)
{
   foreach (string s in words)
      Console.WriteLine(s);
}
```


72

Tratamento de Excepções

73

Tratamento de Excepções
Lançar excepção

- **throw** *<instância de System.Exception ou de classe derivada>*

```
using System;
class Fraccao
{
  private int numerador, denominador;
  public Fraccao(int numerador, int denominador)
  {
     if (denominador == 0)
        throw new ArgumentException("denominador=0!");
    this.numerador = numerador;
    this.denominador = denominador;
  }
}
```

classe declarada no *namespace* System e derivada indirectamente de System.Exception

74

Tratamento de Excepções
Apanhar e tratar excepção

- **try** {instruções} **catch** (excepção)$_{opt}$ {instruções}

```
class FraccaoApp {
  public static void Main() {
     try { Fraccao f = new Fraccao(1,0); }
     catch(Exception e)
     {
        Console.WriteLine(e.Message);
        Console.WriteLine(e.StackTrace);
     }
  }
}
```

apanha excepções desta classe ou de classes derivadas

```
denominador=0!
   at Fraccao..ctor(Int32 numerador, Int32 denominador)
   at FraccaoApp.Main()
```

75

Tratamento de Excepções
Outras *features* *

- Podem-se ter vários blocos `catch` seguidos, para apanhar excepções de diferentes classes
  - blocos mais "especializados" devem estar depois de blocos de mais "genéricos" (senão nunca são executados)
- Pode-se relançar excepção com `throw` sem parâmetros
- Pode-se lançar nova excepção que inclui a original (como *inner exception*)
- Pode-se definir um bloco **`finally`**`{}` a seguir a `try` e `catch`, com código que é executado em qualquer caso
  - Pode-se ter `try{} finally{}` sem `catch`
- Excepção não apanhada ? aplicação abortada

76

Sobrecarga *(Overloading)* de Operadores

77

Overloading de operadores

- Pode-se (re)definir o significado de operadores do C# para classes e estruturas definidas pelo utilizador
- De forma muito semelhante a C++
- Conceito não suportado totalmente pelo CTS
  - mapeados para métodos op_*NomeDeOperador*

78

Operadores Unários e Binários
Exemplo

```
public class Frac {
  int numerador, denominador;
  public Frac(int num, int denom)
  { this.numerador = num; this.denominador = denom;}
  public static Frac operator * (Frac f1, Frac f2)
  {return new Frac(f1.numerador*f2.numerador,
                   f1.denominador*f2.denominador);}
  public static Frac operator * (Frac f, int c)
  {return new Frac(f.numerador * c, f.denominador);}
  public static Frac operator - (Frac f)
  { return new Frac(-f.numerador, f.denominador); }
}
```

```
Frac r1 = new Frac (1,3);
Frac r2 = new Frac (3,1);
Frac r3 = - r1 * r2 * 4;
```

79

Operadores Unários e Binários
Regras *

- Método com nome **`operator` *`símbolo-de-operador`*** (re)define esse operador para instâncias da classe ou estrutura
  - operador fica *overloaded* porque tem várias definições dependendo dos tipos dos parâmetros
- Operadores que podem ser *overloaded:*
  - unários: `+ - ! ~ ++ -- true false`
  - binários: `+ - * / % & | ^ << >> == != > < >= <=`
- Têm de ser `public` e `static`, c/ parâmetros por valor
- Pelo menos um dos parâmetros tem de ser da classe ou estrutura em que o operador é declarado

80

Operadores de Conversão
Exemplo

```
public struct Digit
{
  byte value;
  public Digit(byte value)
  { if (value < 0 || value > 9)
        throw new ArgumentException();
    this.value = value;
  }
  public static implicit operator byte(Digit d)
  { return d.value;}
  public static explicit operator Digit(byte b)
  { return new Digit(b);}
}
```

porque pode lançar excepção

```
Digit d1 = new Digit(4);
byte x = d1;  // OK
Digit d2 = x; // MAL
Digit d3 = (Digit)x; // OK
```

81

Operadores de Conversão
Regras *

- Método com nome **`operator`** ***`nome-de-tipo`*** especifica a conversão de instâncias da classe ou estrutura de/para outro tipo (classe ou estrutura)
  - de: tipo do parâmetro
  - para: tipo (do resultado) a seguir a palavra-chave `operator`
- Conversão pode ser **`implicit`** ou **`explicit`**
  - conversão explícita: invocada com operadores de *cast*
    - aconselhável quando há perda de informação ou lançamento de excepções
- Têm de ser `public` e `static`, c/ parâmetros por valor
- Não se podem redefinir conversões pré-definidas

82

Atributos

83

Atributos
Conceitos

- São anotações "tipadas", que podem ser associadas (como metadados) a diversos elementos de um programa (tipos, membros, parâmetros, valores de retorno, módulos e *assemblies*)
  - URL de documentação para uma classe
  - Nome de um campo em XML
  - Propriedades transaccionais de um método, etc.
- Vantagens: informação mais integrada e consistente, dispensa ficheiros externos (.IDL, .DEF, etc.), componentes auto-descritivos
- Valores de atributos indicados entre [ ] antes dos elementos a que se referem (entre < > em VB)
- Tipos de atributos são classes derivadas de `System.Attribute`
- Consultados em tempo de execução através de reflexão
- Usados intensivamente por muitos serviços do .NET (XML, COM,...)

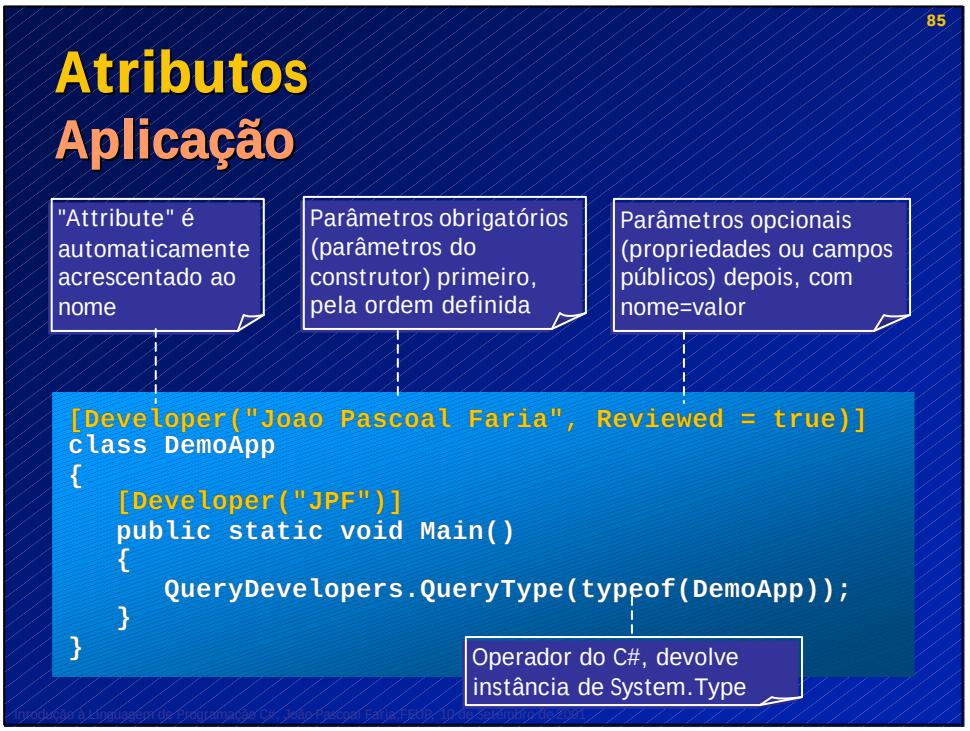
84

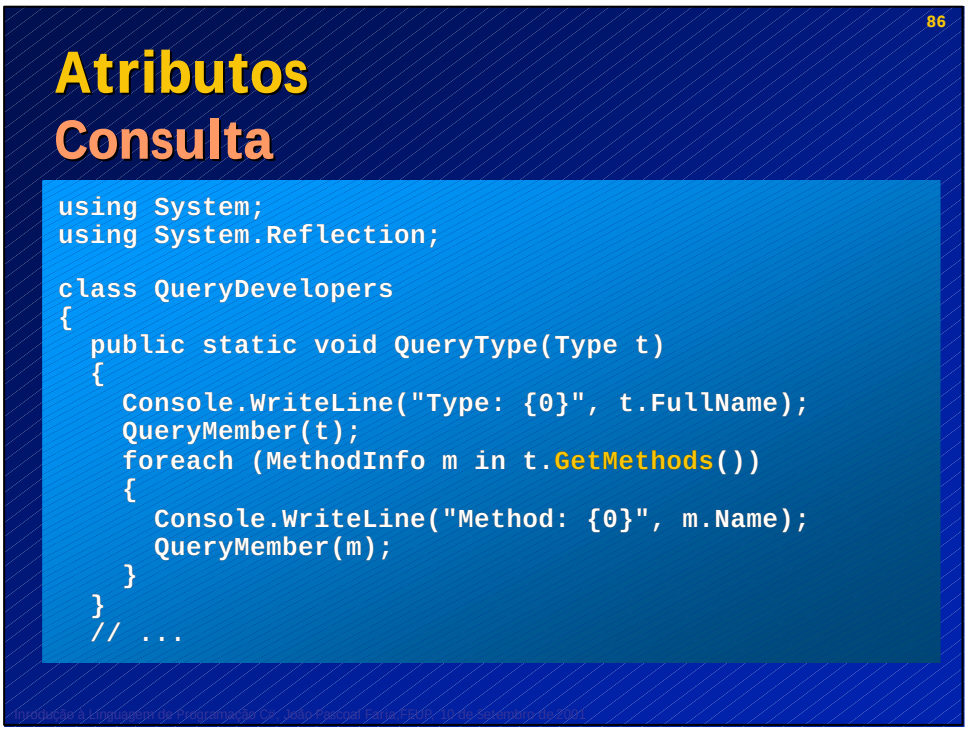
Atributos
Definição

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperAttribute : System.Attribute
{
  // Este construtor define um parâmetro obrigatório
  public DeveloperAttribute(string name)
  { this.name = name; }

  private string name;
  public string Name {
    get { return name; }
  }

  // Esta propriedade define um parâmetro opcional
  private bool reviewed = false;
  public bool Reviewed {
       get { return reviewed; }
       set { reviewed = value; }
  }
}
```


85
Atributos
Aplicação
"Attribute" é automaticamente acrescentado ao nome
Parâmetros obrigatórios (parâmetros do construtor) primeiro, pela ordem definida
Parâmetros opcionais (propriedades ou campos públicos) depois, com nome=valor
[Developer("Joao Pascoal Faria", Reviewed = true)]
class DemoApp
{
[Developer("JPF")]
public static void Main()
{
QueryDevelopers.QueryType(typeof(DemoApp));
}
}
Operador do C#, devolve instância de System.Type


86
Atributos
Consulta
using System;
using System.Reflection;
class QueryDevelopers
{
public static void QueryType(Type t)
{
Console.WriteLine("Type: {0}", t.FullName);
QueryMember(t);
foreach (MethodInfo m in t.GetMethods())
{
Console.WriteLine("Method: {0}", m.Name);
QueryMember(m);
}
}
// ...

87

Atributos
Consulta (cont.)

```
// ...
private static void QueryMember(MemberInfo m)
{
 foreach(Attribute a in m.GetCustomAttributes(false))
 {
  if (a is DeveloperAttribute) {
   DeveloperAttribute d = (DeveloperAttribute)a;
   Console.WriteLine("Developer: {0}", d.Name);
   Console.WriteLine("Reviewed : {0}", d.Reviewed);
  }
 }
}
```

```
Type: DemoApp
Developer: Joao Pascoal Faria
Reviewed : True
Method: GetHashCode
Method: Equals
...
```

```
...
Method: ToString
Method: Main
Developer: JPF
Reviewed : False
Method: GetType
```


88

Atributos *
Outras *features*

- Quando atributo não tem parâmetros, dispensam-se parêntesis
- Para resolver ambiguidade acerca do elemento a que se refere um atributo, pode-se usar um prefixo como em
  - `[return:HRESULT] public long F(){}`
- Meta-atributo `AttributeUsage`:
  - Valores possíveis de primeiro parâmetro: `Assembly`, `Module`, `Class`, `All`, etc.
  - Tem dois parâmetros opcionais:
    - `AllowMultiple` – se pode ser definido várias vezes para mesmo elemento
    - `Inherited` – se atributo é herdado (por classes derivadas, etc.)

89

Interoperação com código não gerido pelo .NET *runtime*

90

Casos de Interoperação

- **Platform Invocation Services (PInvoke)** – permitem a código gerido pelo CLR aceder a funções, estruturas e até *callbacks* em DLL's não geridas
- **Unsafe code** – possibilidade de escrever em C# código de "baixo nível" (com apontadores, etc.) que não é gerido pelo CLR
- **COM interoperability**:
  - código gerido pode usar componentes COM via *wrappers*
  - aplicações COM podem usar serviços .NET (não tratado aqui)
  - expor uma classe .NET como um componente COM (não tratado aqui)

91

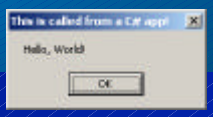
Platform Invocation Services

Invocação de método em DLL não gerida

```
// Exemplifica chamada de MessageBoxA de user32.dll
using System;
using System.Runtime.InteropServices; //DllImport
class PInvoke1App {
  [DllImport("user32.dll")]
  static extern int MessageBoxA(int hWnd, string msg,
                                string caption, int type);

  public static void Main() {
    MessageBoxA(0, "Hello, World!",
                "This is called from a C# app!", 0);
  }
}
```

Implementado externamente



92

Platform Invocation Services

Passagem de *callback* a DLL não gerida *

```
using System;
using System.Runtime.InteropServices;
using StringBuilder = System.Text.StringBuilder;
class CallbackApp {
 [DllImport("user32.dll")] static extern int GetWindowText
                  (int hWnd, StringBuilder text, int count);
 delegate bool CallbackDef(int hWnd, int lParam);
 [DllImport("user32.dll")] static extern int EnumWindows
                       (CallbackDef callback, int lParam);
 static bool PrintWindow(int hWnd, int lParam) {
   StringBuilder text = new StringBuilder(255);
   GetWindowText(hWnd, text, 255);
   Console.WriteLine("Window caption: {0}", text);
   return true; }
 static void Main()
 { EnumWindows(new CallbackDef(PrintWindow), 0); }
}
```

```
Window caption: Microsoft PowerPoint - [C#.ppt] ...
```

93

Platform Invocation Services

Outras *features* de `DllImport` *

- Pode-se dar um nome local ao método
  - ```
    [DllImport("user32.dll",EntryPoint="MessageBoxA")]
    static extern int MsgBox( ....);
    ```
- Pode-se especificar o conjunto de caracteres (Ansi ou Unicode)
  - ```
    [DllImport("user32.dll",CharSet=CharSet.ANSI")]
    static extern int MessageBox( ....);
    ```
- Pode-se especificar a forma como é feito o *marshaling* de cada parâmetro e do valor de retorno
  - ```
    [DllImport("user32.dll", CharSet=CharSet.Unicode)]
    static extern int MessageBox(int hWnd,
       [MarshalAs(UnmanagedType.LPWStr)] string msg,
       [MarshalAs(UnmanagedType.LPWStr)] string caption,
       int type);
    ```


94

Código *unsafe* *

Apontadores

- É possível trabalhar com apontadores para tipos valor, *strings* e *arrays* de tipos valor dentro de código (método ou bloco) marcado com **`unsafe`**
- Operadores como em C/C++:
  - `&` endereço
  - `*` acesso a objecto apontado
  - `->` acesso a membro de objecto apontado
  - `++ --` aritmética de apontadores (com *arrays* e *strings*)

```
class UnsafeDemo {
  public static unsafe void Main() {
    int a = 1; int *aPtr = &a; *aPtr = 2;
    System.Console.WriteLine(a); }
}
```

Compilar com /unsafe
```

95

Código *unsafe* *

Alocação de *arrays* na *stack*

- *type* * *ptr* = **stackalloc** *type* [*nelems*];

```
class StackallocDemo
{
  public static unsafe void Main() {
    const int size = 80;
    long* fib = stackalloc long[size];
    long* p = fib;
    *p++ = *p++ = 1;
    for (int i=2; i<size; ++i, ++p)
      *p = p[-1] + p[-2];
    for (int i=0; i<size; ++i)
      System.Console.WriteLine (fib[i]);
  }
}
```

Ligeiramente mais eficiente do que alocar no *heap*!

96

Código *unsafe* *

Apontadores para objectos geridos

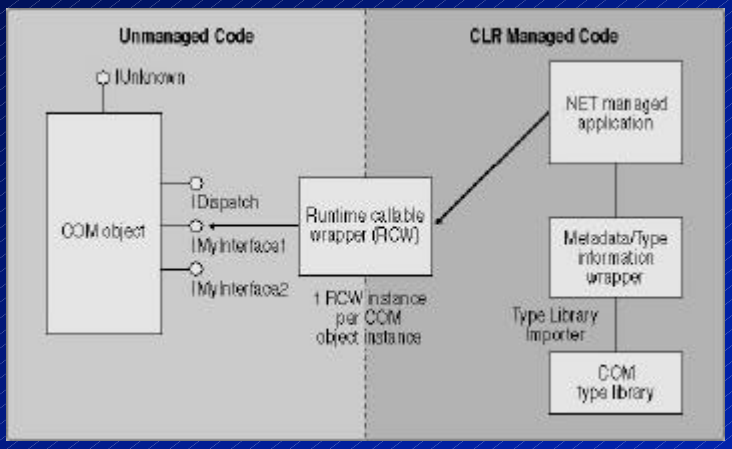
- Para usar endereço de componente de objecto gerido pelo *garbage collector* (GC), é necessário garantir que o mesmo não é movido pelo GC, o que se faz com a instrução:
  - **fixed** (*type* * *ptr* = *expression*) *statement*

```
class FixedApp {
 class A {public int x;}
 public unsafe static void Main()
 {
    A a = new A();
    fixed(int *p = &a.x) *p = 1;
    System.Console.WriteLine("a.x={0}",a.x);
  }
}
```

Objecto "a" não é movido durante a execução da instrução

97

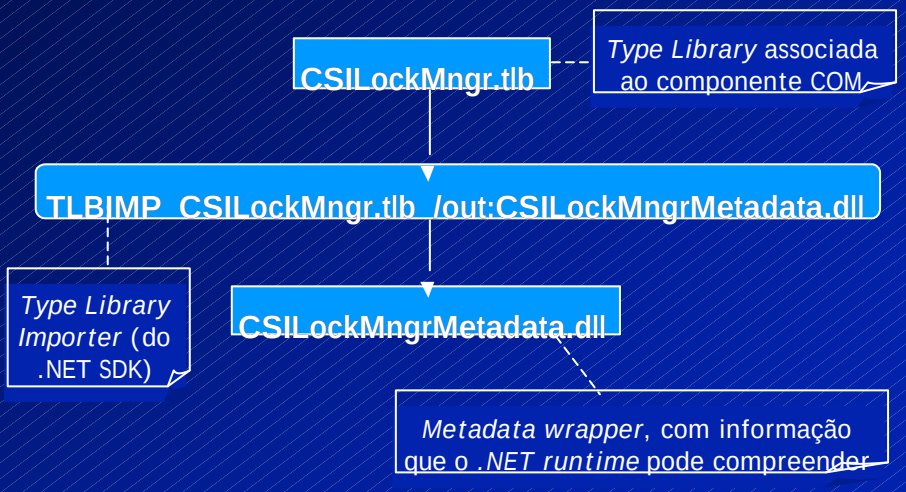
Invocação de componentes COM
Visão geral



Fonte: Tom Archer, "Inside C#"

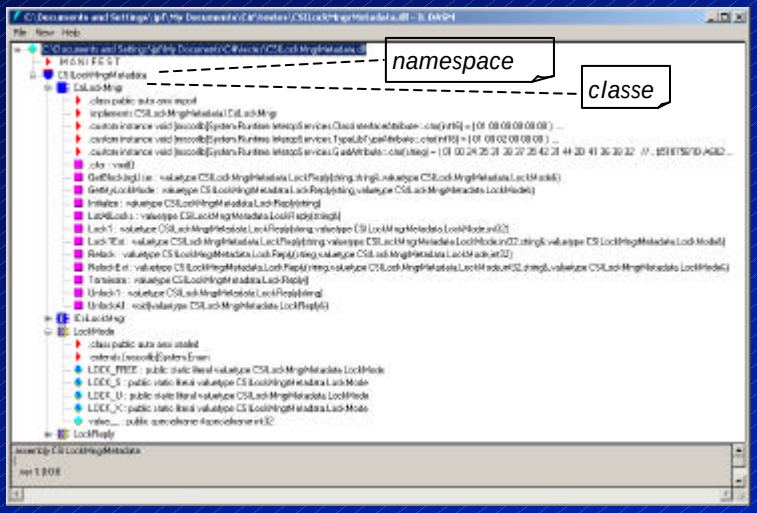
98

Invocação de componentes COM
Importação da *type library* com TLBIMP



99

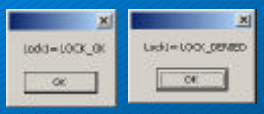
Invocação de componentes COM
Visualização do resultado com ILDASM



100

Invocação de componentes COM
Early binding

```
using System.Windows.Forms;
using CSILockMngrMetadata; //CsiLockMngr,LockMode,...
using System.Runtime.InteropServices; //COMException
public class CsiLockMngrCLientApp {
 public static void Main() {
  try {
   CsiLockMngr lm = new CsiLockMngr();
   LockReply reply = lm.Initialize("JPF");
   reply = lm.Lock1("recurso", LockMode.LOCK_X, 0);
   MessageBox.Show("Lock1= " + reply.ToString());
  }
  catch(COMException e) {
   Console.WriteLine("COM error message={0} code={2}",
                     e.Message,e.ErrorCode); }
 }
}
```



compilar com: /reference:CSILockMngrMetadata.dll

101

Invocação de componentes COM
*Late binding **

```
...
using System.Reflection; // BindingFlags

  ...
  Type t = Type.GetTypeFromProgID(
                        "CsiLockMngr.CsiLockMngr");
  object lm = Activator.CreateInstance(t);
  ...
  object[] args = {"recurso", LockMode.LOCK_X, 0};
  LockReply reply = (LockReply)t.InvokeMember("Lock1",
     BindingFlags.Default | BindingFlags.InvokeMethod,
     null, lm, args);
  ...
 }
}
```


102

Outros Tópicos

103

Documentação em XML

- Compilador gera automaticamente documentação a partir do código fonte, utilizando os comentários XML (///)
  - e verifica consistência entre comentários e código!

```
class XmlElement
{
   /// <summary>
   ///    Returns the attribute with the given name and
   ///    namespace</summary>
   /// <param name="name">
   ///    The name of the attribute</param>
   /// <param name="ns">
   ///    The namespace of the attribute, or null if
   ///    the attribute has no namespace</param>
   /// <return>
   ///    The attribute value, or null if the attribute
   ///    does not exist</return>
   /// <seealso cref="GetAttr(string)"/>
   ///
   public string GetAttr(string name, string ns)
   {   ...   }
}
```

```
csc XmlElement.cs /doc:XmlElement.xml
```

104

Compilação Condicional *

- #define, #undef
- #if, #elif, #else, #endif
  - Lógica booleana simples
- Métodos condicionais – mais prático

```
#define DEBUG
using System;
using System.Diagnostics; // tem ConditionalAttribute
public class Trace {
  [Conditional("DEBUG")] public static void Msg(string msg)
  {Console.WriteLine(msg);}
}
class Test {
  public static void Main()
  {Trace.Msg("Now in Main."); Console.WriteLine("Done.");}
}
```

Método e chamadas não são compilados se #define DEBUG for retirado

105

Assemblies
Conceitos

- *Assembly*
  - empacotamento num ficheiro físico (.dll ou .exe) de um manifesto, código MSIL de um ou mais tipos (classes, interfaces, etc.), e zero ou mais recursos (bitmaps, JPEGs, etc.)
  - unidade fundamental de construção, *deployment*, controlo de versões e controlo de segurança no *framework* .NET
  - auto-descritivo e portável
- Módulo
  - produto intermédio da compilação, com extensão ".netmodule", a adicionar a um *assembly* (faz as vezes de um .obj)
- Aplicação
  - constituída por um ou mais *assemblies*, distribuídos individualmente ou agrupados em ficheiros .CAB ou .msi

106

Assemblies
Controlo de versões

- Nº de versão composto por 4 segmentos: <major>.<minor>.<build>.<revision>
- Nº de versão definido no código fonte com atributo `[assembly:AssemblyVersion("1.1.0.0")]`
- O manifesto de um *assembly* contém o nº de versão e a lista de *assemblies* referenciados e respectivas versões
- Política de versões por defeito: usar versão com nº mais alto que coincide nos 2 primeiros segmentos (para cada *assembly* referenciado)
  - se não for encontrado, é lançada excepção
  - outras políticas indicadas em ficheiro de configuração em XML
- Só são controlados *assemblies* com nomes fortes, registados na *global assembly cache* (com gacutil –i *assembly*)

107

Assemblies

Nomes fortes (partilhados)

- Usados para controlo de versões e segurança
- Nome forte consiste da identidade do *assembly* — nome simples textual, nº de versão e informação sobre cultura (se fornecida) – reforçado com uma **chave pública** e uma **assinatura digital**
  - myDll, Version=1.1.0.0, Culture=en, PublicKeyToken=03689116d3a4ae33
- 1º - usar *Strong Name tool* (sn.exe) para gerar ficheiro com par de chaves (pública e privada)
  - `sn –k Teste.key`
- 2º - indicar no código fonte o ficheiro de chaves a usar
  - `[assembly:AssemblyKeyFile("Teste.key")]`
- 3º - compilador coloca automaticamente no manifesto as chaves públicas dos *assemblies* referenciados

108

Reflexão

- Possibilidade de consultar meta-informação em tempo de execução e mesmo criar novos tipos e gerar ("emitir") e executar código MSIL
- Essencial para o .NET *runtime*
- Já visto anteriormente
  - consulta de atributos
  - instanciação de objectos e invocação de métodos com *late binding* (por nome)
- Muitos outros casos de utilização

109

Multithreading *

- Classe **`Thread`**
  - instanciada com referência para método (com assinatura cf. delegado `ThreadWorker`) que faz o trabalho
  - métodos: `Start`, `Sleep`, `Suspend`, `Resume`, `Interrupt`, `Abort`, ...
  - propriedades: `Priority`, ...
- Classe **`AppDomain`**
  - ambiente isolado para a execução dos vários *threads* de uma aplicação
  - é um processo lógico dentro de um processo físico de um *runtime host* – ASP.NET, IE ou *shell*
- Classe **`Monitor`**
  - sincronização entre *threads* usando *locks* e sinais
  - métodos: `Enter`, `Exit`, `TryEnter`, `Wait`, `Pulse`, ...

110

Mais Informação

111

C# no Visual Studio 6

- Instalar o .NET Framework SDK Beta 2
- Alterar a seguinte entrada no registry:
  - chave: HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\Text Editor\Tabs/Language Settings\C/C++\FileExtensions
  - valor anterior: cpp;cxx;c;h;hxx;hpp;inl;tlh;tli;rc;rc2
  - novo valor: cpp;cxx;c;h;hxx;hpp;inl;tlh;tli;rc;rc2;cs
- Copiar ficheiro usertype.dat com *keywords* do C# para mesma pasta em que se encontra msdev.exe
- Criar projecto do tipo "makefile" no Visual C++ 6.0
- Nos "Settings ..." do projecto, preencher a "Build command line " com algo do tipo:
  csc HelloWorld.cs

112

Referências e Recursos

- Inside C#, Tom Archer, Microsoft Press, 2001
- C# Language Specification (http://msdn.microsoft.com/library/default.asp?URL=/library/dotnet/csspec/vclrfcsharpspec_Start.htm)
- A Linguagem de Programação C#, Manuel Costa, Systems Engineer, Internet Business Group, Microsoft (diapositivos)
- Visual Studio .NET beta 2
- .NET Framework SDK beta 2